Beyond the Code

A Style Manual for Structured Java Programming

By Daniel D. Yim

Introduction

Programming with computers is an abstract mental process that anyone is capable of understanding, and with enough experience, the task eventually becomes less of a struggle and more of an intuition. But as we graduate from simply studying the language, the elementary building blocks, such as the syntax, the structures, and the methods that are required fulfill the tasks we need, are no longer relevant to our immediate conscience. By now, they are embedded into our brains out of repeated practice. At this point, we are familiar enough with the language to begin refining what we have learned.

Throughout our education, we are drilled to show the *utility* of the programming language by solving different problems and are judged solely on whether or not the program yields a correct solution at execution. However, we are never held accountable for what lies within the source code and therefore will never be given a proper guideline for structured coding. To address that issue, this manual is written for programmers (particularly Java programmers) of all skill levels to be an advancement towards better programming practices. We will place an emphasis on reinforcing programming habits that will promote writing *clean and structured* code. Through examples we will explore common mistakes and popular programming conventions to ultimately develop a proper coding style.

1. Organize Code with Proper Spacing

The experienced programmer always keeps an eye on his/her horizontal spacing. By ensuring that we arrange our code according to the scope, we promote readability, which is a quality that is very helpful when we are in the process of debugging or when others view our code.

Let us take a look at the following example:

```
1  public class Scopes {
2      public static void main(String[] args) {
3          int a = 10;
4          int b = 20;
5          if(a < 15) {
6              System.out.println("A is less than 15.");
7              b = b + a;
8              if(b > 25) {
9                  System.out.println("B is greater than 25.");
10             }   // end if
11         }
12         else {
13             System.out.println("A is greater than 15.");
14         }   // end if
15     }   // end method main
16 }   // end class Scopes
```

Output:

```
1 A is less than 15.
2 B is greater than 25.
```

Notice how we enter four scopes in the code above: `public class Scopes`, `public static void main(String[] args)`, `if(a < 15)`, and `if(b > 25)`. In lines 2 through 15, we are under the scope of `public class Scopes`, so we indent everything once. In lines 3 through 13, we are under the scope of `public static void main(String[] args)`, so we must indent everything twice. In lines 6 to 10, we are under the scope of `if(a < 15)`, so we must indent three times. And so on. Every time we use a brace (`{` or `}`), we see that we either define a scope level or close a scope level.

This convention of separating our scope levels promote the idea of having a *structure* in our code that we can easily see and use.

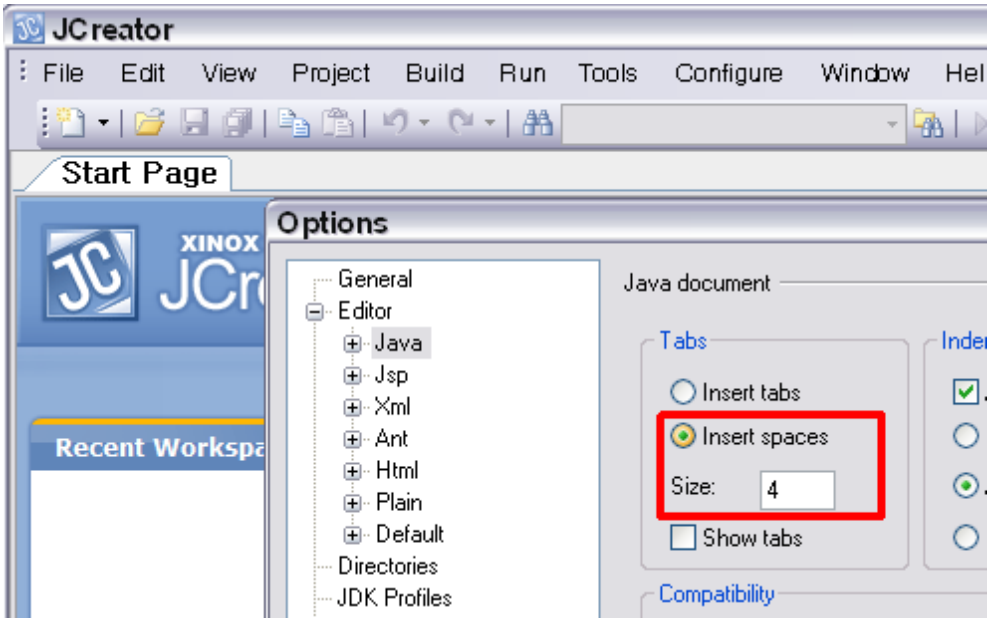
An Issue: Tabs and Editors

In most text editors and integrated development environments (such as JCreator), the editor places a flexible-space `TAB` character where there are supposed to be indents. In most cases, this is both intuitive and acceptable; however, in code writing, this can be very problematic.

As we stated before, the `TAB` character is a flexible space, where the actual character size of that space is defined by the editor. This would be no problem if all editors followed a standard, but unfortunately, most do not and are either 4 or 5 spaces.

