`man android`:
dingen die je in ieder geval moet weten

Nardi Lam

juni 2014

1 Applicaties starten

Android-applicaties voer je niet uit, ze worden voor je uitgevoerd. In tegenstelling tot een "normaal" OS willen mobiele besturingssystemen graag zoveel mogelijk controle houden over de applicaties die op het apparaat worden uitgevoerd, om een bepaalde gebruikerservaring te kunnen garanderen. Hoe erg dat in de praktijk lukt is nog maar de vraag, maar dat is wel de reden waarom de structuur van een Android-applicatie (waarschijnlijk) erg anders in elkaar zit dan de meeste programma's die je tot nu toe geschreven hebt.

1.1 Waar te beginnen...

In een Android-applicatie is geen `public static void main()` te vinden die je gewend bent te zien in Java-programma's. Dit kan verwarrend zijn, omdat je dus niet per se stap voor stap van begin tot eind je programma kan doorlopen en precies kan zeggen welke instructies worden uitgevoerd. In plaats daarvan definiëer je klassen met methodes die op bepaalde momenten door het besturingssysteem worden aangeroepen, waarbinnen je je code moet plaatsen. Hierdoor kan het soms lastig zijn om erachter te komen waar je moet zijn om bepaalde dingen te bereiken, en kunnen sommige acties onverwachte resultaten opleveren omdat je code éigenlijk op een andere manier (qua moment, volgorde, en context) wordt uitgevoerd dan je in eerste instantie dacht.

In plaats van een `main`-methode definiëer je in Android een `Activity`-object. Zodra je het icoontje van jouw applicatie aantikt, wordt deze voor je "opgestart". In deze zin ligt een `Activity` binnen Android dus het dichtst bij een proces. Een applicatie kan meerdere `Activity`s hebben die voor verschillende delen van je applicatie, maar dit is maar zelden echt handig, aangezien `Activity`s bijna volledig los van elkaar staan en communicatie ertussen dus vrij lastig is. Daarom is het het makkelijkst om je `Activity` en je applicatie als hetzelfde ding te beschouwen. Alle applicaties die draaien op je Android-systeem bestaan uit `Activity`s, dus als je met een andere applicatie wilt communiceren (bijvoorbeeld een file-browser wilt gebruiken om een bestand te selecteren) doe je dat door de bijbehorende `Activity` vanuit de jouwe op te starten. (Dit gaat door middel van een `Intent`, zie hiervoor de documentatie van `startActivity()` en `startActivityForResult()`.)

Als je een nieuw Android-project aanmaakt, zul je standaard een lege `Activity` (om precies te zijn, een klasse die de `Activity`-klasse extendt) voorgeschoteld krijgen, die voor je project als opstart-`Activity` ingesteld staat. Als je hem nu uitvoert, wordt deze `Activity` dus opgestart. Maar waar kan je nu beginnen met je programma schrijven?

1.2 Volgorde? Welke volgorde?

De `Activity`-klasse heeft een aantal methodes gedefiniëerd die op bepaalde momenten in de "levensloop" van je programma uitgevoerd worden. Bij het opstarten wordt bijvoorbeeld de `onCreate`-methode aangeroepen, bij het afsluiten `onDestroy()`, en tussendoor nog anderen zoals `onPause()` en `onResume()` (een volledig overzicht is te vinden op `http://developer.android.com/training/basics/activity-lifecycle/starting.html`). De bedoeling is dat je deze methodes in jouw klasse override, en aan de implementatie ervan je eigen code toevoegt. Omdat `onCreate()` in je `Activity` als eerste aangeroepen wordt in je programma, zou je dit dus kunnen zien als je `main`-methode. Echter, deze dient alleen om je applicatie klaar te zetten voor gebruik; het is dan ook de bedoeling dat je zo snel mogelijk weer uit deze methode returnt zodat het besturingssysteem weer verder kan. Als je lange berekeningen in deze methode doet, of bijvoorbeeld iets van het internet ophaalt, zal het ondertussen voor het besturingssysteem lijken alsof je applicatie vastloopt. (Dit moet je dus ook op een andere manier doen, later meer daarover.)

