mobile

Context, What Context?

Written by: Dave Smith on June 20, 2013

Google + Instapaper

Context is probably the most used element in Android applications…it may also be the most misused.

Context objects are so common, and get passed around so frequently, it can be easy to create a situation you didn't intend. Loading resources, launching a new Activity, obtaining a system service, getting internal file paths, and creating views all require a *Context* (and that's not even getting started on the full list!) to accomplish the task. What I'd like to do is provide for you some insights on how *Context* works alongside some tips that will (hopefully) allow you to leverage it more effectively in your applications.

Context Types

Not all *Context* instances are created equal. Depending on the Android application component, the *Context* you have access to varies slightly:

Application – is a singleton instance running in your application process. It can be accessed via methods like *getApplication()* from an Activity or Service, and *getApplicationContext()* from any other object that inherits from

Context. Regardless of where or how it is accessed, you will always receive the same instance from within your process.

Activity/Service – inherit from *ContextWrapper* which implements the same API, but proxies all of its method calls to a hidden internal *Context* instance, also known as its base context. Whenever the framework creates a new Activity or Service instance, it also creates a new *ContextImpl* instance to do all of the heavy lifting that either component will wrap. Each Activity or Service, and their corresponding base context, are unique per-instance.

BroadcastReceiver – is not a *Context* in and of itself, but the framework passes a *Context* to it in *onReceive()* each time a new broadcast event comes in. This instance is a *ReceiverRestrictedContext* with two main functions disabled; calling *registerReceiver()* and *bindService()*. These two functions are not allowed from within an existing *BroadcastReceiver.onReceive()*. Each time a receiver processes a broadcast, the *Context* handed to it is a new instance.

ContentProvider – is also not a *Context* but is given one when created that can be accessed via *getContext()*. If the ContentProvider is running local to the caller (i.e. same application process), then this will actually return the same Application singleton. However, if the two are in separate processes, this will be a newly created instance representing the package the provider is running in.

Saved References

The first issue we need to address comes from saving a reference to a *Context* in an object or class that has a lifecycle that extends beyond that of the instance you saved. For example, creating a custom singleton that requires a *Context* to load resources or access a ContentProvider, and saving a reference to the current Activity or Service in that singleton.

Bad Singleton

```
public class CustomManager {
    private static CustomManager sInstance;

    public static CustomManager getInstance(Context context) {
        if (sInstance == null) {
            sInstance = new CustomManager(context);
        }

        return sInstance;
    }

    private Context mContext;

    private CustomManager(Context context) {
        mContext = context;
    }
}
```

The problem here is we don't know where that *Context* came from, and it is not safe to hold a reference to the object if it ends up being an Activity or a Service. This is a problem because a singleton is managed by a single static reference inside the enclosing class. This means that our object, and ALL the other objects referenced by it, will never be garbage collected. If this *Context* were an Activity, we would effectively hold hostage in memory all the views and other potentially large objects associated with it; creating a leak.

To protect against this, we modify the singleton to always reference the application context:

Better Singleton

```
public class CustomManager {
    private static CustomManager sInstance;

    public static CustomManager getInstance(Context context) {
        if (sInstance == null) {
            //Always pass in the Application Context
            sInstance = new CustomManager(context.getApplicationContext());
        }

        return sInstance;
    }

    private Context mContext;

    private CustomManager(Context context) {
        mContext = context;
    }
}
```

Now it doesn't matter where our *Context* came from, because the reference we are holding is safe. The application context is itself a singleton, so we aren't leaking anything by creating another static reference to it. Another great example of places where this can crop up is saving references to a *Context* from inside a running background thread or a pending *Handler*.

So why can't we ***always*** just reference the application context? Take the middleman out of the equation, as it were, and never have to worry about creating leaks? The answer, as I alluded to in the introduction, is because one *Context* is not equal to another.

Context Capabilities

The common actions you can safely take with a given *Context* object depends on where it came from originally. Below is a table of the common places an application will receive a *Context*, and in each case what it is useful for:

	Application	**Activity**	**Service**	**ContentProvider**	**BroadcastReceiver**
Show a Dialog	NO	YES	NO	NO	NO
Start an Activity	NO[1]	YES	NO[1]	NO[1]	NO[1]
Layout Inflation	NO[2]	YES	NO[2]	NO[2]	NO[2]
Start a Service	YES	YES	YES	YES	YES
Bind to a Service	YES	YES	YES	YES	NO
Send a Broadcast	YES	YES	YES	YES	YES
Register BroadcastReceiver	YES	YES	YES	YES	NO[3]
Load Resource Values	YES	YES	YES	YES	YES

1. An application CAN start an Activity from here, but it requires that a new task be created. This may fit specific use cases, but can create non-standard back stack behaviors in your application and is generally not recommended or considered good practice.
2. This is legal, but inflation will be done with the default theme for the system on which you are running, not what's defined in your application.
3. Allowed if the receiver is *null*, which is used for obtaining the current value of a sticky broadcast, on Android 4.2 and above.

User Interface

You can see from looking at the previous table that there are a number of functions the application context is not properly suited to handle; all of them related to working with the UI. In fact, the only implementation equipped to handle all tasks associated with the UI is Activity; the other instances fare pretty much the same in all categories.

Luckily, these three actions are things an application doesn't really have any place doing outside the scope of an Activity; it's almost like the framework was designed that way on purpose. Attempting to show a *Dialog* that was created with a reference to the application context, or starting an Activity from the application context will throw an exception and crash your application…a strong indicator something has gone wrong.

The less obvious issue is inflating layouts. If you read my last piece on layout inflation, you already know that it can be a slightly mysterious process with some hidden behaviors; using the right *Context* is linked to another one of those behaviors. While the framework will not complain and will return a perfectly good view hierarchy from a *LayoutInflater* created with the application context, the themes and styles from your app will not be considered in the process. This is because Activity is the only *Context* on which the themes defined in your manifest are actually attached. Any other instance will use the system default theme to inflate your views, leading to a display output you probably didn't expect.

The Intersection of these Rules

Invariably, someone will arrive at the conclusion that these two rules conflict. There is a case in the application's current design where a long-term reference must be saved and we must save an Activity because the tasks we want to accomplish include manipulation of the UI. If that is the case, I would urge you to reconsider your design, as this would be a textbook instance of *fighting the framework*.

The Rule of Thumb

In most cases, use the *Context* directly available to you from the enclosing component you're working within. You can safely hold a reference to it as long as that reference does not extend beyond the lifecycle of that component. As soon as you need to save a reference to a *Context* from an object that lives beyond your Activity or Service, even temporarily, switch that reference you save over to the application context.

For future insights and tutorials, please subscribe to our newsletter below the comments.

Dave Smith

Dave Smith is a Senior Engineer at Double Encore, Inc., a leading mobile development company. Dave is an expert in developing mobile applications that integrate with custom hardware and devices. His recent focus lies mainly in integrating the Android platform with embedded SoC hardware. He is a published author and speaks regularly at conferences on topics related to Android development. You can follow Dave on Twitter and Google+.

Article

Add your voice to the discussion:

- Facebook
- Twitter

Denver, CO • Atlanta, GA • Seattle, WA • New York, NY

Email us: mobile@possible.com

Call us toll-free: 1-888-247-4560