100x ANDROID

PROGRAMOVÁNÍ MOBILNÍCH APLIKACÍ

Luděk Hrušák

#JAVA

Autor vystudoval Fakultu aplikovaných věd a Ekonomickou fakultu Západočeské univerzity.

Zabývá se vývojem software pro korporátní sféru - energetiku, telekomunikace a bankovnictví.

100x Android

PROGRAMOVÁNÍ MOBILNÍCH APLIKACÍ, JAVA

ISBN 978-80-11-08354-0
© 2026 Luděk Hrušák, Plzeň, 1. vydání, ver. 1.01
Autorské dílo podléhá licenci Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (BY-NC-ND 4.0 CZ)
https://creativecommons.org/licenses/by-nc-nd/4.0/

Obsah

1 Úvod....10
2 Stručná historie Androidu....11
3 Předpoklady aneb co budeme potřebovat....13
3.1 Zdrojové soubory....13
4 Instalace vývojového prostředí....15
5 První pokusná aplikace....16
5.1 Vývojové prostředí....19
5.2 Hello World!....21
5.3 Vytvoření emulátoru....21
5.4 Připojení fyzického zařízení....23
6 První opravdová aplikace....26
6.1 Java nebo Kotlin?....26
6.2 Jdeme na to....27
7 Životní cyklus aktivity....35
8 Context (kontext)....40
9 Struktura projektu v Android Studiu....41
9.1 Manifest....41
9.2 Oprávnění....42
9.3 Požadované funkce....45
9.4 Java....45
9.5 Res....46
9.6 Barvy....46
9.6.1 Selector....47
9.7 XML obrázky....48
9.7.1 XML obrázek, 1. příklad....48
9.7.2 XML obrázek, 2. příklad....51
10 Ladění aplikace, logování....53
10.1 Ladění aplikace, Stack Trace....54
11 Závislé zdroje – Dependencies....55
12 Prvky formuláře....58
12.1 Textová pole....58
12.2 Tlačítka....61
12.3 Images, obrázky....63
12.4 Plovoucí tlačítko....64

12.5 Záhlaví formuláře....66
13 Rozvržení prvků na formuláři....69
13.1 Linear Layout....69
13.2 Constraint Layout....73
13.2.1 Constraint Layout, 1. příklad....77
13.2.2 Constraint Layout, 2. příklad....79
13.3 Table Layout....83
13.4 Frame Layout....87
13.5 Vkládání layoutů....89
13.5.1 Vkládání layoutů, 1. příklad....89
13.5.2 Vkládání layoutů, 2. příklad....93
13.6 Konverze layoutů....95
14 Kontejnery....98
14.1 Radio Group....98
14.2 Recycler View....100
14.2.1 Recycler View, 1. příklad....101
14.2.2 Recycler View, 2. příklad....108
14.2.3 Recycler View, 3. příklad....112
14.2.4 Recycler View, 4. příklad....114
14.2.5 Recycler View, 5. příklad....118
14.3 Scroll View....126
14.4 Horizontal Scroll View....130
14.5 Vnořený Scroll View (Nested Scroll View)....133
14.6 Fragmenty....136
14.7 View Pager....146
14.7.1 View Pager, 1. příklad....146
14.7.2 View Pager, 2. příklad....150
14.8 Tab Layout....156
14.8.1 Tab Layout, příklad....157
14.9 Search View....163
14.10 Web View....170
15 Legacy prvky....174
15.1 Relative Layout....174
15.2 Grid Layout....177
15.2.1 Grid Layout, 1. příklad....178
15.2.2 Grid Layout, 2. příklad....180

15.3 List View....183
15.3.1 List View, 1. příklad....184
15.3.2 List View, 2. příklad....188
15.4 Expandable List View....192
15.5 Grid View....200
15.5.1 Grid View, 1. příklad....201
15.5.2 Grid View, 2. příklad....205
15.6 Tab Host....210
15.6.1 Tab Host, příklad....211
16 Dialogová okna....216
16.1 Toast....216
16.2 Snackbar....217
16.3 Alert dialog....219
16.4 Oznámení....229
17 Menu....232
17.1 Menu voleb....232
17.2 Vyskakovací menu....234
17.3 Kontextové menu....236
17.4 Vysouvací menu....238
17.4.1 Vysouvací menu, 1. příklad....238
17.4.2 Vysouvací menu, 2. příklad....242
17.4.3 Vysouvací menu, 3. příklad....244
18 Ukládání dat....249
18.1 Sdílená nastavení, Shared Preferences....250
18.2 Vnitřní úložiště, Internal Storage....252
18.3 Externí úložiště, External Storage....256
18.4 Veřejně přístupné adresáře, Public Folders....260
18.5 Dočasné soubory, Cache....262
18.6 Soubory projektu, res, assets....264
18.7 Databáze....266
18.7.1 SQL syntaxe....267
18.7.2 Metody třídy SQLiteDatabase....270
18.7.3 SQLite, 1. příklad....272
18.7.4 SQLite, 2. příklad....276
19 Média....284
19.1 Zvuk....284

19.1.1 Zvuk, 1. příklad....284
19.1.2 Zvuk, 2. příklad....289
19.2 Video....292
19.2.1 Video, 1. příklad....293
19.2.2 Video, 2. příklad....297
19.2.3 Video, 3. příklad....303
20 Připojení k internetu....309
20.1 XML nebo JSON?....309
20.2 Připojení....313
20.3 Připojení k internetu, 1. příklad....314
20.4 Připojení k internetu, 2. příklad....319
21 Mapy....326
21.1 Mapy – registrace....326
21.2 Mapy – první zobrazení....330
21.3 Mapy – komponenta MapView....335
21.4 Mapy – značky....337
21.5 Kreslení do map....341
22 Systémová data....345
22.1 Stav zařízení....345
22.1.1 Stav baterie....345
22.1.2 Datové služby....347
22.1.3 Velikost paměti a datového úložiště....350
22.2 Interní databáze....353
22.2.1 Kalendář, upozornění....355
22.2.2 Telefonní seznam, kontakty....358
22.2.3 Textové zprávy....361
22.3 Inzertní ID (Advertising ID)....363
23 Senzory a periférie....366
23.1 Senzor intenzity osvětlení....367
23.2 Určování polohy, GPS....369
23.2.1 Určování polohy, 1. příklad....371
23.2.2 Určování polohy, 2. příklad....374
23.3 Čtečka otisku prstu....377
23.4 Blesk fotoaparátu....381
23.5 Bluetooth....384
24 Intent....390

24.1 Explicitní intent....390
24.1.1 Explicitní intent, příklad....390
24.2 Implicitní intent....393
24.2.1 Implicitní intent, 1. příklad....393
24.2.2 Implicitní intent, 2. příklad....394
24.3 Intent aplikace....395
24.3.1 Intent aplikace, příklad....395
25 Operace na pozadí....398
25.1 Vlákno (thread)....398
25.2 Časovač (timer)....400
25.3 Služba (service)....403
25.3.1 Služba, 1. příklad....406
25.3.2 Služba, 2. příklad....410
25.3.3 Služba, 3. příklad....413
26 Ikona aplikace....417
27 Lokalizace, překlad aplikace....422
28 Styly a témata....425
28.1 Styly....425
28.1.1 Styly, 1. příklad....426
28.1.2 Styly, 2. příklad....428
28.2 Témata....430
28.2.1 Témata, 1. příklad....432
28.2.2 Témata, 2. příklad....434
29 Reklamy....438
29.1 Služba AdMob....438
29.2 Testovací reklamy....445
30 Widgety....447
30.1 Widgety, 1. příklad....448
30.2 Widgety, 2. příklad....451
31 Testování aplikací....460
31.1 Unit testy....460
31.2 Instrumented testy....462
32 Několik praktických rad a postupů....464
32.1 Změna verze API....464
32.2 Distribuce zdrojových souborů....465
32.3 Přemístění projektu....466

32.4 Přejmenování projektu....466
33 Gradle....469
33.1 Konfigurace Gradle....470
33.2 APK balíček....471
34 Instalace a distribuce aplikací....473
34.1 Verze aplikace....473
34.2 Vygenerování APK balíčku....474
34.3 Instalace APK balíčku....476
34.4 Google Play Store....477
35 Základní termíny, pro lepší orientaci v textu....481

1 Úvod

Příručka je určena čtenáři, který umí pracovat s Androidem na úrovni uživatele a chce udělat první krok k tomu, aby se stal vývojářem. Tedy tím, kdo aplikace nejen používá, ale umí je také vytvářet. Cesta k profesionálnímu programování Androidu je dlouhá, náročná a trnitá. První kroky však nemusí být ani těžké, ani nudné. Příručka ukazuje na stovce jednoduchých příkladů, jak aplikaci vytvořit, jak ji dostat do telefonu nebo tabletu, postupně vylepšovat a jak vyrazit směrem k vývoji aplikací, které budou na Google Play k dispozici celému světu. Každý ze sta probíraných příkladů je prvkem skládačky, z nichž se dá složit celá aplikace.

Popisované postupy jsou navržené tak, aby se uživatel seznámil se základy rychlou a efektivní formou a aby vždy viděl výsledky dřív, než ho práce omrzí. Proto jsou zejména ze začátku probírány pouze věci nezbytné a až s nabytými zkušenostmi se text vrací k jejich nenásilnému rozšíření o pokročilejší techniky. Dobrých Android programátorů není mnoho a poptávka po nich roste, i přes nástup nástrojů umělé inteligence, přímo úměrně k množství zařízení, která tento operační systém využívají.

Zatímco původně byl Android určený pouze pro mobilní telefony a později přirozeně i pro tablety, postupně se ukázalo, že je výhodné jeho užití i v další elektronice. Dnes ho najdeme v televizorech, fotoaparátech, nositelné elektronice, autech a mnoha dalších zařízeních, jejichž množina se stále rozšiřuje. A naprostá většina z nich využívá aplikace třetích stran, což je příležitost pro softwarové vývojáře.

Android ve svých začátcích podporoval výhradně jazyk Java. Později přidal podporu jazyka Kotlin, který je modernějším ekvivalentem Javy. Pro práci na vývoji Android aplikací je potřeba ovládat alespoň základy jednoho z těchto dvou programovacích jazyků. Pokud se rozhodnete naučit programovací jazyk současně s tvorbou Android aplikací, je to možné, jen počítejte s tím, že se budete muset rozhodnout pro jeden z nich. A programování vám logicky půjde o něco pomaleji než zkušeným programátorům.

Obecně se dá říct, že vše, co naprogramujete v Javě, můžete naprogramovat i v jazyce Kotlin. Ten má navíc některé vlastnosti typické pro moderní programovací jazyky. Naproti tomu Javu lze využít mimo Android v podstatně širším spektru úloh. Příklady v knize jsou postavené na jazyku Java, nicméně jedna z podkapitol je věnovaná i aspektům volby programovacího jazyka.

Kniha si nedává za cíl naučit čtenáře psát komplexní aplikace. Záměrem je pochopení principů práce v Android Studiu, architektury aplikací, sestavení jednoduchých příkladů a vytvoření vědomostních základů pro další práci.

2 Stručná historie Androidu

Základy operačního systému Android, původně orientovaného čistě na mobilní telefony, položila v roce 2003 kalifornská společnost Android Inc., působící v Silicon Valley nedaleko amerického San Franciska. Roku 2005 potenciál malé firmy odhalil Google a jednoduše celou firmu odkoupil. O další dva roky později Google uvolnil vývojové prostředí (SDK) a vývoj předal alianci OHA (Open Handset Alliance), kterou pro tento účel sám založil. Hned následujícího roku, v září 2008, přišel na trh první telefon s operačním systémem Android verze 1.0. V té době již bylo na Android Marketu, tehdejším předchůdci Google Play Store, několik tisíc aplikací.

Od samého začátku šlo o operační systém založený na linuxovém jádře, navržený pro ovládání dotykem a gesty. Jednou z jeho hlavních vlastností už tehdy byla možnost nahrávání aplikací z Android Marketu, předchůdce dnešního Google Play.

Do Androidu byly již od jeho první veřejné verze integrovány některé služby Google, do té doby dostupné jen na počítačích – YouTube, Gmail, mapy a kalendář.

Počínaje verzí 3.0, z roku 2011, podporuje Android také tablety. Přesněji řečeno, verze 3.0 byla určena pouze pro tablety, resp. zařízení s velkou úhlopříčkou displeje. Telefony dostaly téhož roku verzi 4.0, ve které se obě vývojové větve zase spojily a dále už je operační systém společný pro obě kategorie.

V roce 2014 Google přišel hned s několika významnými novinkami. Představil Android TV – modifikaci Androidu verze 5, upravenou nejprve pro set top boxy a později integrovanou přímo do televizních přijímačů. Android Wear (nyní Wear OS), Android pro nositelnou elektroniku, zejména hodinky a také Android Auto, koncept, který umožňuje na displeji automobilu zobrazovat informace z mobilního telefonu.

Zajímavou kapitolou v historii Androidu jsou Google Glass – brýle s kamerou a projekčním displejem pro rozšířenou realitu, bezdrátově propojené s telefonem. Vývojáři dostali tato zařízení v testovací verzi v roce 2013 pro účely vývoje prvních aplikací. Od té doby prošly brýle několika evolučními změnami. Google zkoušel jejich užití nejdřív pro veřejnost, později je ale z veřejného testování stáhl a zkoušel nasazení v průmyslu. V roce 2017 zmizela podpora z Android Studia. Jestli a v jaké podobě se případně vrátí, zatím není známo.

Na jaře 2017 se stal, po Javě, dalším oficiálním programovacím jazykem Androidu Kotlin od firmy JetBrains. Google začlenil do Android Studia nástroje pro práci s tímto jazykem a spolu s několika dalšími významnými vývojářskými společnostmi začal tento jazyk preferovat před Javou.

Starší verze Androidu označoval Google kódovými jmény a logy, inspirovanými populárními americkými sladkostmi. Využíváme toho i v této knize, verze Androidu slouží jako vstupní data pro nejednu vzorovou aplikaci. Od roku 2019, resp. verze Android 10, jsou však kódová jména oficiálně nahrazena prostým číslem. Google přesto dál, i když už jenom interně, používá tento způsob označování nových verzí.

Vývoj Androidu se vyznačuje poměrně vysokou rychlostí změn. Každý rok vzniká přinejmenším jedna verze API, obsahující nové funkčnosti a nezřídka také odstraňující ty, které mezitím zastaraly. To klade vysoké nároky na vývojáře, kteří se změnám a tempu jednoduše musí přizpůsobit.

Operační systém Android je od samého začátku vyvíjen pod „open source” licencí, což znamená, že si jeho zdrojové soubory může kdokoliv stáhnout, upravit, sestavit a volně používat. Je ale potřeba odlišit operační systém od dalších služeb, které už pod touto licencí šířené nejsou.

A operační systém sám je dnes svázaný s celou řadou takových služeb, bez nichž nedává jeho užití příliš dobrý smysl. Zejména obchod s aplikacemi Google Play Store je jeho velmi těžce postradatelnou součástí. Týká se to však také vyhledávače, map, e-mailu, cloudového úložiště a dalších.

I když by se tedy na první pohled mohlo zdát, že si s Androidem může dělat co kdo uzná za vhodné, je potřeba zohlednit potřebu licenčních práv na další služby. Díky tomu má Google kontrolu nad vydávanými verzemi Androidu a může tak ovlivňovat podmínky, za kterých ho výrobci zařízení používají. Přesto se čas od času objeví zařízení s nainstalovaným Androidem bez služeb Google. I pro ty je možné psát aplikace, je ale potřeba počítat s alternativními distribučními kanály.

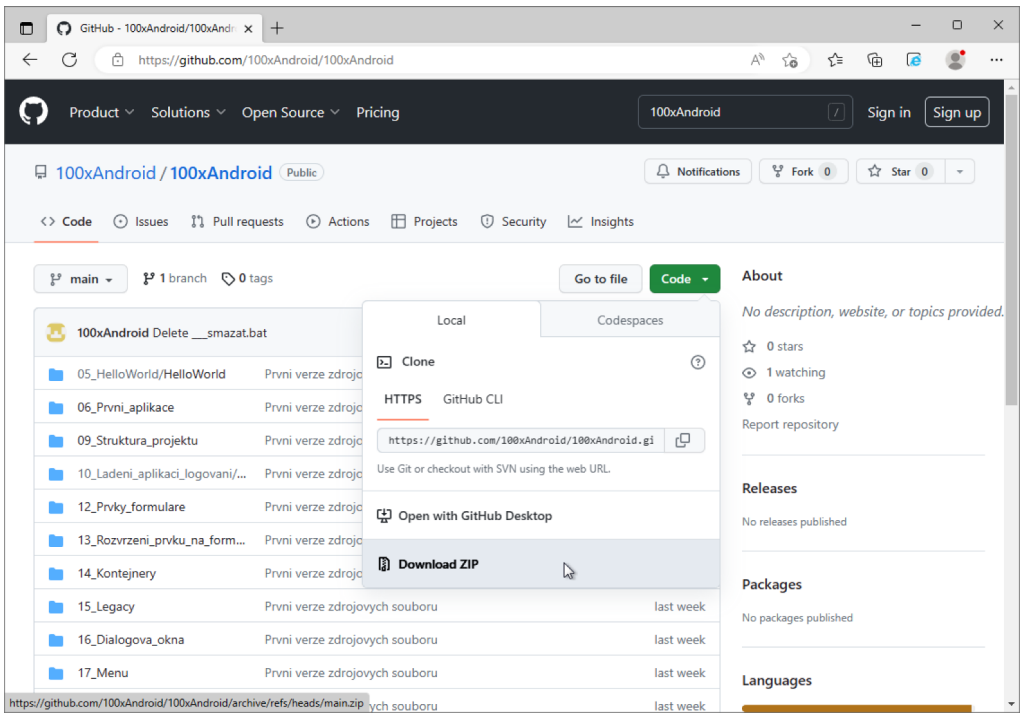
Obrázek 2.1: Loga symbolizující verze Androidu, areál Google, Mountain View, USA

3 Předpoklady aneb co budeme potřebovat

- Přiměřeně výkonný počítač (operační systém Windows, Linux nebo macOS). Pokud na něm hrajete hry, pracujete s grafickými programy nebo používáte vývojová prostředí pro Javu nebo C#, bude pravděpodobně vyhovovat i pro programování Android aplikací.
- Dostatek místa na disku. Pro potřeby Android Studia verze 2025.3.4 alespoň 10 GB, následující verze budou mít nároky pravděpodobně vyšší.
- Volný USB port na počítači a propojovací kabel. Pro zařízení bez USB portu je možné použít propojení přes Wi-Fi nebo Bluetooth.
- Zařízení, na kterém budeme testovat svoje aplikace, pro začátek bude stačit libovolný telefon nebo tablet s operačním systémem Android.
- Přístup na internet. Bude potřeba pro stažení vývojového prostředí a pro průběžné dohledávání informací při vývoji.
- Alespoň základní znalost technické angličtiny. Zatímco zdrojů informací v českém jazyce není mnoho, angličtina dokáže zpřístupnit nepřeberné množství rad, příkladů, zdrojových kódů a dokumentace.
- Hodí se ovládat základy programování. Ne že by bez nich nešlo vytvořit jednoduchou aplikaci, ale učit se programovat tímto způsobem není zrovna efektivní.
- Pochopení základních principů zápisu formátu XML. Zde platí totéž, co v předchozím bodě.
- Hodně času, trpělivosti a pevných nervů.

3.1 Zdrojové soubory

Pro všechny popisované příklady jsou volně k dispozici zdrojové soubory. Uloženy jsou na serveru GitHub, který se používá pro zveřejňování a verzování projektů. Přístupný je přes prohlížeč, na adrese https://www.github.com. Soubory lze stáhnout i bez vlastního uživatelského účtu, stačí najít repozitář „100xAndroid/Java“, kliknout na tlačítko Code a zvolit možnost Download ZIP.



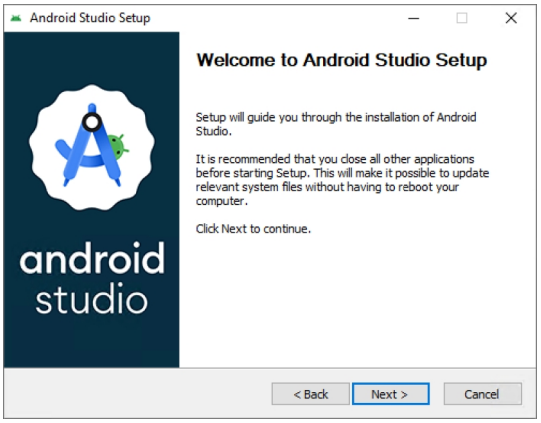
Obrázek 3.1: Stažení zdrojových souborů příklad

Takto získaný soubor obsahuje všech sto příkladů, stačí je rozbalit do vhodné složky na disku počítače a libovolný z nich otevřít v Android Studiu. Který projektový adresář patří ke kterému probíranému příkladu, je uvedeno vždy na začátku příslušné kapitoly.

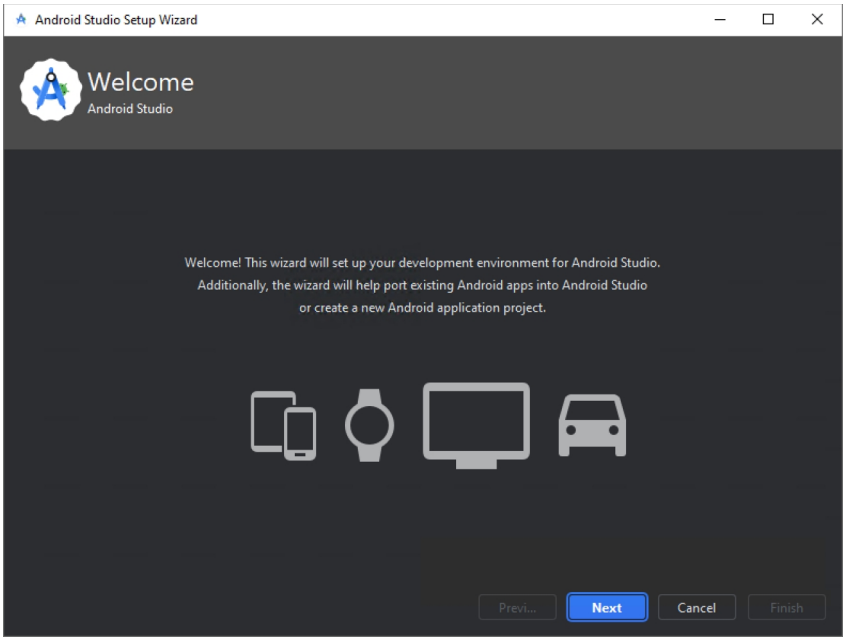
4 Instalace vývojového prostředí

Nejprve je potřeba stáhnout vývojové prostředí, tzv. IDE (Integrated Development Environment), ze stránky developer.android.com/studio. Toto prostředí je k dispozici nejen pro Windows, ale i pro OS X a Linux. IDE je postaveno na vývojovém prostředí IntelliJ IDEA od společnosti JetBrains. Prostředí se jmenuje Android Studio, poskytuje ho firma Google a je k dispozici zcela zdarma.

Instalace není nijak obtížná. Stačí spustit instalátor Android Studia, zvolit všechny komponenty které nabízí, a po odkliknutí zbývajících stránek průvodce se Studio začne instalovat. Proces chvíli potrvá. Následně se spustí konfigurátor, který se zeptá na základní nastavení. Nejjednodušší je zvolit standardní. V opačném případě je možné změnit vzhled Studia. Konfigurátor ještě stáhne a nainstaluje některé komponenty. Prostředí, komponenty a následná instalace emulátoru, ke které se ještě dostaneme, zabere zhruba 10 GB diskového prostoru (platí pro Windows verzi 2025.3.4).



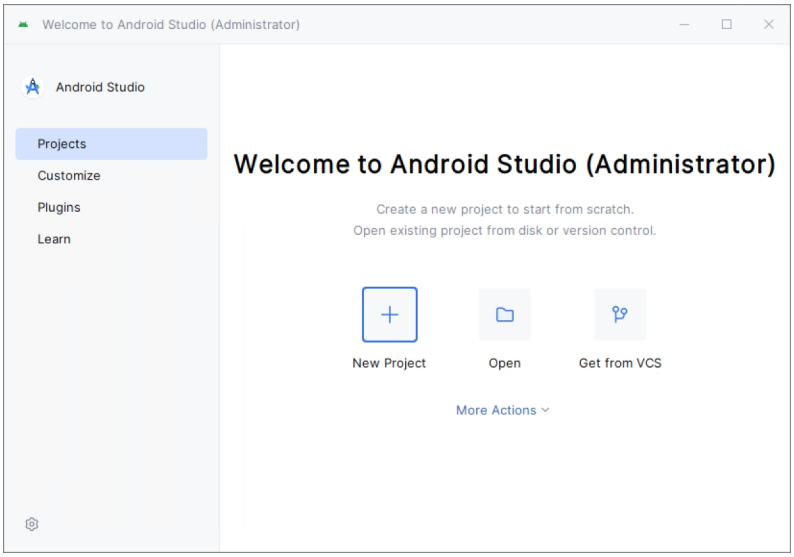
Obrázek 4.1: Instalátor Android Studia



Obrázek 4.2: Proces instalace Android Studia

5 První pokusná aplikace

Pro první aplikaci si stanovíme nepříliš vysoký cíl - vytvořit „něco“, co půjde nahrát do telefonu a tam to bude nějak primitivně reagovat. Víc toho zatím nebudeme potřebovat.



Obrázek 5.1: Spuštění Android Studia

Začneme spuštěním Android Studia. Jako první uvidíme formulář pro výběr akce (obrázek 5.1). Zvolíme 'New Project' - tj. začít práci na novém projektu v Android Studiu. Ostatní možnosti nyní přejdeme, vrátíme se k nim případně později.

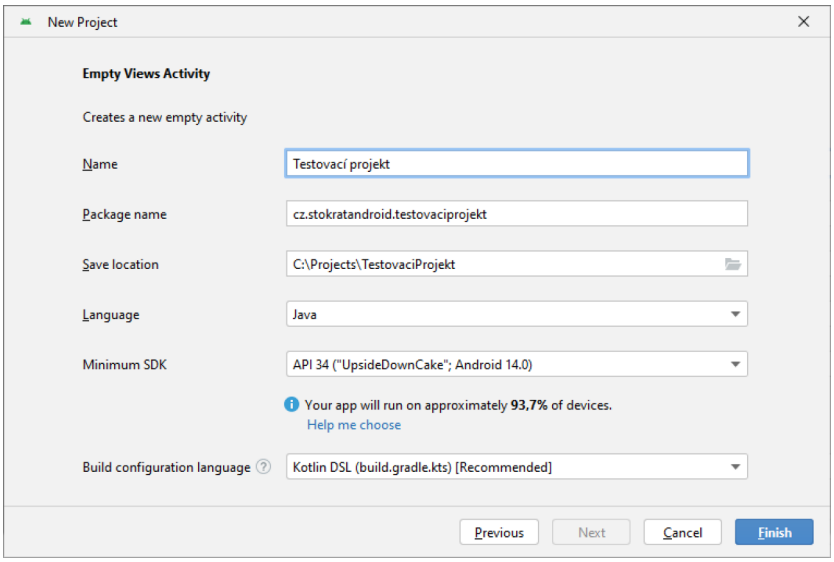
V dalším kroku průvodce nainstaluje komponenty tak, aby odpovídaly zvolenému typu zařízení, a dá nám vybrat typ aktivity.

Aktivita (Activity) je základní stavební prvek aplikace. Je to v podstatě okno či formulář, obsahující prvky grafického rozhraní, například menu, tlačítka, ikonky atd. Jinými slovy to, co uživatel vidí a s čím pracuje. Android aplikace se skládají z aktivit, mezi kterými si uživatel, často aniž by si to uvědomoval, při práci přepíná. Většina aplikací po svém spuštění zobrazí úvodní aktivitu a čeká na reakci uživatele – například zadání hesla, volbu z menu, kliknutí na ikonu nebo tlačítko atd.

Aby nebylo nutné vždy začínat u úplně prázdné aplikace a do ní vytvářet aktivity, jejichž základ je pokaždé stejný, obsahuje průvodce Android Studia generátor základních typů aktivit. Pomocí něj si zvolíme šablonu aktivity a průvodce vygeneruje její kostru. Vyhneme se tak zbytečným chybám a nebudeme ztrácet čas sestavováním aktivity, což je postup, který se jenom opakuje.

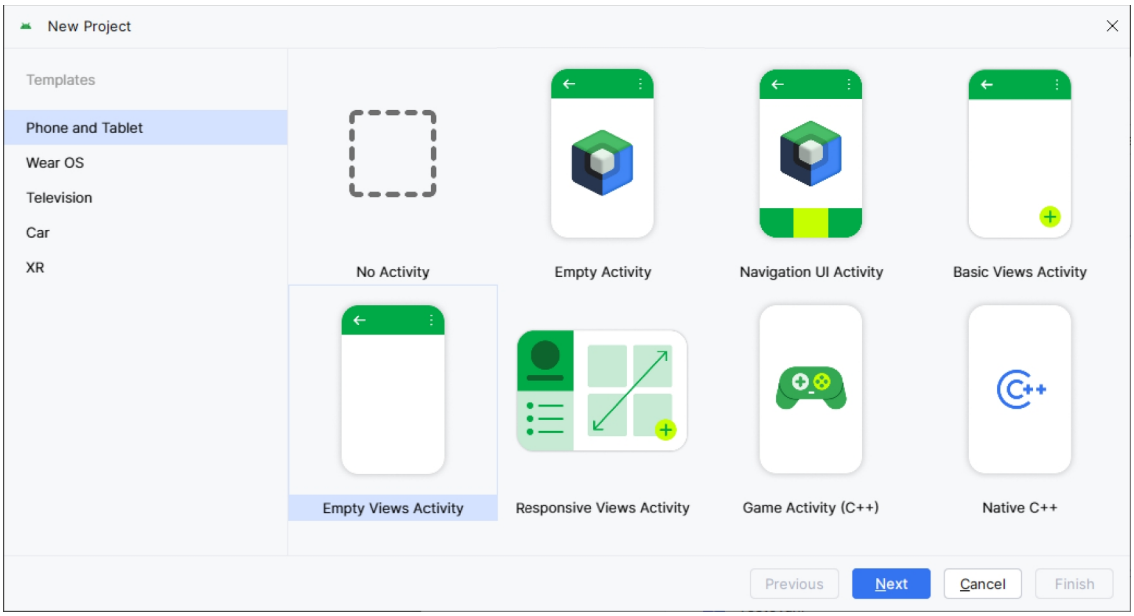
Šablony jsou rozděleny podle platforem, neboli zařízení, na kterých bude možné aplikaci spustit. Android dnes už zdaleka není pouze na mobilních telefonech, ale také v chytrých hodinkách, v televizi nebo v automobilu. Pro každou takovou skupinu zařízení existuje upravená edice Androidu. Každá edice má několik verzí, které Google rozšiřuje a upravuje. Pokud budeme aplikace testovat na telefonu, zůstaneme na záložce Phone and Tablet.

Pro první pokusy zvolíme prázdnou aktivitu, tedy Empty Views Activity. Obsahuje vše potřebné pro jednoduchou aplikaci a zároveň nás nezahltí prvky, které zatím neumíme využít.



Obrázek 5.3: Nastavení nového projektu

Tím jsme se ocitli u formuláře pro nastavení nového projektu, obrázek 5.3. Do řádku Name nastavíme vhodný název, prozatím bude vyhovovat třeba 'Testovací projekt'. Název se později



Obrázek 5.2: Šablony generátoru základních typů aktivit

objeví také u ikony aplikace.

Řádek Package name je určený k nastavení jednoznačné identifikace aplikace. To znamená, že zde musí být text, který pro tyto účely nepoužívá žádná další aplikace na světě. Formát identifikátoru je podobný jako URL adresa serveru, která se běžně používá pro identifikaci webových stránek. Pouze se z praktických důvodů píše v obráceném pořadí. Text si můžeme libovolně zvolit, je ale vhodné používat jednotnou konvenci, tedy na nejvyšší úrovni buď com nebo cz, na druhé úrovni název firmy, projektu nebo třeba jméno autora (tady už je potřeba dát pozor na

unikátnost) a na třetí úrovni název konkrétního projektu nebo jeho části. Tedy například 'cz.stokratandroid.testovaciprojekt'. Úrovní můžeme použít víc (například pokud firma nebo autor vytváří víc typů aplikací – nástroje, hry atd.), ale tři jsou vhodným minimem.

Další věcí, která nás bude zajímat, je Save location, neboli místo na disku, kam se projektové soubory budou ukládat. Můžeme nechat předvyplněné umístění, nebo si nastavit svoje, zcela dle libosti.

V řádku Language vybereme programovací jazyk aplikace, tedy Java nebo Kotlin, podle toho co nám více vyhovuje. Pro konkrétní jazyk je potřeba být rozhodnutý již při zakládání projektu, jeho změna během práce není možná. Podrobněji se k výběru programovacího jazyka dostaneme v kapitole 6.1 Java nebo Kotlin?

Pro každou verzi Androidu existuje jiné SDK (Software Development Kit - balíček vývojových nástrojů). Toto SDK je schopné pracovat s odpovídajícím API (Application Programming Interface - rozhraní pro programování aplikací) a s několika jeho předchozími verzemi. Musíme určit, kterou nejstarší verzi API má naše aplikace podporovat. Je to trochu balancování – pokud použijeme příliš staré API a tím pádem i staré SDK, nebudeme moct využívat některé vlastnosti nových verzí Androidu. Pokud naopak použijeme příliš novou verzi, nepůjde aplikace nainstalovat na stará zařízení a připravíme se o část potenciálního trhu. V nejhorším případě se může stát, že aplikace vyvinutá na příliš novém SDK nepůjde spustit ani na zařízení, které je k dispozici pro testování, protože i to už bude zastaralé. Proto je dobré také vědět, která verze Androidu a s jakým API je uvnitř testovacího zařízení.

Každé Android zařízení umožňuje uživateli zjistit číslo verze systému. Ačkoli se to u jednotlivých zařízení může mírně lišit, ve většině případů lze tuto informaci najít v menu *Nastavení → O zařízení → Verze Android*

SDK nepodporuje všechna předcházející API, ale s tím, jak vznikají nové aktualizace, od podpory těch nejstarších upouští. Není tedy již možné vytvářet s novým SDK aplikace například pro Android 1.0. A ani by to, vzhledem k omezením a k jejich zastoupení na trhu, příliš nedávalo smysl. Pokud však někdo potřebuje vytvořit aplikaci pro nepodporovanou verzi, může použít starší SDK, které je na internetu stále k dispozici.

Rok	Kódové jméno	Číslo verze	Verze API	Poznámka
2003	-			začátek vývoje, alfa verze
2007	-			beta verze, vývoj aplikací, testování, sedm aktualizací
2008	-	1.0	1	první veřejně dostupná verze Androidu
2009	-	1.1	2	
2009	Cupcake (mufin)	1.5	3	
2009	Donut (kobliha)	1.6	4	
2009	Eclair (rakvička)	2.0	5, 6	jedna aktualizace (2.0.1)
2010	Eclair (rakvička)	2.1	7	
2010	Froyo (mražený jogurt)	2.2	8	tři aktualizace (2.2.1 - 2.2.3)
2010	Gingerbread (perník)	2.3	9, 10	sedm aktualizací (2.3.1 - 2.3.7)

Rok	Kódové jméno	Číslo verze	Verze API	Poznámka
2011	Honeycomb (včelí plástev)	3.0	11	
2011	Honeycomb (včelí plástev)	3.1	12	
2011	Honeycomb (včelí plástev)	3.2	13	šest aktualizací (3.2.1 - 3.2.6)
2011	Ice Cream Sandwich (zmrzlinový sendvič)	4.0	14, 15	čtyři aktualizace (4.0.1 - 4.0.4)
2012	Jelly Bean (želé)	4.1	16	dvě aktualizace (4.1.1, 4.1.2)
2012	Jelly Bean (želé)	4.2	17	dvě aktualizace (4.2.1, 4.2.2)
2013	Jelly Bean (želé)	4.3	18	jedna aktualizace (4.3.1)
2013	KitKat (čokoládové tyčinky)	4.4	19, 20	sedm aktualizací (4.4.1 - 4.4.4, 4.4W - 4.4W.2)
2014	Lollipop (lízátko)	5.0	21	dvě aktualizace (5.0.1, 5.0.2)
2015	Lollipop (lízátko)	5.1	22	jedna aktualizace (5.1.1)
2015	Marshmallow (žužu)	6.0	23	jedna aktualizace (6.0.1)
2016	Nougat (nugát)	7.0	24	
2016	Nougat (nugát)	7.1	25	dvě aktualizace (7.1.1, 7.1.2)
2017	Oreo (čokoládové sušenky)	8.0	26	
2017	Oreo (čokoládové sušenky)	8.1	27	
2018	Pie (koláč)	9.0	28	
2019	Android 10 (Quince Tart *)	10.0	29	
2020	Android 11 (Red Velvet Cake *)	11.0	30	
2021	Android 12 (Snow Cone *)	12.0	31, 32	jedna aktualizace (12L)
2022	Android 13 (Tiramisu *)	13.0	33	
2023	Android 14 (Upside Down Cake *)	14.0	34	
2024	Android 15 (Vanilla Ice Cream *)	15.0	35	
2025	Android 16 (Beklava *)	16.0	36	
2026	Android 17 (Cinnamon Bun *)	17.0	37	
* neoficiální označení, používané pouze během vývoje				

Tabulka 5.1: Verze operačního systému Android

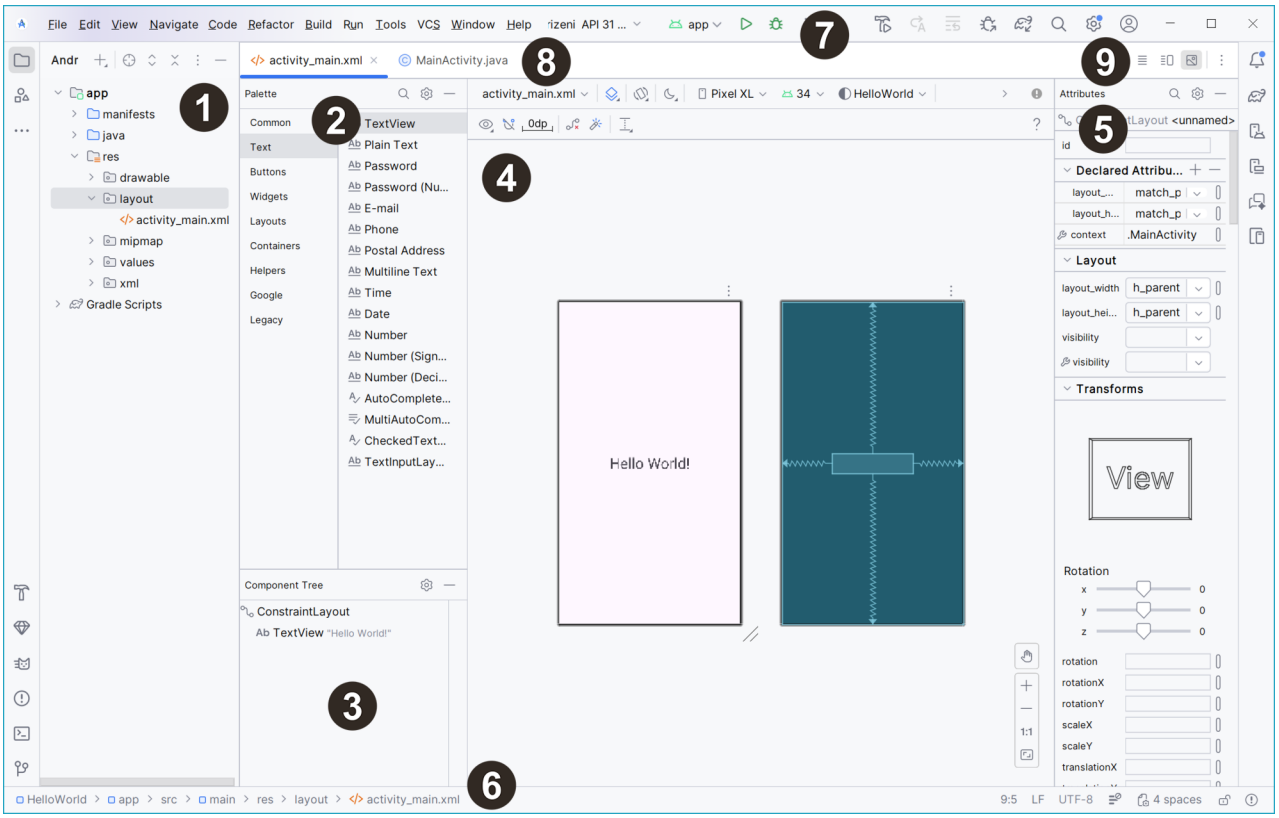
Zvolme tedy kompromis a nastavme API o několik málo verzí starší, než jaké je číslo verze v testovacím zařízení. Ve formuláři je k dispozici nápověda, kolik procent aktuálně používaných zařízení tuto verzi podporuje. V našem zájmu je mít toto číslo co nejvyšší.

Android Studio vygeneruje projekt, který obsahuje právě jednu, v průvodci nadefinovanou aktivitu.

Při prvním spuštění se může stát, že firewall operačního systému zaznamená neznámou komunikaci nové aplikace - Javy. Povolte ji alespoň pro přístup v místní síti.

5.1 Vývojové prostředí

Jakmile je průvodce ukončen, otevře se projekt ve vývojovém prostředí Android Studia.



Obrázek 5.4: Android Studio – vývojové prostředí

❶ Základním prvkem uživatelského rozhraní (GUI) Android Studia je okno projektu. Ještě se k němu později dostaneme, zatím si vystačíme s tím, že právě zde najdeme seznam všech souborů, které s aplikací souvisejí. Klikáním se lze mezi soubory přepínat, upravovat je, přidávat, mazat a tak podobně.

❷ - ❺ Editor. Toto okno mění svůj vzhled v závislosti na souboru, který je označený v okně projektu. Nás bude nyní zajímat především okno grafického editoru, které se otevře při označení souboru content_main.xml.

❷ V paletě komponent (Palette) jsou zobrazeny grafické prvky, které můžeme v grafickém formuláři (tzv. layoutu) použít.

❸ Strom komponent (Component Tree) hierarchicky znázorňuje objekty použité na formuláři. Kliknutím na komponentu dojde k jejímu zvýraznění v editoru.

❹ Grafický editor (Design Editor) zobrazuje formulář v návrhovém režimu a vedle něj v režimu náhledu.

❺ Vlastnosti (Attributes) umožňují prohlížet a nastavovat parametry zvolené komponenty. Obsah okna se mění podle zvoleného objektu.

❻ Navigační lišta (Navigation Bar) pomáhá při orientaci ve struktuře aplikace a umožňuje snazší pohyb mezi komponentami.

❼ Lišta nástrojů (Toolbar), obsahuje tlačítka pro nejčastěji vyvolávané akce, zapínání nástrojů a zejména pro spuštění a krokování aplikace.

❽ Záložky otevřených komponent.

❾ Přepínání typu zobrazení – grafické, textové nebo kombinované. K dispozici je pouze pokud režim návrhu umožňuje kromě textové reprezentace i grafický režim práce se souborem.

Nejjednodušší způsob, jak se s prostředím seznámit, je zkoušení. Klikáním na soubory v okně projektu ❶ získáme představu, co zhruba kde je. Na souboru layoutu content_main.xml si můžeme v grafickém editoru ❹ zkusit přidat komponenty, můžeme si přepnout zobrazení na textový režim ❾ a prohlédnout si textovou reprezentaci prvků formuláře (layoutu) ve formátu XML.

5.2 Hello World!

První aplikaci se často říká „Hello World“. Jejím účelem je pouze ověřit, že všechno správně funguje. Měla by být co nejjednodušší, jedinou funkci kterou obvykle má, je zobrazení textu „Hello World“. Žádné další schopnosti se od ní zpravidla neočekávají.

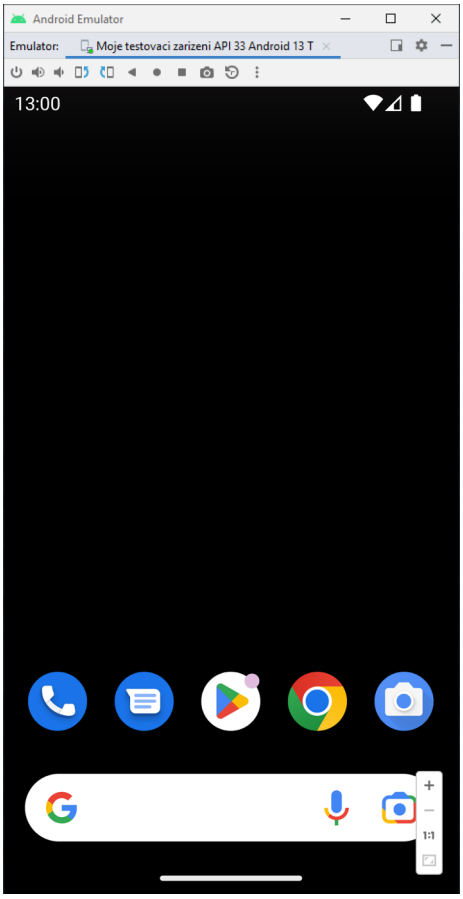
Použijeme projekt, který jsme si připravili v kapitole 5, nebo si pomocí průvodce vygenerujeme nový s aktivitou typu „Empty View Activity“. Průvodce vytvořil projekt se všemi potřebnými soubory, otevřel formulář content_main.xml a doprostřed něj umístil textové pole (TextView) s nápisem „Hello World!“. V Android Studiu by vše mělo vypadat tak, jak je znázorněno na obrázku 5.4.

V zásadě tedy nemusíme nic dalšího dělat a jednoduše prohlásíme první aplikaci za hotovou. Aniž bychom cokoliv nastavovali nebo napsali byť jen jediný řádek kódu. Na Android zařízení by tato aplikace měla zobrazila formulář s nápisem „Hello World!“ uprostřed. Nic dalšího zatím neumí.

K jejímu vylepšování se dostaneme později, teď se ji pokusíme nahrát do emulátoru.

5.3 Vytvoření emulátoru

Emulátor je virtuální napodobenina fyzického zařízení. Běží na něm operační systém Android a ze SDK je možné na emulátoru spustit vyvíjenou aplikaci. To umožňuje pohodlné a rychlé ladění bez nutnosti nahrávat aplikaci do telefonu. Zároveň dává emulátor Android Studia možnost otestovat aplikaci na širokém spektru virtuálních zařízení, která ve fyzické podobě dokonce vůbec nemusí existovat, pouze jsou v konfigurátoru nadefinována. Emulátor podporuje různé verze Androidu, takže se můžeme snadno přesvědčit, jak bude aplikace fungovat na jiných verzích operačního systému, než který je k dispozici na fyzickém zařízení.



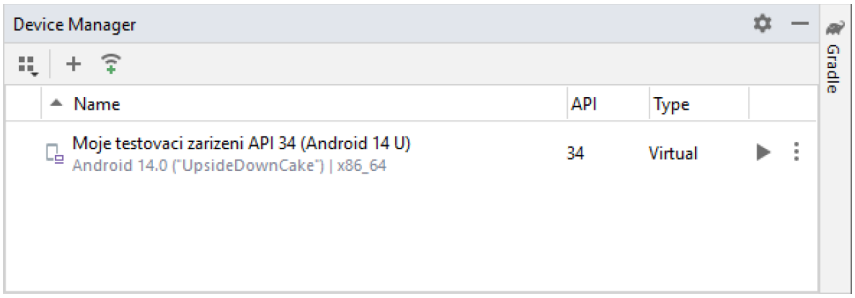
Obrázek 5.5: Emulátor zařízení

Aby bylo prostředí zařízení napodobeno co nejlépe, je možné na emulátoru simulovat volání, odesílání a příjem SMS, detekci geografické polohy atd. Emulátor nedokáže napodobit zařízení dokonale, pro základní odladění aplikace je ale nenahraditelným pomocníkem, který ušetří mnoho času.

Emulátor vytvoříme přímo v Android Studiu, spuštěním průvodce. V menu Tools → Device Manager spustíme správce virtuálních zařízení. Práce s ním je intuitivní, stačí, když si zvolíme existující zařízení, které je ve správci uložené. Nebo si můžeme vytvořit úplně nové, které ani nemusí v reálném světě existovat. Nastavíme základní parametry - rozlišení displeje, velikost paměti,

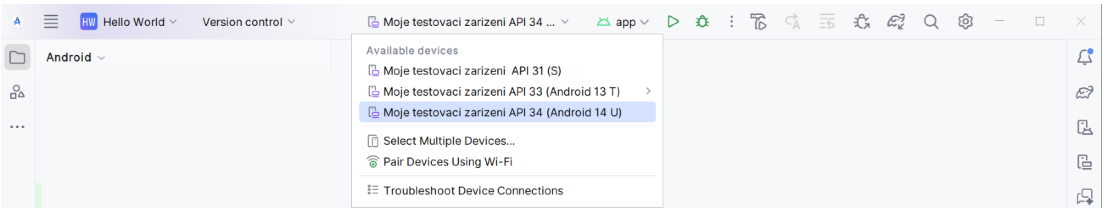
senzory nebo periférie a pod libovolným jménem si ho zařadíme do seznamu podporovaných zařízení.

Vybereme některé z virtuálních zařízení, nastavíme verzi Androidu, která v něm má být nahrána a průvodce ukončíme. Tím se naše nové zařízení dostane do seznamu zařízení připravených k emulaci v okně správce virtuálních zařízení (obrázek 5.6).



Obrázek 5.6: Správce virtuálních zařízení

Pozor, při definování virtuálních zařízení pro emulaci je vhodné používat pouze ASCII znaky bez diakritiky. Průvodce považuje znaky s diakritikou za nepovolené a může kvůli nim odmítnout zařízení vytvořit.

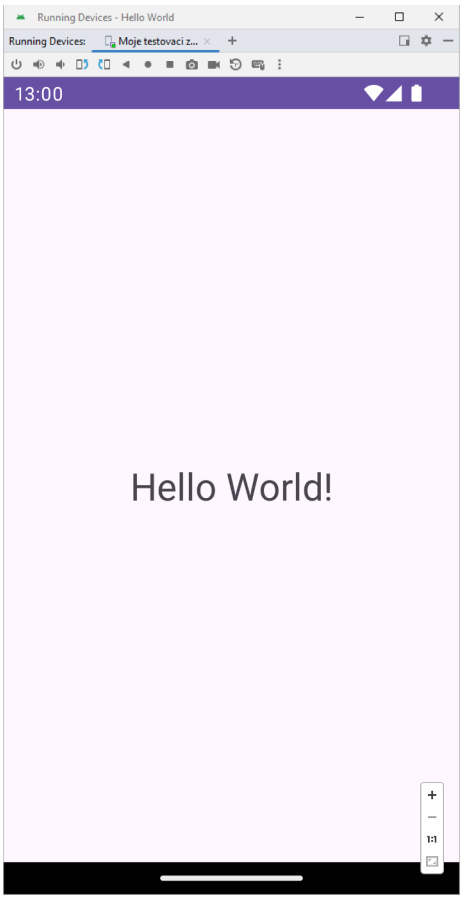


Obrázek 5.7: Menu se seznamem virtuálních zařízení (aktuálně pouze jedno)

V AVD správci zvolíme zařízení, na kterém aplikaci budeme testovat a klikneme na zelené tlačítko Launch ▷. Tím se emulátor spustí. Nejprve se v něm inicializuje operační systém, podobně jako bychom zapnuli fyzické zařízení.

Vrátíme se do Android Studia, kde je v liště nástrojů menu s názvem nového virtuálního zařízení (obrázek 5.7). Klikneme na ikonu Run app ▷, Android Studio se spojí s emulátorem, nahraje aplikaci do virtuálního zařízení a spustí ji.

Aplikace Hello World pouze zobrazí text a na žádnou interakci nereaguje, takže nyní můžeme emulaci vypnout tlačítkem Stop app □. K většímu testování na emulátoru se dostaneme později, až bude naše aplikace disponovat schopností reagovat na podněty uživatele.



Obrázek 5.8: Aplikace spuštěná na emulátoru

Každou aplikaci je vhodné před dokončením vyzkoušet na emulátoru s různými verzemi Androidu a pro různé velikosti displeje.

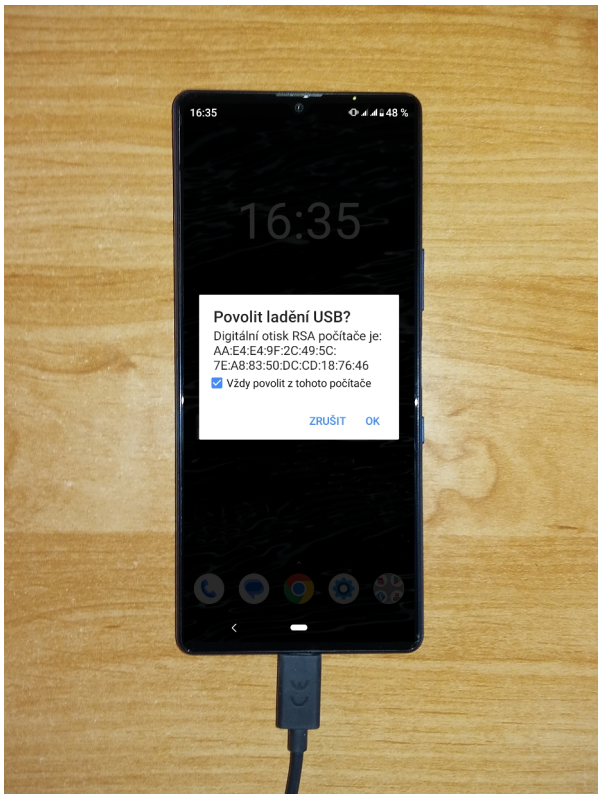
5.4 Připojení fyzického zařízení

Jakmile jsme s aplikací v emulátoru spokojeni, je čas nahrát ji do skutečného telefonu nebo jiného fyzického zařízení. Zařízení se s počítačem obvykle propojuje přes USB kabel, který bývá dodávaný prakticky s každým tabletem nebo chytrým telefonem. Často bývá součástí nabíječky. Ladit aplikace je možné i přes Wi-Fi nebo Bluetooth, je to však poněkud komplikovanější. Výhodou USB je snadné propojení a současné napájení zařízení. Bluetooth se používá zejména při ladění aplikací pro hodinky a jiná nositelná zařízení.

Zařízení některých výrobců může mít menu pro vývojáře skryté. Často se aktivuje v menu *Nastavení → Informace o zařízení → Číslo sestavení* – na něj několikrát klepnout. Poté se zobrazí informace „Stal se z vás vývojář“ a je možné vývojářské menu využívat. Pokud na vašem zařízení přesto nejde vývojářské menu aktivovat, požádejte o radu Google. Klíčovou formulí je Android + *číslo verze* + Ladění USB, nebo v anglické verzi Android + *číslo verze* + Developer options. Případně můžete dotaz rozšířit o značku telefonu.

Aby bylo možné zařízení použít pro spuštění aplikace z Android Studia, je na něm nutné nastavit takzvaný režim ladění (Debug režim). Většinou jde o jednoduchou volbu v menu *Nastavení → Systém → Pro vývojáře → Ladění přes USB*.

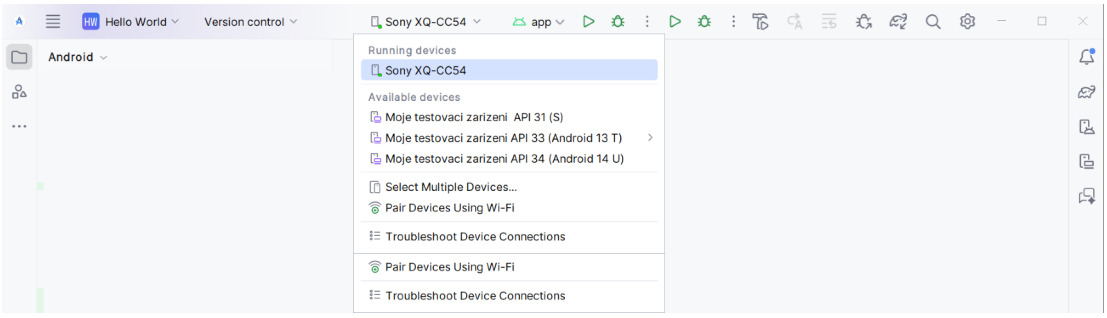
Propojení se zařízením je realizováno pomocí ADB (Android Debug Bridge), nástroje, který je součástí Android Studia. Není třeba se o něj nijak starat – při spouštění aplikací pro ladění na fyzickém zařízení se aktivuje automaticky a pro uživatele je většinu času neviditelný.



Obrázek 5.9: Připojení telefonu k počítači s Android Studiem

Zařízení tedy připojíme kabelem přes USB rozhraní. V tomto okamžiku je vhodné mít na fyzickém zařízení odemčenou obrazovku, aby bylo vidět, co se na displeji odehrává. Měl by se na něm objevit dialog vyvolaný Android Studiem, s dotazem, zda souhlasíte s laděním aplikací (obrázek 5.9).

Po potvrzení se v seznamu připojených zařízení v menu Android Studia objeví kromě virtuálního také název připojeného zařízení (obrázek 5.10).



Obrázek 5.10: Zařízení připravená pro ladění. V menu první, označené, je připojený mobilní telefon, pod ním virtuální zařízení.

V případě, že se zařízení nepodaří připojit, je dobré nejprve zkontrolovat USB kabel. Třeba tak, že zařízení připojíme bez Android Studia a přeneseme z něj nějaký soubor. Pokud toto funguje, je datové propojení v pořádku.

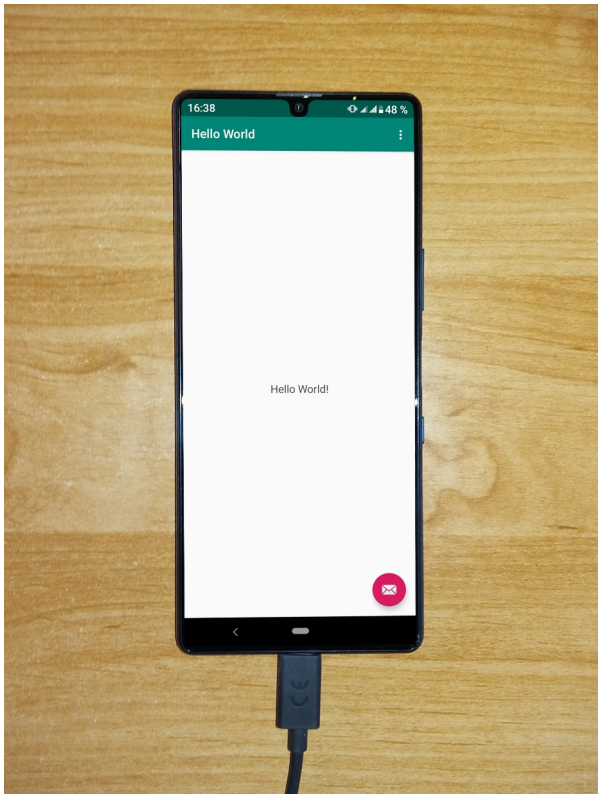
Když se nedaří při spuštěném Android Studiu na displeji zařízení vyvolat dialog s žádostí o povolení ladění, může pomoci odpojení kabelu a jeho opakované připojení. Někdy může dojít k vyvolání dialogu se zpožděním, vždy by k němu ale mělo dojít do cca 20 sekund. Tento dialog se nemusí objevit pokaždé. Jakmile jednou zaškrtneme volbu „Vždy povolit tento počítač“, připojuje se už bez potvrzování uživatelem.

Problémem mohou být také antivirové programy blokující USB rozhraní. To lze ověřit pozastavením takového programu, a pokud to situaci vyřeší, je třeba nastavit výjimku pro připojování ADB k USB portu.

Při přetrvávajících potížích lze využít asistenta v Android Studiu. Najdete ho v menu připojených zařízení→ *Troubleshoot Device Connection* a umí napovědět, zda Studio zařízení vidí a v jakém je aktuálně stavu.

Některé verze Androidu mají v menu pro Vývojáře ještě volbu *Nastavení → Pro vývojáře → Odvolat autorizace ladění USB*. Pomocí ní je možné zařízení donutit, aby na předchozí potvrzení zapomnělo a příště o ně požádalo znovu. Toto se může hodit, pokud se zařízení odmítá připojit, ačkoliv už dříve připojené bylo.

Jakmile je zařízení řádně připojené a v menu Android Studia zvolené jako aktivní, stačí kliknout na ikonu Run app ▷. Android Studio přes ADB nahraje aplikaci do zařízení a spustí ji (obrázek 5.11). Po skončení ladění a odpojení USB kabelu zůstává dál v telefonu, v seznamu nainstalovaných aplikací. Lze se tedy kdykoliv pokochat pohledem na první vlastní výtvor, aplikaci Hello World!



Obrázek 5.11: Aplikace „Hello World!“, spuštěná na mobilním telefonu.

6 První opravdová aplikace

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 06_Prvni_aplikace
projekt PrvniAplikaceJava
PrvniAplikaceKotlin

Zkusíme napsat jednoduchou aplikaci, která zobrazí formulář s tlačítkem. Tlačítko bude reagovat na kliknutí uživatele tak, že zobrazí základní dialogové okno s textovou informací. Více toho zatím nebudeme potřebovat.

Než se ale pustíme do tvorby, musíme vyjasnit první zásadní otázku – který programovací jazyk zvolit? Kdo v tom má jasno, může s klidem v srdci následující podkapitolu přeskočit. Pro ty, kteří váhají, se kterým jazykem začít, nebo případně zda znalost jednoho, zpravidla Javy, rozšířit o znalost druhého, tedy Kotlinu, jsou určeny následující řádky.

6.1 Java nebo Kotlin?

Na počátku byla Java. Hlavní a jediný jazyk pro programování aplikací operačního systému Android. Java byla rozšířeným a oblíbeným jazykem, dostupným na mnoha platformách, s obrovskou základnou uživatelů, knihoven a dokumentace. Byla tenkrát jasnou volbou. Postupem času se ale právě toto, v kombinaci s potřebou zpětné kompatibility, stalo příčinou malé pružnosti při zavádění nových programátorských technik. Java se vyvíjela pomaleji než nově nastupující jazyky.

Jedním z takových jazyků byl nenápadný Kotlin společnosti JetBrains. Tento jazyk vznikl v roce 2011 a díky svým vlastnostem a také trochu díky licenčním problémům s Javou, byl v roce 2017 začleněn do Android Studia a od toho okamžiku je oficiálním a plnohodnotným jazykem pro vývoj Android aplikací. V tomto směru jsou si tedy dnes oba jazyky prakticky rovny. Příjemnou vlastností je, že Kotlin umí využívat knihovny napsané v Javě, takže přechod na něj je možný i dodatečně, stejně jako je do jisté míry možná spolupráce Java a Kotlin vývojářů.

Jaké jsou základní rozdíly? Především je to syntaxe. Ta je u obou jazyků jednoduše rozdílná. Způsob, jak napsat jednoduchou metodu bez parametrů a návratové hodnoty, která v cyklu vypíše obsah seznamu v Javě:

```
private void test() {
    List<String> verzeAndroidu = new ArrayList<>();
    verzeAndroidu.add("Marshmallow");
    verzeAndroidu.add("Nougat");
    verzeAndroidu.add("Oreo");
    for (String verze : verzeAndroidu) {
        System.out.println(verze);
    }
}
```

Na prvním řádku je název privátní metody bez vstupních parametrů a bez návratové hodnoty (void). Na druhém řádku je vytvořena instance seznamu verzeAndroidu, kterou plní další tři řádky. Následuje cyklus, který seznam projde a postupně jednotlivé položky seznamu vypisuje na konzoli. A ta stejná funkce napsaná v Kotlinu:

```
fun test(){
    val verzeAndroidu = listOf("Marshmallow", "Nougat", "Oreo")
    for (verze in verzeAndroidu) {
```

```
        println(verze)
    }
}
```

Kromě rozdílu v syntaxi je vidět, že Kotlin umí být prostorově úspornější. I když by šlo zdrojový kód v Javě dále optimalizovat, kód Kotlinu bude jednoduše kratší. Dalšího zhuštění by se dalo dosáhnout použitím lambda výrazů.

Co se týká dalších významných vlastností, Kotlin je kompatibilní s JDK (Java Development Kit) verze 6, což zajišťuje možnost běhu aplikací i na starších verzích Androidu, vydaných ještě před Kotlinem. Výkon obou jazyků je vzhledem ke shodnému výstupnímu binárnímu kódu prakticky stejný. Rozdíl je pouze u rutin postavených na funkcích, které jsou v Kotlinu dostupné, zatímco v Javě je nutné použít složitější konstrukci. Tam lze s Kotlinem pochopitelně dosáhnout vyššího výkonu.

Důležitou vlastností je interoperabilita. Java má k dispozici nepřeberné množství knihoven, které může programátor při práci do svých projektů začlenit a využívat. To je nezpochybnitelná výhoda Javy. Kotlin je díky interoperabilitě může využít také, nicméně pokud bude v knihovně potřeba udělat úpravu a znovu ji sestavit, je ji nutné buď přepsat do Kotlinu, nebo modifikaci provést v Javě.

Pokud jde o samotný vývoj aplikací, je dobré vědět, že kompilace v obou jazycích trvá přibližně stejně dlouho. Kotlin má ale výrazně strmější učící křivku než Java. Budete-li se učit jazyk od samého začátku, s Kotlinem vám to půjde rychleji.

Podrobnější popis vlastností a rozdílů mezi oběma jazyky by byl na samostatnou knihu. Ať už se rozhodnete pro opravdu široce využitelnou Javu, nebo modernější Kotlin, budete muset investovat dostatek času k jeho zvládnutí. Následující kapitoly a příklady využívají převážně jazyk Java.

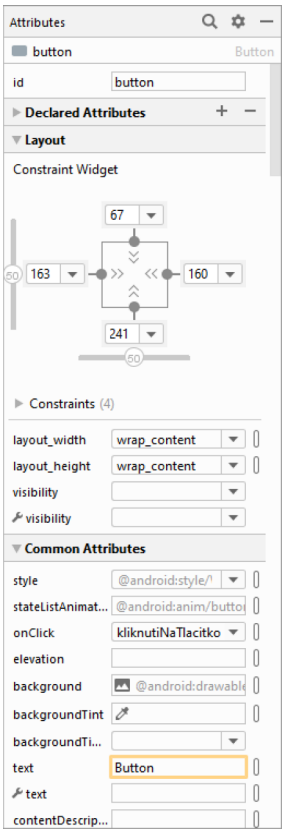
6.2 Jdeme na to

Vygenerujeme si nový projekt nebo použijeme stávající. Vzhledem k tomu, že 'Hello World!' zůstal bez úprav, můžeme na něj prostě navázat. Zkusíme tuto automaticky vygenerovanou aplikaci trochu vylepšit.

Kliknutím na textové pole 'Hello World!' se po pravé straně zobrazí jeho vlastnosti (Attributes), na obrázku 5.4 označené jako ❺. Textové pole můžeme pomocí vlastností upravit, zejména změnit obsažený text (sekce Common Attributes, položka 'text'). Změna se okamžitě promítne do grafického náhledu. Tím jsme upravili jednoduché textové pole.

Zkusíme si ještě přidat v grafickém editoru na plochu formuláře ❹ tlačítko (Button). Stačí ho jednoduše vzít v paletě komponent ❷ a umístit do formuláře. Polohu a velikost můžeme měnit myší přímo v editoru. Kliknutím na tlačítko se po pravé straně opět zobrazí jeho vlastnosti (obrázek 6.1).

Položky vlastností se mírně liší od vlastností textového pole, nás ale zatím bude zajímat položka 'text', ve které si nastavíme popisek na tlačítku. Například na Pokusné tlačítko. Pozornost věnujeme také položce ID (nahoře ve vlastnostech tlačítka). Editor do ní vygeneroval text 'button', což je identifikační klíč tlačítka. Jeho prostřednictvím se

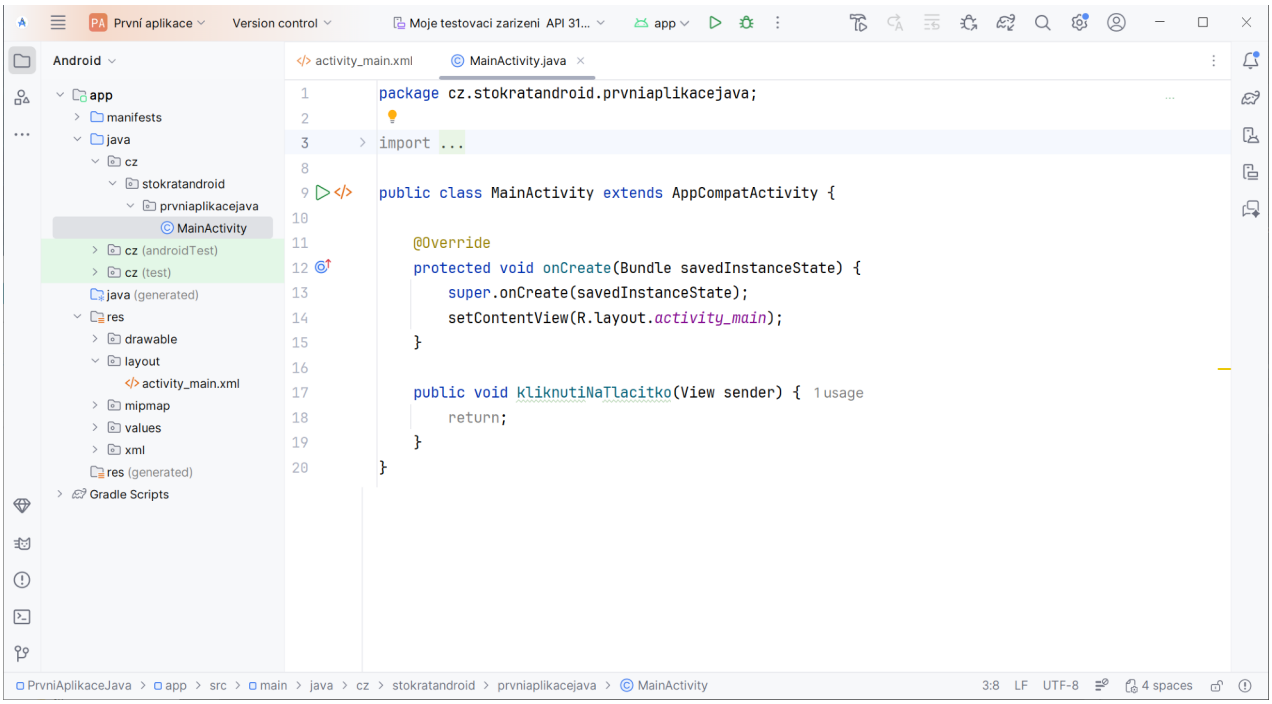


Obrázek 6.1: Vlastnosti tlačítka

budeme na tlačítko odkazovat vždy, když budeme potřebovat programově změnit některou z jeho vlastností.

Teď musíme vyřešit pozici tlačítka. Je umístěné ve formuláři, pravděpodobně typu Constraint Layout, protože ho tak vygeneroval průvodce. Typů formulářů může být více, seznámíme se s nimi podrobně později. Zatím se spokojíme s tím, že tlačítko ve stávajícím formuláři „upevníme“ na svoje místo pomocí konstantního nastavení velikosti jeho okrajů. Kdybychom to neudělali, tak i kdyby bylo v editoru pečlivě umístěné na vhodnou pozici, stejně by se nakonec zobrazilo někde v rohu formuláře. Takže v okně vlastností tlačítka, v části Layout → Constraint Widget, nastavíme jeho okraje, čímž dosáhneme zafixování na požadovaném místě ve formuláři.

Ze všech parametrů tlačítka v okně Attributes je však tím nejdůležitějším pole onClick. Do něj zapíšeme název Java metody, která se spustí po kliknutí na tlačítko. Teprve ta dá tlačítku smysl. Název si vymyslíme, například kliknutiNaTlacitko.



Obrázek 6.2: Třída MainActivity

Pro názvy metod se v Javě i Kotlinu používá konvence první písmeno malé, další písmena ve slově rovněž malá. Pokud je název víceslovný, tak každé další slovo začíná velkým písmenem a následují písmena malá. Mezi slovy se nepoužívá žádný oddělovač. Názvy metod můžou být v českém jazyce, ale zásadně bez diakritiky.

Nyní nastala ta pravá chvíle pro napsání první metody. Té, která bude volána po kliknutí na tlačítko. Její jméno už máme - kliknutiNaTlacitko. Metody pro hlavní okno se píšou do souboru activity_main.xml, který obsahuje třídu MainActivity. Najdeme ho v okně projektu a otevřeme (obrázek 6.2). Obsahuje několik vygenerovaných metod. Přidáme k nim svoji metodu kliknutiNaTlacitko().

Metoda bude mít záhlaví a zatím, kromě instrukce return, prázdné tělo (verze Java):

```
public void kliknutiNaTlacitko(View sender) {
    return;
}
```

Případně v jazyce Kotlin:

```
public fun kliknutiNaTlacitko(sender: View) {
    return
}
```

Klíčové slovo public určuje viditelnost metody. Viditelnost říká, odkud můžeme k objektu přistupovat. Public znamená, že metoda bude viditelná i z jiných tříd. To je pro nás důležité, protože metodu voláme po stisku tlačítka z vnějšku třídy MainActivity. Kromě metod se viditelnost definuje také u proměnných, tříd, konstant, atributů atd. V Kotlinu není klíčové slovo public nutné uvádět, každý objekt, který nemá viditelnost specifikovanou, je automaticky typu public.

Public (veřejné)	Objekty jsou viditelné (tj. proměnné lze plnit nebo číst, metody volat atd.) ze všech ostatních tříd. Příklad deklarace proměnné: `public int i=1;`
Protected (chráněné)	Objekty jsou viditelné pouze ze třídy v níž jsou deklarované (definované, vytvořené), ve třídách oděděných a ve třídách ve stejném balíčku (package). Příklad deklarace proměnné: `protected int i=1;`
Viditelnost neuvedena	Objekty jsou viditelné pouze ze třídy v níž jsou deklarované a ve třídách ve stejném balíčku (package). Příklad deklarace proměnné: `int i=1;`
Private (soukromý, privátní)	Objekty jsou viditelné pouze ze třídy v níž jsou deklarované. Příklad deklarace proměnné: `private int i=1;`

Tabulka 6.1: Viditelnost objektů Java

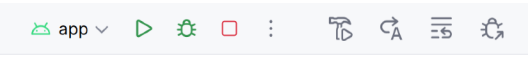
Public (veřejné)	Objekty jsou viditelné (tj. proměnné lze plnit nebo číst, metody volat atd.) ze všech ostatních tříd. V Kotlinu není nutné uvádět, objekt bez uvedené viditelnosti je vždy veřejný.
	Příklady deklarace proměnné:
	`public val i = 1`
	`val j = 1`

Protected (chráněné)	Objekty jsou viditelné pouze ze třídy v níž jsou deklarované, ve třídách oddděných a ve třídách ve stejném balíčku (package). Příklad deklarace proměnné: `protected val i = 1`
Internal (interní)	Objekty jsou viditelné pouze z modulu v němž jsou deklarované. Příklad deklarace proměnné: `internal val i = 1`
Private (soukromý, privátní)	Objekty jsou viditelné pouze ze třídy v níž jsou deklarované Příklad deklarace proměnné: `private val i = 1`

Tabulka 6.2: Viditelnost objektů Kotlin

Void je označení návratové hodnoty - znamená, že tato metoda žádnou návratovou hodnotu neposkytuje. V závorce za názvem metody je parametr typu View, pojmenovaný 'sender'. V něm budou předávány informace o objektu, který metodu zavolal. Tělo metody obsahuje pouze příkaz return, který říká, že má metoda skončit. Máme tedy vytvořenou metodu kliknutiNaTlacitko() a na tlačítku je v atributu onClick nastaveno, že ji po kliknutí zavolá.

Android Studio má v liště nástrojů řadu tlačítek pro sestavení (build, kompilaci) a spuštění aplikace:



Obrázek 6.3: Tlačítka pro sestavení a spuštění aplikace v Android Studiu

Run app (spustit aplikaci) – zkompiluje, nahraje a spustí aplikaci na zvoleném virtuálním nebo fyzickém zařízení. Takto lze vyzkoušet spuštěnou aplikaci, nelze ji ale ladit. Dokud je aplikace spuštěna, je ikona nahrazena symbolem restartu.

Debug app (ladění aplikace) - spustí aplikaci na zvoleném virtuálním nebo fyzickém zařízení. Na rozdíl od Run app umožní aplikaci ladit, neboli zastavovat, krokovat, sledovat hodnoty proměnných atd.

Stop app - zastaví běžící aplikaci. Tlačítko je aktivní pouze pokud je aplikace spuštěna nebo je v režimu ladění. V takovém případě je tlačítko červené.

Make project (sestavit projekt) - zkompiluje aplikaci. Pokud obsahuje chyby, zobrazí je. Aplikaci nespouští, pouze vytvoří potřebné soubory, knihovny, balíčky.

Apply Changes and Restart Activity – vloží změny v kódu nebo resources do běžící aplikace a zrestartuje aktuální aktivitu bez toho, aby ukončil běh celé aplikace. Hodí se při ladění, když potřebujeme rychle udělat úpravu a bez zdlouhavého kompilování a spouštění promítnout změnu do laděné aplikace.

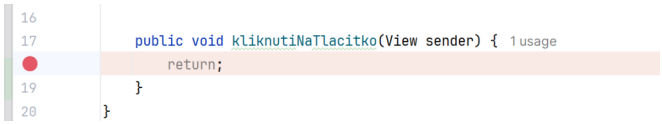
Apply Code Changes - zařadí změny v kódu (pozor, netýká se resources) do běžící aplikace. V zařízení se aplikace ani aktuální aktivita nerestartuje.

