Private Resources in Android Libraries

The Problem

A library will often contain a number of resources intended only for internal usage by the library; they are not intended to be referenced directly by developers using the library.

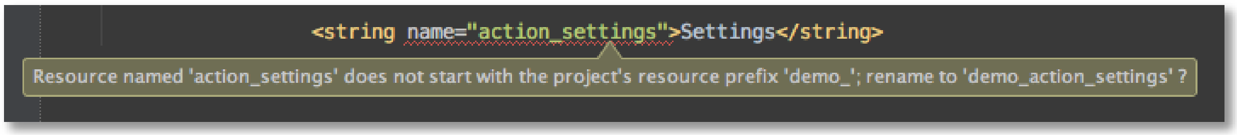
Take appcompat for example: it ships with something like 400 resources. Now that it supports Material Design, some of the attributes are clearly intended for use, such as attributes controlling the Toolbar widget etc, but a large number of the resources are for example involved in implementing the backport of the ActionBar: internal layouts for the different action bar modes, ninepatches for backgrounds, and so on.

Library resources share the same symbol namespace as the user's own code, and this causes two problems:

1. Name conflicts. If two unrelated libraries (or a library and the developer's own code) happen to pick the same intuitive name for a resource, many problems can arise. As a simple example, let's imagine they both define a string message with the same key, one will override the other with completely unrelated and misleading contents; if there are formatting argument mismatches there can be crashes (which is even more likely for overriding layouts where findViewById calls will fail to find id's to bind to etc.)

2. Noise. Code completion (and other resource name lookup such as in the layout editor property sheet) becomes much harder to use when the namespace is filled up with a lot of resources from libraries that are not interesting (or known) to the developer. Take the appcompat example again: AppCompat ships with over a hundred symbols with the prefix "abc", so if you're in the IDE and you're invoking code completion on @drawable/ to find your launcher icon, the only thing you'll see are abc_ resources from appcompat since they sort alphabetically to the top!

To avoid name conflicts, we have a partial solution in the Gradle plugin: The library prefix. It doesn't solve the real namespace problem, but it allows a library author to pick a prefix for their library, and then the tools will flag any resources that are accidentally not using this prefix (therefore polluting the namespace outside of the narrow prefix claimed by the given library). You could still have two libraries picking the same prefix, but that's less likely than general resource name clashes for obvious names.



```
<string name="action_settings">Settings</string>
```

Resource named 'action_settings' does not start with the project's resource prefix 'demo_'; rename to 'demo_action_settings' ?

Private Resources

One possibility which would help is to let developers designate some resources as “private”. This means that the library itself can use the resource, but clients can not.

The Android framework itself ships with both public and private resources; only resources that are explicitly marked as public are exposed in the SDK and are available to developers, and only the framework code itself can reference the other attributes. (Many Google applications used to access them anyway using the special *android syntax, but aapt no longer allows this and the best practice is to copy the resources for private use). Resources are made public in the framework by using the <public> resource type, where you specify the name, type, and assigned constant value.

Proposal

Allow developers to indicate which resources are intended to be exposed as part of the API, and then everything else remains private.

Declare Public Versus Declare Hide

I considered letting you “hide” resources rather than explicitly “show” resources, similar to how public methods in the framework are implicitly part of the SDK unless specifically hidden with @hide. For example, you could mark a layout hidden by placing tools:private=”true” on the root tag of the layout XML file. However, unlike Java code, the same logical resource is often defined in multiple locations, and some of them are not textual (e.g. PNG files).

Public.xml

Proposal: Developers can define resources to be made public in the same way that this is already done in the framework: by adding a <public> declaration in any resource file:

```
<resources>
   <public name="mylib_app_name" type="string"/>
   <public name="mylib_public_string" type="string"/>
</resources>
```

As with all other value resources, you can define these in any file, but the convention we’ll encourage via our New Android Library template is to place these in a file named “public.xml”.

One key difference between this public declaration and the one used in the framework is that in the framework, you also specify the specific R constant you want aapt to assign to it. That attribute does not apply here (and obviously can’t work: the values will have to be adjusted when merged into the final app; otherwise two unrelated libraries could specify the same constant value that would conflict.)

AAR Packaging

The Android Gradle plugin will be modified such that when building a library, it grabs the public resource definitions and extracts them into a special file, “public.txt”, which is then packaged inside the AAR file next to “R.txt”. This file has a very simple syntax: type + space + name + newline. For the example above, it would be

string mylib_app_name
string mylib_public_string

The builder API will also be updated to let clients look up the public symbols for an AAR. The AndroidLibrary class already has a number of getter methods for looking up extracted AAR contents: it would have a “File getPublicResources()” method which would return the location of the public symbols.

(We might want to also record a third piece of data on each line: a unique name assigned to the field at build time; see the “Renaming Resources On The Fly” section below. In the example above, the first line would be “string mylib_app_name mylib_app_name_1f26aab5)

IDE Usage

The IDE needs to make a deliberate decision for each feature whether it wants to know about all resources, or just the public resources.

For layout rendering for example, it will need to track all resources, public and private. But for code completion, it should filter the results to only include public resources.

To handle this, the IDE will read the public.txt files from the libraries and use this to filter resource queries.

Lint Usage

The above filtering will solve problem #2 listed in the introduction: filtering out noise from private resources in large libraries.

But what about problem #1: name conflicts?

First, we can use the new information to distinguish between an **accidental** override from an **intentional** override.

Let’s say a library defines a resource called “list_item_layout”, but marks it private. And then your app also creates a layout named list_item_layout. Clearly, you didn’t intentionally override the layout, so here we should either break the build, or lint should flag a big warning that something is amiss.

We can go one step further and start requiring overriding resources to be deliberate, using for example tools:override=”true”, or something like that; an XML version of the @Override annotation. Just like we’ve configured @Override to be required for any override in the tools codebase, we could have an option in the Gradle plugin which requires all resource overrides (across library/module boundaries) to be deliberately annotated with tools:override.

Renaming Private Resources On The Fly

(NOTE: This is not a required part of the proposal, but it explores a new feature we can implement if let users declare private resources.)

We might be able to even one step further, and **allow** resource name conflicts for private resources!

Assume for a minute that your resources are not accessed via reflection/dynamic name lookup (Resources#getIdentifier()).

In that case we can rename private resources **at build time** to something unique! We would look through the class graph, and perform a field rename of the resource identifiers for private resources. We would also need to record this in the generated public.txt file, perhaps as a third entry on each public.txt line.

The Gradle plugin would run a class visitor over the generated classes.jar, look to see if the private resources leak in any way (for example via Resources#getIdentifier calls, something which is not very likely for most libraries), and then pick a new unique name (for example by hashing it with the library name) to assign to the symbol. This is the name we would then expose via R.txt, also marked a private.

This would effectively remove the name-clash problem across libraries and apps for private resources. There is still a problem for public resources, but at least the population of names is smaller.