Udacity Android Developer Nanodegree - Java Style Guide

Introduction

This style guide acts as the official guide to follow in your projects. Udacity reviewers will use this guide to grade your projects. For a more detailed set of rules with examples refer to this link. These are very strict rules followed in the Android Developers world.

Before you look into the Language and Style rules listed below, remember to BE CONSISTENT. Style guidelines are used to have a common vocabulary of coding.

Java Language Rules

Don't Ignore Exceptions

You must handle every Exception in your code in some principled way. The specific handling varies depending on the case.

Not Recommended:

```
void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) { }
}
```

Recommended:

Throw the exception up to the caller of your method.

```
void setServerPort(String value) throws NumberFormatException {
    serverPort = Integer.parseInt(value);
}
```

Throw a new exception that is appropriate to your level of abstraction.

```
void setServerPort(String value) throws ConfigurationException {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) {
        throw new ConfigurationException("Port " + value + " is nc
    }
}
```

Handle the error gracefully and substitute an appropriate value in the catch {} block.

```
/** Set port. If value is not a valid number, 80 is substituted. 
void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) {
        serverPort = 80;  // default port for server
    }
```

```
}
```

Don't Catch Generic Exception

It is inappropriate to catch generic Exception or Throwable, preferably not Throwable, because it includes Error exceptions as well. It is very dangerous. It means that Exceptions you never expected (including RuntimeExceptions like ClassCastException) end up getting caught in application-level error handling. It obscures the failure handling properties of your code.

Not Recommended:

```
try {
    someComplicatedIOFunction();        // may throw IOException
    someComplicatedParsingFunction();   // may throw ParsingExcep
    someComplicatedSecurityFunction();  // may throw SecurityExce
    // phew, made it all the way
} catch (Exception e) {                 // I'll just catch all exc
    handleError();                      // with one generic handl
}
```

Recommended:

- There might be certain test code and top-level code where you want to catch all kinds of errors (to prevent them from showing up in a UI, or to keep a batch job running). In that case you may catch generic Exception (or Throwable) and handle the error appropriately.
- Catch each exception separately as separate catch blocks after a single

try. Beware repeating too much code in the catch blocks.
- Refactor your code to have more fine-grained error handling, with multiple try blocks. Split up the IO from the parsing, handle errors separately in each case.
- Rethrow the exception. Many times you don't need to catch the exception at this level anyway, just let the method throw it.

Fully Qualify Imports

Not Recommended:

```
import foo.*;
```

Recommended:

```
import foo.Bar;
```

This makes it obvious what classes are actually used. Makes Android code more readable for maintainers.

Don't Use Finalizers

Finalizers are a way to have a chunk of code executed when an object is garbage collected.

In most cases, you can do what you need from a finalizer with good exception handling. If you absolutely need it, define a close() method (or the like), print a short log message from the finalizer and document exactly when that method needs to be called.

Java Style Rules

Javadoc Standard Comments

Recommended:

Every file should have a copyright statement at the top. Then a package statement and import statements should follow. Each block separated by a blank line. Finally, there is the class or interface declaration. In the Javadoc comments, describe what the class or interface does.

```
/*
 * Copyright (C) 2013 The Android Open Source Project
 */

package com.android.internal.foo;

import android.os.Blah;
import android.view.Yada;

/**
 * Does X and Y and provides an abstraction for Z.
 */

public class Foo {
    ...
}
```

Every class and nontrivial public method you write must contain a Javadoc comment with at least one sentence describing what the class or method

does. Make sure you start with a third person descriptive verb.

```
/** Returns the correctly rounded positive square root of a double
static double sqrt(double a) {
    ...
}
```

OR

```
/**
 * Constructs a new String by converting the specified array of
 * bytes using the platform's default character encoding.
 */
public String(byte[] bytes) {
    ...
}
```

You do not need to write Javadoc for trivial get and set methods.

Android does not currently enforce a specific style for writing Javadoc comments, but you should follow the instructions on How to Write Doc Comments for the Javadoc Tool.

Limit Variable Scope

The scope of local variables should be kept to a minimum. This increases the readability and maintainability of the code and reduce the likelihood of error.

Each variable should be declared in the innermost block that encloses all

uses of the variable.

Every local variable declaration should contain an initializer (if you don't have enough information to initialize, you can postpone this).

Loop variables should be declared in the for statement itself unless there is a compelling reason to do otherwise:

```
for (int i = 0; i < n; i++) {
    doSomething(i);
}
```

Order Import Statements

The order of importing statements is:

1. Android Imports
2. Imports from third parties(`com`, `junit`, `net`, `org`)
3. `java` and `javax` imports

To exactly match the IDE settings, the imports should be:

- Alphabetical within each grouping, with capital letters before lower case letters (e.g. Z before a).
- There should be a blank line between each major grouping (`android`, `com`, `junit`, `net`, `org`, `java`, `javax`).