Attach debugger to Android process (ladit běžící proces) - umožní připojit Android Studio k procesu spuštěnému na zařízení a tento proces ladit.

Spusťme aplikaci pomocí tlačítka pro ladění . Aplikace se spustí na zařízení a čeká na akci uživatele. Po kliknutí na tlačítko by se nemělo nic zvláštního stát, spokojíme se s tím, že nedojde k chybě. Pokud je po kliknutí na tlačítko vyvolaná v Android Studiu výjimka (neboli k chybě došlo), znamená to, že špatně voláme metodu kliknutiNaTlacitko() a je potřeba zkontrolovat volání jak v atributu onClick na tlačítku, tak v definici metody.

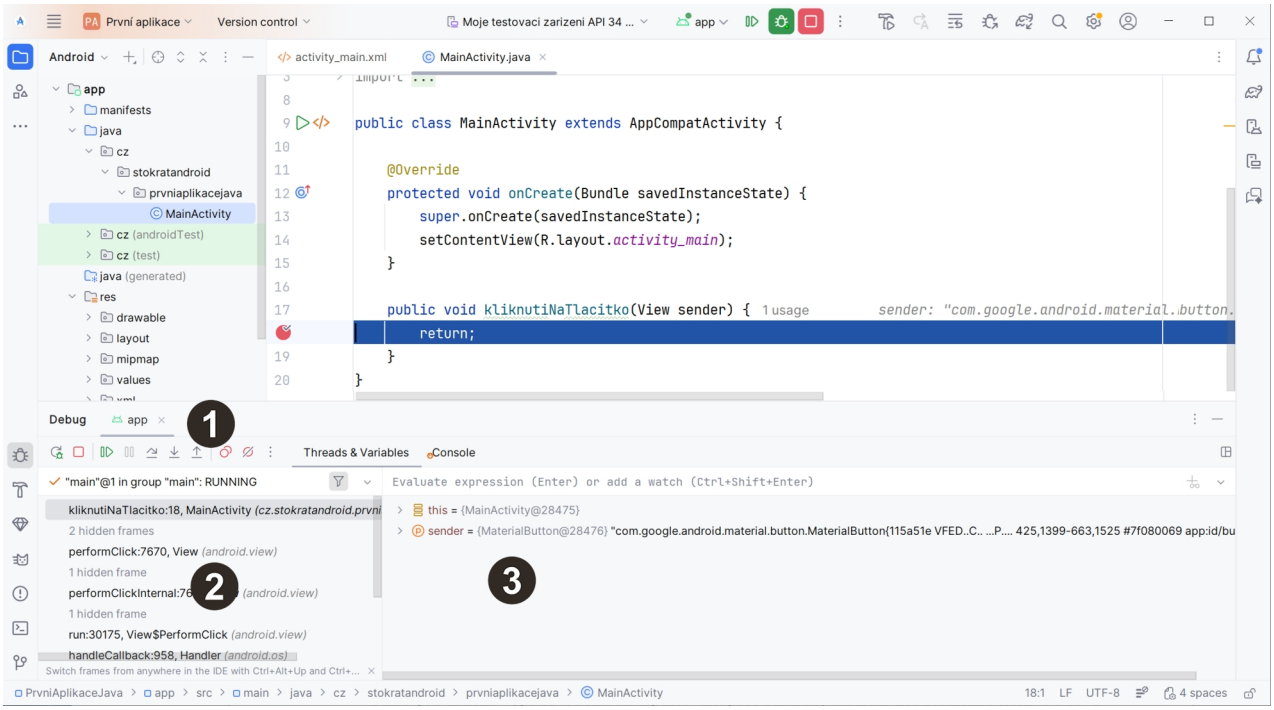
Žádný software není dokonalý. A dokonalý není kompilátor Android Studia ani emulátor. Pokud se stane, že aplikaci jde v Android Studiu přeložit a přesto v emulátoru nebo na koncovém zařízení hned po spuštění spadne, může pomoct vyčistit projekt (menu Build → Clean Project), přegenerování APK (menu Build → Build APK) a nová kompilace projektu (menu Build → Make project).

Jakmile je vše v pořádku a klikání na tlačítko nic špatného nezpůsobuje, zkusíme v editoru Android Studia udělat breakpoint na instrukci return v metodě kliknutiNaTlacitko(). Dosáhneme toho kliknutím napravo od čísla řádku, v okně nalevo od instrukce return. Řádka se podbarví a vedle čísla řádku se objeví červený puntík.



Obrázek 6.4: Nastavený breakpoint, který způsobí zastavení běhu aplikace jakmile bude zpracovávána označená instrukce.

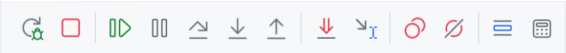
Když teď klikneme na tlačítko na displeji, program se zastaví na instrukci return. Na zařízení to poznáme tak, že aplikace dál nereaguje a v Android Studiu se podbarví řádek return tmavě modrou barvou. Současně se v dolní části otevře okno pro ladění aplikace (obrázek 6.5).



Obrázek 6.5: Ladění aplikace

Okno pro ladění vypadá trochu komplikovaně, ale fakticky na něm nic složitého není. Po levé straně je pás řídících ikon ❶, pod ním je zásobník volání (Call stack) ❷, který nás zatím nebude zajímat. A napravo vedle něj je okno Variables (proměnné), které zobrazuje všechny proměnné viditelné z metody, ve které se program zastavil ❸.

Nad okny zásobníku a proměnných je vodorovný pruh s tlačítky pro krokování:



Obrázek 6.6: Řídící ikony v Android Studiu

Rerun app (restartovat aplikaci) – rychlý restart aplikace během ladění, se zohledněním změn ve zdrojovém kódu.

Stop app (zastavit aplikaci) - ukončí režim ladění aplikace.

Resume Program (pokračovat ve zpracování) – spustí zastavený program. Nechá ho pokračovat dál bez krokování, dokud nenarazí na další breakpoint nebo neskončí.

Pause Program (pozastavit zpracování) - pozastaví aplikaci v místě, kde zrovna probíhá její zpracování. Lze pokračovat krokováním.

Step Over (udělat krok) - program přejde na následující instrukci. Pokud je aktuální instrukcí volání jiné metody, vykoná ji, aniž by v ní zastavil (toto platí pouze pokud ve volané metodě není aktivní breakpoint, jinak zastaví na něm). Pro rychlejší práci se při krokování používá klávesa F8.

Step Into (krok dovnitř) - pokud je aplikace zastavena na volání jiné metody, zavolá ji a zastaví na její první instrukci. Není-li aktuální instrukcí volání, zpracuje ji a udělá krok na instrukci následující. Jinými slovy - Step Into je shodný se Step Over, pouze s tím rozdílem, že krokuje i dovnitř volaných metod. Pro rychlejší práci se při krokování používá klávesa F7.

Step Out (vyskočit ven) - opustí zpracovávanou metodu a zastaví na instrukci která následuje po jejím volání.

Force Step Into (vynucený krok dovnitř) - podobně jako Step Into, jenom s tím rozdílem, že v tomto případě jsou krokovány i volání systémových metod. pro zajímavost si to můžeme zkusit, v praxi to ale zatím nevyužijeme.

Run to Cursor (spustit ke kurzoru) - zastavený program spustí a znovu ho zastaví až na řádce, na které je v editoru aktuálně umístěný kurzor. Pokud mu bude stát v cestě nějaký další aktivní breakpoint, zastaví samozřejmě už na něm.

View Breakpoints (zobrazit breakpointy) - otevře okno s přehledem breakpointů a umožní jejich nastavení (například aby program na breakpointu zastavil pouze pokud je splněna nastavená podmínka).

Mute Breakpoints (deaktivovat breakpointy) - dočasně zneplatní breakpointy. Tyto nezmizí, pouze na nich program při zpracování nezastaví.

Show Execution Point (zobrazit místo zpracování) - najde v editoru místo na kterém je aplikace aktuálně zastavena. Toto se hodí v okamžiku, kdy procházíme více souborů, chceme pokračovat v krokování a už nevíme na kterém breakpointu jsme vlastně skončili.

Evaluate Expression (vyhodnocení výrazu) - otevře dialogové okno ve kterém lze zobrazit stav proměnné nebo výrazu.

Můžeme si vyzkoušet funkci tlačítka pro vyhodnocení výrazu, Evaluate Expression. Do okna, které se po kliknutí na tlačítko otevře, zadáme název parametru metody ‚sender‘ a klikneme na tlačítko Evaluate. Zobrazí se struktura, kterou parametr obsahuje. Obsah nám zatím nic moc neřekne, takže ještě zkusíme krokování a pokračování programu.

Potom aplikaci zastavíme a třídu MainActivity.java rozšíříme o další instrukce. Do záhlaví přidáme řádek:

```
import android.app.AlertDialog;
```

Záhlaví už několik importů obsahuje, takže tento pouze přiřadíme na jejich konec. Tím zajistíme, že se tato třída bude schopna odkázat na metody třídy AlertDialog, které budeme za chvíli volat. Metodu kliknutiNaTlacitko() rozšíříme o další instrukce:

```
public void kliknutiNaTlacitko(View sender) {
    AlertDialog.Builder builder = new AlertDialog.Builder(MainActivity.this);
    builder.setCancelable(true);
    builder.setTitle("Poznámka");
    builder.setMessage("Android je zde!");
    builder.setPositiveButton("Zavřít", null);

    builder.create().show();

    return;
}
```

Verze pro jazyk Kotlin je velmi podobná:

```
fun kliknutiNaTlacitko(sender: View) {
    val builder = AlertDialog.Builder(this@MainActivity)
    builder.setCancelable(true)
    builder.setTitle("Poznámka")
    builder.setMessage("Android je zde!")
    builder.setPositiveButton("Zavřít", null)

    builder.create().show()

    return
}
```

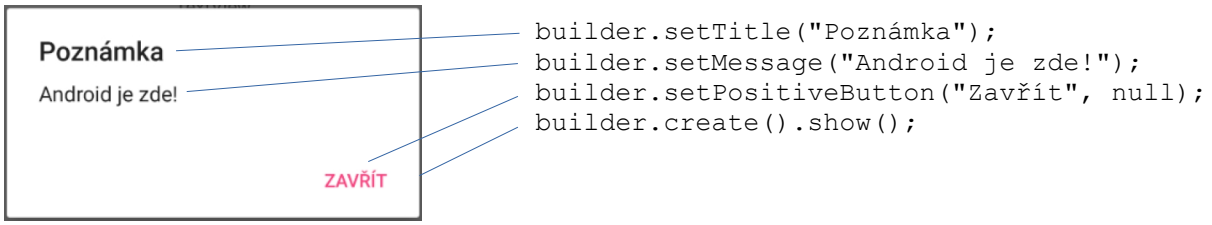
Tato sekvence instrukcí zobrazí po stisknutí tlačítka na displeji dialogové okno s definovaným obsahem. Instrukce vypadají komplikovaně, ale v jádru nejde o nic složitého. První řádka vytvoří novou instanci třídy AlertDialog.Builder a dá jí název builder. Třída AlertDialog je odvozená od systémové třídy Dialog, která slouží k zobrazování různých typů dialogových oken na displeji. AlertDialog je uživatelsky nejjednodušší typ dialogu – prosté okno se záhlavím, textem a tlačítky. Třída Builder se používá pro jednoduché definování konstrukce dialogu. MainActivity.this (resp. this@MainActivity) v parametru odkazuje na kontext, ve kterém instance vznikne, čili na třídu MainActivity, ze které dialog voláme. Toto všechno dohromady zařídí, že je v proměnné builder instance jednoduchého dialogu, který dál nastavíme podle vlastních představ a nakonec zobrazíme.

Při psaní kódu je možné využít rozšířené schránky Android Studia, které si pamatuje několik dříve kopírovaných objektů. Pro vložení stačí stisknout CTRL+SHIFT+V.

V dalších řádcích jsou nastavené atributy, které říkají, jak má dialog vypadat. True v atributu setCancelable říká, že okno bude možné zavřít pomocí tlačítka Zpět na koncovém zařízení.

“Poznámka” v setTitle nastavuje název zobrazený v záhlaví okna, setMessage definuje zobrazenou zprávu a setPositiveButton určuje obsah kladného tlačítka. AlertDialog může mít tři tlačítka – souhlasné (positive), nesouhlasné (negative) a neutrální (neutral). Null u setPositiveButton říká, že po stisknutí dialog pouze zavřeme a žádnou obslužnou rutinu dál volat nebudeme. Kdybychom ji zavolat chtěli, bude definovaná na pozici druhého parametru, namísto hodnoty ‚null‘.

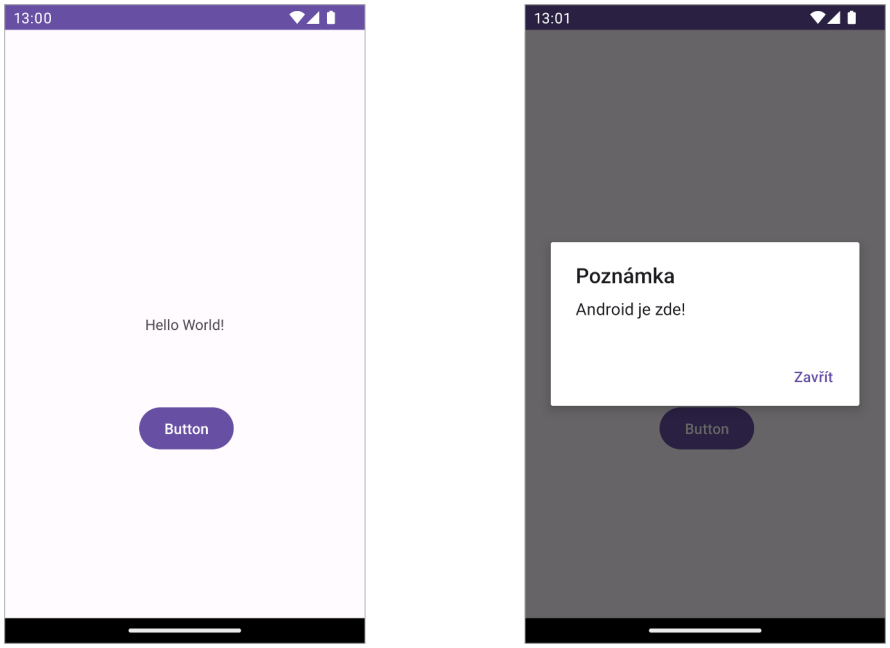
Potom už jen na poslední řádce dialogové okno podle zadaných parametrů vygenerujeme v paměti (create) a rovnou i zobrazíme na displeji (show).



Obrázek 6.7: Dialog typu AlertDialog

Podobným způsobem bychom mohli zobrazit například dialogy pro zadání data a času. Stačí, abychom místo AlertDialog použili DatePickerDialog nebo TimePickerDialog. Zápis instrukcí (tzv. syntaxe) se může jevit jako těžko naučitelný, není si ho ale nutné přesně pamatovat. Většinou se stačí rozhlédnout po internetu a základní rutiny si najít a upravit. Nebo se lze spolehnout na automatické doplňování instrukcí v editoru, který během psaní kódu napovídá, jaké máme možnosti.

Podrobněji se budeme dialogům věnovat v kapitole 16 Dialogová okna. V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 6.8.



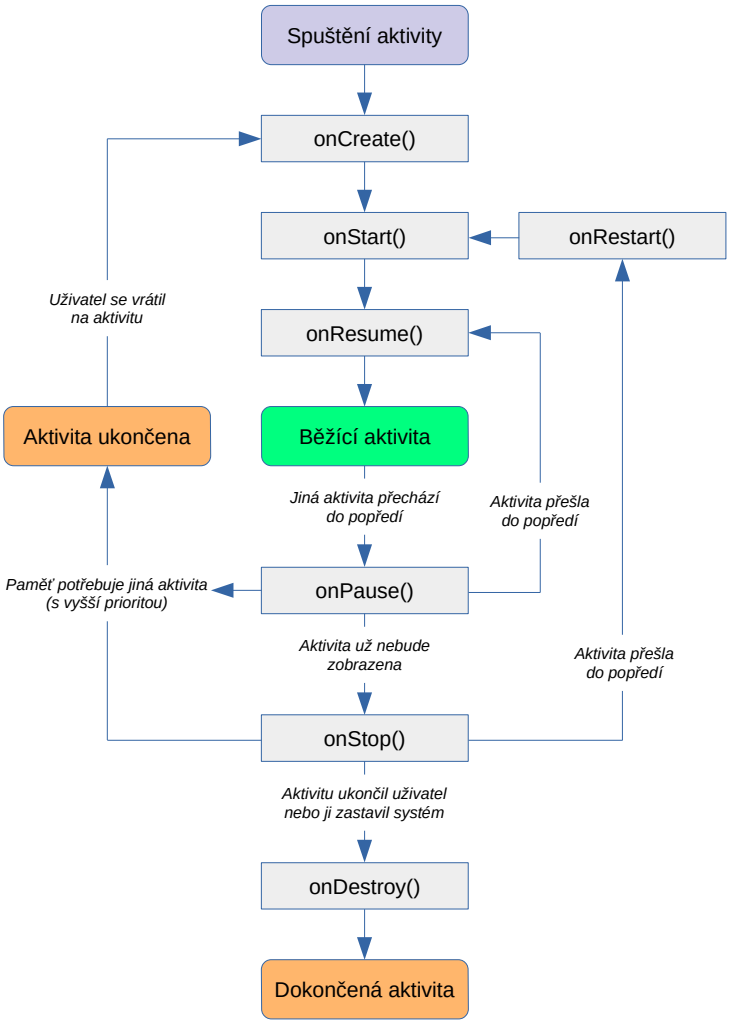
Obrázek 6.8: Naše první opravdová aplikace, spuštěná na emulátoru. Vlevo hned po startu, vpravo po kliknutí na tlačítko.

7 Životní cyklus aktivity

Jak již víme, každá Android aplikace se skládá z aktivit neboli jakýchsi formulářů, mezi kterými uživatel přepíná. Aktivita zahrnuje jak grafiku formuláře, tak i programový kód. Naše první aplikace obsahovala jednu aktivitu, kterou jsme si při vytváření projektu nechali vygenerovat průvodcem jako takzvanou Basic Activity. Klasická aplikace mívá jednu hlavní aktivitu a na ni volně navázané aktivity další, na které může uživatel přejít. Každá z těchto aktivit má takzvaný životní cyklus. Životním cyklem se myslí stavy, ve kterých se může nacházet, a přechody mezi nimi. Například když uživatel přejde na jinou než základní obrazovku aplikace, neboli když přepne z jedné aktivity na druhou, tak se výchozí aktivita přepne z aktivního stavu do stavu čekání.

Uvnitř systému to probíhá tak, že při přechodu na jinou aktivitu je stávající aktivita zastavena a uložena do zásobníku, aby se k ní mohl uživatel vrátit.

Znát životní cyklus je pro vývojáře důležité, protože při přechodu mezi jednotlivými stavy aplikace často potřebuje dělat různé akce, například když uživatel přejde na jinou aktivitu, je potřeba uložit hodnoty, které ve formuláři zadal. Systém k tomu dává hned několik příležitostí při každé takové změně stavu tím, že volá události definované ve třídě dané aktivity. Viz obrázek 7.1.



Obrázek 7.1: Životní cyklus aktivity

Obdélníky se zaoblenými rohy označují stavy aktivity. Obdélníky s ostrými rohy jsou metody, které jsou při přechodu mezi stavy volány pomocí takzvaných událostí. Takže například po spuštění aktivity (u nás tedy hned po startu aplikace) jsou postupně volány události onCreate(), onStart() a onResume(). Potom teprve aktivita přejde do stavu Běžící aktivita a čeká na nějakou reakci uživatele nebo systému. Toho lze využít a před spuštěním aktivity, v některé z těchto metod, provést potřebné akce. Například nastavit ovládací prvky, připojit se k databázi, iniciovat proměnné, upravit vzhled formuláře a tak podobně.

Když se uživatel rozhodne aktivitu ukončit (v našem případě to znamená ukončení celé aplikace), tak jsou zavolány metody onPause(), onStop() a onDestroy(). Toho využijeme například k uložení dat nebo uvolnění systémových prostředků, jako je odpojení od databáze a podobně. Pokud aktivita běží a na displej se dostane jiná aktivita, je zavolána metoda onPause(). A když ještě následně nemá systém dost paměti pro novou aktivitu, tak naši upozaděnou aktivitu odstraní z paměti (stav Aktivita ukončena). Když se k ní uživatel vrátí, proběhne znovu postupné volání onCreate(), onStart() a onResume().

Všechny tyto metody můžeme využít. Povinná je ale pouze jedna – onCreate(). Když se podíváme do třídy MainActivity, zjistíme, že se tam ze všech zmíněných nachází pouze tato jediná. Je to proto, že všechny ostatní jsou v nadřazených třídách, od kterých je MainActivity oddděděna. Na první úrovni je MainActivity oddděděna od třídy AppCompatActivity, viz řádek

```
public class MainActivity extends AppCompatActivity {
```

který říká, že veřejně viditelná třída (public class) MainActivity rozšiřuje neboli dědí (extends) třídu AppCompatActivity. Na název třídy AppCompatActivity můžeme v editoru kliknout pravým tlačítkem myši a vybrat v kontextovém menu Go To → Declaration. Tím se dostaneme na začátek třídy AppCompatActivity, kde zjistíme že i tato je oddděděna od jiné třídy, FragmentActivity. Příliš nás zatím nemusí zajímat důvod, proč to tak je – ten vychází z objektově orientovaného programování, důležité pro nás ale bude, že když hledaná metoda není v MainActivity, najdeme ji v AppCompatActivity. A když ani tam ne, tak ve FragmentActivity už určitě.

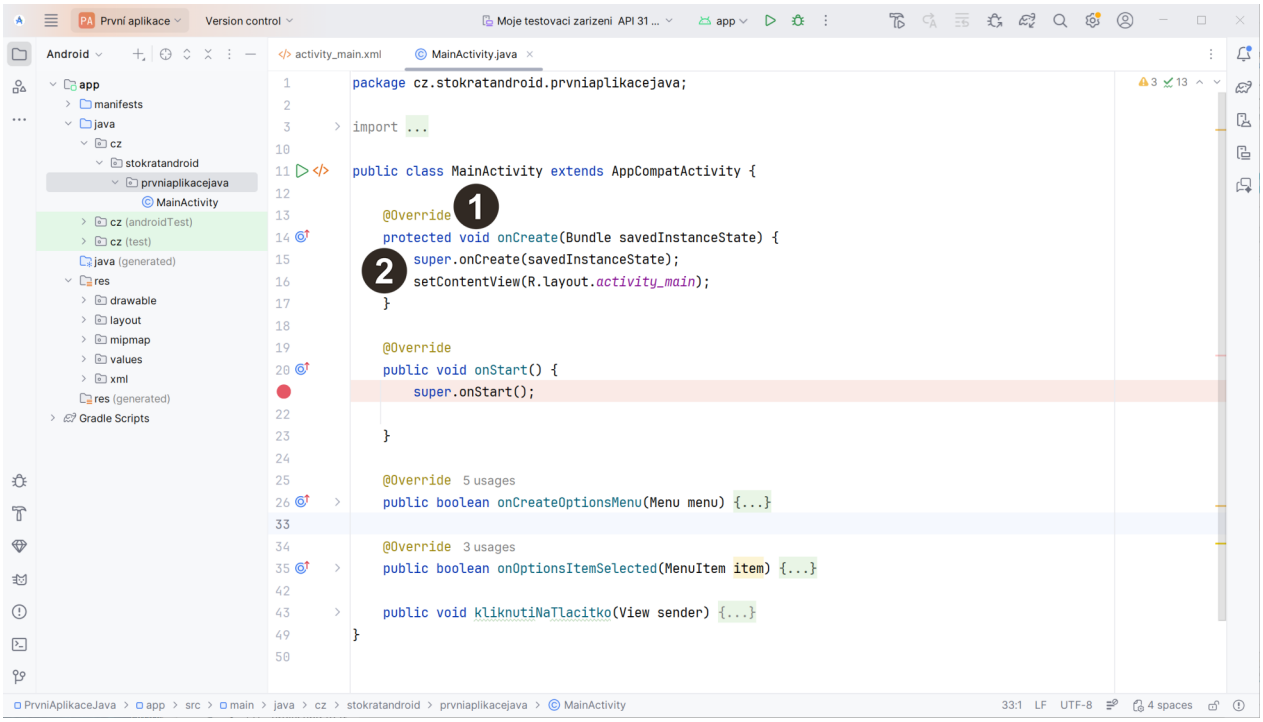
Do události onCreate() si můžeme umístit breakpoint a tak se přesvědčit, že po spuštění aplikace, respektive aktivity, v debug režimu, systém skutečně tuto metodu zavolá a zastaví v ní.

Teoreticky bychom tyto metody mohli upravit pro svoje potřeby už v nadřazených třídách, tj. například už v AppCompatActivity. Problém ale je, že od AppCompatActivity dědí i jiné třídy z jiných aktivit. A všechny naše úpravy by se objevily i v nich. Proto potřebujeme dosáhnout toho, aby se tyto metody objevily v MainActivity a aby je odtud také systém volal.

Jelikož jsou všechny metody definované v třídách, které rozšiřujeme, je pro jejich využití v MainActivity potřeba použít malý trik. Metodu, kterou chceme použít, je potřeba znovu napsat. Jelikož bychom pak ale v naší třídě měli dostupné dvě metody stejného názvu, tak tu první prostě přepíšeme naší novou metodou. K tomu slouží direktiva @Override, viz ❶ na obrázku 7.2, kde je totéž použité ve vygenerované události onCreate(). Nyní, při běhu programu systém nezavolá původní metodu, ale tu naši, předefinovanou. Abychom ale nepřišli o zpracování metody v původní třídě (přece jenom tam z nějakého důvodu je), zavoláme ji hned na začátku (viz ❷).To se udělá takto:

```
super.onCreate(savedInstanceState);
```

Super je klíčové slovo označující nadřazenou třídu, onCreate() je název události, kterou jsme předefinovali a ona tím pádem nebude automaticky volaná. V závorce je tentýž parametr, se kterým byla původní (a teď i naše nová) metoda volaná. Takže nyní, místo aby systém zavolal onCreate() v nadřazené třídě, zavolá ten, který je v MainActivity. A onCreate() z MainActivity nejdřív zavolá onCreate() z nadřazené třídy, tedy z AppCompatActivity. Potom už může zpracovat jakýkoliv další potřebný kód. V našem případě zdrojový kód, který vygeneroval průvodce. Do něj už si můžeme přidat, co potřebujeme. Například breakpoint nebo volání metody pro zobrazení dialogového okna, které potvrdí, že systém metodu po startu aktivity zavolal.



Obrázek 7.2: Metody a události třídy MainActivity

Stejným způsobem můžeme přepsat i všechny ostatní události, které jsou uvedené v životním cyklu aktivity:

```
@Override
public void onStart() {
    super.onStart();
}

@Override
public void onResume() {
    super.onResume();
}

@Override
public void onPause() {
    super.onPause();
}

@Override
public void onStop() {
```

```
        super.onStop();
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
    }

    @Override
    protected void onRestart() {
        super.onRestart();
    }
```

Takto získáme možnost reagovat na každou událost životního cyklu přímo v MainActivity. Všimněte si, že onCreate() byla jedinou metodou, která má vstupní parametr.

Abychom si mohli vyzkoušet, jak jsou jednotlivé metody volané, vytvoříme metodu zobrazitZpravu() s parametry nazev a zprava. Tato nová metoda bude dělat totéž, co v předchozím příkladu dělala metoda kliknutiNaTlacitko(), tedy zobrazí alert, neboli textové okno, které bude mít v záhlaví název předaný parametrem ‚nazev' a text předaný parametrem ‚zprava'. Metodu přidáme do třídy MainActivity:

```
private void zobrazitZpravu(String nazev, String zprava) {
      AlertDialog.Builder builder = new AlertDialog.Builder(MainActivity.this);
      builder.setCancelable(true);
      builder.setTitle(nazev);
      builder.setMessage(zprava);
      builder.setPositiveButton("OK", null);

      builder.create().show();
}
```

Metoda zobrazitZpravu() je tentokrát typu private. Nepotřebujeme totiž, aby byla volaná z jiné třídy než z MainActivity. Parametry nazev a zprava jsou typu string, tj. textový řetězec.

Nyní můžeme upravit libovolnou metodu volanou systémem, například onStart(), tak, aby po spuštění zavolala metodu zobrazitZpravu() s příslušnými parametry:

```
@Override
public void onStart() {
    super.onStart();
    zobrazitZpravu ("Info", "Metoda onStart spuštěna.");
}
```

Po spuštění aktivita zobrazí dialogové okno s informací o volání metody onStart(). Takto můžeme modifikovat všechny události životního cyklu a na koncovém zařízení sledovat, jak a v jakých situacích jsou jednotlivé metody volány.

Zde je však třeba upozornit ještě na jednu okolnost, kterou je dobré mít na zřeteli – zavoláním metody zobrazitZpravu() se aplikace nezastaví. Volání metod je realizované formou asynchronních událostí, takže když například onCreate() zavolá metodu pro zobrazení okna, které by mělo čekat na odkliknutí uživatelem, tak se může hned zavolat metoda onStart(), která svým oknem překryje okno předchozí. Na zařízení se tak nahromadí okna, jejichž postupné odklikávání může působit dojmem volání událostí v opačném, než očekávaném pořadí. Proto je dobré při experimentování doplnit sledování volání jednotlivých událostí ještě breakpointy, které už spolehlivě celou aplikaci zastaví.

Na závěr je potřeba říct, že zde probrané události jsou pouze základní sadou, která je vývojáři k dispozici. Kromě nich má Android řadu dalších, vedlejších nebo podmíněných, které sice nejsou na obrázku 7.1 zakresleny, ale využívat je také lze. V dalších příkladech narazíme například na onPostCreate(), která je vyvolaná po ukončení onStart().

8 Context (kontext)

Toto, tak trochu abstraktní slovo, říká, že všechny objekty aplikace mají vazbu k nějaké její konkrétní části. Jsou k ní logickým způsobem přiřazeny. Objekt může být součástí aktivity, služby, celé aplikace, atd. Kontext objektu potřebujeme znát z několika základních důvodů:

- při vytváření objektu – aby bylo jasné, kam se má objekt hierarchicky zařadit. Pokud ho přiřadíme k aktivitě, tak k němu budeme přistupovat přes aktivitu, a s jejím koncem zanikne i náš objekt. Když ho přiřadíme k aplikaci, zanikne až s celou aplikací (pokud ho pochopitelně nezrušíme samostatně). Objekty při svém vzniku obvykle vyžadují zadání kontextu.

- při přístupu k objektu – jakmile potřebujeme k objektu přistoupit, musíme se nejprve dostat na objekt, který je mu hierarchicky nadřazený, neboli na jeho kontext. Pokud je objektem prvek na formuláři, musíme se nejdřív dostat na formulář, potom až na samotný prvek.

Základní způsoby získání kontextu:

- getContext() - vrátí kontext, ve kterém se běh aplikace právě nachází. Obvykle to bývá aktuální aktivita.

- getApplicationContext() - vrátí kontext celé aplikace. Například pokud potřebujeme přistupovat k souborům ve složce assets.

- getBaseContext() - vrátí kontext z libovolné části aplikace, tedy i z jiné, než která je aktuálně zpracovávaná. Používá se ve spojení s třídou ContextWrapper, která obaluje existující kontext.

- this - dostupné pouze uvnitř aktivity, a říká, že použijeme kontext právě této aktivity.

Příklad různých způsobů použití kontextu při zakládání objektu typu Toast (zobrazení notifikace):

```
Toast.makeText(this, "test", Toast.LENGTH_SHORT).show();
Toast.makeText(getContext(), "test", Toast.LENGTH_SHORT).show();
Toast.makeText(getBaseContext(), "test", Toast.LENGTH_SHORT).show();
Toast.makeText(getApplicationContext(), "test", Toast.LENGTH_SHORT).show();
```

První příklad se typicky používá v aktivitě, kdy je potřeba, aby dialogové okno bylo svázané právě s touto aktivitou. Druhý příklad využijeme ve vlastní třídě, kde není kontext zřejmý a nelze použít this. Třetí způsob se hodí například, pokud toast otvíráme z události (třeba po kliknutí na tlačítko) a chceme ho vztáhnout k aktivitě. A poslední příklad použijeme třeba když potřebujeme aby toast zůstal zobrazený, i když aktivitu mezitím ukončíme – navážeme ho tedy na kontext celé aplikace a jeho život zůstane na aktivitě nezávislý.

9 Struktura projektu v Android Studiu

Dosud jsme se při práci spokojili s grafickým editorem Android Studia a s několika málo ovládacími prvky. S tím si ale nemůžeme vystačit dlouhodobě, práce na projektu vyžaduje zásahy do různých souborů napříč celým projektem. Všechny soubory, které tvoří aplikaci, jsou dostupné v okně projektu (na obrázku 5.4 označené ❶). Projdeme si ho nyní podrobněji.

Okno projektu tvoří ve své podstatě stromovou strukturu, která má v kořeni dvě složky – app a Gradle Scripts. Gradle je systém pro automatické sestavení aplikace, využitý Android Studiem. Slouží k tomu, aby nebylo nutné ručně přemísťovat jednotlivé soubory projektu a prostřednictvím příkazové řádky je kompilovat. Scripts jsou v okně projektu proto, aby je uživatel mohl případně upravit a doplnit o vlastní příkazy, které se při sestavování provedou. Toho zatím nevyužijeme, Gradle můžeme nechat bez dalších úprav, tak jak je přednastavený.

Vše podstatné se nachází ve složce app.

Do struktury projektu je možné přidávat další soubory kliknutím pravým tlačítkem myši a volbou New z kontextového menu. Android Studio nově vzniklé soubory samo zařadí v adresářové struktuře na správné místo, podle toho, kam logicky spadají.

Struktura projektu je zobrazena přesně tak, jak je uložena na disku. Jednotlivé adresáře nebo soubory tedy není bezpodmínečně nutné editovat v Android Studiu, lze s nimi pracovat i ve fyzickém adresáři projektu.

Dostat se snadno a rychle k projektovým složkám a souborům na disku, lze v okně projektu kliknutím na složku či soubor pravým tlačítkem myši a z kontextového menu volbou *Show in Explorer.*

9.1 Manifest

Složka *app* → *manifests* obsahuje jeden jediný soubor – AndroidManifest.xml. Je povinný, nelze ho přemísťovat ani přejmenovávat a bez nadsázky jde o nejdůležitější soubor celé aplikace. Obsahuje základní informace, které jsou při spuštění předány operačnímu systému Android. Díky tomuto souboru se koncové zařízení dozví, jak se aplikace jmenuje, jakou má mít ikonu, jaké obsahuje aktivity, která přístupová oprávnění k prostředkům operačního systému požaduje atd.

Po vygenerování aplikace průvodcem je manifest poměrně jednoduchý, má něco kolem 20 řádků se základním nastavením aplikace. Formát souboru je XML, to znamená, že jsou nastavení zapsána strukturovaně a je potřeba při úpravě souboru dodržovat příslušné konvence. Chyba v manifestu může pochopitelně snadno způsobit nefunkčnost celé aplikace.

Obrázek 9.1: Základní nastavení aplikace v souboru manifestu

Prvky manifestu, viz obr. 9.1:

❶ Záhlaví XML souboru, informace o verzi a použitém kódování znaků. Toto je standardní součást každého XML souboru.

❷ Element manifest, obsahuje základní informaci o balíčku (package), ale jeho hlavní funkcí je vyznačení bloku informací manifestu, uzavřeného mezi elementy `<manifest …>` a `</manifest>`.

❸ Nastavení oprávnění aplikace, podrobněji v následující podkapitole.

❹ Blok informací o aplikaci, uzavřený mezi elementy `<application …>` a `</application>`.

❺ Nastavení aplikace, její název, ikony atd.

❻ Informace o aktivitách aplikace. S každou další aktivitou, obsaženou v aplikaci, sem Android Studio doplní blok informací.

9.2 Oprávnění

Android aplikace jsou spuštěné v prostředí takzvaného sandboxu. To znamená, že nemají přístup k dalším aplikacím a pokud si s nimi chtějí vyměňovat data, musí to udělat nějakým transparentním způsobem, například přes sdílené úložiště. Je to tak z důvodu bezpečnosti. Oddělení aplikací brání zejména možnému šíření virů.

Druhou, pro nás významnější vlastností sandboxu je fakt, že aplikace zejména nemají standardně přístup k některým službám systému. Toto oddělení od citlivých služeb rovněž zabraňuje jejich zneužití. Citlivé služby jsou takové, které by mohly uškodit uživateli. Třeba tím, že budou předávat jeho polohu, odesílat informace, pořizovat záznamy a tak podobně. V praxi to znamená, že aplikace, která chce například odesílat SMS, musí o toto privilegium požádat uživatele buď při instalaci, nebo v průběhu činnosti.

Jak je to zařízené? Autor aplikace má možnost přidat do manifestu žádost o přidělení přístupu mimo sandbox, pomocí elementu uses-permission. Při instalaci je uživatel vyzván, aby povolil

aplikaci přístup k mikrofonu. Pokud souhlasí, může aplikace zvuk nahrávat, v opačném případě skončí pokus o nahrání zvuku vyvoláním výjimky. Programátor by měl s takovou eventualitou počítat a případ, že uživatel přístup nepovolí, mít ošetřený.

Oprávnění jsou rozdělena do několika úrovní:

- běžná (normal), aplikace požaduje přístup mimo svůj sandbox, ale riziko zneužití je malé. U této úrovně není nutné žádat uživatele o povolení, aplikaci ho přidělí systém automaticky. Stačí požadavek uvést v manifestu. Jde například o práva přístupu k zařízením Bluetooth, NFC, čtečce otisků prstů, zjišťování dostupnosti internetu, možnost nastavit alarm a tak dále.

- podpis (signature), pokud dvě aplikace obsahují shodný a platný podpisový certifikát, systém jim požadované povolení automaticky udělí.

- nebezpečné (dangerous), musí uživatel vědomě povolit, jinak aplikace přístup nezíská. Sem patří prakticky vše, co umožní sledovat aktivity uživatele, jeho data, nebo cokoliv, co může způsobit využití zpoplatněných služeb. Typicky tedy přístup ke kameře, mikrofonu, GPS, ke kontaktům, kalendáři, SMS zprávám atd.

Jelikož je oprávnění úrovně ‚nebezpečné‘ větší množství a pro uživatele by jednotlivá práva nemusela být přehledná a jasná, jsou dále dělena do několika kategorií.

Kategorie oprávnění:

- Fotoaparát

- Kalendář

- Kontakty

- Mikrofon

- Poloha

- SMS

- Telefon

- Tělesné senzory

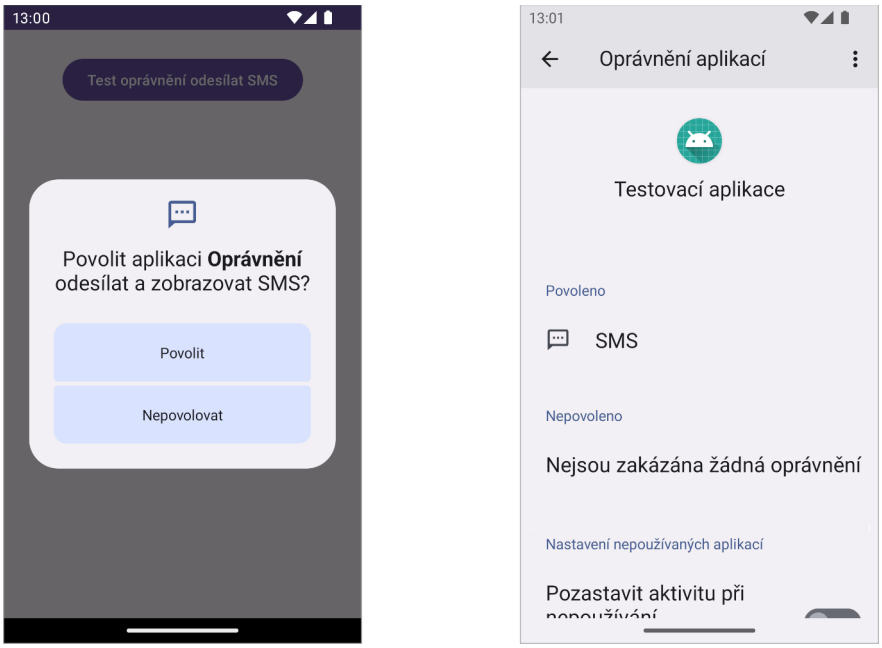
- Úložiště

Uživatel při schvalování vidí pouze kategorii a tu schválí. O jaké konkrétní oprávnění jde, se tedy nemusí starat.

Pokud autor aplikace přidá do manifestu například žádost o povolení odeslání SMS:

```
<uses-permission android:name="android.permission.SEND_SMS" />
```

SEND_SMS je jedno z několika možných oprávnění kategorie SMS. Uživatel však dostane při instalaci aplikace ke schválení celou kategorii SMS. O schválení si také aplikace musí požádat v okamžiku přístupu vyžadujícího oprávnění (tj. v okamžiku, kdy aplikace odesílá SMS) a použití může rovněž řídit v nastaveních zařízení u konkrétní aplikace:



Obrázek 9.2: Nastavení oprávnění pro odesílání textových zpráv

Způsob, jak při programování zjistit, která oprávnění aplikace vlastně potřebuje, není striktně daný. Někdy si to uvědomíme automaticky, z logiky věci, jindy nás upozorní Android Studio a občas se na to přijde, až když při ladění něco nefunguje. Nejspolehlivější je však online dokumentace.

Webové stránky s dokumentací jsou na adrese developer.android.com a najdeme je nejsnáze pomocí vyhledávače Google, zadáním klíčových slov „Android Developer Permissions Overview“.

Programově lze udělat některé testy oprávnění. Jednak je to ověření, jestli uživatel aplikaci oprávnění udělil:

```
if (ContextCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS)!=
                                        PackageManager.PERMISSION_GRANTED) {
    // uzivatel opravneni zatim neudelil
}
```

V případě, že uživatel zatím oprávnění neudělil, lze otestovat, jestli je možné o toto oprávnění požádat za běhu aplikace:

```
if (!ActivityCompat.shouldShowRequestPermissionRationale(this,
                                        Manifest.permission.SEND_SMS)) {
    // muzeme uzivatele o opravneni pozadat
}
```

Pozor na trochu nečekanou logiku. Metoda shouldShowRequestPermissionRationale() vrací true, když uživatel už dříve dialog s povolením odmítl a aplikace by sama neměla o povolení žádat

znovu. False naopak znamená, že je možné dialog zobrazit. Pokud tedy můžeme o oprávnění požádat, lze to udělat takto:

```
int OPRAVNENI_ZADOST = 1;
ActivityCompat.requestPermissions(this,
            new String[]{Manifest.permission.SEND_SMS}, OPRAVNENI_ZADOST);
```

Konstanta OPRAVNENI_ZADOST je unikátním ID požadavku. Výsledkem tohoto konkrétního příkladu je žádost systému o povolení přijímat a číst zprávy SMS.

9.3 Požadované funkce

S oprávněními v manifestu souvisí deklarace požadovaných funkcí, takzvaných uses-feature. Jde o výčet hardwarových nebo softwarových funkcí, které aplikace potřebuje ke své činnosti. Každá se píše na samostatný řádek. Například pokud má aplikace řádně pracovat pouze s kompasem, uvedeme v manifestu:

```
<uses-feature android:name="android.hardware.sensor.compass"
                                            android:required="true" />
```

Tato informace slouží Google Play Store k tomu, aby mohl nabídnout uživateli pouze aplikace vhodné pro jeho zařízení. Pokud je parametr required nastavený na true, jako v příkladu, tak je aplikace k dispozici jen zařízením disponujícím kompasem. Kdyby byla hodnota nastavena na false, obchod by aplikaci zpřístupnil i zařízením bez kompasu, ale uvedl by kompas v seznamu funkcí, které aplikace potřebuje.

Požadované funkce nejsou blokující, aplikaci lze nainstalovat a spustit, i když danou podmínku nesplňuje. A je na vývojáři, aby zařídil její vhodné chování v případě, že některá z funkcí opravdu schází.

9.4 Java

Adresář *app → java* obsahuje všechny zdrojové kódy, které mají souvislost s projektem. Název svádí k tomu, že by adresář měl být určený výhradně pro Java kód, ale stejně tak jsou zde umístěny zdrojové kódy napsané v Kotlinu. Je možné, že v budoucnu dojde na změnu názvu složky, ale aktuálně se jazyk nerozlišuje a všechen kód zkrátka patří sem.

Obsahem jsou především třídy, které se vztahují k aktivitám, ty tam umístí sám průvodce. Ale pochopitelně tam také programátor přidává všechny vlastní soubory s výkonným kódem. Při vygenerování kostry aplikace průvodcem jsou zde tři soubory – první patří k vygenerované aktivitě, druhý je připravený pro testování aplikace programátorem a do třetího se umísťují takzvané unit testy (testy pro automatické ověření funkčnosti). Každá třída je uzavřena v balíčku (package) s názvem, který odpovídá jmennému prostoru (namespace) aplikace.

Obsah adresáře *app → java* je takovým motorem celé aplikace. Vše, co aplikace dělá, ať už viditelným způsobem nebo ‚pouze uvnitř', najdeme zde, ve formě zdrojových souborů.

V Android Studiu v okně projektu lze do adresáře *app → java* přidávat další balíčky a soubory. Je ale potřeba dát pozor na zachování správného názvu jmenného prostoru. Při přesouvání souborů mezi balíčky to většinou není velkým problémem. Stačí soubor myší přetáhnout z jednoho balíčku do jiného a Android Studio samo nabídne funkci Refactor, která už se o změnu namespace postará.

9.5 Res

Res je zkratkou pro ‘resources’ neboli zdroje. A zatímco adresář *app → java* obsahoval pohonnou jednotku aplikace, v *app → res* je uložený její kompletní design. Najdeme zde všechny soubory, které neobsahují Java kód, ale jsou přesto potřeba k tomu, aby mohla být aplikace na displeji vykreslena. Zdroji jsou tedy myšleny především layouty formulářů ve formátu XML, obrázky, texty a tak podobně. Základní složky jsou tyto:

drawable – obrázky. Pokud aplikace pracuje s obrazovými soubory v některém z podporovaných formátů, měly by být uloženy právě sem. Obrázky mohou být v bitmapovém nebo vektorovém formátu. Případně mohou mít formát XML souboru, který obraz popisuje (podrobněji v kapitole 9.7 XML obrázky).

layout – formuláře aktivit. Po kliknutí se zobrazí v editoru, ale ve skutečnosti jde o XML soubory, v nichž je definované, jak má formulář vypadat. Editor je pouze převádí do grafické formy, aby se s nimi uživateli lépe pracovalo. Přesto je často potřeba přepnout editor do režimu XML (ikonky Code/Split/Design v pravém horním rohu editoru) a modifikovat přímo XML elementy těchto souborů.

mipmap – ikony aplikace. Jsou v několika rozměrech, ikona pro každou velikost má svůj vlastní soubor.

values – soubory textů a hodnot, které jsou využity při běhu aplikace. Například ve strings.xml jsou uloženy texty. Je to tak z toho důvodu, že psát texty rovnou do zdrojového kódu není úplně praktické, při přepnutí jazyka potřebujeme, aby aplikace okamžitě nahradila texty cizojazyčným ekvivalentem. V Java/Kotlin kódu jsou tedy pouze odkazy a samotné texty jsou uložené ve strings XML. Podobnou funkci mají i ostatní soubory v adresáři values.

9.6 Barvy

Při vytváření aplikace je jednou ze základních vlastností barevné provedení uživatelského rozhraní. Nastavují se barvy textu, komponent, pozadí, položek menu a seznamů, změnou barvy se uživateli dává najevo průběh akce, například stisknutí tlačítka.

Několik jednoduchých způsobů, jak nastavit barvu textu v textovém poli, pomocí funkcí dostupných v jazyce Java:

```
TextView textovePole = findViewById(R.id.textView);
textovePole.setTextColor(Color.RED);
textovePole.setTextColor(0x40FF0000);
textovePole.setTextColor(Color.parseColor("#FF0000"));
textovePole.setTextColor(Color.argb(128, 255, 0, 0));
```

Barvy lze mít v aplikaci nastavené takto napevno, v kódu, není to ale nejlepší řešení. Stejně jako v případě textů, i u barevného schématu může snadno vzniknout potřeba jednotné úpravy. Proto je praktické, mít nastavení barev centrálně v jednom souboru. K tomu se za chvíli dostaneme.

Druhou formou je zápis do XML souboru. Zejména do designu formuláře, ale i do mnoha dalších souborů, které v XML formátu popisují obrázky, ikonky, nastavují barevná témata nebo definují změny barev při provádění uživatelských akcí.

V XML se pro vyjádření barevných složek používá takřka výhradně šestnáctkový zápis. Barvy se v XML nastavují pomocí hexadecimální konstanty složené z RGB složek (R=červená, G=zelená, B=modrá). Neboli – jde o tři hexadecimální čísla z intervalu 0x00-0xff (čili dekadicky 0-255), která následují po sobě a před kterými je uvedený znak #. Například tedy #ff0000 je vyjádření

červené barvy, #00ff00 zelené, atd. Skládáním jednotlivých složek vznikají další barvy. Například #333333 je tmavě šedá.

Kód barvy může nést také informaci o alfa kanálu, neboli míře průhlednosti. Ta má stejně jako jednotlivé barevné složky hexadecimální rozsah 00-ff. Průhlednost se umísťuje na první místo před barevné složky, tedy například prvek s barvou #80ff0000 bude červený a bude mít 50% průhlednost (0x80 je dekadicky 128, neboli polovina rozsahu). Čím nižší hodnota, tím je výplň průhlednější.

Hexadecimálně vyjádřené a zapsané barvy ale mají jednu nevýhodu. Špatně se nahrazují. Což může být problém, když je potřebujeme změnit barevnost, nebo opravit barevný odstín a najednou zjistíme, že jsou barvy natvrdo zapsané v různých souborech a nejde je úplně snadno změnit. Řeší se to souborem app/res/values/colors.xml, ve kterém jsou definované základní barvy, identifikované textovými názvy, se kterými lze potom pracovat bez nutnosti používání hexadecimálního zápisu kódu barvy.

Základní barvy, plná bílá a plná černá, jsou v souboru colors.xml definované takto:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
</resources>
```

K těmto barvám si vývojář může pochopitelně přidat libovolné vlastní. Stačí přidat element <color … >, se jménem a šestnáctkovým zápisem barvy. A následně se na ně odkázat, například z themes.xml, při definici primární barvy aplikace, následujícím způsobem:

```
<item name="colorPrimary">@color/black</item>
```

Kromě barev definovaných v souboru colors.xml obsahuje Android a systémové knihovny ještě další barvy, na které se lze rovněž odkazovat pomocí textového identifikátoru, aniž by je bylo nutné někde definovat. Například:

```
<item name="colorPrimary">@color/material_dynamic_neutral95</item>
```

Pochopitelně je možné k barvám ze souboru colors.xml přistupovat přímo z programového kódu. A k barvám ze systémových knihoven se lze dostat přes jejich namespace:

```
int bilaBarva = getColor(R.color.white);
int sedaMaterialBarva = getColor(
            com.google.android.material.R.color.material_dynamic_neutral95);
```

9.6.1 Selector

Ve struktuře projektu je možné, ve složce app/res/drawable/, narazit také na soubor, který slouží k přepínání barev na základě uživatelské akce. Zpravidla se jmenuje itemcolors.xml, nebo podobně (název není pevně daný) a obsahuje XML blok <selector … >, jehož každá položka má přiřazenu nějakou barvu a současně akci, při které má být daná barva použita:

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@color/material_dynamic_neutral95"
                                                android:state_pressed="true" />
    <item android:drawable="@color/material_dynamic_neutral60"
                                                android:state_selected="true" />
    <item android:drawable="@android:color/white" />
</selector>
```

Důležité jsou parametry atributů u jednotlivých položek <item … >. Nejdříve je definována barva. Zde tedy odstíny šedé, neutral95, neutral 60 a bílá. Akce, ke které se položka vztáhne, je daná parametrem state, tedy state_pressed pro kliknutí, state_selected pro vybrání položky a bez state, pro klidový stav. Ještě se parametru state nastaví boolean hodnota, true pro potvrzení, že bude položka aktivní při dané akci. False naopak říká, že bude v klidovém stavu položka aktivní a až když k dané akci dojde, tak naopak platit přestane.

A jak s tím vším naložit? Tento soubor může v XML definici komponenty nebo formuláře nahradit barvu. Například místo standardním způsobem nastaveného bílého podbarvení:

```
android:background="@android:color/white"
```

je možné odkázat na selector ze souboru itemcolors.xml:

```
android:background="@drawable/itemcolors"
```

Tím dáme Androidu na vědomí, že objekt bude mít pozadí sice v barvě bílé, ale pokud ho někdo na displeji vybere, tak světle zešedne a při kliknutí se ztmaví do tmavší šedé. Jakmile uživatel objekt uvolní, tak se vrátí zpět k bílé barvě.

Tohoto jednoduchého způsobu dynamické změny barvy se využívá zejména u tlačítek a různých generovaných seznamů, kdy je potřeba uživateli vizuálně označit položku, se kterou pracuje. Využít se ale dá prakticky kdekoliv, kde je nějaká podbarvená plocha, která má reagovat na uživatelskou akci

9.7 XML obrázky

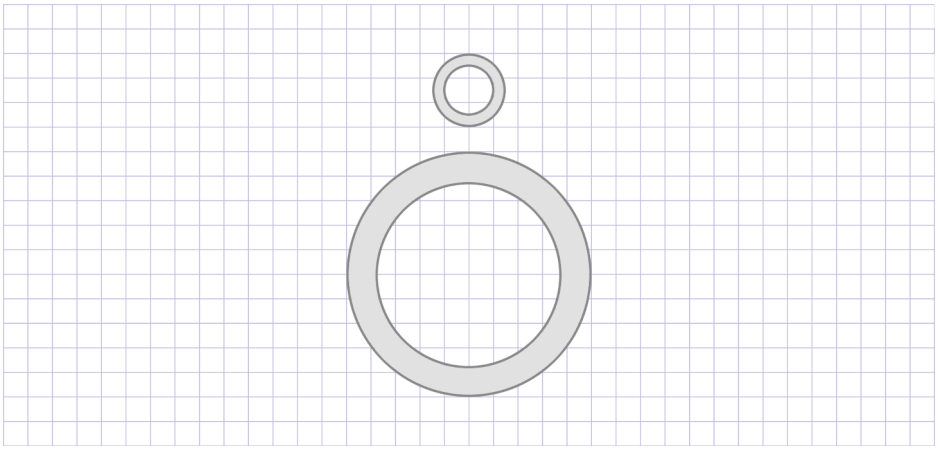
Android umožňuje, kromě klasických grafických souborů, použití jednoduchých vektorových prvků zapsaných v XML formátu. Svoje místo mají ve složce app/res/drawable/ a používají se na ikonky aplikací, pozadí na formulářích nebo na jednoduché grafické prvky uživatelského rozhraní. Mají stejnou výhodu jako jiné vektorové formáty. Malý datový objem, zobrazují se rychle a i při výrazném zvětšení jsou stále dokonale vyhlazené. Nevýhodou je zejména neexistence dobrých grafických editorů, které by uměly vygenerovat obrázek v tomto formátu, takže jsou vývojáři často závislí na tom, jak si obrázek sami navrhnou a do XML formátu přepíší.

Podporovaných elementů, ze kterých se dá složit výsledný grafický objekt, není mnoho. Čára, obdélník, kruh a ovál. Lze jim nastavit pozici v obrázku, velikost, barvu, výplň (včetně barevného přechodu), obdélníku můžeme navíc nastavit, jak mají vypadat rohy. A s tím už se dají dělat malé zázraky.

9.7.1 XML obrázek, 1. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 09_Struktura_projektu
projekt XmlObr

První takový obrázek, se kterým se vývojář setká, je pravděpodobně ikona aplikace, vygenerovaná Android Studiem při založení projektu a uložená ve složce drawable. Jenže ta je pro seznámení se strukturou obrázku příliš složitá. Můžeme si ale zkusit udělat jednoduchý vlastní příklad. Třeba kulaté šedé tlačítko, které vložíme do formuláře.



Obrázek 9.3: Návrh XML obrázku, ve dvou velikostech

Co je potřeba udělat?

- vytvořit XML soubor ve složce drawable

- v XML souboru popsat vzhled budoucího tlačítka

- umístit na formulář obrázkové tlačítko a nastavit mu vzhled definovaný XML obrázkem

Soubor založíme tak, že v okně projektu nad složkou drawable vybereme v kontextovém menu položku New → Drawable Resource File. Nastavíme mu vhodný název, třeba tlacitko1. Vznikne soubor tlacitko1.xml, v němž pomocí XML jazyka popíšeme vzhled tlačítka. Prozatím se nebudeme pokoušet o nic složitého. Řekněme, že tlačítko bude vypadat jako tmavě šedý prstenec.

Element pro nakreslení geometrického tvaru se jmenuje <shape ...> a to, že má jít o prstenec, se mu sděluje pomocí atributu android:shape="ring". Shape má tuto základní strukturu:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="ring"
    android:useLevel="false">
    <solid android:color="#555555" />
</shape>
```

Atributů má tento prvek mnohem víc, ale my si vystačíme s několika základními. Pomocí android:useLevel="false" nastavíme, že se má vykreslit celý prstenec. Hodnota true by se použila v případě, že je potřeba vykreslit jenom část prstence, například když má sloužit jako indikátor průběhu (progress bar). Zajímat nás bude vnořený element <solid android:color ...>, který nastaví barvu prstence. Víc toho teď k popisu vzhledu prstence není potřeba. XML kód vložíme do připraveného souboru tlacitko1.xml.

Aby tlačítko bylo interaktivní, hodí se nastavit, aby během stisknutí mělo trochu jiný vzhled. Třeba jinou barvu. Připravíme si proto stejným způsobem ještě jeden soubor, tlacitko2.xml, který bude obsahovat XML kód s jediným rozdílem, prstenec bude mít nastavenou světle šedou barvu:

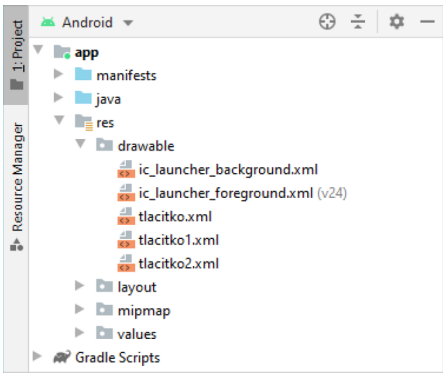
```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="ring"
    android:useLevel="false">
    <solid android:color="#555555" />
</shape>
```

Prvek ImageButton, který se pro vykreslení grafického tlačítka používá, s něčím takovým počítá a pokud mu nastavíme XML soubor se dvěma obrázky, bude je přepínat podle toho, jestli je tlačítko v klidu, nebo jestli ho uživatel stiskl. Které obrázky použít, se definuje ve zvláštním XML, které si nazvěme tlacitko.xml. Soubor bude obsahovat další XML strukturu s elementem selector, odkazující na vzhled tlačítka v různých situacích:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:state_pressed="false"
        android:drawable="@drawable/tlacitko1" />
    <item
        android:state_pressed="true"
        android:drawable="@drawable/tlacitko2" />
</selector>
```

Funkce selectoru je na první pohled jasná. Popisuje, které obrázky se mají v různých situacích vykreslit. V příkladu je pouze jeden atribut, android:state_pressed. Jestliže je tlačítko stisknuté, použije se obrázek s nastavenou hodnotou true, v klidovém stavu bude vykreslený obrázek označený hodnotou false. Atribut android:state_pressed je pouze jedním z mnoha možných. Podobně lze nastavit rozdílný vzhled, pokud je tlačítko vybrané, v případě, že je součástí zaškrtávacího seznamu, tak může vypadat jinak jako zaškrtnuté a nezaškrtnuté, může se vizuálně lišit když je aktivní nebo naopak neaktivní a tak podobně.

Struktura souborů by v okně projektu měla vypadat jako na obrázku 9.4.

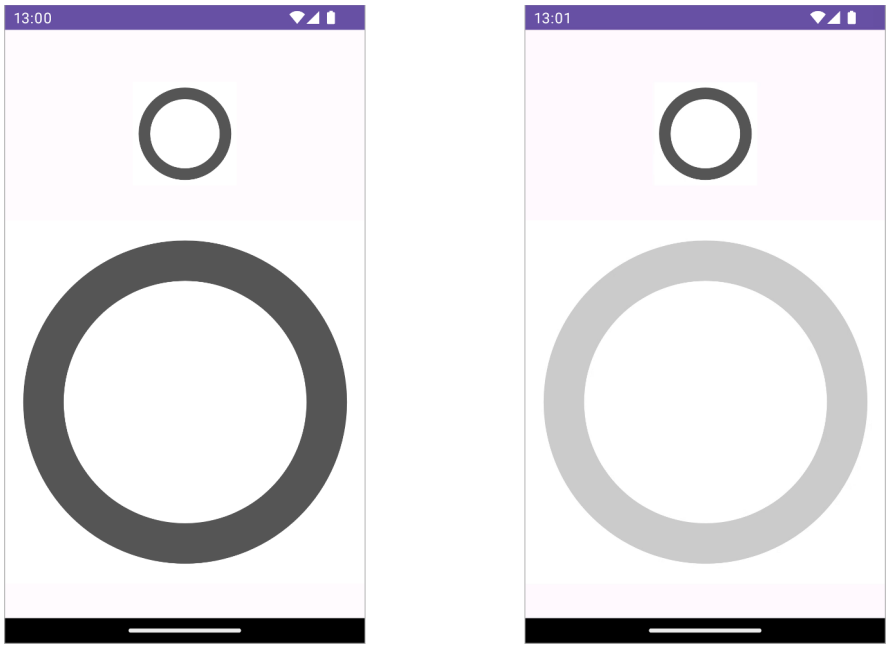


Obrázek 9.4: Soubory s XML definicí tlačítek, v okně projektu

Do formuláře activity_main.xml vložíme v grafickém editoru Android Studia tlačítko ImageButton. Zvolíme vhodné umístění a velikost. Do atributu srcCompat nastavíme odkaz na XML soubor tlacitko: „@drawable/tlacitko“. Pod ně stejným způsobem vložíme druhé tlačítko se

stejným zdrojem srcCompat, ale nastavíme mu větší velikost. Získáme tak dvě tlačítka pod sebou, která budou tvořena XML obrázkem. Na výsledku je vidět, že jak v případě malého, tak i u velkého tlačítka je prstenec hladce vykreslený. To je základní vlastnost vektorového zobrazení. V případě použití bitmapového obrázku bychom byli vždy závislí na tom, jestli jeho velikost odpovídá zobrazovaným rozměrům.

Výsledná aplikace by po spuštění měla vypadat jako na obrázku 9.5.

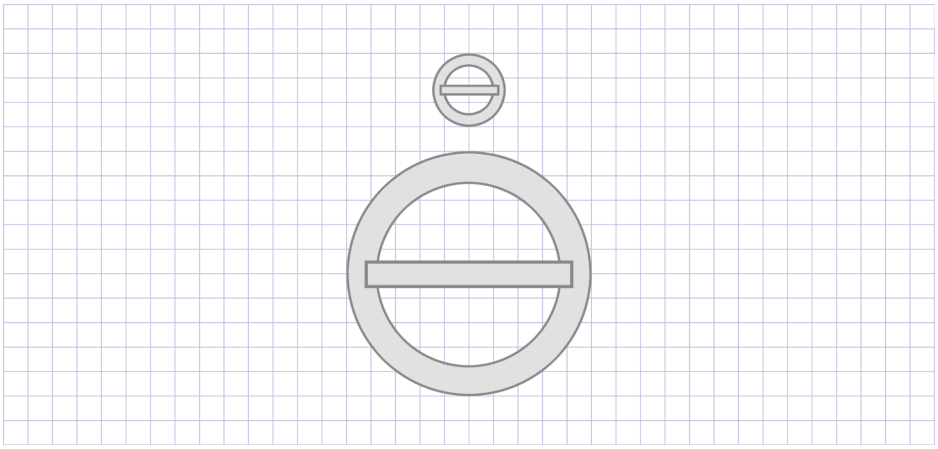


Obrázek 9.5: Tlačítka se vzhledem definovaným pomocí XML. Vlevo jsou tlačítka v klidu, vpravo je větší tlačítko stisknuté.

9.7.2 XML obrázek, 2. příklad

V předchozím příkladu byl obrázek definovaný nejjednodušším možným způsobem. Tvořil ho jediný element, prstenec. S tím si ale většinou nevystačíme, častěji budeme potřebovat obrázek složit z několika elementů.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 09_Struktura_projektu
projekt XmlObrazek2



Obrázek 9.6: Návrh XML obrázku, složený z prstence a obdélníku

Není to nijak složité. Upravíme soubor tlacitko1.xml. Stačí použít prvek layer-list, který obsahuje položky item. Do těch vložíme už známý element <shape ...> s definicí grafického tvaru. Prvky item můžeme podle potřeby opakovat:

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:left="30dp" android:right="60dp">
        <shape
            android:shape="rectangle">
            <size
                android:width="300dp"
                android:height="50dp" />
            <solid android:color="#555555" />
        </shape>
    </item>

    <item>
        <shape
            android:shape="ring"
            android:useLevel="false">
            <solid android:color="#555555" />
        </shape>
    </item>

</layer-list>
```

První položka item obsahuje grafický tvar rectangle, neboli obdélník. Ve druhé je prstenec, stejný jako v předchozím příkladu. Složení těchto dvou položek v prvku layer-list dá dohromady očekávaný obrázek.

Stejně upravíme i soubor tlacitko2.xml, pouze s využitím světlejší barvy. Soubor tlacitko.xml může zůstat v původním stavu, mechanismus přepínání obrázků při kliknutí na tlačítko se nemění. Totéž platí pro formulář activity_main.xml, může zůstat stejný jako v předchozím příkladu. Výsledná aplikace by po spuštění měla vypadat jako na obrázku 9.7.

Obrázek 9.7: Tlačítka se vzhledem definovaným pomocí XML. Vlevo jsou tlačítka v klidu, vpravo je větší tlačítko stisknuté.

10 Ladění aplikace, logování

Aby bylo možné efektivně sledovat chování aplikace během ladění nebo v okamžiku výskytu chyby, jsou k dispozici nástroje pro zápis do logovacího souboru. Pro tyto účely má Android Studio okno Logcat, které obsahuje výpis logů aktivní aplikace. Okno se dá najít v záložce ve spodní části Android Studia, nebo ho lze otevřít přes menu View → Tool Windows → Logcat.

V okně Logcat je potřeba nastavit zařízení, z jehož logů chceme záznamy číst a potom už stačí pouze sledovat, jaké záznamy spuštěná aplikace vypisuje.

Jelikož nás ale zajímají především vlastní informace vypisované za běhu, musíme do aplikace doplnit instrukce pro logování. Ty poskytuje třída android.util.Log. Zápis do logovacího souboru vypadá například takto:

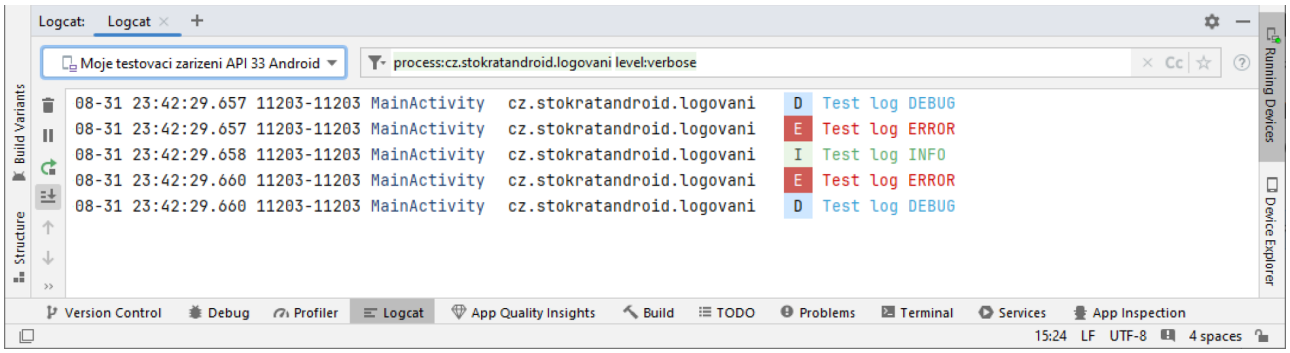
```
Log.d ("MainActivity", "Test log DEBUG");
```

Tato instrukce zapíše na úrovni DEBUG, pod značku MainActivity, text „Test log DEBUG“. A my ho uvidíme v okně Logcat. Úrovně pod kterými můžeme zapisovat se volí písmenem označujícím metodu. Máme následující možnosti, seřazené od nejvyšší po nejnižší úroveň:

```
Log.e(String, String) // error (chyba)
Log.w(String, String) // warning (varovani)
Log.i(String, String) // information (informace)
Log.d(String, String) // debug (ladeni)
Log.v(String, String) // verbose (podrobny)
```

Prvním parametrem logovací metody je označení záznamu. Obvykle stačí použít název třídy, případně metody. Druhým parametrem je text logované zprávy.

Pokud bychom zavolali několikrát po sobě výšeuvedený příklad, pod různou úrovní logování, výsledek by v Logcat okně vypadal takto:



Obrázek 10.1: Zobrazení záznamů s různou úrovní logování v okně Logcat

V okně Logcat si lze nastavit filtr. Se syntaxí pomůže klávesová zkratka CTRL+mezerník. Hodí se nastavení úrovně, takže pokud budeme při práci s aplikací zapisovat záznamy pod úrovní info, nemusíme se probírat velkým množstvím záznamů generovaných na úrovni debug nebo verbose. Jakmile přepneme filtrování na level:info, budou se zobrazovat pouze záznamy úrovně info a vyšší.

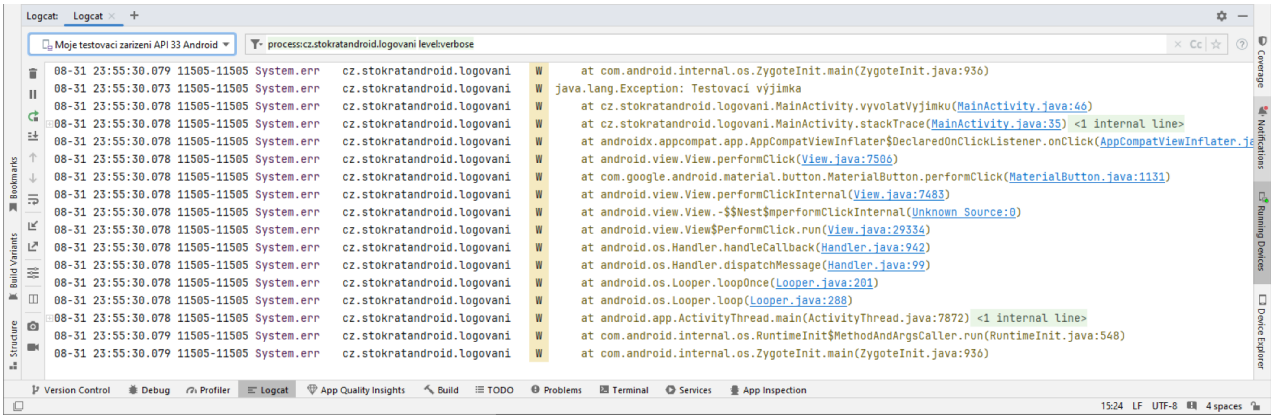
10.1 Ladění aplikace, Stack Trace

Při vyvolání výjimky potřebujeme obvykle zjistit maximum informací nejen o tom, co bylo příčinou chyby, ale také v jakém okamžiku zpracování k ní došlo. Jelikož je v okamžiku pádu už obvykle pozdě začít zapisovat do logu, jakou cestou se aplikace postupně ubírala, pomůže výpis zásobníku, neboli Stacku, takzvaný Stack Trace.

Stack Trace je možné vypsat relativně jednoduše pomocí metody printStackTrace(). Výsledek bude obsahovat informace o postupném volání metod, během jejich zanořování. Ukážeme si na jednoduchém příkladu:

```
public void stackTrace (View sender) {
    try {
        // zavolat metodu, ktera vyvola vyjimku
        vyvolatVyjimku();
    }
    catch (Exception ex) {
        // vypsat stacktrace
        ex.printStackTrace();
    }
}

