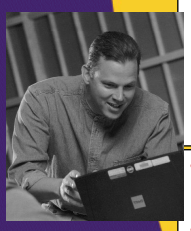
Android Coding Style Conventions

Originals of Slides and Source Code for Examples:
http://www.coreservlets.com/android-tutorial/

For live Android training, please see courses at http://courses.coreservlets.com/.



Taught by the author of *Core Servlets and JSP*, *More Servlets and JSP*, and this Android tutorial. Available at public venues, or customized versions can be held on-site at your organization.

- Courses developed and taught by Marty Hall
  - Android development, JSF 2, servlets/JSP, Ajax, jQuery, Java 6 programming, custom mix of topics
  - Ajax courses can concentrate on 1 library (jQuery, Prototype/Scriptaculous, Ext-JS, Dojo, etc.) or survey several
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Spring, Hibernate/JPA, EJB3, GWT, RESTful and SOAP-based Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- **Why follow conventions?**
- **Valuable conventions**
  - Ones that are widely considered good practice for any Java project (based on general Java industry consensus)
- **Tolerable conventions**
  - Ones that do no harm, but are of questionable value (in Marty's highly subjective opinion)
- **Dubious conventions**
  - Ones that we would have been better off without (in Marty's highly subjective opinion)

Overview

Official Android Code Conventions

- **Required for**
  - Code contributed to Android project
- **Used in**
  - All official tutorials and (supposedly) all source code
- **Suggested for**
  - Code submitted to the app store
  - Any Android project
- **Details**
  - http://source.android.com/source/code-style.html
- **Eclipse preferences file**
  - Downloadable from coreservlets.com from this section of the Android Tutorial.
    - Sets spacing, brace style, and use of @Override

Pros and Cons of Following Conventions

- **Pros**
  - Consistent with official tutorials and Android source
  - More familiar to Android developers who join your team
- **Cons**
  - Inconsistent with Java code you wrote before
  - Less familiar to other Java developers
  - Simply bothers you.
    - Java developers often have strong personal preferences
- **My recommendations**
  - Most conventions are best practices anyhow
    - Definitely follow those
  - Most others are neither worse nor better than alternatives
    - Probably follow those
  - A few are (arguably) bad or at least wrong in some situations
    - Ignore those if the situation warrants it

Conventions that are Good Standard Practice

(For any Java project)

Indentation: blocks that are nested more should be indented more

- **Yes**

```
blah;
blah;
for(...) {
    blah;
    blah;
    for(...) {
        blah;
        blah;
    }
}
```

- **No**

```
blah;
blah;
for(...) {
blah;
blah;
for(...) {
blah;
blah;
}
}
```

Indentation: blocks that are nested the same should be indented the same

- **Yes**

```
blah;
blah;
for(...) {
    blah;
    blah;
    for(...) {
        blah;
        blah;
    }
}
```

- **No**

```
blah;
 blah;
for(...) {
   blah;
    blah;
    for(...) {
       blah;
        blah;
  }
  }
```


Break Things into Small Pieces

- **Write short methods**
  - No official limit, but try to keep methods short and focused. Think often about how to refactor your code to break it into smaller and more reusable pieces.
    - This is good advice in any language.
    - This also shows why overly strict rules on the length of comments can be counter productive by encouraging developers to write long methods to avoid writing docs.
- **Keep lines short**
  - They have a strict rule of 100 characters except for imports or comments that contain URLs or commands that cannot be broken up.
    - Not sure 100 is the magic number, but short lines are good practice anyhow.

Follow Normal Capitalization Rules

- **Classes start with uppercase letter**

```
public class SomeClass { … }
```

- **Constants use all caps**

```
public static final double GOLDEN_RATIO =
    (1 + Math.sqrt(5.0))/2;
```

- **Everything else starts with lowercase letter**
  - Instance variables, local variables, parameters to methods, package names
- **Extra rule**
  - Use words for acronyms, not all uppercase
    - getUrl, not getURL
      - This is good advice in Web apps also

Use JavaDoc

- **Use JavaDoc from the beginning**
  - Don't wait until the code is finished. Short comments are fine, but use *some*. Explain purpose and non-obvious behavior. Don't explain standard Java constructs.
- **Document every class**

```
/** Represents a collection of Blahs. Used to … **/
public class Foo { … }
```

