Προγραμματισμός II
Java Style

Διομήδης Σπινέλλης
Τμήμα Διοικητικής Επιστήμης και Τεχνολογίας
Οικονομικό Πανεπιστήμιο Αθηνών

dds@aueb.gr
http://www.dmst.aueb.gr/dds
@CoolSWEng

2024-11-20

Ποιότητα κώδικα

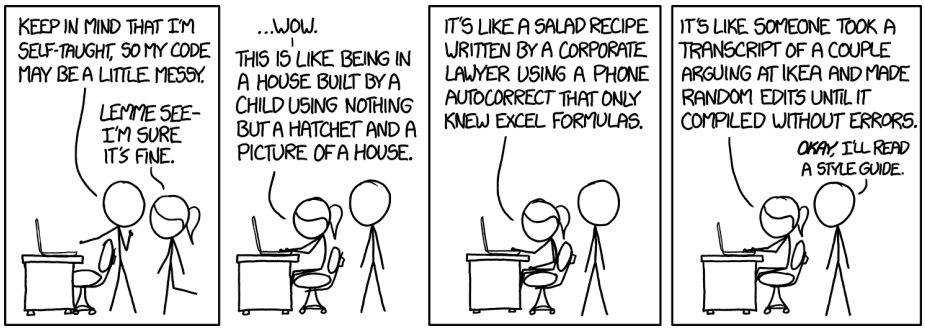

Figure 1: Ποιότητα κώδικα

Overview

- Coding style
- Writing testable code
- Best practices

Recommended reading

- Google's Java Style Guide.
- Oracle's Code Conventions for the Java Programming Language
- Twitter's Java Style Guide (covered here)
- Effective Java
- Java Practices
- Java Concurrency in Practice
- Code Complete 2

Coding style

Formatting

Use line breaks wisely

There are generally two reasons to insert a line break:

1. Your statement exceeds the column limit.
2. You want to logically separate a thought.

Writing code is like telling a story. Written language constructs like chapters, paragraphs, and punctuation (e.g. semicolons, commas, periods, hyphens) convey thought hierarchy and separation. We have similar constructs in programming languages; you should use them to your advantage to effectively tell the story to those reading the code.

Indent style

Use the “one true brace style” (1TBS). Indent size is 4 columns.

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

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

// Also not like this.
if (x < 0) negative(x);
```

Continuation style

Continuation indent is 8 columns.

```
// Bad.
//   - Line breaks are arbitrary.
//  - Scanning the code makes it difficult to piece the message together.
throw new IllegalStateException("Failed to process request" + request.getId()
   + " for user " + user.getId() + " query: '" + query.getText()
    + "'");

// Good.
```

```
//   - Each component of the message is separate and self-
contained.
//  - Adding or removing a component of the message requires minimal reformatting.
throw new IllegalStateException("Failed to process"
    + " request " + request.getId()
    + " for user " + user.getId()
    + " query: '" + query.getText() + "'");
```

Don’t break up a statement unnecessarily.

```
// Bad.
final String value =
    otherValue;

// Good.
final String value = otherValue;
```

Method declaration continuations.

```
// Sub-optimal since line breaks are arbitrary and only filling lines.
String downloadAnInternet(Internet internet, Tubes tubes,
    Blogosphere blogs, Amount<Long, Data> bandwidth) {
  tubes.download(internet);
  ...
}

// Acceptable.
String downloadAnInternet(Internet internet, Tubes tubes, Blogosphere blogs,
    Amount<Long, Data> bandwidth) {
  tubes.download(internet);
  ...
}

// Nicer, as the extra newline gives visual separation to the method body.
String downloadAnInternet(Internet internet, Tubes tubes, Blogosphere blogs,
    Amount<Long, Data> bandwidth) {

  tubes.download(internet);
  ...
}

// Also acceptable, but may be awkward depending on the column depth of the opening pare
public String downloadAnInternet(Internet internet,
                                 Tubes tubes,
                                 Blogosphere blogs,
```

