Code Style Guidelines for Contributors

The rules below are not guidelines or recommendations, but strict rules. Contributions to Android generally *will not be accepted* if they do not adhere to these rules.

Not all existing code follows these rules, but all new code is expected to.

Java Language Rules

We follow standard Java coding conventions. We add a few rules:

Don't Ignore Exceptions

Sometimes it is tempting to write code that completely ignores an exception like this:

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

You must never do this. While you may think that your code will never encounter this error condition or that it is not important to handle it, ignoring exceptions like above creates mines in your code for someone else to trip over some day. You must handle every Exception in your code in some principled way. The specific handling varies depending on the case.

Anytime somebody has an empty catch clause they should have a creepy feeling. There are definitely times when it is actually the correct thing to do, but at least you have to think about it. In Java you can't escape the creepy feeling. -James Gosling

Acceptable alternatives (in order of preference) are:

- 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's 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 not valid.");

  }

}
```

- 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

  }

}
```

- Catch the Exception and throw a new `RuntimeException`. This is dangerous: only do it if you are positive that if this error occurs, the appropriate thing to do is crash.

```
/** Set port. If value is not a valid number, die. */
```

```
void setServerPort(String value) {

  try {

    serverPort = Integer.parseInt(value);

  } catch (NumberFormatException e) {

    throw new RuntimeException("port " + value " is invalid, ", e);

  }

}
```

Note that the original exception is passed to the constructor for RuntimeException. If your code must compile under Java 1.3, you will need to omit the exception that is the cause.

- Last resort: if you are confident that actually ignoring the exception is appropriate then you may ignore it, but you must also comment why with a good reason:

```
/** If value is not a valid number, original port number is used. */

void setServerPort(String value) {

  try {

    serverPort = Integer.parseInt(value);

  } catch (NumberFormatException e) {

    // Method is documented to just ignore invalid user input.

    // serverPort will just be unchanged.

  }

}
```

Don't Catch Generic Exception

Sometimes it is tempting to be lazy when catching exceptions and do something like this:

```
try {
    someComplicatedIOFunction();        // may throw IOException
    someComplicatedParsingFunction();   // may throw ParsingException
    someComplicatedSecurityFunction();  // may throw SecurityException
    // phew, made it all the way
} catch (Exception e) {                 // I'll just catch all exceptions
    handleError();                      // with one generic handler!
}
```

You should not do this. In almost all cases 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. It means if someone adds a new type of Exception in the code you're calling, the compiler won't help you realize you need to handle that error differently. And in most cases you shouldn't be handling different types of exception the same way, anyway.

There are rare exceptions to this rule: 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. You should think very carefully before doing this, though, and put in comments explaining why it is safe in this place.

Alternatives to catching generic Exception:

- Catch each exception separately as separate catch blocks after a single try. This can be awkward but is still preferable to catching all Exceptions. 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.

Remember: exceptions are your friend! When the compiler complains you're not catching an exception, don't scowl. Smile: the compiler just made it easier for you to catch runtime problems in your code.

Don't Use Finalizers

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

Pros: can be handy for doing cleanup, particularly of external resources.

Cons: there are no guarantees as to when a finalizer will be called, or even that it will be called at all.

Decision: we don't use finalizers. 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) and document exactly when that method needs to be called. See InputStream for an example. In this case it is appropriate but not required to print a short log message from the finalizer, as long as it is not expected to flood the logs.

Fully Qualify Imports

When you want to use class Bar from package foo,there are two possible ways to import it:

1. `import foo.*;`

Pros: Potentially reduces the number of import statements.

1. `import foo.Bar;`

Pros: Makes it obvious what classes are actually used. Makes code more readable for maintainers.

Decision: Use the latter for importing all Android code. An explicit exception is made for java standard libraries (`java.util.*`, `java.io.*`, etc.) and unit test code (`junit.framework.*`)

Java Library Rules

There are conventions for using Android's Java libraries and tools. In some cases, the convention has changed in important ways and older code might use a deprecated pattern or library. When working with such code, it's okay to continue the existing style. When creating new components never use deprecated libraries.