// Metoda umele zpusobi chybu
private void vyvolatVyjimku () throws Exception {
    // vyvolat vyjimku
    throw new Exception("Testovací výjimka");
}
```

Tento kód nejprve ve větvi try metody stackTrace() zavolá další metodu, vyvolatVyjimku() a ta uměle způsobí chybu, jako by došlo k chybě aplikace. Chybu vrátí a metoda stackTrace() ji následně zpracuje ve větvi catch – jednoduše zapíše obsah Stack Trace do Logcatu:



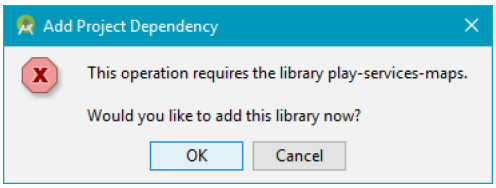
Obrázek 10.2: Výpis chyby v okně Logcat

Výpis se čte odzdola, kde je nejvyšší úroveň. Nahoře je potom metoda v níž došlo k chybě. V našem případě jsou v dolní části systémová volání která nás příliš nezajímají, až druhá odshora je metoda stackTrace() která volá metodu vyvolatVyjimku(). Ta je úplně nahoře, protože tam k chybě došlo. Nad tím už je jenom popis chyby. V tomto případě ale není zajímavý, v rámci testování jsme si jeho znění nadefinovali sami.

11 Závislé zdroje – Dependencies

Během práce na projektech se bude často stávat, že některé komponenty nebo funkce nebudou přímo součástí SDK. V takovém případě je potřeba do projektu přidat externí knihovny, neboli závislé zdroje, anglicky dependencies, které požadovanou funkčnost poskytnou. Stane se to třeba při práci s mapami, kdy bude potřeba připojit knihovnu Play Services Maps (com.google.android.gms:play-services-maps), nebo pro zobrazení reklam bude zapotřebí přidat knihovnu Play Services Ads (com.google.android.gms:play-services-ads).

Že nejsou některé knihovny dostupné se pozná buď tak, že designér neumožní vložit komponentu do formuláře a místo toho zobrazí hlášení:

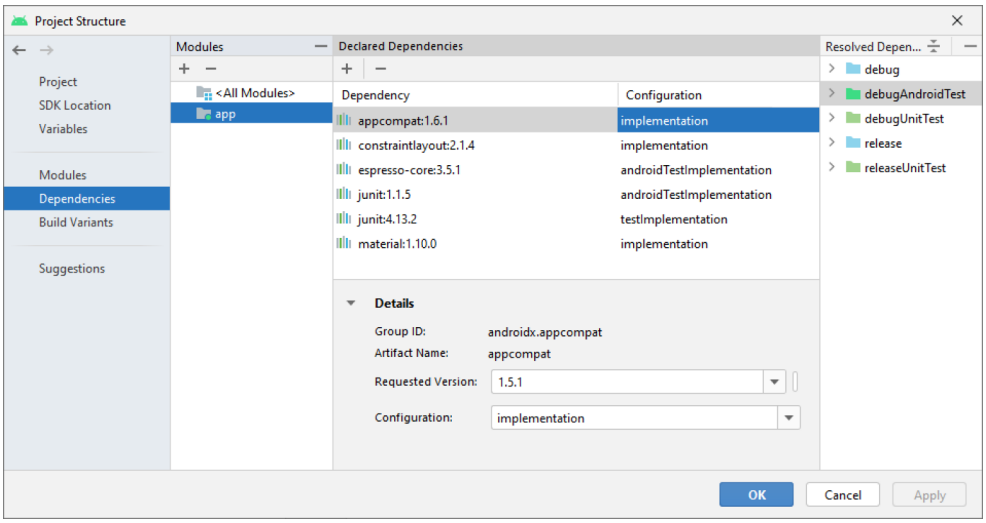


Obrázek 11.1: Varování Android Studia, že není k dispozici nezbytná knihovna

Nebo že nebude možné pracovat v editoru s třídami, které jsou obsaženy pouze v externích knihovnách:

Obrázek 11.2: Editor červeným podbarvením signalizuje, že nemůže pracovat s třídou Play Services Maps, protože v projektu chybí příslušná knihovna

Přidání dalších knihoven se provede v okně Project Structure, tj. menu File → Project Structure, záložka Dependencies a označení modulu app. Některé knihovny přidává už samotný průvodce novým projektem, takže i na samém začátku jich je několik zaregistrovaných:



Obrázek 11.3: Přidání knihovny do projektu

Tlačítkem + přidáme knihovnu požadovaného typu. Takto připojené knihovny budou při kompilaci projektu připojeny do instalačního balíčku, viz kapitola 33.2 APK balíček.

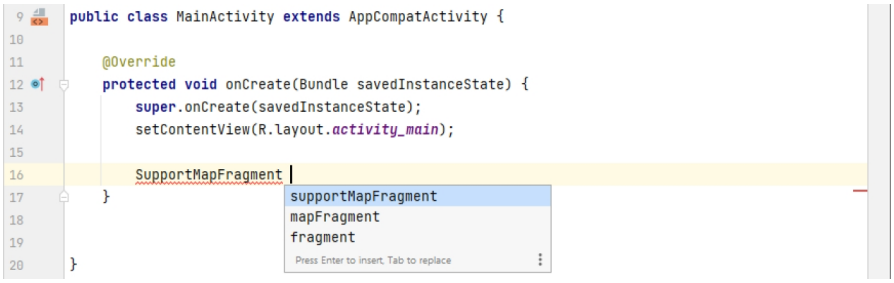
Gradle se spustí při každém přidání nebo odebrání knihovny. Tím se není nutné nijak zneklidňovat, vazby v projektu je potřeba aktualizovat, proto proběhne jeho nové sestavení.

Stejného efektu je možné dosáhnout přímou editací souboru Gradle Scripts → build.gradle (Module.app) v okně Projektu. Záložka Dependencies je ve své podstatě jenom uživatelsky přívětivějším rozhraním pro editaci tohoto souboru.

Obrázek 11.4: Editace Dependencies v souboru build.gradle

Pozor, jelikož projekt používá konfigurační jazyk Kotlin DSL, tak je konfigurace knihoven rozdělena do dvou souborů. V *Gradle Scripts → build.gradle.kts (Module :app)* je výčet použitých knihoven a v *Gradle Scripts → libs.versions.toml (Version catalog)* jsou jejich verze.

Že je knihovna v pořádku připojena se pozná tak, že není problém komponentu přidat do formuláře a že třídy v kódu jsou standardním způsobem dostupné:



Obrázek 11.5: Třída Play Services Maps je po přidání knihovny opět dostupná

12 Prvky formuláře

V předchozích kapitolách jsme okrajově zmínili některé prvky uživatelského rozhraní, které lze v grafickém editoru vložit do formuláře z palety komponent. Takových prvků je celá řada, s rozličným významem. V paletě komponent jsou seřazeny do skupin podle způsobu užití.

Jednotlivé skupiny není nutné podrobně popisovat, stejně jako není nutné popisovat všechny prvky. Jednak zdaleka ne vše programátor při tvorbě aplikace využije a druhým důvodem je fakt, že tvůrci Androidu množinu prvků stále rozšiřují a skupiny komponent přeorganizovávají.

Projdeme tedy základní prvky, které s nejvyšší pravděpodobností vývojář využije v každé aplikaci. Především tedy textová pole, tlačítka a prvek pro zobrazení obrázků. K dalším komponentám se vrátíme později, ve specializovaných kapitolách.

12.1 Textová pole

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 12_Prvky_formulare
projekt TextovaPole

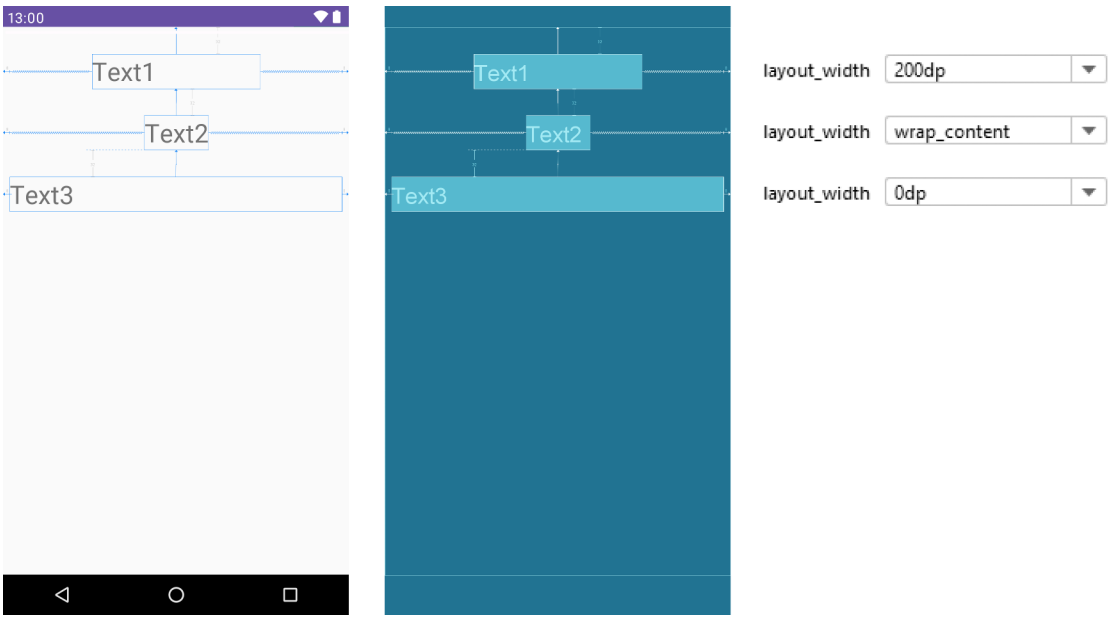
Textová pole jsou dvojího druhu. TextView pro zobrazení textové informace na displeji a EditText k zadávání a editaci textu. Mezi poli je dědičnost - EditText je odvozený od komponenty TextView a až ta je odvozená od třídy View.

Nejprve TextView – toto pole nedělá nic jiného, než že na displeji zobrazí požadovaným způsobem textovou informaci, kterou má přednastavenou nebo kterou mu programově přiřadíme. Po vložení do formuláře má nastavených několik předdefinovaných parametrů. Obvykle první, co vývojář udělá je, že v závislosti na použitém typu formuláře textové pole ukotví do požadované pozice na displeji. Potom nastaví velikost, typ, barvu textu a pochopitelně i samotný text. Pokud ho bude plnit programově, může text nechat zatím prázdný, jenom je potřeba dát pozor, aby se v grafickém editoru pole příliš nezmenšilo v důsledku přednastaveného rozměru wrap_content (přizpůsobení šířky pole podle velikosti obsahu). Špatně by se s ním potom pracovalo. Někdy je vhodné také změnit přednastavené id textového pole, zejména pokud se na ně bude programově odkazovat.

Rozměry pole je potřeba nastavit s rozvahou. K dispozici je několik možností stanovení velikostí, které se nastavují zvlášť v každé ose. Nejjednodušší je nastavit pevnou velikost v jednotkách dp (Density-independent Pixel - jednotka, která určuje velikost pole nezávisle na rozlišení displeje). Tím se zajistí, že si pole bude udržovat předepsaný rozměr bez ohledu na rozložení prvků v okolí. Další možností je nastavit rozměr na wrap_content, neboli automatické přizpůsobování velikosti. Pole bude mít nejmenší možnou velikost, ovšem takovou, aby se do něj vešel požadovaný obsah. Pokud obsah změníme, změní se i velikost pole.

Další alternativou je nastavení match_constraint, které se používá pouze u formuláře typu Constraint Layout. Místo konstanty match_constraint lze použít také rozměr 0dp. Ať už jeden zápis, nebo druhý, oba znamenají totéž – velikost pole nebude pevně nastavena, přizpůsobí se vazbám v Constraint Layoutu. To znamená, že pokud bude mít pole nastavený match_constraint ve vodorovné ose a bude přichycené pouze k levému okraji formuláře, bude se chovat jako jako wrap_content, neboli bude se snažit být co nejmenší. Pokud bude přichycené i k pravému okraji, bude se naopak snažit být v rámci formuláře co největší. Stejně tak, pokud bude přichycené k dalšímu prvku, využije maximum plochy, které mu druhý prvek ponechá. U ostatních typů formulářů je k dispozici konstanta match_parent, která má podobný význam. Také nastavuje maximální možnou velikost pole, v rámci volného prostoru který může toto pole využít.

Nastavení šířky pole na pevnou hodnotu, wrap_content a match_constraint (resp. 0dp) v designéru formuláře, viz obrázek 12.1.



Obrázek 12.1: Textová pole v designéru Android Studia s různým nastavením šířky

Vývojáře bude pravděpodobně zajímat i mnoho dalších parametrů, jako je font, řez písma, zarovnání a tak dále. Toto všechno je ale v grafickém editoru poměrně intuitivně zpracované a s nalezením a účelem parametrů většinou není žádný problém. Po chvilce zkoušení se v nastaveních textového pole bude cítit jako ryba ve vodě. Kdo si nevystačí se základními parametry, může kliknout na přepínač All attributes, který je pod ikonkou ozubeného kolečka nahoře v okně Vlastností a tím přejde k seznamu úplně všech dostupných parametrů komponenty.

Práce ve vizuálním editoru je sice rychlá i pohodlná a programátorovi nic nebrání dělat v něm základní návrh formuláře, tak jelikož je potřeba mít kontrolu nad jednotlivými komponentami a jejich vlastnostmi, určitě se dříve nebo později dostane od grafické reprezentace k přímé editaci xml souboru, který grafický editor vytváří. Pokud si tedy na formulář umístí textové pole, nastaví mu základní parametry a přepne zobrazení na xml (tlačítka vpravo nahoře, v okně grafického editoru), uvidí, že je popsané pomocí xml takto:

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="231dp"
    android:layout_height="38dp"
    android:layout_marginStart="16dp"
    android:layout_marginTop="16dp"
    android:text="Zobrazení textu"
    android:textSize="24sp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Prvním parametrem je id, neboli unikátní název prvku. Naše textové pole se jmenuje textView1. Za chvíli si ukážeme, jak se na ně pomocí tohoto id dá odkázat, když bude potřeba programově změnit některý z parametrů. Další dva řádky stanovují velikost. V tomto případě pevnou, v jednotkách dp, následuje velikost okrajů kolem textového pole a poté je samotný text, který bude pole zobrazovat. Další parametr určuje velikost písma, tentokrát v jednotkách sp (Scale-

independent Pixel - jednotka, která určuje velikost textu bez ohledu na uživatelem nastavenou velikost písma). Poslední dva parametry jsou kvůli umístění v rámci typu formuláře ConstraintLayout.

Na toto textové pole se následně lze v programovém kódu odkázat tímto způsobem:

```
TextView textovePole = (TextView)findViewById(R.id.textView1);
textField.setText(”Nový text”);
```

První řádek najde ve formuláři textové pole s názvem textView1 a uloží odkaz do proměnné textovePole. Druhý řádek prostřednictvím této proměnné nastaví obsah textového pole na „Nový text“. Po zpracování tohoto kódu by tedy pole mělo na displeji zobrazovat text „Nový text“, přičemž všechny ostatní parametry pole zůstanou zachovány.

Dalším druhem textového pole je EditText. Slouží k editaci textu a může mít více podob. Přesněji řečeno, vstup je formátovaný, přičemž maska je daná atributem inputType. Základním typem masky je textPersonName – prostý jednořádkový text, který lze použít nejen pro zadání jména osoby. EditText s touto maskou má v okně komponent název Plain Text. Další možnosti jsou textMultiLine pro víceřádkový prostý text, textPassword pro zadávání hesla, textEmailAddress pro e-mailovou adresu, number pro čísla a tak dále. Každý druh textového pole má vlastní položku v okně komponent. Další typy vstupů je možné nastavit v okně Vlastností v atributu inputType.

Podobně jako pole pro zobrazení textu, tak i pole pro jeho editaci mají nastavení základních parametrů. Jelikož je prvek EditText odvozený od TextView, má s ním shodnou většinu parametrů. Také zde je možné, kliknutím na View all attributes, přepnout jejich zobrazení na všechny dostupné parametry.

V XML vyjádření vypadá textové pole takto:

```
<EditText
    android:id="@+id/editText"
    android:layout_width="231dp"
    android:layout_height="48dp"
    android:layout_marginStart="16dp"
    android:layout_marginTop="16dp"
    android:inputType="textPersonName"
    android:text="Editace textu"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

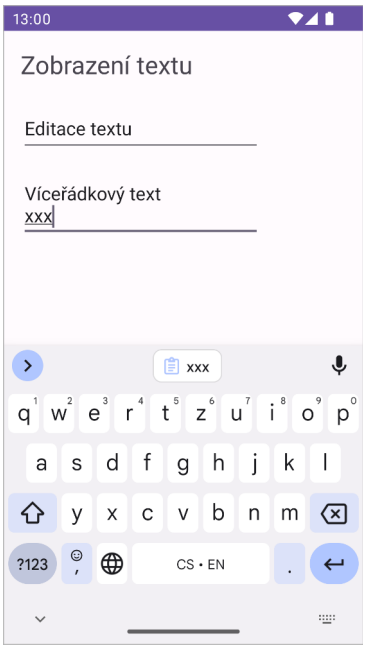
První parametr je opět id. Naše textové pole má id editText. Další dva řádky stanovují velikost pole, následuje velikost okrajů kolem textového pole a poté název masky, neboli inputType, který udává chování pole při zadávání hodnot. Parametr text obsahuje textový řetězec který se zobrazí předtím, než uživatel pole vyplní. Poslední dva parametry jsou kvůli umístění v rámci typu formuláře ConstraintLayout.

Na toto textové pole se následně dá v programovém kódu odkázat tímto způsobem:

```
EditText textovePole = (EditText)findViewById(R.id.editText);
String vstupniText = textovePole.getText().toString();
```

První řádek najde ve formuláři textové pole s názvem editText a uloží odkaz do proměnné textovePole. Druhý řádek prostřednictvím této proměnné přečte obsah textového pole a přiřadí ho do proměnné vstupniText.

Popsaná textová pole budou na displeji zařízení vypadat jako na obrázku 12.2. Nahoře pole pro zobrazení textu, pod ním pole pro editaci textu a poslední je pole typu TextMultiLine pro zadání víceřádkového textu.



Obrázek 12.2: Aplikace s formulářem obsahujícím různá textová pole, spuštěná na emulátoru

12.2 Tlačítka

Tlačítka jsou k dispozici ve dvou základních podobách. Klasické, označené jako Button a grafické, ImageButton. I když se to tak na první pohled nemusí jevit, Button je rozšířením třídy TextView, ImageButton rozšířením třídy ImageView.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 12_Prvky_formulare
projekt Tlacitka

V XML podobě vypadá klasické tlačítko, umístěné do formuláře typu ConstraintLayout, takto:

```
<Button
    android:id="@+id/tlacitko1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="24dp"
    android:layout_marginEnd="24dp"
    android:layout_marginTop="64dp"
    android:onClick="tlacitko_click"
    android:text="Tlačítko"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

První parametr obsahuje id, neboli unikátní název tlačítka. V tomto případě je tím nezaměnitelným názvem „tlacitko1“. Následují dva parametry udávající velikost tlačítka. Zde platí totéž co u textových polí, velikost může být nastavena buď jako fixní, jako wrap_content nebo jako match_constraint či match_parent. Parametry marginStart, marginEnd a marginTop určují velikost

odsazení tlačítka od okolních objektů. Parametr onClick říká, která událost bude zavolána po kliknutí na tlačítko. Následuje text který je zobrazený na tlačítku a parametry umístění ve formuláři.

Jsou dva základní způsoby, jak volat událost po stisku tlačítka. Jednodušší je ten, který je naznačený v XML – parametr onClick obsahuje název události, ve třídě která je k formuláři přiřazena stačí napsat metodu s tímto názvem a nastavit ji jako veřejnou (pomocí klíčového slova public):

```
public void tlacitko_click (View view) {
    // kod, ktery se provede po stisknuti tlacitka
}
```

V parametru view jsou předány vlastnosti tlačítka které událost vyvolalo, takže je možné pro více tlačítek použít stejnou událost a až v ní rozlišit, které tlačítko ji vyvolalo.

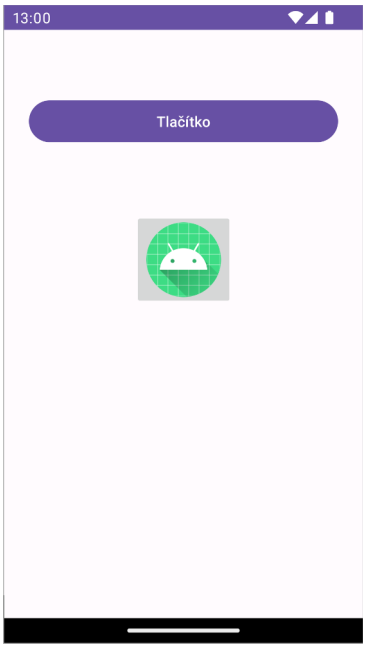
Druhou možností je dynamické vytvoření listeneru, který událost vyvolá. Tento postup se dá použít pokud je potřeba událost kliknutí zaregistrovat například v události onCreate() formuláře:

```
Button button = (Button) findViewById(R.id.tlacitko1);
button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
        // kod, ktery se provede po stisknuti tlacitka
    }
});
```

Z hlediska finální funkčnosti jsou oba postupy identické. Parametr tlačítka onClick, který jsme minule vyplnili názvem metody, zůstane tentokrát prázdný.

ImageButton se od prvku Button liší pouze tou jedinou základní věcí – namísto systémového vzhledu a vloženého textu je tvořen bitmapovým obrázkem. Obrázek se vkládá do parametru srcCompat a musí být uložený ve složce res projektu. Způsob volání události je stejný jako v případě tlačítka typu Button.

Popsaná tlačítka budou na displeji zařízení vypadat jako na obrázku 12.3. Horní je tlačítko typu Button, spodní tlačítko typu ImageButton. Místo obrázku je na druhém tlačítku použita systémová ikona Android.



Obrázek 12.3: Aplikace s formulářem obsahujícím standardní a obrázkové tlačítko, spuštěná na emulátoru

12.3 Images, obrázky

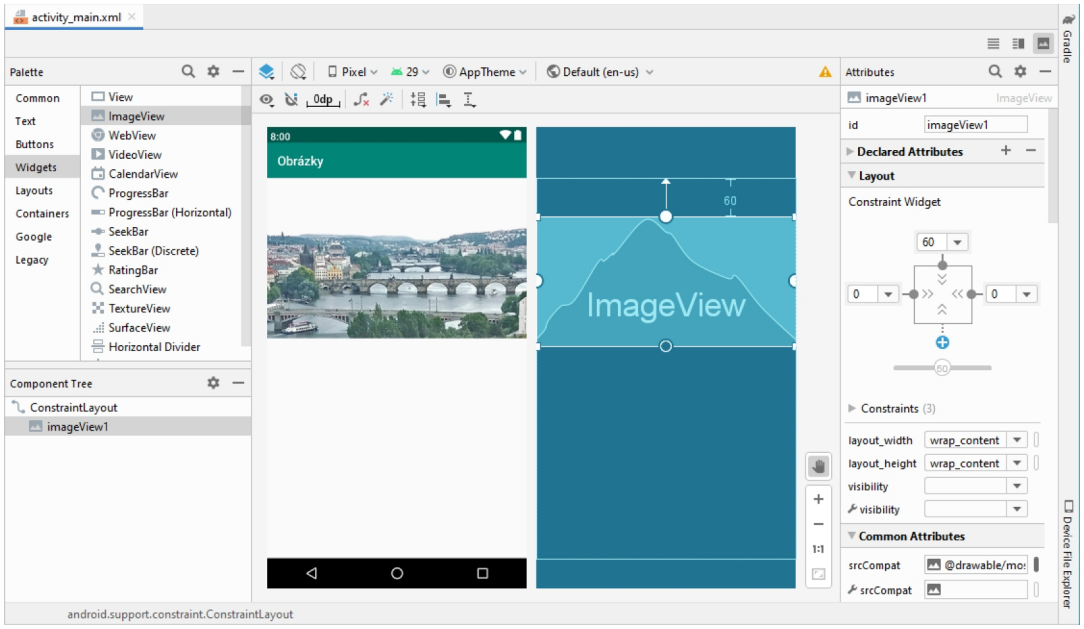
Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 12_Prvky_formulare
projekt Obrazky

Pro vkládání obrázků všeho druhu je určena komponenta ImageView, odvozená od třídy View. Pomocí této komponenty se na displeji zobrazují jak jednotlivé obrazové prvky, tak i kompletní grafický vzhled aplikací. Aby bylo možné komponentu využít, je nutné mít ve složce app/res/drawable/ uložený bitmapový soubor, který komponenta následně zobrazí.

Podporované formáty souborů jsou png, jpg a gif. Pro komplexní grafiku se doporučuje používat zejména png, pro fotografie je vhodný formát jpg.

Komponentu ImageView lze do formuláře vložit v grafickém editoru, nachází se v paletě komponent ve skupinách Common a Widgets. Před jejím vykreslením ještě Android Studio požádá o specifikaci grafického souboru.

Parametrů, které lze komponentě nastavit není mnoho. Nejdůležitější většinou bývá správné umístění, zejména vůči ostatním grafickým prvkům formuláře.



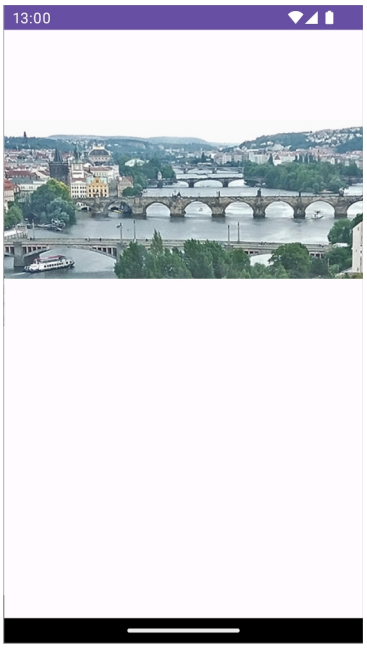
Obrázek 12.4: Práce s komponentou ImageView v designéru Android Studia

XML popis komponenty ImageView, znázorněné na předchozím obrázku, vypadá ve formuláři typu ConstraintLayout následovně:

```
<ImageView
    android:id="@+id/imageView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="60dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:srcCompat="@drawable/mosty" />
```

První je, jako vždy, id prvku. V tomto případě „imageView1”. Následují dva atributy pro nastavení velikosti, oba wrap_content, neboli dynamické přizpůsobení rozměrů komponenty velikosti vloženého obsahu. Následuje velikost odsazení 60dp a způsob umístění ve formuláři. Posledním parametrem je odkaz na grafický soubor umístěný ve složce drawable.

Výsledek by měl na displeji vypadat takto:



Obrázek 12.5: Aplikace s komponentou ImageView, spuštěná na emulátoru

12.4 Plovoucí tlačítko

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 12_Prvky_formulare
projekt PlovouciTlacitko

Floating Action Button (FAB), neboli plovoucí tlačítko, je prvkem uživatelského rozhraní, který lze vložit do formuláře a nechat ho jakoby vznášet nad ostatními elementy. Má kruhový tvar, spouští jednu akci, zpravidla bývá umístěné v pravém dolním rohu a zejména na menších displejích uživateli často spíše překáží. Proto je jeho použití potřeba vždy pečlivě zvážit a sáhnout po něm pouze v případě, kdy jsme si jistí, že uživatel jeho snadnou dostupnost ocení.

Pozor, plovoucí tlačítko je součástí externí knihovny Design Support, resp. „com.android.support:design“, kterou je potřeba do projektu nejprve přidat, viz kapitola 11 Závislé zdroje – Dependencies.

Tlačítko lze vložit do formuláře v grafickém editoru, z palety komponent. Nachází se ve skupině Buttons. V XML souboru formuláře potom bude vypadat nějak takto:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/floatingActionButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="16dp"
    android:layout_marginEnd="16dp"
    android:clickable="true"
    android:contentDescription="Plovoucí tlačítko"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:srcCompat="?android:attr/actionModeShareDrawable" />
```

Jde o tlačítko umístěné v kontejneru ConstraintLayoutu, s nastavenou standardní ikonou pro sdílení. Umístění tlačítka je řešené pomocí atributů layout_constraintBottom_toBottomOf

a layout_constraintEnd_toEndOf doprava dolů. Zde je nutné upozornit, že umístění do pravého dolního rohu se dělá v různých typech formulářů různým způsobem. V uvedeném ConstraintLayout je to:

```
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toEndOf="parent"
```

V RelativeLayout se použijí atributy:

```
android:layout_alignParentBottom="true"
android:layout_alignParentRight="true"
```

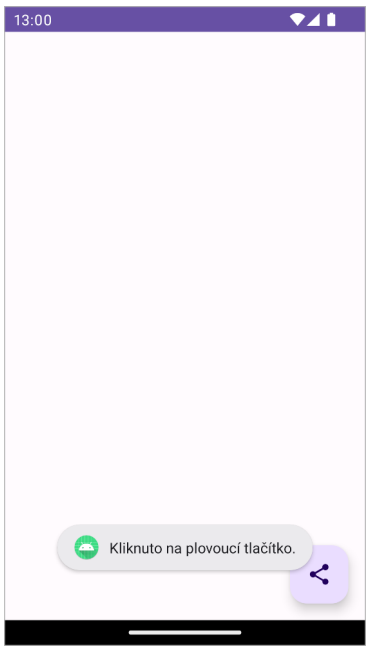
A v kontejneru typu LinearLayout potom jediný atribut:

```
android:layout_gravity="bottom|end"
```

Aby tlačítko mělo praktický smysl, musí se kromě jeho umístění také zaregistrovat listener, neboli posluchač události pro kliknutí. Může se tak učinit v události formuláře onCreate():

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    ...

    // registrace listeneru kliknuti na tlacitko
    FloatingActionButton fab = findViewById(R.id.floatingActionButton);
    fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Toast.makeText(getApplicationContext(),
                  "Kliknuto na plovoucí tlačítko.", Toast.LENGTH_SHORT).show();
        }
    });
}
```

Nastavené plovoucí tlačítko bude na displeji zařízení nebo na emulátoru vypadat jako na obrázku 12.6.



Obrázek 12.6: Aplikace s plovoucím tlačítkem, spuštěná na emulátoru

12.5 Záhlaví formuláře

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 12_Prvky_formulare
projekt ZahlaviFormulare

Ve starších verzích Androidu bývalo standardní součástí hlavního formuláře záhlaví, takzvaný Toolbar, obvykle s názvem aplikace. Spolu s Androidem 12 byl ale uvedený také design uživatelského rozhraní, nazvaný Material 3, který již záhlaví neobsahuje. Pokud vývojář záhlaví potřebuje, tak si ho od té doby musí do aktivity doplnit sám. Ve většině případů se bez něj dá obejít, jsou ale situace, kdy se záhlaví opravdu hodí, nebo se mu dá jen obtížně vyhnout. Záhlaví může být, kromě názvu aplikace nebo aktivity, doplněné o funkce otevírání menu, ikonky, může v něm být integrované pole pro vyhledávání a může mít mnoho dalších, uživatelsky užitečných vlastností. Záhlaví nemusí být nutně stále viditelné, dá se upravit tak, aby bylo průhledné, nebo aby se skrývalo, pokud není z uživatelského hlediska zrovna potřeba.

Záhlaví lze do formuláře umístit přímo z menu designéru. Tam je k dispozici jednak základní Toolbar a navíc i od něj odvozený Material Toolbar, který má doplněné některé vlastnosti Material designu. Nebo se dá záhlaví, poměrně snadno, vložit přímo do XML kódu formuláře:

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:minHeight="?attr/actionBarSize"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:titleTextColor="@android:color/white"
    app:title="@string/app_name"
    android:background="?attr/colorPrimary">
</androidx.appcompat.widget.Toolbar>
```

V takové konfiguraci není na záhlaví nic speciálního. Je nastavena pouze barva pozadí a textu, přičemž samotným textem je název aplikace, uložený v souboru app/res/values/strings.xml. Jenom je potřeba dát pozor na pozicování dalších komponent ve formuláři, ty už se nebudou vztahovat relativně k horní hraně formuláře, ale ke spodní hraně záhlaví.

Aby toolbar fungoval, musí být programově inicializován. Obvykle se to dělá v události formuláře onCreate():

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    ...

    // nastaveni Toolbaru
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
}
```

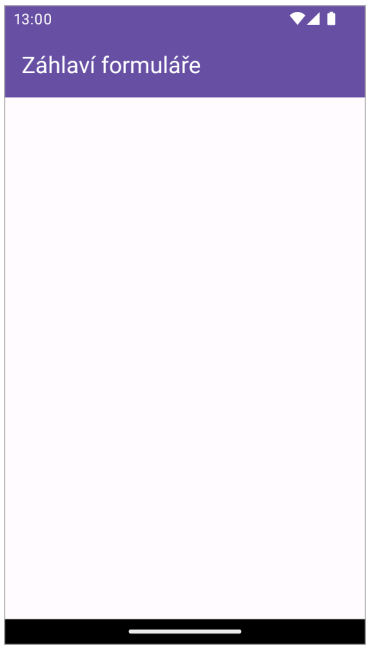
Kdyby bylo potřeba text v záhlaví dynamicky měnit, dá se to zařídit jednoduchým příkazem:

```
getSupportActionBar().setTitle("Test");
```

Podobně snadno se dá změnit barva záhlaví, v tomto případě na červenou, s 50 procentní průhledností:

```
ColorDrawable barvaZahlavi = new ColorDrawable(Color.argb(128, 255, 0, 0));
getSupportActionBar().setBackgroundDrawable(barvaZahlavi);
```

Další úpravy a rozšíření záhlaví si ukážeme až na příkladech, které se záhlavím pracují, nebo pracují s komponentami, které se bez záhlaví neobejdou. Formulář se základní verzí záhlaví bude na displeji zařízení nebo na emulátoru vypadat jako na obrázku 12.7.

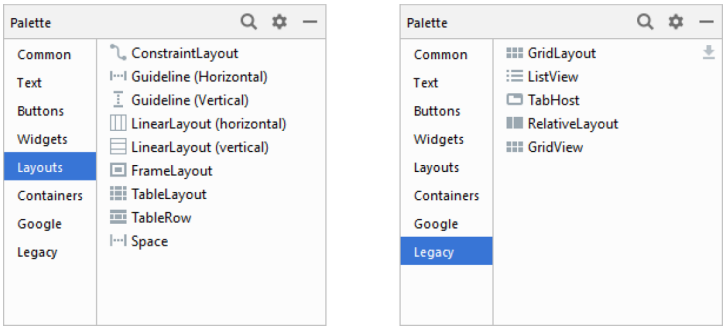


Obrázek 12.7: Formulář s komponentou Toolbar v nejjednodušším možném nastavení

13 Rozvržení prvků na formuláři

V předchozím příkladu jsme prvky do formuláře umístili jednoduše v editoru a nijak jsme se nezatěžovali přizpůsobováním jejich polohy v závislosti na orientaci nebo velikosti displeje. V případě jednoho tlačítka na displeji nás to ani nijak trápit nemuselo. S tím se ale při tvorbě aplikací nedá vystačit, v naprosté většině případů je nutné mít na formuláři větší množství prvků s přesně daným rozložením. Často je potřeba zohlednit i otočení displeje a pokud nejde o aplikaci pouze pro jeden konkrétní typ zařízení, musí se brát v potaz také různé velikosti displeje.

Pro řešení umístění prvků na formuláři existuje několik standardních postupů. A jak už to tak bývá, každý má nějaké výhody a nějaké nevýhody. Je tedy na vývojáři, kterou metodu vyhodnotí pro daný účel jako nejvhodnější. Všechny mají jednu společnou vlastnost – jsou řešeny pomocí takzvaných kontejnerů, neboli neviditelných prvků, jež se vloží do formuláře a všechny objekty, které mají být zobrazeny, jsou umístěny v těchto kontejnerech. Kontejnery jsou v grafickém editoru k dispozici v paletě nástrojů pod záložkou Layouts (pozor, neplést se záložkou Containers). Dva starší kontejnery, GridLayout a RelativeLayout, jsou pod záložkou Legacy. Byly tam přesunuty ze záložky Layouts, protože je lze nahradit kontejnerem ConstraintLayout a v budoucnu se počítá s ukončením jejich podpory. Zatím však jejich použití nic nebrání.



Obrázek 13.1: Kontejnery v paletě nástrojů

13.1 Linear Layout

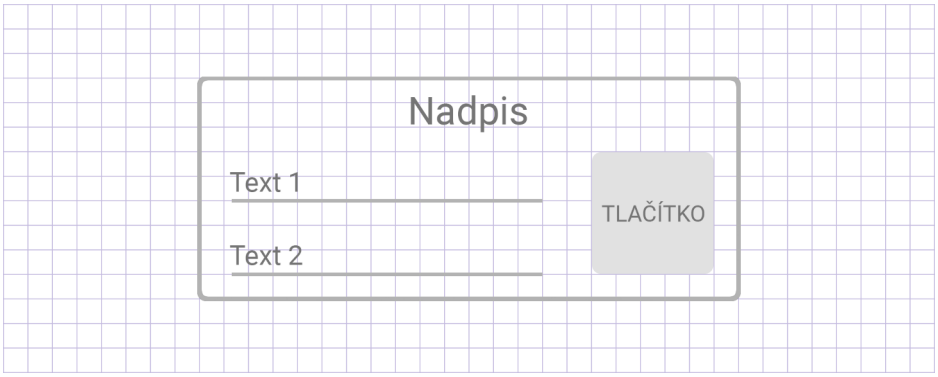
Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 13_Rozvrzeni_prvku_na_formulari
projekt LinearLayout

Linear Layout je často používaný typ rozvržení objektů. Umožňuje umístění několika prvků buď vedle sebe nebo pod sebou. Proto existuje ve dvou variantách kontejnerů – vertikální uspořádá objekty pod sebe, horizontální vedle sebe. Aby bylo možné vytvořit složitější strukturu objektů na displeji, dají se kontejnery zanořovat neboli vkládat do sebe. Jeden horizontální tak může obsahovat třeba tři vertikální, čímž se dosáhne potřebného rozložení prvků ve dvou osách displeje. Linear Layout není v základní vrstvě výpočetně příliš náročný, takže se prvky na displeji zobrazí rychle. Je ale dobré mít na paměti, že s každým zanořením kontejneru rychlost zpracování výrazně klesá. Proto je vhodné mít finální strukturu kontejnerů co nejjednodušší.

Jak vzájemně kombinovat vertikální a horizontální kontejnery si ukážeme na příkladu. Vytvoříme formulář který bude vypadat jako na obrázku 13.2. Budeme na něj mít tyto požadavky:

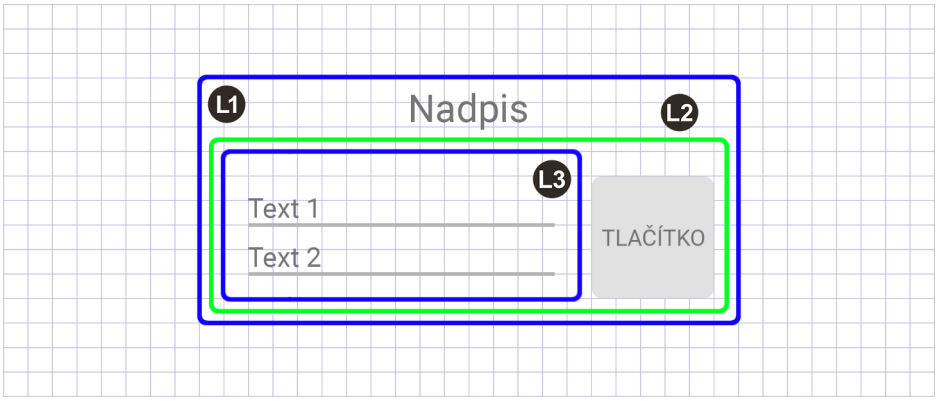
- šířka bude přes celý displej, bez ohledu na to jak široký displej je

- po otočení telefonu se roztáhne/zkrátí tak, aby stále byl po celé šířce displeje
- velikost tlačítka bude stále stejná, roztahovat a zkracovat se budou pouze textová pole
- výška všech prvků bude pevná, nebude se měnit ani při otočení displeje



Obrázek 13.2: Očekávaný vzhled formuláře

Pro Linear Layout navrhneme strukturu vnoření kontejnerů tak, aby byly prvky uspořádány požadovaným způsobem a abychom přitom do sebe vložili co nejmenší počet kontejnerů:



Obrázek 13.3: Návrh formuláře typu Linear Layout

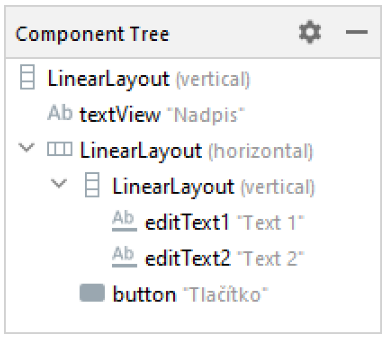
Kontejner L1 je vertikální (protože bude obsahovat dva prvky nad sebou, pole s nadpisem a pod ním kontejner L2), je do něj vloženo pole typu TextView s textem Nadpis a pevně nastavenou výškou layout_height na 50 dp. Šířka kontejneru layout_width je nastavena na 0 dp, což je ekvivalent pro hodnotu match_parent - tedy aby se roztáhl přes celý displej.

Do kontejneru L1 vsazený kontejner L2 je horizontální, protože bude obsahovat dva prvky (kontejner L3 a vedle něj tlačítko) s šířkou layout_width nastavenou na match_parent, aby se roztáhl přes celý kontejner L1.

Kontejner L3 je opět vertikální (bude obsahovat dvě textová pole nad sebou). Šířka kontejneru je match_parent, aby se roztáhl na možné maximum a má nastavenou layout_weight na 1, aby měl při rozhodování o velikosti nejvyšší váhu. Vedle něj je vložené tlačítko s rozměry 100 x 100 dp. Váha u tlačítka není nastavena, zajistí se tím kontejneru L3 vždy větší váha při rozhodování o změně šířky uvnitř kontejneru L2.

Kontejner L3 obsahuje dvě pole typu Plain Text s šířkou match_parent, aby využily maximum prostoru, který jim dává kontejner L3 a s pevnou výškou 50 dp.

Výsledek by měl v okně Component Tree vypadat jako na obrázku 13.4.



Obrázek 13.4: Prvky formuláře v okně Component Tree

Vkládání kontejnerů a dalších objektů je možné udělat v grafickém editoru Android Studia jejich přetažením z okna Palette do designu aktivity. Alternativně je lze přetahovat do okna Component Tree, rovnou na hierarchicky správnou úroveň.

Takto se dá ušetřit zakládání většiny atributů - ty vygeneruje sám editor. Přesto ale bude pravděpodobně potřeba během sestavování formuláře sahat do XML kódu, protože trefit umístění objektů není úplně snadné. Výsledný XML kód by měl vypadat takto:

```
<!-- linearni layout L1, vertikalni, sirka max, vyska 150dp, odsazeni 32dp -->
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="150dp"
    android:layout_marginTop="32dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent">

    <!-- textove pole Nadpis, sirka max, vyska 50dp -->
    <!-- text vycentrovany, velikost textu 30sp -->
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:text="Nadpis"
        android:textAlignment="center"
        android:textSize="30sp" />

    <!-- linearni layout L2, horizontalni, sirka max, vyska 100dp -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="100dp"
        android:orientation="horizontal">

        <!-- linearni layout L3, vertikalni, sirka max, vyska 100dp -->
        <!--  vaha=1 -->
        <LinearLayout
            android:layout_width=""
            android:layout_height="100dp"
            android:layout_weight="1"
            android:orientation="vertical">

            <!-- prvni pole pro zadani textu, typ text -->
```

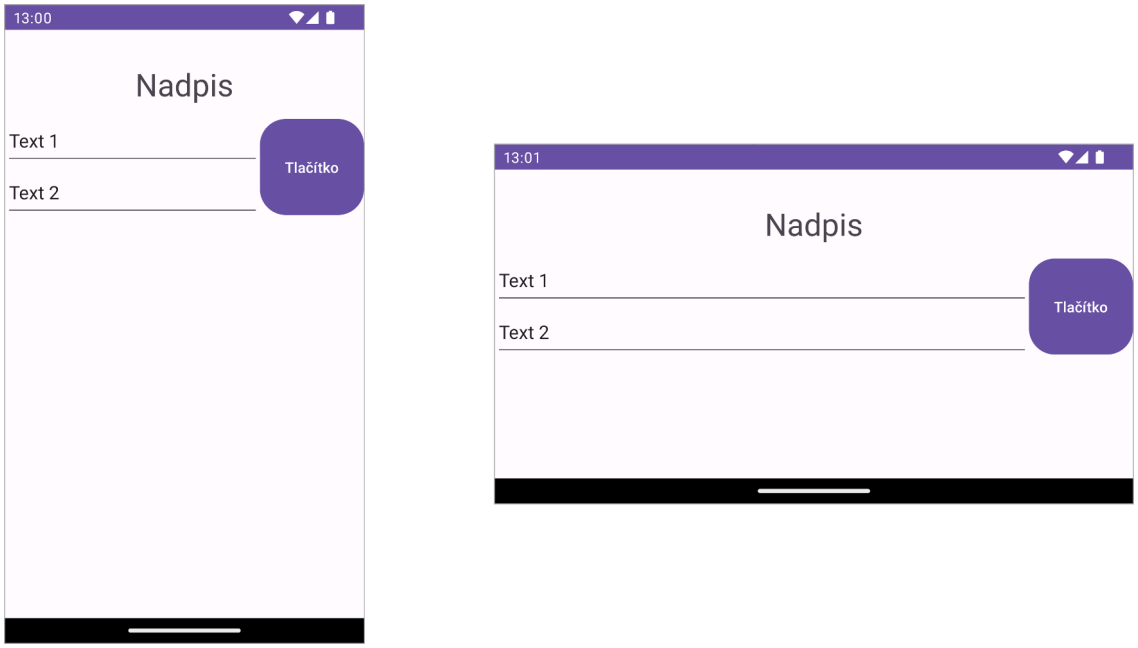
```
            <!-- sirka max, vyska 50dp, prednastaveny obsah 'Text 1' -->
            <EditText
                android:id="@+id/editText1"
                android:layout_width="match_parent"
                android:layout_height="50dp"
                android:inputType="text"
                android:text="Text 1" />

            <!-- druhe pole pro zadani textu, typ text -->
            <!-- sirka max, vyska 50dp, prednastaveny obsah 'Text 2' -->
            <EditText
                android:id="@+id/editText2"
                android:layout_width="match_parent"
                android:layout_height="50dp"
                android:inputType="text"
                android:text="Text 2" />
        </LinearLayout>

        <!-- tlacitko 100x100dp -->
        <Button
            android:id="@+id/button"
            android:layout_width="100dp"
            android:layout_height="100dp"
            android:text="Tlačítko"
            app:cornerRadius="25dp" />
    </LinearLayout>
</LinearLayout>
```

XML kód vygenerovaný grafickým editorem může obsahovat další atributy. Můžeme je v kódu nechat a případně podle potřeby upravit. Zde v příkladu jsou uvedeny pouze ty, které jsou pro sestavení formuláře nezbytně nutné.

V XML kódu je nutné věnovat zvýšenou pozornost ukončovacím prvkům </LinearLayout>, určujícím hranici kontejneru. Grafický editor ne vždy pochopí, zda bylo našim záměrem vložit prvek do kontejneru nebo mimo něj a často se až v XML pozná, jestli je skutečně tam, kde ho potřebujeme mít.

Zanoření prvků je možné kontrolovat v grafickém editoru v okně Component Tree, kde jsou zobrazeny všechny prvky včetně hierarchicky znázorněného vnoření. Pokud je potřeba špatně umístěný prvek přemístit do správného kontejneru, stačí ho v tomto okně chytit a přetáhnout do správné úrovně vnoření.

Na emulátoru nebo v zařízení by měl výsledek vypadat jako na obrázku 13.5.



Obrázek 13.5: Aplikace s kontejnerem typu Linear Layout, spuštěná na emulátoru

13.2 Constraint Layout

Constraint Layout je pravděpodobně tím nejpoužívanějším typem formuláře. Pozice prvků se v něm nastavuje relativně vůči ostatním objektům. Z hlediska výkonu jde o kontejner, který je rychlejší než vnořené Linear Layouty, ale nastavení polohy prvků je programátorsky o něco náročnější. Přesto se mu pro složitější návrhy většinou dává před Linear Layoutem přednost. Zanořování v tomto případě odpadá.

V Constraint Layout má každý objekt nastavenou svoji relativní polohu vůči jinému prvku uvnitř formuláře. K dispozici jsou možnosti pod, nad a vedle jiného objektu. S těmi si vývojář musí vystačit. Pro každý vkládaný prvek je tedy potřeba vybrat jiný, už existující, ke kterému se nový objekt vztáhne. Pozici objektů lze také vztáhnout k hranicím kontejneru, což se využije zejména pro první vložený objekt, u kterého ostatně ani jiná možnost není.

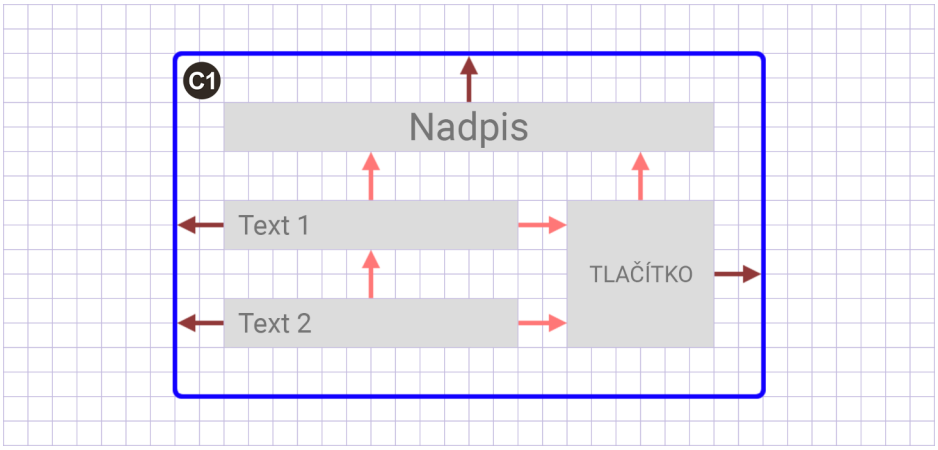
Kontejner Constraint Layout vznikl původně proto, aby zejména začínající tvůrci měli ulehčenou práci a mohli sestavovat relativní rozvržení pomocí grafického editoru. Neméně důležitou vlastností se ale stala také rychlost návrhu, která jde se snadným sestavením designu ruku v ruce. Díky grafickému editoru je možné formulář navrhnout podstatně rychleji než jakýkoliv jiný typ layoutu.

S Constraint Layoutem se Android vývojář setkává poměrně často. Jednak je značně oblíbeným kontejnerem a hlavně, pokud si při zakládání formuláře nechá vygenerovat Empty Layout, vytvoří v něm průvodce automaticky právě Constraint Layout. Vnitřní funkčností se od Relative Layoutu prakticky neliší, pouze je přidán atribut Bias a již zmíněné možnosti vizuálního návrhu.

Přestože je možné prakticky vše v tomto typu kontejneru nakreslit, tak si obvykle s grafickým editorem nejde vystačit. Dá se v něm ale snadno a rychle udělat hrubá práce a přesná nastavení

dodatečně upravit v XML. I tak je ale rychlost návrhu nesrovnatelná s jakoukoliv jinou technikou tvorby formuláře v Android Studiu.

Opět si ukážeme na příkladu, jak objekty k sobě navzájem navázat, aby vznikl formulář stejný jako v kapitole o Lineárním Layoutu.



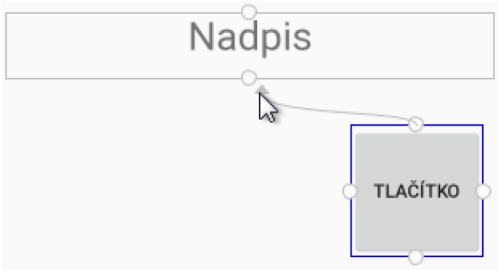
Obrázek 13.6: Návrh formuláře typu Constraint Layout

Kontejner C1 typu Constraint Layout je roztažený přes celou šířku displeje. Do kontejneru C1 je vloženo pole typu TextView s textem Nadpis a pevně nastavenou výškou layout_height na 50 dp. Pole je navázáno na horní hranu C1 a roztažené přes celou šířku kontejneru.

V C1 je vložené tlačítko s rozměry 100 x 100 dp. Tlačítko je navázané na pravou hranu kontejneru C1, aby bylo při každém rozměru zcela vpravo. Současně je navázané na textové pole Nadpis, aby mělo pevné vertikální umístění.

Nejvíce vazeb mají dvě pole typu Plain Text. Na levé straně jsou navázané na levou stranu kontejneru C1 a na pravé straně na tlačítko. Tím je zajištěné, že při změně velikosti displeje se pole podle potřeby roztáhnou. Horní hrana prvního pole je navázána na TextView Nadpis, aby získala pevné umístění ve svislé ose. Ze stejného důvodu je druhé pole PlainText navázáno na spodní hranu prvního pole. Úplně stejně by ale mohlo být navázané na pole Nadpis, výsledný efekt by byl identický.

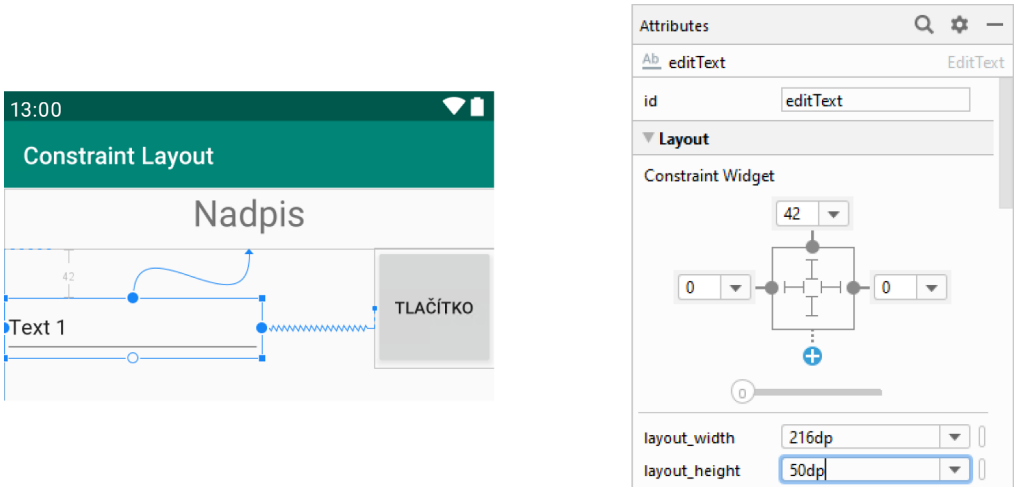
Na obrázku jsou pro názornost mezi objekty mezery, ve skutečnosti je však potřebujeme, aby se pole dotýkala hranami, proto u všech objektů nastavíme okraje velikosti 0 dp.



Obrázek 13.7: Nastavení vazby mezi objekty v grafickém editoru Android Studia

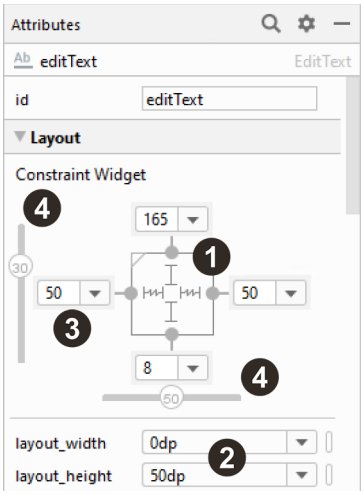
Základem pro umístění objektu je vazba vůči stěnám kontejneru a mezi jednotlivými prvky navzájem. V grafickém editoru tyto vazby nastavujeme jakoby vytažením šipky z hrany objektu a připojením jejího konce k hraně kontejneru nebo dalšího objektu.

Tímto způsobem vytvoříme všechny potřebné vazby a jejich parametry potom nastavíme v okně vlastností prvku.



Obrázek 13.8: Vazby mezi objekty a jejich parametry v okně vlastností

Na obrázku 13.8 je označené pole pro vkládání textu. Toto pole je napevno svázáno s levým okrajem kontejneru a s prvkem Nadpis. Při změně velikosti displeje bude mít vůči těmto prvkům stále stejnou polohu. Na obrázku je pro názornost odsazené od pole Nadpis o 42 bodů. Toto je potřeba v okně vlastností nastavit na 0, aby k sobě tato pole doléhala. Na pravé straně je pole navázané na tlačítko. Jelikož je na levé straně svázané s hranou kontejneru, je nutné definovat co se má stát při změně velikosti displeje. Levá strana zůstává stále u kontejneru (pevná vazba, vzdálenost 0), ale napravo se pole bude rozšiřovat (opět pevná vazba, tentokrát na tlačítko). Aby bylo jasné co udělat v případě zvětšení displeje, je dole v okně nastavení posuvník, který určuje takzvaný Bias, neboli vychýlení, v procentech. Tento parametr rozhoduje o vyvážení prvku v dané ose. 0 znamená, že bude prvek stále vlevo. Pokud by bylo nastavené 100, držel by se prvek u tlačítka, jakákoliv hodnota mezi způsobí, že se bude prvek při změně velikosti displeje posunovat v uvedeném poměru vůči oběma krajním mezím.



Obrázek 13.9: Parametry vazeb v okně vlastností

V okně vlastností je možné snadno nastavit:

❶ Typ velikost prvku (fixní/maximální/podle obsahu). Kliknutím se typ mění.

❷ Velikost prvku, odpovídá typu ❶, pro fixní je uvedena konkrétní hodnota, pro maximální je uvedena 0 a pro odpovídající obsahu je uvedeno wrap_content.

❸ Velikost okrajů prvku.

❹ Vychýlení (bias), které udává vyvážení polohy prvku v dané ose. Posuvníky jsou zobrazeny pouze pokud má objekt vazbu z obou protilehlých stran.

Způsob, jak v XML zapsat vazby na kontejner (tmavší šipky na obrázku 13.6):

`app:layout_constraintTop_toTopOf="parent"` – vazba na horní hranu kontejneru

`app:layout_constraintBottom_toBottomOf="parent"` – vazba na spodní hranu kontejneru (zde nevyužito)

`app:layout_constraintLeft_toLeftOf="parent"` – vazba na levou hranu kontejneru

`app:layout_constraintRight_toRightOf="parent"` – vazba na pravou hranu kontejneru

Vždy je nejprve nastavena hrana objektu ze kterého se vychází (Top, Bottom, Left, Right – horní, spodní, levá, pravá), potom hrana objektu na nějž se vazba odkazuje (opět toTop, toBottom, toLeft, toRight – na horní, na spodní, na levou, na pravou) následované identifikátorem odkazovaného objektu. Tím je v našem případě kontejner, neboli „parent“.

Nastavení vazby na jiný objekt je identické, opět nejdřív hrana pozicovaného objektu, potom hrana objektu ke kterému se bude vázat a nakonec identifikace odkazovaného objektu (světlejší šipky na obrázku 13.6):

`app:layout_constraintTop_toTopOf="@+id/textView"` – vazba na spodní hranu pole textView Nadpis

`app:layout_constraintTop_toTopOf="@+id/button"` – vazba na horní hranu tlačítka (v příkladu nevyužito)

`app:layout_constraintRight_toRightOf="@+id/button"` – vazba na levou hranu tlačítka

`app:layout_constraintRight_toRightOf="@+id/button"` – vazba na pravou hranu tlačítka (v příkladu nevyužito)

Nastavení velikosti okrajů objektů:

`android:layout_marginTop="0dp"` - okraj nad horní hranou objektu

`android:layout_marginBottom="0dp"` - okraj pod spodní hranou objektu

`android:layout_marginStart="0dp"` - okraj nalevo od objektu

`android:layout_marginEnd="0dp"` - okraj napravo od objektu

Nastavení vychýlení (bias) objektů v XML:

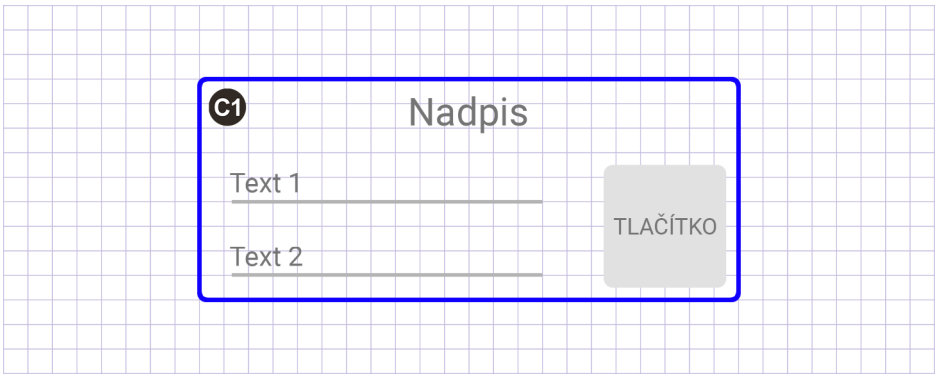
`app:layout_constraintHorizontal_bias="0.15"` – vodorovné vychýlení 15%, objekt bude z 15 procent vzdálenosti od hrany objektu navázaného vlevo a z 85 procent vzdálenosti od hrany objektu navázaného vpravo

`app:layout_constraintVertical_bias="0.65"` – svislé vychýlení 65%, objekt bude z 65 procent od horní hrany navázaného objektu a 35 procent od hrany objektu pod ním

13.2.1 Constraint Layout, 1. příklad

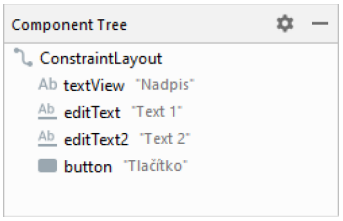
Pokusíme se sestavit stejný formulář jako u Linear Layoutu. Budeme postupovat podle obrázku 13.6, nicméně kotvení objektů nemusí být nutně stejné. Je ho možné nastavit mnoha způsoby, výsledek by ale měl být shodný. Navíc využijeme pouze vlastnost vychýlení.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 13_Rozvrzeni_prvku_na_formulari
projekt ConstraintLayout1



Obrázek 13.10: Návrh formuláře typu Constraint Layout

Jednotlivé prvky umístíme do formuláře v grafickém editoru a případně doladíme v XML. Constraint Layout má, stejně jako Linear Layout, pochopitelně také plochou strukturu, takže v okně Component Tree bychom měli mít uspořádání prvků jako na obrázku 13.11.



Obrázek 13.11: Prvky formuláře v okně Component Tree

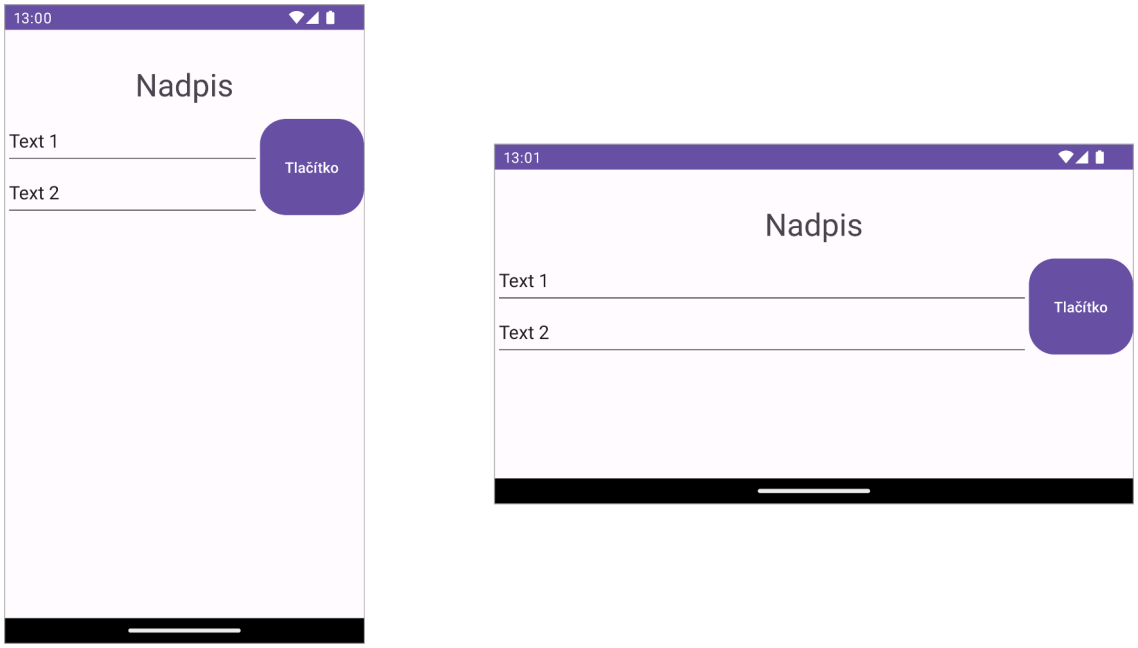
Výsledný XML kód:

```
<!-- constraint layout C1, sirka max, vyska 150dp -->
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="150dp"
```

```
tools:context="cz.stokratandroid.constraintlayout.MainActivity">

<!-- textove pole Nadpis, sirka max, vyska 50dp
     text vycentrovany, velikost textu 30sp
     umistene k leve, horni a prave hrane C1,
     horizontalni okraje 8dp, horni okraj 32dp -->
<TextView
    android:id="@+id/textView"
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:layout_marginLeft="8dp"
    android:layout_marginRight="8dp"
    android:layout_marginTop="32dp"
    android:text="Nadpis"
    android:textAlignment="center"
    android:textSize="30sp"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<!-- tlacitko 100x100dp
     vztazene ke spodni hrane textWiev
       a k prave hrane C1 -->
<Button
    android:id="@+id/button"
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:text="Tlačítko"
    app:layout_constraintTop_toBottomOf="@+id/textView"
    app:layout_constraintRight_toRightOf="parent"
    app:cornerRadius="25dp" />

<!-- prvni pole pro zadani textu, typ text
     sirka max, vyska 50dp, prednastaveny obsah 'Text 1'
     vztazene ke spodni hrane objektu textWiev, vzdalenost 0dp
       a k leve hrane R1
       a k leve hrane objektu button, vyvazeni 0 (vzdy vlevo) -->
<EditText
    android:id="@+id/editText"
    android:layout_width="0dp"
    android:layout_height="50dp"
    android:inputType="text"
    android:text="Text 1"
    app:layout_constraintHorizontal_bias="0.0"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toLeftOf="@+id/button"
    app:layout_constraintTop_toBottomOf="@+id/textView" />

<!-- druhe pole pro zadani textu, typ text
     sirka max, vyska 50dp, prednastaveny obsah 'Text 2'
     vztazene ke spodni hrane objektu editText, vzdalenost 0dp
       a k leve hrane C1
       a k leve hrane objektu button, vyvazeni 0 (vzdy vlevo) -->
<EditText
    android:id="@+id/editText2"
    android:layout_width="0dp"
    android:layout_height="50dp"
    android:inputType="text"
    android:text="Text 2"
    app:layout_constraintHorizontal_bias="0.0"
    app:layout_constraintLeft_toLeftOf="parent"
```

```
        app:layout_constraintRight_toLeftOf="@+id/button"
        app:layout_constraintTop_toBottomOf="@+id/editText" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Po spuštění v emulátoru nebo na zařízení by měl být zobrazen formulář s identickým chováním jako v předchozím příkladu.



Obrázek 13.12: Aplikace s kontejnerem typu Constraint Layout, spuštěná na emulátoru

13.2.2 Constraint Layout, 2. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 13_Rozvrzeni_prvku_na_formulari
projekt ConstraintLayout2

Formulář typu Constraint Layout nabízí několik pomocných prvků, které v některých případech dokážou zjednodušit celkový návrh. Tyto prvky jsou dostupné v grafickém editoru, po kliknutí do formuláře pravým tlačítkem myši v kontextovém menu, pod položkou Helper. Obzvláště zajímavé jsou vertikální a horizontální Guideline. Tyto prvky jsou dostupné také v paletě nástrojů pod záložkou Layouts.

Guideline je jakési vodítko či pomocná linka, vedoucí vodorovně nebo svisle formulářem v definované pozici. Má jednu důležitou funkci – je k ní možné vázat prvky formuláře. Ty tak nemusí být napojeny relativně na ostatní prvky nebo vůči okrajům formuláře. Využijeme toho ve druhém příkladu, vložíme si několik vodítek do formuláře a jednotlivé objekty budeme umísťovat relativně vůči nim.

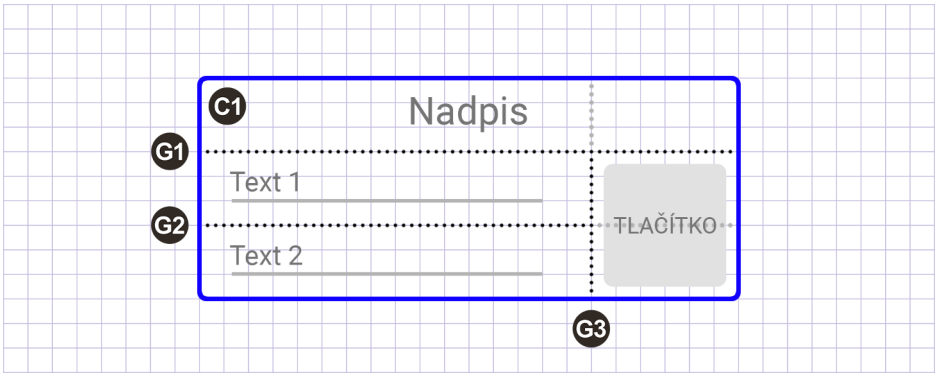
Umístění Guideline se ve formuláři řídí těmito atributy:

`app:layout_constraintGuide_begin="80dp"` – vazba na výchozí hranu kontejneru C1

`app:layout_constraintGuide_begin="130dp"` – vazba na koncovou hranu kontejneru C1

`app:layout_constraintGuide_percent="0.5"` – poměrné umístění mezi výchozí a koncovou hranou

U vodorovného vodítka je výchozí hranou horní a koncovou spodní hrana kontejneru, u svislého je výchozí levá a koncovou pravá hrana kontejneru C1. V případě poměrného umístění se udává procentuální podíl vzdálenosti od výchozí hrany ke Guideline, vůči vzdálenosti od Guideline ke koncové hraně C1. Poměr se zapisuje desetinným číslem od 0 do 1.



Obrázek 13.13: Návrh formuláře typu Constraint Layout se třemi vodítky Guideline G1-G3

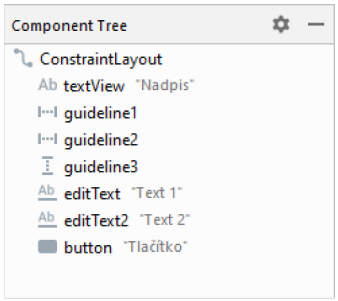
Začneme vytvořením formuláře Constraint Layout. Do něj umístíme první Guideline, G1, která bude vodorovná a bude ve výšce 80 dp od horního okraje. V XML zápisu bude vypadat takto:

```
<androidx.constraintlayout.widget.Guideline
    android:id="@+id/guideline1"
    android:orientation="horizontal"
    app:layout_constraintGuide_begin="80dp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Atribut orientation říká že jde o vodorovné vodítko, layout_constraintGuide_begin stanovuje vzdálenost od výchozí hrany kontejneru C1. Atributy udávající šířku a výšku vodítka, layout_width a layout_height jsou sice povinné, ale nemají žádný zvláštní význam. Vodorovné vodítko má vždy výšku 0 a šířku rovnou šířce kontejneru C1, svislé jsou pouze vzájemně prohozené. Nejjednodušší je tedy nechat u obou atributů hodnotu „wrap_content“.

Podobně vložíme do formuláře další vodorovné vodítko, tentokrát odsazené od horní hrany kontejneru 130 dp. A svislé vodítko, navázané na pravou hranu kontejneru C1, odsazené 100 dp. Tím máme tři vodítka, která stačí k tomu, aby mohli být všechny objekty přesně umístěny a nemusela mezi nimi být použita ani jedna vazba. Vše se vztáhneme buď k okrajům kontejneru C1 nebo k vodítkům.

Pro umístění vodítek a vytvoření vazeb mezi objekty použijeme grafický editor a výsledek případně doladíme v XML V okně Component Tree by mělo vzniknout uspořádání prvků jako na obrázku 13.14.



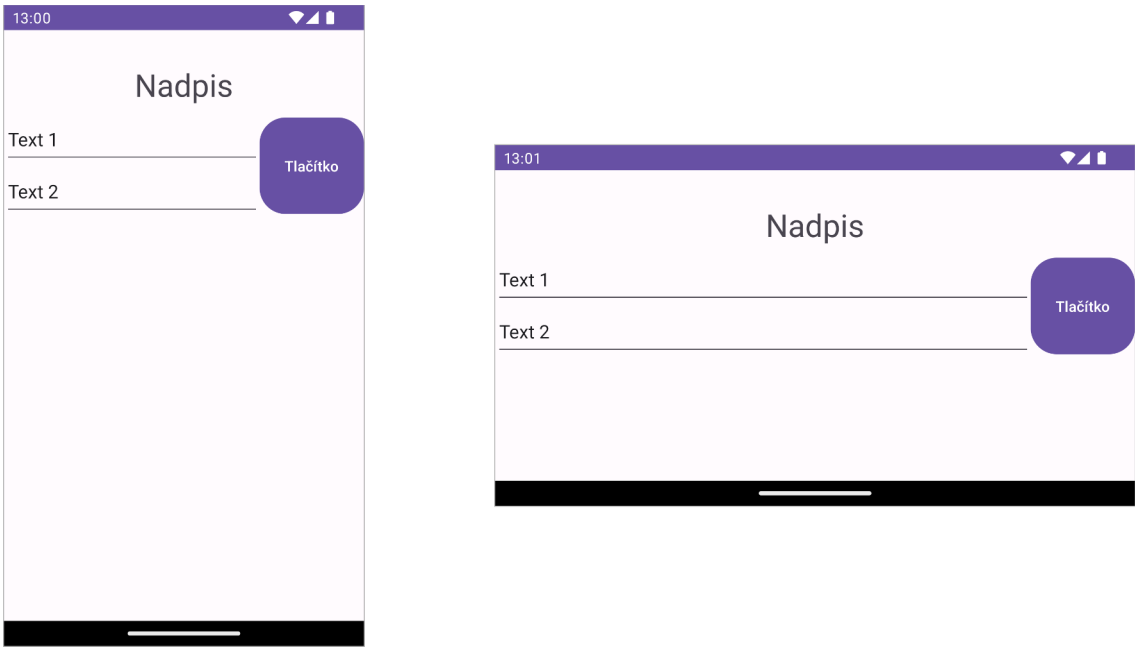
Obrázek 13.14: Prvky formuláře v okně Component Tree

Výsledný XML kód:

```
<!-- constraint layout C1, sirka max, vyska 150dp -->
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="150dp"
    tools:context="cz.stokratandroid.constraintlayout2.MainActivity">

    <!-- textove pole Nadpis, sirka max, vyska 50dp
         text vycentrovany, velikost textu 30sp
         umistene k leve, horni a prave hrane C1,
         horizontalni okraje 8dp, horni okraj 32dp -->
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:text="Nadpis"
        android:textAlignment="center"
        android:textSize="30sp"
        android:layout_marginTop="32dp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <!-- vodorovne voditko, 80dp od horni hrany C1 -->
    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/guideline1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        app:layout_constraintGuide_begin="80dp" />

    <!-- vodorovne voditko, 100dp od horni hrany C1 -->
    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/guideline2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        app:layout_constraintGuide_begin="100dp" />

    <!-- svisle voditko, 130dp od prave hrany C1 -->
    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/guideline3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        app:layout_constraintGuide_end="130dp" />

    <!-- tlacitko 100x100dp
         vztazene ke spodni hrane textWiev
           a k prave hrane C1 -->
    <Button
        android:id="@+id/button"
        android:layout_width="100dp"
        android:layout_height="100dp"
```

```
        android:text="Tlačítko"
        app:cornerRadius="25dp"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="@+id/guideline1" />

    <!-- prvni pole pro zadani textu, typ text
         sirka max, vyska 50dp, prednastaveny obsah 'Text 1'
         vztazene ke guideline1, vzdalenost 0dp
           a k leve hrane R1
           a ke guideline3, vyvazeni 0 (vzdy vlevo) -->
    <EditText
        android:id="@+id/editText"
        android:layout_width="0dp"
        android:layout_height="50dp"
        android:inputType="text"
        android:text="Text 1"
        app:layout_constraintRight_toLeftOf="@+id/guideline3"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toLeftOf="@+id/button"
        app:layout_constraintTop_toTopOf="@+id/guideline1" />

    <!-- druhe pole pro zadani textu, typ text
         sirka max, vyska 50dp, prednastaveny obsah 'Text 2'
         vztazene ke guideline2, vzdalenost 0dp
           a k leve hrane C1
           a ke guideline3, vyvazeni 0 (vzdy vlevo) -->
    <EditText
        android:id="@+id/editText2"
        android:layout_width="0dp"
        android:layout_height="50dp"
        android:inputType="text"
        android:text="Text 2"
        app:layout_constraintRight_toLeftOf="@+id/guideline3"
        app:layout_constraintHorizontal_bias="1.0"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toLeftOf="@+id/button"
        app:layout_constraintTop_toTopOf="@+id/guideline2" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Po spuštění v emulátoru nebo na zařízení by se měl zobrazit formulář s identickým chováním jako v předchozím příkladu.



Obrázek 13.15: Aplikace s kontejnerem typu Constraint Layout, spuštěná na emulátoru

13.3 Table Layout

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 13_Rozvrzeni_prvku_na_formulari
projekt TableLayout

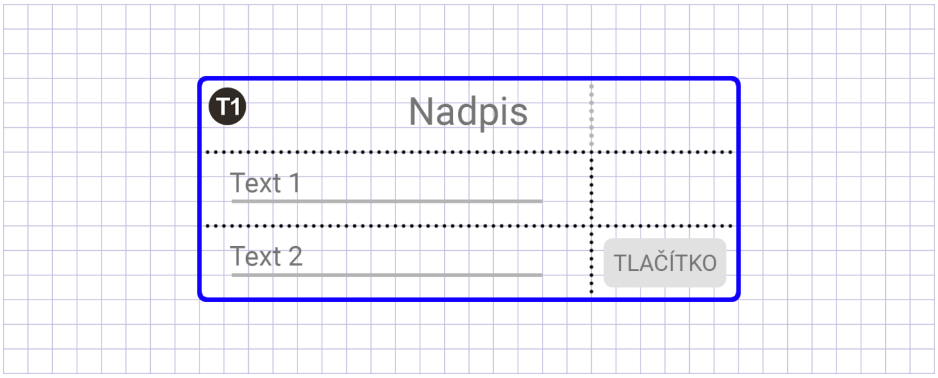
Pro rozvržení objektů do podoby tabulky se hodí TableLayout. Má dva kontejnery, jednak samotný TableLayout a dále TableRow, který reprezentuje jeden řádek tabulky. Tabulka má tedy tolik řádek, kolik je v kontejneru TableLayout vložených kontejnerů TableRow. Zobrazitelné objekty se vkládají do TableRow, kde se řadí automaticky vedle sebe.

Počet sloupců tabulky je dán maximálním počtem objektů v libovolné TableRow. Čili když budou použity například tři řádky, v prvním budou dva objekty, ve druhém čtyři a ve třetím tři, tak bude mít tabulka čtyři sloupce.

V kontejneru TableRow je možné objekt rozprostřít přes více sloupců a tím buňky tabulky jakoby sloučit. Nelze však sloučit buňky přes několik TableRow, neboli svisle, přes více řádků. Proto je tento typ layout nevhodný pro příklad, který jsme použili u Linear a Relative Layoutu. Tlačítko, které je na výšku přes dvě textové řádky, takto nelze jednoduše vložit do Table Layoutu kvůli nemožnosti slučování buněk nad sebou.

Table Row umožňuje vložení dalších kontejnerů, takže teoreticky by bylo možné obě textová pole vložit do dalšího Table Layoutu (případně Linear či jiného) a tak dosáhnout toho, že by tvořily jeden objekt o výšce tlačítka. Vnořování jde však i zde na vrub výkonu, je tedy lepší použít jiný typ kontejneru. Aby dával Table Layout v tomto případě smysl, změníme požadavek na velikost tlačítka tak, aby mělo výšku stejnou jako jednotlivá textová pole. Zbývající požadavky nechme stejné jako v předchozích příkladech.

Tabulkové rozložení objektů by tedy mělo vypadat takto:



Obrázek 13.16: Návrh formuláře typu Table Layout

Kontejner TableLayout obsahuje tři řádky, tvořené vloženými kontejnery TableRow a dva sloupce, dané dvěma prvky ve třetím řádku. Tabulce nastavíme pomocí parametru stretchColumns=”0” vlastnost, že první sloupec (tj. sloupec s indexem 0) bude mít vždy maximální šířku. Tím se zajistí, že druhý sloupec bude konstantní šířky a při otočení displeje bude změněna pouze velikost prvního sloupce s textovými poli.

Pole typu TextView, s textem Nadpis, vložíme do prvního řádku, nastavíme mu parametr span=2 pro rozložení přes dvě buňky a pole roztáhneme přes celou dostupnou šířku.

Do druhé řádky vložíme textové pole a opět je roztáhneme přes celou šířku buňky. Druhou buňku v tomto řádku necháme prázdnou.

Způsob, jak v XML zapsat potřebné parametry tabulky:

`android:stretchColumns="0"` - indexy sloupců, které mají být maximálně roztažené

`android:shrinkColumns="1,2"` - indexy sloupců, které mají být co nejužší (v příkladu nevyužito)

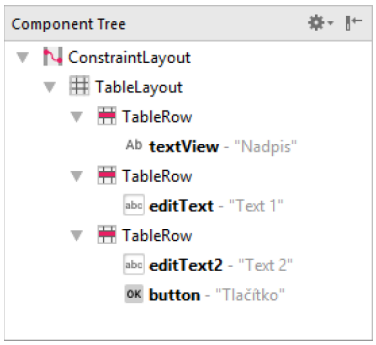
`android:collapseColumns="2,3"` - indexy sloupců, které mají být skryté (v příkladu nevyužito)

Způsob, jak v XML zapsat potřebné parametry řádku:

`android:layout_span="2"` - sloučení buněk, neboli počet sloupců, které má objekt obsadit

Kontejner TableLayout má ještě jednu zajímavou vlastnost – objekty je možné vkládat i mimo kontejnery TableRow. V takovém případě se chovají jako by byly roztažené přes celou šířku tabulky. Pokud je takto mezi kontejnery TableRow vloženo víc objektů, jsou řazené pod sebou a každý má opět šířku celé tabulky. V našem příkladu by toho šlo využít pro vložení pole Nadpis, kdybychom je nechtěli vkládat do samostatného kontejneru TableRow.

V okně Component Tree bychom měli mít uspořádání prvků jako na obrázku 13.17.