```
                                Amount<Long, Data> bandwidth) {
  tubes.download(internet);
  ...
}

// Preferred for easy scanning and extra column space.
public String downloadAnInternet(
    Internet internet,
    Tubes tubes,
    Blogosphere blogs,
    Amount<Long, Data> bandwidth) {

  tubes.download(internet);
  ...
}
```

Chained method calls

```
// Bad.
//   - Line breaks are based on line length, not logic.
Iterable<Module> modules = ImmutableList.<Module>builder().add(new LifecycleModule()
   .add(new AppLauncherModule()).addAll(application.getModules()).build();

// Better.
//   - Calls are logically separated.
//  - However, the trailing period logically splits a statement across two lines.
Iterable<Module> modules = ImmutableList.<Module>builder().
    add(new LifecycleModule()).
    add(new AppLauncherModule()).
    addAll(application.getModules()).
    build();

// Good.
//   - Method calls are isolated to a line.
//  - The proper location for a new method call is unambiguous.
Iterable<Module> modules = ImmutableList.<Module>builder()
    .add(new LifecycleModule())
    .add(new AppLauncherModule())
    .addAll(application.getModules())
    .build();
```

No tabs

An oldie, but goodie. We've found tab characters to cause more harm than good.

70 column limit

You should follow the convention set by the body of code you are working with. We tend to use 70 columns for a balance between fewer continuation lines but still easily fitting two editor tabs side-by-side on a reasonably-high resolution display.

CamelCase for types, camelCase for variables, UPPER_SNAKE for constants

No trailing whitespace

Trailing whitespace characters, while logically benign, add nothing to the program. However, they do serve to frustrate developers when using keyboard shortcuts to navigate code.

Field, class, and method declarations

Modifier order

Follow the Java Language Specification for modifier ordering (sections 8.1.1, 8.3.1 and 8.4.3).

```
// Bad.
final volatile private String value;

// Good.
private final volatile String value;
```

Variable naming

Extremely short variable names should be reserved for instances like loop indices.

```
// Bad.
//  - Field names give little insight into what fields are used for.
class User {
  private final int a;
  private final String m;

  ...
}

// Good.
class User {
  private final int ageInYears;
```

```
  private final String maidenName;

  ...
}
```

Include units in variable names

```
// Bad.
long pollInterval;
int fileSize;

// Good.
long pollIntervalMs;
int fileSizeGb.

// Better.
//   - Unit is built in to the type.
//  - The field is easily adaptable between units, readability is high.
Amount<Long, Time> pollInterval;
Amount<Integer, Data> fileSize;
```

Don’t embed metadata in variable names

- A variable name should describe the variable’s purpose.
- Adding extra information like scope and type is generally a sign of a bad variable name.
- Avoid embedding the field type in the field name.

  // Bad. Map<Integer, User> idToUserMap; String valueString;

  // Good. Map<Integer, User> usersById; String value;

Don’t embed scope in variable names

Avoid embedding scope information in a variable. Hierarchy-based naming suggests that a class is too complex and should be broken apart.

```
// Bad.
String _value;
String mValue;

// Good.
String value;
```

Space pad operators and equals.

```
// Bad.
//   - This offers poor visual separation of operations.
int foo=a+b+1;

// Good.
int foo = a + b + 1;
```

Be explicit about operator precedence

Don't make your reader open the spec to confirm, if you expect a specific operation ordering, make it obvious with parenthesis.

```
// Bad.
return a << 8 * n + 1 | 0xFF;

// Good.
return (a << (8 * n) + 1) | 0xFF;
```

It's even good to be *really* obvious.

