Be consistent

One of the simplest rules is BE CONSISTENT. If you're editing code, take a few minutes to look at the surrounding code and determine its style. If that code uses spaces around the if clauses, you should, too. If the code 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 readers can concentrate on what you're saying, rather than on how you're saying it. We present global style rules here so that you know the vocabulary, but local style is also important. If the code that you add to a file looks drastically different from the existing code around it, it throws readers out of rhythm when they read it. Try to avoid this.

Java language rules

Android follows standard Java coding conventions with the additional rules described below.

Don't ignore exceptions

It can be tempting to write code that ignores an exception, such as:

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

Don't do this. While you may think your code will never encounter this error condition or that it isn't important to handle it, ignoring this type of exception creates mines in your code for someone else to trigger some day. You must handle every exception in your code in a 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.
- /** 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 exceptions

It can be 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!
}
```

Don't do this. In almost all cases, it's inappropriate to catch generic Exception or Throwable (preferably not Throwable because it includes Error exceptions). It's dangerous because it means that exceptions you never expected (including runtime exceptions like ClassCastException) get caught in app-level error handling. It obscures the failure- handling properties of your code, meaning if someone adds a new type of exception in the code you're calling, the compiler won't point out that you need to handle the error differently. In most cases you shouldn't be handling different types of exceptions in the same way.

The rare exception to this rule is 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 these cases, you may catch generic Exception (or Throwable) and handle the error appropriately. Think carefully before doing this, though, and put in comments explaining why it's safe in this context.

Alternatives to catching generic exceptions:

- Catch each exception separately as part of a multi-catch block, for example:
- ```
  try {
      ...
  } catch (ClassNotFoundException | NoSuchMethodException e) {
      ...
  }
  ```

- Refactor your code to have more fine-grained error handling, with multiple try blocks. Split up the IO from the parsing, and 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 that exceptions are your friend! When the compiler complains that 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. While finalizers can be handy for cleanup (particularly of external resources), there are no guarantees as to when a finalizer will be called (or even that it will be called at all).

Android doesn't use finalizers. In most cases, you can use good exception handling instead. If you absolutely need a finalizer, 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's appropriate but not required to print a short log message from the finalizer, as long as it's 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:

- import foo.*;
  Potentially reduces the number of import statements.

- import foo.Bar;
  Makes it obvious what classes are used and the code is more readable for maintainers.

Use import foo.Bar; 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 however, never use deprecated libraries.

Java style rules

Use Javadoc standard comments

Every file should have a copyright statement at the top, followed by package and import statements (each block separated by a blank line), and finally the class or interface declaration. In the Javadoc comments, describe what the class or interface does.

/*

```
 * Copyright 2020 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 that 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 third 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 don't 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 has an important side effect), then you must document it. If it's not obvious what the property "Foo" means, you should document it.

Every method you write, public or otherwise, would benefit from Javadoc. Public methods are part of an API and therefore require Javadoc. Android doesn't enforce a specific style for writing Javadoc comments, but you should follow the instructions in How to Write Doc Comments for the Javadoc Tool.

Write short methods

When feasible, keep methods small and focused. We recognize 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

Define fields either at the top of the file or immediately before the methods that use them.

Limit variable scope

Keep the scope of local variables to a minimum. This increases the readability and maintainability of your code and reduces the likelihood of error. Declare each variable in the innermost block that encloses all uses of the variable.

Declare local variables at the point where 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, postpone the declaration until you do.

The exception is 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 can't 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));
```

However, you can even avoid this case 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));
```

Declare loop variables in the for statement itself unless there's 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 (for example, Z before a)
- Separated by a blank line between each major grouping (android, com, junit, net, org, java, javax)

Originally, there was no style requirement on the ordering, meaning IDEs were either always changing the ordering or IDE developers had to disable the automatic import management features and manually maintain the imports. This was deemed bad. When Java-style was asked, the preferred styles varied wildly and it came down to Android needing to simply "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, imports in all packages will match this pattern without extra engineering effort.

We chose this style such that:

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

Put static imports above all the other imports ordered the same way as regular imports.

Use spaces for indentation

We use four (4) space indents for blocks and never tabs. When in doubt, be consistent with the surrounding code.

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

Recommended

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

Not 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.

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

Put braces on the same line as the code before them, not on their own line:

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

We require braces around the statements for a conditional. Exception: 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. For example, this is acceptable:

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

and this is acceptable:

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

but this is not acceptable:

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

Limit line length

Each line of text in your code should be at most 100 characters long. While much discussion has surrounded this rule, the decision remains that 100 characters is the maximum *with the following exceptions*:

- 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).

