Android development: What I wish I had known earlier

July 30, 2013

At the beginning of the year, I jumped from web development to Android development because of this. Both Java in general and programming for Android in particular were totally new for me (and everyone else on the team who jumped with me). In fact, one of the first things I did was adding something back to Java that I was really missing very quickly after having worked with C# for a long time: the `var` keyword.

Half a year later I've learned a lot, and there are quite a few things I know today that I would have loved to know back then. By no means is this a "Beginning Android" tutorial or a list of all things you need to know. And I'm obviously still lightyears away from being able to call myself an Android pro. But these are some tips that would have been great back then, when I knew *even less*. So here's hoping that they may help others as well.

I've tried to order them from most to least likely to be relevant to others. As always, I may be wrong.

Embrace the source

This is something I did from the start, so not actually of the category "I wish someone had told me," but it deserves to be said anyway. Android is open source, so if you're trying to track down some weird behavior or understand the underpinnings of what's going on, just look at the source code! For the most part, it's very readable code, and sooner or later you'll feel quite comfortable skimming it, even if you don't understand every last detail. Personally I use the GitHub mirrors of the core framework and the v4 support library, because GitHub lends itself quite well for browsing, and its code search has come in handy a bunch of times as well.

Assume things make sense

For someone coming in fresh to Android development, quite a few things look weird and different. It doesn't help that there are often inconsistencies like

> *Should I override* `onSomething()`*? Or does this class have* `setOnSomethingListener()`*? Or* `addOnSomethingListener()`*, so there can be multiple? Or any combination of the*

> *above?*

and that the quality of documentation out there is, let's say, very heterogeneous.

This makes it easy to just say "Meh, I'll just do it the way I've always done it," and use ugly hacks to shoehorn your approach into your code. The next two sections give some examples of this, but it's worth to keep in mind in general: Chances are that Android actually has a way of solving the problem you're having, it may just look different to what you're expecting. `setArguments()` vs. custom constructors, `startActivityForResult()` instead of return values, `onSaveInstanceState()` vs. global variables, et cetera. More often then not, there may be good reasons for the differences, for example to accommodate the asynchronous RPC nature of activities, which may seem strange but has a lot of plus sides.

So when you feel you're working against the framework, take a step back and look for a *different* way to solve the *same* problem. It may exist, and using it may save you from wanting to rip your hair out six months later.

Your application is not immortal

If you load some data in your launcher activity and then store it on your application object (statically or not), you're in for a surprise. Unfortunately, a great many people on the internet will tell you to do just that.

But it's entirely possible – and likely – that at some point your app gets moved to the background, Android kills the process to make room for other apps the user is interacting with, and then later the user returns to your app, and Android restores your full back stack of activities to the state in which the user left it. If you now expect to have the cached data on your application object (after all, the user is deep inside your app), your app will probably crash with a NullPointerException.

I'll leave the details to Developer Phil's excellent post. Read it.

A great way to test your app's behavior when the process is killed and restarted, at least in new Android versions that have this option, is this: In the advanced developer settings, you can change the number of background processes. Set this to "no background processes", then play with your app for a bit. Then press the home button and open, say, the browser. Leave the browser again and via "recents" go back to your app, and you'll see how it handles being restarted.

Just remember to turn off this developer setting when you're done, because it eats battery like crazy and can cause your phone to overheat.

Fragments

Fragments were introduced in Android 3, a.k.a. Honeycomb. That's no coincidence; Honeycomb was tailored specifically for tablets, and the raison d'être for fragments is

the possibility to create user interface elements that can exist both more or less on their own (e.g. on phones, where there's not a lot of space), and as part of a larger interface (e.g. on tablets, where there's enough space to show a list of articles and the article itself at the same time).

There's a joke in here about the fragmented Android ecosystem, but I'm not one to make bad puns.

Fragments are really a light flavor of activities. The problem with this is that it looks like you have to make a choice: Should I use several fragments in a single activity for task X, or should I use several activities? The answer when just starting out with Android development is "use activites."

It takes a while to wrap your head around the various concepts that Android introduces. Intents, intent flags, activities, tasks, backstack, lifecycles of various components, and so on: As a beginner, when these things haven't fallen into place yet for you, chances are you'll unconsciously make a lot of assumptions, and that many of these assumptions are wrong. Fragments are yet another element in this bucket of new stuff that you're learning.