Het is dus nodig om je programma op een andere manier te structureren dan je gewend bent. In plaats van een boel regels code die achter elkaar uitgevoerd worden waarna het programma eindigt, moet je je programma opdelen in een aantal methodes die bij bepaalde gebeurtenissen uitgevoerd worden. Deze programmastructuur wordt *event-based programming* genoemd; het besturingsysteem houdt continu in de gaten wat er allemaal gebeurt (of de gebruiker je programma opstart, of ze op een knop drukken, of ze naar een andere applicatie wisselen of hem misschien wel helemaal afsluiten), en roept in die gevallen jouw code aan. Over wanneer dat precies gebeurt (waar in de tijd, of voor of na dit andere stukje code) heb je dus echter geen controle. Daarom is het erg belangrijk om goed de documentatie te lezen, zodat je zo goed mogelijk weet wat wanneer uitgevoerd wordt.

2 Programmastructuur

2.1 Views

Als je eenmaal een `Activity` draaiende hebt, wil je natuurlijk functionaliteit gaan toevoegen aan je programma. De inhoud van een `Activity` is gestructureerd via een hiërarchie van `Views`. Dit zijn alle elementen die een visuele representatie hebben. Je kan deze klasse zelf helemaal customizen om ieder mogelijk uiterlijk te hebben, en er zijn net als op de `Activity` bepaalde methodes gedefiniëerd die je kan overriden en dan op bepaalde momenten aangeroepen worden. Zo is er een `onTouchEvent`-methode, die aangeroepen word wanneer iemand het deel van het scherm aanraakt waar de `View` zich bevindt. Deze zou je dus bijvoorbeeld kunnen gebruiken om een `View` als een knop te laten functioneren, dan plaats je de code die bij een druk op de knop uitgevoerd moet worden in `onTouchEvent()`.

Echter kan je in dit geval beter de ingebouwde `Button`-klasse gebruiken. Dit is een `View`-kind dat standaard het uiterlijk van een knop heeft, de mogelijkheid bied om een tekst te tonen, en waar een *listener* aan toegekend kan worden die een bepaalde actie uitvoert zodra je op de knop drukt. Dat is namelijk de tweede manier om code uit te laten voeren: voorgedefinieerde elementen hebben vaak listeners voor bepaalde acties. Daarbij override je geen methode op je `View`, maar plaats je je code in een specifiek listener-object en geef je dat er aan door. De `Button`-klasse kent bijvoorbeeld de methode `setOnClickListener()`, waar je een listener aan kan meegeven die dan uitgevoerd wordt wanneer je op de knop drukt. Ieder object wat de benodigde interface implementeert (bij `setOnClickListener()` bijvoorbeeld `View.OnClickListener`), kan als listener gebruikt worden;

wat je soms ook ziet is dat een custom `View`-klasse als zijn eigen listener wordt gebruikt, om het aantal benodigde objecten te minimaliseren.

Er zijn veel standaardviews die je kan gebruiken, en het is over het algemeen een goed idee om een van deze gebruik te maken, mogelijk met wat aanpassingen als bijvoorbeeld het standaard Android-uiterlijk je niet aanstaat. Bij de wat complexere `View`s (zoals de `ListView`, die een aantal items in een lijstje kan weergeven) kan dit echter wat lastiger zijn, omdat ze je dan dwingen op een bepaalde manier te werken (je móet bijvoorbeeld Android's resource-systeem gebruiken, of bepaalde dingen moeten in XML gedefiniëerd zijn (daarover later meer)). In dit geval is het het soms toch wel waard om je eigen alternatief in elkaar te zetten (`ListView` bijvoorbeeld is het het echt niet waard).

2.2 Layouts

Naast `View`-klassen die een bepaald grafisch element voorstellen heb je ook *layouts*: `View`s die dienen om andere views goed op het scherm te positioneren. Deze stammen af van de klasse `ViewGroup` (die op zijn beurt weer een `View` is), en delen allemaal de functionaliteit dat je een `View` aan ze toe kan voegen om in de layout geplaatst te worden. Een `Activity` heeft één hoofd-`View` die over het hele scherm weergegeven wordt, deze specificeer je via de `setContentView`-methode. Dit betekent dus dat om meer dan 1 `View` te laten zien of om hem niet over het volledige scherm uit te rekken, je je `View`s in een `ViewGroup` moet plaatsen.