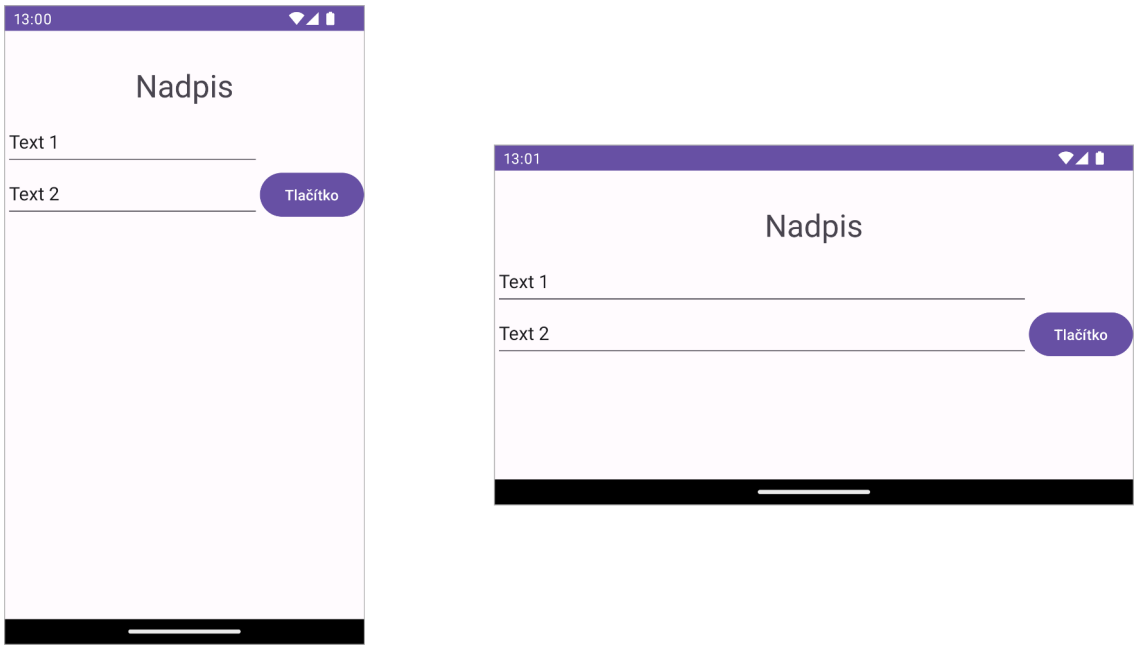
Obrázek 13.17: Prvky formuláře v okně Component Tree

Výsledný XML kód:

```
<!-- tabulkovy layout T1, sirka max, vyska 150dp
     prvni sloupec (tj. sloupec s indexem 0) vzdy max sirka -->
<TableLayout
    android:layout_width="0dp"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_height="150dp"
    android:stretchColumns="0"
    android:layout_marginTop="32dp">

    <!-- prvni radek tabulky, sirka max, vyska 50dp -->
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="50dp" >

        <!-- textove pole Nadpis, sirka max, vyska 50dp
             text vycentrovany, velikost textu 30sp
             roztazeny pres dva sloupce -->
        <TextView
            android:id="@+id/textView"
            android:text="Nadpis"
            android:layout_width="match_parent"
            android:layout_height="50dp"
            android:textAlignment="center"
            android:textSize="30sp"
            android:layout_span="2" />
    </TableRow>

    <!-- druhy radek tabulky, sirka max, vyska 50dp -->
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="50dp" >

        <!-- prvni pole pro zadani textu, typ text
             sirka max, vyska 50dp, prednastaveny obsah 'Text 1' -->
        <EditText
            android:id="@+id/editText"
            android:layout_width="match_parent"
            android:layout_height="50dp"
            android:inputType="text"
```

```
            android:text="Text 1" />

        <!-- druhy sloupec v tomto radku neni vyuzity -->
    </TableRow>

    <!-- treti radek tabulky, sirka max, vyska 50dp -->
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="50dp" >

        <!-- druhe pole pro zadani textu, typ text
             sirka max, vyska 50dp, prednastaveny obsah 'Text 2' -->
        <EditText
            android:id="@+id/editText2"
            android:layout_width="match_parent"
            android:layout_height="50dp"
            android:inputType="text"
            android:text="Text 2" />

        <!-- tlacitko 100x50dp, text Tlacitko -->
        <Button
            android:id="@+id/button"
            android:layout_width="100dp"
            android:layout_height="50dp"
            android:text="Tlačítko"
            app:cornerRadius="25dp" />
    </TableRow>

</TableLayout>
```

Na emulátoru nebo zařízení bychom měli vidět formulář s identickým chováním jako v předchozích příkladech.



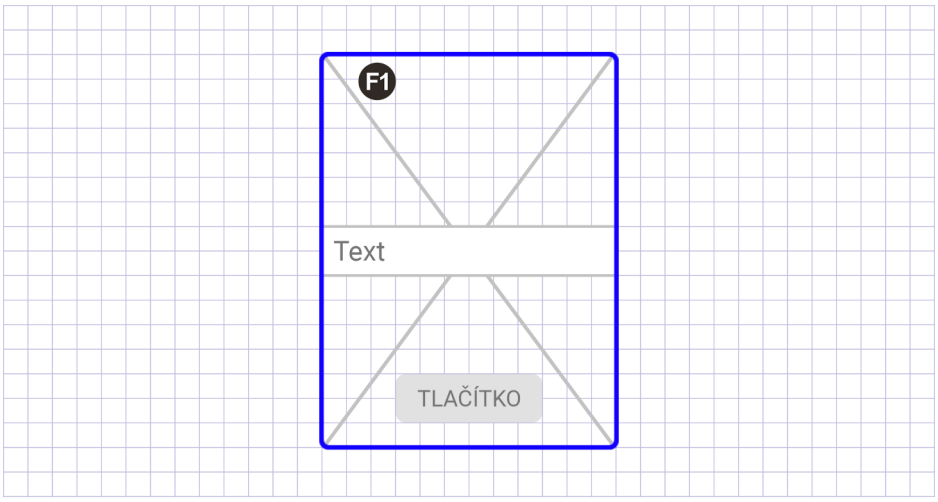
Obrázek 13.18: Aplikace s kontejnerem typu Table Layout, spuštěná na emulátoru

13.4 Frame Layout

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 13_Rozvrzeni_prvku_na_formulari
projekt FrameLayout

FrameLayout je jedním z nejjednodušších kontejnerů. Nedá se sám o sobě využít pro přesné umístění většího počtu prvků na formuláři, jeho síla je naopak ve velmi jednoduchých formulářích a v možnosti vnořování dalších kontejnerů. Často se také využívá jeho schopnosti překrývání elementů, například při vytváření úvodních obrazovek aplikací.

Můžeme si to zkusit na příkladu – vytvoříme formulář s obrázkem v pozadí, přes který umístíme pruh s textem a tlačítko.



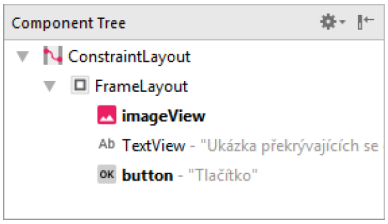
Obrázek 13.19: Návrh formuláře typu Frame Layout

Kontejner Frame Layoutu je v tomto případě roztažený přes celý displej, jak na šířku tak i na výšku. Do kontejneru je vložený element <ImageView ...> který zobrazí obrázek uložený ve složce drawable. Doprostřed kontejneru vložíme textové pole a do spodní části umístíme tlačítko. Nejsou v tom žádné záludnosti, pouze způsob umístění prvků je trošku těžkopádný – vše se dělá přes atribut gravity s případným zpřesněním pomocí neviditelných okrajů.

Způsob, jak v XML zapsat potřebné parametry prvků:

`android:layout_gravity="bottom|center"` – umístění objektu na pozici dolů+doprostřed

V hierarchické struktuře objektů v okně Component Tree, obrázek 13.20, je vidět, že jsou všechny prvky přímo v komponentě FrameLayout.



Obrázek 13.20: Prvky formuláře v okně Component Tree

Výsledný XML kód:

```
<!-- ramcovy layout F1, sirka max, vyska max -->
<FrameLayout
    android:layout_width="0dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_height="0dp"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent">

    <!-- image view, sirka max, vyska max
         umisteny na stred a oriznuty -->
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="centerCrop"
        android:src="@drawable/test_image" />

    <!-- textove pole, sirka max, vyska podle obsahu, na stred displeje
         modra barva pozadi, okraj 15 dp
         text 'Ukazka..', bila barva textu, velikost textu 20sp -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:background="?attr/colorPrimaryDark"
        android:padding="15dp"
        android:text="Ukázka překrývajících se elementů v kontejneru Frame
                      Layout"
        android:textColor="@android:color/white"
        android:textSize="20sp" />

    <!-- tlacitko, vyska podle obsahu, sirka podle obsahu
         umisteni dolu doprostred, text 'Tlacitko' -->
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|center"
        android:text="Tlačítko" />

</FrameLayout>
```

V emulátoru nebo na koncovém zařízení by měl výsledek vypadat jako na obrázku 13.21.

Obrázek 13.21: Aplikace s kontejnerem typu Frame Layout, spuštěná na emulátoru

13.5 Vkládání layoutů

Pro případ, kdy je formulář příliš komplikovaný, nebo kdy jsou některé jeho části použity ve více formulářích a hodí se je mít navržené samostatně, je k dispozici XML značka <include/>. Ta umožní vložit jeden formulář do jiného, nebo složit výsledný formulář z několika dílčích.

Základní syntaxe je prostá, povinný je pouze parametr, odkazující na vkládaný layout:

```
<include layout="@layout/activity_1"/>
```

Značku include je možné v případě potřeby doplnit dalšími parametry, zejména pozičními, ale také těmi, které přepíšou stejnojmenné parametry ve vkládaném formuláři. Tady je ale potřeba upozornit na jednu zvláštnost. Aby byly přepisované parametry zohledněny, musí mezi nimi být i nastavení šířky a výšky vkládaného layoutu. Například:

```
<include
      android:id="@+id/include1"
      layout="@layout/activity_1"
      app:layout_constraintTop_toTopOf="parent"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_marginTop="8dp" />
```

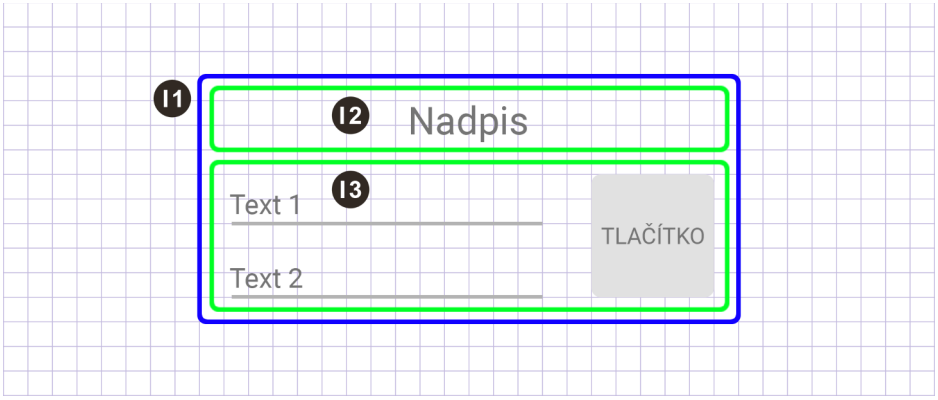
Takto nastavené parametry značky include říkají, že má id „include1“, což se hodí, pokud k ní bude například potřeba ukotvit nějaký další element. Vkládaný formulář se jmenuje „activity_1“ a je připojený k horní části hierarchicky nadřazeného kontejneru. Má šířku nastavenou na maximum a výšku podle velikosti obsahu. Posledním parametrem je odsazení na horní hraně. Jde o parametr,

který přepíše ten stejný parametr ve vkládaném formuláři. Pokud by ale značka „include“ šířku a výšku neobsahovala, byl by parametr odsazení ignorován.

13.5.1 Vkládání layoutů, 1. příklad

Můžeme si to vyzkoušet na praktickém příkladu. Řekněme, že si definujeme dva jednoduché formuláře, z nichž první bude obsahovat nadpis a druhý pole pro vložení textu a tlačítko. Tak jak je znázorněno na obrázku:

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 13_Rozvrzeni_prvku_na_formulari
projekt VlozenyLayout1



Obrázek 13.22: Návrh formuláře se dvěma vloženými layouty

Co je potřeba udělat?

- vytvořit formulář layout_1 s nadpisem
- vytvořit formulář layout_2 s textovými poli a tlačitkem
- oba vložit do formuláře activity_main

Pro všechny formuláře použijeme Constraint Layout, viz 13.2 Constraint Layout. Výsledný formulář by měl vypadat stejně jako v příkladu u této kapitoly. Z příkladu si můžeme pro ušetření práce vykopírovat textové pole, které nyní použijeme. První formulář, layout_1 bude obsahovat pouze nadpis:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!-- textove pole Nadpis, sirka max, vyska 50dp
         text vycentrovany, velikost textu 30sp
         umistene k leve, horni a prave hrane formulare,
         horizontalni okraje 8dp, vertikalni 0dp -->
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
```

```
        android:text="Nadpis"
        android:textAlignment="center"
        android:textSize="30sp"
        app:layout_constraintLeft_toLeftOf="parent"

        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Podobně si připravíme formulář, layout_2. Textová pole a tlačítko můžeme opět vykopírovat z příkladu u kapitoly 13.2 Constraint Layout:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!-- tlacitko 100x100dp,
         vztazene k horni a prave hrane kontejneru -->
    <Button
        android:id="@+id/button"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:text="Tlačítko"
        app:cornerRadius="25dp"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" /> />

    ...

</androidx.constraintlayout.widget.ConstraintLayout>
```

A teď už nezbývá, než oba formuláře spojit do jednoho v activity_main. To se dá udělat více způsoby. Nejjednodušší je bez zbytečných parametrů prostě připojit první a druhý formulář:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- vlozit layout layout_1,
         umisteny k horni hrane displeje,
         max sirka, vyska podle velikosti obsahu -->
    <include
        android:id="@+id/include1"
        layout="@layout/layout_1"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <!-- vlozit layout layout_2,
         umisteny pod include1 -->
    <include
```

```
        android:id="@+id/include2"
        layout="@layout/layout_2"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/include1" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Formuláře je ale možné do activity_main připojit i s nastavením rozměrů vložených formulářů a s přepsáním některých jejich parametrů:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- vlozit layout layout_1,
         umisteny k leve, horni a prave hrane displeje,
         max sirka, vyska podle velikosti obsahu -->
    <include
        android:id="@+id/include1"
        layout="@layout/layout_1"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginTop="32dp"
        android:layout_width="0dp"
        android:layout_height="wrap_content" />

    <!-- vlozit layout layout_2,
         umisteny pod include1, odsazení 32dp
         max sirka, vyska podle velikosti obsahu -->
    <include
        android:id="@+id/include2"
        layout="@layout/layout_2"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/include1"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

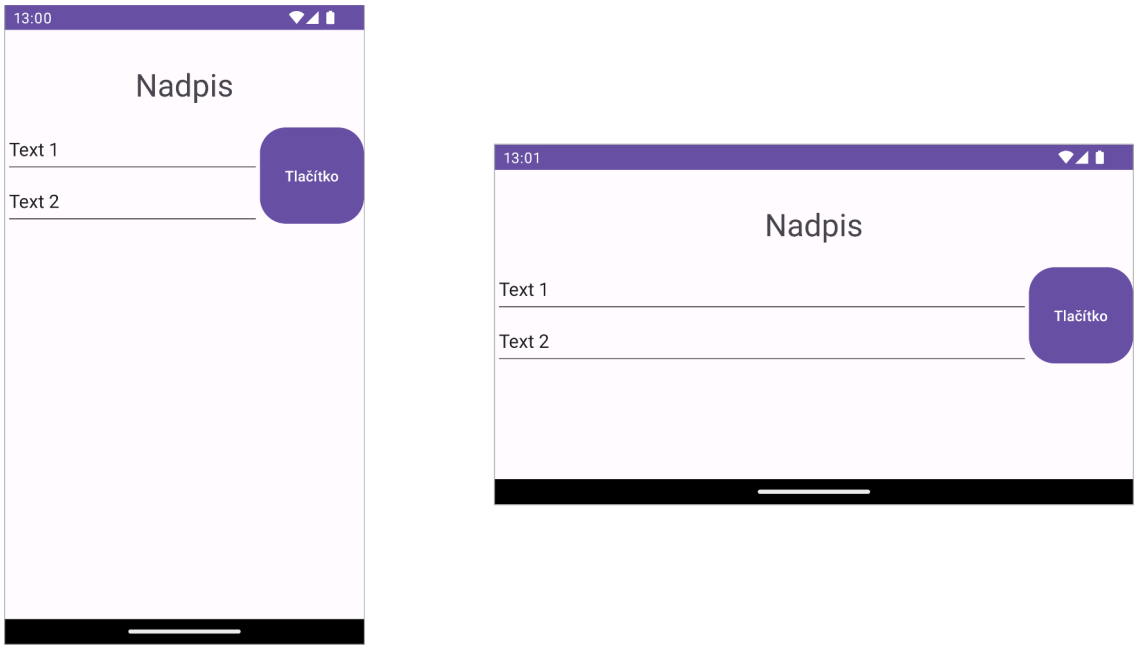
Takto jsme dosáhli nastavení polohy formulářů bez ohledu na jejich vlastní vnitřní nastavení. U druhého formuláře je navíc nastavené odsazení 8 dp, které přepisuje odsazení nastavené ve formuláři layout_2. Přesněji řečeno, formulář layout 2 odsazení nemá nastavené, takže mu ho tímto způsobem vlastně přidáme. Bez nastavení rozměrů druhého vkládaného formuláře by tento parametr nefungoval.

Nevýhoda vkládání formulářů pomocí <include/> je ta, že fakticky vznikne vnoření dvou layoutů, často stejného typu. V našem případě jsme pomocí <include/> vložili Constraint Layout do jiného formuláře typu Constraint Layout. Takže výsledek vypadal takto:

```
<ConstraintLayout>
   <ConstraintLayout>
      <TextView />
   </ConstraintLayout>
   <ConstraintLayout>
      <Button />
      <EditText />
      <EditText />
   </ConstraintLayout>
</ConstraintLayout>
```

Do sebe vložené formuláře způsobují zvýšené nároky na vykreslování a je lepší se jim vyhýbat. Takto navržený formulář bude méně efektivní, než formulář sestavený v jediném layoutu. Jak tento problém řešit, to si ukážeme v dalším příkladu.

V emulátoru nebo na koncovém zařízení by měl výsledek, s první verzí activity_main, vypadat jako na obrázku 13.23, čili ve své podstatě k nerozpoznání od výsledku příkladu z kapitoly 13.2 Constraint Layout. Ve druhé verzi by byla část s tlačítkem odsazena od nadpisu o 8 dp níž.



Obrázek 13.23: Aplikace se dvěma vloženými layouty, spuštěná na emulátoru

13.5.2 Vkládání layoutů, 2. příklad

Nevýhodou předchozího řešení je vnoření formulářů stejného typu do sebe. Zbytečně se tím zvyšují nároky na vykreslování formuláře. Pro tyto případy Android obsahuje XML element <merge/>, kterým se nahradí ve vnořeném formuláři typ layoutu. Ten je při vykreslování nahrazen layoutem z hierarchicky nadřazeného formuláře.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 13_Rozvrzeni_prvku_na_formulari
projekt VlozenyLayout2

Základní syntaxe vypadá takto:

```
<merge xmlns:android="http://schemas.android.com/apk/res/android">

...

</merge>
```

Zkusíme upravit předchozí příklad tak, aby se obešel bez vnořování formulářů stejného typu.

Co je potřeba udělat?

- upravit formulář layout_1 s nadpisem
- upravit formulář layout_2 s textovými poli a tlačítkem
- oba vložit do formuláře activity_main

Značka merge ve své podstatě nahradí definici typu formuláře. Tu si převezme kompilátor z hierarchicky nadřazeného formuláře. Značka může a nemusí obsahovat další parametry pro velikost, umístění formuláře a tak podobně. První formulář, layout_1 bude tedy vypadat takto:

```
<?xml version="1.0" encoding="utf-8"?>
<merge
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!-- textove pole Nadpis, sirka max, vyska 50dp
         text vycentrovany, velikost textu 30sp
         umistene k leve, horni a prave hrane formulare,
         horizontalni okraje 8dp, vertikalni 0dp -->
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:text="Nadpis"
        android:textAlignment="center"
        android:textSize="30sp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</merge>
```

Definice formuláře se tedy, od té v předchozím příkladu, liší pouze použitím značky merge namísto ConstraintLayout. Obsah zůstává a zůstat můžou i použité parametry. Pozor ale, parametry umístění prvku se teď už nebudou vztahovat ke kontejneru merge, ale k formuláři, do kterého bude blok vložený.

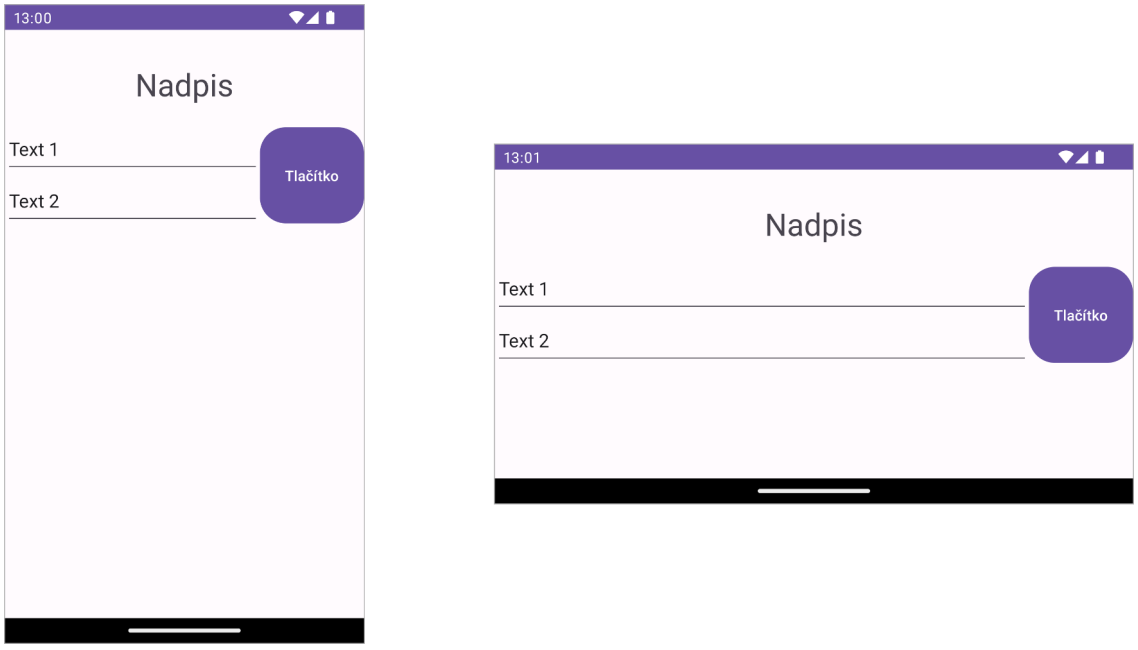
Stejnou operaci uděláme i s druhým formulářem. ConstraintLayout nahradíme značkou <merge/> a upravíme umístění prvků tak, jako by byly s prvky v layout_1 na společném formuláři. Oba formuláře bychom měli vložit do activity_main. Jelikož se ale umístění prvků už neřídí parametry bloku include, tak se activity_main zjednoduší:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- vlozit layout layout_1 -->
    <include layout="@layout/layout_1" />

    <!-- vlozit layout layout_2 -->
    <include layout="@layout/layout_2" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Výsledkem bude vnoření, které by se dalo znázornit takto:

```
<ConstraintLayout>
   <TextView />
   <Button />
   <EditText />
   <EditText />
</ConstraintLayout>
```

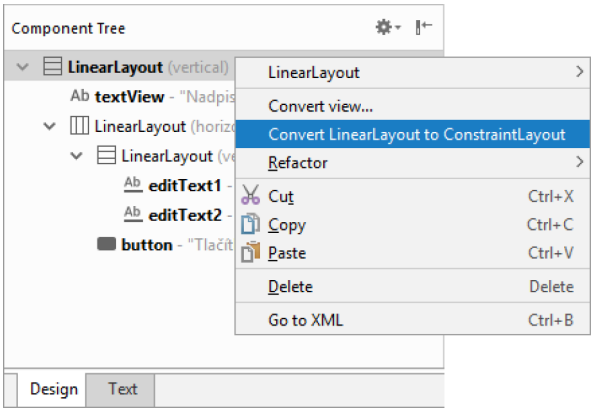
V emulátoru nebo na koncovém zařízení by měl výsledek, s první verzí activity_main, vypadat jako na obrázku 13.24, neboli úplně stejně jako v předchozím příkladu. Pouze se nepatrně zrychlí vykreslení formuláře.



Obrázek 13.24: Aplikace se dvěma vloženými layouty, spuštěná na emulátoru

13.6 Konverze layoutů

Editor Android Studia poskytuje možnost automatické konverze všech standardních typů layoutů do formátu Constraint Layout. Stačí kliknout pravým tlačítkem myši v okně Component



Obrázek 13.25: Konverze na Constraint Layout v okně Component Tree

Tree na layout a z kontextového menu vybrat Convert to Constraint Layout.

Tímto postupem je možné například z přísně hierarchického Linear Layoutu udělat plochý Constraint Layout s relativně umístěnými elementy. Constraint Layout však nedokáže nahradit všechny atributy ostatních kontejnerů, proto je zpravidla nutné výsledek ještě ručně upravit.

Při konverzi kontejneru na Constraint Layout se není potřeba obávat že se při experimentování ztratí již hotové rozložení prvků. Konverzi je možné vrátit zpět funkcí undo, resp. stisknutím tlačítek CTRL+Z.

Existuje ještě jedna snadná konverze - vzhledem k podobnosti Constraint a Relative Layoutu (k tomu se dostaneme později, v kapitole 15.1 Relative Layout), je možné změnit vnořený Constraint na Relative Layout, stačí pouze v XML kódu přepsat záhlaví kontejneru:

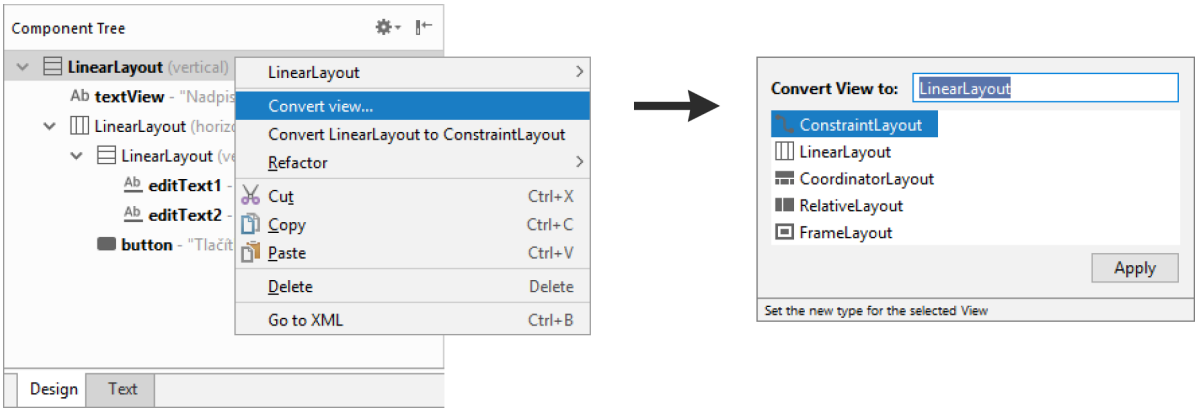
```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="138dp"
    android:layout_height="495dp"
    tools:layout_editor_absoluteY="8dp"
    tools:layout_editor_absoluteX="8dp">

</androidx.constraintlayout.widget.ConstraintLayout>
```

na:

```
<RelativeLayout
    android:layout_width="138dp"
    android:layout_height="495dp"
    tools:layout_editor_absoluteY="8dp"
    tools:layout_editor_absoluteX="8dp">

</RelativeLayout>
```

Je pochopitelně nutné dát pozor na prvky kontejneru, které by mohly obsahovat atribut vychýlení (Bias), který Relative Layout nepodporuje.

Zatímco konverze na Constraint Layout zajistila co možná nejpřesnější nastavení prvků formuláře tak, aby byly vizuálně podobné původnímu typu layoutu, nabízí Android Studio ještě jednu možnost konverze – tentokrát z jakéhokoliv na jakýkoliv jiný typ formuláře. K tomuto účelu slouží volba Convert View v kontextovém menu, které se opět vyvolá nad formulářem v okně Component Tree. A tento nástroj žádná nastavení prvků neřeší.



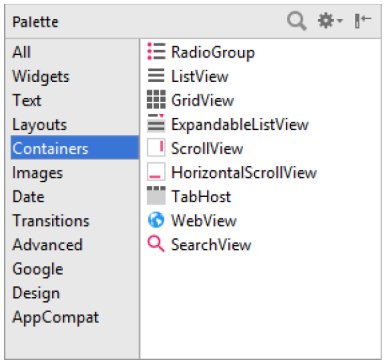
Obrázek 13.26: Konverze layoutu v okně Component Tree

Podobně jako jsme si před chvílí ukázali převedení Constraint Layout na Relative v XML, tak i v tomto případě se pouze změní hlavička layoutu. Co se stane s prvky formuláře už je na vývojáři.

Tímto způsobem lze konvertovat layouty mezi sebou prakticky neomezeně. Dokonce je možné do pole „Convert View to“ napsat název neexistujícího typu layoutu a Android Studio ho ochotně převede. Pochopitelně bez konverze jednotlivých atributů.

14 Kontejnery

Tento termín jsme hojně používali v kapitole o rozvržení prvků ve formuláři. Mělo to svůj význam, každý layout je neviditelným kontejnerem, který obsahuje další elementy uživatelského rozhraní Androidu. Google však v paletě komponent použil tento obecný název poměrně nešťastně pro množinu prvků, které jsou ve své podstatě také rozvržením, ale už nejsou tak komplexní, neviditelné a nelze je nazývat layouty. Pokud narazíme na termín kontejner, je nutné podle kontextu rozlišit, jestli jde o prvek typu Layout nebo Container. Prvky z obou kategorií jsou kontejnery.



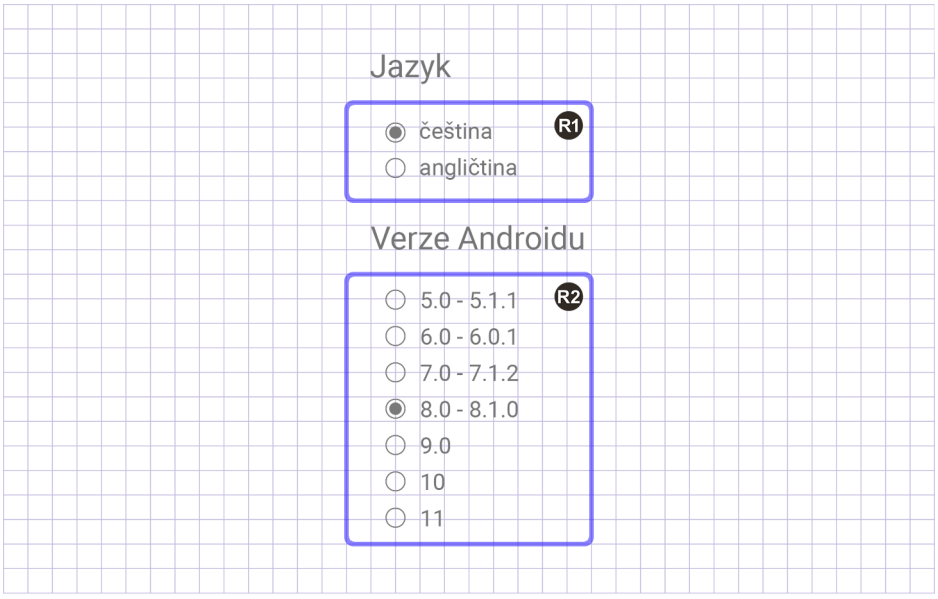
Obrázek 14.1: Paleta nástrojů - kontejnery

14.1 Radio Group

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt RadioGroup

Je to ten nejjednodušší kontejner, který má vývojář k dispozici. Jeho účel je přesně daný a jak název napovídá, slouží k seskupení prvků typu Přepínač (Radio Button). Na nic jiného se nedá využít. Zpravidla se uplatní v různých nastaveních nebo formulářích, kde je potřeba dát uživateli vybrat z několika konkrétních možností. Přepínače musí být ve skupinách, aby bylo jasné, mezi kterými prvky uživatel vybírá. Proto je není možné umístit do formuláře jen tak samy o sobě, ale jsou ohraničené právě kontejnerem Radio Group.

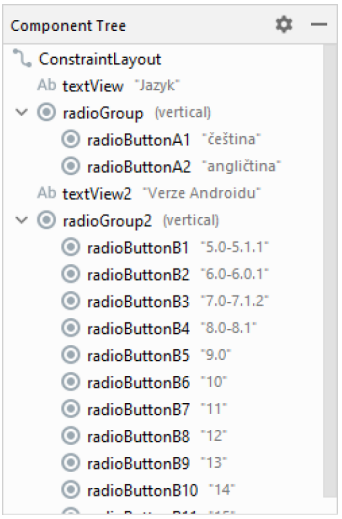
Radio Group je odvozený od kontejneru Linear Layout, pracuje se s ním tedy podobným způsobem. Na příkladu si ukážeme, jak vnořit přepínače do kontejneru Radio Group tak, aby vznikly dvě nezávislé skupiny.



Obrázek 14.2: Návrh formuláře s kontejnerem Radio Group

Použijeme Constraint Layout, do něj vložíme textové pole a pod něj umístíme kontejner Radio Group. Ten následně naplníme přepínači z palety komponent. Vše se dá udělat jednoduše a rychle v grafickém designéru. Potom totéž zopakujeme a vytvoříme druhou skupinu přepínačů.

V okně Component Tree, obrázek 14.3, vznikne snadno předvídatelná hierarchická struktura jednotlivých komponent.



Obrázek 14.3: Prvky formuláře v okně Component Tree

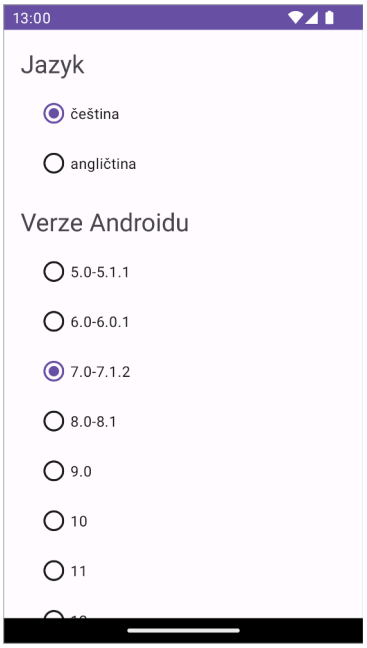
Výsledný XML kód:

```
<!-- prvni kontejner RadioGroup -->
<RadioGroup
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginLeft="32dp"
    android:layout_marginTop="8dp"
    app:layout_constraintLeft_toLeftOf="parent"
```

```
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView"
        android:id="@+id/radioGroup">

        <!-- prvni prepinac, prednastaveny -->
        <RadioButton
            android:id="@+id/radioButton"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:checked="true"
            android:text="čeština" />

        <!-- druhy prepinac -->
        <RadioButton
            android:id="@+id/radioButton2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="angličtina" />
    </RadioGroup>
```

Ze XML kódu je zobrazena pouze část popisující první kontejner. Druhý kontejner je shodný, pouze má více přepínačů.

Výsledek bude v emulátoru nebo na koncovém zařízení vypadat jako na obrázku 14.4.



Obrázek 14.4: Aplikace s kontejnerem typu Radio Group, spuštěná na emulátoru

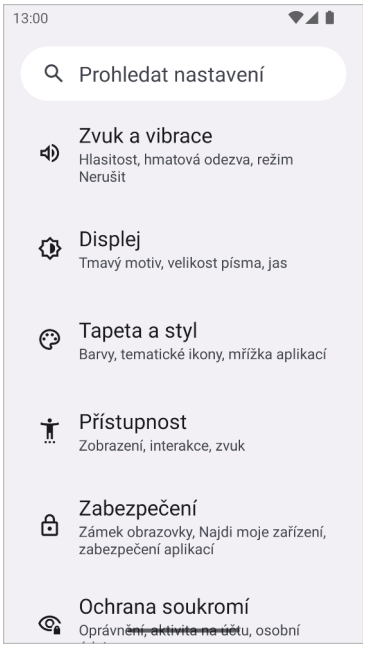
14.2 Recycler View

Zatímco předchozí kontejner měl jednoznačné a velmi omezené využití, kontejner Recycler View je jedním z nejuniverzálnějších. Jeho primárním účelem je seskupit a zobrazit seznam položek, umožnit jejich rolování a výběr. Položky můžou být úplně jednoduché, typicky textový seznam, ale i velmi komplikované, složené z obrázků, textů, tlačítek a jiných objektů. Seznam je přes virtuální adaptér napojený na zdroj dat, kterým bývá nejčastěji pole nebo databáze.

Recycler View není úplně jednoduchým typem kontejneru a práce s ním je pro začátečníky často obtížná. Je ho ale nutné zvládnout, pokud se u Androidu dá mluvit o základních kamenech aplikací, tak Recycler View je bezpochyby jedním z nich.

Seznamů typu RecyclerView obsahuje systém Android nepřeberné množství. V telefonu najdeme seznam kontaktů, historie hovorů nebo textových zpráv, záložky v internetovém prohlížeči, skladby v přehrávači, fotografie v galerii a tak dále.

Typickým příkladem je seznam položek z menu Nastavení na obrázku 14.5. Seznam Recycler View seskupí položky, z nichž každá je sestavená z ikony a dvou různých textových polí. Položek je více než kolik se vejde na displej, takže jimi může uživatel rolovat, poté si zvolit tu, která ho zajímá a ťuknutím přejít na další aktivitu.

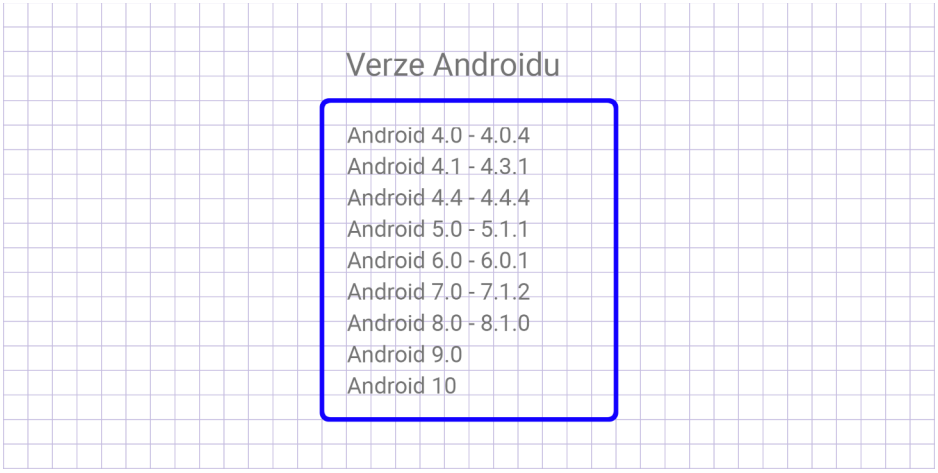


Obrázek 14.5: Příklad aplikace s kontejnerem typu Recycler View

14.2.1 Recycler View, 1. příklad

Vyzkoušíme si tu nejjednodušší variantu, udělat prostý výpis položek, zobrazený na základě vstupních dat reprezentovaných seznamem řetězců. Formulář položky bude obsahovat pouhé jedno textové pole. Vstupní data předáme prostřednictvím adaptéru kontejneru Recycler View a ten je pomocí holderu vyplní do formuláře položky.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt RecyclerView1



Obrázek 14.6: Návrh formuláře s kontejnerem List View

Co je potřeba udělat?

- vytvořit formulář s kontejnerem Recycler View

- vytvořit seznam hodnot k zobrazení

- navrhnout formulář položky

- napsat holder, který bude plnit formulář položky daty dodanými adaptérem

- vytvořit adaptér

- propojit data s adaptérem a adaptér s kontejnerem RecyclerView

Do formuláře (v tomto příkladu používejme Constraint Layout) vložíme textové pole a kontejner Recycler View. Rozměry nastavíme tak, aby byla vyplněna celá plocha displeje a aby se při natočení displeje velikost kontejneru přizpůsobila (viz kapitola 13.2 Constraint Layout). Kontejneru ještě ve vlastnostech nastavíme id recViewVerzeAndroidu a to je vše.

XML kód kontejneru bude tedy vypadat nějak takto:

```
<!-- RecyclerView recViewVerzeAndroidu
    sirka a vyska max, okraje 8dp,
    navazany zeshora na textove pole,
    z ostatnich stran na parent kontejner -->
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recViewVerzeAndroidu"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginBottom="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginEnd="8dp"
    app:layout_constraintTop_toBottomOf="@+id/textView"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent" />
```

Významnější část příkladu se odehraje ve třídě MainActivity. Právě tam totiž vytvoříme ArrayList<String>, neboli seznam textových řetězců, který bude zdrojem dat. Nebudeme dělat nic velkého, jednoduše vytvoříme jeho instanci a zapíšeme do ní hodnoty určené k zobrazení. Celé to dáme do metody kterou nazveme naplnitRecyclerView() a tu zavoláme z události onCreate(). Zdrojový kód bude vypadat nějak takto:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    naplnitRecyclerView();
}

private void naplnitRecyclerView()
{
    ArrayList<String> lstVerzeAndroidu = new ArrayList<String> ();
    lstVerzeAndroidu.add("Android 4.0 – 4.0.4");
    lstVerzeAndroidu.add("Android 4.1 – 4.3.1");
    ...
    lstVerzeAndroidu.add("Android 15");

}
```

Metodu naplnitRecyclerView() později doplníme ještě o inicializaci adaptéru a nastavení kontejneru RecyclerView. Adaptér ale zatím nechme být, vrátíme se k němu později.

Kromě zmíněného adaptéru potřebuje RecyclerView pro svoje fungování ještě dvě další části. Aby věděl, jak data zobrazit, musí mít definovaný formulář zobrazované položky a takzvaný holder, třídu, která každou zobrazenou položku naplní daty. RecyclerView si dokáže poradit i s položkami různých typů, dají se kombinovat řádky, které zobrazují různé typy formulářů v jednom seznamu, my se ale spokojíme s tím, že každá řádka bude tvořená stejným layoutem. A tím pádem bude naplněna prostřednictvím stejného holderu. Měnit se budou pouze zobrazovaná data.

Design položky se dá vytvořit relativně snadno. V okně projektu ve složce res/layout/ založíme nový soubor (kontextové menu New → Layout resource file). Pojmenujeme ho třeba ‚polozka'. Při návrhu použijeme některý z již známých kontejnerů, například Linear Layout. V něm uspořádáme objekty tak, aby odpovídaly naší představě o vizuální podobě položky seznamu. Při navrhování vzhledu je potřeba obzvlášť dávat pozor na to, aby bylo zobrazení formuláře efektivní, protože se na displeji odehraje tolikrát, kolik položek má seznam. Je velmi žádoucí, aby byl formulář co nejjednodušší a s minimem vnoření. Pro účely tohoto příkladu se spokojíme s jednoduchým formulářem, který bude obsahovat jedinou textovou položku s názvem txtPolozkaText.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@drawable/itemcolors"
    android:orientation="horizontal"
    android:padding="10dp">

    <TextView
        android:id="@+id/txtPolozkaText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:textSize="16sp" />

</LinearLayout>
```

Jedním z atributů je „android:background“, který odkazuje na soubor itemcolors. To je XML soubor, který obsahuje popis barev pozadí, použitých při různých uživatelských akcích, jak je popsáno v kapitole 9.6.1 Selector. Atribut není povinný, ale aby seznam působil interaktivnějším dojmem, hodí se tímto způsobem nastavit trošku jiné podbarvení během stisknutí položky. Připravíme si proto ve složce drawable soubor itemcolors.xml se selectorem, který bude definovat nastavení barev pro různé situace:

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@color/material_dynamic_neutral95"
                                                android:state_pressed="true" />
    <item android:drawable="@color/material_dynamic_neutral60"
                                                android:state_selected="true" />
    <item android:drawable="@android:color/white" />
</selector>
```

Abychom položku dokázali plnit daty, bude potřeba holder, který data přenese do polí formuláře. Máme jediné pole, takže holder může být poměrně jednoduchý. V podstatě bude stačit rozšířit k tomu určenou třídu RecyclerView.ViewHolder. V okně projektu, ve složce java, založíme nový soubor (kontextové menu New → java Class) s názvem RecycleHolder. Kód bude vypadat nějak takto:

```
public class RecyclerHolder extends RecyclerView.ViewHolder
    public TextView txtPolozkaText;

    public RecyclerHolder(View itemView) {
        super(itemView);

        txtPolozkaText = itemView.findViewById(R.id.txtPolozkaText);
    }
};
```

V podstatě jsme jen vytvořili public proměnnou txtPolozkaText, ke které je možné přistupovat z ostatních tříd a do ní jsme v konstruktoru vložili odkaz na textové pole formuláře. Teď se můžeme z adaptéru dostat prostřednictvím této veřejné proměnné na textové pole a zapsat na ně požadovanou hodnotu. Toho využijeme za chvíli, až budeme psát adaptér.

Holder je zároveň jedním z míst, kde lze nastavit událost onClick(), která bude reagovat na kliknutí na položku seznamu. Z hlediska struktury aplikace nebývá tím nejvhodnějším místem, ale pro účely jednoduché interakce spočívající v zobrazení informace pomocí Toast zprávy je dostačující. Další možnosti si ukážeme až v příštím příkladu. Zde tedy pouze rozšíříme třídu holderu o implementaci interface View.OnClickListener, aby byl schopný na kliknutí reagovat a přepíšeme událost onClick(), kterou necháme zobrazit Toast zprávu s textovou informací z položky seznamu.

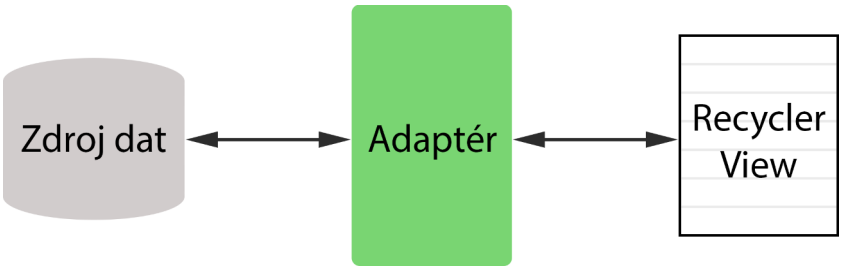
Rozšíření třídy o implementaci interface View.OnClickListener:

```
public class RecyclerHolder extends RecyclerView.ViewHolder
                                    implements View.OnClickListener {
...
}
```

Přepsání události onClick():

```
@Override
public void onClick(View view) {
    Toast.makeText(view.getContext(), txtPolozkaText.getText(),
                                            Toast.LENGTH_SHORT).show();
}
```

Dostáváme se k adaptéru. Jeho funkce je znázorněna na následujícím obrázku:



Obrázek 14.7: Propojení zdroje dat a kontejneru Recycler View pomocí adaptéru

Komponenta RecyclerView sice umí skvěle data zobrazovat, nedokáže si je však žádným rozumným způsobem zajistit. Naštěstí se ale dá napojit na adaptér, který naopak nic nezobrazí, ale zato má schopnost data získávat z rozličných zdrojů. V našem případě ze seznamu textových řetězců, který máme připravený ve třídě MainActivity.

Dá se říct, že napsat vlastní adaptér, který transformuje zdrojová data do položky kontejneru RecyclerView je na celém příkladu to nejsložitější. Nejdřív založíme Java soubor (kontextové menu New → Java Class). Zvolíme příhodný název, třeba ‚RecyclerAdapter'. První, co je potřeba udělat, je oddědit novou třídu od třídy RecyclerView.Adapter, ze které bude adaptér vycházet.

```
class RecyclerAdapter extends RecyclerView.Adapter<RecyclerHolder> { }
```

A hned nato vytvoříme konstruktor nové třídy, který uloží data předaná v parametru do pomocné proměnné androidVerze. Tím se zajistí, že k datům budou mít přístup i přepsané události, které budeme vyžívat.

```
private List<String> androidVerze;
public RecyclerAdapter (List<String> data){
    this.androidVerze = data;
}
```

Poté musíme předefinovat tři metody adaptéru. Nejprve událost onCreateViewHolder(), která se spustí vždy když se má na displeji zařízení poprvé zobrazit položka seznamu. Událost vrátí instanci holderu, který bude mít v parametru její formulář. To následně umožní komponentě RecyclerView položku vykreslit. Událost je volaná vždy při prvním zobrazení každé položky.

```
@Override
public RecyclerHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View rowItem = LayoutInflater.from(parent.getContext())
                                .inflate(R.layout.polozka, parent, false);
    return new RecyclerHolder(rowItem);
}
```

Další předefinovanou událostí je onBindViewHolder(). Ta se spouští v okamžiku přiřazení dat holderu položky, což je obvykle hned po onCreateViewHolder(). Položka se musí nejdřív vykreslit a následně doplnit daty. Zde využijeme veřejné proměnné holderu txtPolozkaText, která odkazuje na textové pole formuláře a rovnou jí nastavíme hodnotu z pole androidVerze.

```
@Override
public void onBindViewHolder(RecyclerHolder holder, int position) {
    holder.txtPolozkaText.setText(androidVerze.get(position));
}
```

Poslední předefinovanou metodou adaptéru je getItemCount(). Ta dělá jednoduchou věc, vrací počet položek seznamu. Využívá ji RecyclerView k tomu, aby věděl, kolik položek se bude vykreslovat. Takže pouze vrátíme počet položek v proměnné androidVerze.

```
@Override
public int getItemCount() {
    return androidVerze.size();
}
```

Tím jsme z nejhoršího venku, struktura adaptéru je vytvořena, máme formulář, který zobrazí data, máme i holder, který data do formuláře doplní. Teď už to stačí, spolu s daty, spojit do jednoho celku. To uděláme opět v MainActivity, v připravené metodě naplnitRecyclerView().

```
private void naplnitRecyclerView()
{
    ArrayList<String> lstVerzeAndroidu = new ArrayList<String> ();
    lstVerzeAndroidu.add("Android 4.0 – 4.0.4");
    lstVerzeAndroidu.add("Android 4.1 – 4.3.1");
    ...
    lstVerzeAndroidu.add("Android 15");

    // vytvorit instanci adapteru
    RecyclerAdapter adapter = new RecyclerAdapter(lstVerzeAndroidu);

    // najit na formulari kontejner RecyclerView
    RecyclerView recyclerView =
                        (RecyclerView)findViewById(R.id.recViewVerzeAndroidu);

    // vytvorit a priradit Layout Manager
    recyclerView.setLayoutManager(new LinearLayoutManager(this));

    // nastaveni oddelovace radku
    recyclerView.addItemDecoration(new DividerItemDecoration(
                        this, DividerItemDecoration.VERTICAL));

    // propojeni adapteru s kontejnerem RecyclerView
    recyclerView.setAdapter(adapter);
}
```

Po definici vstupních dat následuje zřízení adaptéru. Aby bylo možné adaptér použít, je ho potřeba nejdřív inicializovat, neboli vytvořit jeho instanci. Odkaz na ni umístíme do pomocné proměnné adapter. Pomocí findViewById() najdeme podle identifikátoru recViewVerzeAndroidu kontejner RecyclerView a i na něj uložíme do pomocné proměnné ukazatel.

V dalším kroku nastavíme LayoutManager. To je třída, která řídí způsob zobrazování položek v komponentě RecyclerView. Nabízí se hned několik možností, jak zobrazovaná data organizovat. Za základní je považovaný LinearLayoutManager, který z položek vytváří klasický seznam. Ten dále může být rolovatelný vertikálně nebo horizontálně. Dalším, často užívaným je GridLayoutManager, který sestaví výsledek do mřížky o definovaném počtu sloupců a řádek. Používá se zejména pro obrázkové náhledy. Kromě těchto dvou existují ještě další specializované managery, ale nám pro tuto chvíli bude stačit prvně zmíněný LinearLayoutManager, jehož instancí budeme řídit zobrazování položek seznamu.

LinearLayoutManager umožňuje skrolování jak vertikálně, tak i horizontálně. Přednastavený je směr vertikální, což nyní potřebujeme, takže kromě kontextu není potřeba vyplňovat žádné další parametry. Kompletní zápis nové instance LinearLayoutManager by však měl vypadat takto:

```
new LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false)
```

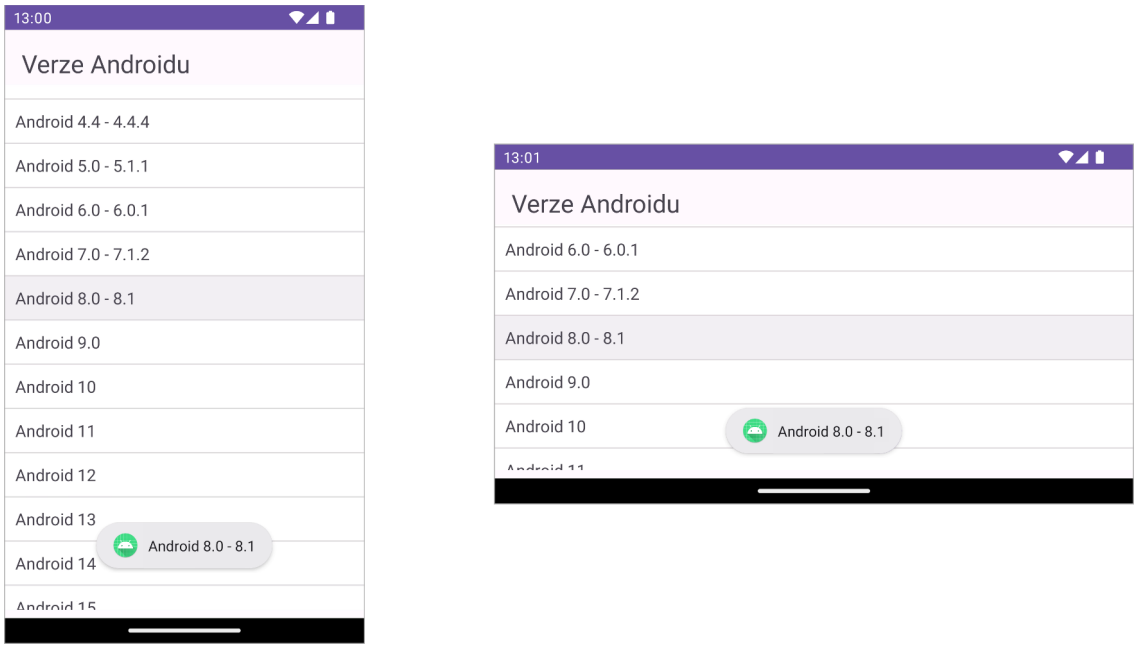
Tento LayoutManager bude položky zobrazovat lineárně, čili pod sebe, skrolovat je bude vertikálně, neboli nahoru a dolů a seřadí je od první nahoře po poslední dole (třetí parametr označuje převrácené pořadí). Toto obrácené pořadí se používá zejména u výpisu textových zpráv, kdy chceme mít ty nejnovější dole a starší necháme odrolovat nahoru. Kdybychom chtěli položky posouvat zleva doprava a zpět, tedy horizontálně a mít první položku na konci a poslední na začátku, vypadal by zápis takto:

```
new LinearLayoutManager(this, LinearLayoutManager.HORIZONTAL, true)
```

V dalším kroku nastavíme vizuální oddělovač jednotlivých položek seznamu. K tomu použijeme instanci třídy DividerItemDecoration, která má v parametru kromě kontextu ještě orientaci oddělovače. Zde pravděpodobně nedává smysl nic jiného, než pro vertikální skrolování použít vertikální oddělovač a pro horizontální skrolování horizontální oddělovač.

Posledním a zároveň nejdůležitějším krokem je propojení adaptéru s komponentou RecyclerView. To už se udělá poměrně prostě, dosazením ukazatele adapter pomocí setAdapter() do recyclerView.

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 14.8.



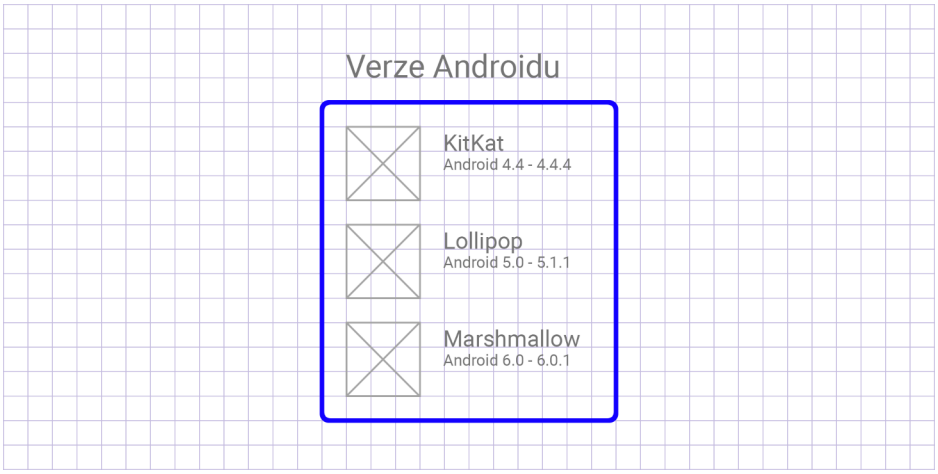
Obrázek 14.8: Aplikace s jednoduchým seznamem, realizovaným kontejnerem typu Recycler View

14.2.2 Recycler View, 2. příklad

První příklad nastínil základní možnosti kontejneru. Zároveň obsahuje v základní podobě vše potřebné pro vytváření i mnohem složitějších seznamů položek. V tomto příkladu se ho pouze pokusíme rozšířit a trochu zdokonalit, žádné významné nové funkce nebudeme přidávat.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt RecyclerView2

Cílem bude zobrazení komplexnější položky s obrázkem a dvěma textovými informacemi, názvem Androidu a s čísly jeho verzí. Také zkusíme upravit volání události kliknutí na položku tak, aby nemusela být umístěna v holderu položky. Řekněme tedy, že budeme chtít zobrazit seznam, který bude vypadat zhruba takto:



Obrázek 14.9: Návrh formuláře s kontejnerem Recycler View

Pro zjednodušení vyjdeme z předchozího příkladu a kód budeme pouze upravovat a doplňovat.

Co je potřeba udělat?

- uložit do resources vhodné obrázky
- rozšířit pole vstupních hodnot o názvy verzí Androidu
- upravit formulář položky
- přizpůsobit holder, aby formulář položky plnil daty
- upravit adaptér tak, aby parsoval vstupní data
- rozšířit adaptér o zachytávání události kliknutí na položku

Do složky res/drawable/ na disku počítače, je potřeba vložit loga jednotlivých verzí Androidu, nebo jiné vhodné obrázky. Měly by mít velikost odpovídající odhadnutému rozlišení displeje koncového zařízení. Příliš malé obrázky by působily kostrbatým dojmem a naopak příliš velké by zbytečně zvětšovaly aplikaci a zpomalovaly její běh.

Potom je potřeba doplnit v MainActivity v metodě naplnitRecyclerView vstupní data. Kromě čísel verzí Androidu budou potřebovat také názvy každé z verzí. Abychom si vystačili se stávajícím seznamem textových řetězců, doplníme názvy do každé položky tak, že je jenom oddělíme dvojtečkou. Na samostatné textové informace si je potom rozdělí sám adaptér.

```
private void naplnitRecyclerView()
{
    ArrayList<String> lstVerzeAndroidu = new ArrayList<String> ();
    lstVerzeAndroidu.add("Android 4.0 - 4.0.4:Ice Cream Sandwich");
    lstVerzeAndroidu.add("Android 4.1 - 4.3.1:Jelly Bean");
    ...
    lstVerzeAndroidu.add("Android 10 a vyšší:Android 10");
}
```

Design položky upravíme relativně snadno. Stačí otevřít formulář v designéru a přidat dvě komponenty, ImageView a TextView. Objekty uspořádáme tak, aby odpovídaly naší představě o vizuální podobě položky seznamu. Při navrhování vzhledu je potřeba obzvlášť dávat pozor na to aby bylo zobrazení formuláře efektivní, protože se na displeji odehraje tolikrát, kolik položek má seznam. Je velmi žádoucí, aby byl formulář co nejjednodušší a s minimem vnoření.

Ze třídy RecyclerHolder nejdřív odstraníme událost kliknutí na tlačítko. Nebude potřeba, později ji nahradíme novou událostí v adaptéru. Do holderu přidáme další veřejné proměnné pro udržování ukazatelů na nové obrázkové a textové pole. Celkem tedy budeme mít tři, txtNazev, txtVerze a imgLogo. Ty je potřeba v konstruktoru třídy naplnit. Přes tyto proměnné bude adaptér plnit formulář daty. Celý RecyclerHolder bude vypadat takto:

```
public class RecyclerHolder extends RecyclerView.ViewHolder {

    public TextView txtNazev;
    public TextView txtVerze;
    public ImageView imgLogo;

    public RecyclerHolder(View itemView) {
        super(itemView);

        // nalezeni jednotlivych objektu formulare a prirazeni do promennych
        txtNazev = itemView.findViewById(R.id.txtName);
```

```
        txtVerze = itemView.findViewById(R.id.txtVerze);
        imgLogo = itemView.findViewById(R.id.imgLogo);
    }
}
```

Třída RecyclerHolder je tedy připravena plnit hodnoty do formuláře. Teď už je jenom na adaptéru, aby jí dokázal data předat ve vhodné formě. To znamená, že pomocí každé ze tří definovaných veřejných proměnných holderu předá adaptér tu správnou hodnotu přímo do textového pole nebo do ImageView komponenty.

V adaptéru slouží k zpracování dat událost onBindViewHolder(), která má prostřednictvím parametru přístup do holderu a zároveň získává informaci o čísle položky, kterou právě zpracovává. Zejména číslo položky nás bude zajímat, podle něj budeme v datech dohledávat, co vlastně máme zobrazit. K přístupu k datům z pole slouží metoda get(), které dáme jako parametr číslo položky. Toto číslo se shoduje s pozicí v poli vstupních dat, takže pro metodu get() není těžké hodnotu z pole vzít a vrátit ji. Uložíme ji do proměnné typu String, kterou nazveme strItemPolozka.

To stačí k tomu, aby si ze seznamu androidVerzeData vzal tu správnou položku, tak jak jsme ji definovali v MainActivity. Tedy jako dvě informace spojené do jednoho řetězce, oddělené dvojtečkou.

```
String strPolozka = androidVerzeData.get(pozice);
```

Dál už je jenom na fantazii vývojáře, jak formulář pomocí vstupních dat vyplní. V našem příkladu jsou v poli dat uloženy vždy dvojice informací – číslo a název verze Androidu, odděleny dvojtečkou. Zároveň jsou ve složce res/drawable/ uložena loga jednotlivých verzí Androidu, přičemž název loga odpovídá názvu verze Androidu. Takže ze strPolozka získáme název a číslo verze, přiřadíme je do polí ve formuláři, podle názvu dohledáme logo a to přiřadíme do kontejneru ImageView. Celý kód události onBindViewHolder() bude vypadat takto:

```
@Override
public void onBindViewHolder(RecyclerHolder holder, int pozice) {

    // barva pozadi polozky
    holder.itemView.setSelected(holder.itemView.isSelected());

    // ziskani retezce z pole, podle pozice
    String strPolozka = androidVerzeData.get(pozice);

    // rozdeleni retezce na casti
    String[] strVerze = strPolozka.split(":");

    // zpristupneni resources
    Resources res = rowItem.getResources();

    // jmeno resource s logem androidu
    String resNazev = strVerze[1].replaceAll(" ", "").toLowerCase();

    // id resource s logem androidu
    int imgLogoId = res.getIdentifier(resNazev, "drawable",
                                    rowItem.getContext().getPackageName());

    // predani hodnot prostrednictvim holderu k zobrazeni
    holder.txtNazev.setText(strVerze[1]);
    holder.txtVerze.setText(strVerze[0]);
```

```
    holder.imgLogo.setImageResource(imgLogoId);
}
```

Nyní můžeme aplikaci spustit na emulátoru nebo zařízení. Měla by zobrazovat data i s logy Androidu podle návrhu. Jinak by se měl seznam položek, s výjimkou kliknutí, chovat stejně jako se choval seznam z minulého příkladu.

Jelikož bylo v minulém příkladu umístění události pro zachycení kliknutí v holderu sice jednoduché, ale pro další práci, jako je například otevření jiné aktivity, poměrně nepraktické, přemístíme ho do adaptéru.

Ve třídě RecyclerAdapter přidáme novou událost

```
private View.OnClickListener mItemClick = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        TextView txtPolozkaText = (TextView)v.findViewById(R.id.txtName);

        // zobrazeni toast zpravy
        Toast.makeText(v.getContext(), txtPolozkaText.getText(),
                                                Toast.LENGTH_SHORT).show();
    }
};
```

Aby mohla být událost kliknutí vyvolaná, zaregistrujeme ji v již existující události adaptéru onCreateViewHolder():

```
rowItem.setOnClickListener(mItemClick);
```

Když teď spustíme aplikaci, bude reagovat na kliknutí na položku seznamu vypsáním Toast zprávy, která je vyvolaná z adaptéru.

Adaptér se ale stále nemusí jevit jako ideální místo pro zpracování události. Přece jen je určený spíše k transformaci dat než k akcím, které nezřídka vedou k otevírání nových aktivit a podobně. Hodilo by se, tuto událost přenést do MainActivity a tam už ji zpracovat podle potřeby. K tomu použijeme malý trik. Vyvolání události necháme v adaptéru, ale místo aby ji plnohodnotně zpracoval, tak ho pouze necháme zavolat veřejnou metodu, kterou si k tomuto účelu připravíme v samotné MainActivity.

Veřejná metoda onClickCalled() v MainActivity:

```
// metoda volana z RecycleAdapter
public void onClickCalled(String anyValue) {
    Toast.makeText(this, anyValue, Toast.LENGTH_SHORT).show();
}
```

Prozatím se spokojíme s tím, že opět zobrazí pouze Toast zprávu s informací o verzi Androidu. Aby ji dokázal zavolat, je potřeba dát nějakým způsobem adaptéru vědět, že se tato metoda v MainActivity nachází. To uděláme tak, že v RecyclerAdapter přidáme pomocnou proměnnou mainActivity, která bude udržovat ukazatel na volající třídu MainActivity. Ukazatel předáme v konstruktoru parametrem volajici:

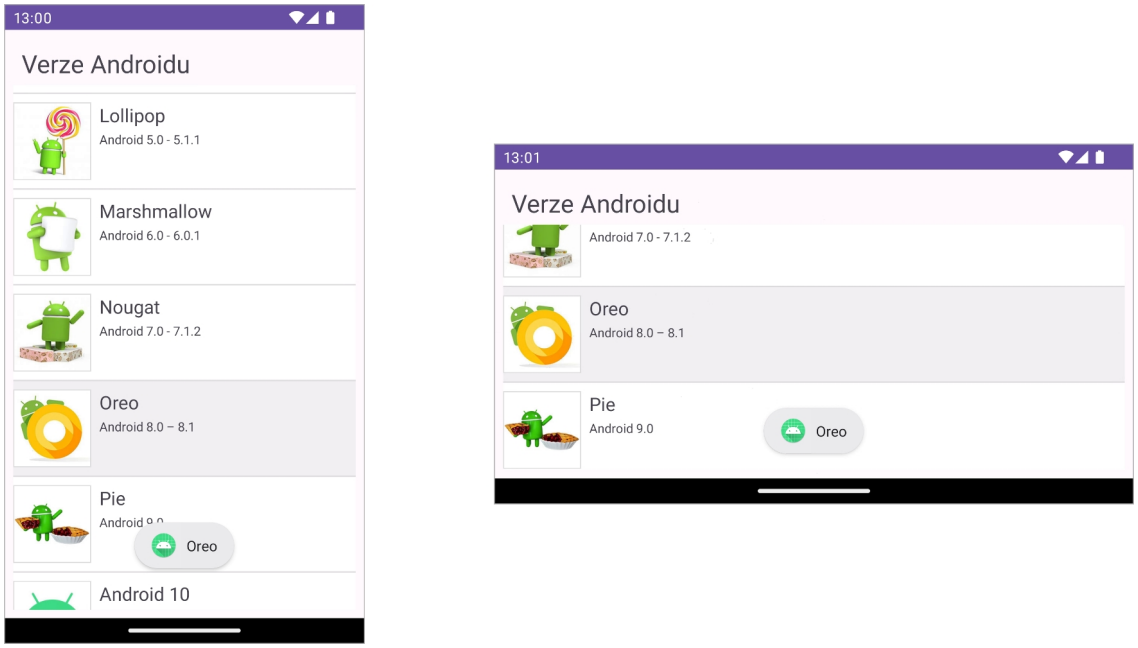
```
private MainActivity mainActivity;
```

```
// konstruktor adapteru
public RecyclerAdapter(List<String> data, MainActivity volajici) {
    androidVerzeData=data;
    mainActivity=volajici;
}
```

Potom už bude stačit upravit posluchač OnClickListener tak, aby připravenou metodu v MainActivity zavolal:

```
private View.OnClickListener mItemClick = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        TextView txtPolozkaText = (TextView)v.findViewById(R.id.txtName);

        // volani metody tridy MainActivity
        mainActivity.onClickCalled(txtPolozkaText.getText().toString());
    }
};
```

Na koncovém zařízení nebo v emulátoru by měl výsledek vypadat jako na obrázku 14.10.

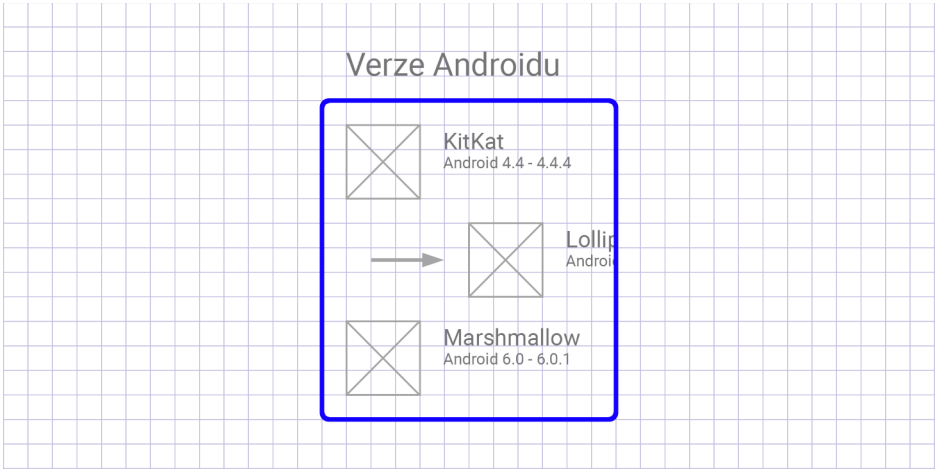


Obrázek 14.10: Aplikace se složitějším seznamem, realizovaným kontejnerem typu Recycler View

14.2.3 Recycler View, 3. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt RecyclerView3

Kromě prostého zobrazení položek všeho druhu dává Recycler View velký prostor pro grafické zpracování uživatelských operací. I v samotném operačním systému Android jsou často k vidění zejména dvě základní, swipe, to je jakési odsunutí položky do strany a tím její odstranění ze seznamu a drag and drop, neboli přesunutí položky na jiné místo v seznamu. Zkusíme upravit předchozí příklad tak, aby bylo možné položky ze seznamu odstraňovat jejich odsunutím do strany, mimo displej zařízení.



Obrázek 14.11: Návrh formuláře s kontejnerem Recycler View, šipka znázorňuje odsunutí položky

Opět tedy vyjdeme z předchozího příkladu a kód budeme pouze upravovat a doplňovat.

Co je potřeba udělat?

- definovat událost zpětného volání při posunutí položky

- zaregistrovat událost a propojit ji s Recycler View

Tentokrát toho tedy nemusíme dělat moc. Využijeme možnosti standardní Android knihovny ItemTouchHelper, která umí vše podstatné sama zařídit. Pouze vytvoříme událost zpětného volání typu SimpleCallback, které v parametru předáme nezbytné informace o tom, jak se má položka seznamu chovat. Android Studio během psaní události samo doplní metody k přepsání, onMove() a onSwiped().

Jelikož je zpětné volání SimpleCallback společné pro odsouvání (swipe) i přesouvání (drag and drop) položek, aby bylo možné obě tyto jeho funkce kombinovat, má i svoje dva parametry společné pro oba typy volání. První slouží pro přesouvání, což teď nevyužijeme, takže předáme nulu. Druhý parametr nastavuje možné směry odsunutí. Odsunutí doleva se nastaví binárním příznakem výčtového typu ItemTouchHelper.LEFT, Odsunutí doprava ItemTouchHelper.RIGHT. Příznaky se dají kombinovat pomocí OR operátoru, takže lze nastavit i odsunutí na libovolnou stranu.

```
ItemTouchHelper.SimpleCallback itemTouchCallback =
                  new ItemTouchHelper.SimpleCallback(0,
                        ItemTouchHelper.START | ItemTouchHelper.END) {
    @Override
    public boolean onMove(RecyclerView recyclerView,
                  RecyclerView.ViewHolder viewHolder,
                        RecyclerView.ViewHolder target) {
        return false;
    }

    @Override
    public void onSwiped(RecyclerView.ViewHolder viewHolder, int direction) {
        lstVerzeAndroidu.remove(viewHolder.getAdapterPosition());
        adapter.notifyDataSetChanged();
```

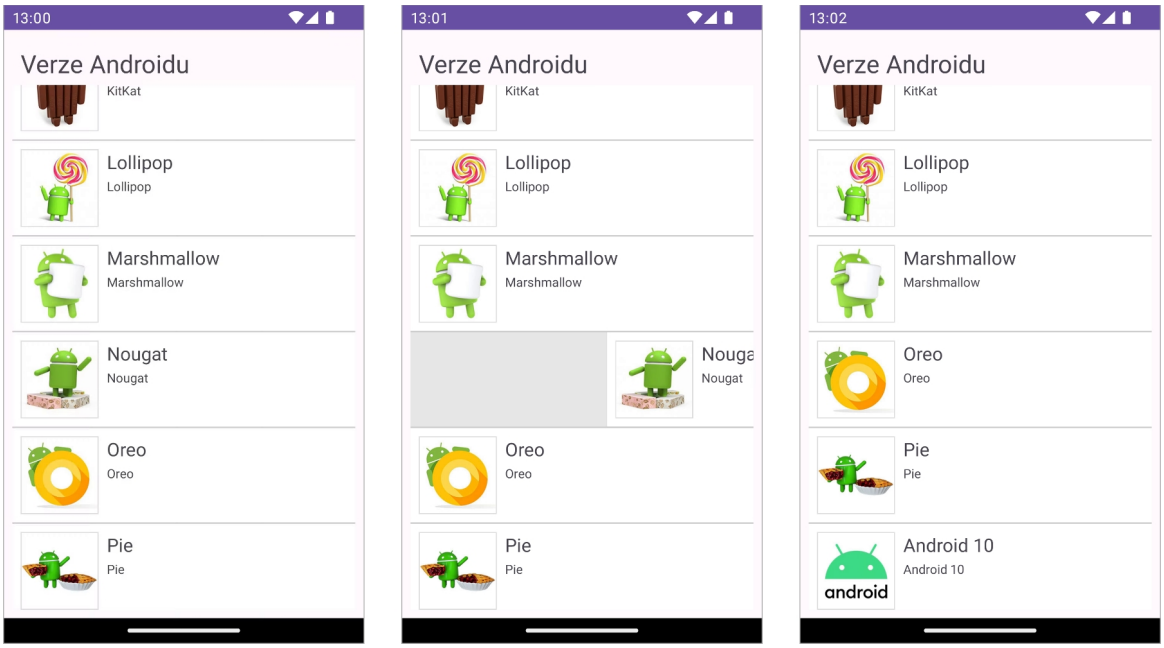
```
    }
};
```

Rozhraní zpětného volání SimpleCallback vyžaduje dvě přepsané události. První, onMove() je vyvolaná vždy při přesunutí položky na jiné místo v seznamu. Což teď nepotřebujeme, takže v ní pouze vrátíme hodnotu false a žádnou další operaci neprovádíme. Druhá událost, onSwiped() je vyvolaná po odsunutí položky a tedy jejím odstraněním ze seznamu. Je to tak proto, aby bylo možné programově reagovat na její vizuální smazání i tím, že zařídíme smazání příslušných dat. To je v našem případě jednoduché, prostě odstraněnou položku smažeme i v seznamu verzí Androidu lstVerzeAndroidu. V tom nám pomůže první parametr události onSwiped(), viewHolder, ze kterého se dá zjistit pozice odstraněné položky. Kdybychom chtěli rozdílně zpracovávat odsunutí položky vlevo a vpravo, lze využít parametr direction.

Zpětné volání musíme zaregistrovat v události onCreate() aktivity tak, že si vytvoříme instanci třídy ItemTouchHelper a tu potom propojíme s komponentou RecyclerView.

```
ItemTouchHelper itemTouchHelper = new ItemTouchHelper(itemTouchCallback);
itemTouchHelper.attachToRecyclerView(recyclerView);
```

Po spuštění na koncovém zařízení nebo v emulátoru by mělo být možné položky ze seznamu odstraňovat jejich odsunutím mimo plochu formuláře. Výsledek bude vypadat jako na obrázku 14.12.