Programming Convention: Many programmers eliminate `TAB` character the issue by exclusively using **four spaces** instead.

In JCreator, you can change the tab behavior by going to **Configure** on the menu bar and then selecting **Options**. Then, under **Editor**, select **Java** and choose **Insert spaces**. *When working in new environments, it is important that we check to make sure that we are using spaces and not tabs.*



2. Use Short but Meaningful Variable Names

In Java, variable identifiers are very flexible in terms of naming, but poor programmers deliberately choose to ignore that freedom and continue to use meaningless names. Why should we rely on nondescript identifiers such as x, y, and z to represent complex variables in our programs? As an advanced programmer, one should strive to create **short yet meaningful names** for his/her variables.

Programming Convention: There exist several naming conventions that vary by programming languages; however, for Java, it is considered proper to name variables as such:

1. The first letter should always be lowercase
2. Every "word" of the variable should be capitalized

A good example of a proper variable identifier is `totalTaxRev`. `totalTaxRev` is relatively short and very meaningful. (To the programmer who named it, `totalTaxRev` is instantly recognizable as "total tax revenue".) `totalTaxRev` also follows the two aforementioned naming conventions.

3. Embrace Modularity

Traversing through a lengthy program can be extremely difficult to an unexperienced eye. However, one practice that can alleviate the confusion is **modularization**. By "modularization", we mean to *separate complex or repetitive tasks from cluttering our simple or unique statements.*

If we can foresee that we will be executing calculations or a set of statements over multiple times, we can develop that process into a separate method. In other words, we divide the program into many "modules" that, when combined, will execute exactly the same as before–except that the code will look much more manageable to the programmer. As we see in our typical Java programs, there is always at least one method, the `public static void main(String[] args)`, but we ask: is that enough? In theory, yes. But in practice, never. In general, having several methods for different (or complex) operations is much easier to parse through than having one unusually long `public static void main(String[] args)` method.

4. Document Descriptively and Uniformly

Documentation can range from a simple comment next to a statement to full-fledged literature covering every detail of a program. However, we are concerned with documenting with consistency. sadfa4g3gOn this topic, the reader has the choice of either naturally developing a documentation style or following an industry standard. , but try to create a standard that promotes clean, streamlined code.

For example:

```
1 index++; //increment
```

versus

```
1 index++;    // Increments the array index by 1
```

If you're going to go out of your way to document something, make sure it doesn't disturb the readability of the actual code. We see that without proper spacing, the comment in the first example interferes with the flow of the code. Try to separate the two entities.

Daniel's convention: I generally capitalize all my comments and use action verbs, but if the code that I am describing is particularly complex, I opt for a more descriptive "semi-sentence." That is, I use the pronoun "we" in these sentences to convey that both the author and reader are doing a task together. I also either use a tab and a comment above the statement I am describing or I tab after the semicolon and insert a commentï£¡only if it's short enough. // Increments the counter by 1 x++; or x++; // Increments the counter by 1

Additionally, I discourage letting any line of my code pass the 80 (horizontal) character mark. Note that Java is particularly friendly towards excess whitespace between statements, so examples such as these are valid:

String str = "Hello guys! This is just an example of how I would essentially " + " c̈utä String in half so that I don't violate my rule about " + "exceeding 80 horizontal characters."; Note: Don't forget the spaces after words or the concatenation operator ('+' in Java).

5. Take advantage of syntax coloring and brace matching

You can always click on an open or close curly brace to see where it is matched to. When you get the compiler error "End of file reached while parsing" it is almost always the case that you are missing a curly brace!

6. List your variable/object declarations near the beginning of the method

(Advanced note: this is to follow the Unified Process' UML standard).

7. Choose one of the two brace habits, but be CONSISTENT (choose one and only one way):

if(x > y) code; else code;

or

if(x > y) code; else code;

Daniel's convention: I use a mix of the two examples above: if (x < y) code; else code;

8. Know that there is always a resource online

Although most problems you will encounter will be specific to your program, try to generalize it in a phase and then Google itï£¡you will be amazed at what a few keywords can do for you and your issues. Also note that Java has an API (application programming interface) online that documents what EVERY Java function does.

9. "When it doubt, print it out!"

If your program is being unpredictable or if you're getting a runtime error, try adding simple and informative System.out.println() statements all throughout your program. This will help you see the values that the computer has stored, which may help you fix your code.

10. Readable code is happy code

Don’t try to be clever and clutter your programs with quick fixes here and there. If the situation calls for a proper "remodeling" of the code, then do so by all means. I stress the readability of code because it, first and foremost, helps you as you begin debugging your code and also because a clean, easier to read program is more inviting for outside help.

Conclusion

Thoroughly start these all of these good habits on one program and then just start every subsequent program with that template. Soon enough, you’ll get used to the extra work and will appreciate how much friendlier your code looks!