```
if ((values != null) && (10 > values.size())) {
  ...
}
```

Make structure match intent

```
// Bad
if (booleanExpression) {
    return TRUE;
} else {
    return FALSE;
}

// Good
return booleanExpression;
```

Documentation

The more visible a piece of code is (and by extension - the farther away consumers might be), the more documentation is needed.

"I'm writing a report about…"

Your elementary school teacher was right - you should never start a statement this way. Likewise, you shouldn't write documentation this way.

```
// Bad.
/**
 * This is a class that implements a cache.  It does caching for you.
 */
class Cache {
  ...
}

// Good.
/**
 * A volatile storage for objects based on a key, which may be invalidated and discarded
 */
class Cache {
  ...
}
```

Documenting a class

Documentation for a class may range from a single sentence to paragraphs with code examples. Documentation should serve to disambiguate any conceptual blanks in the API, and make it easier to quickly and *correctly* use your API. A thorough class doc usually has a one sentence summary and, if necessary, a more detailed explanation.

```
/**
 * An RPC equivalent of a unix pipe tee.  Any RPC sent to the tee input is guaranteed to h
 * been sent to both tee outputs before the call returns.
 *
 * @param <T> The type of the tee'd service.
 */
public class RpcTee<T> {
  ...
}
```

Documenting a method

A method doc should tell what the method *does*. Depending on the argument types, it may also be important to document input format.

```
// Bad.
//  - The doc tells nothing that the method declaration didn't.
//  - This is the 'filler doc'.  It would pass style checks, but doesn't help anybody.
/**
 * Splits a string.
 *
 * @param s A string.
```

```
 * @return A list of strings.
 */
List<String> split(String s);

// Better.
//   - We know what the method splits on.
//   - Still some undefined behavior.
/**
 * Splits a string on whitespace.
 *
 * @param s The string to split.  An {@code null} string is treated as an empty string.
 * @return A list of the whitespace-delimited parts of the input.
 */
List<String> split(String s);

// Great.
//   - Covers yet another edge case.
/**
 * Splits a string on whitespace.  Repeated whitespace characters are collapsed.
 *
 * @param s The string to split.  An {@code null} string is treated as an empty string.
 * @return A list of the whitespace-delimited parts of the input.
 */
List<String> split(String s);
```

Be professional

We've all encountered frustration when dealing with other libraries, but ranting about it doesn't do you any favors. Suppress the expletives and get to the point.

```
// Bad.
// I hate xml/soap so much, why can't it do this for me!?
try {
  userId = Integer.parseInt(xml.getField("id"));
} catch (NumberFormatException e) {
  ...
}

// Good.
// TODO(Jim): Tuck field validation away in a library.
try {
  userId = Integer.parseInt(xml.getField("id"));
} catch (NumberFormatException e) {
  ...
}
```

Don’t document overriding methods (usually)

```
interface Database {
  /**
   * Gets the installed version of the database.
   *
   * @return The database version identifier.
   */
  String getVersion();
}

// Bad.
//   - Overriding method doc doesn't add anything.
class PostgresDatabase implements Database {
  /**
   * Gets the installed version of the database.
   *
   * @return The database version identifier.
   */
  @Override
  public String getVersion() {
    ...
  }
}

// Good.
class PostgresDatabase implements Database {
  @Override
  public int getVersion();
}

// Great.
//  - The doc explains how it differs from or adds to the interface doc.
class TwitterDatabase implements Database {
  /**
   * Semantic version number.
   *
   * @return The database version in semver format.
   */
  @Override
  public String getVersion() {
    ...
  }
}
```

Use javadoc features

No author tags

Code can change hands numerous times in its lifetime, and quite often the original author of a source file is irrelevant after several iterations. We find it's better to trust commit history and `OWNERS` files to determine ownership of a body of code.

Imports

Import ordering

Imports are grouped by top-level package, with blank lines separating groups. Static imports are grouped in the same way, in a section below traditional imports.

