Collateral Problems

- *Overengineering*
- *Underengineering*
- *Wasting time on fixing the consequences, not the problem*
- *Unreliable and unscalable code*
- *Dirty code*

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

Martin Fowler

Pros

1. *Maintainable Codebase*
2. *Easier Troubleshooting*
3. *Faster Onboarding*

Cons

1. *Longer in short term*
2. *Requires energy*
3. *Required knowledge*

Structure

A source file consists of, in order:

The package statement is not line-wrapped. The column limit does not apply to package statements.

Wildcard imports, static or otherwise, are not used.

Import statements are not line-wrapped. The column limit does not apply to import statements.

Imports are ordered as follows:

All static imports in a single block.

All non-static imports in a single block.

If there are both static and non-static imports, a single blank line separates the two blocks. There are no other blank lines between import statements.

Within each block the imported names appear in ASCII sort order. (Note: this is not the same as the import statements being in ASCII sort order, since '.' sorts before ';'.)

Static import is not used for static nested classes. They are imported with normal imports.

Exactly one blank line separates each section that is present.

A source file consists of, **in order**:

3.1 Package statement

The package statement is **not line-wrapped**. The column limit does not apply to package statements.

3.2 Import statements

3.2.1 No wildcard imports

Wildcard imports, static or otherwise, **are not used**.

3.2.2 No line-wrapping

Import statements are **not line-wrapped**. The column limit does not apply to import statements.

3.2.3 Ordering and spacing

Imports are ordered as follows:

1. All static imports in a single block.
2. All non-static imports in a single block.

If there are both static and non-static imports, a single blank line separates the two blocks. There are no other blank lines between import statements.

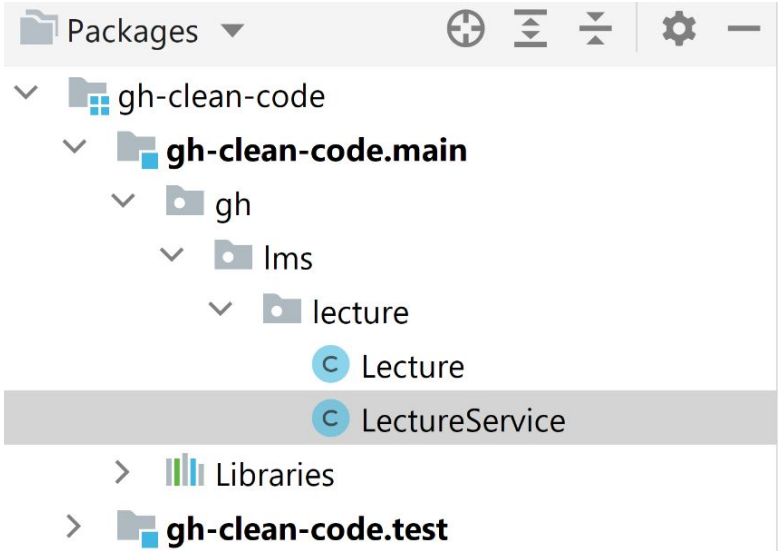
Within each block the imported names appear in ASCII sort order. (**Note:** this is not the same as the import *statements* being in ASCII sort order, since '.' sorts before ';'.)

3.2.4 No static import for classes

Static import is not used for static nested classes. They are imported with normal imports.

Exactly one blank line separates each section that is present.

Table of Contents



Content

```
1   package gh.lms.lecture;
2
3   public class LectureService {
4
5       public void createLecture(String name) {}
8
9       public void deleteLecture() {}
12
13      public Lecture getLecture(String name) {
14          return null;
15      }
16  }
17
```

Naming

a Use ONLY ASCII letters and digits, and, in a smallNumber of cases noted below, under_scores. Thus each valid ***a*** name ismatched by the regular EXPRESSION \w+ .

In google.style, SPECIAL prefixes or suffixes are not Used.

Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores. Thus each valid **identifier** name is matched by the regular expression \w+ .

In Google Style, special prefixes or suffixes are not used.

Naming Convention

```
package <packagename>;

public class <ClassName> {

    public static final String <CONSTANT_NAME> = "";

    private String <fieldName>;

    public String <methodName>(String <parameterName>) {}
}
```

noun

verb

Shortest Meaningful Names

```
int[] mm
int[] intArrayOfInputNumbers
int[] inputNumbers
```

Methods

Keep them Small

ideal size is 20 lines

Make Sure They Just Do One Thing

~~*getAndMultiplyAndPrint*~~*()*

~~*get*~~*()*

parseInput(), multiplyNumbers(), printResult()

Encapsulate Conditionals in Functions

```
if (areAsleep(students) && lecturer.isPresent() && lecture.isActive())
```

```
boolean boringLecturer = areAsleep(students) && lecturer.isPresent() && lecture.isActive()
if (boringLecturer)
```

Decompose Methods

```
public Lecture getOrCreateLecture(String name) throws LectureValidationException {
    if (!isValid(name)) {
        throw new LectureValidationException(name);
    }

    if (isLectureExists(name)) {
        return getLecture(name);
    } else {
        return createLecture(name);
    }
}
```

Leave the campground cleaner than you found it.

When Clean Code doesn't work

Readable

```
static int fibonacci(int n) {
    if (n ⩽ 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Optimized

```
/*
* Fast matrix method. Easy to describe, but has a constant factor slowdown compared to
doubling method.
* [1 1]^n   [F(n+1) F(n)  ]
* [1 0]   = [F(n)   F(n-1)].
*/
private static BigInteger fastFibonacciMatrix(int n) {
    BigInteger[] matrix = {BigInteger.ONE, BigInteger.ONE, BigInteger.ONE,
BigInteger.ZERO};
    return matrixPow(matrix, n)[1];
}

// Computes the power of a matrix. The matrix is packed in row-major order.
private static BigInteger[] matrixPow(BigInteger[] matrix, int n) {
    if (n < 0)
        throw new IllegalArgumentException();
    BigInteger[] result = {ONE, ZERO, ZERO, ONE};
    while (n ≠ 0) {  // Exponentiation by squaring
        if (n % 2 ≠ 0)
            result = matrixMultiply(result, matrix);
        n ⊨ 2;
        matrix = matrixMultiply(matrix, matrix);
    }
    return result;
}

// Multiplies two matrices.
private static BigInteger[] matrixMultiply(BigInteger[] x, BigInteger[] y) {
    return new BigInteger[] {
        x[0].multiply(y[0]).add(x[1].multiply(y[2])),
        x[0].multiply(y[1]).add(x[1].multiply(y[3])),
        x[2].multiply(y[0]).add(x[3].multiply(y[2])),
        x[2].multiply(y[1]).add(x[3].multiply(y[3]))
    };
}
```

Tools

IDEA autoformatting (Ctrl + Alt + L)

SonarLint

Resources

Clean Code

Java Google Style Guide

Java Naming Conventions

IDEA Keyboard Shortcuts