Er zijn, net zoals `View`s, veel meegeleverde `ViewGroup`s. Ieder positioneren ze hun kind-`View`s op een andere manier: De `LinearLayout` plaatst alles horizontaal of verticaal naast elkaar, de `RelativeLayout` laat je `View`s relatief aan elkaar plaatsen, en dan zijn er nog enkele waarvan je eigenlijk vrij weinig reden hebt om ze te gebruiken. Om te specificeren hoe je `View` precies gelayout moet worden, heeft iedere layout z'n eigen versie van het `LayoutParams`-object. Hierin kan je afhankelijk van het type layout specificeren hoe je `View` weergegeven moet worden. Bij de `LinearLayout` bevat deze bijvoorbeeld alleen dingen als grootte, maar bij de `RelativeLayout` kan je hierin ook bijvoorbeeld aangeven dat je deze `View` graag rechtsboven `View` 3 wil hebben. Om ingewikkeldere layouts te maken, kan je bijvoorbeeld ook `ViewGroup`s in andere `ViewGroup`s nesten.

Maar let wel op: de eigenschappen van iedere `View` en layout qua grootte en dergelijke kunnen erg verschillen. Soms kunnen de standaard `LayoutParams` onverwachte dingen bevatten of kunnen bepaalde instellingen in de `LayoutParams` niet werken zoals je verwacht. Dit kan tot heel vreemde situaties leiden. Onthoud dat de `LayoutParams` van een `View` bedoeld zijn voor de layout waar deze `View` inzit, en dus afhankelijk zijn van het type van de buitenste layout (ook als je een layout in een andere layout stopt). Lees goed de documentatie, bepaalde dingen kunnen bij verschillende layouts andere betekenissen hebben.

2.3 Programmeren of definiëren?

Om je `View`-structuur vast te leggen zijn er in Android twee manieren. De eerste, en degene die duidelijk sterk aangeraden wordt door het framework (te zien aan hoe het is opgezet), is door je layouts (en eventuele custom `View`s) te definiëren in XML. Dit biedt weliswaar een simpele syntax om snel een layout op het scherm te krijgen, maar mist wel het dynamische aspect, omdat je niet bijvoorbeeld aan de hand van een variabele een tekst ergens kan aanpassen, dit zal alsnog ergens in code moeten gebeuren, wat soms nog vrij lastig kan zijn. Hiervoor moet je eerst de XML-layout omzetten een Java-object via een `LayoutInflater`, en daarna (na in XML `android:id`'s

toe te kennen aan je views) via `findViewById()` de afzonderlijke views hieruit ophalen om hun eigenschappen aan te passen.

Een andere oplossing is om de XML gewoon te vergeten en zelf in je Java-code je hele `View`-structuur op te bouwen, door ze gewoon aan te maken met `new EenOfAndereView(...)` en je hoofd-`View` uiteindelijk aan te geven via `setContentView()`. Het voordeel hiervan is dat je meer controle houdt over de structuur van je applicatie, en je nooit krijgt dat je rare sprongetjes moet gaan uithalen om een dynamische waarde op een of andere manier toch in een uit XML-geladen `View` te krijgen. Wat wel een nadeel is is dat sommige dingen in code vervelend tot heel lastig te doen zijn, omdat toch veel API's erop rekenen dat je XML-layouts gebruikt. Je zult dan ook best af en toe voor dingen een hulpfunctie schrijven om iets wat eigenlijk ingebouwd zou moeten zijn te doen, en je zal merken dat sommige widgets en dergelijke in de standaardlibrary nauwelijks tot niet bruikbaar zijn als je je `View`s niet op hun manier definieert.

Naar mijn mening is de beste methode een combinatie van de twee, met een voorkeur voor de Java-manier. Het is een stuk simpeler om je layouts te definiëren in XML, maar te veel gebruik hiervan leidt tot een boel nutteloze code om je data daarin te krijgen. Je moet dus afwegen voor ieder deel wat het handigst is: als je layout dusdanig visueel ingewikkeld is met veel layoutparameters en dergelijke is het fijner om deze eerst te definiëren in XML en vervolgens hem daaruit te laden en je data erin te zetten. Als de layout echter niet al te veel onderdelen bevat maar wel veel dynamische gegevens erin geplaatst moeten worden, is het vaak minder werk om hem gewoon volledig in code te programmeren. Daarnaast heb je in je programmacode ook nog de mogelijkheid om zelf de structuur te bepalen: als je merkt dat je bepaalde dingen vaak herhaalt kan je deze in een methode plaatsen om de code wat handzamer te maken.