```
import java.*
import javax.*

import scala.*

import com.*

import net.*

import org.*

import com.twitter.*

import static *
```

No wildcard imports

Wildcard imports make the source of an imported class less clear. They also tend to hide a high class fan-out. *See also texas imports*

```
// Bad.
//   - Where did Foo come from?
import com.twitter.baz.foo.*;
import com.twitter.*;

interface Bar extends Foo {
  ...
}

// Good.
```

```
import com.twitter.baz.foo.BazFoo;
import com.twitter.Foo;

interface Bar extends Foo {
  ...
}
```

Use annotations wisely

@Nullable

By default - disallow `null`. When a variable, parameter, or method return value may be `null`, be explicit about it by marking @Nullable. This is advisable even for fields/methods with private visibility.

```
class Database {
  @Nullable private Connection connection;

  @Nullable
  Connection getConnection() {
    return connection;
  }

  void setConnection(@Nullable Connection connection) {
    this.connection = connection;
  }
}
```

@VisibleForTesting

Sometimes members and functions may be required for good test coverage. Make these package-private and tag with @VisibleForTesting to indicate the purpose for visibility.

Constants are a great example of things that are frequently exposed in this way.

```
// Bad.
//  - Any adjustments to field names need to be duplicated in the test.
class ConfigReader {
  private static final String USER_FIELD = "user";

  Config parseConfig(String configData) {
    ...
  }
}
public class ConfigReaderTest {
  @Test
```

```
  public void testParseConfig() {
    ...
   assertEquals(expectedConfig, reader.parseConfig("{user: bob}"));
  }
}

// Good.
//   - The test borrows directly from the same constant.
class ConfigReader {
 @VisibleForTesting static final String USER_FIELD = "user";

  Config parseConfig(String configData) {
    ...
  }
}
public class ConfigReaderTest {
  @Test
  public void testParseConfig() {
    ...
    assertEquals(expectedConfig,
     reader.parseConfig(String.format("{%s: bob}", ConfigReader.USER_FIELD)));
  }
}
```

Use interfaces

- Interfaces decouple functionality from implementation, allowing you to use multiple implementations without changing consumers.
- Interfaces are a great way to isolate packages - provide a set of interfaces, and keep your implementations package private.

Small interfaces

Many small interfaces can seem heavyweight, since you end up with a large number of source files. Consider the pattern below as an alternative.

```
interface FileFetcher {
  File getFile(String name);

 // All the benefits of an interface, with little source management overhead.
 // This is particularly useful when you only expect one implementation of an interface
  static class HdfsFileFetcher implements FileFetcher {
    @Override File getFile(String name) {
      ...
    }
  }
```

```
}
```

Leverage existing interfaces

Sometimes an existing interface allows your class to easily ‘plug in’ to other related classes. This leads to highly cohesive code.

```
// An unfortunate lack of consideration.  Anyone who wants to interact with Blobs will n
// write specific glue code.
class Blobs {
  byte[] nextBlob() {
    ...
  }
}

// Much better.  Now the caller can easily adapt this to standard collections, or do mor
// complex things like filtering.
class Blobs implements Iterable<byte[]> {
  @Override
  Iterator<byte[]> iterator() {
    ...
  }
}
```

Warning - don’t bend the definition of an existing interface to make this work. If the interface doesn’t conceptually apply cleanly, it’s best to avoid this.

Writing testable code

Writing unit tests doesn’t have to be hard. You can make it easy for yourself if you keep testability in mind while designing your classes and interfaces.

Let your callers construct support objects

```
// Bad.
//  - A unit test needs to manage a temporary file on disk to test this class.
class ConfigReader {
  private final InputStream configStream;
  ConfigReader(String fileName) throws IOException {
    this.configStream = new FileInputStream(fileName);
  }
}

// Good.
//  - Testing this class is as easy as using ByteArrayInputStream with a String.
class ConfigReader {
```

