Project Conventions
Coding Standards
Java/Android

Content

Conventions

We follow standard Java coding conventions. We add a few Android specific rules.

Package and Import Statements

The first non-comment line of most Java source files is a package statement. After that, import statements can follow. For example:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```

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`).

Fully Qualify Imports

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

```
import foo.*;
import foo.Bar;
```

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.*`)

Number per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level
int size; // size of table
```

Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

• No space between a method name and the parenthesis “(“ starting its parameter list
• Open brace “{” appears at the end of the same line as the declaration statement
• Closing brace “}” starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the “}” should appear immediately after the “{“

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }
    int emptyMethod() {}
    ...
}
```

- Methods are separated by a blank line

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;
}
```

Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples

Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.	`com.sun.eng`
		`com.apple.quicktime.v2`
		`edu.cmu.cs.bovik.cheese`

Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	`class Raster;` `class ImageSprite;`
Interfaces	Interface names should be capitalized like class names and always prefix with capital “I”.	`interface IDelegate;` `interface IStoring;`

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	`run();`
		`runFast();`
		`getBackground();`

Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters.	`int i;`
		`char c;`
		`float myWidth;`

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	`static final int MIN_WIDTH = 4;` `static final int MAX_WIDTH = 999;`

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.

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

while Statements

A while statement should have the following form:

```
while (condition) {
    statements;
}
```

do-while Statements

A do-while statement should have the following form:

```
do {
    statements;
} while (condition);
```

switch Statements

A switch statement should have the following form:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */

case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;

default:
    statements;
    break;
}
```

Comment Formats

- Block comments are used to provide descriptions of files, methods, data structures and algorithms.
- Short comments can appear on a single line indented to the level of the code that follows.
- Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements.
- The `//` comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code.

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.

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
    }
}
```

- 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.

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.

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) 2010 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.

Limit Variable Scope

The scope of local variables should be kept to a minimum. 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());
}
```

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.*
- `@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. 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.

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.").

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.

Sources

http://source.android.com/source/code-style.html
http://www.oracle.com/technetwork/java/codeconvtoc-136057.html