3 Wat je verder nog zal doen

3.1 Concurrency

Zoals eerder behandeld werkt een Android-applicatie via *events*: het besturingssysteem houdt bij wat er allemaal gebeurt en bij bepaalde gebeurtenissen voert het bepaalde stukken code uit. Dit gaat echter wel allemaal op volgorde: nieuwe gebeurtenissen (bijvoorbeeld het drukken op een knop) worden in een queue gezet, voor in het geval de applicatie nog met iets anders bezig is (bijvoorbeeld het updaten van de layout). Zodra dat afgerond is, komt de volgende dan bovenaan te staan en wordt de bijbehorende actie uitgevoerd. Dat betekent dus wel dat je in het ontwerpen van je *event-handlers* hier rekening mee moet houden: de rest van de applicatie wacht tot jij klaar bent. Als je dus bijvoorbeeld een groot bestand wil inladen en er een lastige berekening op wilt doen, of iets van het internet wil downloaden om te laten zien, en je stopt de code daarvoor in de `OnClickListener` van een `Button`, blijft je hele applicatie hangen zodra je op die knop drukt, totdat je langdurige taak afgelopen is. Als het lang genoeg duurt krijg je hierdoor ook errors dat je applicatie niet reageert (want hij is bezig met jouw code), en dit wil je natuurlijk vermijden.

De oplossing daarvoor is de code die te veel tijd in beslag neemt in een andere *thread* te draaien. Op die manier kan je een methode parallel aan alle andere code uitvoeren, waardoor je de rest van de applicatie niet ophoudt. Op zich is dit niet lastig, je kunt een `Thread`-object aanmaken met een bepaalde methode en klaar. Maar er zijn daarnaast een heleboel dingen waar je rekening mee moet houden zodra je dingen parallel aan elkaar gaat draaien. In principe wil je nooit variabelen vanaf twee verschillende threads gebruiken, want dit kan voor erg vreemde en vooral totaal onvoorspelbare situaties zorgen. In Android hebben ze daarom in ieder geval de regel opgesteld dat je niets aan

de UI mag veranderen vanaf een andere thread dan de hoofdthread. Om die reden moet je code die nieuwe `Views` toevoegt of iets dergelijks altijd laten uitvoeren op de *UI thread*. Hiervoor is een methode op `Activity`-objecten aanwezig: `runOnUiThread()`, die wel ongeveer doet wat je zou verwachten.

Omdat je vaak aan het einde van je langdurige berekening een resultaat op het scherm wil laten zien, hebben dit soort taken meestal een soortgelijke opbouw:

1. Set-up voor tijdens de uitvoering van de taak (laadicoontje weergeven, knop uitzetten zodat de taak niet meerdere keren tegelijk uitgevoerd kan worden)
2. Starten van een nieuwe `Thread`
3. (binnen de `Thread`) Uitvoeren van de taak
4. (binnen de `Thread`) Code op de UI-thread uitvoeren
5. (binnen `runOnUiThread()`) Aanpassen van de viewstructuur aan de hand van het resultaat

Je kunt ook een hulpklasse als `AsyncTask` gebruiken om dit voor elkaar te krijgen, wat vooral handig is als je tussentijds ook de UI wilt aanpassen. Het belangrijkst is hoe dan ook dat je de dingen die je op andere threads uitvoert zoveel mogelijk gescheiden van de rest van de code houdt. Op die manier kunnen er zo min mogelijk dingen misgaan (en als het misgaat, gaat het ook *erg* mis).

3.2 Low-level graphics

Tot nu toe hebben we het alleen gehad over het samenvoegen van `Views` om het uiterlijk van je applicatie bij elkaar te krijgen. Wat nou als je dingen wil laten zien die niet uit bestaande `Views` op te bouwen zijn, of gewoon meer vrijheid wil hebben in wat er precies (op pixelniveau) op het scherm weergegeven wordt? Dat kan: maak hiervoor een subklasse van `View` en override de `onDraw`-methode. Deze wordt aangeroepen wanneer het uiterlijk van de view opnieuw getekend moet/mag worden. Je krijgt hiervoor een `Canvas`-object als parameter. Deze laat je het oppervlak van de `View` als canvas gebruiken, en heeft allerlei methodes om dingen daarop te tekenen. Je parameters waarmee je hierbij tekent (lijndikte, kleur, en ook bijvoorbeeld dingen als tekstgrootte als je tekst tekent) geef je aan in `Paint`-objecten.

Er zijn wel een paar dingen waarmee je hier rekening mee moet houden. Ten eerste teken je nu dingen op een pixel-oppervlak, waarbij je zelf moet beslissen hoe groot dingen zijn. Zorg er daarom van buitenaf voor dat de `View` waarop je tekent een gewenste grootte heeft voor het huidige apparaat, en maak gebruik van de grootte van je *View* (die is nauwkeuriger dan die van je canvas) om de grootte van de dingen die je tekent te bepalen.