```
  private final InputStream configStream;
  ConfigReader(InputStream configStream){
    this.configStream = checkNotNull(configStream);
  }
}
```

Testing antipatterns

Time-dependence

Code that captures real wall time can be difficult to test repeatably, especially when time deltas are meaningful. Therefore, try to avoid `new Date()`, `System.currentTimeMillis()`, and `System.nanoTime()`. A suitable replacement for these is Clock; using Clock.SYSTEM_CLOCK when running normally, and FakeClock in tests.

The hidden stress test

Avoid writing unit tests that attempt to verify a certain amount of performance. This type of testing should be handled separately, and run in a more controlled environment than unit tests typically are.

Thread.sleep()

Sleeping is rarely warranted, especially in test code. Sleeping is expressing an expectation that something else is happening while the executing thread is suspended. This quickly leads to brittleness; for example if the background thread was not scheduled while you were sleeping.

Sleeping in tests is also bad because it sets a firm lower bound on how fast tests can execute. No matter how fast the machine is, a test that sleeps for one second can never execute in less than one second. Over time, this leads to very long test execution cycles.

Avoid randomness in tests

Using random values may seem like a good idea in a test, as it allows you to cover more test cases with less code. The problem is that you lose control over which test cases you're covering. When you do encounter a test failure, it may be difficult to reproduce. Pseudorandom input with a fixed seed is slightly better, but in practice rarely improves test coverage. In general it's better to use fixed input data that exercises known edge cases.

Best practices

Defensive programming

Preconditions

Preconditions checks are a good practice, since they serve as a well-defined barrier against bad input from callers. As a convention, object parameters to public constructors and methods should always be checked against null, unless null is explicitly allowed.

```
// Bad.
//  - If the file or callback are null, the problem isn't noticed until much later.
class AsyncFileReader {
  void readLater(File file, Closure<String> callback) {
    scheduledExecutor.schedule(new Runnable() {
      @Override public void run() {
        callback.execute(readSync(file));
      }
    }, 1L, TimeUnit.HOURS);
  }
}

// Good.
class AsyncFileReader {
  void readLater(File file, Closure<String> callback) {
    checkNotNull(file);
   checkArgument(file.exists() && file.canRead(), "File must exist and be readable.");
    checkNotNull(callback);

    scheduledExecutor.schedule(new Runnable() {
      @Override public void run() {
        callback.execute(readSync(file));
      }
    }, 1L, TimeUnit.HOURS);
  }
}
```

Minimize visibility

In a class API, you should support access to any methods and fields that you make accessible. Therefore, only expose what you intend the caller to use. This can be imperative when writing thread-safe code.

```
public class Parser {
  // Bad.
 //  - Callers can directly access and mutate, possibly breaking internal assumptions.
```

```
  public Map<String, String> rawFields;

  // Bad.
 //  - This is probably intended to be an internal utility function.
  public String readConfigLine() {
    ..
  }
}

// Good.
//   - rawFields and the utility function are hidden
//  - The class is package-private, indicating that it should only be accessed indirect
class Parser {
  private final Map<String, String> rawFields;

  private String readConfigLine() {
    ..
  }
}
```

Favor immutability

Mutable objects carry a burden - you need to make sure that those who are *able* to mutate it are not violating expectations of other users of the object, and that it's even safe for them to modify.

```
// Bad.
//  - Anyone with a reference to User can modify the user's birthday.
//  - Calling getAttributes() gives mutable access to the underlying map.
public class User {
  public Date birthday;
 private final Map<String, String> attributes = Maps.newHashMap();

  ...

  public Map<String, String> getAttributes() {
    return attributes;
  }
}

// Good.
public class User {
  private final Date birthday;
 private final Map<String, String> attributes = Maps.newHashMap();

  ...
```

```
  public Map<String, String> getAttributes() {
    return ImmutableMap.copyOf(attributes);
  }

 // If you realize the users don't need the full map, you can avoid the map copy
  // by providing access to individual members.
  @Nullable
  public String getAttribute(String attributeName) {
    return attributes.get(attributeName);
  }
}
```

Be wary of null

Use `@Nullable` where prudent, but favor Optional over `@Nullable`. `Optional` provides better semantics around absence of a value.

Clean up with finally

```
FileInputStream in = null;
try {
  ...
} catch (IOException e) {
  ...
} finally {
  Closeables.closeQuietly(in);
}
```

Use finally to avoid leaks

Even if there are no checked exceptions, there are still cases where you should use try/finally to guarantee resource symmetry.