- **Document anything public**
  - Methods
  - Constructors
  - Instance variables (but *very* rare to have public ones)
- **Review Oracle JavaDoc guidelines**
  - http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Use @Override

- **Use @Override when you override methods from parent class**
  - Won't be caught until run time

```
public void oncreate(Bundle savedInstanceState) {
    …
}
```

  - Will be caught at compile time

```
@Override
public void oncreate(Bundle savedInstanceState) {
    …
}
```

- **Guidelines are silent regarding interfaces**
  - But, in Java 6 or later, I prefer to also use @Override when implementing methods from interface

Use Other Standard Annotations when Warranted (but Rarely)

- **@Deprecated**
  - If you use a deprecated method, add this annotation to your method. Also add @deprecated JavaDoc tag explaining why it was necessary to use depracated code.
    - Of course, try hard to avoid use of deprecated methods
- **@SuppressWarnings**
  - Generic collections are prohibited from doing extra work at run time, so casting to generic type can cause warning that Java can't verify the types. Sometimes unavoidable
    - @SuppressWarnings("unchecked")
    - Other similar situations when *making* generic types
  - Android guidelines require a TODO comment in these cases, saying why you cannot avoid the situation

Limit the Scope of Variables

- **Use narrowest scope possible**
  - Variables should be declared in the innermost block that encloses all uses of the variable.
    - E.g., if variable is only used inside if statement, declare it inside if statement.
  - Yes

```
if (…) {
    double d = someCalculation(…);
    doSomethingWith(d);
} else {
    // No use of d
}
```

  - No

```
double d = 0;
if (…) { … } else { … }
```


Initialize Local Variables when Declared

- **Initialize (almost) all local variables**
  - Yes

```
String s = "Hello";
```

  - No

```
String s;
…
s = "Hello";
```

  - Exception: try/catch blocks

```
int n;
try {
    n = Integer.parseInt(someString);
} catch(NumberFormatException nfe) {
    n = 10;
}
```

Put Braces on Conditionals

- **Always use braces for if statements**
  - Even if there is only one thing to do
    - Yes

```
if (…) {
    doSomething();
}
```

    - No

```
if (…)
    doSomething();
```

- **Guidelines give small exception**
  - If there is only one thing to do *and* it is all on one line
    - Tolerated (grudgingly?)

```
if (…) doSomething();
```


Use TODO Comments for Temporary Code

- **Use "// TODO: … " for code that needs to be changed later**
  - Situations
    - Temporary fix
    - OK but not great
    - Works for small sizes, but bad performance in future when data sets get bigger.
  - Examples:

```
// TODO: Switch to a Map when you have more entries
// TODO: Remove after UrlTable2 has been checked in
```

- **Eclipse note**
  - Eclipse puts TODO in bold and puts check mark in left margin of code

Avoid Finalizers

- **Do not use finalize()**
  - Idea
    - finalize() gets called when an object is garbage collected, so you can do cleanup work then (such as closing socket connections)
  - Problem
    - No guarantee when (or even if) finalizer will be called
  - Guidelines
    - Don't use them.
- **Good news**
  - Finalizers have long ago fallen out of favor, and many Java developers don't even know what they are.

Conventions that Don't Hurt

(No harm in following them, but their value is questionable)

Put Open Braces with Preceding Code

- **Put { with previous line, not on its own line**
  - Yes

```
public void foo() {
    if (...) {
        doSomething();
    }
}
```

  - No

```
public void foo()
{
    if (...)
    {
        doSomething();
    }
}
```


Indent 4 Spaces for Blocks

- **Indent 4 spaces when starting a block**
  - Yes

```
public void foo() {
    if (...) {
        doSomething();
    }
}
```

  - No

```
public void foo() {
  if (...) {
    doSomething();
  }
}
```

Indent 8 Spaces for Lines

- **Indent 8 spaces when splitting a line**
  - Yes

```
String s =
        somethingVeryLong(…);
```