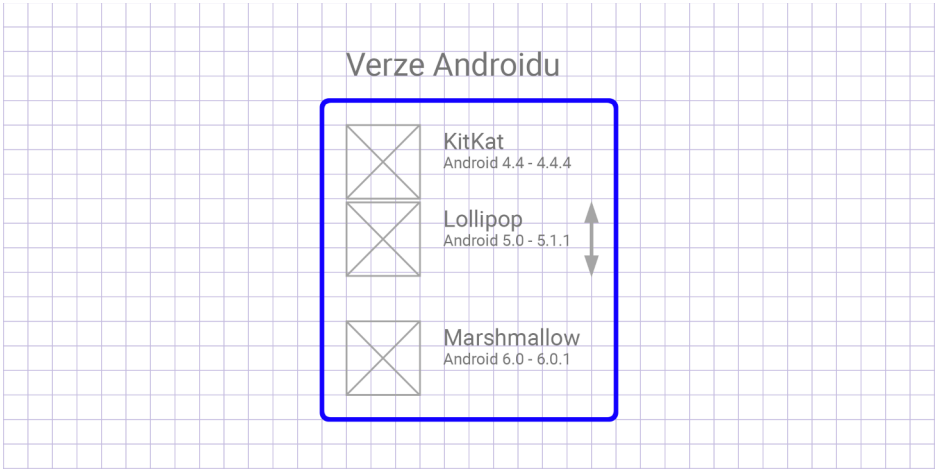


Obrázek 14.12: Odsunutí položky seznamu, realizovaného kontejnerem typu Recycler View

14.2.4 Recycler View, 4. příklad

Stejně jako jsme v minulém příkladu položku ze seznamu odstranili jejím odsunutím do strany, můžeme Recycler View nastavit tak, aby přetažením položky změnily svoje pořadí v seznamu. Vyjdeme ze stejného příkladu jako minule a pouze jej upravíme, aby umožňoval přetažení položky na jiné místo.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt RecyclerView4



Obrázek 14.13: Návrh formuláře s kontejnerem Recycler View, šipka znázorňuje přesun položky na jiné místo v seznamu

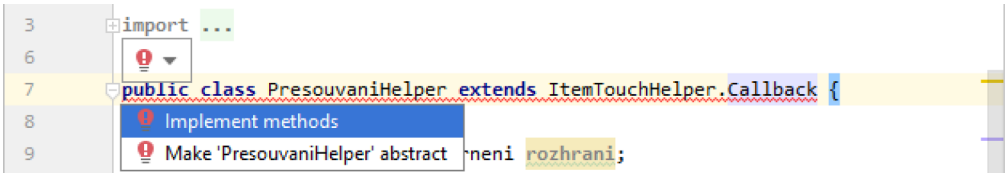
Na rozdíl od předchozího příkladu situaci tentokrát trochu zkomplikujeme tím, že vytvoříme pomocnou třídu, která bude obsahovat události zpětného volání a bude definovat rozhraní, pomocí nějž upozorní adaptér na změnu v pořadí položek. Samotnou změnu pořadí zařídíme v adaptéru, který je pro tyto operace vhodnější než MainActivity.

Celé by to tedy mělo fungovat tak, že pomocná třída bude obsahovat události volané systémem při přesunu položky. Ale jediné, co následně udělá, bude volání metod rozhraní. Toto rozhraní si implementujeme v adaptéru, takže nakonec skončí volání události právě v něm. Zní to zbytečně komplikovaně, ale jde jenom o jednoduchou techniku, kdy pomocná třída volá metody, o kterých dopředu neví, kde budou použity.

Co je potřeba udělat?

- vytvořit pomocnou třídu pro přesouvání s rozhraním pro upozornění o konci přesunu
- implementovat v adaptéru rozhraní pomocné třídy
- zaregistrovat událost a propojit ji s Recycler View

Založíme si tedy novou třídu (kontextové menu New → Java Class). Dáme jí nějaký příhodný název, třeba ‚PresouvaniHelper‘. Jako první ji musíme oddědit od třídy ItemTouchHelper.Callback, ze které bude vycházet a kterou bude rozšiřovat. Při zápisu rozšíření pomocník Android Studia nabídne vytvoření potřebných událostí k přepsání. Necháme si je tedy vygenerovat a rovnou získáme metody getMovementFlags(), onMove() a onSwiped().



Obrázek 14.14: Vygenerování událostí třídy ItemTouchHelper.Callback

Konstruktor třídy bude zahrnovat parametr ‚rozhrani‘, který bude říkat kdo třídu použije. V našem případě to bude adaptér, ale k tomu se ještě dostaneme. Obsah parametru uložíme do pomocné proměnné. Celá třída bude vypadat takto:

```
public class PresouvaniHelper extends ItemTouchHelper.Callback {

    private PresunUkoncenUpozorneni rozhrani;
    public PresouvaniHelper(PresunUkoncenUpozorneni rozhrani) {
        this.rozhrani = rozhrani;
    }

    @Override
    public int getMovementFlags(RecyclerView recyclerView,
                                RecyclerView.ViewHolder viewHolder) {
        int dragFlags = ItemTouchHelper.UP | ItemTouchHelper.DOWN;
        return makeMovementFlags(dragFlags, 0);
    }

    @Override
    public boolean onMove(RecyclerView recyclerView, RecyclerView.ViewHolder
                                viewHolder, RecyclerView.ViewHolder target)
    {
        rozhrani.polozkaPresunuta(viewHolder.getAdapterPosition(),
                                target.getAdapterPosition());
        return true;
    }

    @Override
    public void onSwiped(RecyclerView.ViewHolder viewHolder, int direction) {
    }

    public interface PresunUkoncenUpozorneni {
        void polozkaPresunuta(int puvodniPozice, int novaPozice);
        void polozkaOdsunuta(int pozice);
    }}
```

V události getMovementFlags() si stejně jako v předchozím příkladu pouze binárně nastavíme příznak pro typ operace a směr pohybu položek. Tentokrát nás zajímá operace drag and drop, tedy první parametr metody makeMovementFlags() a směr pohybu nahoru a dolů. Samozřejmě nic nebrání, abychom zde zkombinovali přesun položek s jejich mazáním pomocí odsunutí do strany. Pro jednoduchost ale zůstaneme pouze u změny jejich pořadí.

Důležité je rozhraní PresunUkoncenUpozorneni, na samém konci třídy. To definuje metody, které budeme volat jakmile bude operace přesunu (nebo odsunutí) položky dokončena. Jde o rozhraní, takže metody nemají tělo, které bude definované až v místě implementace. Zde pouze říkáme, jak se metody budou jmenovat a jaké mají mít parametry.

Události onMove() a onSwiped(). První z nich, onMove(), bude volaná při přesunu položky. Vyhodnocení operace ale chceme nechat na adaptéru, proto zde pouze zavoláme metodu rozhraní polozkaPresunuta() a do parametrů ji dáme původní a nové umístění položky v seznamu. U onSwiped() nemusíme dělat nic, protože odsunutí položky nepoužijeme. Kdybychom se ho ale rozhodli také implementovat, bylo by v něm stejným způsobem volání metody polozkaOdsunuta().

Rozhraní PresouvaniHelper.PresunUkoncenUpozorneni implementujeme v adaptéru tak, že je přidáme do záhlaví deklarace třídy:

```
class RecyclerAdapter extends RecyclerView.Adapter<RecyclerHolder>
                        implements PresouvaniHelper.PresunUkoncenUpozorneni {
```

Rovnou můžeme nechat vygenerovat metody polozkaPresunuta() a polozkaOdsunuta(), které je nutné přepsat.

```
@Override
public void polozkaPresunuta(int puvodniPozice, int novaPozice) {
    String presouvanaPolozka = androidVerzeData.get(puvodniPozice);
    androidVerzeData.remove(puvodniPozice);
    androidVerzeData.add(novaPozice, presouvanaPolozka);

    notifyItemMoved(puvodniPozice, novaPozice);
}

@Override
public void polozkaOdsunuta(int pozice) {
}
```

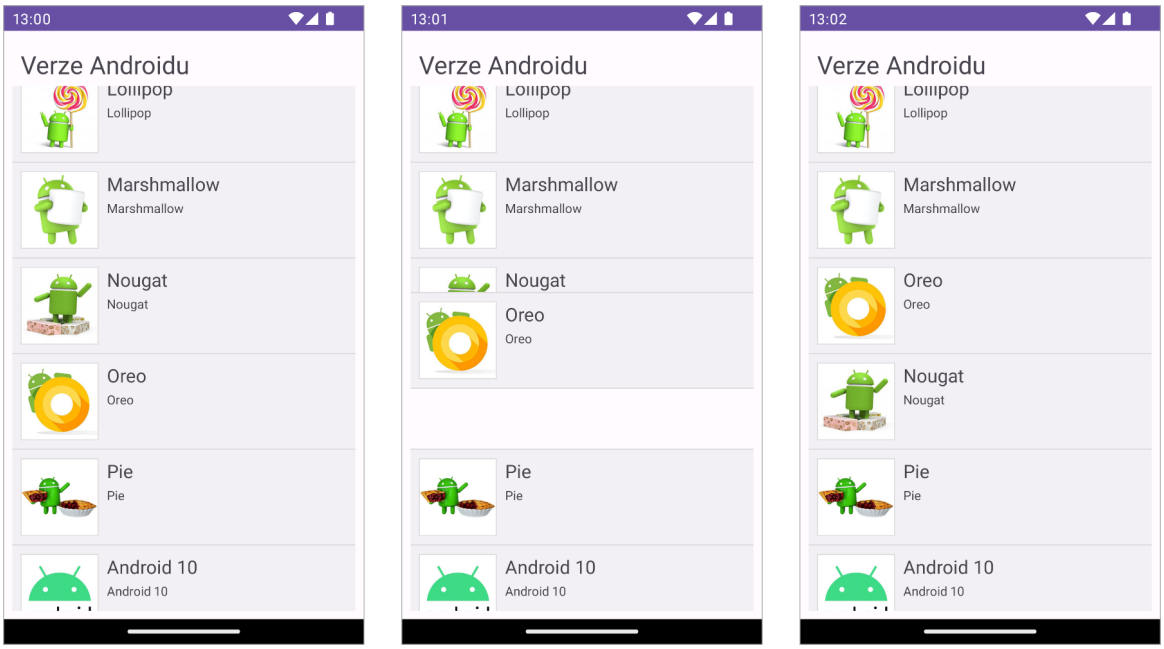
Metoda polozkaPresunuta() prohodí v seznamu verzí androidVerzeData vcelku prostým způsobem požadované položky a zavolá metodu notifyItemMoved() třídy RecyclerView, která zařídí správné vykreslení na displeji. V případě metody polozkaOdsunuta() není potřeba nic dělat, stačí ji nechat prázdnou, použili bychom ji pouze v případě, že bychom kombinovali přesouvání položek s jejich odsunutím do strany.

Poslední nezbytnou záležitostí je registrace události a propojení s komponentou Recycler View v události onCreate() třídy MainActivity:

```
PresouvaniHelper presouvaniHelper = new PresouvaniHelper(adapter);
ItemTouchHelper touchHelper = new ItemTouchHelper(presouvaniHelper);
touchHelper.attachToRecyclerView(recyclerView);
```

Vytvoříme instanci pomocné třídy, v parametru ji předáme odkaz na adaptér, aby věděla koho volat, hned poté vytvoříme instanci třídy ItemTouchHelper a pomocnou třídu mu předáme v parametru. Nakonec to celé připojíme ke komponentě RecyclerView.

Po spuštění na koncovém zařízení nebo v emulátoru by mělo být možné měnit pořadí položek v seznamu jejich přesouváním nahoru a dolů, jak je znázorněno na obrázku 14.15.



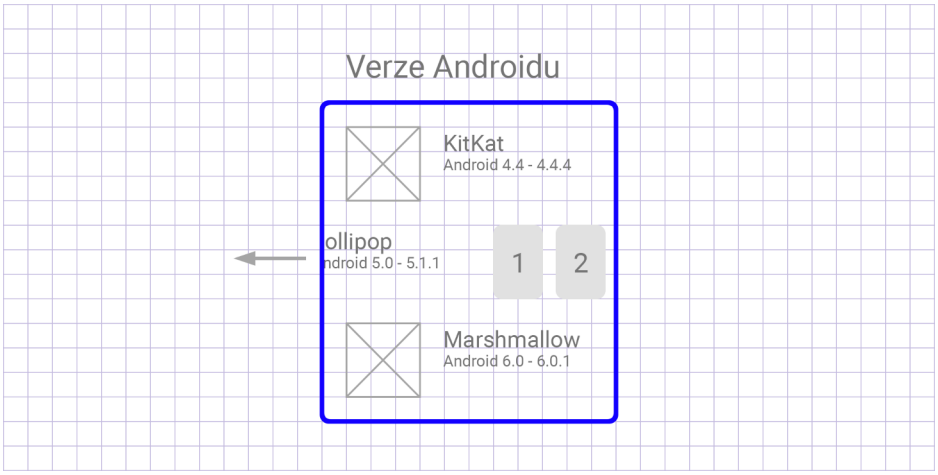
Obrázek 14.15: Přesunutí položky seznamu, realizovaného kontejnerem typu Recycler View

14.2.5 Recycler View, 5. příklad

V jednom z předchozích příkladů jsme pomocí operace swipe mazali položku seznamu jejím odsunutím do strany. Recycler View má ale kromě prostého odsunutí také možnost položku posunout a zobrazit na jejím místě další objekty, nejčastěji graficky naznačená tlačítka.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt RecyclerView5

Toho už nejde docílit pouhým nastavením vlastností položky, tentokrát bude nutné sáhnout ve větší míře k programování. Nejen události, ale i samotné zobrazení objektů bude nutné popsat pomocí Java kódu. Příklad bude tedy o poznání složitější. Cílem je možnost posunout položku do strany a na jejím místě zobrazit dvě tlačítka. Tlačítka nebudou grafická, budou tvořena barevným obdélníkem s textem uprostřed. Tento příklad bude navazovat zejména na 14.2.3 Recycler View, 3. příklad.



Obrázek 14.16: Návrh formuláře s kontejnerem Recycler View, šipka znázorňuje posunutí položky a zobrazení dvou tlačítek

Co je potřeba udělat?

- vytvořit pomocnou třídu pro posouvání položky
- implementovat v adaptéru rozhraní pomocné třídy
- zaregistrovat událost a propojit ji s Recycler View

Základem bude třída PosouvaniHelper, ve které budou soustředěny všechny rutiny podstatné pro ovládání a zobrazení položky. Tato třída bude o dost komplikovanější než pomocné třídy v předchozích příkladech, všechno ostatní ale už bude dost podobné.

Vytvoříme tedy abstraktní třídu PosouvaniHelper a rovnou ji rozšíříme o rozhraní zpětného volání SimpleCallback(), které už známe z předchozích příkladů.

```
public abstract class PosouvaniHelper extends ItemTouchHelper.SimpleCallback {
}
```

Jde o abstraktní třídu, která bude obsahovat abstraktní metody nakreslitTlacitko1 a nakreslitTlacitko2. K jejich implementaci se dostaneme až ve třídě MainActivity. K tomu si ještě připravíme interface posluchače události kliknutí na tlačítko.

```
public abstract NakreslitTlacitko nakreslitTlacitko1(
                                    RecyclerView.ViewHolder viewHolder);
public abstract NakreslitTlacitko nakreslitTlacitko2(
                                    RecyclerView.ViewHolder viewHolder);

public interface NakreslitTlacitkoClickListener {
    void onClick(int pos);
}
```

Třída NakreslitTlacitko se postará o grafické vykreslení tlačítek a musíme ji samozřejmě také vytvořit. Bude mít sadu parametrů, kterými předáme text tlačítka, jeho barvu a odkaz na posluchače události kliknutí, jehož interface jsme si připravili. Tyto parametry následně uchováme v lokálních proměnných. Pokud by tlačítka byla grafická, předali bychom do parametrů také ID resource příslušného obrázku. Teď si ale nebudeme situaci zbytečně komplikovat a spokojíme se s konfigurovatelným textem a barvou.

```
public static class NakreslitTlacitko {
    private String textTlacitka;
    private int barvaTlacitka;

    public NakreslitTlacitko(String textTlacitka, int barvaTlacitka,
                             NakreslitTlacitkoClickListener clickListener) {
        this.textTlacitka = textTlacitka;
        this.barvaTlacitka = barvaTlacitka;
        this.clickListener = clickListener;
    }

    ...

}
```

Třída bude mít metodu pro vykreslení tlačítka a událost pro obsluhu kliknutí na ně. Metodu pojmenujme kreslitTlacitko(), v parametru se bude předávat canvas, což je plocha na kterou se má kreslit, umisteni pro vymezení obdélníkové oblasti a pozice s číslem položky, do které se tlačítka vykreslují.

```
private int pozice;
private RectF clickRegion;

public void kreslitTlacitko(Canvas c, RectF umisteni, int pozice) {

    Paint p = new Paint();

    // vykresleni pozadi tlacitka
    p.setColor(barvaTlacitka);
    c.drawRect(umisteni, p);


    // sem prijde vykresleni textu tlacitka


    clickRegion = umisteni;
    this.pozice = pozice;
}
```

Metoda je zatím jednoduchá. Do oblasti vymezené parametrem rect nakreslí obdélník v barvě barvaTlacitka, do pomocných proměnných pozice a clickRegion uloží pozici položky a oblast, která má být senzitivní na kliknutí. Takto dosáhneme vykreslení barevného obdélníku na místo definované parametrem umisteni, který představuje tlačítko.

Komplikace se skrývá až v poznámce o vykreslení textu. Nakreslit obdélník je výrazně jednodušší operace, než do něj vložit text. Ten totiž musíme nejdřív vykreslit, potom změřit jeho velikost a až potom vypočítat umístění.

```
p.setColor(Color.WHITE);
p.setTextSize(Resources.getSystem().getDisplayMetrics().density * 12);

Rect r = new Rect();
p.setTextAlign(Paint.Align.LEFT);
p.getTextBounds(textTlacitka, 0, textTlacitka.length(), r);

float x = umisteni.width() / 2f - r.width() / 2f;
float y = umisteni.height() / 2f + r.height() / 2f;

c.drawText(textTlacitka, umisteni.left + x, umisteni.top + y, p);}
```

První dva řádky nastaví barvu písma na bílou a velikost na 12sp. Potom je potřeba vytvořit obdélníkovou oblast, do které se vloží text. Toto je trik, který umožní zjistit rozměry textu, protože oblast se mu svojí velikostí přizpůsobí. Výškou textu tedy bude r.height() a šířkou r.width(). Následuje výpočet souřadnic textu. Oblast, do jejíhož středu chceme text vložit je v proměnné umisteni. Počátek, neboli souřadnice [0, 0] má v levém horním rohu. Oblast r, v níž je umístěný text, má proti tomu počátek v levém dolním rohu. Vodorovnou souřadnici tedy najdeme tak, že od poloviny šířky umisteni odečteme polovinu šířky r. A podobně ve svislé ose, k polovině výšky tentokrát přičteme polovinu výšky textu. Přičítáme proto, že počátek má text dole, nikoliv nahoře.

Potom už stačí zavolat metodu pro zakreslení textu na zvolené souřadnice. Protože jsou vypočítané souřadnice x a y vztažené k oblasti umisteni, ne k položce, musí se ještě připočíst poloha její horní a levé strany.

Událost kliknutí na takto vykreslené tlačítko už jen obsahuje kontrolu, jestli je kliknutí ve vymezené oblasti. Pokud ano, zavolá událost posluchače clickListener.onClick().

```
private NakreslitTlacitkoClickListener clickListener;

public boolean onClick(float x, float y) {
    if (clickRegion != null && clickRegion.contains(x, y)) {
        clickListener.onClick(pozice);
        return true;
    }
    return false;
}
```

NakreslitTlacitkoClickListener je ve třídě PosouvaniHelper pouhý interface. Jeho obsah, tedy to co se má po kliknutí na tlačítko stát, dodáme až při volání z MainActivity. Je to tak proto, že tlačítko kreslíme sice univerzální, ale pokaždé po něm budeme požadovat jinou funkčnost. Proto zde můžeme skončit zmíněným voláním onClick() proměnné clickListener, která bude obsahovat definici požadované operace typu NakreslitTlacitkoClickListener.

Teď tedy máme definovanou třídu NakreslitTlacitko, kterou požijeme jako typ a můžeme si na začátku PosouvaniHelper vytvořit lokální proměnné pro obě tlačítka.

```
NakreslitTlacitko tlacitko1;
NakreslitTlacitko tlacitko2;
```

A rovnou založíme i konstruktor, který bude obsahovat volání nadřazené třídy, tedy SimpleCallback. Té předá v prvním parametru informaci, že položku nechceme přesouvat, ve druhém že s ní chceme posouvat a to tak, že směrem zprava doleva. Také si vytvoříme instanci GestureDetector, třídy, která zachytává prostřednictvím posluchače gesta, neboli akce uživatele prováděné na displeji. Instanci uložíme rovněž do lokání proměnné, kterou pro tento účel deklarujeme.

```
private GestureDetector gestureDetector;

public PosouvaniHelper(Context context) {
    super(0, ItemTouchHelper.LEFT);
    gestureDetector = new GestureDetector(context, gestureListener);
}
```

Aby zachytávání gest fungovalo, musíme mít připravený posluchač gestureListener, který se bude o zpracování gest starat.

```
private GestureDetector.SimpleOnGestureListener gestureListener =
                              new GestureDetector.SimpleOnGestureListener() {
    @Override
    public boolean onSingleTapConfirmed(MotionEvent e) {
        tlacitko1.onClick(e.getX(), e.getY());
        tlacitko2.onClick(e.getX(), e.getY());
        return true;
    }
};
```

Posluchač gestureListener obsahuje událost onSingleTapConfirmed(), která je zavolaná po kliknutí uživatele na displej. Jakmile se tak stane, zavolá pro každé tlačítko událost onClick(), v níž se zkontroluje, jestli oblast do které uživatel klikl, neodpovídá ploše samotného tlačítka.

Na začátku jsme třídu PosouvaniHelper rozšířili o rozhraní ItemTouchHelper.SimpleCallback. Teď je potřeba přepsat metody tohoto interface. První, onMove() je jednoduchá. Položky nebudeme přesouvat, takže pouze vrátí false a záležitost je vyřízena.

```
@Override
public boolean onMove(RecyclerView recyclerView,
      RecyclerView.ViewHolder viewHolder, RecyclerView.ViewHolder target) {
    return false;
}
```

Následuje několik v podstatě konfiguračních metod:

```
@Override
public float getSwipeThreshold(RecyclerView.ViewHolder viewHolder) {
    // hodnota, o kolik musi uzivatel polozku posunout, aby zustala posunuta
    return 200;
}

@Override
public float getSwipeEscapeVelocity(float defaultValue) {
```

```
    // rychlost animace
    return 0.1f * defaultValue;
}

@Override
public float getSwipeVelocityThreshold(float defaultValue) {
    // rychlost, kterou musi uzivatel polozku posunout, aby zustala posunuta
    return 5.0f * defaultValue;
}
```

První, getSwipeThreshold(), vrací hodnotu posunutí, která se bude brát jako mezní. Pokud uživatel posune položkou méně než o 200 bodů, tak se položka vrátí zpět. Když posune o víc, zůstane posunutá. Zbývající dvě události, getSwipeEscapeVelocity() a getSwipeVelocityThreshold() nastavují minimální a mezní rychlost posunutí, aby se dalo považovat za úplné posunutí i bez nutnosti jet celou cestu prstem po displeji. V obou případech stačí vrátit přednastavenou hodnotu.

Následují komplikovanější události. První z nich je onSwiped(), neboli položka byla posunuta. Zde je nutné poznamenat, že nestačí položky posouvat do strany, ale musíme je také vracet na původní místo. A to i v okamžiku, kdy odsuneme položku další. Kvůli tomu potřebujeme další lokální proměnné, ve kterých bude jednak položka se kterou právě manipulujeme a potom také položka, kterou jsme posouvali minule. Událost onSwiped() nejdřív naplní proměnné a následně zjistí, jestli není už z dřívějška některá položku posunuta. Pokud ano, vrátí ji na původní místo. K tomu potřebuje metodu vratitOdsunutouPolozku(), která upozorní adaptér na nutnost uvedení předchozí položku do normálního stavu. Až poté odstraní příznak z pomocné proměnné.

```
private int odsunutaPolozka = -1;
private int minuleOdsunutaPolozka = -1;

@Override
public void onSwiped(RecyclerView.ViewHolder viewHolder, int direction) {
    minuleOdsunutaPolozka = odsunutaPolozka;
    odsunutaPolozka = viewHolder.getAdapterPosition();
    if (odsunutaPolozka == minuleOdsunutaPolozka)
        minuleOdsunutaPolozka=-1;

    vratitOdsunutouPolozku();
}

private void vratitOdsunutouPolozku() {
    if (minuleOdsunutaPolozka > -1) {
        recyclerView.getAdapter().notifyItemChanged(minuleOdsunutaPolozka);
        minuleOdsunutaPolozka = -1;
    }
}
```

Další z komplikovanějších událostí je onChildDraw(). Ta je opakovaně vyvolávaná během posunování položky a měla by zajistit její průběžné vykreslování tak, jak se posunuje po displeji. Přepsaná událost umožní, před vykreslení položky vložit vlastní kód pro vykreslení tlačítek.

```
@Override
public void onChildDraw(Canvas c, RecyclerView recyclerView,
        RecyclerView.ViewHolder viewHolder, float dX, float dY, int actionState,
        boolean isCurrentlyActive) {

    int odsouvanaPolozka = viewHolder.getAdapterPosition();
    if (odsouvanaPolozka < 0) {
```

```
            odsunutaPolozka = -1;
            return;
        }

        float posunX = dX;
        View itemView = viewHolder.itemView;

        if (actionState == ItemTouchHelper.ACTION_STATE_SWIPE && dX <= 0) {
            tlacitko1=nakreslitTlacitko1(viewHolder);
            tlacitko2=nakreslitTlacitko2(viewHolder);

            View itemView = viewHolder.itemView;
            posunX = 2 * BUTTON_WIDTH * dX / itemView.getWidth();
            kreslitTlacitka(c, itemView, tlacitko1, tlacitko2, odsouvanaPolozka,
                    posunX);
        }
        super.onChildDraw(c, recyclerView, viewHolder, posunX, dY, actionState,
                    isCurrentlyActive);
    }
```

Událost má několik důležitých parametrů. Jednak jsou to odkazy na komponentu RecyclerView a její ViewHolder, potom velikost posunu v obou osách (nás teď bude zajímat pouze posun vodorovně, dX), stav položky, čili jestli se ní uživatel posunuje nebo dělá něco jiného a příznak, jestli položkou pohybuje opravdu uživatel, nebo jestli jde o posun automatický. Automaticky se položka pohybuje například když s ní uživatel přestane posunovat a ona se sama vrací do výchozí pozice.

V události nejdřív uděláme kontrolu, jestli skutečně uživatel položkou posunuje. Potom podle záporného dX ověříme jestli jde o posun doleva a do pomocných proměnných si uložíme odkaz na metody pro jejich vykreslení. Teď musíme spočítat umístění nových tlačítek v rámci položky. Aby tlačítka začínala tam kde položka končí, je potřeba vzít jejich společnou šířku, čili dvojnásobek šířky tlačítka a vynásobit ji poměrem posunu položky vůči její šířce.

```
posunX = 2 * BUTTON_WIDTH * dX / itemView.getWidth();
```

Takto získaná vodorovná souřadnice, pozor pouze v rámci tlačítka, ne celého displeje, říká, o kolik bodů od pravého kraje položky má být tlačítko umístěno. Kdybychom nechtěli aby se tlačítka s posunující se položkou jakoby přisunula, ale aby byla staticky pod ní a jenom se jakoby odkryla, vypustili bychom ze vzorce dX a nechali pouze:

```
posunX = - 2 * BUTTON_WIDTH;
```

Neboli záporný posun o dvojnásobek šířky tlačítka. Tomu bychom ale museli upravit i následné vykreslování. To je totiž dalším krokem který je potřeba udělat. Pro přehlednost si tedy bude lepší dát vykreslení tlačítek do samostatné metody kreslitTlacitka() a výpočet udělat až v ní . Nakonec ještě nesmíme zapomenout zavolat super.onChildDraw(), neboli tutéž metodu z nadřazené třídy. Pokud bychom to neudělali, překreslovala by se sice tlačítka, ale už ne posunovaná položka.

Metoda pro vykreslení tlačítek zavolá pro každé z nich metodu kreslitTlacitko() s příslušnými parametry, které řeknou jakým způsobem má k vykreslení dojít. Zde je opět prostor k tvůrčímu pojetí celé záležitosti. Tlačítka se můžou objevit třeba tak, že se při posouvání položky jakoby zvětšují z nuly na plnou šířku. Nebo prostě přijedou zprava, jakoby ukotvená k posunované položce. Pro druhou možnost stačí zjistit polohu pravé hrany položky. Ta bude zároveň místem, na kterém bude levá hrana prvního tlačítka. Tento bod se zjistí přičtením hodnoty posunu k šířce

položky. Posun doleva je záporný, takže se vlastně od velikosti položky jednoduše odečte. Od takto získaného místa necháme zobrazit první tlačítko a s patřičným odsazením i tlačítko druhé.

```
private void kreslitTlacitka(Canvas c, View itemView, NakreslitTlacitko tl1,
                  NakreslitTlacitko tl2, int pos, float dX) {
    float hranaPolozky = itemView.getRight()+dX;
    tl1.kreslitTlacitko(c, new RectF(hranaPolozky, itemView.getTop(),
                  hranaPolozky+120, itemView.getBottom()), pos);
    tl2.kreslitTlacitko(c, new RectF(hranaPolozky+120, itemView.getTop(),
                  hranaPolozky+240, itemView.getBottom()), pos);
}
```

Zbývá událost posluchače kliknutí na položku, onTouchListener(). Ta obsahuje předefinovanou událost onTouch(), ve které se zjišťuje jestli došlo k posunutí položky. V parametru má souřadnice posunu které vrací pomocí getRawX() a getRawY(). Událost zjistí, se kterou položkou se manipuluje, otestuje jestli jde o posouvání a pokud ano, tak jestli probíhá nad plochou položky. Pokud nad ní probíhá, vyvolá událost gestureDetector.onTouchEvent(). Jakmile prst přejede mimo plochu položky, tak naopak zavolá metodu vratitOdsunutouPolozku() pro uvedení položky do původního stavu.

```
private View.OnTouchListener onTouchListener = new View.OnTouchListener() {
    @Override
    public boolean onTouch(View view, MotionEvent e) {
        if (odsunutaPolozka < 0)
            return false;

        RecyclerView.ViewHolder swipedViewHolder =
            recyclerView.findViewHolderForAdapterPosition(odsunutaPolozka);

        View swipedItem = swipedViewHolder.itemView;

        Rect rect = new Rect();
        swipedItem.getGlobalVisibleRect(rect);

        if (e.getAction() == MotionEvent.ACTION_DOWN ||
            e.getAction() == MotionEvent.ACTION_UP ||
            e.getAction() == MotionEvent.ACTION_MOVE) {

            Point point = new Point((int) e.getRawX(), (int) e.getRawY());
            if (rect.top < point.y && rect.bottom > point.y)
                gestureDetector.onTouchEvent(e);
            else {
                minuleOdsunutaPolozka=odsunutaPolozka;
                odsunutaPolozka = -1;
                vratitOdsunutouPolozku();
            }
        }
        return false;
    }
};
```

Poslední metodou, kterou ještě potřebujeme, je pripojitRecyclerView(). Je to veřejná metoda, která vytvoří instanci pomocné třídy, propojí ji s komponentou RecyclerView a nastaví posluchače událostí.

```
public void pripojitRecyclerView(RecyclerView recyclerView) {
    this.recyclerView = recyclerView;
    this.recyclerView.setOnTouchListener(onTouchListener);
    ItemTouchHelper itemTouchHelper = new ItemTouchHelper(this);
    itemTouchHelper.attachToRecyclerView(this.recyclerView);
}
```

O co byla třída PosouvaniHelper oproti předchozím příkladům složitější, o to méně práce bude s adaptérem. Třídu RecycleAdapter můžeme převzít ze třetího příkladu beze změny. Nepotřebujeme vůbec nic navíc.

Poslední nezbytnou záležitostí je registrace události a propojení s komponentou Recycler View v události onCreate() třídy MainActivity:

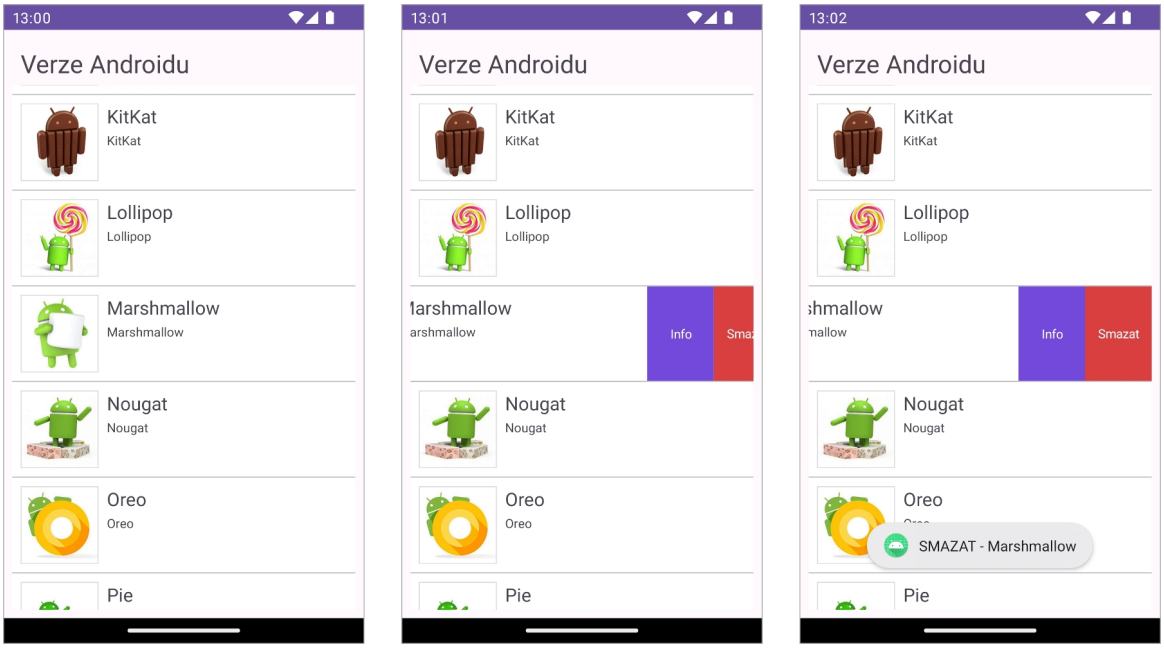
```
PosouvaniHelper posouvaniHelper = new PosouvaniHelper(this) {
    @Override
    public NakreslitTlacitko nakreslitTlacitko1(
                            RecyclerView.ViewHolder viewHolder) {
        return new PosouvaniHelper.NakreslitTlacitko(
            "Info",
            Color.parseColor("#744bdd"),
            new PosouvaniHelper.NakreslitTlacitkoClickListener() {
                @Override
                public void onClick(final int pos) {
                    String[] data = lstVerzeAndroidu.get(pos).split(":");
                    Toast.makeText(getApplicationContext(), "INFO -
                            ".concat(data[1]), Toast.LENGTH_LONG).show();
}
            }
        );
    }
    @Override
    public NakreslitTlacitko nakreslitTlacitko2(
                            RecyclerView.ViewHolder viewHolder) {
        return new PosouvaniHelper.NakreslitTlacitko(
            "Smazat",
            Color.parseColor("#db4141"),
            new PosouvaniHelper.NakreslitTlacitkoClickListener() {
                @Override
                public void onClick(int pos) {
                    String[] data = lstVerzeAndroidu.get(pos).split(":");
                    Toast.makeText(getApplicationContext(), "SMAZAT -
                            ".concat(data[1]), Toast.LENGTH_LONG).show();
}
            }
        );
    }
};
posouvaniHelper.pripojitRecyclerView(recyclerView);
```

Vytvoříme instanci pomocné třídy PosouvaniHelper, v parametru ji předáme kontext a jelikož jde o abstraktní třídu, definujeme metody nakreslitTlacitko1() a nakreslitTlacitko2(). V metodách vytvoříme instanci třídy NakreslitTlacitko, která bude v parametrech obsahovat jeho text, barvu a definici události onClick(), tedy výkonný kód, který se spustí po kliknutí na tlačítko. Kód události onClick() v tomto případě pouze najde zvolenou položku seznamu, rozparsuje data která se k ní vážou a zobrazí pomocí Toast uživatelskou informaci.

Toto uděláme dvakrát, zvlášť pro každé tlačítko. V tomto příkladu jsou konfigurace obou tlačítek velmi podobné, v praktickém užití by se ale lišily dle zvoleného účelu. Parametry jsou pro jednoduchost jenom tři, ale mohli bychom k nim přidat i další, například ID obrázku z resources, který by se použil pro vykreslení tlačítka, kdyby mělo být vyvedené graficky.

Jako poslední krok zavoláme metodu pripojitRecyclerView(), kterou jsme si připravili v pomocné třídě PosouvaniHelper a která se postará o nastavení posluchače a propojení pomocné třídy s komponentou RecyclerView.

Po spuštění na koncovém zařízení nebo v emulátoru by mělo být možné měnit posouvat položkami seznamu a klikat na tlačítka, jak je znázorněno na obrázku 14.17.



Obrázek 14.17: Posouvání položky seznamu s tlačítky, realizovaného kontejnerem typu Recycler View

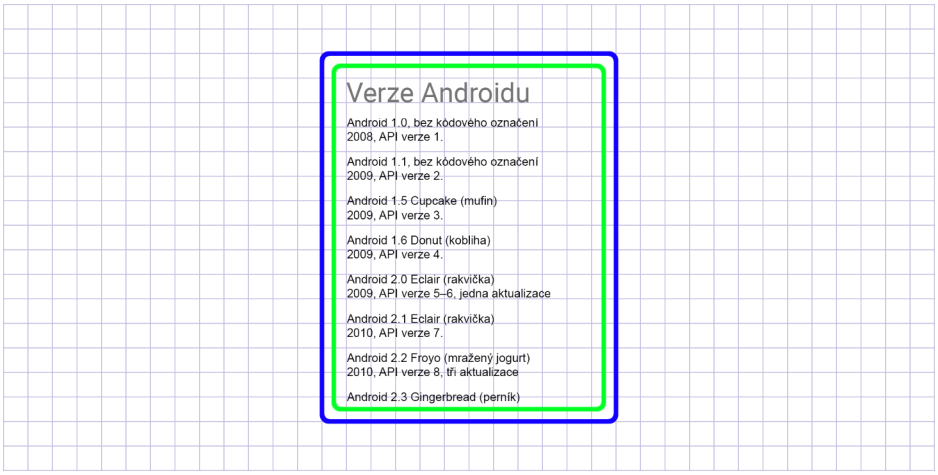
14.3 Scroll View

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt ScrollView

Pro případy, kdy se komponenty uživatelského rozhraní nevejdou na displej a je s nimi potřeba posunovat, jsou k dispozici kontejnery Scroll View (pro posun nahoru a dolů) a Horizontal Scroll View (posun do strany). Krom toho lze vložit Scroll View a Horizontal Scroll View do sebe a dosáhnout tak možnosti posouvání v obou osách. Nejčastěji se pravděpodobně setkáme s prostým vertikálním Scroll View, zejména všude tam, kde je potřeba pracovat s příliš dlouhým textem nebo formuláři. Je to ale jenom jeho nejjednodušší využití. Ačkoliv jde o kontejner poměrně jednoduchý, obsahovat může layout a ten už umožní posunovat prakticky jakkoliv komplikovaným formulářem. To se hodí pokaždé, když není jisté, jestli se jeho obsah vejde na displej.

Platí zde jedno základní pravidlo – Scroll View může obsahovat pouze jedinou komponentu. Pokud je tedy potřeba posunovat prostým textem, dá se do kontejneru vložit rovnou element <TextView ...>. Jakmile jsou ale potřeba byť jen dvě textová pole, už se musí do ScrollView vložit layout a až do něj lze umístit potřebné prvky.

Jako příklad si uděláme jednoduchý formulář se dvěma textovými poli, z nichž první bude zobrazovat nadpis a druhé bude obsahovat dlouhý text, který se nevejde na displej. Posunovat se tedy budou oba současně. Jelikož jde o dva elementy, je nutné uvnitř Scroll View použít layout.



Obrázek 14.18: Návrh formuláře s kontejnerem Scroll View

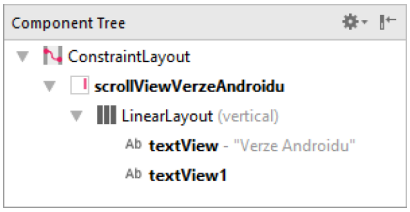
Práce s komponentou Scroll View je podstatně jednodušší, než použití již dříve popsaných kontejnerů.

Co je potřeba udělat?

- vytvořit formulář s kontejnerem Scroll View
- uložit do resources dlouhý text k zobrazení
- načíst text do textového pole

Otevřeme soubor activity_main.xml a do formuláře (v příkladu použijeme Constraint Layout) vložíme kontejner Scroll View. Rozměry nastavíme tak, aby byla vyplněna celá plocha displeje a aby se při natočení displeje velikost kontejneru přizpůsobila (viz kapitola 13.2 Constraint Layout).

Ve Scroll View budou potřeba dvě textová pole, jedno na nadpis a druhé na dlouhý text. Jak jsme si již řekli, Scroll View umožňuje vložit pouze jednu komponentu, umístíme do něj tedy formulář, například Linear Layout. Pokud použijeme designer a Scroll View vložíme do formuláře z palety nástrojů, vygeneruje se Linear Layout automaticky. Do formuláře umístíme zmíněná dvě pole. Důležité je, aby pole pro dlouhý text mělo nastavený parametr výšky wrap_content a mohlo se tedy dynamicky rozšířit i za hranice velikosti kontejneru Scroll View.



Obrázek 14.19: Prvky formuláře v okně Component Tree

Výsledný XML kód:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- ScrollView scrollViewVerzeAndroidu
         sirka a vyska max, bez okraju,
         navazany na parent kontejner -->
    <ScrollView
        android:id="@+id/scrollViewVerzeAndroidu"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_margin="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent">

        <!-- Linearni layout, se scrolovatelnym obsahem
             sirka max, vyska podle obsahu, vertikalni orientace -->
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical">

            <!-- textove pole s nadpisem, sirka max, vyska 35dp
                 velikost textu 20sp -->
            <TextView
                android:id="@+id/textView"
                android:layout_width="match_parent"
                android:layout_height="35dp"
                android:text="Verze Androidu"
                android:textSize="20sp" />

            <!-- textove pole s informacemi o verzich Androidu
                 sirka max, vyska podle obsahu, velikost textu 14sp -->
            <TextView
                android:id="@+id/textView1"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textSize="14sp" />
        </LinearLayout>
    </ScrollView>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Tím je formulář připravený. Data pro zobrazení vytvoříme tak, že do zdrojů res/values/strings.xml vložíme dlouhý řetězec s názvem verzeAndroidu:

```
<string name="verzeAndroidu">
    Android Alpha
    \nPrvní interní verze Androidu, rok 2007.
    \n
    \nAndroid Beta
    \n2007, první veřejná verze Androidu, sedm aktualizací.
    \n
```

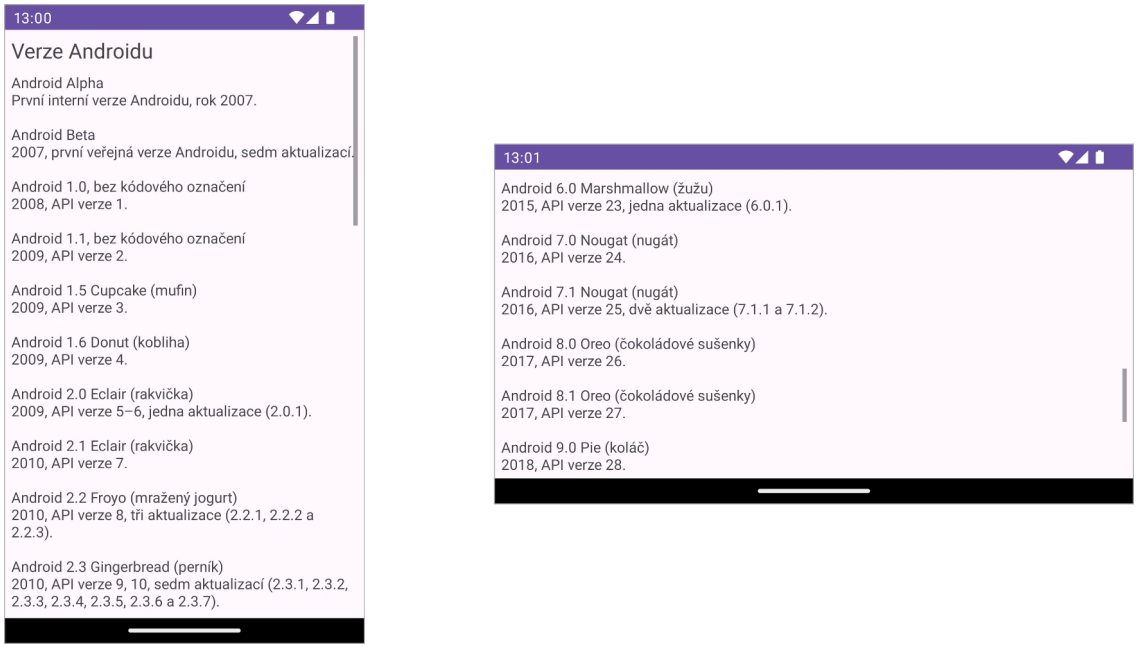
```
    \nAndroid 1.0, bez kódového označení
    \n2008, API verze 1.
    \n

    ...

    \nAndroid 8.1 Oreo (čokoládové sušenky)
    \n2017, API verze 27.
    \n

    ...

</string>
```

Teď už zbývá jenom nastavit dlouhý řetězec verzeAndroidu do textového pole. Pro tyto účely upravíme třídu MainActivity. V události onCreate() zavoláme vlastní metodu naplnitScrollView(), která řetězec načte a vyplní jím textové pole.

Metoda naplnitScrollView():

```
// nastavit text do textoveho pole
private void naplnitScrollView()
{
    // ziskat komponentu textView
    TextView textField = (TextView)findViewById(R.id.textView1);
    // do textView vlozit text definovany v resourcech string
    textField.setText(R.string.verzeAndroidu);
}
```

A to je vše. V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 14.20.



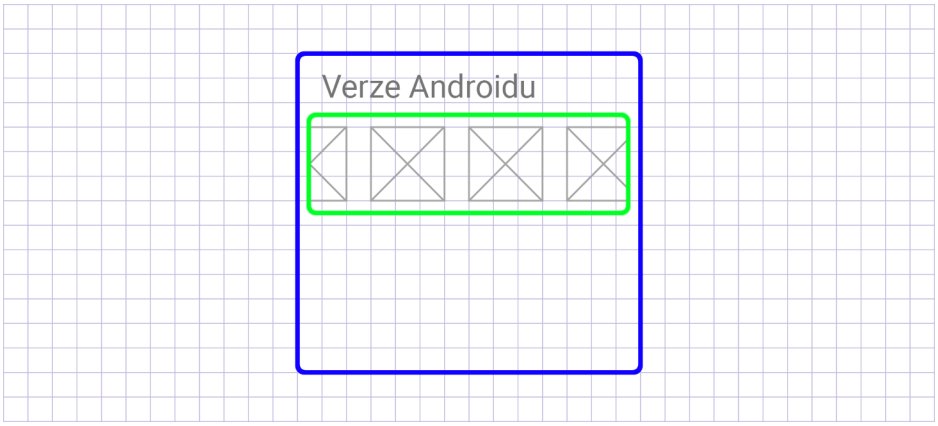
Obrázek 14.20: Aplikace s kontejnerem typu Scroll View, spuštěná na emulátoru

14.4 Horizontal Scroll View

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt HorizontalScrollView

Horizontal Scroll View je kontejner ekvivalentní ke Scroll View. Liší se pouze tím, že svůj obsah umožňuje posouvat do strany. Ostatní základní vlastnosti zůstávají beze změny.

Jako příklad si uděláme jednoduchý formulář s obrázky verzí Androidu, vloženými do horizontálně orientovaného lineárního layoutu. Tím získáme obsah, který bude širší než displej zařízení. Lineární layout bude vložený do Horizontal Scroll View, aby bylo možné široký obsah zobrazit a uživatel jím měl možnost posunovat.



Obrázek 14.21: Návrh formuláře s kontejnerem Horizontal List View

Co je potřeba udělat?

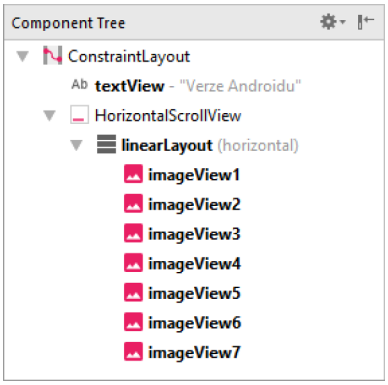
- uložit do resources vhodné obrázky

- vytvořit formulář s kontejnerem Horizontal Scroll View

Do složky res/drawable/ na disku počítače, vložíme loga jednotlivých verzí Androidu, nebo jiné vhodné obrázky. Měly by mít velikost a rozlišení odpovídající odhadnutému rozlišení displeje koncového zařízení. Příliš malé obrázky by působily kostrbatým dojmem a naopak příliš velké by zbytečně zvětšovaly aplikaci a zpomalovaly její běh.

Otevřeme soubor activity_main.xml a do formuláře vložíme kontejner Horizontal Scroll View. Rozměry nastavíme tak, aby byla vyplněna celá plocha displeje a aby se při natočení displeje velikost kontejneru přizpůsobila (viz kapitola 13.2 Constraint Layout).

Jelikož budeme pracovat s více obrázky, je potřeba do Horizontal Scroll View vložit formulář, například Linear Layout. Pokud použijeme designer a Horizontal Scroll View vložíme do formuláře z palety nástrojů, vygeneruje se Linear Layout automaticky. Do formuláře umístíme komponenty ImageView s obrázky uloženými v resources.



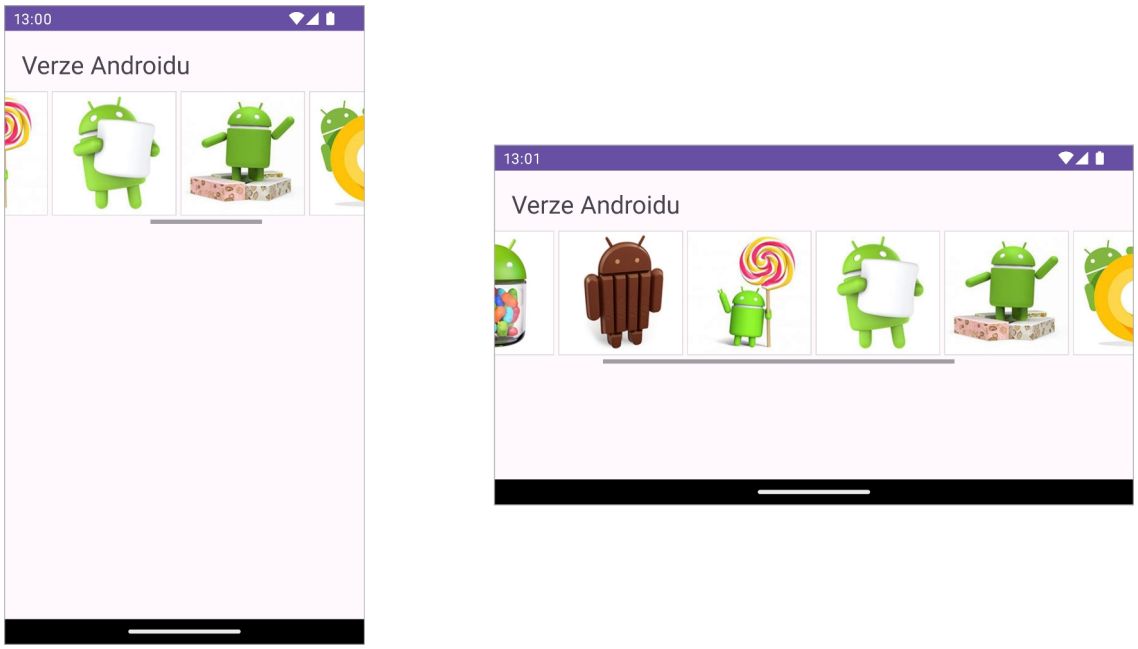
Obrázek 14.22: Prvky formuláře v okně Component Tree

Výsledný XML kód:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- textove pole s nadpisem, sirka max, vyska 40dp
         text vycentrovany, velikost textu 24sp, okraje 16dp
         prichycene zleva a ze shora k nadrazenemu kontejneru -->
    <TextView
        android:id="@+id/textView"
        android:text="Verze Androidu"
        android:layout_width="0dp"
        android:layout_height="40dp"
        android:textSize="24sp"
        android:layout_marginTop="16dp"
        android:layout_marginLeft="16dp"
        android:layout_marginStart="16dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintLeft_toLeftOf="parent" />

    <!-- HorizontalScrollView horizontalScrollViewVerzeAndroidu
         sirka max, vyska 130dp,
         navazany na parent kontejner a textView -->
    <HorizontalScrollView
        android:layout_width="0dp"
        android:layout_height="130dp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView">

        <!-- Linearni layout, se scrolovatelnym obsahem
             vyska max, sirka podle obsahu, horizontalni orientace -->
        <LinearLayout
            android:id="@+id/linearLayout"
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:orientation="horizontal" >

            <!-- jednotlive imageView s obrazky verzi Androidu
```

```
            vyska, sirka 120dp, okraj 2dp,
            oramovani barvou #dddddd o tloustce 1 dp -->
        <ImageView
            android:id="@+id/imageView1"
            android:layout_width="120dp"
            android:layout_height="120dp"
            android:layout_margin="2dp"
            android:background="#dddddd"
            android:padding="1dp"
            app:srcCompat="@drawable/icecreamsandwich" />

        <ImageView
            android:id="@+id/imageView2"
            android:layout_width="120dp"
            android:layout_height="120dp"
            android:layout_margin="2dp"
            android:background="#dddddd"
            android:padding="1dp"
            app:srcCompat="@drawable/jellybean" />

        .....

    </HorizontalScrollView>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Jelikož příklad neobsahuje žádnou další funkčnost ani logiku, jsme hotovi. Pro praktické využití bychom mohli napevno umístěné komponenty Image View nahradit jednoduchým formulářem a ten plnit programovým kódem. U obrázku by tak mohl být navíc popis, mohl by reagovat na kliknutí a podobně. To už ale nebylo cílem tohoto jednoduchého příkladu. V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 14.23.



Obrázek 14.23: Aplikace s kontejnerem typu Horizontal Scroll View, spuštěná na emulátoru

14.5 Vnořený Scroll View (Nested Scroll View)

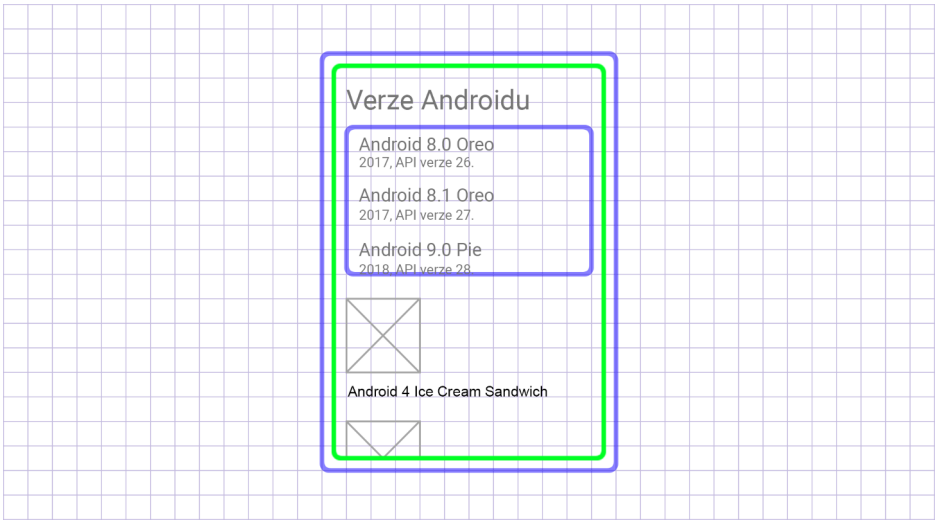
Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt NestedScrollView

Vertikální i horizontální Scroll View fungují vcelku jasně a spolehlivě. O poznání komplikovanější situace nastává, když je potřeba vložit vertikální Scroll View dovnitř jiného, také vertikálního. Vytvořit takové vnoření sice jde, ale Android potom dost komplikovaně a nespolehlivě zkouší identifikovat, v případě vertikálního tažení prstem po formuláři, jestli chce uživatel posunovat hlavní nebo vnořený formulář. Výsledek nevypadá hezky a způsobuje zmatek ve volaných událostech.

Android pro tyto účely zavedl komponentu NestedScrollView, která je určená primárně k tomu, aby byla uvnitř jiného ScrollView. A tomu jsou uzpůsobeny i její vlastnosti. Využití NestedScrollView je širší než by se mohlo zdát. Kromě toho, že může být vnořené do ScrollView, může být vnořené i do jiné NestedScrollView. Všechno přitom bude fungovat tak jak má. A nejen to, NestedScrollView může být použito místo ScrollView úplně samostatně, bez vnořování a nahradit tak ScrollView.

Uděláme si příklad formuláře, který bude moct uživatel posunovat nahoru a dolů, přičemž jeho součástí bude menší oblast, která bude také samostatně posouvatelná nahoru a dolů. Jako hlavní posouvací komponentu zvolíme ScrollView, jako vnořenou potom NestedScrollView. Ale samozřejmě nic nebrání tomu, aby případně byly obě komponenty NestedScrollView.

Ve vnější posouvatelné oblasti budou loga Androidu, ve vnitřní textová informace o verzích Androidu a jejich API. Loga nastavíme napevno, přímo do formuláře, stejně jako v předchozím příkladu a textové informace do vnořené posouvatelné oblasti načteme z resources, obdobně jako jsme to udělali v příkladu z kapitoly 14.3 Scroll View.



Obrázek 14.24: Návrh formuláře s kombinací kontejnerů Scroll View a Nested Scroll View

Co je potřeba udělat?

- uložit do resources vhodné obrázky
- vytvořit formulář s kontejnerem Scroll View a Nested Scroll View
- do formuláře vložit obrázky tak, aby přesahovaly vertikální rozměr displeje
- uložit do resources dlouhý text k zobrazení

- načíst text do textového pole

Do složky res/drawable/ na disku počítače, vložíme loga jednotlivých verzí Androidu, nebo jiné vhodné obrázky přiměřené velikosti. Tento krok se dá vyřešit prostým zkopírováním obrázků z předchozího příkladu.

V souboru activity_main.xml vytvoříme formulář, jehož součástí bude ScrollView, roztažený přes celou plochu displeje, v něm bude nadpis, pod ním vnořená NestedScrollView komponenta a následně loga Androidu složená z obrázků v resources. Celé to bude vypadat nějak takto:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- ScrollView scrollViewVerzeAndroidu
         sirka a vyska max, okraje 6dp,
         navazany na parent kontejner -->
    <ScrollView
        android:id="@+id/scrollViewVerzeAndroidu"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_margin="6dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent">

        <!-- Linearni layout, se scrolovatelnym obsahem
             sirka max, vyska podle obsahu, vertikalni orientace -->
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical">

            <!-- textove pole Nadpis, sirka max, vyska 35dp
                 velikost textu 20sp -->
            <TextView
                android:id="@+id/textView"
                android:layout_width="match_parent"
                android:layout_height="35dp"
                android:text="Verze Androidu"
                android:textSize="20sp" />

            <!-- NestedScrollView - vnorena scrolovatelna oblast
                 vyska 180dp, sirka max, sede podbarveni -->
            <androidx.core.widget.NestedScrollView
                android:layout_width="match_parent"
                android:layout_height="180dp"
                android:background="#EFEFEF"
                android:fillViewport="true">

                <!-- LinearLayout s obsahem scrolovatelné oblasti -->
                <LinearLayout
```

```
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:orientation="vertical">

                    <!-- TextView, které bude naplněno dlouhym textem -->
                   <TextView
                        android:id="@+id/textView2"
                        android:layout_width="match_parent"
                        android:layout_height="wrap_content"
                        android:textSize="14sp" />

                </LinearLayout>
            </androidx.core.widget.NestedScrollView>

            <!-- ImageView s logem Androidu
                 sirka, vyska 120dp, okraj 2dp, odsazeni 1dp -->
            <ImageView
                android:layout_width="120dp"
                android:layout_height="120dp"
                android:layout_margin="2dp"
                android:background="#dddddd"
                android:padding="1dp"
                app:srcCompat="@drawable/icecreamsandwich" />

            <!-- textove pole s verzi Androidu
                 sirka max, vyska 50dp, okraj 5dp, velikost textu 30sp -->
            <TextView
                android:text="Android 4.0 Ice Cream Sandwich"
                android:layout_width="match_parent"
                android:layout_height="50dp"
                android:layout_margin="5dp"
                android:textSize="20sp"/>

...

        </LinearLayout>
    </ScrollView>
</androidx.constraintlayout.widget.ConstraintLayout>
```

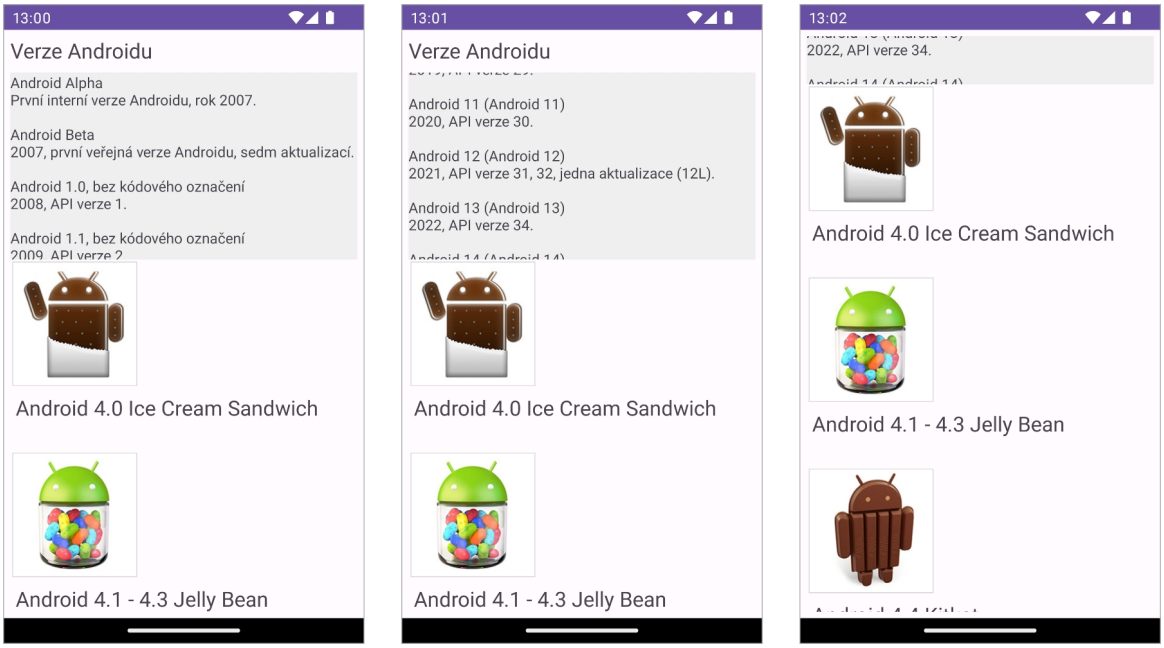
Tím je formulář připravený. Do zdrojů res/values/strings.xml vložíme dlouhý řetězec s názvem verzeAndroidu, který bude sloužit jako zdroj dat pro vnořený NestedScrollView. Dá se jednoduše zkopírovat z příkladu v kapitole 14.3 Scroll View.

Tento dlouhý řetězec je potřeba umístit do textového pole. Pro tyto účely upravíme třídu MainActivity. V události onCreate() zavoláme vlastní metodu naplnitNestedScrollView(), která řetězec načte a vyplní jím textové pole.

Metoda naplnitNestedScrollView():

```
// nastavit text do textoveho pole
private void naplnitNestedScrollView()
{
    // ziskat komponentu textView
    TextView textField = (TextView)findViewById(R.id.textView1);
    // do textView vlozit text definovany v resourcech string
    textField.setText(R.string.verzeAndroidu);
}
```

A to je vše. V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 14.25. S hlavním i vnořeným boxem by mělo jít volně posunovat nahoru a dolů.



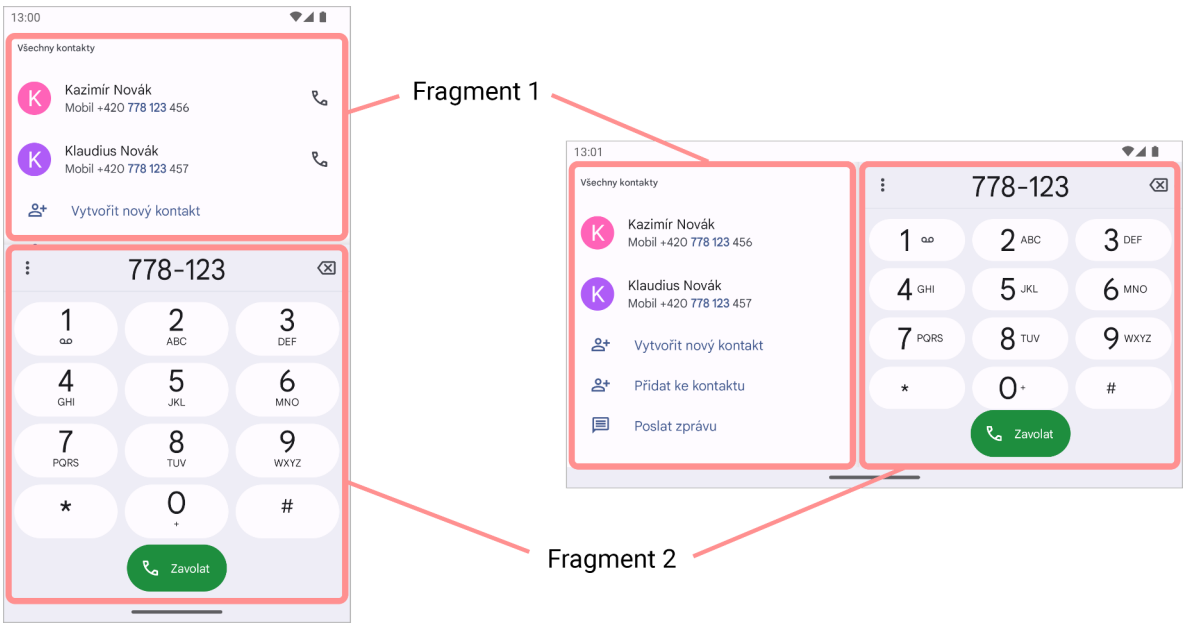
Obrázek 14.25: Aplikace s kontejnery typu Scroll View a Nested Scroll View, spuštěná na emulátoru

14.6 Fragmenty

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt Fragment

Fragmenty se v Androidu objevily poprvé ve verzi 3 (tj. API verze 11), která jako první podporovala zařízení s velkou úhlopříčkou, tedy zejména tablety. Účelem fragmentů je vytvořit malé bloky uživatelského rozhraní, jakési „podaktivity“, které je možné podle potřeby zobrazovat, nebo kombinovat v závislosti na tom, kolik místa je na displeji k dispozici. Díky tomu lze displej rozdělit na více samostatných částí, typicky – mobilní telefon nastojato může mít dva fragmenty nad sebou, pokud je naležato, fragmenty se přestěhují vedle sebe.

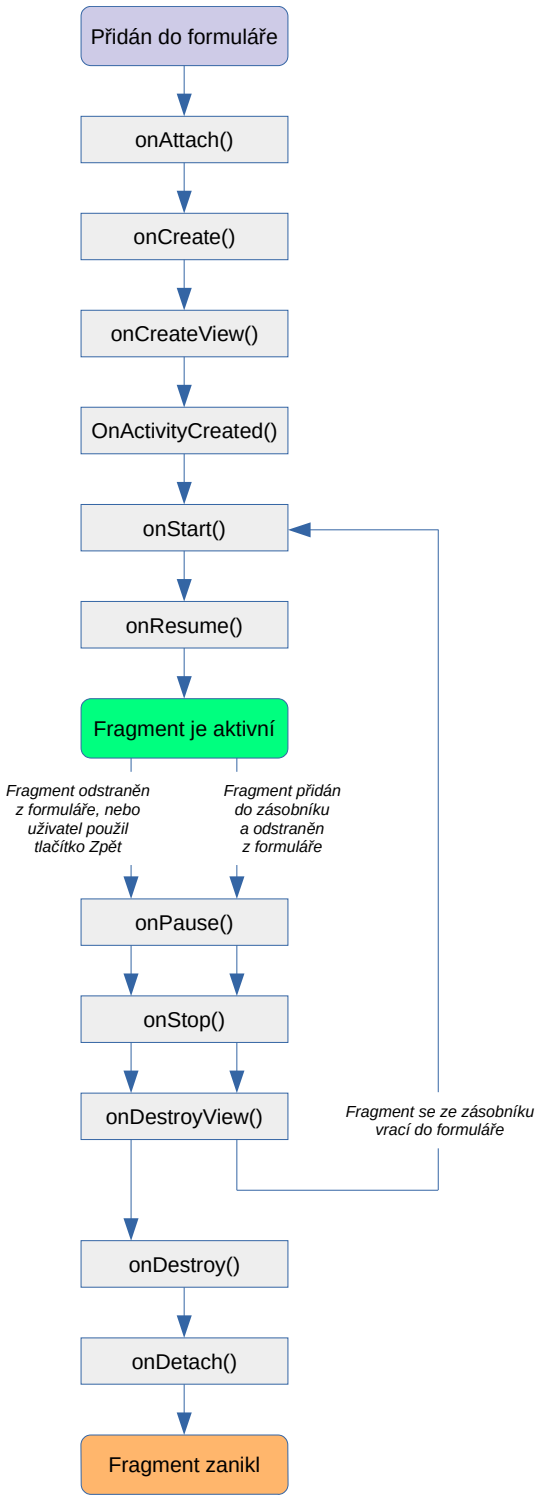
Má-li aktivita pouze jeden fragment, vizuálně se chová stejně jako by byla bez fragmentů, vidět jsou všechny prvky, vždy stejným způsobem. Pokud má fragmentů víc, dá se definovat jakým způsobem se na různých displejích, nebo při jakém natočení téhož displeje uspořádají. Pro představu se stačí podívat na displej při vytáčení čísla na mobilním telefonu, když je orientovaný nastojato a potom ho pootočit naležato.



Obrázek 14.26: Chování formuláře s fragmenty, při otočení zařízení o 90 stupňů

V tomto případě jsou oba fragmenty stále zobrazené. Je ale možné využít v aktivitě pro mobilní telefon pouze jeden z fragmentů a druhý zobrazit až po přepnutí na jinou aktivitu, zatímco při zobrazení na tabletu, kde je místa více, zobrazit oba fragmenty najednou.

Fragmenty jsou tedy komponenty uživatelského rozhraní, jakési panely, které se dají na formuláři skládat podle potřeby. Vždy musí být součástí aktivity, umísťují se do objektu ViewGroup a jejich životní cyklus je přímo závislý na životním cyklu aktivity. Pokud je například aktivita pozastavena, je pozastaven i fragment. Vlastní životní cyklus fragmentu se uplatňuje pouze když je aktivita ve stavu ‚Běžící aktivita‘.



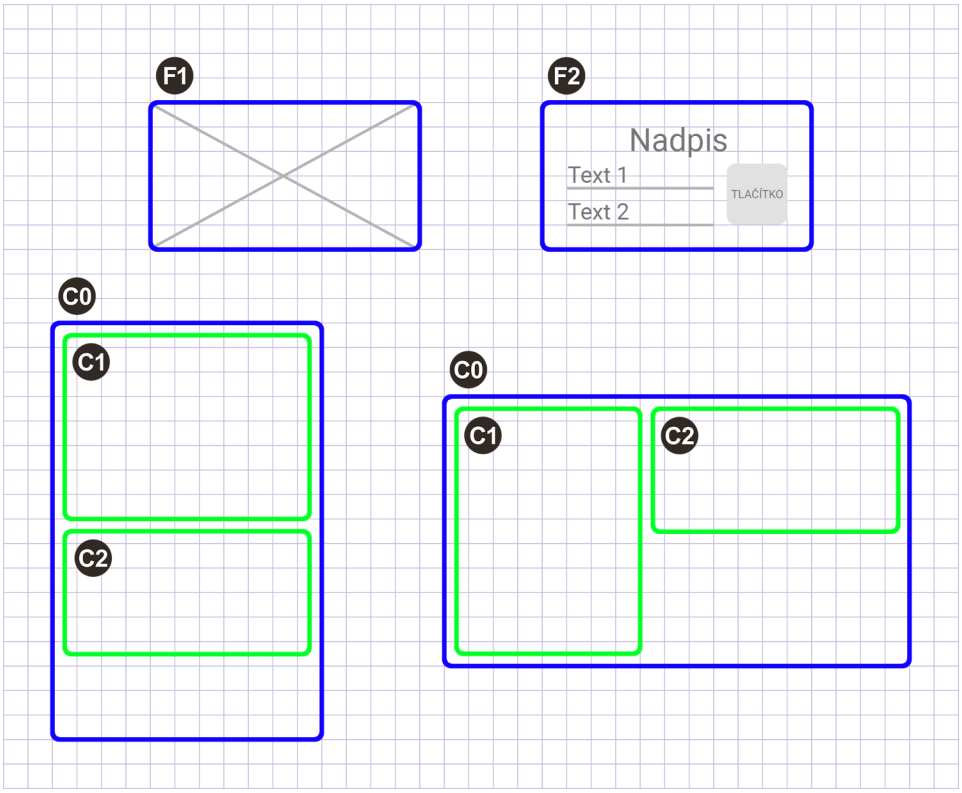
Obrázek 14.27: Životní cyklus fragmentu

Využití fragmentů je nebývale široké. Kromě skládání formulářů pro různé rozměry displeje se dají využít pro tzv. reuse - vícenásobné použití bloku prvků. Jakmile jsou v aplikaci na více místech stejně uspořádané objekty, lze je relativně snadno uložit do fragmentu a ten potom vložit do libovolného počtu aktivit. Fragmenty jsou také užívané dalšími komponentami, například Vew Pager nebo Tab Layout. Ty je využívají pro zobrazení stránek, které přepínají.

Fragmenty je možné v aktivitě vyměňovat za běhu programu. Je ale potřeba dát pozor na to, aby byly umístěny ve vhodném layoutu, který modifikace fragmentu za běhu umožňuje. Pro jednodušší formuláře se doporučuje Frame Layout nebo Linear Layout. Pro složitější lze použít Constraint Layout, ale je dobré mít na paměti, že ten má největší systémové nároky.

Jak fragment vytvořit? Nejsnáze kliknout pravým tlačítkem myši v okně projektu a z kontextového menu vybrat položku New → Fragment → Fragment (Blank). Průvodce se zeptá na název fragmentu a následně vytvoří ve složce java stejnojmennou třídu a ve složce res/layout XML soubor s prázdným formulářem. Stačí XML soubor otevřít a dál s ním pracovat jako s aktivitou.

Příklad aplikace se dvěma fragmenty, které se sestaví do formulářů v závislosti na otočení displeje:



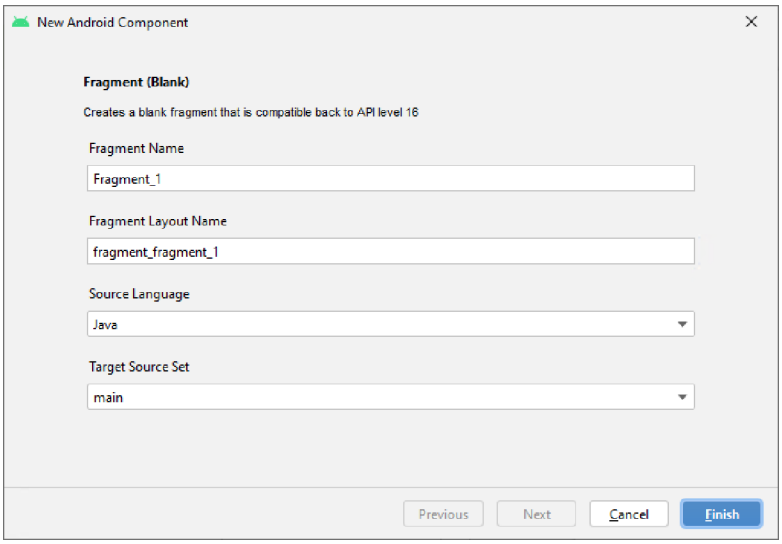
Obrázek 14.28: Návrh formuláře s fragmenty. Fragment F1 bude programově vložen do formulářů C1, fragment F2 do C2.

Co všechno budeme potřebovat udělat?

- dát do složky /res/drawable vhodný obrázek pro fragment1
- založit fragment F1
- založit fragment F2
- připravit formuláře, zvlášť pro svislou a vodorovnou orientaci displeje
- vložit kontejnery C1 a C2 do kontejneru C0, ve formulářích activity_main.xml
- ve třídě MainActivity zajistit vložení fragmentů do kontejnerů

Budeme potřebovat vhodný obrázek, který umístíme na disk do složky /res/drawable. Hned poté se automaticky objeví v okně projektu.

Fragment1 se založí v okně projektu prostřednictvím kontextového menu New → Fragment



Obrázek 14.29: Průvodce zakládání nového fragmentu

→ Fragment (Blank).

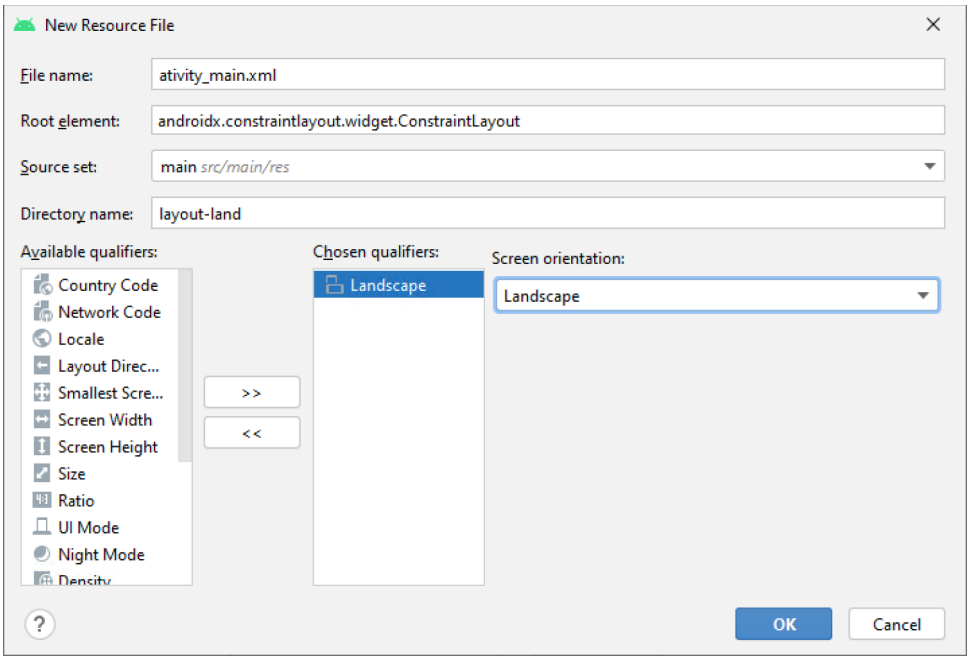
Průvodce vygeneruje ve složce Java třídu Fragment_1 a ve složce res formulář fragment_fragment_1.xml. V XML formuláře vytvoří kontejner Frame Layout odkazující se na Java třídu Fragment1. Tím vznikne fragment F1. Ten bude obsahovat pouze obrázek, takže do kontejneru vložíme prvek ImageView s obrázkem, který jsme umístili do složky /res/drawable/:

```
<!-- frame layout F1, sirka max, vyska max,
     obsah Fragment1 -->
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".Fragment1">

        <!-- image view, sirka max, vyska max
         umisteny na stred a oriznuty -->
        <ImageView
            android:id="@+id/imageView"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:scaleType="centerCrop"
            android:src="@drawable/test_image" />

</FrameLayout>
```

Stejným způsobem se vygeneruje i fragment F2. Tento fragment bude obsahovat malý formulář, již známý z předchozích příkladů. Průvodcem vygenerovaný Frame Layout tedy v XML formuláře nahradíme XML kódem z příkladu Constraint Layoutu. Získáme tak kontejner Constraint Layout s nadpisem, dvěma textovými poli a tlačítkem.

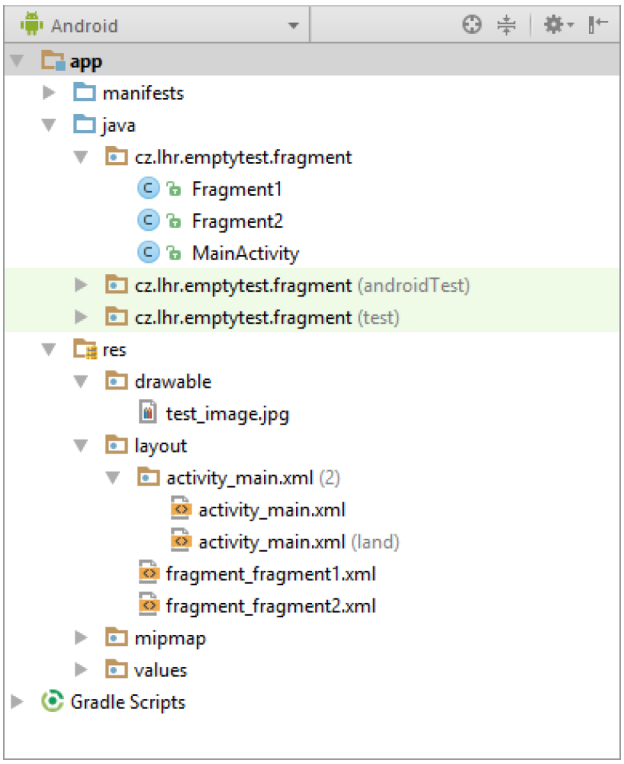
Nyní se zaměříme na design aktivity activity_main.xml. Aby se aplikace chovala jinak při otočení displeje, potřebujeme v ní mít dva různé formuláře – jeden navržený nastojato, druhý naležato. V okně projektu klikneme pravým tlačítkem myši do activity_main.xml a v kontextovém

menu zvolíme New→ Layout resource file. Spustí se průvodce novým zdrojovým souborem. Do pole File name dáme název souboru který rozšiřujeme, tj. „activity_main.xml“. V nabídce Available qualifiers (Dostupné parametry) zvolíme Orientation a tlačítkem “>>” ji přesuneme do nabídky Chosen qualifiers (Zvolené parametry). Objeví se možnosti parametru, nastavíme Landscape (displej naležato).



Obrázek 14.30: Průvodce založením nového zdrojového souboru

Tím vzniknou dva XML soubory pod aktivitou activity_main.xml. Struktura aplikace v okně projektu bude tedy vypadat takto:



Obrázek 14.31: Okno projektu, struktura aplikace s fragmenty

Teoreticky by takto šlo snadno vytvořit formuláře pro různé velikosti a orientace displejů, do nichž bychom potom vkládali připravené fragmenty. Mohli bychom dokonce zohlednit jemnost displeje, verzi systému, lokalizaci, operátora, přepnutí na denní či noční režim, a tak dále. A pro každý takový parametr, nebo kombinaci parametrů, vytvořit samostatný design formuláře. Nyní se ale spokojíme se dvěma základními – svislým a vodorovným displejem.

Do svislého formuláře activity_main.xml, s constraint layoutem C0, vložíme v editoru designu nebo XML pod sebe dva constraint layouty C1 a C2. Layouty budou pouze vytvořené, nebudou mít žádný obsah. Ten se do nich vloží později, programově. Výsledný XML kód bude vypadat přibližně takto:

```
<!-- constraint layout C0,
     sirka max, vyska max, horizontalni orientace,
     obsahuje kod MainActivity -->
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/mainLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- constraint layout C1,
         sirka max, vyska 200dp -->
    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/firstLayout"
        android:layout_width="368dp"
        android:layout_height="200dp"
        tools:layout_editor_absoluteY="0dp"
```

```
        tools:layout_editor_absoluteX="8dp">
        <!-- layout C1 neobsahuje zadne elementy
              doplnime sem programove fragment -->
    </androidx.constraintlayout.widget.ConstraintLayout>

    <!-- constraint layout C2,
         sirka max, vyska 200dp,
         ukotveni ze shora k C1, zezdola k C0 -->
    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/secondLayout"
        android:layout_width="368dp"
        android:layout_height="200dp"
        app:layout_constraintTop_toBottomOf="@+id/firstLayout"
        app:layout_constraintBottom_toBottomOf="parent"
        tools:layout_editor_absoluteX="8dp">
        <!-- layout C2 neobsahuje zadne elementy
              doplnime sem programove fragment -->
    </androidx.constraintlayout.widget.ConstraintLayout>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Totéž uděláme s vodorovným formulářem. Vložíme do něj constraint layouty C1 a C2, jen s tím rozdílem, že budou vhodně umístěny vedle sebe. Výsledný XML kód:

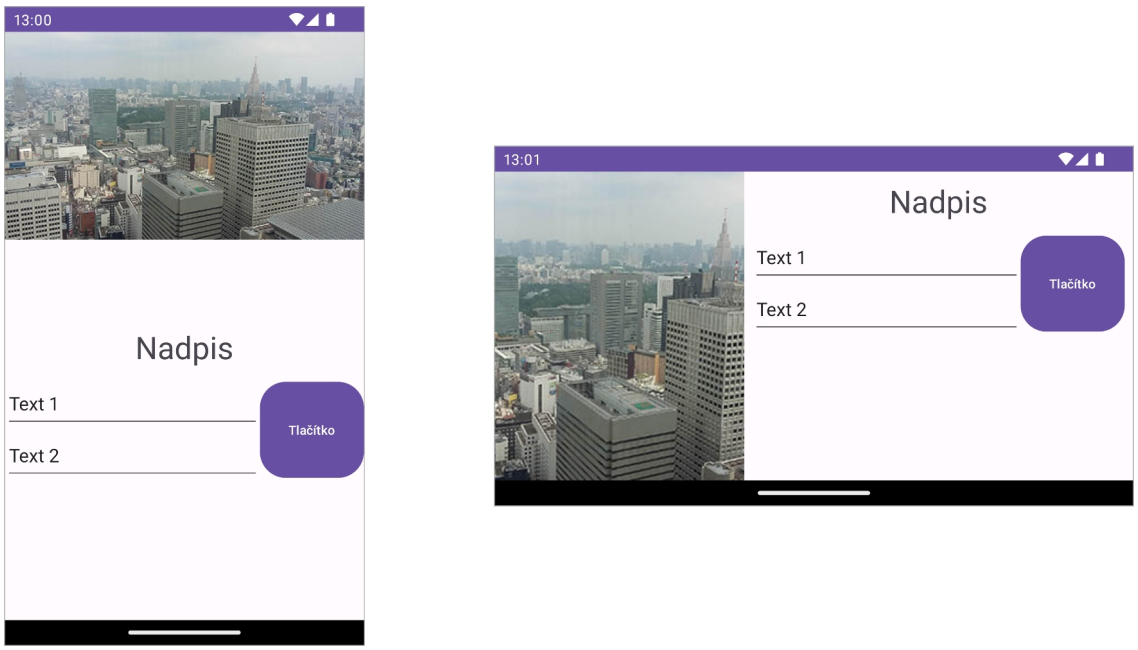
```
<!-- constraint layout C0,
     sirka max, vyska max, vertikalni orientace,
     obsahuje kod MainActivity -->
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/mainLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- constraint layout C1,
         sirka 240dp, vyska max,
         ukotveni zleva a ze shora k C0 -->
    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/firstLayout"
        android:layout_width="240dp"
        android:layout_height="match_parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toTopOf="parent">
        <!-- layout C1 neobsahuje zadne elementy
              doplnime sem programove fragment -->
    </androidx.constraintlayout.widget.ConstraintLayout>

    <!-- constraint layout C2,
         sirka max, vyska 240dp,
         ukotveni zleva k C1, zprava a ze shora k C0,
         okraje 8dp -->
    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/secondLayout"
        android:layout_width="0dp"
        android:layout_height="200dp"
        app:layout_constraintLeft_toRightOf="@+id/firstLayout"
        app:layout_constraintRight_toRightOf="parent"
```

```
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginTop="8dp"
        android:layout_marginRight="8dp"
        android:layout_marginLeft="8dp">
        <!-- layout C2 neobsahuje zadne elementy
              doplnime sem programove fragment -->
    </androidx.constraintlayout.widget.ConstraintLayout>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Tím jsou připravené formuláře i fragmenty a zbývá jen zajistit jejich propojení. To se udělá ve třídě MainActivity, kde je průvodcem vygenerovaná událost onCreate(). Ta se vyvolá, jak víme ze životního cyklu aktivity, ještě před vlastním zobrazením aktivity na displeji. Toho využijeme a událost rozšíříme o nastavení fragmentů:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // ziskat spravce pro praci s fragmenty
    FragmentManager manager = getSupportFragmentManager();

    // vytvorit instanci fragmentu Fragment1
    Fragment1 f1 = new Fragment1();
    // v transakci priradit fragment do prvniho constraint layoutu
    manager.beginTransaction()
                .replace(R.id.firstLayout, f1, f1.getTag())
                .commit();

    // vytvorit instanci fragmentu Fragment2
    Fragment2 f2 = new Fragment2();
    // v transakci priradit fragment do druheho constraint layoutu
    manager.beginTransaction()
                .replace(R.id.secondLayout, f2, f2.getTag())
                .commit();
}
```

V emulátoru nebo na koncovém zařízení by měl výsledek vypadat jako na obrázku 14.32.



Obrázek 14.32: Aplikace s fragmenty, spuštěná na emulátoru

14.7 View Pager

Pro přepínání formulářů se v grafickém rozhraní Androidu poměrně často používá posouvání prstem z jedné stránky na následující. Toto posouvání je ve své podstatě grafickou transformací jedné stránky ve druhou. Ve své nejjednodušší podobě je implementováno jako prosté posunutí z jedné na druhou stranu, nebo případně ze shora dolů či obráceně. Toto vše zajišťuje komponenta ViewPager.

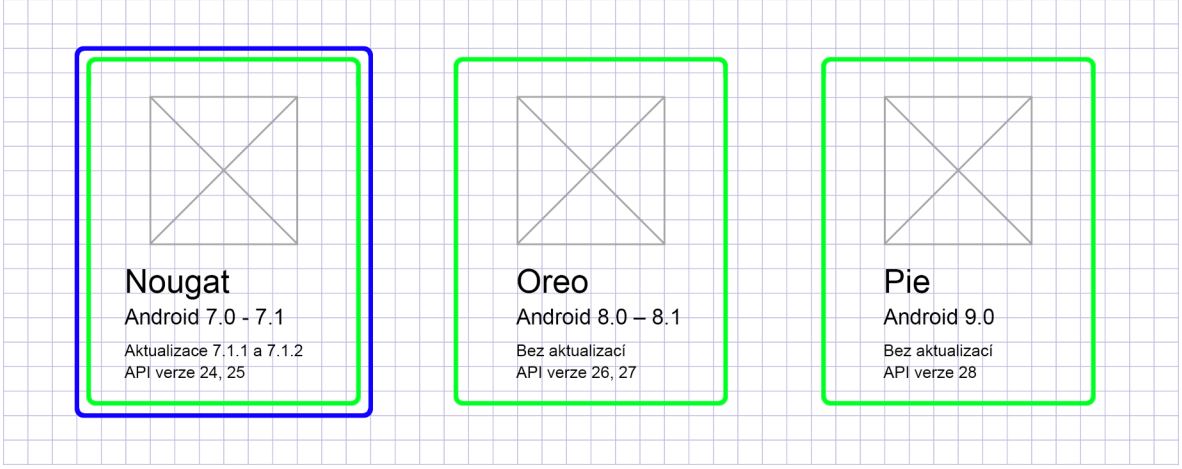
Přesněji řečeno, existují dvě verze této komponenty, jedna se jmenuje prostě ViewPager a druhá, modernější a s výrazně širšími možnostmi, ViewPager2. Android Studio v paletě komponent nabízí zatím starší verzi, ale v editoru ji lze vyměnit za novější. Toho využijeme a budeme se zabývat výhradně ViewPager2.

ViewPager nedokáže stránky zobrazovat sám. Ke správnému fungování potřebuje komponentu Fragment, která nese samotný obsah. ViewPager pouze instance jednotlivých fragmentů prostřednictvím adaptéru přepíná. Jelikož jde o instance, je možné použít jak samostatný fragment pro každou stránku, tak pouze jeden, od kterého vytvoříme více instancí. Komponentě ViewPager je to v podstatě jedno, co jí připravíme, to bude ochotně přepínat. Vzájemné propojení jednotlivých prvků tedy bude ViewPager2 ↔ adaptér ↔ fragmenty.

14.7.1 View Pager, 1. příklad

Na prvním příkladu si ukážeme možnosti komponenty se zcela minimalistickou potřebou programování. Pouze si dáme ViewPager na formulář a necháme ho, aby přepínal tři fragmenty které ponesou obsah nakreslený v editoru formulářů. Každý z fragmentů bude obsahovat informace o jedné verzi Androidu.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt ViewPager1



Obrázek 14.33: Návrh formuláře s kontejnerem View Pager (modrá barva) a třemi fragmenty (zelená barva)

Co je potřeba udělat?

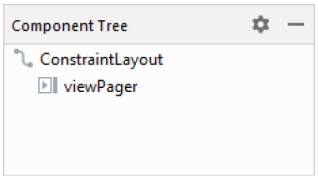
- uložit do resources vhodné obrázky
- vytvořit formulář s kontejnerem ViewPager
- připravit několik fragmentů, každý bude obsahovat informace o jedné verzi Androidu
- napsat jednoduchý adaptér, který se postará o propojení komponenty ViewPager a fragmentů

- ve třídě MainActivity.java v události onCreate() iniciovat adaptér

Do složky res/drawable/ na disku počítače, uložíme loga jednotlivých verzí Androidu, nebo jiné vhodné obrázky přiměřené velikosti.

Otevřeme soubor activity_main.xml a vložíme do něj komponentu ViewPager. Pokud ji vkládáme v editoru formuláře, přepneme se do textového editoru a v XML kódu změníme ViewPager na ViewPager2. Při vkládání komponenty v XML tato úprava odpadá, vložíme rovnou modernější verzi komponenty. Umístíme ji tak, aby vyplňovala celou plochu formuláře. Komponentě nastavíme id „viewPager“, budeme se na ni později potřebovat odkázat. Tím je aktivita připravena.

Okno Component Tree neobsahuje nic překvapivého. Pouze základní layout a v něm kontejner typu ViewPager2, nazvaný „viewPager“.



Obrázek 14.34: Prvky formuláře activity_main v okně Component Tree

XML kód by měl vypadat takto:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/viewPager"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Trochu víc práce bude s přípravou fragmentů, které ponesou samotný obsah. Fragment lze vytvořit kliknutím pravého tlačítka myši do okna projektu, z kontextového menu potom vybrat New→Fragment → Fragment (blank). Zadáme název, například „StrankaNougat“, pro verzi Androidu Nougat.

Vzniknou dva soubory, první StrankaNougat.java, který bude obsahovat inicializační rutiny. Ten nás nyní nebude zajímat, necháme ho beze změny tak, jak je. Druhým souborem je fragment_stranka_nougat.xml s formulářem typu FrameLayout. Ten sám o sobě není úplně vhodný k rozmísťování objektů, takže si do něj vložíme pouze ConstraintLayout a ten vyplníme obrázkem verze Androidu v ImageView, přidáme několik řádek informací o verzi Androidu a fragment je v základní podobě připravený.

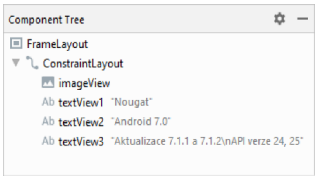
Dá se ještě vylepšit vhodným rámečkem, aby ohraničení fragmentu lépe vyniklo. Z hlediska systémových prostředků se k tomuto účelu skvěle hodí v XML zapsaný grafický prvek. Jeho užití je

popsané v kapitole 9.7 XML obrázky. Může mít podobu třeba obdélníku se zaoblenými rohy, což v XML vypadá následovně:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
                        android:shape="rectangle">
    <solid android:color="#04000000" />
    <stroke android:width="3dp"
                        android:color="@color/material_dynamic_neutral80" />
    <corners android:radius="20dp" />
</shape>
```

XML kód se uloží do složky drawable, s názvem třeba drawable/ramecek.xml. Odkaz na něj přijde do parametru background formuláře ConstraintLayout.

Okno Component Tree obsahuje pro každý z fragmentů základní FrameLayout s formulářem ConstraintLayout a v něm textové a obrazový prvek.



Obrázek 14.35: Prvky fragmentů v okně Component Tree

XML kód fragmentu bude vypadat přibližně takto:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".StrankaNougat">

    <!-- constraint layout, s obsahem prvni stranky
         sirka a vyska max, ale s okrajem 30dp,
         s rameckem definovanym v drawable/ramecek.xml -->
    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_margin="30dp"
        android:background="@drawable/ramecek">

        <!-- ImageView - obrazek verze androidu z resources
             sirka max, vyska podle obsahu, s okrajem 24dp
             obrazek vycenttrovany, prizpusobena velikost -->
        <ImageView
            android:id="@+id/imageView"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_margin="24dp"
            android:scaleType="fitCenter"
            android:src="@drawable/nougat"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <!-- textove pole s nazvem verze Androidu,
```

```
            sirka max, vyska podle obsahu, okraj 32dp,
            velikost textu 26sp -->
        <TextView
            android:id="@+id/textView1"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_margin="32dp"
            android:text="Nougat"
            android:textSize="26sp"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/imageView" />

        ...

    </androidx.constraintlayout.widget.ConstraintLayout>
</FrameLayout>
```

Stejným způsobem se vytvoří fragmenty i pro další verze Androidu. Lišit se budou obrázkem a použitými texty, struktura formuláře zůstane stejná.

Jakmile máme k dispozici potřebný počet fragmentů, musíme zařídit jejich přiřazování komponentě ViewPager. K tomu slouží adaptér, který vznikne rozšířením k tomu určené třídy FragmentStateAdapter. Založíme k tomuto účelu novou třídu PageAdapter, která bude stát mezi komponentou ViewPager a jednotlivými fragmenty:

```
public class PageAdapter extends FragmentStateAdapter {
    public PageAdapter(@NonNull FragmentActivity fragmentActivity) {
        super(fragmentActivity);
    }

    @Override
    public Fragment createFragment(int position) {
        switch (position){
            case 0: return new StrankaNougat();
            case 1: return new StrankaOreo();
            case 2: return new StrankaPie();
        }
        return null;
    }

    @Override
    public int getItemCount() {
        return 3;
    }
}
```

V nové třídě jsou dvě předefinované metody. První, createFragment(), slouží k samotnému přiřazování fragmentů. Princip je jednoduchý. V okamžiku, kdy ViewPager potřebuje zobrazit první stránku, zavolá adaptér a v něm vyvolá tuto událost. Předá jí číslo první stránky, tj. 0. Podle něj náš kód založí instanci fragmentu StrankaNougat a tu předá v návratové hodnotě. Tím se instance fragmentu dostane ke komponentě ViewPager a ten ji vykreslí na displej. Jakmile uživatel přejede prstem po displeji směrem k další stránce, požádá ViewPager adaptér o instanci následující stránky a proces se opakuje. Fragmenty samozřejmě nemusíme mít připravené pro všechny verze Androidu, můžeme je dynamicky generovat na základě dat. K tomu se ale dostaneme až v dalším příkladu, zde je pro jednoduchost použijeme pouze staticky.

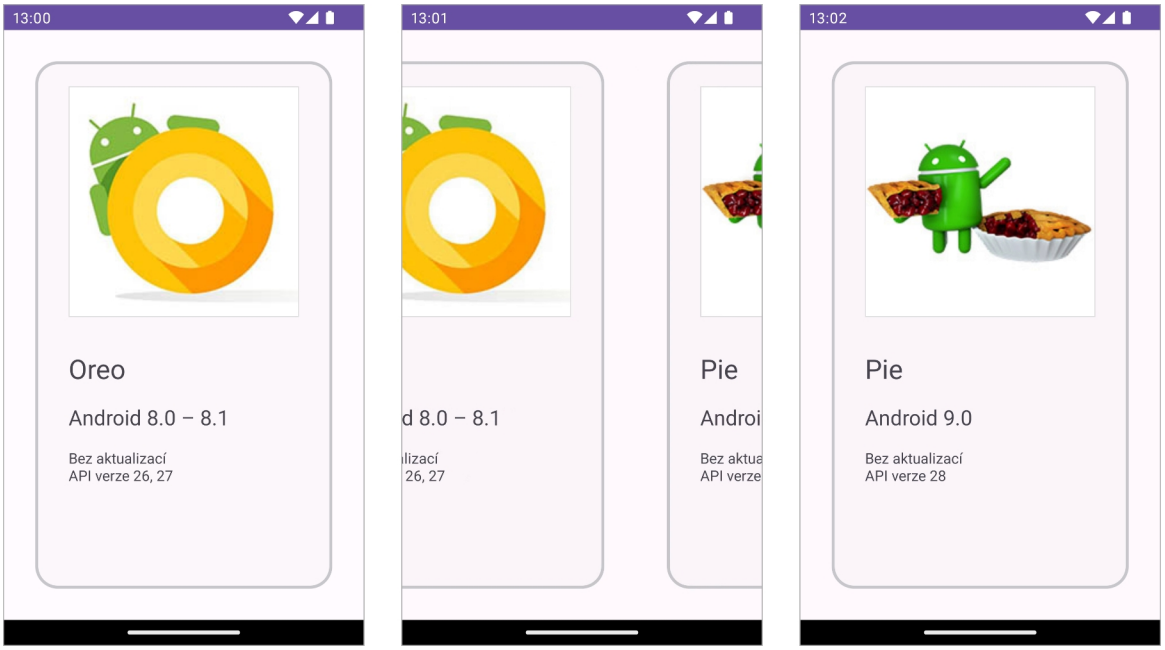
Druhou, podstatně jednodušší předefinovanou metodou je getItemCount(). Tu volá ViewPager tehdy, když potřebuje zjistit kolik stránek má vlastně zobrazovat. Stačí, když mu v ní vrátíme hodnotu odpovídající počtu připravených fragmentů. Pokud bychom vraceli číslo nižší, nezobrazil by stránky pro všechny fragmenty. Nižší číslo by naopak způsobilo chybu, protože při přechodu na neexistující stránku by událost createFragment() vrátila namísto instance fragmentu pouze hodnotu null.

Zbývá upravit třídu MainActivity. V ní, v události onCreate() propojíme adaptér s komponentou ViewPager:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // nova instance adapteru
    PageAdapter mAdapter = new PageAdapter(this);

    // prirazeni adapteru k ViewPager
    ViewPager2 viewPager = (ViewPager2) findViewById(R.id.viewPager);
    viewPager.setAdapter(mAdapter);
}
```

V události vytvoříme instanci adaptéru, najdeme si na formuláři ViewPager a adaptér jí jednoduše přiřadíme.

V emulátoru nebo na koncovém zařízení by měl výsledek vypadat jako na obrázku 14.36.



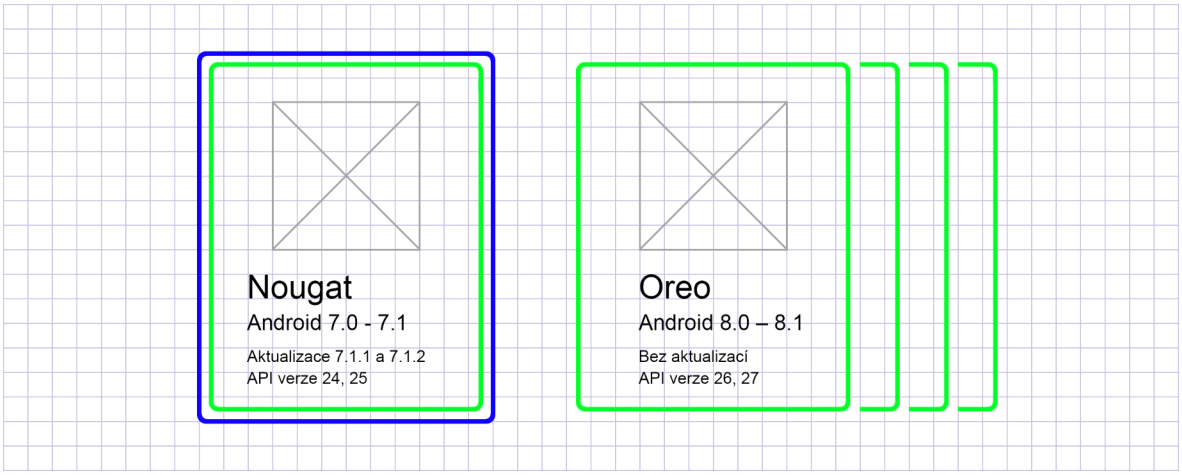
Obrázek 14.36: Aplikace s kontejnerem View Pager, spuštěná na emulátoru

14.7.2 View Pager, 2. příklad

Ve druhém příkladu nebude mít každá verzi Androidu samostatný fragment, ale uděláme jen jeden, do kterého se budou informace dynamicky doplňovat. ViewPager tedy

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt ViewPager2

bude zobrazovat stále ten stejný fragment, jenom pokaždé s jiným obsahem. Potřebná data se předají do fragmentu prostřednictvím adaptéru. Zároveň si vyzkoušíme využití grafické transformace při přepínání stránek.



Obrázek 14.37: Návrh formuláře s kontejnerem View Pager (modrá barva) a generovanými fragmenty (zelená barva)

Co je potřeba udělat?

- uložit do resources vhodné obrázky
- vytvořit formulář s kontejnerem ViewPager
- připravit fragment s formulářem, ve kterém se budou zobrazovat data o verzích Androidu
- napsat adaptér, který se postará o propojení komponenty ViewPager a fragmentů
- ve třídě MainActivity.java v události onCreate() iniciovat adaptér a odeslat data
- přidat třídu která se postará o grafickou transformaci stránky

Stejně jako v předchozím příkladu i teď uložíme do složky res/drawable/ loga jednotlivých verzí Androidu, nebo jiné vhodné obrázky přiměřené velikosti.

Neliší se ani příprava základního formuláře. V souboru activity_main.xml vložíme do Constraint Layoutu komponentu ViewPager a v XML ji změníme na ViewPager2. Umístíme ji tak, aby vyplňovala celou plochu formuláře. Komponentě nastavíme id „viewPager“, aby se na ni pod tímto id dalo odkazovat. Tím je aktivita připravena.

Změna nastane u fragmentu. Ten lze vytvořit kliknutím pravého tlačítka myši do okna projektu, z kontextového menu potom vybrat New→Fragment → Fragment (blank). Zadáme vhodný název, například „Stranka“. Fragment se sice nebude lišit vizuálně, takže můžeme klidně použít pro sestavení formuláře jeho XML z minulého příkladu. Tentokrát však využijeme také jeho java část, tedy soubor Stranka.java.

Ve třídě Stranka je podstatná zejména událost onCreateView(). V ní se uloží do pomocných proměnných parametry stránky, předávané z adaptéru. Adaptér bude jejich hodnoty posílat v textové formě. Ke vstupním datům se lze dostat pomocí metody getArguments(), kterou k tomuto účelu poskytuje třída Fragment pro vyzvednutí objektu Bundle a následné získání předaných hodnot různých datových typů (v tomto případě textového řetězce):

```
String strLogo = getArguments().getString("logo");
```

Objekt Bundle se předá fragmentu v adaptéru. Takto se potom načtou parametry logo, nazev, verze a data, z nichž první reprezentuje název resource obrázku a zbývající textové informace o verzi Androidu. Odešlou se pod těmito názvy z adaptéru, kde se objekt Bundle následně předá fragmentu, k tomu se dostaneme za chvíli.

V události onCreateView() bude ještě nutné najít komponentu ImageView a všechna potřebná textová pole formuláře. Odkazy na ně se také uloží do pomocných proměnných. Podle názvu obrázku se najde v resources vždy právě ten obrázek, který bude zobrazený ve formuláři a přiřadí se do ImageView. Potom se přiřadí do textových polí zbývající parametry.

```
public class Stranka extends Fragment {
    public Stranka() {
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_stranka, container,
                        false);

        // nacteni a prirazeni vstupnich dat do pomocnych promennych
        String strLogo = getArguments().getString("logo");
        String strNazev = getArguments().getString("nazev");
        String strVerze = getArguments().getString("verze");
        String strData = getArguments().getString("data");

        // odkazy na objekty formulare
        ImageView imgLogo = view.findViewById(R.id.imageView);
        TextView txtNazev = view.findViewById(R.id.textView1);
        TextView txtVerze = view.findViewById(R.id.textView2);
        TextView txtData = view.findViewById(R.id.textView3);

        // zpristupneni resources
        Resources res = getContext().getResources();

        // id resource s logem androidu
        int imgLogoId = res.getIdentifier(strLogo,
                        "drawable", getContext().getPackageName());

        // prirazeni loga do ImageView
        imgLogo.setImageResource(imgLogoId);

        // zapis textovych vstupnich dat na formular
        txtNazev.setText(strNazev);
        txtVerze.setText(strVerze);
        txtData.setText(strData);

        return view;
    }
}
```

Zatím jsme se stále nedostali k přípravě vstupních dat. Necháme si je až na konec, až budeme mít připravený adaptér. Předpokládejme ale, že si je ve třídě MainActivity uložíme do pole textových řetězců a předáme je adaptéru. Ten si připravíme právě teď. Stejně jako v předchozím příkladu založíme novou třídu PageAdapter, která bude rozšířením k tomu určené třídy FragmentStateAdapter. Nově si ale do konstruktoru přidáme parametr s polem řetězců, který bude sloužit pro předání vstupních dat:

```
public PageAdapter(@NonNull FragmentActivity fragmentActivity,
                                          String [] predanaData) { }
```

Vstupní data bychom nemuseli nutně předávat z MainActivity, mohli bychom je mít definovaná uvnitř adaptéru. Ale logickou úlohou adaptéru je být jakýmsi technickým prostředníkem, ne zdrojem dat, takže je necháme jednoduše protéct z MainActivity přes adaptér do příslušných fragmentů.

Kód adaptéru bude vypadat zhruba takto:

```
public class PageAdapter extends FragmentStateAdapter {
    private String [] data;
    public PageAdapter(@NonNull FragmentActivity fragmentActivity,
                       String [] predanaData) {
        super(fragmentActivity);

        data=predanaData;
    }

    @Override
    public Fragment createFragment(int position) {
        Stranka fragment = new Stranka();

        // ziskani retezce z pole, podle pozice
        String strItem = data[position];

        // rozdeleni retezce na casti
        String[] strVersion = strItem.split(":");

        // jmeno resource s logem Androidu
        String resName = strVersion[1].replaceAll(" ", "").toLowerCase();

        // odeslani dat do fragmentu
        Bundle bundle=new Bundle();
        bundle.putString("logo", resName);
        bundle.putString("nazev", strVersion [1]);
        bundle.putString("verze", strVersion [0]);
        bundle.putString("data", strVersion [2]);
        fragment.setArguments(bundle);

        return fragment;
    }

    @Override
    public int getItemCount() {
        return data.length;
    }
}
```

Stejně jako v předchozím příkladu, i zde je zásadní metodou adaptéru předefinovaná událost createFragment(). V ní si vezmeme ze vstupních dat záznam odpovídající zobrazované stránce, ten rozparsujeme do textových řetězců, v případě názvu obrázku odstraníme mezery a výsledek zabalíme do struktury Bundle, určené k předávání dat. Vytvoříme instanci fragmentu Stranka, té data přiřadíme a odevzdáme ji v návratové hodnotě.

Druhá předefinovaná událost, getItemCount(), opět jenom informuje o počtu stránek které jsou připravené. Tento počet je rovný počtu záznamů v poli vstupních dat.

Zde je vhodné poznamenat, že kromě použité třídy adaptéru FragmentStateAdapter existuje také třída FragmentStatePagerAdapter s velmi podobným účelem. Rozdíl mezi nimi je ten, že zatímco FragmentStateAdapter je vhodný pro omezený a pevně daný počet zobrazovaných stránek, FragmentStatePagerAdapter se hodí spíše pro dlouhé seznamy s komplikovanějšími stránkami. FragmentStateAdapter totiž udržuje instance všech stránek, po jejich inicializaci, trvale v paměti, zatímco FragmentStatePagerAdapter si využití paměti hlídá a stránky které zrovna nepotřebuje tak zahazuje. Z toho důvodu je sice o něco pomalejší, ale zato výrazně šetrnější z hlediska spotřeby paměti zařízení. Obecně se dá říct, že pokud použijeme ViewPager pro zobrazování konečného počtu předem daných stránek, jako v našem případě, je vhodnější FragmentStateAdapter. Kdybychom ale dělali dlouhý seznam, například katalog výrobků, který by se generoval z databáze, určitě by se slušelo využít FragmentStatePagerAdapter.

Nyní je tedy připravena komponenta ViewPager, adaptér i fragment a už jen zbývá toto všechno propojit ve třídě MainActivity. K tomu opět využijeme její událost onCreate(). Nejdřív vytvoříme pole vstupních dat. Každý element pole bude reprezentovat jednu verzi Androidu. Verzi, název a informace o aktualizacích a API. Tato textová data budou vzájemně oddělena dvojtečkou a dohromady budou tvořit jeden dlouhý text. Výsledkem tedy bude jednorozměrné pole textových řetězců. Data samozřejmě můžeme pro předání uložit jakýmkoliv jiným způsobem, důležité je, abychom je potom uměli v adaptéru zase rozložit na jednotlivé atributy.

Vytvoříme instanci adaptéru, které připravená data předáme ve vstupním parametru a adaptér propojíme s komponentou ViewPager.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // vstupní data
    // vzdy tri texty oddelene dvojteckou, adapter si je rozdeli
    String [] arrVerzeAndroidu = {

        ...

        "Android 7.0:Nougat:Aktualizace 7.1.1 a 7.1.2\nAPI verze 24, 25",
        "Android 8.0 - 8.1:Oreo:Bez aktualizací\nAPI verze 26.27",
        "Android 9.0:Pie:Bez aktualizací\nAPI verze 28",
        "Android 10:Android 10:Bez aktualizací\nAPI verze 29"
    };

    // nova instance adapteru
    PageAdapter mAdapter = new PageAdapter(this, arrVerzeAndroidu);

    // prirazeni adapteru k ViewPager
    ViewPager2 viewPager = (ViewPager2) findViewById(R.id.viewPager);
    viewPager.setAdapter(mAdapter);
}
```

Teď bychom mohli výsledek zkompilovat a spustit na emulátoru nebo koncovém zařízení. Takto spuštěná aplikace by se měla chovat v podstatě stejně jako aplikace z předchozího příkladu. Zobrazí informaci o verzi Androidu, posunováním se dá přecházet k dalším verzím a kromě většího počtu stránek by se neměla vizuálně nijak výrazně lišit.

Jsou zde ale další možnosti úprav. Pokud by nám nevyhovoval horizontální směr přesouvání stránek, čili zleva doprava a obráceně, lze ho snadno změnit na vertikální. Pouze ve třídě MainActivity v události onCreate() nastavíme komponentě ViewPager vertikální orientaci:

```
viewPager.setOrientation(ViewPager2.ORIENTATION_VERTICAL);
```

Podobně snadno můžeme ovlivnit i rozestup jednotlivých stránek při posouvání a nastavit mezi nimi větší mezeru, například na 500 obrazových bodů:

```
viewPager.setPageTransformer(new MarginPageTransformer(500));
```

Takové drobné úpravy se jistě mohou hodit, ale při používání Androidu narážíme na mnohem efektnější transformace při přepínání stránek. Jak se jich dosahuje? K tomu slouží rozhraní ViewPager2.PageTransformer, které umožňuje nadefinovat vlastní grafickou transformaci, která se bude zobrazovat při vykreslování posunovaných stránek.

Pro vlastní transformaci musíme udělat novou třídu, nazvěme ji například PageTransformer. Tu rozšíříme o interface ViewPager2.PageTransformer. Důležitá je její metoda transformPage(), ve které se odehrává vše podstatné. Má dva parametry, view který obsahuje referenci na frame, kterým se posunuje a position, která udávý aktuální polohu posunutého frame. Position nabývá hodnot z otevřeného intervalu od -1 do 1, kde 0 je stav kdy je frame uprostřed displeje, ve svojí výchozí pozici, -1 když je posunutý úplně vlevo a 1 když je posunutý do krajní pozice vpravo. Kód jednoduché třídy která provádí grafickou transformaci:

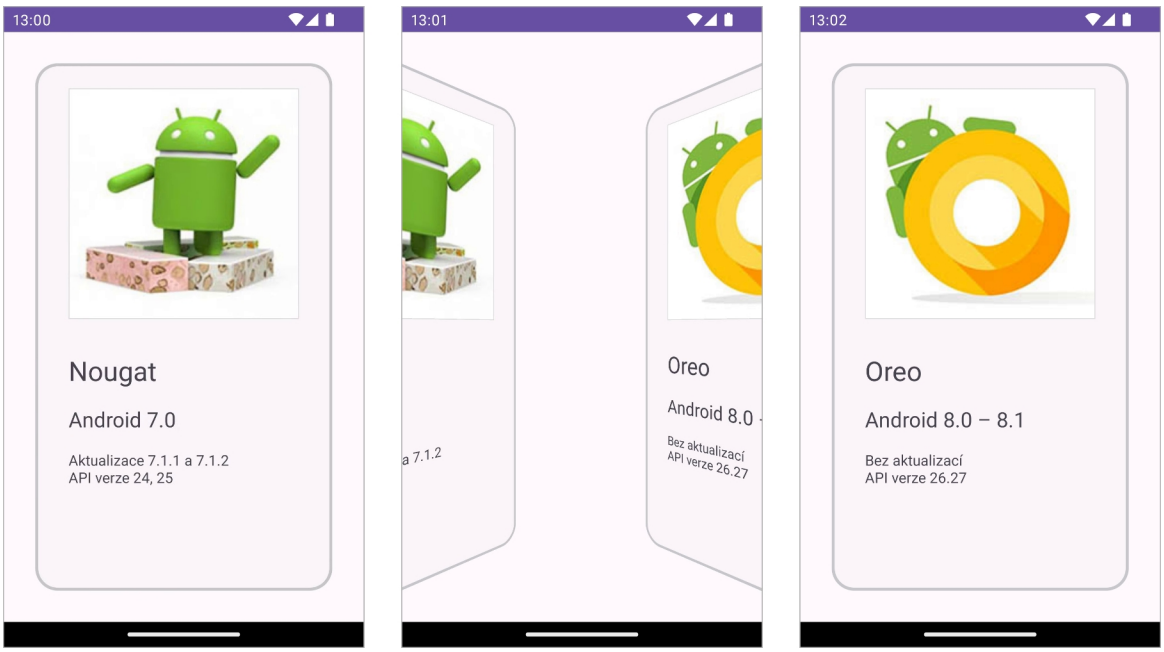
```
public class PageTransformer implements ViewPager2.PageTransformer{
    @Override
    public void transformPage(View view, float position) {

        // transformace 3d rotace karty
        view.setRotationY(-position*35f);
    }
}
```

Během posunování po displeji se hodnota position mění a před každým vykreslením je zavolána metoda transformPage(). Ta v tomto případě prostorově pootočí frame o 0 až 35 stupňů, podle toho ve které pozici se zrovna nachází. Tím vznikne optický dojem jakési jednoduché plastičnosti. PageTransformer je ještě potřeba zaregistrovat v MainActivity:

```
viewPager.setPageTransformer(new PageTransformer());
```

Transformace bývají často mnohem složitější. Využívají změny průhlednosti frame, jeho barvy, posouvají ho různými způsoby nebo s ním divoce rotují. Nejsou zde žádné hranice, vše je jenom na fantazii programátora. Velké množství transformací je volně k dispozici na internetu, stačí je vzít, případně upravit a podle potřeby rovnou použít.

Výsledek naší jednoduché 3d transformace by měl v emulátoru nebo na koncovém zařízení vypadat jako na obrázku 14.38.

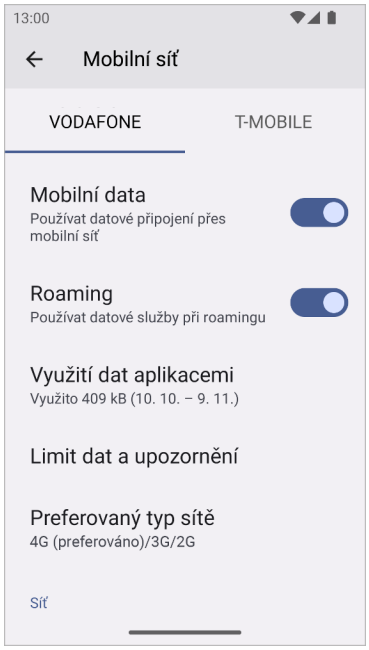


Obrázek 14.38: Aplikace s kontejnerem View Pager, spuštěná na emulátoru

14.8 Tab Layout

Ačkoliv je Tab Layout komponenta, kterou lze používat samostatně, většinou ji budeme vídat pohromadě s View Pager. Jejím nejčastějším užitím je prvek uživatelského rozhraní, takzvané záložky. Tab Layout se stará o jejich přepínání a View Pager o zobrazení obsahu. Ten, jak již víme z předchozí kapitoly, obvykle bývá zapouzdřený ve fragmentu.

Komponenta Tab Layout byla dřív v systému Android hojně využívána, v modernějších verzích ale mizí a je nahrazována jinými prvky. Stále se s ní dá ale setkat například při nastavení mobilních sítí. Pozná se podle panelu umístěného zpravidla v horní části displeje, který sdružuje několik záložek, z nichž právě zvolená je označena podtržením. Přepínat se mezi záložkami lze buď kliknutím na jednu z nich, nebo posunutím celého formuláře do strany.



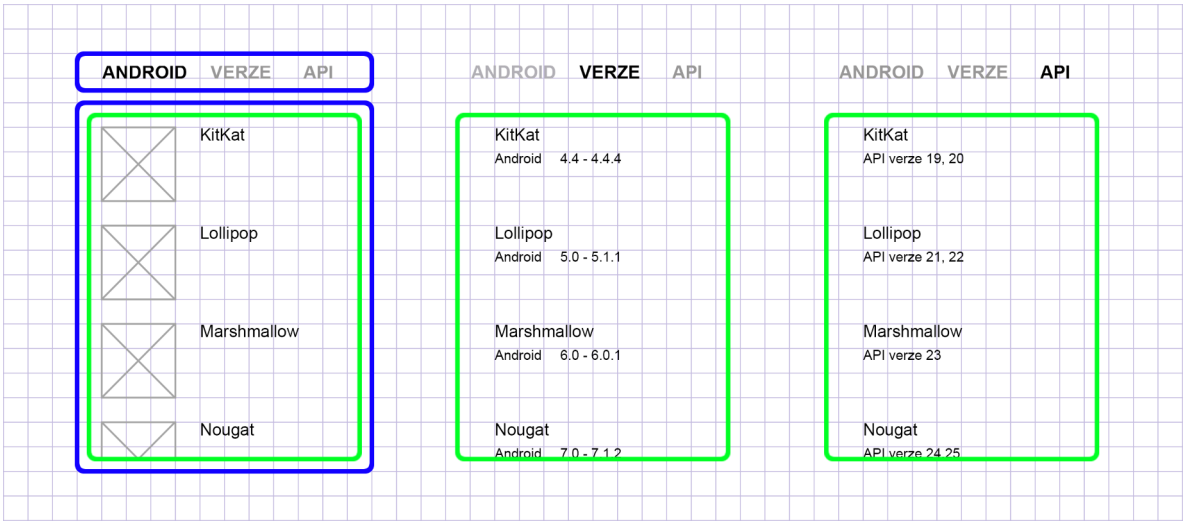
Obrázek 14.39: Příklad kontejneru typu Tab Layout v systému Android

S komponentou View Pager budeme pracovat stejně jako v předchozí kapitole, propojíme ji s fragmenty pomocí adaptéru. Tab layout umožňuje přímé propojení s komponentou ViewPager, ale pouze v jeho první verzi. Novější, ViewPager2 se takto zatím připojit nedá, musí se použít takzvaný mediátor. Naštěstí není příliš složitý, představuje pouhých několik řádek kódu. Vzájemné propojení jednotlivých prvků tedy bude TabLayout ↔ mediátor ↔ ViewPager2 ↔ adaptér ↔ fragmenty.

14.8.1 Tab Layout, příklad

Příklad použití Tab Layout kontejneru bude co možná nejjednodušší. Využijeme komponentu TabLayout, která bude ovládat ViewPager2 se třemi frame. Na každém bude jednoduchý Scroll View. První záložka bude obsahovat posouvatelný seznam log Androidu, druhá seznam verzí Androidu a třetí seznam API Androidu. Záložky budou mít pouze textový nadpis.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt TabLayout



Obrázek 14.40: Návrh formuláře s kontejnerem Tab Layout

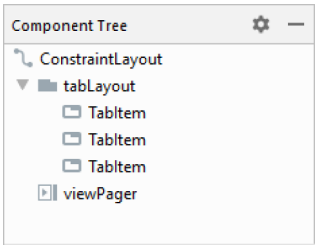
Co je potřeba udělat?

- uložit do resources vhodné obrázky pro první a texty pro další dvě záložky
- vytvořit formulář s kontejnery TabLayout a ViewPager2
- připravit tři fragmenty, každý bude sloužit jako jedna záložka
- napsat jednoduchý adaptér, který se postará o propojení ViewPager2 s fragmenty
- ve třídě MainActivity.java iniciovat v události onCreate() záložky
- vyplnit první fragment kontejnerem Scroll View a další dva textovým obsahem

Do složky res/drawable/ na disku počítače, vložíme loga jednotlivých verzí Androidu, nebo jiné vhodné obrázky přiměřené velikosti. Použijeme je pro první záložku. Texty pro zobrazení ve druhé a třetí záložce vytvoříme tak, že do zdrojů res/values/strings.xml vložíme dva dlouhé řetězce s názvy verzeAndroidu a verzeAPI. Postup pro vytvoření textových řetězců je stejný jako v kapitole 14.3 Scroll View.

V souboru activity_main.xml vložíme kontejner TabLayout tak, aby měl na výšku nastavený rozměr wrap_content. Tedy aby byl pouze tak vysoký, jak je to nezbytně nutné pro zobrazení obsahu. Šířku nastavíme přes celý displej, tj. na match_constraint. Název nastavíme na „tabLayout“, abychom se na něj mohli později odkazovat. Do kontejneru můžeme vložit tři záložky TabItem. Není to nutné, budeme je přidávat programově, ale když si je tam dáme i v designeru, tak lépe uvidíme jak bude výsledek vypadat.

Pod TabLayout přidáme komponentu ViewPager. Pokud ji vkládáme v editoru formuláře, přepneme se do textového editoru a v XML kódu změníme ViewPager na ViewPager2. Při vkládání komponenty v XML tato úprava odpadá, vložíme rovnou modernější verzi komponenty. Umístíme ji tak, aby vyplňovala celou zbývající plochu formuláře. Komponentě nastavíme id „viewPager“. Tím je aktivita připravena.



Obrázek 14.41: Prvky formuláře activity_main v okně Component Tree. Záložky TabItem nejsou nezbytně nutné.

Aby měla komponenta ViewPager2 co přepínat, musíme jí připravit fragmenty s obsahem. Fragment se vytvoří v panelu projektu, v kontextovém menu New → Fragment → Fragment (Blank). Pojmenujeme ho ZalozkaAndroid a zatím v něm necháme automaticky vygenerovaný obsah. Svůj vlastní doplníme později. Stejně vygenerujeme fragmenty ZalozkaVerze a ZalozkaApi.

Na to, abychom mohli přepínat jednotlivé fragmenty v komponentě ViewPager2, potřebujeme adaptér který zajistí jejich střídání podle stavu TabLayout. Adaptér si celkem jednoduše napíšeme. Nejdřív je třeba založit prázdnou třídu v panelu projektu, v kontextovém menu New → Java Class a nazvat ji nějakým příhodným způsobem, například TabAdapter. Adaptér bude rozšiřovat třídu FragmentStateAdapter a bude vypadat takto:

```
public class TabAdapter extends FragmentStateAdapter {

    public TabAdapter(@NonNull FragmentActivity fragmentActivity) {
        super(fragmentActivity);
    }

    @Override
    public Fragment getItem(int position) {
        switch (position){
            case 0: return new ZalozkaAndroid();
            case 1: return new ZalozkaVerze();
            case 2: return new ZalozkaApi();
        }
        return null;
    }

    @Override
    public int getItemCount() {
        return 3;
    }
}
```

V konstruktoru pouze zavoláme, pomocí instrukce super, nadřazenou třídu. Důležité jsou předefinované metody – getItem je volaná při přepínání stránek a stará se o přiřazení nové instance správného Frame. Stačí, abychom v ní vytvořili instanci toho správného fragmentu podle parametru position. Předefinovaná metoda getItemCount() pouze indikuje počet záložek které mají být zobrazeny.

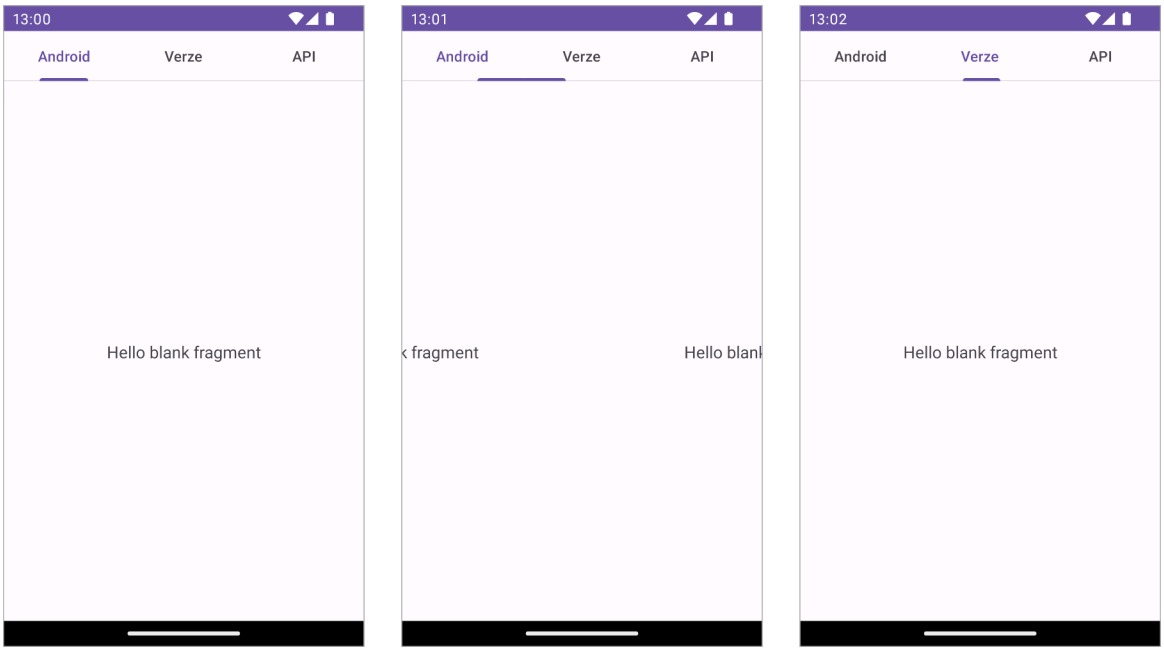
Zbývá propojit TabLayout s ViewPager2 a ViewPager2 s adaptérem. Ve třídě MainActivity přidáme do události onCreate() kód který najde na formuláři jak TabLayout tak ViewPager2 a na oba si uloží ukazatel do lokálních proměnných. Vytvoříme instanci adaptéru a připojíme ji metodou setAdapter() k viewPageru.

Trošku složitější je propojení TabLayout a ViewPager2. TabLayout nemá podporu pro ViewPager2 který jsme použili. Nelze ho tedy (alespoň prozatím) připojit přímo, ale musí se to udělat prostřednictvím TabLayoutMediator – třídy, která umí synchronizovat posunování stránky ve ViewPager2 a záložky v TabLayout. Vytvoříme instanci TabLayoutMediator, která bude mít v parametrech tabLayout i viewPager a instanci konfigurátoru TabConfigurationStrategy, jejíž událost onConfigureTab() se vyvolá během inicializace. Jediné co je potřeba konfigurátoru předat jsou názvy záložek. Takto vypadá upravený kód události onCreate() třídy MainActivity:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // nova instance adapteru
    TabAdapter mAdapter = new TabAdapter(this);

    // najit na formulari TabLayout a ViewPager
    TabLayout tabLayout = (TabLayout) findViewById(R.id.tabLayout);
    ViewPager2 viewPager = (ViewPager2) findViewById(R.id.viewPager);

    // propojeni ViewPager s adapterem
    viewPager.setAdapter(mAdapter);

    // nazvy zalozek
```

```
    final String[] zalozky = new String[]{"Android", "Verze", "API"};

    // propojeni TabLayoutu s ViewPager
    new TabLayoutMediator(tabLayout, viewPager,
            new TabLayoutMediator.TabConfigurationStrategy() {
                @Override public void onConfigureTab(TabLayout.Tab tab,
                                                     int position) {
                    tab.setText(zalozky[position]);
                }
            }).attach();
}
```

Tím je hotový základ pro přepínání fragmentů. Když spustíme aplikaci na emulátoru, měl by se zobrazit TabLayout se třemi záložkami, mezi kterými jde přepínat posouváním do stran nebo klikáním na záhlaví.



Obrázek 14.42: Přepínání TabLayoutTab posouváním do strany

Zbývá doplnit obsah fragmentů. To už uděláme v XML, bez dalšího Java programování. V designéru otevřeme soubor fragment_zalozka_android.xml, vložíme do něj kontejner ScrollView. Do toho běžným způsobem umístíme sadu formulářů LinearLayout, z nichž každý bude tvořit jeden řádek s obrázkem a názvem verze Androidu.

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ZalozkaAndroid">

    <!-- linear layout, s obsahem prvni zalozky
         sirka a vyska max, vertikalni -->
    <LinearLayout
        android:id="@+id/tab1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
```

```
        <!-- scroll view, sirka a vyska max -->
        <ScrollView
            android:layout_width="match_parent"
            android:layout_height="match_parent">

            <!-- linear layout, sdruzuje vsechny radky
                 sirka a vyska max, vertikalni -->
            <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:orientation="vertical">
                <!-- linear layout, tvori jednu radku (obrazek + text)
                     sirka a vyska max, horizontalni -->
                <LinearLayout
                    android:layout_width="match_parent"
                    android:layout_height="match_parent"
                    android:orientation="horizontal">
                    <!-- ImageView - obrazek verze androidu z resources
                         sirka a vyska 120dp, okraj 2dp, sedy ramecek 1dp -->
                    <ImageView
                        android:layout_width="120dp"
                        android:layout_height="120dp"
                        android:layout_margin="2dp"
                        android:background="#dddddd"
                        android:padding="1dp"
                        android:src="@drawable/icecreamsandwich" />
                    <!-- textove pole s verzi Androidu
                         sirka max, vyska 50dp, okraj 5dp
                         velikost textu 30sp -->
                    <TextView
                        android:text="Android 4.0\nIce Cream Sandwich"
                        android:layout_width="match_parent"
                        android:layout_height="50dp"
                        android:layout_margin="5dp"
                        android:textSize="20sp"/>
                </LinearLayout>

                ...

            </LinearLayout>
        </ScrollView>
    </LinearLayout>
</FrameLayout>
```

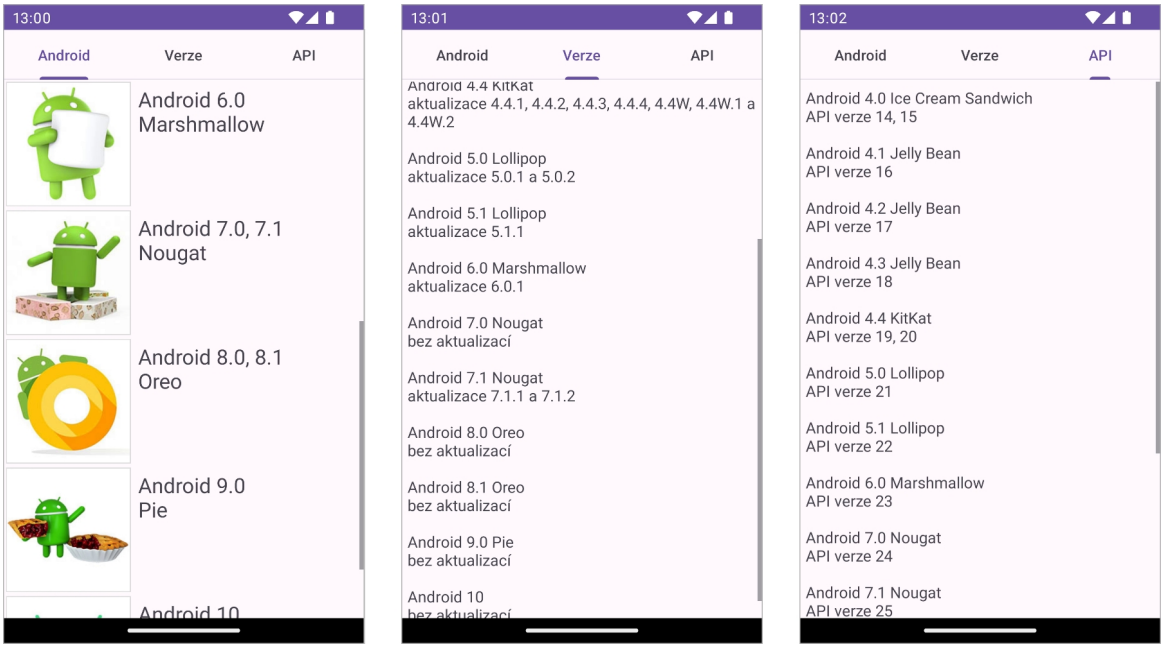
Druhá záložka bude obsahovat ve ScrollView pole TextView s dlouhým textovým řetězcem, který je připravený v resources pod názvem verzeAndroidu.

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ZalozkaVerze">

    <!-- linear layout, s obsahem zalozky Verze
         sirka a vyska max, vertikalni -->
    <LinearLayout
        android:id="@+id/tab2"
        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"
        android:orientation="vertical">

        <!-- scroll view, sirka a vyska max -->
        <ScrollView
            android:layout_width="match_parent"
            android:layout_height="match_parent">

            <!-- textove pole, sirka max, vyska 35dp
                 okraj 5dp, velikost textu 15sp -->
            <TextView
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:layout_margin="5dp"
                android:textSize="15sp"
                android:text="@string/verzeAndroidu" />
        </ScrollView>
    </LinearLayout>
</FrameLayout>
```

Prakticky stejným způsobem se naplní i fragment třetí záložky, pouze vyměníme obsah textového pole za řetězec verzeAPI. Java kód ve všech třech fragmentech můžeme nechat tak, jak ho Android Studio vygenerovalo. Nebudeme ho potřebovat.

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 14.43.



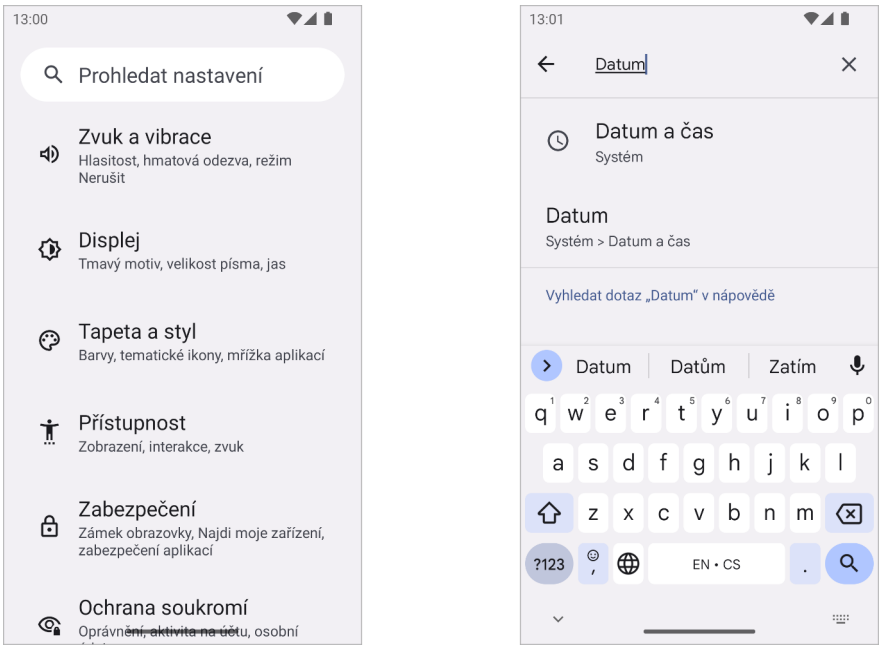
Obrázek 14.43: Aplikace s kontejnerem typu Tab Layout, spuštěná na emulátoru

14.9 Search View

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt SearchView

Jakmile má uživatel k dispozici větší množství dat, vzniká potřeba snadného vyhledávání. Pro tyto účely je v Androidu k dispozici univerzálně použitelná komponenta Search View. S její pomocí lze snadno a rychle implementovat rozhraní pro vyhledávání v textu, seznamech, videích atd. V uživatelském rozhraní může být komponenta umístěna buď v záhlaví displeje nebo kdekoliv ve formuláři.

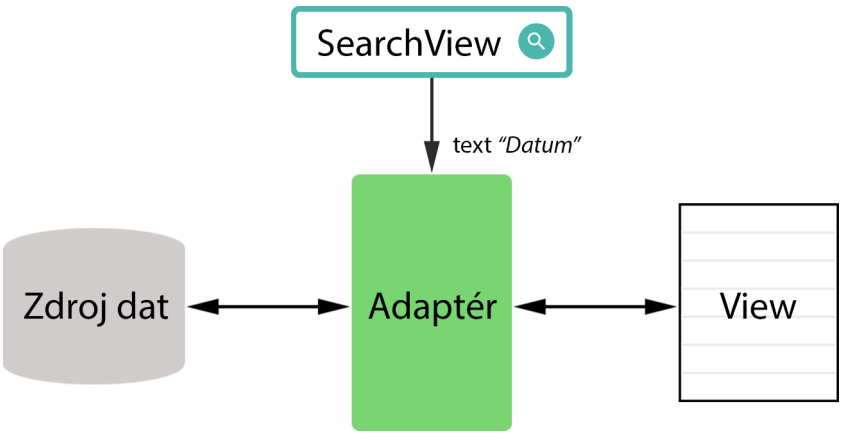
S komponentou Search View se v systému Android můžeme setkat například při kliknutí na ikonku lupy v nastaveních telefonu.



Obrázek 14.44: Příklad aplikace s kontejnerem typu Search View

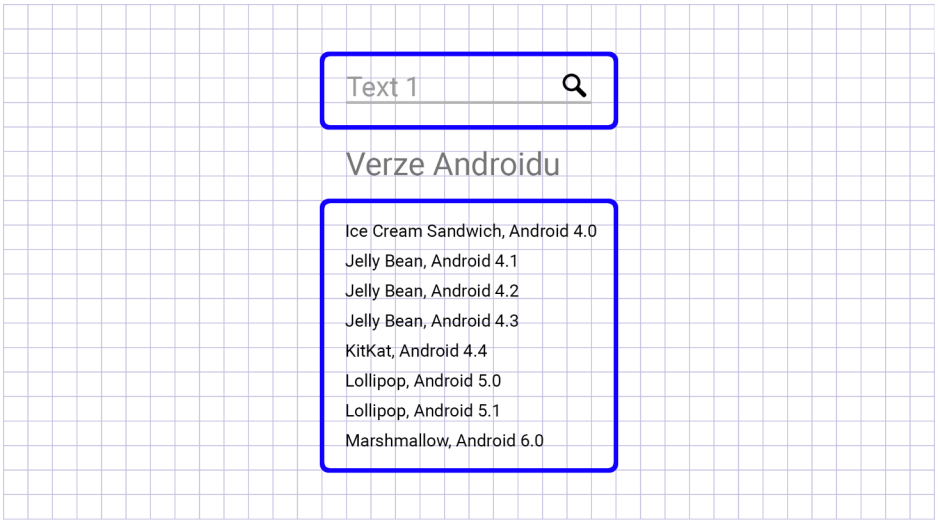
Základní princip vyhledávání přes SearchView spočívá v tom, že data pro zobrazení dodává zdroj dat prostřednictvím adaptéru, který určuje co se má zobrazit. Komponenta SearchView, na základě akce uživatele, vygeneruje text pro hledání. Například „Datum“. Tento text předá adaptéru který zpracovává data pro View a ten už zcela sám zařídí, aby do View přišly pouze ty záznamy, které vyhledávanému řetězci odpovídají.

Funkce adaptéru je znázorněna na následujícím obrázku:



Obrázek 14.45: Propojení zdroje dat a kontejneru Recycler View pomocí adaptéru a Search View

Vytvoříme si příklad který bude vycházet z prvního příkladu použití RecyclerView. Pouze rozšíříme seznam položek a upravíme je tak, aby se v nich dalo hledat. Do záhlaví formuláře přidáme vyhledávací pole a pokud uživatel zadá text, budeme podle něj filtrovat zobrazené položky.



Obrázek 14.46: Návrh formuláře s kontejnerem Search View

Co je potřeba udělat?

- vytvořit layout položky formuláře a holder
- napsat adaptér
- připravit data a zajistit jejich načtení
- propojit adaptér s kontejnerem RecyclerView
- založit menu s komponentou SearchView
- přidat do formuláře komponentu ToolBar, která bude tvořit záhlaví
- nastavit výstup ze SearchView do adaptéru

S položkou formuláře a jejím holderem to bude snadné. Použijeme ho z prvního příkladu komponenty RecyclerView. Nejsou potřeba žádné modifikace, obojí vyhovuje v původní podobě.

S adaptérem to bude trochu složitější. Vezmeme ho také z uvedeného příkladu RecyclerView, ale rozšíříme ho o rozhraní Filterable. Vyhledávání bude vlastně takovým filtrováním, které začneme vždy nad seznamem všech položek a z nich vybereme jenom ty, které odpovídají uživatelem zadanému řetězci. Toto uděláme vždy, když do vyhledávacího pole nějaký znak připíše, nebo ho naopak smaže. Proto budeme v adaptéru potřebovat navíc seznam všech položek, ke kterému se budeme při každém filtrování vracet. Zřídíme si pro něj lokální proměnnou androidVerzeVse a v konstruktoru třídy ji naplníme kopií dat. Do původní proměnné androidVerze budeme ukládat data už vyfiltrovaná. Základ adaptéru bude tedy vypadat takto:

```
public class RecyclerAdapter extends RecyclerView.Adapter<RecyclerHolder>
                                            implements Filterable {
    private List<String> androidVerze;
    private List<String> androidVerzeVse;

    public RecyclerAdapter (List<String> data) {
        this.androidVerze = data;
        this.androidVerzeVse = new ArrayList<>(data);
    }
...

}
```

Přepsané události, týkající se holderu, onCreateViewHolder() a onBindViewHolder() vezmeme z prvního příkladu RecyclerView, není je potřeba nijak měnit. Stejně tak událost getItemCount} nepotřebuje žádnou úpravu, bude vracet počet záznamů seznamu vyfiltrovaných položek androidVerze.

S rozšířením o rozhraní Filterable získáme událost getFilter(), která slouží k předání filtru, který bude následně volaný pokaždé, když uživatel v poli pro hledání něco změní. Filtr si nazvěme třeba androidVerzeFiltr.

```
@Override
public Filter getFilter() {
    return androidVerzeFiltr;
}
```

Samotný filtr musíme samozřejmě také implementovat. Bude to privátní metoda typu Filter, což je abstraktní třída se dvěma metodami, performFiltering() a publishResults(). Obě musíme přepsat, první slouží k vyhledání záznamů, které odpovídají uživatelem zadanému vyhledávacímu řetězci, předanému parametrem constraint a druhá k předání výsledků v okamžiku, kdy je komponenta RecyclerView bude potřebovat k zobrazení.

```
private Filter androidVerzeFiltr = new Filter() {
    @Override
    protected FilterResults performFiltering(CharSequence constraint) {
        List<String> filteredList = new ArrayList<>();
        if (constraint == null || constraint.length() == 0) {
            filteredList.addAll(androidVerzeVse);
       } else {

            String filtrovacíText = constraint.toString().toLowerCase().trim();
            for (String item : androidVerzeVse) {
                if (item.toLowerCase().contains(filtrovacíText)) {
                    filteredList.add(item);
```

```
                }
            }
        }
        FilterResults results = new FilterResults();
        results.values = filteredList;
        return results;
    }
    @Override
    protected void publishResults(CharSequence constraint,
                                                FilterResults results) {
        androidVerze.clear();
        androidVerze.addAll((List) results.values);
        notifyDataSetChanged();
    }
};
```

Metoda performFiltering() nejdřív zkontroluje, jestli je zadaný nějaký vyhledávaný řetězec. Pokud ne, vrátí všechny záznamy, jak je má uložené v seznamu androidVerzeVse. Pokud řetězec zadaný je, prochází postupně jeden záznam ze seznamu androidVerzeVse za druhým a testuje, jestli řetězci odpovídá. Odpovídající záznamy si ukládá do pomocného seznamu. Nakonec výsledek vrátí ve struktuře FilterResults. V této metodě můžeme úpravou algoritmu zařídit, aby se filtrování vztahovalo pouze na první písmena v textu položky nebo aby se za odpovídající bral i text uprostřed, můžeme zajistit rozlišování velkých a malých písmen, nebo naopak jejich ignorování. Můžeme také třeba přeskakovat mezery a tak podobně. V zásadě se vše podstatné skrývá v podmínce

```
if (item.toLowerCase().contains(filtrovacíText)) {
    ...
}
```

Tak jak je, vyhledává všechny položky, které kdekoliv v jejím textu obsahují řetězec z proměnné filtrovacíText, přičemž nezohledňuje malá a velká písmena.

Druhá metoda, publishResults(), má v parametru strukturu s vyfiltrovaným seznamem. Tím jednoduše nahradí data v seznamu androidVerze a dá komponentě RecyclerView zavoláním její metody notifyDataSetChanged() vědět, že došlo ke změně vstupních dat. O jejich obnovení na displeji už se RecyclerView postará sám.

To je vše, co potřebujeme aby adaptér uměl. Nyní tedy víme, že dostane seznam dat a když uživatel něco změní ve vyhledávacím poli, tak bude mít proměnné androidVerze pouze ty položky, které odpovídají řetězci z vyhledávacího pole.

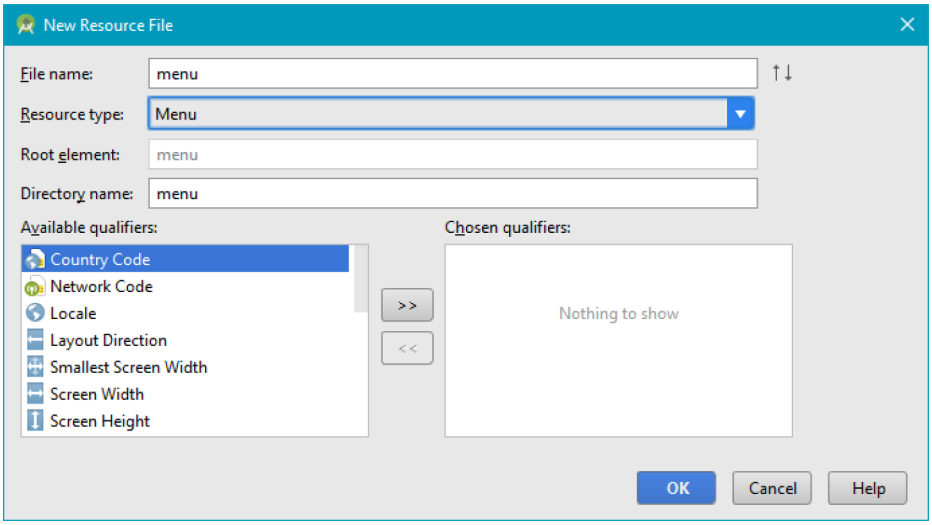
Data pro zobrazení si vytvoříme tak, že do zdrojů res/values/strings.xml vložíme pole řetězců s názvem androidVerze:

```
<string-array name="androidVerze">
    <item>Alpha</item>
    <item>Beta</item>
    <item>Cupcake, Android 1.5</item>
    <item>Donut, Android 1.6</item>
    <item>Eclair, Android 2.0</item>
    …
</string-array>
```

Ve třídě MainActivity v události onCreate() zavoláme vlastní metodu naplnitRecyclerView(), která vygeneruje vstupní data, vytvoří instanci našeho nového adaptéru a napojí ho na kontejner RecyclerView. Postup je podobný jako v příkladu RecyclerView, pouze si navíc uděláme proměnnou s ukazatelem na adaptér a vstupní data načteme z resources.

```
// promenna s ukazatelem na adapter
RecyclerAdapter adapter;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    naplnitRecyclerView();
}

// metoda nastavi hodnoty do RecyclerView recViewVerzeAndroidu
private void naplnitRecyclerView() {

    // nacist data z resources
    ArrayList<String> arrVerze = new ArrayList<String>();
    arrVerze.addAll(Arrays.asList(getResources()
                    .getStringArray(R.array.androidVerze)));

    // vytvorit instanci adapteru
    adapter = new RecyclerAdapter(lstVerzeAndroidu);

    // najit na formulari kontejner RecyclerView
    RecyclerView recyclerView =
                        (RecyclerView) findViewById(R.id.recViewVerzeAndroidu);

    // vytvorit a priradit Layout Manager
    recyclerView.setLayoutManager(new LinearLayoutManager(this));

    // nastavit oddelovace radku
    recyclerView.addItemDecoration(new DividerItemDecoration(this,
                        DividerItemDecoration.VERTICAL));

    // nastavit pevnou velikost recyclerView (nemenit ji podle poctu polozek)
    recyclerView.setHasFixedSize(true);

    // propojeni adapteru s kontejnerem RecyclerView
    recyclerView.setAdapter(adapter);
}
```

Založíme menu, které bude obsahovat komponentu SearchView. Nejprve vytvoříme složku menu. V okně projektu zvolíme nad složkou res pomocí kontextové nabídky New → Android resource directory. Otevře se dialog, v něm nastavíme název „menu“ a typ resource „Menu“.



Obrázek 14.47: Průvodce založením nového zdrojového souboru typu Menu

Vygenerujeme soubor menu - pomocí kontextové nabídky New → Menu resource file založíme soubor menu_search.xml. Otevřeme nově vzniklý soubor a přidáme do něj položku menu.