```
// Bad.
//   - Mutex is never unlocked.
mutex.lock();
throw new NullPointerException();
mutex.unlock();

// Good.
mutex.lock();
try {
  throw new NullPointerException();
} finally {
```

```
  mutex.unlock();
}

// Bad.
//   - Connection is not closed if sendMessage throws.
if (receivedBadMessage) {
  conn.sendMessage("Bad request.");
  conn.close();
}

// Good.
if (receivedBadMessage) {
  try {
    conn.sendMessage("Bad request.");
  } finally {
    conn.close();
  }
}
```

Clean code

Disambiguate

Favor readability - if there's an ambiguous and unambiguous route, always favor unambiguous.

```
// Bad.
//  - Depending on the font, it may be difficult to discern 1001 from 100l.
long count = 100l + n;

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

Remove dead code

Delete unused code (imports, fields, parameters, methods, classes). They will only rot.

Use general types

When declaring fields and methods, it's better to use general types whenever possible. This avoids implementation detail leak via your API, and allows you to change the types used internally without affecting users or peripheral code.

```
// Bad.
//  - Implementations of Database must match the ArrayList return type.
```

```
//  - Changing return type to Set<User> or List<User> could break implementations and u
interface Database {
  ArrayList<User> fetchUsers(String query);
}

// Good.
//  - Iterable defines the minimal functionality required of the return.
interface Database {
  Iterable<User> fetchUsers(String query);
}
```

Always use type parameters

Java 5 introduced support for generics. This added type parameters to collection types, and allowed users to implement their own type-parameterized classes. Backwards compatibility and type erasure mean that type parameters are optional, however depending on usage they do result in compiler warnings.

We conventionally include type parameters on every declaration where the type is parameterized. Even if the type is unknown, it's preferable to include a wildcard or wide type.

Stay out of Texas

Try to keep your classes bite-sized and with clearly-defined responsibilities. This can be *really* hard as a program evolves.

- texas imports
- texas constructors: Can the class be cleanly broken apart? If not, consider builder pattern.
- texas methods

We could do some science and come up with a statistics-driven threshold for each of these, but it probably wouldn't be very useful. This is usually just a gut instinct, and these are traits of classes that are too large or complex and should be broken up.

Avoid typecasting

Typecasting is a sign of poor class design, and can often be avoided. An obvious exception here is overriding equals.

Use final fields

See also favor immutability

Final fields are useful because they declare that a field may not be reassigned. When it comes to checking for thread-safety, a final field is one less thing that needs to be checked.

Avoid mutable static state

Mutable static state is rarely necessary, and causes loads of problems when present. A very simple case that mutable static state complicates is unit testing. Since unit tests runs are typically in a single VM, static state will persist through all test cases. In general, mutable static state is a sign of poor class design.

Exceptions

Catch narrow exceptions

Sometimes when using try/catch blocks, it may be tempting to just `catch Exception`, `Error`, or `Throwable`. This is a bad idea, for example, `catch Exception` would capture `NullPointerException`, and `catch Throwable` would capture `OutOfMemoryError`.

```
// Bad.
//  - If a RuntimeException happens, the program continues rather than aborting.
try {
  storage.insertUser(user);
} catch (Exception e) {
  LOG.error("Failed to insert user.");
}

try {
  storage.insertUser(user);
} catch (StorageException e) {
  LOG.error("Failed to insert user.");
}
```

Don’t swallow exceptions

An empty `catch` block is usually a bad idea, as you have no signal of a problem. Coupled with narrow exception violations, it’s a recipe for disaster.