Use standard Java annotations

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

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

- Use the @Deprecated annotation 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, add the @Deprecated annotation.
- Use the @Override annotation whenever a method overrides the declaration or implementation from a superclass. 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.
- Use the @SuppressWarnings annotation only under circumstances where it's impossible to eliminate a warning. If a warning passes this "impossible to eliminate" test, the @SuppressWarnings annotation *must* be used, 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 normally identifies 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, refactor the code 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 to make names more readable:

Good	Bad
XmlHttpRequest	XMLHTTPRequest
getCustomerId	getCustomerID

class Html	class HTML
String url	String URL
long id	long ID

As both the JDK and the Android code bases are inconsistent around acronyms, it's virtually impossible to be consistent with the surrounding code. Therefore, always 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. These comments 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 specific date ("Fix by November 2005") or a specific event ("Remove this code after all production mixers understand protocol V7.").

Log sparingly

While logging is necessary, it has a negative impact on performance and loses its usefulness if not kept reasonably terse. The logging facilities provide five different levels of logging:

- ERROR: Use when something fatal has happened, that is, something will have user-visible consequences and won't be recoverable without deleting some data, uninstalling apps, wiping the data partitions, or reflashing the entire device (or worse). This level is always logged. Issues that justify some logging at the ERROR level are good candidates to be reported to a statistics-gathering server.
- WARNING: Use when something serious and unexpected happened, that is, 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 app or rebooting the device. This level is always logged. Issues that justify logging at the WARNING level might also be considered for reporting to a statistics-gathering server.
- INFORMATIVE: Use to note that something interesting happened, that is, 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 believes that it's the most authoritative in that domain (to avoid duplicate logging by nonauthoritative components). This level is always logged.

- DEBUG: Use to further note what's happening on the device that could be relevant to investigate and debug unexpected behaviors. Log only what's needed to gather enough information about what's going on with your component. If your debug logs are dominating the log, then you should use verbose logging.
  This level is 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's a possibility to disable all such logging. Therefore, there must 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. Don't refactor the logging call out into a method call if it's going to cause the string building to take place outside of the if (LOCAL_LOG) block.

  There's some code that still says if (localLOGV). This is considered acceptable as well, although the name is nonstandard.

- VERBOSE: Use for everything else. This level is only 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 is 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 isn't otherwise available to the caller. Specifically, there's no need to log situations where an exception is thrown (the exception should contain all 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 apps, and conditions caused by third-party apps that are properly handled by the framework shouldn't 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 app 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 are fully expected, and shouldn't 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).
- Having a full file system on a file system that's accessible to or on behalf of third-party apps shouldn't 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 a network connection) is considered expected and shouldn't trigger any logging at a level higher than DEBUG when it's detected to be invalid (and even then logging should be as limited as possible).
- When used on String objects, the + operator implicitly creates a StringBuilder instance with the default buffer size (16 characters) and potentially other temporary String objects. So explicitly creating StringBuilder objects isn't more expensive than relying on the default + operator (and can be a lot more efficient). 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's meant to be read by other people and to be available in release builds should be terse without being cryptic, and should be understandable. This includes all logging up to the DEBUG level.
- When possible, keep logging on a single line. Line lengths up to 80 or 100 characters are acceptable. Avoid lengths longer than about 130 or 160 characters (including the length of the tag) if possible.
- If logging reports successes, never use it at levels higher than VERBOSE.
- If you're using temporary logging to diagnose an issue that's hard to reproduce, keep it at the DEBUG or VERBOSE level and enclose it with if blocks that allow for disabling it at compile time.
- Be careful about security leaks through the log. Avoid logging private information. In particular, avoid logging information about protected content. This is especially important when writing framework code as it's not easy to know in advance what will and won't be private information or protected content.
- Never use System.out.println() (or printf() for native code). System.out and System.err get redirected to /dev/null, so your print statements 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.*

Javatests style rules

Follow test method naming conventions and use an underscore to separate what's being tested from the specific case being tested. This style makes it easier to see which 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))
}
  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 instance of RuntimeException. This is dangerous, so do it only 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: The original exception is passed to the constructor for RuntimeException. If your code must compile under Java 1.3, you must omit the exception that is the cause.
As a last resort, if you're confident that ignoring the exception is appropriate then you may ignore it, but you must also comment why with a good reason.