Java Style Rules

Use Javadoc Standard Comments

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. And then 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
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.android.internal.foo;

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

import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * 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. This sentence should start with a 3rd person descriptive verb.

Examples:

```
/** Returns the correctly rounded positive square root of a double value. */
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 such as setFoo() if all your Javadoc would say is "sets Foo". If the method does something more complex (such as enforcing a constraint or having an important side effect), then you must document it. And if it's not obvious what the property "Foo" means, you should document it.

Every method you write, whether public or otherwise, would benefit from Javadoc. Public methods are part of an API and therefore require Javadoc.

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

Write Short Methods

To the extent that it is feasible, methods should be kept small and focused. It is, however, recognized that long methods are sometimes appropriate, so no hard limit is placed on method length. If a method exceeds 40 lines or so, think about whether it can be broken up without harming the structure of the program.

Define Fields in Standard Places

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

Limit Variable Scope

The scope of local variables should be kept to a minimum. By doing so, you increase the readability and maintainability of your code and reduce the likelihood of error. Each variable should be declared in the innermost block that encloses all uses of the variable.

Local variables should be declared at the point they are first used. Nearly every local variable declaration should contain an initializer. If you don't yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do.

One exception to this rule concerns try-catch statements. If a variable is initialized with the return value of a method that throws a checked exception, it must be initialized inside a try block. If the value must be used outside of the try block, then it must be declared before the try block, where it cannot yet be sensibly initialized:

```
// Instantiate class cl, which represents some sort of Set
Set s = null;
try {
   s = (Set) cl.newInstance();
} catch(IllegalAccessException e) {
   throw new IllegalArgumentException(cl + " not accessible");
} catch(InstantiationException e) {
   throw new IllegalArgumentException(cl + " not instantiable");
}

// Exercise the set
s.addAll(Arrays.asList(args));
```

But even this case can be avoided by encapsulating the try-catch block in a method:

```
Set createSet(Class cl) {
   // Instantiate class cl, which represents some sort of Set
   try {
      return (Set) cl.newInstance();
   } catch(IllegalAccessException e) {
      throw new IllegalArgumentException(cl + " not accessible");
   } catch(InstantiationException e) {
      throw new IllegalArgumentException(cl + " not instantiable");
   }
}

...

// Exercise the set
Set s = createSet(cl);
s.addAll(Arrays.asList(args));
```

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);
}
```

and

```
for (Iterator i = c.iterator(); i.hasNext(); ) {
   doSomethingElse(i.next());
}
```

Order Import Statements

The ordering of import statements is:

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

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).

Originally there was no style requirement on the ordering. This meant that the IDE's were either always changing the ordering, or IDE developers had to disable the automatic import management features and maintain the imports by hand. This was deemed bad. When java-style was asked, the preferred styles were all over the map. It pretty much came down to our needing to "pick an ordering and be consistent." So we chose a style, updated the style guide, and made the IDEs obey it. We expect that as IDE users work on the code, the imports in all of the packages will end up matching this pattern without any extra engineering effort.

This style was chosen such that:

- The imports people want to look at first tend to be at the top (android)
- The imports people want to look at least tend to be at the bottom (java)
- Humans can easily follow the style
- IDEs can follow the style

The use and location of static imports have been mildly controversial issues. Some people would prefer static imports to be interspersed with the remaining imports, some would prefer them reside above or below all other imports. Additionally, we have not yet come up with a way to make all IDEs use the same ordering.