```
<menu xmlns:tools="http://schemas.android.com/tools"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/android">

    <!-- polozka menu, typ SearchView
         zobrazovat v zahlavi vzdy, default ikonka, text Hledat -->
    <item
        android:id="@+id/menuSearch"
        app:actionViewClass="android.widget.SearchView"
        app:showAsAction="always"
        android:title="Hledat" />
</menu>
```

Jelikož formulář nemá žádné vhodné záhlaví, ve kterém by mohlo být umístěné pole pro vyhledávání, tak do activity_main.xml vložíme z palety komponentu ToolBar, stejně jako v příkladu 12.5 Záhlaví formuláře. Nastavíme jí základní vlastnosti, umístění, velikost, barvu a vložený text:

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:minHeight="?attr/actionBarSize"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:titleTextColor="@android:color/white"
    app:title="@string/app_name"
    android:background="?attr/colorPrimary">
</androidx.appcompat.widget.Toolbar>
```

V MainActivity doplníme do události onCreate() dva řádky, z nichž první dohledá na formuláři ToolBar a druhý sdělí aktivitě, že s ním má pracovat jako s vlastním záhlavím:

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

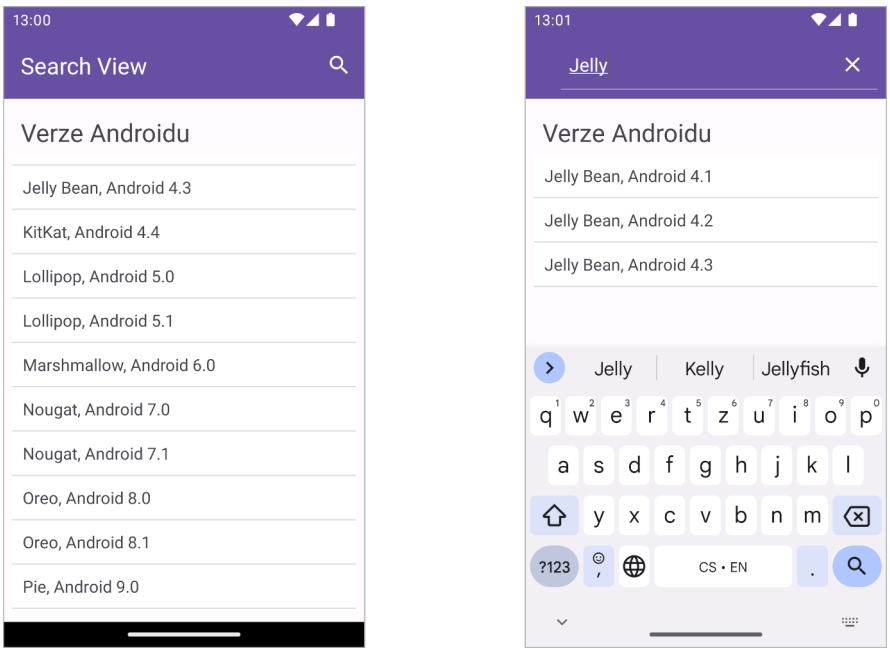
Tím jsou všechny potřebné prvky připravené. Zbývá v ToolBar zobrazit menu a propojit SearchView s adaptérem. Potřebujeme, aby se informace o filtru předávala adaptéru vždy, když uživatel udělá jakoukoliv změnu ve vyhledávacím poli.

Ve třídě MainActivity přepíšeme událost onCreateOptionsMenu() a vložíme do ní iniciaci vyhledávacího boxu. Dále do ní dáme posluchač události setOnQueryTextListener(). Ten se zavolá při každé změně v poli pro vyhledávání, přičemž jako vstupní parametr má hodnotu která se nachází v onom vyhledávacím poli. Potom už stačí pouze tuto hodnotu předat adaptéru.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // pridat do Menu box pro hledani
    MenuInflater inflanter = getMenuInflater();
    inflanter.inflate(R.menu.menu_search, menu);
    MenuItem item = menu.findItem(R.id.menuSearch);
    SearchView searchView = (SearchView)item.getActionView();

    // posluchac udalosti zmeny hodnoty v poli pro hledani
    searchView.setOnQueryTextListener(new SearchView.OnQueryTextListener() {
        @Override
        public boolean onQueryTextSubmit(String query) {
            return false;
        }

        @Override
        public boolean onQueryTextChange(String newText) {
            // nastaveni filtru adapteru
            adapter.getFilter().filter(newText);

            return false;
        }
    });
    return super.onCreateOptionsMenu(menu);
}
```

Co se tedy stane, když uživatel ťukne na ikonku lupy a do vyhledávacího pole začne zadávat text? Při každém vloženém znaku se vyvolá událost onQueryTextChange(), která z vyhledávacího pole celý text vezme a předá adaptéru ve formě filtru. To je všechno. O zbytek se postará adaptér, který je propojený s komponentou RecyclerView a sám si řídí přísun dat určených k zobrazení.

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 14.48.



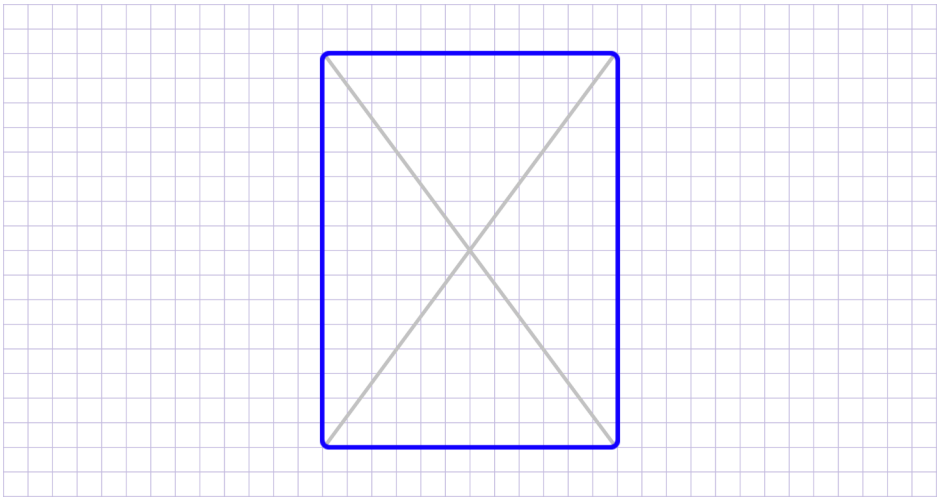
Obrázek 14.48: Aplikace s kontejnerem typu Search View, spuštěná na emulátoru

14.10 Web View

Android umožňuje zobrazování HTML obsahu nejen prostřednictvím browseru, ale i přímo v aplikaci. K tomuto účelu slouží komponenta WebView, která se chová jako integrovaný browser, ovšem s řadou omezení. Dokáže však zobrazit jak lokální HTML dokumenty, tak i online webové stránky. HTML stránky se v aplikacích často používají jako nápověda či dokumentace.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 14_Kontejnery
projekt WebView

Na dvou jednoduchých příkladech si vyzkoušíme zobrazení internetové stránky a vlastního HTML souboru. Obojí zobrazíme v kontejneru WebView.



Obrázek 14.49: Návrh formuláře s kontejnerem Web View. Je opravdu velmi jednoduchý.

Nejprve tedy zobrazení webové stránky.

Co je potřeba udělat?

- vytvořit formulář s kontejnerem Web View

- do kontejneru načíst internetovou stránku

Otevřeme soubor activity_main.xml a do formuláře (v příkladu použijeme Constraint Layout) vložíme kontejner Web View. Rozměry nastavíme tak, aby byla vyplněna celá plocha displeje a aby se při natočení displeje velikost kontejneru přizpůsobila (viz kapitola Constraint Layout). Kontejneru ve vlastnostech nastavíme id webView.

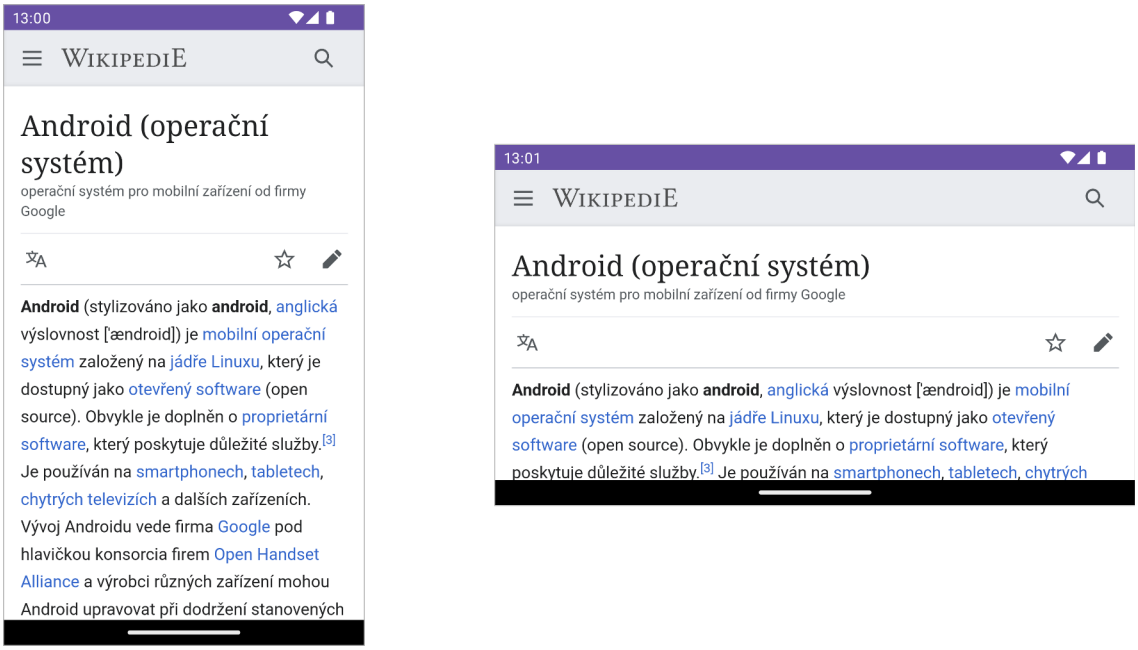
XML kód kontejneru bude tedy vypadat nějak takto:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- kontejner WebView
         sirka a vyska max -->
    <WebView
        android:id="@+id/browser"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Ve třídě MainActivity.java vytvoříme metodu spustitBrowser(), které iniciuje komponentu WebView a načte do ní požadovanou stránku.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    spustitBrowser();
}

private void spustitBrowser()
{
    String url="https://cs.wikipedia.org/wiki/Android_(operační_systém)";
    // najit komponentu WebView
    WebView browser = (WebView) this.findViewById(R.id.webView);
    // nastavit parametry browseru
    browser.getSettings().setJavaScriptEnabled(true);
    browser.setWebViewClient(new WebViewClient());
    // zobrazit webovou stranku
    browser.loadUrl(url);
}
```

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 14.50.



Obrázek 14.50: Aplikace s kontejnerem typu Web View, spuštěná na emulátoru

Stejně jako se občas hodí zobrazit internetovou stránku, tak bývá potřeba zobrazit i lokálně uloženou stránku HTML. Upravíme si předchozí příklad tak, aby komponenta načetla náš vlastní HTML soubor.

Co je potřeba udělat?

- vytvořit HTML soubor a uložit ho na vhodné místo

- obsah souboru načíst do komponenty Web View

Vhodným místem pro umístění vlastního HTML souboru, například s nápovědou pro aplikaci, je složka assets. Je součástí projektu a zároveň, podobně jako složka res, umožňuje pracovat se soubory nezávisle na Android Studiu. Tam lze umístit až výslednou strukturu HTML stránek.

Nyní se spokojíme s jedinou stránkou bez obrázků a odkazů, pouze s formátovaným textem. Nejprve si vytvoříme v adresáři app složku assets, pomocí kontextového menu v okně projektu New → Folder → Assets Folder. Do vzniklé složky umístíme soubor verzeAndroidu.html. Ten si můžeme vytvořit mimo Android Studio, v některé z aplikací určených pro tvorbu HTML stránek. Ale můžeme to udělat například i v Poznámkovém bloku nebo přímo v Android Studiu.

Zde není cílem naučit se pracovat HTML, takže si pouze ukážeme, jak by měl vypadat výsledek:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
```

```
  <head>
    <meta http-equiv="Content-Language" content="cs">
    <meta http-equiv="Content-Type" content="text/html; charset=windows-1250">
    <title>Verze Androidu</title>
  </head>

  <body>
    <font size="5">
      Verze Androidu<br/>
    </font>
      <font size="4">
        <br/>
        <b>Alpha</b><br/>
        <b>Beta</b><br/>
        <b>Cupcake</b>, Android 1.5<br/>
        <b>Donut</b>, Android 1.6<br/>
        <b>Eclair</b>, Android 2.0<br/>
        ...
    </font>
  </body>
</html>
```

Ve třídě MainActivity.java upravíme metodu spustitBrowser(). Url, ve kterém je uložena adresa stránky, nahradíme adresou odkazující na soubor na disku:

```
String url="file:///android_asset/verzeAndroidu.html";
```

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 14.51.

Obrázek 14.51: Aplikace s kontejnerem typu Web View, spuštěná na emulátoru

Komponenta Web View poskytuje celou řadu dalších užitečných služeb. Vývojář může například stránku stáhnout, ale před zobrazením ji upravit nebo finálně zobrazit úplně jiný HTML kód. Bohužel ale Web View obsahuje také chyby a odladění komplikovanější aplikace není úplně jednoduché.

Další nevýhodou je, že aplikace s touto komponentou jsou při schvalování do Google Play posuzovány přísněji, protože je současně bezpečnostním rizikem. Přesto všechno ale existuje mnoho případů, kdy je použití Web View tím nejvhodnějším řešením.

15 Legacy prvky

Během vývoje Androidu postupně dochází k zastarávání jeho částí. Obvykle bývají nahrazeny modernějšími prvky a ty původní jsou označeny jako ‚obsolete' - zastaralé, určené k vyřazení. Kromě nich ale existují prvky, které už jsou sice také nahrazeny modernější variantou, přesto jejich trvalé zrušení Google, přinejmenším prozatím, neplánuje. Je to tak proto, že tyto prvky mají některé vlastnosti, které je potřeba zachovat a které jejich novější alternativy nedokážou plně zastoupit. Jedním z důvodů je například využití prvků uživatelského rozhraní v aplikacích typu Widget. Ty jsou uzpůsobené pouze starším formulářům a komponentám.

Legacy prvkům by se měl vývojář, je-li to možné, vyhnout. Jsou méně výkonné než jejich novější alternativy a obvykle nejsou tak dobře konfigurovatelné. Na druhou stranu ale nic nebrání jejich zcela volnému využití. A důvody pro používání by se našly. Jednak to může být kvůli kontinuitě vývoje, kdy starší aplikace tyto komponenty už dlouho používají, vyhovují a není je potřeba nahrazovat. Dalším důvodem může být snadnost použití. Zatímco například novější Recycler View potřebuje definici vlastního adaptéru, formulář položky a holder, tak starší ListView, aniž by cokoliv z toho potřeboval, zvládne v zásadě totéž na několika řádcích řídícího kódu. A potom jsou to samozřejmě již zmíněné Widget aplikace, kde ani jiná možnost není.

Legacy prvky jsou v designéru Android Studia v paletě komponent v části Legacy. Nahrazení těchto prvků novějšími alternativami, viz tabulka 15.1.

Legacy prvek	Modernější alternativa
RelativeLayout	ConstraintLayout
GridLayout	Constraint Layout
ListView	RecyclerView
GridView	ConstraintLayout nebo RecyclerView spolu s GridLayoutManager
TabHost	TabLayout

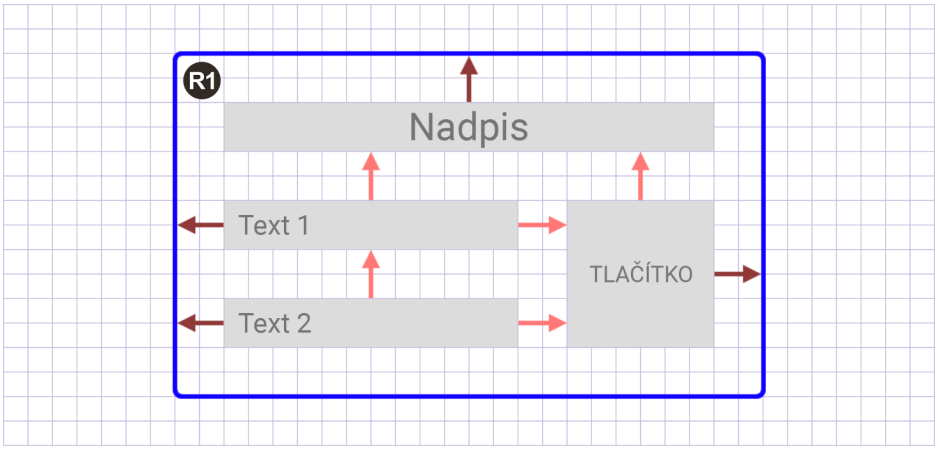
Tabulka 15.1: Legacy prvky a jejich nahrazení moderními alternativami

15.1 Relative Layout

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 15_Legacy
projekt RelativeLayout

V Relative Layoutu se, v souladu s názvem a podobně jako u Constraint Layoutu, nastavuje pozice prvků relativně vůči ostatním objektům uvnitř kontejneru. Relative Layout má pro každý objekt nastavenou svoji relativní polohu vůči jinému prvku. K dispozici jsou možnosti pod, nad a vedle druhého objektu. S těmi si musí vývojář vystačit. Pro každý vkládaný prvek se tedy vybere jiný, už existující, ke kterému se nový objekt vztáhne. Pozici objektů lze také vztáhnout k hranicím kontejneru, což se využije zejména pro první vložený objekt, u kterého ostatně ani jiná možnost není.

Na příkladu si ukážeme, jak objekty k sobě navzájem navázat. Od Constraint Layoutu se uspořádání prakticky neliší:



Obrázek 15.1: Návrh formuláře typu Relative Layout

Kontejner R1 typu Relative Layout je roztažený přes celou šířku displeje. Do kontejneru R1 je vloženo pole typu TextView s textem Nadpis a pevně nastavenou výškou layout_height na 50 dp. Pole je navázáno na horní hranu R1 a je roztažené přes celou šířku kontejneru.

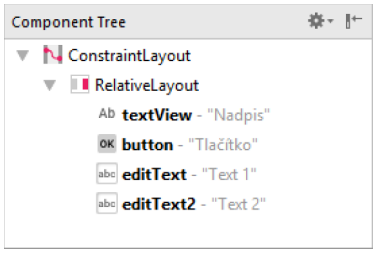
V R1 je vložené tlačítko 100 x 100 dp, navázané na pravou hranu kontejneru R1. Tím je zajištěno, že bude pro libovolný rozměr displeje zcela vpravo. Současně je navázané na textové pole Nadpis, aby mělo pevné vertikální umístění.

Pole Text 1 a Text 2, typu Plain Text, mají nejvíc vazeb. Zleva jsou navázané na hranu kontejneru R1 a napravo na tlačítko. Tím je zajištěno, že se pole při změně velikosti displeje adekvátně roztáhnou. Horní hrana prvního pole je navázána na TextView Nadpis, aby získala pevné umístění ve svislé ose. Ze stejného důvodu je druhé pole PlainText navázáno na spodní hranu prvního pole.

Na obrázku jsou pro názornost mezi objekty mezery, ve skutečnosti však potřebujeme, aby se pole dotýkala hranami, proto u všech objektů nastavíme okraje velikosti 0 dp.

Způsob, jak v XML zapsat vazby na kontejner (tmavší šipky na obrázku 15.1), nastavení vazeb na jiný objekt (světlejší šipky na obrázku 15.1) a nastavení velikosti okrajů objektů jsou shodné s kontejnerem Constraint Layout.

Relative Layout má plochou strukturu bez nutnosti zanořování, v okně Component Tree by tedy mělo být uspořádání prvků jako na obrázku 15.2.



Obrázek 15.2: Prvky formuláře v okně Component Tree

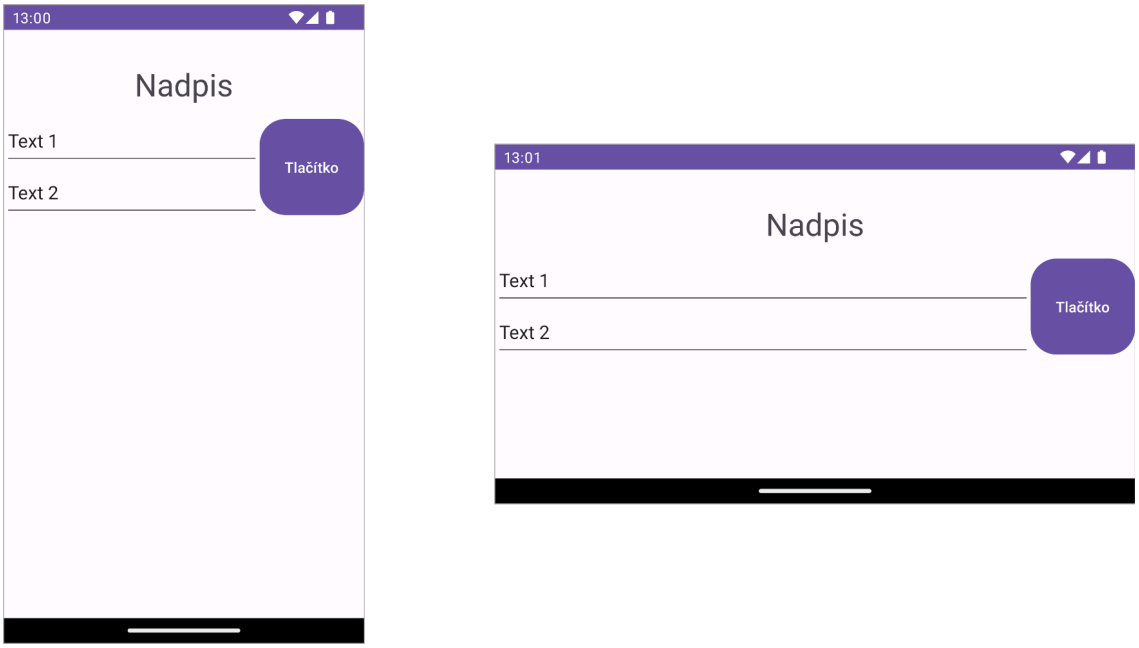
Výsledný XML kód:

```
<!-- relativni layout R1, sirka max, vyska 150dp -->
<RelativeLayout
    android:layout_width="match_parent"
```

```
android:layout_height="match_parent"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintStart_toStartOf="parent"
tools:layout_editor_absoluteX="0dp"
tools:layout_editor_absoluteY="0dp">

<!-- textove pole Nadpis, sirka max, vyska 50dp
     text vycentrovany, velikost textu 30sp
     umistene k horni hrane R1 -->
<TextView
    android:id="@+id/textView"
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:layout_alignParentTop="true"
    android:layout_centerHorizontal="true"
    android:text="Nadpis"
    android:textAlignment="center"
    android:textSize="30sp" />

<!-- tlacitko 100x100dp
     vztazene ke spodni hrane textWiev, okraje 0dp
       a k prave hrane R1, okraje 0dp -->
<Button
    android:id="@+id/button"
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:layout_alignParentEnd="true"
    android:layout_below="@+id/textView"
    android:layout_marginTop="0dp"
    android:text="Tlačítko" />

<!-- prvni pole pro zadani textu, typ text
     sirka max, vyska 50dp, prednastaveny obsah 'Text 1'
     vztazene ke spodni hrane objektu textWiev, vzdalenost 0dp
       a k leve hrane R1, okraje 0dp
       a k leve hrane objektu button, okraje 0dp -->
<EditText
    android:id="@+id/editText"
    android:layout_width="wrap_content"
    android:layout_height="50dp"
    android:layout_alignParentStart="true"
    android:layout_below="@+id/textView"
    android:layout_marginEnd="0dp"
    android:layout_marginStart="0dp"
    android:layout_marginTop="0dp"
    android:layout_toStartOf="@+id/button"
    android:inputType="text"
    android:text="Text 1" />

<!-- druhe pole pro zadani textu, typ text
     sirka max, vyska 50dp, prednastaveny obsah 'Text 2'
     vztazene ke spodni hrane objektu editText, vzdalenost 0dp
       a k leve hrane R1, okraje 0dp
       a k leve hrane objektu button, okraje 0dp -->
<EditText
    android:id="@+id/editText2"
    android:layout_width="wrap_content"
    android:layout_height="50dp"
    android:layout_alignParentStart="true"
    android:layout_below="@+id/editText"
    android:layout_marginEnd="0dp"
```

```
            android:layout_marginStart="0dp"
            android:layout_marginTop="0dp"
            android:layout_toStartOf="@+id/button"
            android:inputType="text"
            android:text="Text 2" />

    </RelativeLayout>
```

Po spuštění v emulátoru nebo na zařízení bychom měli vidět formulář s identickým chováním jako v předchozím příkladu.



Obrázek 15.3: Aplikace s kontejnerem typu Relative Layout, spuštěná na emulátoru

15.2 Grid Layout

Grid Layout je druhým typem formuláře, který je v designéru Android Studia zařazený pod záložku Legacy. Platí doporučení nahradit jej alternativou, nejčastěji pomocí formuláře Constraint Layout.

Přestože jde o Legacy formulář, i zde existuje dost dobrý důvod proč ho znát. Zatímco v případě plnohodnotných aplikací se bez něj dá obejít, při programování widgetů, ke kterým se dostaneme v kapitole 30 Widgety, není k dispozici Constraint Layout, ani jiný vhodný typ formuláře, který by ho mohl bezproblémově zastoupit. Je to ale zřejmě jediná výjimka, ve všech ostatních případech je lepší volit alternativu.

Jak tedy Grid Layout funguje? Zdánlivě se podobá kontejneru Table Layout, ale zatímco ten umožnil pracovat s řádky tabulky, Grid Layout dává možnosti pracovat s objekty v plnohodnotné mřížce. Prvky je možné v kontejneru Grid Layout umísťovat do buněk pomocí souřadnic a hlavně je zde možnost buňky slučovat. Jak horizontálně, tak vertikálně. Podobně jako při práci v tabulkovém procesoru.

Na druhou stranu je nutné přiznat, že se s Grid Layoutem nepracuje úplně komfortně a zejména nastavování relativní velikosti buněk je někdy na dlouhé zkoušení a ladění. Přesto se můžou objevit formuláře, pro které je právě tento typ rozvržení tím nejvhodnějším.

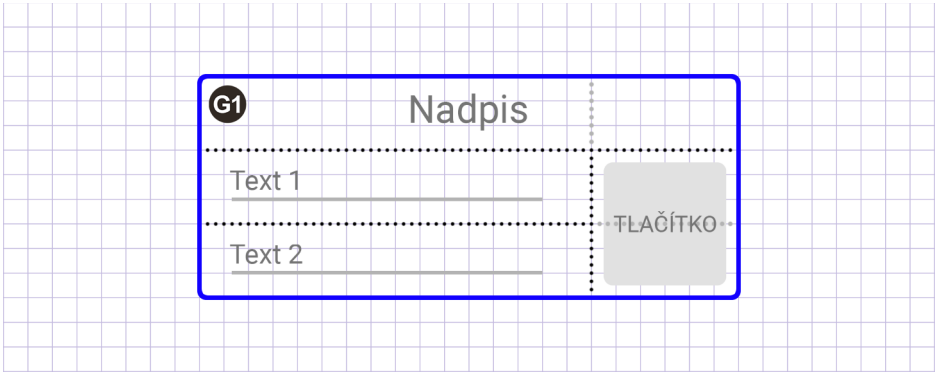
Grid Layout má dva režimy umísťování prvků. První je podle souřadnic. V takovém případě není potřeba znát velikost tabulky, její šířka je daná objektem, který je umístěný nejvíc vpravo a výška objektem který je nejníže. Vyplněné jsou pouze buňky, do kterých umístíme nějaký prvek, zbývající zůstávají prázdné.

Proti tomu druhý režim nepoužívá souřadnice. Je daný rozměr a počet buněk tabulky. Elementy se vkládají postupně do všech buněk, počínaje buňkou vlevo nahoře.

15.2.1 Grid Layout, 1. příklad

Nejprve si zkusíme v Grid Layoutu udělat známý příklad s textovými poli a tlačítkem, při kterém použijeme metodu vkládání prvků pomocí souřadnic:

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 15_Legacy
projekt GridLayout1



Obrázek 15.4: Návrh formuláře typu Grid Layout

Mřížka v kontejneru GridLayout je tvořena třemi řádky a dvěma sloupci. Další kroky jsou snadno předvídatelné – každý element umístíme do příslušných souřadnic mřížky, přičemž levá horní buňka má souřadnice 0, 0.

Textové pole Nadpis roztáhneme přes dvě buňky vodorovně a tlačítko naopak přes dvě buňky svisle. Potom už zbývá pouze zařídit, aby při změně velikosti displeje měnil svoji šířku první sloupec, zatímco druhý potřebujeme zachovat konstantní. Buňky mřížky nemají v XML svoje značky, vše se řídí parametry jednotlivých prvků.

Způsob, jak v XML zapsat potřebné parametry prvků:

`android:layout_column="0"` - index sloupce, do kterého má být element umístěný

`android:layout_row="0"` - index řádky, do které má být element umístěný

`android:layout_columnSpan="2"` - prvek se rozprostře přes dvě buňky vedle sebe (sloučí je v jednu)

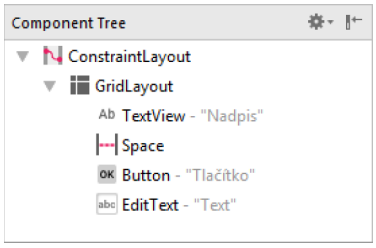
`android:layout_rowSpan="2"` - prvek se rozprostře přes dvě buňky pod sebou (sloučí je v jednu)

`android:layout_gravity="fill_horizontal"` – buňka vyplní vodorovně maximum prostoru v mřížce

`android:layout_gravity="fill_vertical"` – buňka vyplní svisle maximum prostoru v mřížce (v příkladu nevyužito)

`android:layout_gravity="fill"` – buňka vyplní v obou osách maximum prostoru v mřížce (v příkladu nevyužito)

Hierarchická struktura objektů v okně Component Tree, obrázek 15.5, je tentokrát velmi jednoduchá. Všechny prvky jsou přímo v komponentě GridLayout.



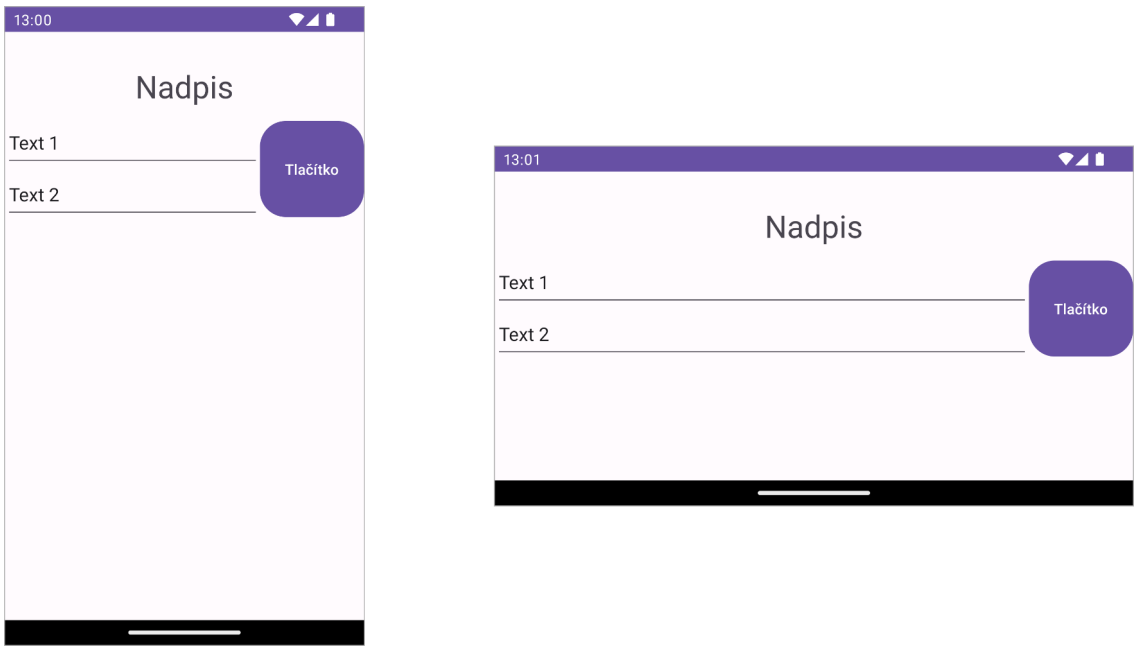
Obrázek 15.5: Prvky formuláře v okně Component Tree

Výsledný XML kód:

```
<!-- mrizkovy layout G1, sirka max, vyska 150dp -->
<GridLayout
    android:layout_width="0dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_height="150dp">

    <!-- textove pole, sirka max, vyska 50dp
         pozice sloupec 0, radek 0, roztazeny pres dva sloupce
         text Nadpis, vycentrovany, velikost textu 30sp -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:layout_column="0"
        android:layout_row="0"
        android:layout_columnSpan="2"
        android:text="Nadpis"
        android:textAlignment="center"
        android:textSize="30sp" />

    <!-- prvni pole pro zadani textu, typ text
         sirka max, vyska 50dp
         pozice sloupec 0, radek 1, vodorovne vyplnit bunku
         prednastaveny obsah 'Text 1' -->
    <EditText
        android:inputType="text"
        android:layout_width="wrap_content"
        android:layout_height="50dp"
        android:layout_column="0"
        android:layout_row="1"
        android:layout_gravity="fill_horizontal"
        android:text="Text 1" />

    <!-- druhe pole pro zadani textu, typ text
         sirka max, vyska 50dp
         pozice sloupec 0, radek 2, vodorovne vyplnit bunku
         prednastaveny obsah 'Text 2' -->
    <EditText
        android:inputType="text"
        android:layout_width="wrap_content"
```

```
        android:layout_height="50dp"
        android:layout_column="0"
        android:layout_row="2"
        android:layout_gravity="fill_horizontal"
        android:text="Text 2" />

    <!-- tlacitko 100x100dp
         pozice sloupec 1, radek 1, roztazeny pres dva radky
         text 'Tlacitko' -->
    <Button
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_column="1"
        android:layout_row="1"
        android:layout_rowSpan="2"
        android:text="Tlačítko" />

</GridLayout>
```

Po spuštění na emulátoru nebo zařízení bychom měli vidět formulář s identickým chováním jako v předchozích příkladech.

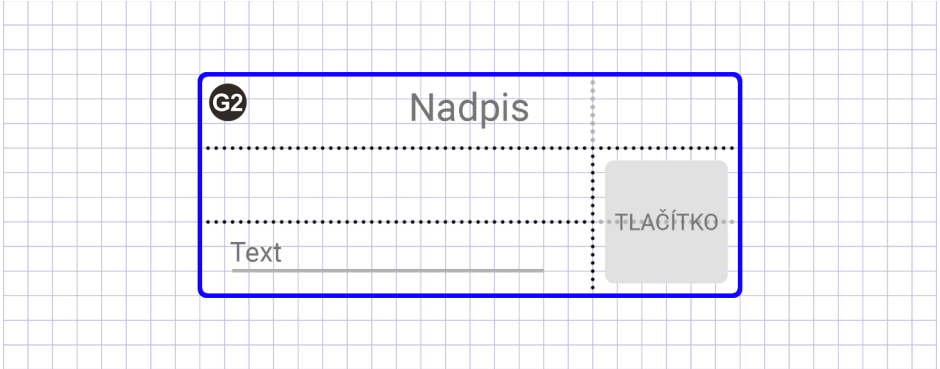


Obrázek 15.6: Aplikace s kontejnerem typu Grid Layout, spuštěná na emulátoru

15.2.2 Grid Layout, 2. příklad

Druhou možností jak vyplnit mřížku je postupné vkládání prvků do všech jejích buněk. Příklad si trochu upravíme, místo prvního textového pole necháme prázdnou buňku. Je to tak proto, abychom si mohli ukázat jak nechat ve formuláři prázdné místo přesto, že musíme vložit prvky do všech buněk.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 15_Legacy
projekt GridLayout2



Obrázek 15.7: Návrh formuláře typu Grid Layout

Mřížka v kontejneru GridLayout je tvořena třemi řádky a dvěma sloupci. Na rozdíl od předchozího příkladu je zde potřeba nastavit alespoň šířku tabulky, aby bylo jasné po kolika buňkách chceme řádek zalomit a pokračovat na dalším. Dá se zadat i počet řádků, není to ale bezpodmínečně nutné, výška tabulky bude jasná podle počtu vložených prvků.

Jako první vložíme textové pole Nadpis a roztáhneme ho přes dvě buňky vodorovně. Tím budou první dvě buňky zaplněné a můžeme pokračovat dalším elementem, který bude na druhé řádce. Tam je ale mezera. Jelikož není možné buňku přeskočit, vyřešíme to vložením prvku Space, který neudělá nic víc, než že vyplní prázdné místo v buňce, aniž by na displeji cokoliv zobrazil.

Jelikož zde, na rozdíl od předchozího příkladu, záleží na pořadí vkládání prvků, musíme nyní vložit tlačítko. Roztáhneme ho přes dva řádky, čímž vyplníme dvě buňky nad sebou. Jako poslední vložíme textové pole.

Způsob, jak v XML zapsat potřebné parametry kontejneru:

`android:columnCount="2"` – počet sloupců mřížky

`android:rowCount="3"` - počet řádků mřížky (v příkladu nevyužito)

Způsob, jak v XML zapsat potřebné parametry prvků:

`android:layout_columnSpan="2"` - prvek se rozprostře přes dvě buňky vedle sebe (sloučí je v jednu)

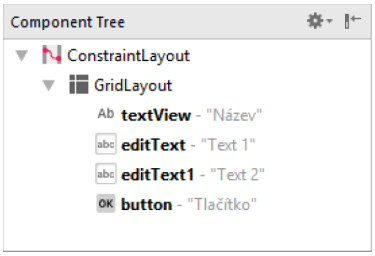
`android:layout_rowSpan="2"` - prvek se rozprostře přes dvě buňky pod sebou (sloučí je v jednu)

`android:layout_gravity="fill_horizontal"` – buňka vyplní vodorovně maximum prostoru v mřížce

`android:layout_gravity="fill_vertical"` – buňka vyplní svisle maximum prostoru v mřížce (v příkladu nevyužito)

`android:layout_gravity="fill"` – buňka vyplní v obou osách maximum prostoru v mřížce (v příkladu nevyužito)

Hierarchická struktura objektů v okně Component Tree, obrázek 15.8, je podobná jako v předchozím příkladu, opět jsou všechny prvky přímo v komponentě GridLayout. Liší se pouze tím, že nyní záleží na pořadí, ve kterém jsou uvedeny.

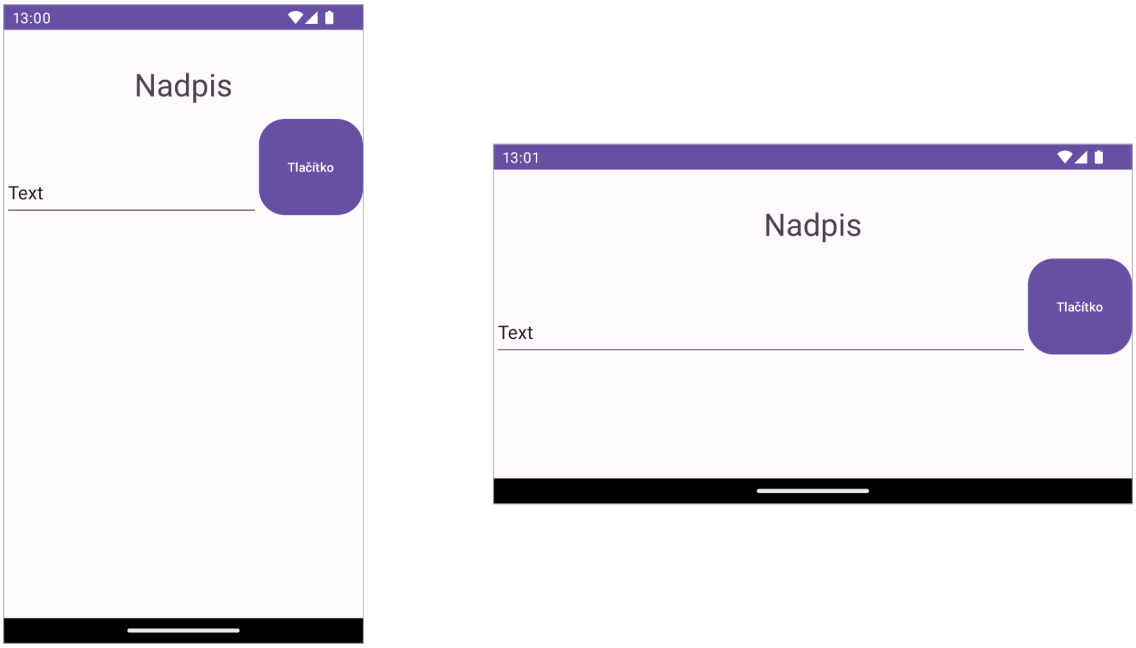


Obrázek 15.8: Prvky formuláře v okně Component Tree

Výsledný XML kód:

```
<!-- mrizkovy layout G2, sirka max, vyska 150dp -->
<GridLayout
    android:layout_width="0dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_height="150dp"
    android:columnCount="2">

    <!-- textove pole, sirka max, vyska 50dp
         roztazeny pres dva sloupce
         text Nadpis, vycentrovany, velikost textu 30sp -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:layout_columnSpan="2"
        android:text="Nadpis"
        android:textAlignment="center"
        android:textSize="30sp" />

    <!-- mezera, sirka max, vyska 50dp
         vodorovne vyplnit bunku -->
    <Space
        android:layout_width="wrap_content"
        android:layout_height="50dp"
        android:layout_gravity="fill_horizontal" />

    <!-- tlacitko 100x100dp, roztazene pres dva radky
         text 'Tlacitko' -->
    <Button
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_rowSpan="2"
        android:text="Tlačítko" />

    <!-- druhe pole pro zadani textu, typ text
         sirka max, vyska 50dp, vodorovne vyplnit bunku
         prednastaveny obsah 'Text' -->
    <EditText
        android:inputType="text"
        android:layout_width="wrap_content"
        android:layout_height="50dp"
        android:layout_gravity="fill_horizontal"
        android:text="Text" />

</GridLayout>
```

Opět se na výsledek podíváme v emulátoru nebo na koncovém zařízení. Vidět bychom měli totéž jako na obrázku 15.9.



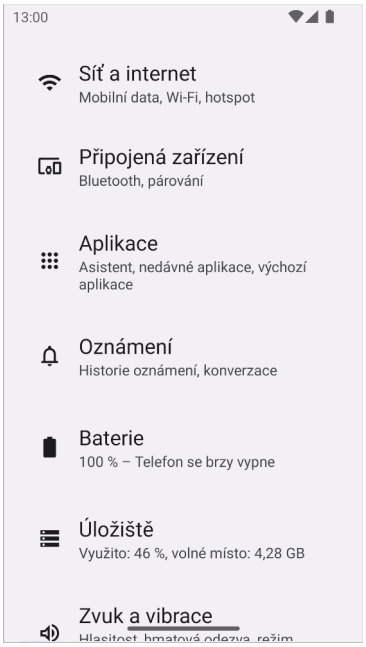
Obrázek 15.9: Aplikace s kontejnerem typu Grid Layout, spuštěná na emulátoru

15.3 List View

Zatímco předchozí kontejner měl jednoznačné a velmi omezené využití, kontejner List View je jedním z nejuniverzálnějších. Jeho primárním účelem je seskupit a zobrazit seznam položek, umožnit jejich vertikální rolování a výběr. Položky můžou být úplně jednoduché, typicky textový seznam, ale i velmi komplikované, složené z obrázků, textů, tlačítek a jiných objektů. Seznam je přes virtuální adaptér napojený na zdroj dat, kterým bývá nejčastěji pole nebo databáze.

Seznamů typu ListView obsahuje systém Android nepřeberné množství. V telefonu najdeme seznam kontaktů, historie hovorů nebo textových zpráv, záložky v internetovém prohlížeči, skladby v přehrávači, fotografie v galerii a tak dále.

Typickým příkladem je seznam položek z menu Nastavení na obrázku 15.10. Seznam List View seskupí položky, z nichž každá je sestavená z ikony a dvou různých textových polí. Položek je víc, než kolik se na vejde na displej, takže jimi může uživatel rolovat, poté si zvolit tu, která ho zajímá a ťuknutím přejít na další aktivitu.

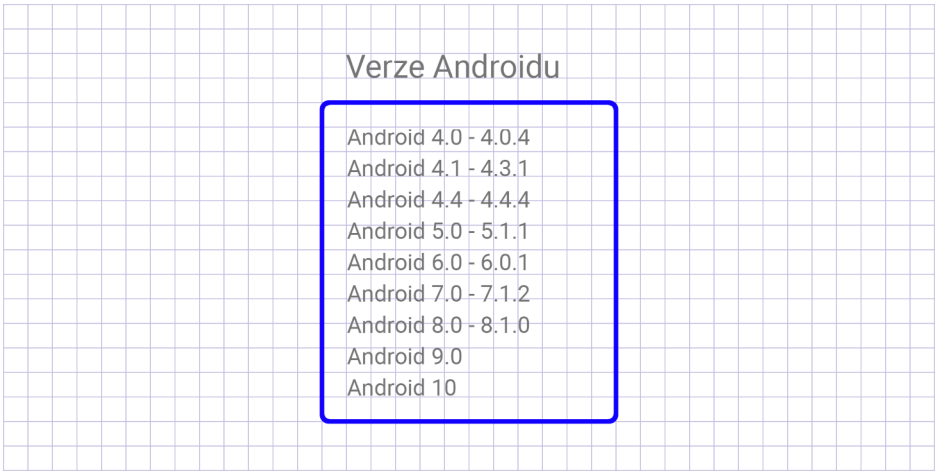


Obrázek 15.10: Příklad aplikace s kontejnerem typu List View

15.3.1 List View, 1. příklad

Vyzkoušíme si udělat jednodušší z obou variant, prostý seznam položek, definovaných v poli. Vzhled položek nebudeme nijak definovat, pouze předáme kontejneru List View seznam hodnot, které má zobrazit.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 15_Legacy
projekt ListView1



Obrázek 15.11: Návrh formuláře s kontejnerem List View

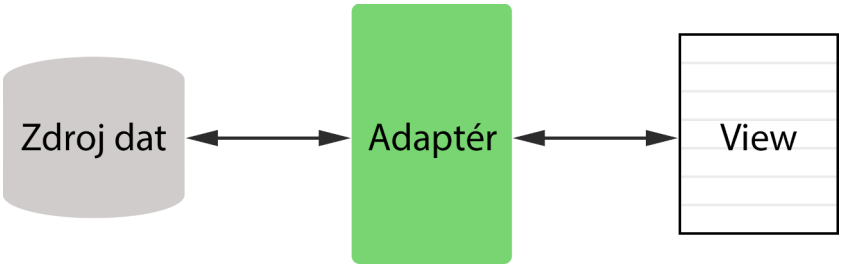
Do formuláře (v tomto příkladu používejme Constraint Layout) vložíme textové pole a kontejner List View. Rozměry nastavíme tak, aby byla vyplněna celá plocha displeje a aby se při natočení displeje velikost kontejneru přizpůsobila (viz kapitola 13.2 Constraint Layout). Kontejneru ještě ve vlastnostech nastavíme id listViewVerzeAndroidu a to je vše.

XML kód kontejneru bude tedy vypadat nějak takto:

```
<!-- ListView listViewVerzeAndroidu
     sirka a vyska max, okraje 8dp,
     navazana zezhoda na textove pole,
     z ostatnich stran na parent kontejner -->
<ListView
    android:id="@+id/listViewVerzeAndroidu"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_marginBottom="8dp"
    android:layout_marginLeft="8dp"
    android:layout_marginRight="8dp"
    android:layout_marginTop="8dp"
    app:layout_constraintTop_toBottomOf="@+id/textView"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent" />
```

Významnější část příkladu se odehraje ve třídě MainActivity.java. Právě tam totiž vytvoříme pole, které bude zdrojem dat a nadefinujeme adaptér, který propojí toto pole s kontejnerem ListView.

Funkce adaptéru je znázorněna na následujícím obrázku:



Obrázek 15.12: Propojení zdroje dat a kontejneru List View pomocí adaptéru

Komponenta ListView sice umí skvěle data zobrazovat, nedokáže si je však žádným rozumným způsobem zajistit. Naštěstí se ale dá napojit na adaptér, který naopak nic nezobrazí, ale zato má schopnost data získávat z rozličných zdrojů. V našem případě z pole, které si sami nadefinujeme. Přesněji řečeno, pro každý typ zdroje existuje samostatný adaptér, odvozený od třídy BaseAdapter, který se už o dodání dat postará. Adaptér, který umí pracovat s polem, se jmenuje ArrayAdapter.

Co je potřeba udělat?

- vytvořit pole hodnot

- vytvořit adaptér

- propojit adaptér s kontejnerem ListView

Pole vytvoříme poměrně jednoduše. Jelikož jde o pole textových řetězců, použijeme následující Java zápis, viz příklad pole o dvou řetězcích:

```
String [] poleRetezcu = { "retezec1", "retezec2" };
```

Následuje zřízení adaptéru. Abychom mohli adaptér použít, musíme ho nejdřív inicializovat, neboli vytvořit jeho instanci:

```
ListAdapter adapter = new ArrayAdapter<>(this,
      android.R.layout.simple_list_item_1, poleRetezcu);
```

Tato instrukce založí novou instanci třídy ArrayAdapter. Parametr this určuje, že se tak stane v kontextu třídy MainActivity. Druhý parametr stanovuje způsob zobrazení položky v ListView. Parametr simple_list_item_1 říká, že položky budou na výstupu ve formátu jednoduchého textového pole TextView, se kterým si ListView hravě poradí. K dispozici je dalších několik desítek způsobů zobrazení, Android Studio je při psaní kódu samo napoví. Posledním atributem je zdroj dat, neboli pole textových řetězců.

Vytvořený adaptér propojíme s kontejnerem ListView:

```
ListView listView = (ListView) findViewById(R.id.listViewVerzeAndroidu);
listView.setAdapter(adapter);
```

První řádek najde pole podle jeho id a přiřadí jej do proměnné listView. Druhý řádek už jen zavolá metodu kontejneru, která k němu adaptér přiřadí.

ListView získá svůj význam až v okamžiku, kdy můžeme pracovat s položkou, kterou si uživatel vybere a klikne na ni. To se realizuje implementací tzv. posluchače neboli listeneru. Listener je rutina, která čeká, až nastane stanovená událost, v našem případě kliknutí na položku seznamu a potom se provede sekvence příkazů uvedená ve složených závorkách.

```
AdapterView.OnItemClickListener clickListener = new
            AdapterView.OnItemClickListener() { .... }
```

Nejjednodušším způsobem jak událost obsloužit je předefinování původní metody kontejneru onItemClick() tak, aby provedla to, co potřebujeme.

```
@Override
public void onItemClick(AdapterView<?> parent, View view, int pozice, long id)
{ ... }
```

Jelikož je metoda onItemClick() univerzální a nemusí být nutně vázaná listenerem na jeden seznam, má v parametrech základní informace – který kontejner ji vyvolal, číslo položky, kterou si uživatel zvolil atd. Díky tomu dokážeme následně rozklíčovat úmysl uživatele. Snadno zjistíme název vybrané položky:

```
String textPolozky = String.valueOf(parent.getItemAtPosition(pozice));
```

Parent je objekt nadřazený položce, čili v našem případě kontejner ListView. Využijeme jeho metodu getItemAtPosition(), která vrací text položky na pozici uvedené v parametru. A pozice položky je předaná v parametru ‚pozice‘.

Dál už je na vývojáři, jak se získaným textem naloží. Android umožňuje snadno zobrazovat informace pomocí jednoduchých textů na displeji, takzvaných toast zpráv:

```
Toast.makeText(MainActivity.this, textPolozky, Toast.LENGTH_SHORT).show();
```

Parametry nejsou nijak složité – první je kontext ve kterém se má zpráva zobrazit, v našem případě aktivita MainActivity, následuje text zprávy a doba, po kterou má zůstat na displeji.

Takto vytvořený listener už jen zbývá přiřadit ke kontejneru (nebo ke kontejnerům) typu ListView pomocí metody setOnItemClickListener(), jejímž parametrem je zmíněný listener.

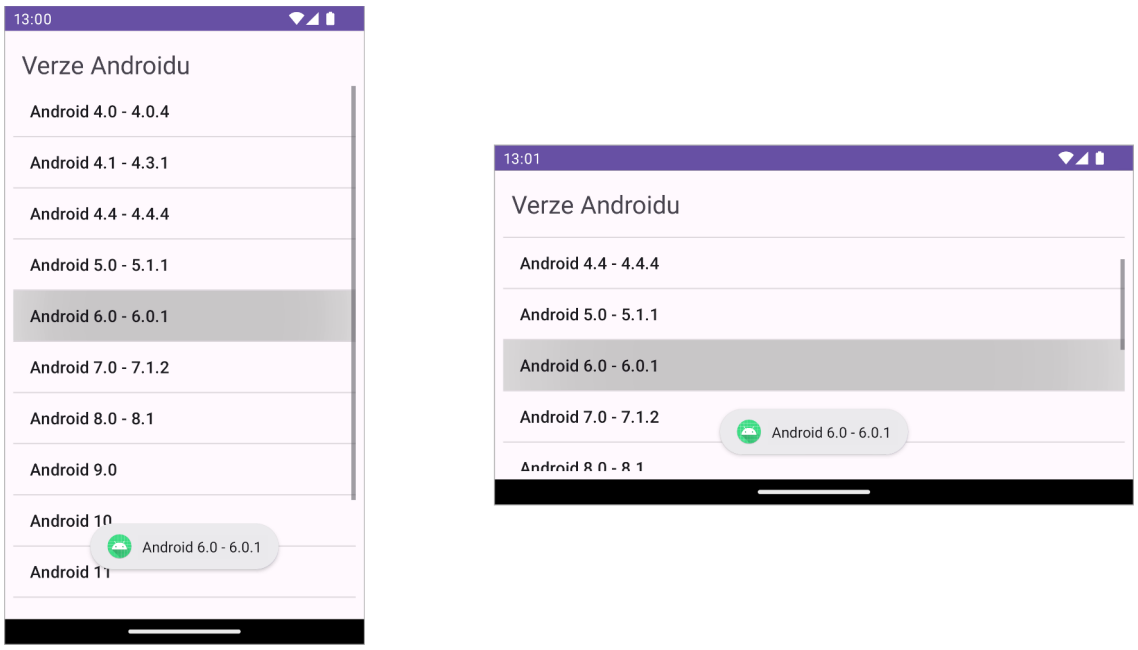
```
listView.setOnItemClickListener(clickListener);
```

Popsané příkazy upravíme pro naše potřeby, vložíme do vlastní metody a tu zavoláme z události onCreate() třídy MainActivity. Výsledný kód bude vypadat takto:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    naplnitListView();
}

// Metoda nastavi hodnoty do ListView listViewVerzeAndroidu
private void naplnitListView()
{
    String [] arrVerzeAndroidu = {
            "Android 4.0 - 4.0.4",
            "Android 4.1 - 4.3.1",
            "Android 4.4 - 4.4.4",
            "Android 5.0 - 5.1.1",
            "Android 6.0 - 6.0.1",
            "Android 7.0 - 7.1.2",
            "Android 8.0 - 8.1",
            ... };

    // vytvorit instanci adapteru
    ListAdapter adapter = new ArrayAdapter<>(this,
            android.R.layout.simple_list_item_1, arrVerzeAndroidu);
    // propojeni adapteru s kontejnerem ListView
    ListView listView = (ListView) findViewById(R.id.listViewVerzeAndroidu);
    listView.setAdapter(adapter);

    // vytvoreni instance posluchace udalosti kliknuti na polozku
    AdapterView.OnItemClickListener clickListener = new
            AdapterView.OnItemClickListener() {
        @Override
        // prepsani puvodni udalosti onItemClick() kontejneru
        public void onItemClick(AdapterView<?> parent, View view, int pozice,
                long id) {
            // zjistit text na vybrane polozce
            String textPolozky =
                  String.valueOf(parent.getItemAtPosition(pozice));
            // zobrazit text na displeji
            Toast.makeText(MainActivity.this, textPolozky,
                  Toast.LENGTH_SHORT).show();
        }
    };

    // nastaveni posluchace udalosti
```

```
    listView.setOnItemClickListener(clickListener);
}
```

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 15.13.



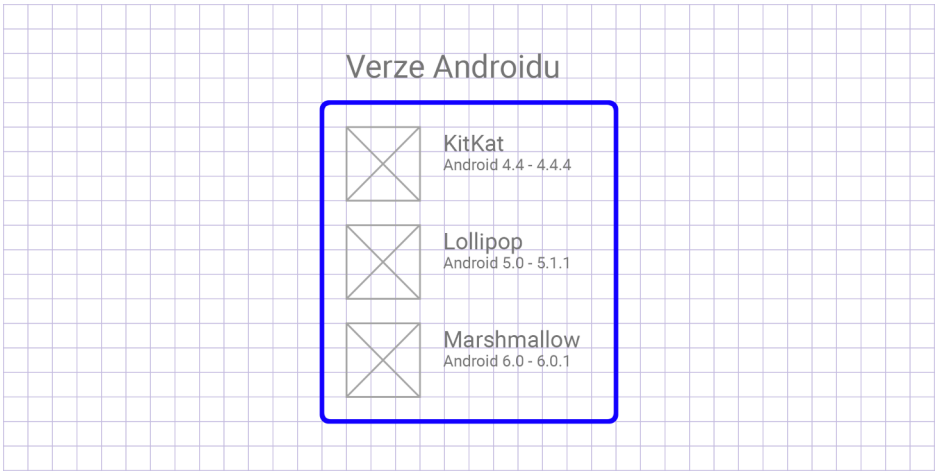
Obrázek 15.13: Aplikace s kontejnerem typu List View, spuštěná na emulátoru

15.3.2 List View, 2. příklad

S předchozí variantou ListView si vystačíme, pokud nepotřebujeme zobrazit víc než prostý text v každé řádce. Jakmile ale situace vyžaduje komplexnější položku, musíme sáhnout po vlastní definici pomocí formuláře a potažmo také po vlastním adaptéru.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 15_Legacy
projekt ListView2

Řekněme, že budeme chtít zobrazit seznam, který bude vypadat zhruba takto:

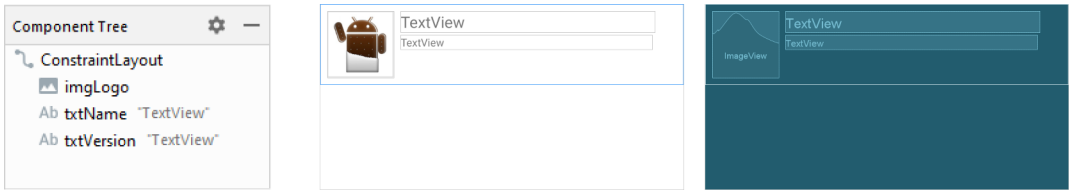


Obrázek 15.14: Návrh formuláře s kontejnerem List View

Co je potřeba udělat?

- vytvořit formulář položky
- napsat vlastní adaptér
- definovat pole hodnot, které bude obsahovat zobrazované údaje
- propojit adaptér s kontejnerem ListView

Design položky se vytvoří relativně snadno. V okně projektu ve složce res/layout/ založíme nový soubor (kontextové menu New → Layout resource file). Pojmenujeme ho třeba ‚polozka'. V návrhu se dá použít některý z kontejnerů z předchozích kapitol, například Constraint Layout. V něm je potřeba uspořádat objekty tak, aby odpovídaly představě vývojáře o vizuální podobě položky seznamu. Při navrhování vzhledu je nutné dávat obzvlášť pozor, aby bylo zobrazení formuláře efektivní, protože se na displeji odehraje tolikrát, kolik položek má seznam. Je velmi žádoucí, aby byl formulář co nejjednodušší a s minimem vnoření.



Obrázek 15.15: Struktura položky seznamu v okně Component Tree a na formuláři

Napsat vlastní adaptér, který transformuje zdrojová data do položky ve formátu View, bude složitější. Kvůli jednoduchosti i v tomto příkladu předpokládejme, že vstupem bude pole textových řetězců. Nejdřív založíme Java soubor (kontextové menu New → Java Class), s nějakým příhodným názvem, třeba ‚PolozkaAdapter'. Jako první je potřeba oddědit tuto novou třídu od třídy ArrayAdapter, ze které bude adaptér vycházet.

```
class PolozkaAdapter extends ArrayAdapter<String> { }
```

A hned nato založíme konstruktor nové třídy, který zavolá třídu nadřazenou. Tím se zajistí, že adaptér bude mít funkčnost adaptéru ArrayAdapter, kterou navíc rozšíříme.

```
public PolozkaAdapter(Context context, String[] androidVerze) {
    super(context, R.layout.polozka, androidVerze);
}
```

Poté předefinujeme metodu ArrayAdapteru getView(), která za normálních okolností vrací textovou položku k zobrazení. Upravíme ji tak, aby vracela položku podle našich představ, tj. tu, kterou jsme si nadefinovali v souboru /res/layout/polozka.xml.

```
@Override
public View getView(int pozice, View convertView, ViewGroup parent) { }
```

Tím jsme z nejhoršího venku, struktura adaptéru je vytvořena. Teď se musí ještě dát metodě getView() funkčnost, která zajistí správné vyplnění polí formuláře a sestavený formulář odešle návratovou hodnotou kontejneru ListView. Nejdřív transformujeme sestavený formulář do formátu View. Používá se k tomu takzvaný inflater:

```
convertView = LayoutInflater.from(getContext()).inflate(R.layout.polozka,
      parent, false);
```

V parametrech metody getView() dostáváme několik důležitých informací. Jednak číslo položky, kterou adaptér zrovna zpracovává a potom transformovaný pohled, pokud už existuje a jako třetí parametr je nadřazený objekt, v našem případě ListView. Zejména číslo položky je důležité, podle něj budeme v datech dohledávat, co se vlastně má zobrazit. K přístupu k datům z pole slouží metoda getItem(), které dáme jako parametr číslo položky. Toto číslo se shoduje s pozicí v poli vstupních dat, takže pro metodu getItem() není těžké hodnotu z pole vzít a vrátit ji. Uložíme ji do proměnné typu String, kterou nazveme strItem.

```
String strItem = getItem(pozice);
```

Dál už je jenom na fantazii tvůrce, jak formulář pomocí vstupních dat vyplnit. V našem příkladu budou v poli dat uloženy vždy dvojice informací – číslo a název verze Androidu, odděleny dvojtečkou. Zároveň jsou ve složce res/drawable/ uložena loga jednotlivých verzí Androidu, přičemž název loga odpovídá názvu verze Androidu. Takže ze strItem získáme název a číslo verze, přiřadíme je do polí ve formuláři, podle názvu dohledáme logo a to přiřadíme do kontejneru ImageView. Nakonec nesmíme zapomenout předat vytvořenou položku v návratové hodnotě.

Celý kód adaptéru bude vypadat takto:

```
class PolozkaAdapter extends ArrayAdapter<String> {
    // konstruktor adapteru
    public PolozkaAdapter(Context context, String[] androidVerze) {
        super(context, R.layout.polozka, androidVerze);
    }

    @Override
    // predefinovani metody getView tak, aby vracela nadefinovanou polozku
    public View getView(int pozice, View convertView, ViewGroup parent) {

        // prevedeni polozky na View
        convertView = LayoutInflater.from(getContext())
                        .inflate(R.layout.polozka, parent, false);

        // ziskani retezce z pole, podle pozice
```

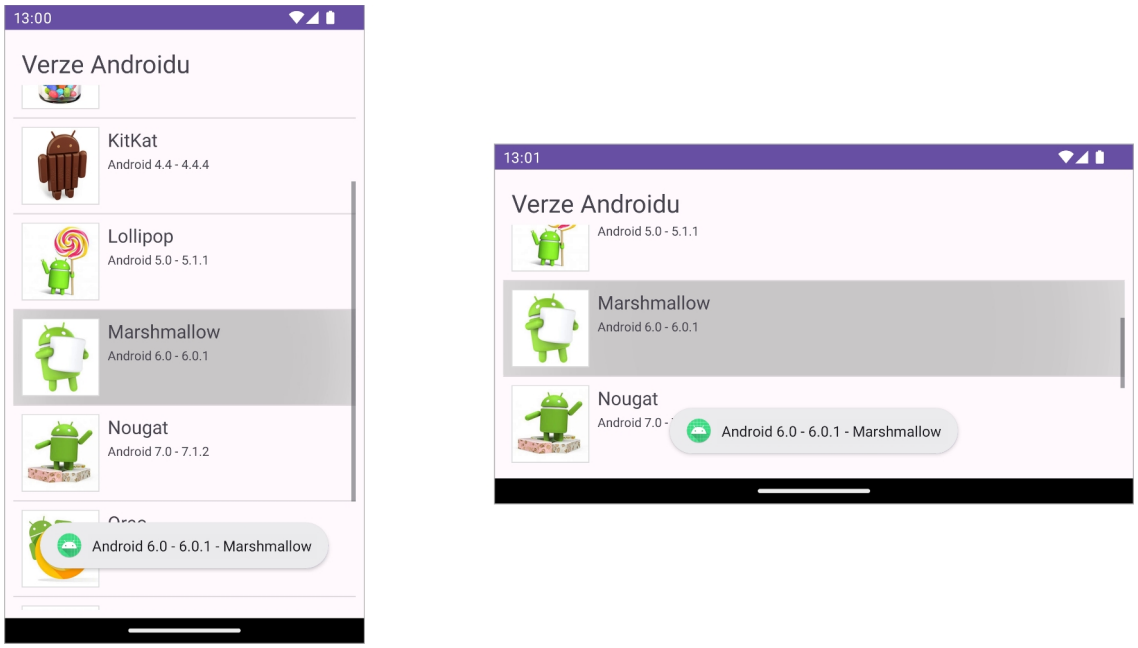
```
        String strItem = getItem(pozice);

        // rozdeleni retezce na casti
        String[] strVersion = strItem.split(":");

        // nalezeni jednotlivych elementu a prirazeni do promennych
        ImageView imgLogo = (ImageView) convertView.findViewById(R.id.imgLogo);
        TextView txtName = (TextView) convertView.findViewById(R.id.txtName);
        TextView txtVersion = (TextView) convertView
                                    .findViewById(R.id.txtVersion);
        // zpristupneni resources
        Resources res = getContext().getResources();

        // jmeno resource s logem androidu
        String resName = strVersion[1].replaceAll(" ", "").toLowerCase();

        // id resource s logem androidu
        int imgLogoId = res.getIdentifier(resName, "drawable", getContext()
                            .getPackageName());

        // prirazeni loga do ImageView
        imgLogo.setImageResource(imgLogoId);

        // prirazeni nazvu a verze do textovych poli
        txtName.setText(strVersion[1]);
        txtVersion.setText(strVersion[0]);

        // upravene View odesleme jako navratovou hodnotu
        return convertView;
    }
}
```

Tím je připravený formulář položky i adaptér. Ještě je potřeba zajistit vstupní data a vše ve třídě MainActivity propojit. Postup je podobný jako v předchozím příkladu. Ve třídě MainActivity v události onCreate() budeme zavolat vlastní metodu naplnitListView(), která vygeneruje vstupní data, vytvoří instanci nového adaptéru, napojí ho na kontejner ListView a připojí posluchač události, který bude čekat na kliknutí na položku, které následně zpracuje.

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        naplnitListView();
    }

    // Metoda nastavi hodnoty do ListView listViewVerzeAndroidu
    private void naplnitListView()
    {
        // vstupní data
        // vzdy dva texty oddelene dvojteckou, adapter si je rozdeli
        String [] arrVerzeAndroidu = {
                        "Android 4.0 - 4.0.4:Ice Cream Sandwich",
                        "Android 4.1 - 4.3.1:Jelly Bean",
                        "Android 4.4 - 4.4.4:KitKat",
                        "Android 5.0 - 5.1.1:Lollipop",
                        "Android 6.0 - 6.0.1:Marshmallow",
```

```
                    "Android 7.0 - 7.1.2:Nougat",
                    "Android 8.0 - 8.1:Oreo",
                    "Android 9.0:Pie",
                    "Android 10 a vyšší:Android 10"
            };

        // vytvorit instanci adapteru
        ListAdapter adapter = new PolozkaAdapter(this, arrVerzeAndroidu);

        // propojeni adapteru s kontejnerem ListView
        ListView listView = (ListView) findViewById(R.id.listViewVerzeAndroidu);
        listView.setAdapter(adapter);

        // vytvoreni instance posluchace udalosti kliknuti na polozku
        AdapterView.OnItemClickListener clickListener = new
                  AdapterView.OnItemClickListener() {
            @Override
            // prepsani puvodni udalosti onItemClick kontejneru
            public void onItemClick(AdapterView<?> parent, View view,
                                    int pozice, long id) {
                // zjisti text na vybrane polozce
                String textPolozky =String
                        .valueOf(parent.getItemAtPosition(pozice));
                // zobrazi text na displeji
                Toast.makeText(MainActivity.this, textPolozky
                        .replace(":", " - "), Toast.LENGTH_SHORT).show();
            }
        };

        // nastaveni posluchace udalosti
        listView.setOnItemClickListener(clickListener);
    }
}
```

Na koncovém zařízení nebo v emulátoru by měl výsledek vypadat jako na obrázku 15.16.



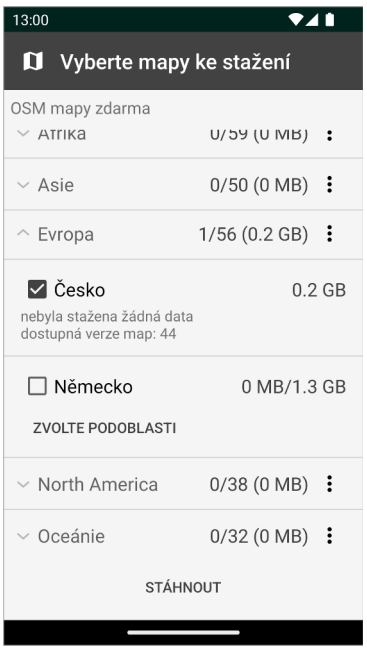
Obrázek 15.16: Aplikace s kontejnerem typu List View, spuštěná na emulátoru

15.4 Expandable List View

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 15_Legacy
projekt ExpandableListView

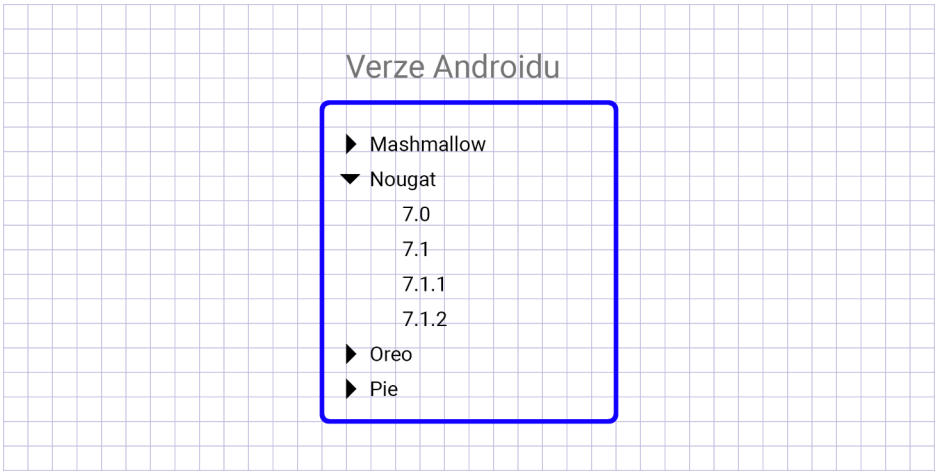
Tento kontejner je potomkem List View, patří mezi složitější a je určený k zobrazení dvojúrovňového vertikálně skrolovatelného seznamu. Položky jsou roztříděny do skupin, přičemž uživatel má možnost otevřít ťuknutím pouze ty skupiny, které ho zajímají. Stejným způsobem může ostatní skupiny zavřít. Navzdory svým možnostem není příliš často využívaný v nových operačních systémech Android.

Pracuje s ním však nespočet aplikací, které využívají širokých možností úprav a přizpůsobení. Expandable List View umožňuje nahradit standardní vzhled skupiny i jednotlivých položek formulářem, takže můžou obsahovat prakticky jakékoliv další vnořené komponenty. Nastavit lze oddělovače skupin, separátory mezi položkami a tak dále.



Obrázek 15.17: Příklad aplikace s kontejnerem typu Expandable List View

Vytvoříme jednoduchou verzi kontejneru Expandable List View, která bude tvořena seznamem verzí Androidu, které po otevření zobrazí seznam položek, z nichž každá bude zastupovat jednu aktualizaci. Položky budou jednoduché, budou obsahovat pouze textové pole pro číselné označení verze, bez dalších efektních komponent.



Obrázek 15.18: Návrh formuláře s kontejnerem Expandable List View

Co je potřeba udělat?

- vytvořit formulář s kontejnerem Expandable List View
- vytvořit formulář skupiny
- vytvořit formulář položky
- vytvořit pole hodnot, která budou sloužit jako zdroj dat
- vytvořit vlastní adaptér
- propojit adaptér s kontejnerem Expandable List View

Otevřeme soubor activity_main.xml a do formuláře (v příkladu použijeme Constraint Layout) vložíme textové pole a kontejner Expandable List View. Rozměry nastavíme tak, aby byla vyplněna celá plocha displeje a aby se při natočení displeje velikost kontejneru přizpůsobila (viz kapitola Constraint Layout). Kontejneru v okně vlastností nastavíme id expListViewNastaveniAndroidu.

XML kód kontejneru bude tedy vypadat zhruba takto:

```
<!-- ExpandableListView expListViewNastaveniAndroidu
     sirka a vyska max, okraje 8dp,
     navazana ze shora na textove pole,
     z ostatnich stran na parent kontejner -->
<ExpandableListView
     android:id="@+id/expListViewNastaveniAndroidu"
     android:layout_width="0dp"
     android:layout_height="0dp"
     android:layout_margin="8dp"
     android:clipToPadding="false"
     app:layout_constraintBottom_toBottomOf="parent"
     app:layout_constraintLeft_toLeftOf="parent"
     app:layout_constraintRight_toRightOf="parent"
     app:layout_constraintTop_toBottomOf="@+id/textView" />
```

Design skupiny se vytvoří tak, že v okně projektu, ve složce res/layout/, založíme nový soubor (kontextové menu New → Layout resource file). Pojmenujeme ho třeba ‚skupina'. Při návrhu použijeme některý z kontejnerů, které už známe, například Linear Layout. V něm uspořádáme objekty tak, aby odpovídaly naší představě o vizuální podobě skupiny. I tentokrát je potřeba, při navrhování vzhledu, dávat pozor na to, aby bylo zobrazení formuláře efektivní, protože se na displeji odehraje tolikrát, kolik budeme mít skupin.

Podobně vytvoříme i položku. V okně projektu ve složce res/layout/ založíme stejným způsobem další soubor. Pojmenujeme ho třeba ‚polozka'. Opět použijeme Linear Layout, nebo jiný kontejner, podle vlastního uvážení. V něm uspořádáme objekty tak, aby odpovídaly naší představě o vizuální podobě položky seznamu. A zase dáme pozor na efektivitu výsledného návrhu. U položky bývá často více komponent, složitější vnořování by se mohlo projevit na rychlosti vykreslování.

Ve třídě MainActivity.java vytvoříme pole, které bude zdrojem dat. U Expandable List View je to o něco složitější než u předchozích kontejnerů, protože jsou zde dvě úrovně – skupiny a položky. Nadefinujeme jedno pole se seznamem skupin a druhé se seznamem položek.

Pole skupin je obyčejné jednorozměrné pole textových řetězců:

```
// list se seznamem skupin
ArrayList<String> skupiny = new ArrayList<String>();
public void setGroupData() {
    skupiny.add("Ice Cream Sandwich");
    skupiny.add("Jelly Bean");
    skupiny.add("KitKat");
    skupiny.add("Lollipop");
    skupiny.add("Marshmallow");
    skupiny.add("Nougat");
    skupiny.add("Oreo");
    skupiny.add("Pie");
    skupiny.add("10");
```

```
    ...
}
```

Naproti tomu jsou položky reprezentované dynamickým polem objektů, kde každý objekt je tvořený dalším polem, tentokrát již textových řetězců. Každý objekt tedy zastupuje skupinu a každé pole textových řetězců je seznamem položek.

```
// list se seznamem objektu, kde kazdym objektem je seznam polozek
ArrayList<Object> polozky = new ArrayList<Object>();
public void setChildGroupData() {

    // Ice Cream Sandwich
    ArrayList<String> polozka = new ArrayList<String>();
    polozka.add("4.0");
    polozka.add("4.0.1");
    polozka.add("4.0.2");
    polozky.add(polozka);

    // Jelly Bean
    polozka = new ArrayList<String>();
    polozka.add("4.1");
    polozka.add("4.1.1");
    polozka.add("4.1.2");
    polozka.add("4.2.1");

    ......

}
```

Vytvořit vlastní adaptér, který transformuje zdrojová data do položky ve formátu View bude náročnější. Nejdřív založíme Java soubor (kontextové menu New → Java Class). Dáme mu příhodný název, třeba ‚PolozkaAdapter‘. Jako první je potřeba oddědit novou třídu od BaseExpandableListAdapter, ze které bude adaptér vycházet.

Zde je potřeba poznamenat, že existuje více typů adaptérů, ze kterých by se dalo vyjít, postavených na třídě ExpandableListAdapter. Základní BaseExpandableListAdapter se hodí pro většinu běžných datových zdrojů, jeho potomek SimpleExpandableListAdapter se dá využít pro data uložená v XML struktuře. Dalším možným adaptérem je CursorTreeAdapter, který je výhodné použít, pokud data poskytují dva databázové kurzory, každý pro jednu úroveň zanoření. A i tento adaptér má další specializované potomky. Je tedy z čeho vybírat, pro naše účely se však spokojíme s vcelku univerzálním a přitom ne moc komplikovaným BaseExpandableListAdapter.

```
class PolozkaAdapter extends BaseExpandableListAdapter { }
```

Hned nato založíme konstruktor nové třídy a lokální proměnné, do kterých budeme ukládat kontext a data.