Static imports can be a little tricky. It can either be interspersed with the remaining imports or below all other imports. We leave it to your judgement. Please be consistent.

Use Spaces for Indentation

We use 4 space indents for blocks. We never use tabs. When in doubt, be

consistent with code around you.

We use 8 space indents for line wraps, including function calls and assignments.

Not Recommended:

```
Instrument i =
    someLongExpression(that, wouldNotFit, on, one, line);
```

Recommended:

```
Instrument i =
        someLongExpression(that, wouldNotFit, on, one, line);
```

Follow Field Naming Conventions

- Non-public, non-static field names start with m.
- Static field names start with s.
- Other fields start with a lower case letter.
- Public static final fields (constants) are ALL_CAPS_WITH_UNDERSCORES.

Recommended:

```
public class MyClass {
    public static final int SOME_CONSTANT = 42;
    public int publicField;
    private static MyClass sSingleton;
    int mPackagePrivate;
```

```
    private int mPrivate;
    protected int mProtected;
}
```

Use Standard Brace Style

Braces do not go on their own line; they go on the same line as the code before them.

```
class MyClass {
    int func() {
        if (something) {
            // ...
        } else if (somethingElse) {
            // ...
        } else {
            // ...
        }
    }
}
```

We require braces around the statements for a conditional. Except, if the entire conditional (the condition and the body) fit on one line, you may (although not obligated to) put it all on one line.

Not Recommended:

```
if (condition)
    body();  // bad!
```

Recommended:

```
if (condition) {
    body();
}
```

and

```
if (condition) body();
```

Use TODO Comments

Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect.

TODOs should include the string TODO in all caps, followed by a colon:

```
// TODO: Remove this code after the UrlTable2 has been checked in
```

and

```
// TODO: Change this to use a flag instead of a constant.
```

If your TODO is of the form "At a future date do something" make sure that you either include a very specific date ("Fix by November 2005") or a very specific event ("Remove this code after all production mixers understand protocol V7.").

Treat Acronyms as Words

Treat acronyms and abbreviations as words in naming variables, methods, and classes. This way, the names are much more readable.

Not Recommended:

```
XMLHTTPRequest
getCustomerID();
class HTML{}
String URL;
long ID;
```

Recommended:

```
XmlHttpRequest
getCustomerId();
class Html{}
String url;
long id;
```

Avoid using Magic Numbers

A Magic Number is a hard-coded value that may change at a later stage, but that can be therefore hard to update.

Not Recommended:

```
public class Foo {
    public void setPin(String pin) {
        // don't do this
```

```
        if (pin.length() > 4) {
            throw new InvalidArgumentException("pin");
        }
    }
}
```

Make sure you refactor the above code by defining a numeric constant to store the pin size value. NOTE: Define numeric constants as `final`

Recommended:

```
public class Foo {
    public static final int MAX_PIN_SIZE = 4;
    public void setPin(String pin) {
        if (pin.length() > MAX_PIN_SIZE) {
            throw new InvalidArgumentException("pin");
        }
    }
}
```

Use Standard Java Annotations

Annotations should precede other modifiers for the same language element. Simple marker annotations (e.g. `@Override`) can be listed on the same line with the language element. If there are multiple annotations, or parameterized annotations, they should each be listed one-per-line in alphabetical order.

Android standard practices for the three predefined annotations in Java are:

- `@Deprecated`: The `@Deprecated` annotation must be used whenever

the use of the annotated element is discouraged. If you use the `@Deprecated` annotation, you must also have a `@deprecated` Javadoc tag (and vice versa) and it should name an alternate implementation.

- `@Override`: The `@Override` annotation must be used whenever a method overrides the declaration or implementation from a super-class.
- `@SuppressWarnings`: The `@SuppressWarnings` annotation should only be used under circumstances where it is impossible to eliminate a warning.When a `@SuppressWarnings` annotation is necessary, it must be prefixed with a TODO comment that explains the "impossible to eliminate" condition. For example:

```
// TODO: The third-party class com.third.useful.Utility.rotat
@SuppressWarnings("generic-cast")
List<String> blix = Utility.rotate(blax);
```

Log Sparingly

While logging is necessary, it has a significantly negative impact on performance and quickly loses its usefulness if it's not kept reasonably terse.

Define Fields in Standard Places

Fields should be defined either at the top of the file, or immediately before the methods that use them.

Write Short Methods

Methods should be kept small and focused. Although there is no hard limit on method length, think about whether method definitions that go above 40 lines can be broken up without harming the structure of the program.

Limit Line Length

Each line of text in your code should be at most 100 characters long.

Exception:

If a comment line contains an example command or a literal URL longer than 100 characters, that line may be longer than 100 characters for ease of cut and paste.

Import lines can go over the limit because humans rarely see them. This also simplifies tool writing.