Since most people consider this a low priority issue, just use your judgement and 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. For example, this is correct:

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

and this is not correct:

```
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.

For example:

```
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. So:

```
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 (but are not obligated to) put it all on one line. That is, this is legal:

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

and this is legal:

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

but this is still illegal:

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

Limit Line Length

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

There has been lots of discussion about this rule and the decision remains that 100 characters is the maximum.

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.

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

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 it should name an alternate implementation. In addition, remember that a @Deprecated method is *still supposed to work.*
- If you see old code that has a @deprecated Javadoc tag, please add the @Deprecated annotation.

- @Override: The @Override annotation must be used whenever a method overrides the declaration or implementation from a super-class.
- For example, if you use the @inheritdocs Javadoc tag, and derive from a class (not an interface), you must also annotate that the method @Overrides the parent class's method.
- @SuppressWarnings: The @SuppressWarnings annotation should only be used under circumstances where it is impossible to eliminate a warning. If a warning passes this "impossible to eliminate" test, the @SuppressWarnings annotation *must* be used, so as to ensure that all warnings reflect actual problems in the code.
- When a @SuppressWarnings annotation is necessary, it must be prefixed with a TODO comment that explains the "impossible to eliminate" condition. This will normally identify an offending class that has an awkward interface. For example:
- // TODO: The third-party class com.third.useful.Utility.rotate() needs generics
- @SuppressWarnings("generic-cast")
- List<String> blix = Utility.rotate(blax);
- When a @SuppressWarnings annotation is required, the code should be refactored to isolate the software elements where the annotation applies.

Treat Acronyms as Words

Treat acronyms and abbreviations as words in naming variables, methods, and classes. The names are much more readable:

Good	Bad
XmlHttpRequest	XMLHTTPRequest
getCustomerId	getCustomerID
class Html	class HTML
String url	String URL
long id	long ID

Both the JDK and the Android code bases are very inconsistent with regards to acronyms, therefore, it is virtually impossible to be consistent with the code around you. Bite the bullet, and treat acronyms as words.

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.").

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. The logging facilities provides five different levels of logging:

- `ERROR`: This level of logging should be used when something fatal has happened, i.e. something that will have user-visible consequences and won't be recoverable without explicitly deleting some data, uninstalling applications, wiping the data partitions or reflashing the entire phone (or worse). This level is always logged. Issues that justify some logging at the ERROR level are typically good candidates to be reported to a statistics-gathering server.
- `WARNING`: This level of logging should used when something serious and unexpected happened, i.e. something that will have user-visible consequences but is likely to be recoverable without data loss by performing some explicit action, ranging from waiting or restarting an app all the way to re-downloading a new version of an application or rebooting the device. This level is always logged. Issues that justify some logging at the WARNING level might also be considered for reporting to a statistics-gathering server.
- `INFORMATIVE:` This level of logging should used be to note that something interesting to most people happened, i.e. when a situation is detected that is likely to have widespread impact, though isn't necessarily an error. Such a condition should only be logged by a module that reasonably believes that it is the most authoritative in that domain (to avoid duplicate logging by non-authoritative components). This level is always logged.
- `DEBUG`: This level of logging should be used to further note what is happening on the device that could be relevant to investigate and debug unexpected behaviors. You should log only what is needed to gather enough information about what is going on about your component. If your debug logs are dominating the log then you probably should be using verbose logging.
  This level will be logged, even on release builds, and is required to be surrounded by an `if (LOCAL_LOG)` or `if (LOCAL_LOGD)` block, where `LOCAL_LOG[D]` is defined in your class or subcomponent, so that there can exist a possibility to disable all such logging. There must therefore be no active logic in an `if (LOCAL_LOG)` block. All the string building for the log also

needs to be placed inside the if (LOCAL_LOG) block. The logging call should not be re-factored out into a method call if it is going to cause the string building to take place outside of the if (LOCAL_LOG) block.
There is some code that still says if (localLOGV). This is considered acceptable as well, although the name is nonstandard.

