Java Code Style

Table of Contents

Read Effective Java by Josh Bloch

This is required reading for all Java programmers at Proofpoint. And yes there will be a quiz.

Readability

- Readability is the one thing we optimize our code for
- Your code should be obvious and straightforward
- Extra care should be taken to assure that code containing performance optimization are even more readable than normal, due to the complexity of such code
- Tests should also be as readable (if not more) than the code they are testing

Favor Immutable objects

Immutable objects are simply objects whose state (the object's data) cannot change after construction. Examples of immutable objects from the JDK include String and Integer.

Immutable objects greatly simplify your program, since they:

- are simple to construct, test, and use
- are automatically thread-safe and have no synchronization issues
- do not need a copy constructor
- do not need to be copied defensively when used as a field
- make good Map keys and Set elements (these objects must not change state while in the collection)

When implementing:

- mark all fields as private final
- defensively copy any mutable constructor arguments (like maps and collections)
- if an attributes is mutable, return the caller a copy

Control flow structures

- Use **for-each** instead of traditional **for** or **while** if possible.

```
for (String value : list) {
   ...
}
```

instead of:

```
Iterator<String> values = list.iterator();
while (values.hasNext()) {
   String value = values.next();
   ...
}
```

or

```
for (Iterator<String> values = list.iterator(); values.hasNext(); ) {
   String value = values.next();
   ...
}
```

- For maps, iterate over entrySet instead of iterating over keySet and getting the associated value in each loop

```
for (Map.Entry<Foo, Bar> entry : map.entrySet()) {
   Foo key = entry.getKey();
   Bar value = entry.getValue();
   ...
}
```

instead of:

```
for (Foo key : map.keySet()) {
   Bar value = map.get(key);
   ...
}
```

Raw types

Use parameterized types:

```
private final Map<String, Foo> map;
```

instead of raw types:

```
private final Map map;
```

All sorts of strange things start happening in the Java language when raw types are encountered. See the Java Puzzlers book for many examples.

Final variables

Final should only be applied to class member variables. Specifically, final should not be applied parameters or local variables unless required because the reference is use in an anonymous inner class.

Avoid non-static inner classes

Instances of a non-static inner class have an implicit reference to the outer class and this is very difficult to see. Inner classes also have implicit access to the private fields in the outer class and can cause subtle bugs.

Cloneable

Java's clone is truly broken, and should not be used. Josh Bloch on Design - Copy Constructor versus Cloning

> *Object's clone method is very tricky. It's based on field copies, and it's "extra-linguistic." It creates an object without calling a constructor. There are no guarantees that it preserves the invariants*

> *established by the constructors. There have been lots of bugs over the years, both in and outside Sun, stemming from the fact that if you just call super.clone repeatedly up the chain until you have cloned an object, you have a shallow copy of the object. The clone generally shares state with the object being cloned. If that state is mutable, you don't have two independent objects. If you modify one, the other changes as well. And all of a sudden, you get random behavior.*

If you need to copy a mutable object, add a copy constructor to the class. For more information see Effective Java: Item 11.

Logging

Logger Declaration

The correct declaration of a logger is private static final named "log". For example,

```
public class MyService
{
    private static final Logger log = Logger.get(MyService.class);
}
```

The lower case log name is a violation of the normal rules for a constant, which is acceptable here.

Levels

- DEBUG: for things an application developer would need to know
- INFO: for things an administrator would need to know
- WARNING: for things indicating a non-critical or transient problem
- ERROR: for critical errors that require immediate attention

Naming

Packages

- Package names are lowercase (specifically not camelCased).
- Only alphanumeric characters
- Should start with the name of the maven module in which they are defined. E.g., com.proofpoint.platform.logging for the "logging" module. This helps avoid conflicts when multiple modules are imported into a project.

Unit tests

- The test for Foo is named TestFoo.java

Javadocs

- No empty javadocs. It's better to not have a javadoc section than to have an empty one.
- Avoid auto-generated javadocs. If they can be auto generated, they don't contain any unique information.
- No @author tags. That information is usually inaccurate and is available via the source control system, anyway.

Formatting

Indentation, spacing, braces

- 4 spaces, no tabs
- IDEA code style file (Instructions on how to do this: http://musingsofaprogrammingaddict.blogspot.com/2010/03/import-code-style-settings-into.html)
- Eclipse formatter file and code cleanup file . See comment below on where to put these.
- Always use braces for loops and conditional code blocks, even if they contain only one line.

Example:

```
public class Foo
    extends Bar, Baz
    implements SomeInterface
{
    public void foo(boolean a, int x, int y, int z)
        throws AnException
    {
        do {
            try {
                if (x > 0) {
                    int someVariable = a ?  x : y;
                }
                else if (x < 0) {
                    int someVariable = (y + z);
                    someVariable = x = x + y;
                }
                else {
                    for (int i = 0; i < 5; i++) {
                        doSomething(i);
                    }
                }

                switch (a) {
                    case 0:
                        doCase0();
                        break;
                    default:
                        doDefault();
                }
            }
            catch (Exception e) {
                processException(e.getMessage(), x + y, z, a);
            }
            finally {
                processFinally();
            }
        }
        while (true);

        if (2 < 3) {
            return;
        }

        if (3 < 4) {
            return;
        }

        do {
            x++
        }
        while (x < 10000);

        while (x < 50000) {
            x++;
        }

        for (int i = 0; i < 5; i++) {
            System.out.println(i);
        }
    }

    private class InnerClass
        implements I1, I2
    {
        public void bar()
```

```
            throws E1, E2
        {
        }
    }
}
```

Imports

- Fully qualified. I.e., no wildcard imports like java.util.*
- Should be sorted alphabetically with java and javax packages listed separately, followed by static imports. E.g.

```
import com.proofpoint.platform.configuration.Config;
import com.proofpoint.platform.logging.Logger;

import java.util.List;
import javax.ws.rs.GET;

import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertNotNull;
```

IDE Warnings

- Code should show a "green light" in IDEA (or equivalent in other IDEs)
- TBD - standardize warning levels (e.g., code inspections in IDEA)