Throw specific exceptions

Let your API users obey catch narrow exceptions, don’t throw Exception. You should also make an effort to hide implementation details from your callers when it comes to exceptions.

```
// Bad.
//  - Caller is forced to catch Exception, trapping many unnecessary types of issues.
interface DataStore {
  String fetchValue(String key) throws Exception;
}

// Better.
//  - The interface leaks details about one specific implementation.
interface DataStore {
 String fetchValue(String key) throws SQLException, UnknownHostException;
}

// Good.
//  - A custom exception type insulates the user from the implementation.
//  - Different implementations aren't forced to abuse irrelevant exception types.
interface DataStore {
  String fetchValue(String key) throws StorageException;

  static class StorageException extends Exception {
    ...
  }
}
```

Use newer/better libraries

StringBuilder over StringBuffer

StringBuffer is thread-safe, which is rarely needed.

List over Vector

`Vector` is synchronized, which is often unneeded. When synchronization is desirable, a synchronized list can usually serve as a drop-in replacement for `Vector`.

equals() and hashCode()

- If you override one, you must implement both.
- In brief, the equals/hashCode contract specifies:
  - Whenever invoked on the same object, hashCode must return the same integer, provided no information used in equals comparisons is modified
  - For objects where equals(Object) is true, calling hashCode on each must produce the same result.
  - Unequal objects should but need not return different hash codes.

- `Objects.equal()` and `Objects.hashCode()`, which handle `null` values, make it very easy to follow these contracts.

Premature optimization is the root of all evil.

Donald Knuth is a smart guy, and he had a few things to say on the topic.

Unless you have strong evidence that an optimization is necessary, it's usually best to implement the un-optimized version first (possibly leaving notes about where optimizations could be made).

So before you spend a week writing your memory-mapped compressed huffman-encoded hashmap, use the stock stuff first and *measure*.

Special comments

- Use TODO for work that still needs to be done.
- Use XXX in a comment to flag something that is bogus but works.
- Use FIXME to flag something that is bogus and broken.

Leave TODOs early and often

A TODO isn't a bad thing - it's signaling a future developer (possibly yourself) that a consideration was made, but omitted for various reasons. It can also serve as a useful signal when debugging.

Leave no TODO unassigned

TODOs should have owners, otherwise they are unlikely to ever be resolved.

```
// Bad.
//   - TODO is unassigned.
// TODO: Implement request backoff.

// Good.
// TODO(George Washington): Implement request backoff.
```

Adopt TODOs

You should adopt an orphan if the owner has left the company/project, or if you make modifications to the code directly related to the TODO topic.

Obey the Law of Demeter (LoD)

The Law of Demeter is most obviously violated by breaking the one dot rule, but there are other code structures that lead to violations of the spirit of the law.

Follow the SOLID principles

- S: Single Responsibility - Each class should have one reason to change.
- O: Open/Closed - Extend classes, rather than modifying them
- L: Liskov Substitution - Subtypes should be replaceable by their base types.
- I: Interface Segregation - Create specific rather than wide interfaces.
- D: Dependency Inversion - High level modules should not depend on low-level ones; both should depend on abstractions.

In classes

Take what you need, nothing more. This often relates to texas constructors but it can also hide in constructors or methods that take few parameters. Defer assembly to code that knows enough to assemble and instead just take the minimal interface you need to get your work done.

```
// Bad.
//  - Weigher uses hosts and port only to immediately construct another object.
class Weigher {
  private final double defaultInitialRate;

 Weigher(Iterable<String> hosts, int port, double defaultInitialRate) {
   this.defaultInitialRate = validateRate(defaultInitialRate);
   this.weightingService = createWeightingServiceClient(hosts, port);
  }
}

// Good.
class Weigher {
  private final double defaultInitialRate;

 Weigher(WeightingService weightingService, double defaultInitialRate) {
   this.defaultInitialRate = validateRate(defaultInitialRate);
    this.weightingService = checkNotNull(weightingService);
  }
}
```

Convenience constructors

If you want to provide a convenience constructor, a factory method or an external factory in the form of a builder you still can, but by making the fundamental constructor of a Weigher only take the things it actually uses it becomes easier to unit-test and adapt as the system involves.