  - No

```
String s =
    somethingVeryLong(…);
```


Fully Qualify Imports

- **List each class name; don't use ***
  - Yes
    - import android.widget.Button;
    - import android.widget.CheckBox;
    - import android.widget.EditText;
  - No
    - import android.widget.*;
- **Exception**
  - Can use * for java or javax packages
    - Permitted
      - import java.util.*;

Order Import Statements

- **First**
  - Android packages
    - import android.foo.Bar;
- **Second**
  - Third party packages
    - import com.coreservlets.utils.RandomUtils;
- **Third**
  - Standard java or javax packages
    - import java.util.*;
- **Within each group**
  - Alphabetical (uppercase Z before lowercase a)
- **Separating groups**
  - Blank line between each major grouping

Start JavaDoc Comments with 3rd Person Verb

- **Examples**
  - Yes
    - Represents a …
    - Responds to mouse clicks with …
    - Deletes …
  - No
    - This class …
    - This method …
- **Android's own docs are inconsistent**
  - Many (most?) classes start with "This class" or similar.
    - E.g., View, Activity, LinearLayout

Questionable Conventions

(You would have been better off without them)

Start Instance Variables with "m" (normal) or "s" (static)

- **Use "m" for non-public, non static fields**
  - "m" for "member variable" or "data member"
    - Yes
      - private String mFirstName;
      - private boolean mIsMarried;
    - No
      - private String firstName;
      - private boolean isMarried;
- **Use "s" for static (non-final) fields**
    - Yes
      - private static double sBiggestRadius;
    - No
      - private static double biggestRadius;
- **Marty's opinion**
  - Results in less readable names with no real benefit

Impact of Naming Convention on Constructors

Standard Style

```
public class Person {
    public String firstName, lastName;

    public Person(String firstName,
                  String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    ...
}
```

Android Style

```
public class Person {
    public String mFirstName, mLastName;

    public Person(String firstName,
                  String lastName) {
        mFirstName = firstName;
        mLastName = lastName;
    }

    ...
}
```


Never Ignore Exceptions

- **Avoid empty catch blocks**
  - Yes

```
try {
    …
} catch(SomeException se) {
    doSomethingReal();
}
```

  - No

```
try {
    …
} catch(SomeException se) { }
```

  - Marty's opinion
    - Usually, but not *always*, a good rule

Why Ignoring Exceptions Rule is Too Strict

- **Can make shorter code with same safety**
  - Android style

```
int n;
try {
    n = Integer.parseInt(…);
} catch(NumberFormatException nfe) {
    n = 10;
}
```

  - Shorter style if you could ignore exceptions

```
int n = 10;
try {
    n = Integer.parseInt(…);
} catch(NumberFormatException nfe) { }
```


Why Ignoring Exceptions Rule is Too Strict (Continued)

- **Sometimes there is nothing to be done**

```
try {
    Thread.sleep(…);
} catch(InterruptedException ie) {
    // What could you do here?
}
doSomethingAfterThePause();
```

Don't Catch Generic Exception

- **List each Exception type**
  - Yes

```
try {
    …
} catch(ExceptionType1 et1) {
    …
}  catch(ExceptionType2 et2) {
    …
}
```

  - No

```
try {
    …
} catch(Exception e) {
    …
}
```


Why Generic Exception Rule is (Arguably) Too Strict

- **Listing each type is almost always best**
  - So exceptions you didn't expect don't get caught there
  - So real failure-handling is not obscured
- **Sometimes combining is concise and safe**
  - E.g., if someString could be null, you could have either NumberFormatException or NullPointerException. But, in both cases, you just want to use original value for n.

```
int n = 10;
try {
    n = Integer.parseInt(someString);
} catch(Exception e) { }
```

Wrap-Up

Summary

- **Strictly follow conventions that reflect widely accepted best practices**
  - Also, familiarize yourself with best practices.
    - All developers who have worked with Java more than two years full time should read Josh Bloch's *Effective Java* (2nd Edition).
      - Even experts will learn something new and valuable
- **For other conventions, if you don't strongly object, follow the conventions anyhow**
  - Even if you don't see any real value
- **If convention really bothers you, ignore it**
  - Assuming it is not in category of generally accepted best practices. Personal taste plays role in many of them.

Questions?