- VERBOSE: This level of logging should be used for everything else. This level will only be logged on debug builds and should be surrounded by an if (LOCAL_LOGV) block (or equivalent) so that it can be compiled out by default. Any string building will be stripped out of release builds and needs to appear inside the if (LOCAL_LOGV) block.

Notes:

- Within a given module, other than at the VERBOSE level, an error should only be reported once if possible: within a single chain of function calls within a module, only the innermost function should return the error, and callers in the same module should only add some logging if that significantly helps to isolate the issue.
- In a chain of modules, other than at the VERBOSE level, when a lower-level module detects invalid data coming from a higher-level module, the lower-level module should only log this situation to the DEBUG log, and only if logging provides information that is not otherwise available to the caller. Specifically, there is no need to log situations where an exception is thrown (the exception should contain all the relevant information), or where the only information being logged is contained in an error code. This is especially important in the interaction between the framework and applications, and conditions caused by third-party applications that are properly handled by the framework should not trigger logging higher than the DEBUG level. The only situations that should trigger logging at the INFORMATIVE level or higher is when a module or application detects an error at its own level or coming from a lower level.
- When a condition that would normally justify some logging is likely to occur many times, it can be a good idea to implement some rate-limiting mechanism to prevent overflowing the logs with many duplicate copies of the same (or very similar) information.
- Losses of network connectivity are considered common and fully expected and should not be logged gratuitously. A loss of network connectivity that has consequences within an app should be logged at the DEBUG or VERBOSE level (depending on whether the consequences are serious enough and unexpected enough to be logged in a release build).
- A full filesystem on a filesystem that is acceessible to or on behalf of third-party applications should not be logged at a level higher than INFORMATIVE.
- Invalid data coming from any untrusted source (including any file on shared storage, or data coming through just about any network connections) is considered expected and should not trigger any logging at a level higher then DEBUG when it's detected to be invalid (and even then logging should be as limited as possible).
- Keep in mind that the + operator, when used on Strings, implicitly creates a StringBuilder with the default buffer size (16 characters) and potentially quite a few other temporary String objects, i.e. that explicitly creating StringBuilders isn't more expensive than relying on the default '+' operator (and can be a lot more efficient in fact). Also keep in mind that code that calls Log.v() is compiled and executed on release builds, including building the strings, even if the logs aren't being read.

- Any logging that is meant to be read by other people and to be available in release builds should be terse without being cryptic, and should be reasonably understandable. This includes all logging up to the DEBUG level.
- When possible, logging should be kept on a single line if it makes sense. Line lengths up to 80 or 100 characters are perfectly acceptable, while lengths longer than about 130 or 160 characters (including the length of the tag) should be avoided if possible.
- Logging that reports successes should never be used at levels higher than VERBOSE.
- Temporary logging that is used to diagnose an issue that's hard to reproduce should be kept at the DEBUG or VERBOSE level, and should be enclosed by if blocks that allow to disable it entirely at compile-time.
- Be careful about security leaks through the log. Private information should be avoided. Information about protected content must definitely be avoided. This is especially important when writing framework code as it's not easy to know in advance what will and will not be private information or protected content.
- `System.out.println()` (or `printf()` for native code) should never be used. System.out and System.err get redirected to /dev/null, so your print statements will have no visible effects. However, all the string building that happens for these calls still gets executed.
- *The golden rule of logging is that your logs may not unnecessarily push other logs out of the buffer, just as others may not push out yours.*

Be Consistent

Our parting thought: BE CONSISTENT. If you're editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their if clauses, you should too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.

The point of having style guidelines is to have a common vocabulary of coding, so people can concentrate on what you're saying, rather than on how you're saying it. We present global style rules here so people know the vocabulary. But local style is also important. If code you add to a a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it. Try to avoid this.

Javatests Style Rules

Follow Test Method Naming Conventions

When naming test methods, you can use an underscore to seperate what is being tested from the specific case being tested. This style makes it easier to see exactly what cases are being tested.

For example:

```
testMethod_specificCase1 testMethod_specificCase2

void testIsDistinguishable_protanopia() {
    ColorMatcher colorMatcher = new ColorMatcher(PROTANOPIA)
    assertFalse(colorMatcher.isDistinguishable(Color.RED, Color.BLACK))
    assertTrue(colorMatcher.isDistinguishable(Color.X, Color.Y))
}
```

Reference

https://source.android.com/source/code-style.html