And because fragments behave a lot like activites, you'll treat them like they're pretty much the same thing, because all the details aren't clear to you yet. And then at some point, these details are going to bite you, hard. A few examples:

- A fragment's lifecycle is similar to an activity's, but there are subtle differences. And since you're only learning about the lifecycles, having two similar but slightly different things to learn at the same time is bound to cause confusion.
- Fragments inside an activity have their own back stack; it's a sub-back stack if you will. But while launch modes and intent flags give you a lot of flexibility with managing the "real" back stack, the fragment manager's back stack inside a single activity is harder to manage and (even) more opaque. You may not notice this at the beginning, but later when refining your app's back button user experience, this may cause difficulties.
- Fragments do a lot of magic when recreating themselves (e.g. when a user returns to your app after a longer time). This magic is probably helpful once you're used to everything, but at the beginning where you're just trying to understand this whole new way of developing apps, it's much easier to have things be explicit. If a view hierarchy is restored automatically that sounds like a great thing, but if this restoration has subtle rules in order to work correctly, and the restoration happens at unexpected times, then you'll wish there was less magic going on.
- Compared to activities, fragments feel a lot more like "normal" Java objects. This can seduce you into handling them as such – custom constructor signatures on fragments, holding references and passing them around freely, etc. But when the fragment manager suddenly *re*-instantiates your fragment and never calls the custom constructor, and somewhere in your code you hold a reference to the *previous* version of the fragment, all of that breaks down. When juggling activities on the other hand,

it's much easier to see that they are something special that you have to work with in the special Android ways, and not treat them like any random POJO.

So: I would advise to start with activites only. Once you're more familiar with the Android way of doing things, and want to improve your app's behavior on tablets, by all means have a look at fragments. But don't try to learn riding a bicycle and a unicycle at the same time.

Propagation of events

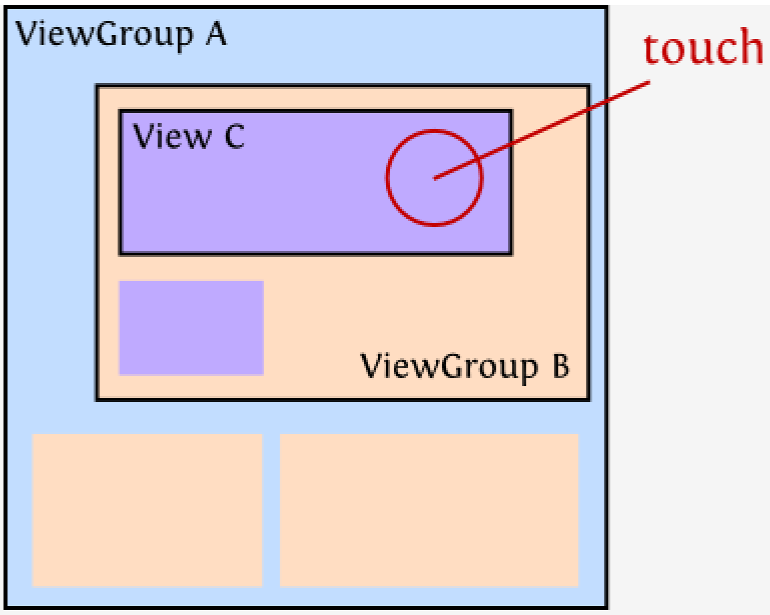
At some point you'll probably have to handle touch events yourself, and can't rely on the correct `onSomethingListener` to be available. I had to do such a thing, and what I would've loved to have is a simple post somewhere explaining how touch events propagate through the view hierarchy. Of course you need more than just the simple version eventually, but it would've been a great starting point to go from and then to fill in the details. Here's my try. It looks a bit long-ish, but that's because I'm really going step-by-step.

The assumptions

We'll only look at the most important events (actually, actions): DOWN, MOVE, UP, and CANCEL. A *gesture* is a series of events beginning with a single DOWN event when the user touches the screen, zero or more MOVE events when the user moves the finger around, and a single UP or CANCEL event, when the user releases the screen or the system tells you that the gesture is over for some other reason. When I talk about "the rest of this gesture" I mean all subsequent MOVE events and the final UP or CANCEL event.

I'm also ignoring multi-touch gestures here (we just assume a single finger) and the fact that multiple MOVE events can be grouped. Finally, we'll assume none of the views has an `onTouchListener` registered.

This is the view hierarchy that we'll be looking at, ignoring the possibility of overlapping views. There's an outer ViewGroup A, which has one or several children, including ViewGroup B. This ViewGroup B in turn has children, including a View C which is not a ViewGroup.



The point marked as "touch" is the point where the user *first* touches the screen, i.e. it's the point of the DOWN event. The user then moves the finger around and finally releases it. It does not matter if they leave the area of View C at some point; what counts is where the gesture started.

The default case

Assuming that none of our Views override the default event handling behavior, this is what happens:

- The DOWN event is passed to View C's `onTouchEvent` method. This method returns `false`, meaning "I don't care about this gesture."
- Because of this, the DOWN event is passed to ViewGroup B's `onTouchEvent` method, which doesn't care about the gesture either and thus returns `false` as well.
- Same thing: Because ViewGroup B didn't care, the event is now passed to ViewGroup A's `onTouchEvent`, which also returns `false`.

Because none of the Views cared about the event, they will not receive any events from the rest of this gesture.