Daarnaast is dit een methode die, in tegenstelling tot andere events als `onTouchEvent()`, *erg* vaak kan worden aangeroepen, eens in de zoveel milliseconden. Als je hier goed gebruik van wil maken, om bijvoorbeeld een geanimeerd iets te laten zien (zonder dat het hapert), moet je zo min mogelijk dingen binnen deze methode doen. Verplaats berekeningen die eigenlijk maar één keer gedaan te hoeven worden naar een andere plek, en let ook vooral op met het aanmaken van objecten: dit kan grote vertragingen opleveren. Dit komt omdat in dat geval je na de methode een aantal keer uitgevoerd te hebben met een heleboel objecten zit, op welk moment je applicatie even stopgezet wordt om geheugen vrij te maken. Dit veroorzaakt veel kleine pauzes, waardoor je

applicatie niet soepel loopt. Je hebt bijvoorbeeld vaak veel `Paint`-objecten nodig: maak deze niet iedere keer opnieuw aan, maar doe dat één keer, bijvoorbeeld in je constructor. Ook als er iedere keer verschillende parameters nodig zijn, verander dan je eigenschappen van het bestaande object in plaats van dat je een nieuwe aanmaakt. Het is dus een goede richtlijn om ervoor te zorgen dat er zo min mogelijk, maar het liefst nooit `new` in je `onDraw`-methode staat.

4 Valkuilen

Er zijn nog een aantal algemene dingen waar je mogelijk vroeg of laat tegenaan zult lopen. Hier zijn een paar daarvan uitgelicht en hoe je deze moet oplossen:

1. **Problemen bij draaien van het apparaat**

   Hoewel het layout en rendering-proces van Android vrij ingewikkeld is, is er in sommige gevallen voor een ietwat vreemde oplossing gekozen om alles op een rijtje te krijgen. Zodra er namelijk een *configuration change* optreedt, wordt (normaal gezien) voor de zekerheid gewoon je hele `Activity` afgesloten en opnieuw opgestart. Dit klinkt wel oke, maar een belangrijk voorbeeld hiervan is het kantelen van je toestel. Zodra dat gebeurt worden dus niet al je `Views` alleen geresized, maar wordt je draaiende `Activity` via `onDestroy()` volledig afgebroken en wordt er een nieuw exemplaar (weer via `onCreate()`) aangemaakt. Er zijn wel enkele methodes om informatie vast te houden zodra dit gebeurt, zoek daarvoor naar info over `onSaveInstanceState()` en `onRetainNonConfigurationInstance()` (of gebruik liever een `Fragment` met `setRetainInstance()`, maar dat is weer een heel ander verhaal...). Maar de makkelijkste optie is vaak, als het afbreken van je `Activity` voor jouw applicatie in principe niet nodig is, om aan te geven dat je het zelf afhandelt. Dit kan door aan de `<activity>`-node in je `AndroidManifest.xml` de attribuut `android:configChanges="orientation|screenSize"` toe te voegen. Daarna kun je in `onConfigurationChanged()` zelf nog mogelijke code plaatsen om je `Views` correct aan te passen, als dat nodig is. Een voorbeeld hiervan is te vinden in de *Hello World*-applicatie op Blackboard.

2. **Onverwachte waarden van eigenschappen van `Views`**

   Veel `Views` zijn nog niet af zodra je ze aangemaakt hebt (met `new` of via een inflater). Vaak krijgen ze bijvoorbeeld pas echt een grootte en plek zodra alles getekend wordt, wat betekent dat de eigenschappen van een `View` onverwachte waarden kunnen hebben als je ze te snel opvraagt. Als je bijvoorbeeld twee views aanmaakt, en de tweede dezelfde hoogte wil geven als de eerste, dus je doet `getHeight()` om de goede hoogte op te halen, is de kans groot dat je een hoogte van `0` terugkrijgt. Probeer in dit geval eerst je layout anders vorm te geven. `Views` in `LinearLayouts` kunnen bijvoorbeeld gebruikmaken van de `weight`-parameter om twee `Views` beide de helft van de beschikbare ruimte te geven, zonder dat je van te voren precies hoeft te weten hoe groot die ruimte is. Als dat niet lukt, zet dan je code op een latere plek waar de afmetingen wel bekend zijn (in iets als `onMeasure()` bijvoorbeeld), of schrijf als het echt niet anders kan je eigen layout waarin je gewoon zelf de afmetingen van je kind-`Views` kan berekenen en aanpassen.