In methods

If a method has multiple isolated blocks consider naming these blocks by extracting them to readable helper methods that do just one thing. The classic case is branched variable assignment. In the extreme, never do this:

```
void calculate(Subject subject) {
  double weight;
  if (useWeightingService(subject)) {
    try {
      weight = weightingService.weight(subject.id);
    } catch (RemoteException e) {
    throw new LayerSpecificException("Failed to look up weight for " + subject, e)
    }
  } else {
    weight = defaultInitialRate * (1 + onlineLearnedBoost);
  }

  // Use weight here for further calculations
}
```

Better method example

```
void calculate(Subject subject) {
  double weight = calculateWeight(subject);

  // Use weight here for further calculations
}

private double calculateWeight(Subject subject) throws LayerSpecificException {
  if (useWeightingService(subject)) {
    return fetchSubjectWeight(subject.id)
  } else {
    return currentDefaultRate();
  }
}

private double fetchSubjectWeight(long subjectId) {
  try {
    return weightingService.weight(subjectId);
  } catch (RemoteException e) {
   throw new LayerSpecificException("Failed to look up weight for " + subject, e)
  }
}

private double currentDefaultRate() {
```

```
  defaultInitialRate * (1 + onlineLearnedBoost);
}
```

A code reader that generally trusts methods do what they say can scan calculate quickly now and drill down only to those methods where I want to learn more.

Don’t Repeat Yourself (DRY)

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

For a more long-winded discussion on this topic, read here.

Extract constants whenever it makes sense

Centralize duplicate logic in utility functions

Avoid unnecessary code

Superfluous temporary variables.

```
// Bad.
//  - The variable is immediately returned, and just serves to clutter the code.
List<String> strings = fetchStrings();
return strings;

// Good.
return fetchStrings();
```

Unneeded assignment.

```
// Bad.
//   - The null value is never realized.
String value = null;
try {
  value = "The value is " + parse(foo);
} catch (BadException e) {
  throw new IllegalStateException(e);
}

// Good
String value;
try {
  value = "The value is " + parse(foo);
} catch (BadException e) {
  throw new IllegalStateException(e);
}
```

The ‘fast’ implementation

Don’t bewilder your API users with a ‘fast’ or ‘optimized’ implementation of a method.

```
int fastAdd(Iterable<Integer> ints);

// Why would the caller ever use this when there's a 'fast' add?
int add(Iterable<Integer> ints);
```

Complete example

```
/*
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */


package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */

    /** classVar1 documentation comment */
    public static int classVar1;

    /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;

    /** instanceVar1 documentation comment */
    public Object instanceVar1;
```

```
    /** instanceVar2 documentation comment */
    protected int instanceVar2;

    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;

    /**
     * ...constructor Blah documentation comment...
     */
    public Blah() {
        // ...implementation goes here...
    }

    /**
     * ...method doSomething documentation comment...
     */
    public void doSomething() {
        // ...implementation goes here...
    }

    /**
     * ...method doSomethingElse documentation comment...
     * @param someParam description
     */
    public void doSomethingElse(Object someParam) {
        // ...implementation goes here...
    }
}
```

Sources

- These notes are based on Twitter's Java Style Guide, licensed under the Apache License Version 2.0.
- See also the SEI CERT Oracle Coding Standard for Java, which focuses on reliability and security.


SOME RIGHTS RESERVED