Handling the event

Now let's assume that View C actually *does* care about the event, maybe because it was set to be clickable, or because you implemented a custom `onTouchEvent`.

- The DOWN event is passed to View C's `onTouchEvent` which does whatever it needs to do, and then returns `true`.
- Because View C says it's handling the gesture, the event is **not** passed do ViewGroup B's nor ViewGroup A's `onTouchEvent` methods.
- Because View C says it's handling the gesture, the events from the rest of this gesture will also be passed to View C's `onTouchEvent` method. It doesn't actually matter what the method returns from these subsequent calls, but for consistency it should probably be `true`.

onInterceptTouchEvent

We will talk about a new method now: `onInterceptTouchEvent`, which only exists on `ViewGroup` objects, not on regular views.

Before the `onTouchEvent` is called on any View, all its ancestors are first given the chance to *intercept* this event. In other words, they can steal it. We've left this out previously, so lets revisit the section "Handling the event", and include these calls:

- The DOWN event is passed to ViewGroup A's `onInterceptTouchEvent`, which returns `false`, meaning it doesn't want to intercept.
- The event is then passed to ViewGroup B's `onInterceptTouchEvent`, which doesn't want to intercept either and returns `false` as well.
- Now the DOWN event is passed to View C's `onTouchEvent` which returns `true` because it handles the event.*
- Now comes the next event of this gesture. This MOVE event is again passed to ViewGroup A's `onInterceptTouchEvent`, which again returns false. Same for ViewGroup B.
- Then it is passed to View C's `onTouchEvent`, just like in the previous section.
- The same game continues for the rest of this gesture, assuming A and B continue to return `false` from `onInterceptTouchEvent`.

Note that even though the ViewGroups returned `false` previously, the `onInterceptTouchEvent` method is still called for subsequent events. In that sense, the method's behavior is different from `onTouchEvent`.

* If View C's `onTouchEvent` returned `false` here, things would continue like in the "default case" section – even though the ViewGroups said they didn't want to intercept, they would still receive the event in `onTouchEvent` because the child didn't handle it.

Intercepting the event

Now let's take it one step further and say that ViewGroup B does *not* intercept the DOWN event, but it *does* intercept the next MOVE event. A reason for this could be that ViewGroup B is a scrolling view. If the user just taps inside it, whatever element is being clicked should be able to handle it. But once the user moves the finger a certain distance, it's not a click anymore – they clearly want to scroll. And that's why the ViewGroup takes over the gesture. Here's the order of things:

- The DOWN event is passed to ViewGroup A's `onInterceptTouchEvent`, which returns `false`.
- The event is then passed to ViewGroup B's `onInterceptTouchEvent`, which doesn't want to intercept *yet* and returns `false` as well.
- View C's `onTouchEvent` returns `true`.
- In comes the MOVE event. ViewGroup A's `onInterceptTouchEvent` still returns `false`.
- ViewGroup B's `onInterceptTouchEvent` receives the event, and notices that the user has moved the finger beyond a certain threshold (or "slop" in Android parlance). Thus it decides to take over this gesture by returning true.

- The event is changed from a MOVE event to a CANCEL event, and this changed event is given to View C's `onTouchEvent`.
- Now comes the next MOVE event. It's passed to ViewGroup A's `onInterceptTouchEvent` which continues to not care and returns `false`.
- The event is **not** passed to ViewGroup B's `onInterceptTouchEvent`. As soon as you return `true` from this method once, it won't be called again. Instead, the event is passed to ViewGroup B's `onTouchEvent`, as will the remaining events of this gesture (unless ViewGroup A decides to intercept after all).
- View C will **not** receive any more events from this gesture.

A little thing that may be surprising is the following:

1. If a ViewGroup intercepts the initial DOWN event, this event will also be passed to the ViewGroup's `onTouchEvent`.
2. On the other hand, if the ViewGroup only intercepts a later (say, MOVE) event, this event will be changed to a CANCEL event and passed to the child that handled this gesture previously, but will *not* be passed (neither as a MOVE nor as a CANCEL event) to the ViewGroup's `onTouchEvent`. Only the then-next event will end up there.

One to keep in mind.

From here on you can go even further. There is, for example, the mouthful-method requestDisallowInterceptTouchEvent with which View C could have prevented ViewGroup B from stealing the event. And if you really want to go crazy, you can override `dispatchTouchEvent` on your own `ViewGroup` implementations and do whatever you want to with the events that come in. You may be breaking a few expectations though, so be careful.

Finally, there's of course more than just touch events; who knows if your app may one day run on a device that doesn't even have a touch screen? But this is another story and shall be told another time.

In closing

Alright, that's it. Since this is just a list of a few things, there's no real conclusion here; let me just say that I've really come to enjoy Android development. It's a lot of fun once you get a general feeling of how things go. And I'm certainly not done learning yet.

---

previous post: Plain text considered harmful: A cross-domain exploit

next post: Catastrophic backtracking: When regular expressions explode