```
private Context context;
private ArrayList<String> skupiny, aktualniPolozka;
private ArrayList<Object> polozky = new ArrayList<Object>();

public PolozkaAdapter(Context context, ArrayList<String> groupList,
            ArrayList<Object> itemList) {
    this.context = context;
    skupiny = groupList;
```

```
     polozky = itemList;
}
```

Následně je potřeba předefinovat metody, které vrací vyplněný formulář jednak položky seznamu a potom i rozbalovací skupiny. K získání layoutu položky je určena metoda getChildView(), která za normálních okolností vrací textovou položku k zobrazení. Upravíme ji, aby vracela položku podle našich představ, tj. tak, jak je navržena a definována v souboru formuláře /res/layout/polozka.xml.

```
@Override
public View getChildView(int groupPosition, int childPosition,
            boolean isLastChild, View customView, ViewGroup parent) {
    aktualniPolozka = (ArrayList<String>) polozky.get(groupPosition);

    // pokud dosud nebyl formular serializovan, serializovat
    if (customView == null) {
        LayoutInflater inflater = (LayoutInflater)
            context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        customView = inflater.inflate(R.layout.polozka, null);
    }

    // ziskat ze seznamu nazev
    String itemText = aktualniPolozka.get(childPosition);

    // nazev zapsat do textoveho pole
    TextView textField = (TextView) customView.findViewById(R.id.textView1);
    textField.setText(itemText);

    // vratit vytvorenou polozku
    return customView;
}
```

Ve své podstatě jde o to, že se pomocí inflateru serializuje formulář, naplní daty a předá v návratové hodnotě typu View. Podobně je potřeba předefinovat i záhlaví skupiny, definované v souboru /res/layout/polozka.xml.

```
@Override
public View getGroupView(int groupPosition, boolean isExpanded,
            View convertView, ViewGroup parent) {

    // pokud dosud nebyl formular serializovan, serializovat
    if (convertView == null) {
        LayoutInflater inflater = (LayoutInflater)
            context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        convertView = inflater.inflate(R.layout.skupina, null);
    }

    // ziskat ze seznamu nazev skupiny
    String textNazvuSkupiny = skupiny.get(groupPosition);

    // nazev zapsat do textoveho pole
    TextView nazevSkupiny = (TextView)convertView.findViewById(R.id.textView1);
    nazevSkupiny.setText(textNazvuSkupiny);

    // vratit vytvorene zahlavi skupiny
    return convertView;
}
```

V adaptéru je nutné předefinovat některé metody, které předají kontejneru nezbytné informace o datech. Bude potřebovat znát počet položek ve skupině. Ty mu poskytuje metoda getChildrenCount(). Stačí ji upravit tak, aby vracela počet položek pole, které odpovídá zvolené skupině.

```
@Override
public int getChildrenCount(int groupPosition) {
    return ((ArrayList<String>) polozky.get(groupPosition)).size();
}
```

A podobně počet skupin, což je informace, kterou získává pomocí metody getGroupCount(). Tady se dá záležitost vyřešit úpravou, která vrátí počet položek seznamu skupin.

```
@Override
public int getGroupCount() {
        return skupiny.size();
}
```

Celý kód adaptéru bude vypadat takto:

```
public class PolozkaAdapter extends BaseExpandableListAdapter {

    private Context context;
    private ArrayList<String> skupiny, aktualniPolozka;
    private ArrayList<Object> polozky = new ArrayList<Object>();

    public PolozkaAdapter(Context context, ArrayList<String> groupList,
ArrayList<Object> itemList) {
        this.context = context;
        skupiny = groupList;
        polozky = itemList;
    }

    @Override
    public Object getChild(int groupPosition, int childPosition) {
        return null;
    }

    @Override
    public long getChildId(int groupPosition, int childPosition) {
        return childPosition;
    }

    @Override
    public View getChildView(int groupPosition, int childPosition, boolean
isLastChild, View customView, ViewGroup parent) {
        aktualniPolozka = (ArrayList<String>) polozky.get(groupPosition);

        if (customView == null) {
            LayoutInflater inflater = (LayoutInflater)
context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
            customView = inflater.inflate(R.layout.polozka, null);
        }

        // ziskat ze seznamu nazev
        String itemText = aktualniPolozka.get(childPosition);
```

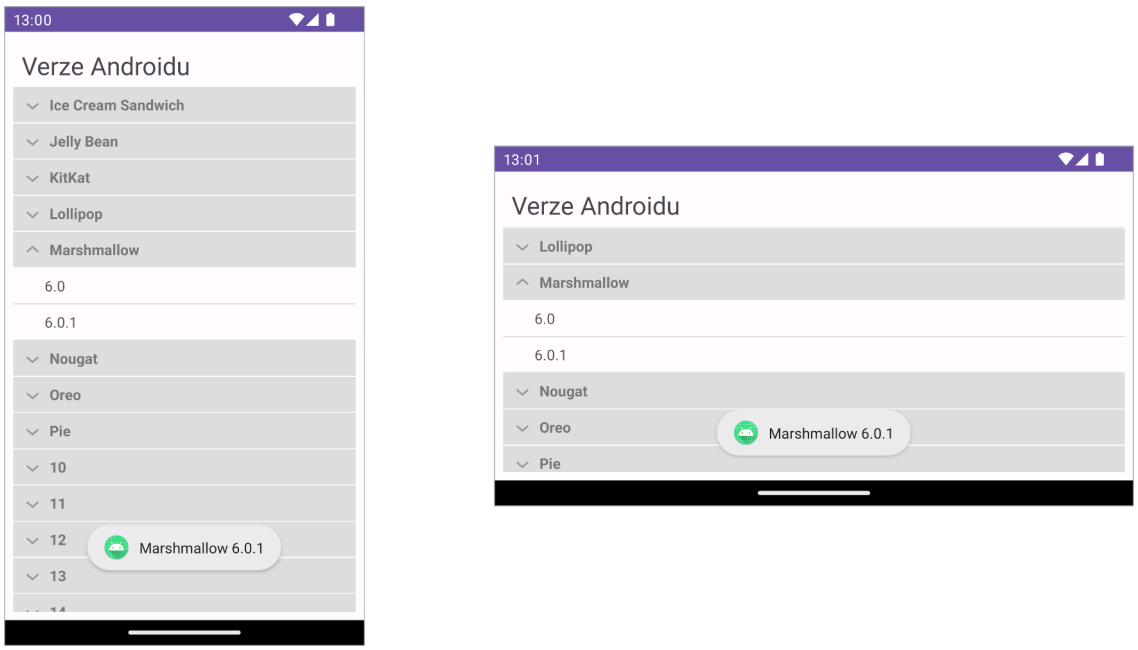
```
        // nazev zapsat do textoveho pole
        TextView textField = (TextView) customView.findViewById(R.id.textView1);
        textField.setText(itemText);

        // vratit vytvorenou polozku
        return customView;
    }

    @Override
    public int getChildrenCount(int groupPosition) {
        return ((ArrayList<String>) polozky.get(groupPosition)).size();
    }

    @Override
    public Object getGroup(int groupPosition) {
        return null;
    }

    @Override
    public int getGroupCount() {
        return skupiny.size();
    }

    @Override
    public long getGroupId(int groupPosition) {
        return 0;
    }

    @Override
    public View getGroupView(int groupPosition, boolean isExpanded, View
convertView, ViewGroup parent) {
        if (convertView == null) {
            LayoutInflater inflater =
(LayoutInflater)context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
            convertView = inflater.inflate(R.layout.skupina, null);
        }
        // ziskat ze seznamu nazev skupiny
        String textNazvuSkupiny = skupiny.get(groupPosition);

        // nazev zapsat do textoveho pole
        TextView nazevSkupiny =
(TextView)convertView.findViewById(R.id.textView1);
        nazevSkupiny.setText(textNazvuSkupiny);

        // vratit vytvorenou polozku
        return convertView;
    }

    @Override
    public boolean hasStableIds() {
        return true;
    }

    @Override
    public boolean isChildSelectable(int groupPosition, int childPosition) {
       return true;
    }
}
```

Tím je připravený formulář skupiny, formulář položky, data i adaptér. Ve třídě MainActivity toto všechno propojíme. V události onCreate() zavoláme naše vlastní metody pro načtení dat setGroupData() a setChildGroupData(). Následně založíme instanci adaptéru, tato data jí předáme a adaptér napojíme na kontejner expListViewNastaveniAndroidu z formuláře activity_main.xml.

Programový kód, bez zdroje dat (ten jsme si popsali výše):

```
private ExpandableListView expandbleList;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // nacist data do komponenty ExpandableListView
    setGroupData();
    setChildGroupData();

    // ziskat odkaz na ExpandableListView
    expandbleList =
        (ExpandableListView)findViewById(R.id.expListViewNastaveniAndroidu);
    // zrusit oddelovac polozek, pouzijeme vlastni
    expandbleList.setDividerHeight(0);

    // vytvorit adapter a predat mu vstupni data
    PolozkaAdapter mNewAdapter = new PolozkaAdapter(MainActivity.this,
            skupiny, polozky);
    expandbleList.setAdapter(mNewAdapter);

    // listener kliknuti na polozku
    expandbleList.setOnChildClickListener(new
            ExpandableListView.OnChildClickListener() {
        @Override
        public boolean onChildClick(ExpandableListView parent, View v,
                int groupPosition, int childPosition, long id) {
            ArrayList<String> aktualniPolozka = (ArrayList<String>)
                    polozky.get(groupPosition);

            Toast.makeText(getBaseContext(), skupiny.get(groupPosition) + " "
                    + aktualniPolozka.get(childPosition),
                    Toast.LENGTH_SHORT).show();

            return false;
        }
    });
}
```

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 15.19.

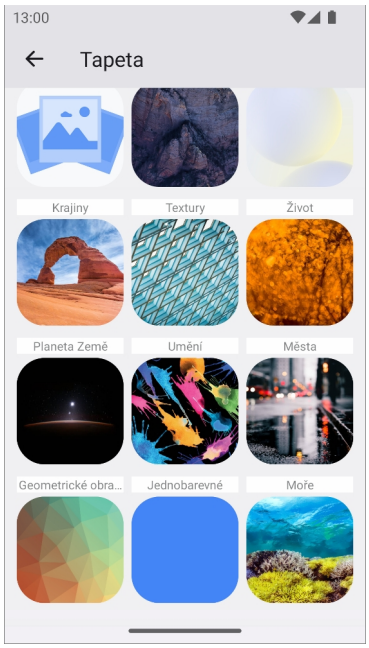


Obrázek 15.19: Aplikace s kontejnerem typu Expandable List View, spuštěná na emulátoru

15.5 Grid View

Grid View je kontejner určený k zobrazení elementů ve dvojrozměrné vertikálně rolovací mřížce. V základních rysech se podobá Recycler View – k datům se přistupuje pomocí adaptéru, který generuje položky mřížky a kontejner se stará o jejich zobrazení. Zdrojem dat je pro adaptér nejčastěji pole nebo databáze.

Stejně jako u Recycler View, i v tomto případě lze do podoby položky transformovat prostý text (první příklad), obrázek nebo vlastní složený formulář (druhý příklad). V systému Android je pomocí Grid View řešena například galerie fotografií, videa, nebo nastavení tapety.

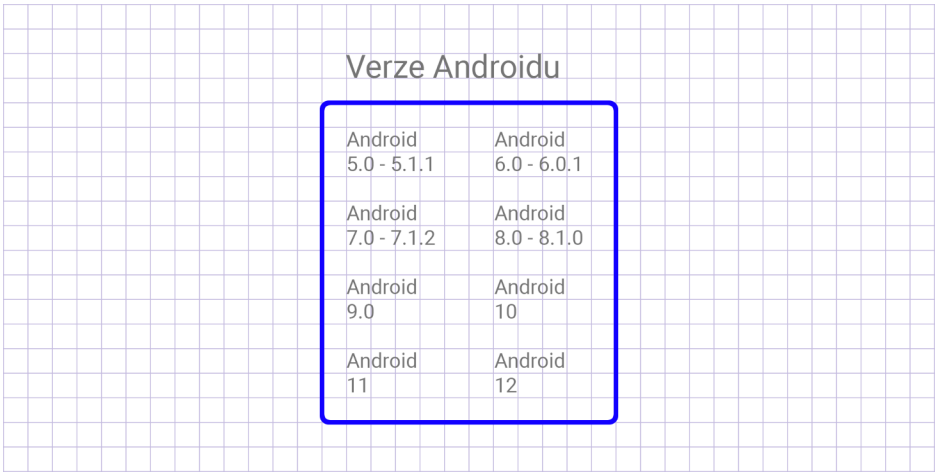


Obrázek 15.20: Příklad aplikace s kontejnerem typu Grid View

15.5.1 Grid View, 1. příklad

Začneme opět jednodušší z obou variant – seznamem položek seřazených do matice, vygenerovaných na základě dat definovaných v poli. Vzhled položek nebudeme nijak nastavovat, pouze předáme kontejneru Grid View seznam hodnot, které má zobrazit.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 15_Legacy
projekt GridView1



Obrázek 15.21: Návrh formuláře s kontejnerem Grid View

Co je potřeba udělat?

- vytvořit formulář s kontejnerem GridView
- napsat vlastní adaptér
- definovat pole hodnot, které bude obsahovat zobrazované údaje
- propojit adaptér s kontejnerem GridView

Otevřeme soubor activity_main.xml a do formuláře (v příkladu použijeme Constraint Layout) vložíme textové pole a kontejner Grid View. Rozměry se hodí nastavit tak, aby byla vyplněna celá plocha displeje a aby se při natočení displeje velikost kontejneru přizpůsobila (viz kapitola Constraint Layout). Kontejneru ve vlastnostech nastavíme id gridViewVerzeAndroidu. Kvůli rozložení polí přidáme ještě některé nezbytné parametry.

Způsob, jak v XML zapsat potřebné parametry Grid View:

`android:columnWidth="120dp"` – šířka sloupce mřížky

`android:numColumns="auto_fit"` - počet sloupců v mřížce. Hodnota „auto_fit“ povolí automatické nastavení maximálního počet sloupců které se do mřížky vejdou.

`android:verticalSpacing="16dp"` – mezera pod každou položkou v mřížce

XML kód kontejneru bude tedy vypadat nějak takto:

```
<!-- GridView gridViewVerzeAndroidu
    sirka a vyska max, okraje 8dp,
    navazana ze shora na textove pole,
    z ostatnich stran na parent kontejner,
    sirka sloupce 120dp, max pocet sloupcu, mezera 16dp -->
<GridView
    android:id="@+id/gridViewVerzeAndroidu"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginBottom="8dp"
    android:layout_marginLeft="8dp"
    android:layout_marginRight="8dp"
    android:layout_marginTop="8dp"
    android:columnWidth="120dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="16dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView" />
```

Ve třídě MainActivity.java vytvoříme pole, které bude zdrojem dat a nadefinujeme adaptér, který propojí toto pole s kontejnerem GridView.

Pole definujeme snadno, stejně jako u ListView i zde sestavíme pole textových řetězců. Použijeme následující Java zápis, viz příklad pole o dvou řetězcích:

```
String [] poleRetezcu = { "retezec1", "retezec2" };
```

Následuje zřízení adaptéru. Aby bylo možné adaptér použít, musíme ho nejdřív inicializovat, neboli vytvořit jeho instanci:

```
ListAdapter adapter = new ArrayAdapter<>(this,
      android.R.layout.simple_list_item_1, poleRetezcu);
```

Tím vznikne nová instance třídy ArrayAdapter s parametrem this, který určuje, že se tak stane v kontextu třídy MainActivity. Druhý parametr stanovuje způsob zobrazení položky v GridView na standardní, neupravený typ položky seznamu, tedy ve formátu jednoduchého textového pole TextView, se kterým umí GridView pracovat. Posledním atributem je zdroj dat, neboli pole textových řetězců.

Vytvořený adaptér propojíme s kontejnerem GridView:

```
GridView gridView = (GridView) findViewById(R.id.gridViewVerzeAndroidu);
gridView.setAdapter(adapter);
```

První řádek najde pole podle id a přiřadí je do proměnné gridView. Druhý řádek už jen zavolá metodu kontejneru, která k němu adaptér přiřadí.

Aby GridView reagoval na kliknutí uživatele na zvolenou položku, je potřeba tzv. posluchač neboli listener. Listener je rutina, která čeká, až nastane stanovená událost, v našem případě kliknutí. Jakmile k němu dojde, provede sekvenci příkazů uvedených ve složených závorkách.

```
AdapterView.OnItemClickListener clickListener = new
            AdapterView.OnItemClickListener() { .... }
```

Nejjednodušším způsobem jak událost obsloužit je předefinování původní události kontejneru onItemClick() tak, aby provedla to, co potřebujeme.

```
@Override
public void onItemClick(AdapterView<?> parent, View view, int pozice, long id)
{ ... }
```

Jelikož je událost onItemClick() univerzální a nemusí být nutně vázaná listenerem na jeden seznam, má v parametrech základní informace – který kontejner ji vyvolal, číslo položky, kterou si uživatel zvolil atd. Díky tomu lze následně rozklíčovat úmysl uživatele. Snadno se zjistí název vybrané položky:

```
String textPolozky = String.valueOf(parent.getItemAtPosition(pozice));
```

Parent je objekt nadřazený položce, čili v tomto případě kontejner GridView. Využijeme jeho metodu getItemAtPosition(), která vrací text položky na pozici uvedené v parametru. A pozice položky je předána v parametru pozice.

Stejně jako u ListView, i zde se spokojíme se zobrazením výsledku ve formě jednoduché toast zprávy:

```
Toast.makeText(MainActivity.this, textPolozky, Toast.LENGTH_SHORT).show();
```

Parametry jsou jednoduché – první je kontext, ve kterém se má zpráva zobrazit, v našem případě aktivita MainActivity, následuje text a doba po kterou má zpráva zůstat na displeji.

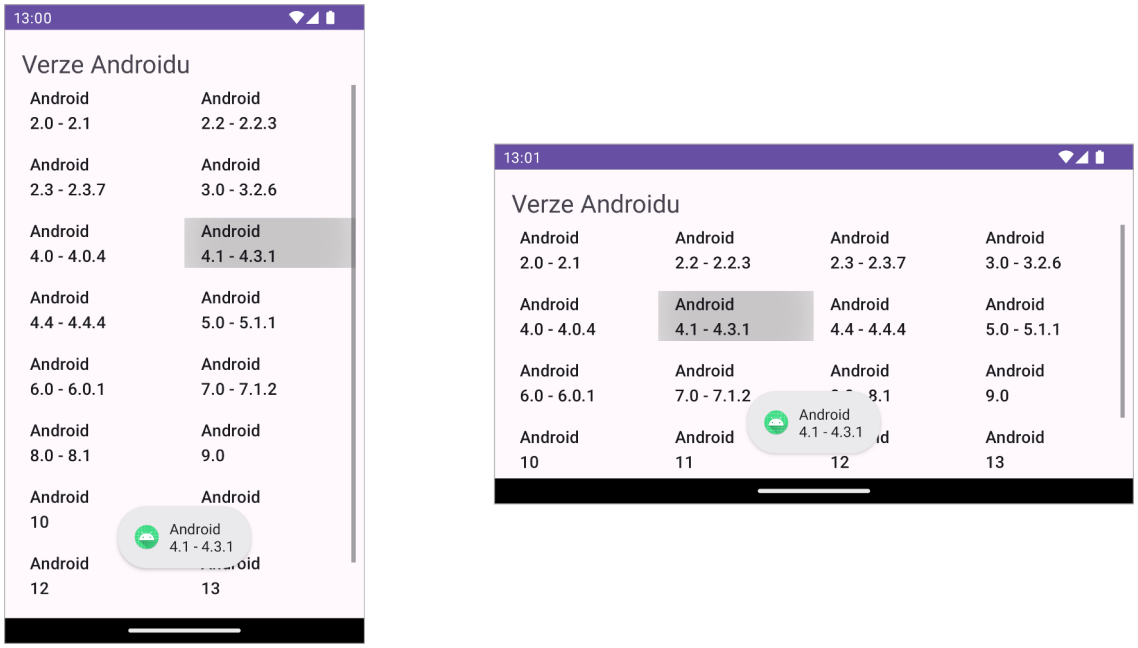
Takto vytvořený listener už zbývá jenom přiřadit ke kontejneru (nebo ke kontejnerům) typu GridView pomocí metody setOnItemClickListener(), jejímž parametrem je právě zmíněný listener.

```
gridView.setOnItemClickListener(clickListener);
```

Popsané příkazy upravíme pro naše potřeby, vložíme do vlastní metody a tu zavoláme z události onCreate() třídy MainActivity. Výsledný kód bude vypadat takto:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    naplnitGridView();
}

// Metoda nastavi hodnoty do GridView gridViewVerzeAndroidu
private void naplnitGridView()
{
    // vstupní data
    // znak \n je pouzity kvuli zalomeni textu
    String [] arrVerzeAndroidu = {
            "Android\n2.0 - 2.1",
            "Android\n2.2 - 2.2.3",
            "Android\n2.3 - 2.3.7",
            "Android\n3.0 - 3.2.6",
            "Android\n4.0 - 4.0.4",
            "Android\n4.1 - 4.3.1",
            "Android\n4.4 - 4.4.4",
            "Android\n5.0 - 5.1.1",
            "Android\n6.0 - 6.0.1",
            "Android\n7.0 - 7.1.2",
            "Android\n8.0 - 8.1",
            "Android\n9.0",
            "Android\n10",
            ...
    };

    // vytvorit instanci adapteru
    ListAdapter adapter = new ArrayAdapter<>(this,
                android.R.layout.simple_list_item_1, arrVerzeAndroidu);
    // propojeni adapteru s kontejnerem GridView
    GridView gridView = (GridView) findViewById(R.id.gridViewVerzeAndroidu);
    gridView.setAdapter(adapter);


    // vytvoreni instance posluchace udalosti kliknuti na polozku
    AdapterView.OnItemClickListener clickListener = new
                AdapterView.OnItemClickListener() {
        @Override
        // prepsani puvodni udalosti onItemClick kontejneru
        public void onItemClick(AdapterView<?> parent, View view, int pozice,
                long id) {
            // zjisti text na vybrane polozce
            String textPolozky =
                String.valueOf(parent.getItemAtPosition(pozice));
            // zobrazi text na displeji
            Toast.makeText(MainActivity.this, textPolozky,
                Toast.LENGTH_SHORT).show();
        }
    };

    // nastaveni posluchace udalosti
    gridView.setOnItemClickListener(clickListener);
}
```

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 15.22.



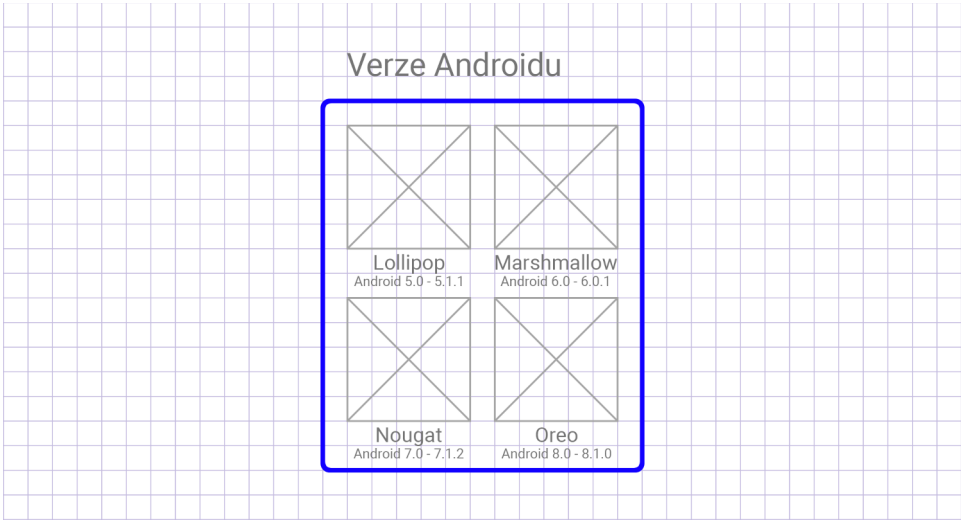
Obrázek 15.22: Aplikace s kontejnerem typu Grid View Layout, spuštěná na emulátoru

15.5.2 Grid View, 2. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 15_Legacy
projekt GridView2

Předchozí příklad se dá snadno upravit tak, aby položka místo textu obsahovala obrázek. Při praktickém použití ale pravděpodobně rychle zjistíme, že si nevystačíme ani s textem, ani s obrázkem, ale že bude potřeba na položce zobrazit formulář obsahující různé prvky, s předem definovaným rozložením. Stejně jako v příkladu ListView2 bude i zde potřeba použít vlastní definici formuláře a vlastní adaptér.

Vytvoříme mřížku s položkami, které budou uspořádány následujícím způsobem:



Obrázek 15.23: Návrh formuláře s kontejnerem Grid View

Co je potřeba udělat?

- vytvořit formulář položky
- vytvořit vlastní adaptér
- vytvořit pole hodnot, které bude obsahovat zobrazované údaje
- propojit adaptér s kontejnerem GridView

Jako první navrhneme design položky. V okně projektu ve složce res/layout/ založíme nový soubor (kontextové menu New → Layout resource file). Měl by mít nějaký příhodný název, třeba ‚polozka'. Při návrhu použijeme vhodný kontejner, například Constraint Layout. Objekty uspořádáme tak, aby odpovídaly naší představě o vizuální podobě položky seznamu. Opět platí, že je při návrhu vzhledu potřeba obzvlášť dávat pozor na efektivitu zobrazování formuláře, protože to se na displeji odehraje tolikrát, kolik položek mřížka obsahuje. Formulář by tedy měl být co nejjednodušší, s minimem vnoření.

Vytvořit vlastní adaptér, který transformuje zdrojová data do položky ve formátu View bude složitější. Kvůli jednoduchosti i v tomto příkladu předpokládejme, že vstupem bude pole textových řetězců. Nejdřív založíme Java soubor (kontextové menu New → Java Class) s nějakým vhodným názvem, třeba ‚PolozkaAdapter'. Jako první je potřeba oddědit novou třídu od třídy ArrayAdapter, ze které bude adaptér vycházet.

```
class PolozkaAdapter extends ArrayAdapter<String> { }
```

Hned nato založíme konstruktor nové třídy, který zavolá třídu nadřazenou. Tím se zajistí, že adaptér bude mít funkčnost adaptéru ArrayAdapter, kterou navíc rozšíříme.

```
public PolozkaAdapter(Context context, String[] androidVerze) {
    super(context, R.layout.polozka, androidVerze);
}
```

Poté předefinujeme metodu ArrayAdapteru getView() která za normálních okolností vrací textovou položku k zobrazení. Upravíme ji tak, aby vracela položku podle našich představ, tj. tak, jak jsme nadefinovali v souboru /res/layout/polozka.xml.

```
@Override
public View getView(int pozice, View convertView, ViewGroup parent) { }
```

Tím je struktura adaptéru vytvořena. Teď je ještě potřeba dát metodě getView() funkčnost, která zajistí správné vyplnění polí formuláře a sestavený formulář odešle návratovou hodnotou kontejneru GridView. Nejdřív transformujeme sestavený formulář do formátu View. Používá se k tomu takzvaný inflater:

```
convertView = LayoutInflater.from(getContext()).inflate(R.layout.polozka,
      parent, false);
```

Metoda getView() získává v parametrech několik důležitých informací. Jednak číslo položky kterou adaptér zrovna zpracovává a potom transformovaný pohled, pokud už existuje. jako třetí

parametr je nadřazený objekt, v tomto případě GridView. Zejména číslo položky nás bude zajímat, podle něj budeme v datech dohledávat co je vlastně potřeba zobrazit. K přístupu k datům z pole slouží metoda getItem(), které dáme jako parametr číslo položky. Toto číslo se shoduje s pozicí v poli vstupních dat, takže pro metodu getItem() není těžké hodnotu z pole vzít a vrátit ji. Uložíme ji do proměnné typu String, kterou nazveme strItem.

```
String strItem = getItem(pozice);
```

Dál už je jenom na vlastní fantazii vývojáře, jak formulář pomocí vstupních dat vyplní. V našem příkladu budeme mít v poli dat uloženy vždy dvojice informací – číslo a název verze Androidu, odděleny dvojtečkou. Zároveň jsou ve složce res/drawable/ uložena loga jednotlivých verzí Androidu, přičemž název loga odpovídá názvu verze Androidu. Takže ze strItem získáme název a číslo verze, přiřadíme je do polí ve formuláři, podle názvu se dohledá a přiřadí do kontejneru ImageView logo. Nakonec nesmíme zapomenout vrátit vytvořenou položku v návratové hodnotě.

Celý kód adaptéru bude vypadat takto:

```
class PolozkaAdapter extends ArrayAdapter<String> {

    // lokalni promenne
    private Context context;
    private String[] androidVerze;

    // konstruktor adapteru
    public PolozkaAdapter(Context context, String[] androidVerze) {
        super(context, R.layout.polozka, androidVerze);

        // ulozeni parametru do lokalnich promennych
        this.context = context;
        this.androidVerze = androidVerze;
    }

    @Override
    // predefinovani metody getView tak, aby vracela nadefinovanou polozku
    public View getView(int pozice, View convertView, ViewGroup parent) {

        // prevedeni polozky na View
        convertView = LayoutInflater.from(getContext())
                .inflate(R.layout.polozka, parent, false);

        // ziskani retezce z pole, podle pozice
        String strItem = getItem(pozice);

        // rozdeleni retezce na casti
        String[] strVersion = strItem.split(":");

        // nalezeni jednotlivych elementu a prirazeni do promennych
        ImageView imgLogo = (ImageView) convertView.findViewById(R.id.imgLogo);
        TextView txtName = (TextView) convertView.findViewById(R.id.txtName);
        TextView txtVersion = (TextView) convertView
                .findViewById(R.id.txtVersion);

        // zpristupneni resources
        Resources res = context.getResources();

        // jmeno resource s logem androidu
```

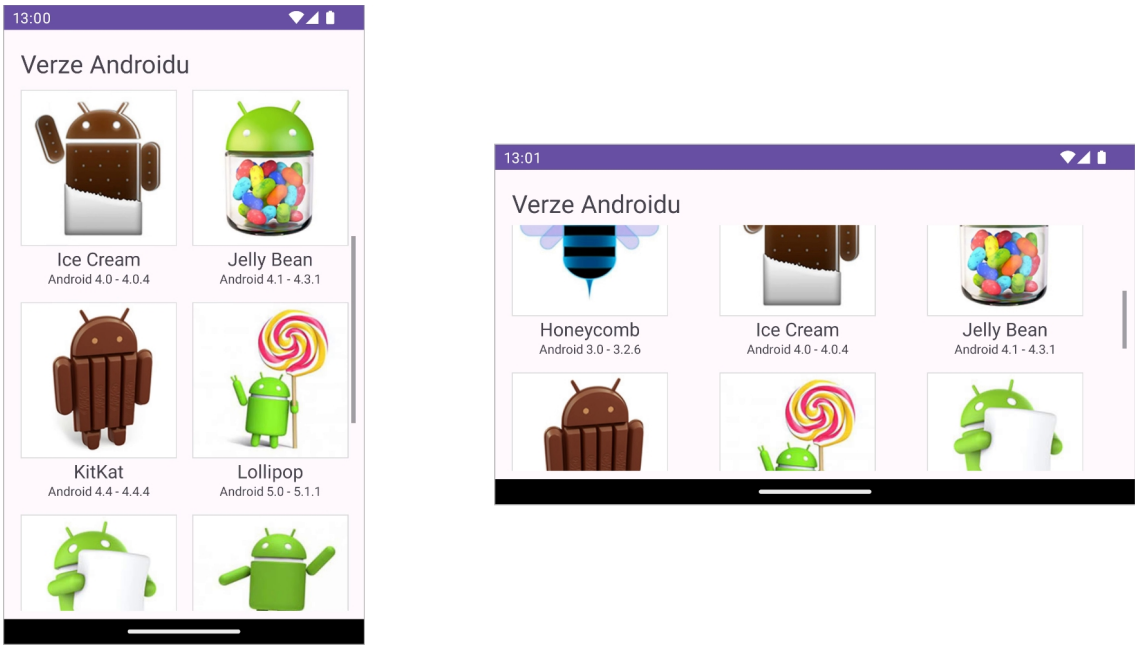
```
        String resName = strVersion[1].replaceAll(" ", "").toLowerCase();

        // id resource s logem androidu
        int imgLogoId = res.getIdentifier(resName, "drawable",
                  context.getPackageName());

        // prirazeni loga do ImageView
        imgLogo.setImageResource(imgLogoId);

        // prirazeni nazvu a verze do textovych poli
        txtName.setText(strVersion[1]);
        txtVersion.setText(strVersion[0]);

        // upravene View odeslat jako navratovou hodnotu
        return convertView;
    }
}
```

Tím je připravený formulář položky i adaptér. Vytvoříme si data a ve třídě MainActivity vše propojíme. Postup je podobný jako v předchozím příkladu. Ve třídě MainActivity v události onCreate() zavoláme vlastní metodu naplnitGridView(), která vygeneruje vstupní data, vytvoří instanci nového adaptéru, napojí ho na kontejner GridView a připojí posluchač události, který bude čekat na kliknutí na položku, které následně zpracuje.

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        naplnitGridView();
    }

    // Metoda nastavi hodnoty do GridView gridViewVerzeAndroidu
    private void naplnitGridView()
    {
        // vstupní data
        // vzdy dva texty oddelene dvojteckou, adapter si je rozdeli
        String [] arrVerzeAndroidu = {
                "Android 2.0 - 2.1:Eclair",
                "Android 2.2 - 2.2.3:Froyo",
                "Android 2.3 - 2.3.7:Gingerbread",
                "Android 3.0 - 3.2.6:Honeycomb",
                "Android 4.0 - 4.0.4:Ice Cream Sandwich",
                "Android 4.1 - 4.3.1:Jelly Bean",
                "Android 4.4 - 4.4.4:KitKat",
                "Android 5.0 - 5.1.1:Lollipop",
                "Android 6.0 - 6.0.1:Marshmallow",
                "Android 7.0 - 7.1.2:Nougat",
                "Android 8.0 - 8.1:Oreo",
                "Android 9.0:Pie",
                "Android 10 a vyšší:Android 10"
        };

        // vytvorit instanci adapteru
        ListAdapter adapter = new PolozkaAdapter(this, arrVerzeAndroidu);

        // propojeni adapteru s kontejnerem GridView
```

```
        GridView gridView = (GridView) findViewById(R.id.gridViewVerzeAndroidu);
        gridView.setAdapter(adapter);


        // vytvoreni instance posluchace udalosti kliknuti na polozku
        AdapterView.OnItemClickListener clickListener = new
                  AdapterView.OnItemClickListener() {

            @Override
            // prepsani puvodni udalosti onItemClick kontejneru
            public void onItemClick(AdapterView<?> parent, View view,
                   int pozice, long id) {

                // zjisti text na vybrane polozce
                String textPolozky = String.valueOf(parent
                        .getItemAtPosition(pozice));

                // zobrazi text na displeji
                Toast.makeText(MainActivity.this, textPolozky
                        .replace(":", " - "), Toast.LENGTH_SHORT).show();
            }
        };

        // nastaveni posluchace udalosti
        gridView.setOnItemClickListener(clickListener);
    }
}
```

Na koncovém zařízení nebo v emulátoru by měl výsledek vypadat jako na obrázku 15.24.

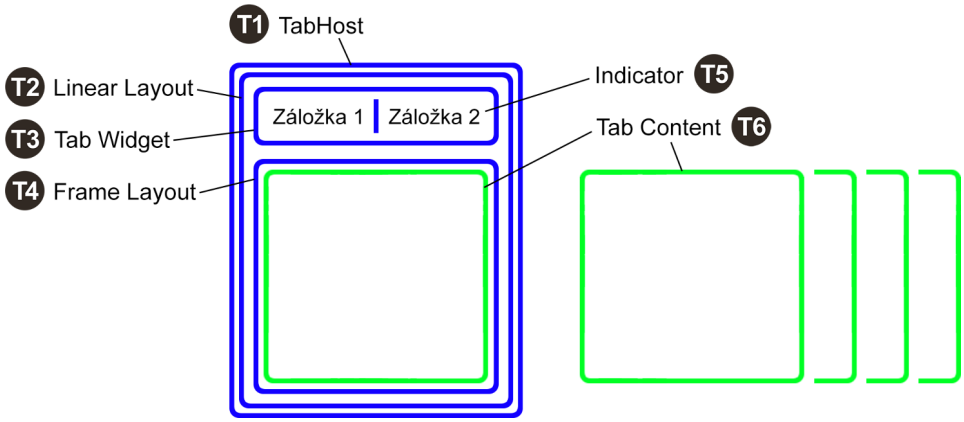


Obrázek 15.24: Aplikace s kontejnerem typu Grid View, spuštěná na emulátoru

15.6 Tab Host

Tato legacy komponenta je vizuálně shodná s modernější TabLayout. I ona představuje jednu z možností, jak dostat na displej více aktivit nebo views najednou a nechat uživatele aby si vybral, která ho v danou chvíli zajímá.

Struktura kontejneru Tab Host je následující:



Obrázek 15.25: Struktura kontejneru Tab Host

Na nejvyšší úrovni je samotný kontejner Tab Host T1. V něm je vložený vertikální Linear Layout T2, protože je nutné uspořádat pod sebe dvě další komponenty. Těmi jsou Tab widget T3 a Frame Layout T4. Tab Widget T3 obsahuje záhlaví záložek, takzvaný Indicator T5 a Frame Layout T4 slouží k přepínání Tab Content T6, neboli obsahu záložky.

Existují dva základní způsoby práce s TabHost:

1. přepínání více Views uvnitř jedné aktivity
2. přepínání mezi více aktivitami

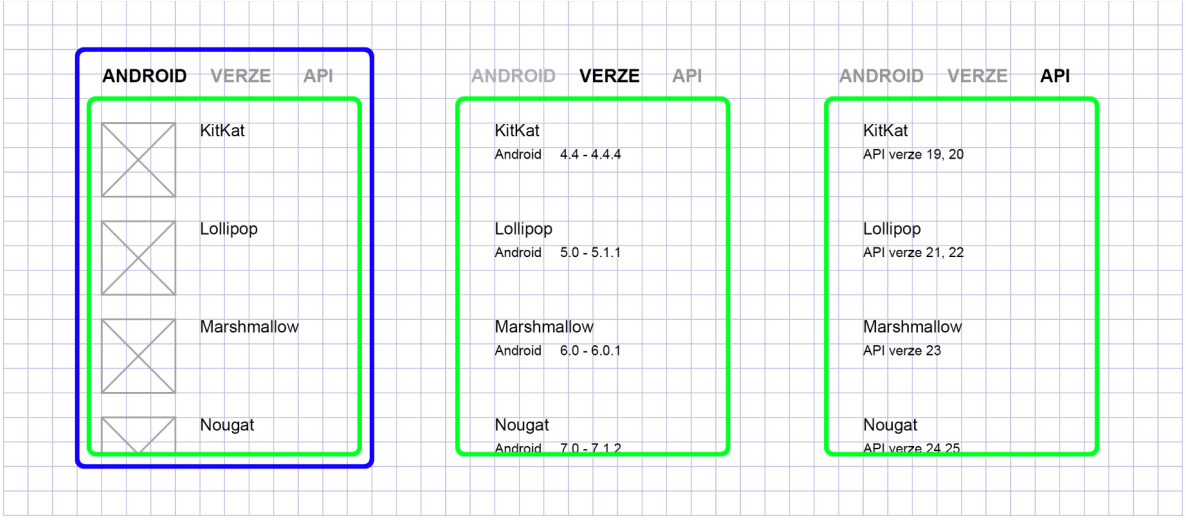
Kontejner Tab Host obsahuje záložky, z nichž každá využívá vlastní aktivitu. Aktivity na záložkách nejsou vzájemně nijak provázané, každá se řídí vlastním životním cyklem, shodným s životním cyklem standardní aktivity.

Nedílnou součástí komponenty je záhlaví s ikonkami pro každou záložku. Ikonky jsou zvlášť pro aktivní a zvlášť pro neaktivní záložky.

15.6.1 Tab Host, příklad

Příklad využití Tab Host kontejneru bude co možná nejjednodušší. Využijeme komponentu TabHost, která bude mít tři záložky. Na každé bude jednoduchý Scroll View. První záložka bude obsahovat posunovatelný seznam log Androidu, druhá seznam verzí Androidu a třetí seznam API Androidu. Záložky budou mít pouze textový nadpis.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 15_Legacy
projekt TabHost



Obrázek 15.26: Návrh formuláře s kontejnerem Tab Host

Co je potřeba udělat?

- uložit do resources vhodné obrázky pro první a texty pro další dvě záložky

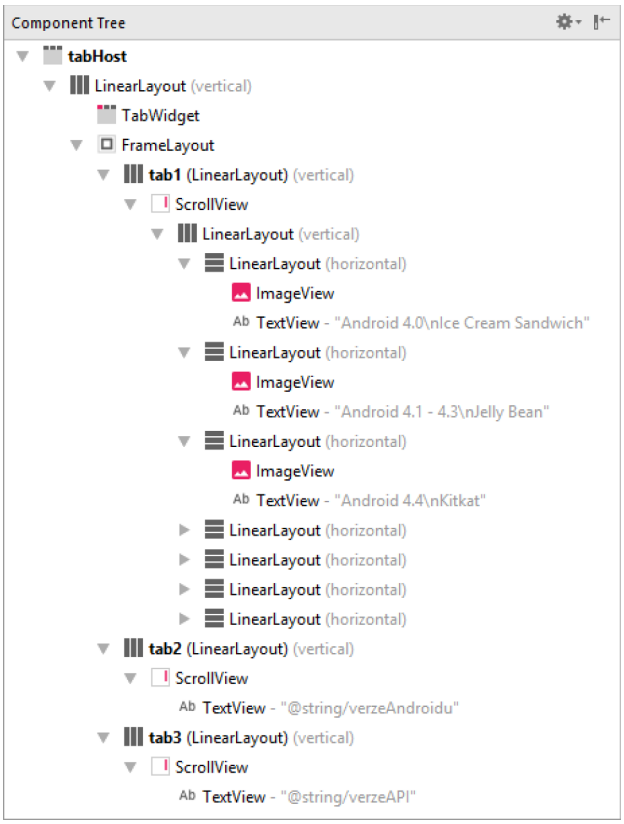
- vytvořit formulář s kontejnerem Tab Host, se třemi záložkami, každá se Scroll View a příslušným obsahem

- ve třídě MainActivity.java iniciovat v události onCreate() záložky

Do složky res/drawable/ na disku počítače, vložíme loga jednotlivých verzí Androidu, nebo jiné vhodné obrázky přiměřené velikosti. Texty pro zobrazení si vytvoříme tak, že do zdrojů res/values/strings.xml vložíme dva dlouhé řetězce s názvy verzeAndroidu a verzeAPI. Postup pro vytvoření textových řetězců je stejný jako v kapitole Scroll View.

V souboru activity_main.xml vložíme kontejner TabHost. Jelikož bude na nejvyšší úrovni jediným prvkem a všechny další komponenty budou vloženy do něj, můžeme (není to však nezbytné) odmazat kontejner vygenerovaný průvodcem, takže prvním elementem formuláře bude právě náš TabHost. Rozměry nastavíme na match_parent, aby byla vyplněna celá plocha displeje a aby se při natočení displeje velikost kontejneru přizpůsobila.

Při vložení TabHost komponenty v designeru, se nám zároveň vygenerovala celá struktura, potřebná pro naplnění komponenty jednotlivými záložkami a stránkami. Stačí tedy xml kód mírně upravit, vložit na každou stránku kontejner ScrollView a do něj už běžným způsobem obsah stránky. Na první záložce bude se ScrollView sada formulářů LinearLayout, z nichž každý bude tvořit jeden řádek s obrázkem a názvem verze Androidu. Druhá a třetí záložka budou obsahovat pole TextView s dlouhými textovými řetězci, které máme připravené v resources.



Obrázek 15.27: Prvky formuláře v okně Component Tree

Výsledný XML kód:

```
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/tabHost"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TabWidget
            android:id="@android:id/tabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        <!-- frame layout, s obsahem vsech zalozek
             sirka a vyska max -->
        <FrameLayout
            android:id="@android:id/tabcontent"
            android:layout_width="match_parent"
            android:layout_height="match_parent">

            <!-- linear layout, s obsahem prvni zalozky
                 sirka a vyska max, vertikalni -->
            <LinearLayout
```

```
    android:id="@+id/tab1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <!-- scroll view, sirka a vyska max -->
    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <!-- linear layout, sdruzuje vsechny radky
             sirka a vyska max, vertikalni -->
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical">
            <!-- linear layout, tvori jednu radku (obrazek + text)
                 sirka a vyska max, horizontalni -->
            <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:orientation="horizontal">
                <!-- ImageView - obrazek verze androidu z resources
                     sirka a vyska 120dp, okraj 2dp
                     sedy ramecek 1dp -->
                <ImageView
                    android:layout_width="120dp"
                    android:layout_height="120dp"
                    android:layout_margin="2dp"
                    android:background="#dddddd"
                    android:padding="1dp"
                    app:srcCompat="@drawable/icecreamsandwich" />
                <!-- textove pole s verzi Androidu
                     sirka max, vyska 50dp, okraj 5dp
                     velikost textu 30sp -->
                <TextView
                    android:text="Android 4.0\nIce Cream Sandwich"
                    android:layout_width="match_parent"
                    android:layout_height="50dp"
                    android:layout_margin="5dp"
                    android:textSize="20sp"/>
            </LinearLayout>

            ...

        </LinearLayout>
    </ScrollView>
</LinearLayout>

<!-- linear layout, s obsahem druhe zalozky
     sirka a vyska max, vertikalni -->
<LinearLayout
    android:id="@+id/tab2"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <!-- scroll view, sirka a vyska max -->
    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">
```

```
                <!-- textove pole, sirka max, vyska 35dp
                     okraj 5dp, velikost textu 15sp -->
                <TextView
                    android:layout_width="match_parent"
                    android:layout_height="match_parent"
                    android:layout_margin="5dp"
                    android:textSize="15sp"
                    android:text="@string/verzeAndroidu" />
            </ScrollView>
        </LinearLayout>

        <!-- linear layout, s obsahem treti zalozky
             sirka a vyska max, vertikalni -->
        <LinearLayout
            android:id="@+id/tab3"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical">

            <!-- scroll view, sirka a vyska max -->
            <ScrollView
                android:layout_width="match_parent"
                android:layout_height="match_parent">

                <!-- textove pole, sirka max, vyska max
                     okraj 5dp, velikost textu 15sp -->
                <TextView
                    android:layout_width="match_parent"
                    android:layout_height="match_parent"
                    android:layout_margin="5dp"
                    android:textSize="15sp"
                    android:text="@string/verzeAPI" />
            </ScrollView>
        </LinearLayout>
    </FrameLayout>
</LinearLayout>
</TabHost>
```

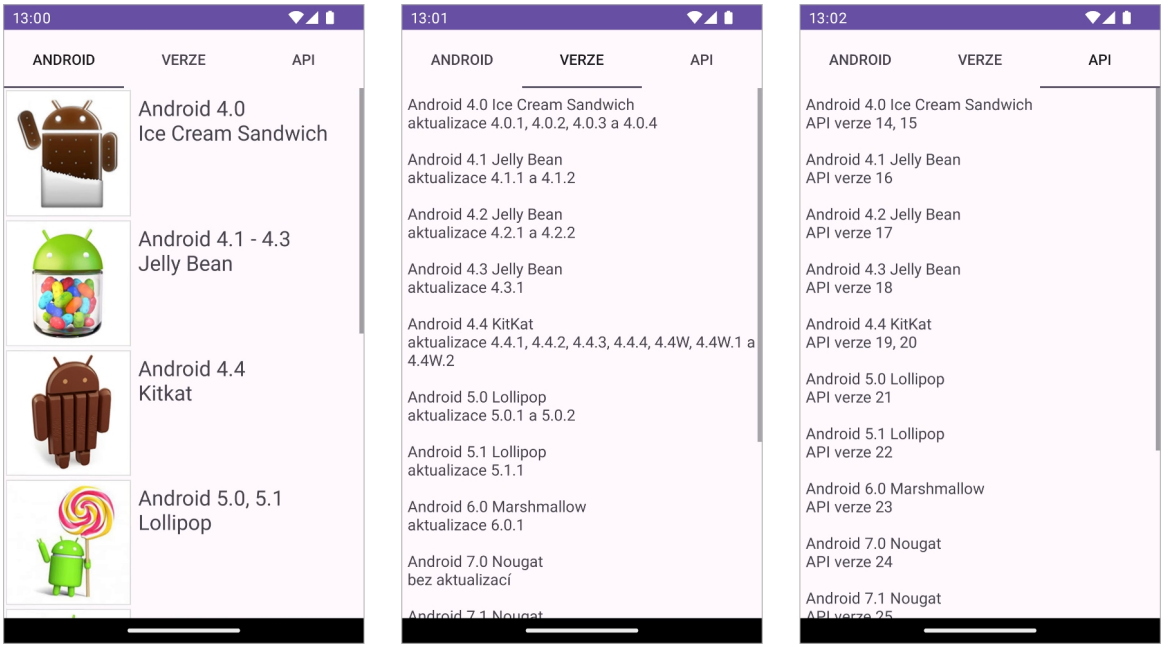
Aby komponenta HostView mohla něco zobrazit, musíme ji iniciovat. V souboru MainActivity.java ve vygenerované události onCreate() si nejprve do proměnné tabHost uložíme ukazatel na kontejner. Potom zavoláme standardní metodu setup(), kterou je nutné volat vždy před přidáváním záložek, pokud byl kontejner nalezen pomocí findViewById().

Uděláme instanci záložky TabSpec, do indikátoru nastavíme její název a naplníme ho obsahem který jsme si připravili v activity_main.xml. Takto vzniklou záložku připojíme ke kontejneru.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // najdeme kontejner TabHost
    // metodu setup je nutne zavolat pred vkladanim zalozek
    TabHost tabHost = (TabHost)findViewById(R.id.tabHost);
    tabHost.setup();

    // 1. zalozka (Android)
```

```
    TabHost.TabSpec tabSpec = tabHost.newTabSpec("Tab1");
    tabSpec.setIndicator("Android");
    tabSpec.setContent(R.id.tab1);
    tabHost.addTab(tabSpec);

    // 2. zalozka (Verze)
    tabSpec = tabHost.newTabSpec("Tab2");
    tabSpec.setIndicator("Verze");
    tabSpec.setContent(R.id.tab2);
    tabHost.addTab(tabSpec);

    // 3. zalozka (API)
    tabSpec = tabHost.newTabSpec("Tab3");
    tabSpec.setIndicator("API");
    tabSpec.setContent(R.id.tab3);
    tabHost.addTab(tabSpec);
}
```

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 15.28.



Obrázek 15.28: Aplikace s kontejnerem typu Tab Host, spuštěná na emulátoru

16 Dialogová okna

Dialogová okna neboli dialogy jsou malá okna, která se zobrazí na displeji zařízení ve chvíli, kdy je uživateli potřeba sdělit nějakou informaci nebo ho požádat o rozhodnutí či zadání jednoduchých dat. Na rozdíl od aktivity, dialogové okno nepřekryje celou plochu displeje, ale pouze jeho část. Původní okno, které bylo na displeji v okamžiku vyvolání dialogu, zůstává stále aktivní a zobrazené, pouze je dialogem částečně překryté. Po zavření dialogu se uživatel vrací na původní aktivitu. Samotné zobrazení dialogu nemá vliv na životní cyklus aktivity.

16.1 Toast

Tento nejjednodušší a zároveň nejčastěji používaný typ “dialogu“ už důvěrně známe z předchozích příkladů. Slovo dialog je v tomto případě potřeba brát s rezervou, protože toast nemá žádnou zpětnou vazbu. Pouze uživateli předá informaci a po uplynutí krátké doby se sám zavře. Přesnější by tedy bylo hovořit spíše o zprávě nebo notifikaci.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 16_Dialogova_okna
projekt Toast

Základní způsob zobrazení toast zprávy:

```
Context kontext = getApplicationContext();
String zprava = "Toast-test";
int trvani = Toast.LENGTH_LONG;
Toast toast = Toast.makeText(kontext, zprava, trvani);
toast.show();
```

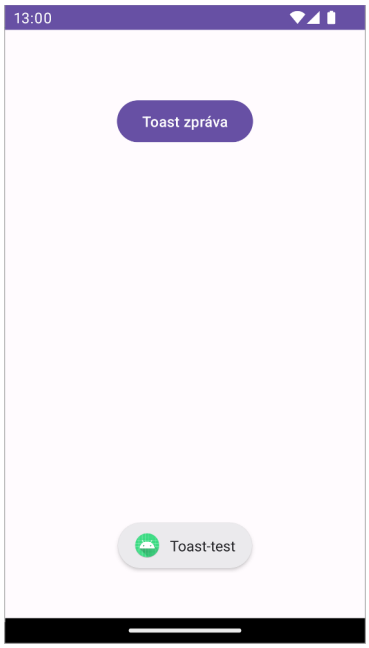
Zpráva se vytváří pomocí metody makeText() třídy Toast a musí mít tři parametry – kontext, ve kterém se má vytvořit, text zprávy a dobu, po kterou má zůstat na displeji. Vytvořenou zprávu lze následně zobrazit pomocí metody show().

Toto celé je možné zapsat i ve zhuštěné podobě na jednom řádku:

```
Toast.makeText(getApplicationContext(), "Toast-test", Toast.LENGTH_LONG).show();
```

Standardně se toast zobrazuje v dolní části displeje, zarovnaný na střed. Ve starších verzích systému bylo možné toast umístit na libovolné místo na displeji, nebo použít pro vzhled toast zprávy vlastní, uživatelsky definovaný formulář, který mohl obsahovat standardní objekty jako textová pole nebo obrázky. Od API verze 30 (Android 11) už ale takové úpravy nejsou podporovány, použít lze pouze jednoduché textové zprávy. Další nepříjemnost přinesla verze API 31 (Android 12). Zavedla omezení délky zobrazeného textu na maximálně dva řádky. Pro zprávy s větším počtem řádků je tedy vhodnější použít dialog Snackbar.

Toast zprávy potom vypadají jako na obrázku 16.1.



Obrázek 16.1: Toast zpráva, zobrazená v aplikaci spuštěné na emulátoru

16.2 Snackbar

Kromě základního dialogu typu Toast lze využít i další podobný dialog. Jmenuje se Snackbar a oproti Toastu má i jakousi minimalistickou možnost interakce, spočívající v tlačítku, kterým může uživatel na zobrazení zprávy reagovat. Standardně zobrazuje, stejně jako Toast, nejvýše dva řádky textu. Jednoduchým nastavením je ale možné toto omezení změnit a maximální počet řádků zvýšit.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 16_Dialogova_okna
projekt Snackbar

Základní způsob zobrazení snackbar zprávy:

```
View view = findViewById(android.R.id.content);
String zprava = "Snackbar - test";
int trvani = Snackbar.LENGTH_LONG;
Snackbar snackbar = Snackbar.make(view, zprava, trvani);
snackbar.show();
```

Zpráva se vytváří metodou make() třídy Snackbar a musí mít tři parametry – hierarchicky nejvyšší view formuláře, samotný text zprávy a dobu, po kterou má zůstat na displeji. Vytvořená zpráva se, podobně jako toast, zobrazí pomocí metody show().

Toto celé jde zapsat i ve zhuštěné podobě:

```
Snackbar.make(findViewById(android.R.id.content), "Snackbar - test",
                                                  Snackbar.LENGTH_LONG).show();
```

Snackbar zpráva se zobrazuje v dolní části displeje, s textem zarovnaným na levou stranu. Pokud jsou na tomto místě nějaké prvky formuláře, například plovoucí tlačítko, zpráva je odsune.

Zprávu je možné upravit několika dalšími jednoduchými způsoby. Dá se nastavit barva pozadí zprávy, barva textu, je možné přidat tlačítko a také jemu lze nastavit barvu.

Pro nastavení barvy pozadí zprávy je potřeba vzít view zprávy a na něm zavolat metodu setBackgroundColor() s požadovanou barvou v parametru:

```
View snackbarView = snackbar.getView();
snackbarView.setBackgroundColor(Color.GRAY);
```

Podobně se postupuje při nastavení barvy textu zprávy. Stačí najít textové pole se zprávou a na něm zavolat metodu setTextColor() s požadovanou barvou v parametru:

```
TextView textView = (TextView) snackbarView.findViewById(
                                com.google.android.material.R.id.snackbar_text);
textView.setTextColor(Color.WHITE);
```

O něco složitější je přidání tlačítka. Nejprve je třeba připravit posluchače události neboli listener, který bude čekat na stisknutí tlačítka:

```
View.OnClickListener onClickListener;
onClickListener = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // sem prijde kod, ktery se vyvola stisknutim tlacitka
    }
};
```

Potom se přidá tlačítko do zprávy. Přitom je možné rovnou nastavit barvu textu zavoláním metody setActionTextColor():

```
snackbar.setAction("Tlačítko", onClickListener);
snackbar.setActionTextColor(Color.YELLOW);
```

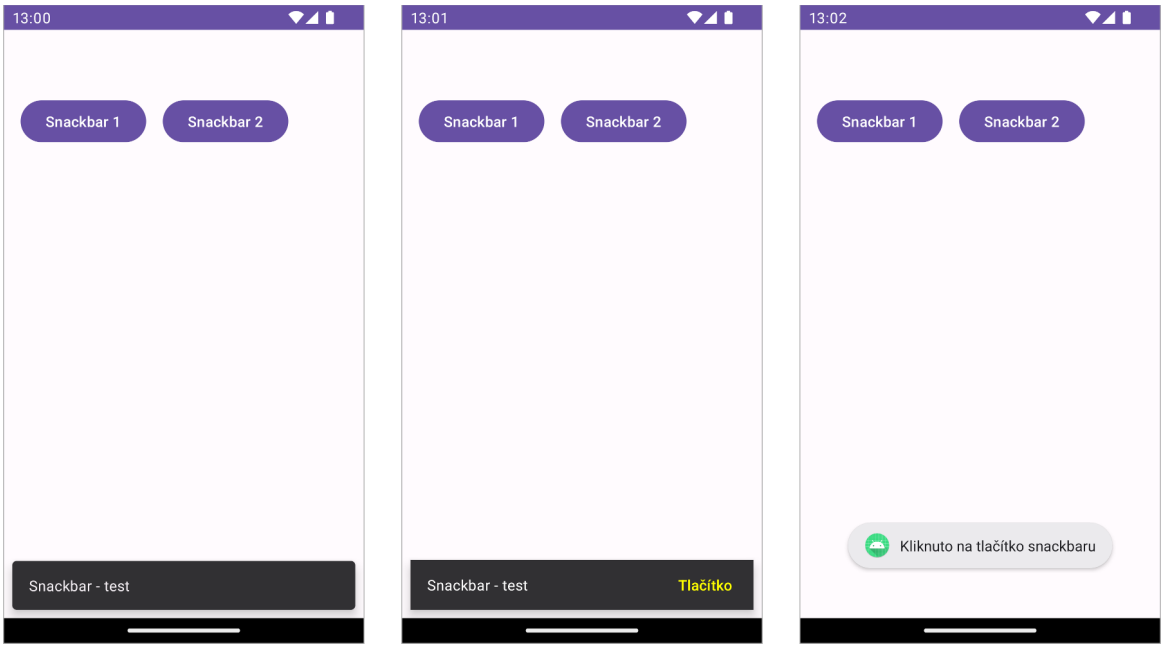
Metoda setAction() nastavila zprávě tlačítko s textem „Tlačítko“ a přiřadila mu posluchač události. Po kliknutí bude zavolána událost onClickListener().

Když se toto všechno spojí dohromady a na událost kliknutí na tlačítko se zobrazí klasická informační toast zpráva, bude kód vypadat zhruba takto:

```
// posluchac udalosti kliknuti na tlacitko
View.OnClickListener onClickListener;
onClickListener = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Toast.makeText(getApplicationContext(), "Kliknuto na tlačítko snackbaru"
                                    , Toast.LENGTH_LONG).show();
    }
};

// definice zprávy
Snackbar snackbar = Snackbar.make(findViewById(android.R.id.content)
                                    , "Snackbar - test", Snackbar.LENGTH_LONG);

// nastaveni tlacitka
```

```
snackbar.setAction("Tlačítko", onClickListener);
snackbar.setActionTextColor(Color.YELLOW);

// nastaveni barvy snackbaru
View snackbarView = snackbar.getView();
snackbarView.setBackgroundColor(Color.GRAY);

// nastaveni barvy textu
TextView textView = (TextView) snackbarView.findViewById(
                                com.google.android.material.R.id.snackbar_text);
textView.setTextColor(Color.WHITE);

// nastaveni maximalniho poctu radku (standardne zobrazi max dva radky)
textView.setMaxLines(5);

// zobrazeni snackbaru
snackbar.show();
```

Snackbar zprávy potom vypadají jako na obrázku 16.2 – vlevo jednoduchá zpráva, uprostřed zpráva s upravenou barvou pozadí a s tlačítkem a vpravo toast, který se zobrazí po kliknutí na tlačítko.



Obrázek 16.2: Aplikace s dialogy typu SnackBar (vlevo a uprostřed) a se zprávou typu Toast, zobrazenou po kliknutí na tlačítko doplněné do dialogu SnackBar

16.3 Alert dialog

Jde o dialog, který slouží k jednoduché interakci s uživatelem. Přesto má, kromě základních vlastností, poměrně mnoho možností úprav. Alert Dialog má tři základní části – záhlaví, určené pro ikonu a titulek, které může a nemusí být zobrazeno, potom oblast obsahu, kde je

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 16_Dialogova_okna
projekt Alert

možné zobrazit text zprávy, seznam volitelných položek nebo uživatelský formulář a oblast maximálně tří tlačítek.

Jednoduchý dialog, který zobrazí záhlaví, zprávu a zavírací tlačítko:

```
// vytvorit instanci tridy AlertDialog
AlertDialog.Builder builder = new AlertDialog.Builder(this);
// nastavit parametry dialogu
builder.setTitle("Záhlaví");
builder.setMessage("Test - alert dialog");
builder.setPositiveButton("Zavřít", null);
// dialog zobrazit
builder.create().show();
```

Jako první je potřeba vytvořit instanci třídy AlertDialog.Builder a předat jí v parametru kontext. Potom se vyplní text záhlaví a samotné zprávy, doplní definice tlačítka, které má pouze zavřít dialog, bez spouštění dalšího kódu a nakonec dialog vytvořit a zobrazit zavoláním metod create() a show().

Typy tlačítek se nastavují pomocí metod setPositiveButton(), setNegativeButton() nebo setNeutralButton(). Kliknutím na libovolné tlačítko se dialog vždy zavře, pozitivní nebo negativní reakci je nutné napsat až v kódu, který je vyvolaný událostí kliknutí na to které tlačítko. Rozdíl mezi setPositiveButton(), setNegativeButton() a setNeutralButton() je pouze v umístění tlačítka. V API před verzí 11 (Android 3, Honeycomb) se tlačítka zobrazovala v pořadí pozitivní – neutrální – negativní. Novější verze používají ergonomičtější neutrální – negativní – pozitivní.

Zavírání dialogu je také možné ovlivnit pomocí metody setCancelable(). Hodnota parametru false říká, že dialog nebude možné zavřít tlačítkem systému Zpět, nebo ťuknutím mimo dialog. Neboli že ho půjde zavřít pouze tlačítkem uvnitř dialogu:

```
builder.setCancelable(false);
```

Prvním parametrem metod pro zobrazení dialogu je text, který se na něm má zobrazit. Druhý parametr říká, jaká se má zavolat metoda. Pokud je uvedena hodnota null, nevolá se nic, ale předáním listeneru lze docílit zavolání metody, která se zpracuje v okamžiku kliknutí, tedy v okamžiku zavření dialogu.

```
// vytvorit instanci tridy AlertDialog
AlertDialog.Builder builder = new AlertDialog.Builder(this);
// nastaveni parametru dialogu
builder.setTitle("Záhlaví");
builder.setMessage("Test - alert dialog");
// definice listeneru pro tlačítko
DialogInterface.OnClickListener butListener =
                            new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
        Toast.makeText(getApplicationContext(), "Stisknuto tlačítko Zavřít",
                            Toast.LENGTH_LONG).show();
    }
};
builder.setPositiveButton("Zavřít", butListener);
// zobrazeni dialogu
builder.create().show();
```

Tím ale možnosti alert dialogu nekončí. Objekt builder má přístupnou také nenápadnou metodu setView(), díky které lze zobrazit vlastní formulář pro vzhled zprávy. Ta potom může vypadat úplně jinak než jako prostý textový dialog. Může obsahovat formátované texty, obrázky atd., zkrátka prakticky vše, co je přístupné ve formuláři. Vzhled je možné nastavit třeba jako obdélník se dvěma řádky textu. Nejprve je nutné definovat uživatelský formulář, ten uložit do samostatného xml souboru v projektové složce app/res/layout, pojmenovaný například uzivatelsky_alert.xml. Řekněme, že formulář bude typu Linear Layout a bude obsahovat dvě jednoduchá, editovatelná textová pole.

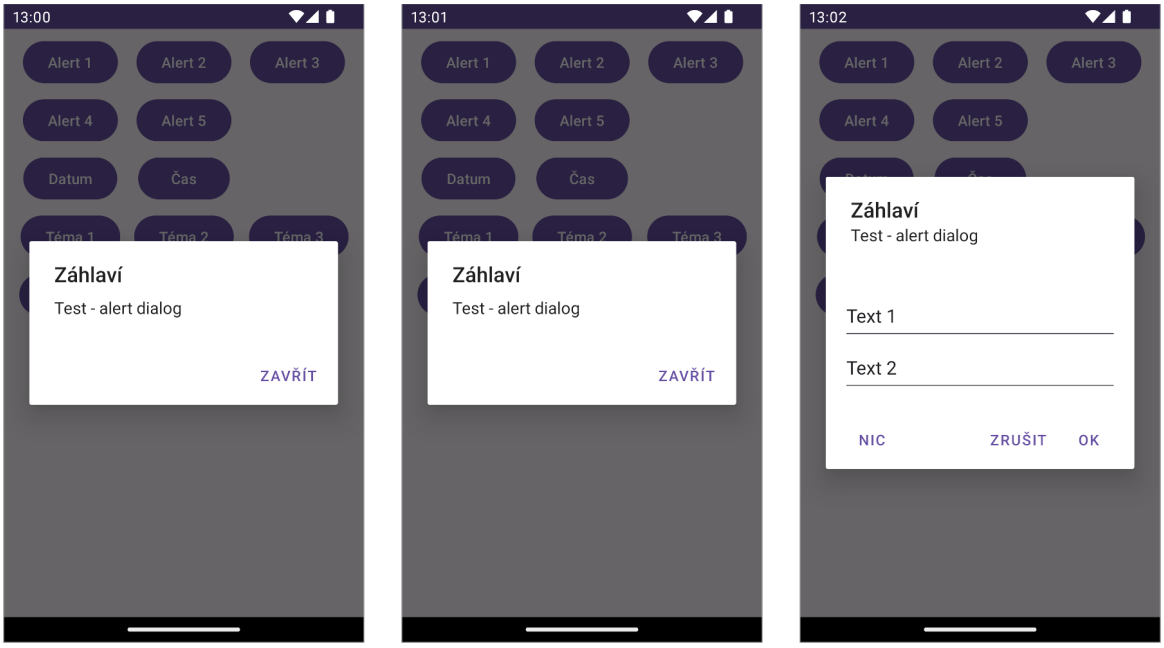
```
<!-- layout ktery definuje vzhled dialogu -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/custom_alert_container"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="16dp">

    <!-- prvni pole pro zadani textu, typ text
     sirka max, vyska 50dp, prednastaveny obsah 'Text 1' -->
    <EditText
        android:id="@+id/editText1"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:inputType="text"
        android:text="Text 1" />

    <!-- druhe pole pro zadani textu, typ text
         sirka max, vyska 50dp, prednastaveny obsah 'Text 2' -->
    <EditText
        android:id="@+id/editText2"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:inputType="text"
        android:text="Text 2" />
</LinearLayout>
```

Takto nadefinovaný formulář se pomocí inflateru serializuje a předá objektu builder metodou setView() jako součást dialogového okna.

```
// vytvorit instanci tridy AlertDialog
AlertDialog.Builder builder = new AlertDialog.Builder(this);

// serializace formulare
LayoutInflater inflater = getLayoutInflater();
View alertLayout = inflater.inflate(R.layout.uzivatelsky_alert, null);

// nastaveni parametru dialogu a tlacitka
builder.setTitle("Záhlaví");
builder.setMessage("Test - alert dialog");
builder.setPositiveButton("OK", null);
builder.setNegativeButton("Zrušit", null);
builder.setNeutralButton("Nic", null);

// vlozit do dialogu pripraveny formular
builder.setView(alertLayout);
```

```
// zobrazit dialog
builder.create().show();
```

Výsledek předchozích tří pokusů by měl vypadat jako na obrázku 16.3 – obyčejný alert dialog, vizuálně stejný dialog s listenerem pro tlačítko a dialog s uživatelským formulářem.



Obrázek 16.3: Různé typy Alert dialogů, zobrazené v aplikaci spuštěné na emulátoru

Alert dialog může mít také podobu seznamu, ze kterého si uživatel vybere jednu z možností. Stačí nad objektem builder zavolat metodu setItems() a té předat připravené pole textových řetězců, které následně Android zobrazí v grafické formě seznamu.

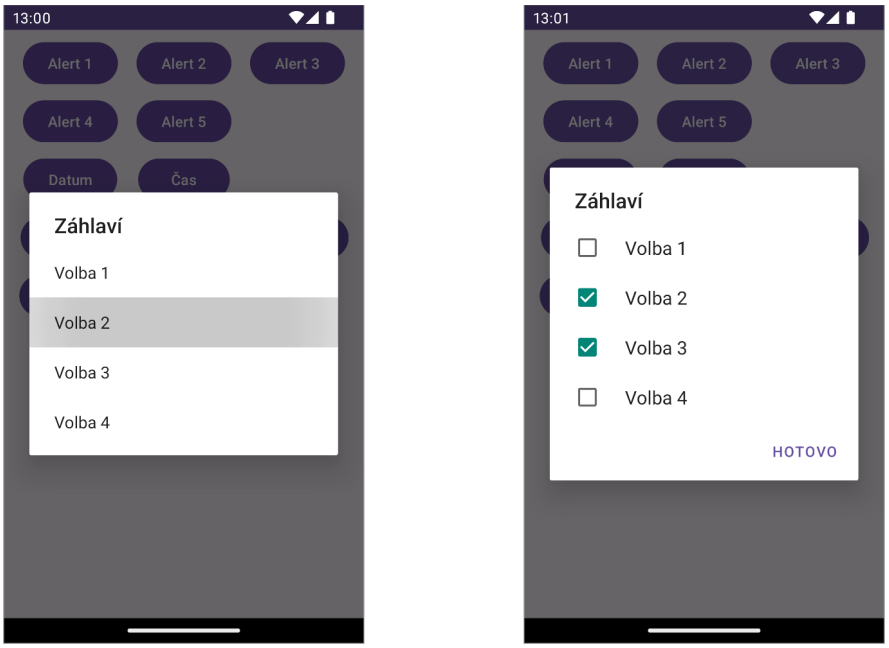
```
// pole nazvu polozek seznamu
String [] poleNazvu = { "Volba 1", "Volba 2", "Volba 3", "Volba 4" };

// vytvorit instanci tridy AlertDialog
AlertDialog.Builder builder = new AlertDialog.Builder(this);

// nastavit zahlavi dialogu
builder.setTitle("Záhlaví");

// nastavit seznam a listener pro kliknuti
builder.setItems(poleNazvu, new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int volba) {
            Toast.makeText(getApplicationContext(),
            "Zvolena možnost č. " + volba, Toast.LENGTH_LONG).show();
      }
});

// zobrazit dialog
builder.create().show();
```

V případě, že je po uživateli požadována volba více položek, je k dispozici zaškrtávací seznam, který se iniciuje voláním metody setMultiChoiceItems(). Seznam položek se opět předá parametrem, ve formátu pole textových řetězců. Tentokrát je ale seznam zaškrtnutých položek nutné

spravovat programově. To se nejsnáze udělá pomocí separátního pole, do kterého se budou průběžně ukládat indexy položek, které uživatel vybere.

```
// pole, do ktereho se budou ukladat vybrane polozky
ArrayList<Integer> zvolenePolozky = new ArrayList<Integer>();

// pole nazvu seznamu
String [] poleNazvu = { "Volba 1", "Volba 2", "Volba 3", "Volba 4" };

// vytvorit instanci tridy AlertDialog
AlertDialog.Builder builder = new AlertDialog.Builder(this);

// nastavit zahlavi dialogu
builder.setTitle("Záhlaví");

// nastavit seznam a listener pro kliknuti
builder.setMultiChoiceItems(poleNazvu, null, new
                        DialogInterface.OnMultiChoiceClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int volba, boolean zaskrtnuto)
      {
            if (zaskrtnuto) {
                // pri zaskrtnuti pridat do seznamu
                zvolenePolozky.add(volba);
            } else if (zvolenePolozky.contains(volba)) {
                // pri odskrtnuti vyradit ze seznamu
                zvolenePolozky.remove(Integer.valueOf(volba));
            }
      }
});

// listener pro tlačítko
DialogInterface.OnClickListener butListener = new
                        DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int id) {
            Toast.makeText(getApplicationContext(), "Zvolené položky: " +
                        zvolenePolozky.toString(), Toast.LENGTH_LONG).show();
      }
};
builder.setPositiveButton("Hotovo", butListener);

// zobrazit dialog
builder.create().show();
```

Výsledek pokusů se seznamy by měl vypadat jako na obrázku 16.4 – seznam a zaškrtávací seznam.



Obrázek 16.4: Alert dialog, rozšířený o seznam ze kterého si uživatel může zvolit odpovídající hodnotu nebo hodnoty. Zobrazeno v aplikaci spuštěné na emulátoru.

Android má několik tříd rozšiřujících Alert dialog. Prakticky velmi použitelný je DatePickerDialog, který slouží k zadání data. Funguje jako Alert dialog, ale v parametrech se mu předává jednak datum, které je potřeba zobrazit a potom také listener, který reaguje na samotnou volbu data.

```
// nastavit datum, napr. 1.1.2040
int iRok = 2040;
int iMesic = 0; // pozor, mesice se v Java kalendari cisluji od 0
int iDen = 1;

// listener pro zvolene datum v dialogu
DatePickerDialog.OnDateSetListener datumListener = new
                  DatePickerDialog.OnDateSetListener() {
      @Override
      public void onDateSet(DatePicker view, int rok, int mesic, int den) {
            String strVysledek = String.format("Zvolené datum %d.%d.%d",
                                          den, mesic, rok);
            Toast.makeText(getApplicationContext(), strVysledek,
                                          Toast.LENGTH_LONG).show();
      }
};

// vytvorit instanci dialogu
DatePickerDialog datePickerDialog = new DatePickerDialog(this, datumListener,
                                          iRok, iMesic, iDen);
// zobrazit dialog
datePickerDialog.show();
```

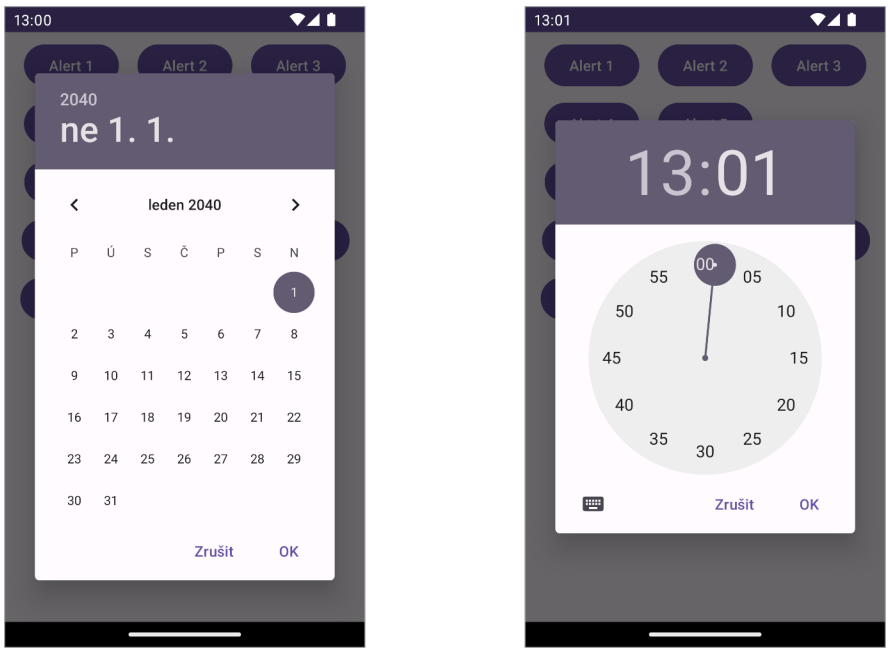
Rutina nejprve uloží do tří proměnných datum k zobrazení, zvlášť rok, měsíc a den. Nastavit lze jak aktuální datum, získané pomocí třídy Calendar, tak libovolné konkrétní datum, řekněme 1.1.2040. Potom si připraví instanci listeneru, který se zavolá po volbě data v dialogu. Listener pouze zobrazí pomocí toast zprávy datum, které bylo vybráno. Následně se vytvoří instance objektu

typu DatePickerDialog, předá se jí kontext, listener a všechny složky data. Nakonec je dialog uživateli zobrazen.

Stejně funguje i TimePickerDialog, který je rovněž rozšířením Alert dialogu a který slouží k zadávání času. Od DatePickerDialogu se mírně liší parametry – očekává, kromě kontextu a listeneru, hodinu a minutu, kterou má zobrazit. A také přepínač, který určuje, jestli se použije 12 nebo 24 hodinový formát.

```
// ziskat aktualni cas
final Calendar kalendar = Calendar.getInstance();
int iHodina = kalendar.get(Calendar.HOUR_OF_DAY);
int iMinuta = kalendar.get(Calendar.MINUTE);

// listener pro zvoleny cas v dialogu
TimePickerDialog.OnTimeSetListener casListener = new
                                        TimePickerDialog.OnTimeSetListener() {
      @Override
      public void onTimeSet(TimePicker view, int hod, int min) {
            String strVysledek = String.format("Zvolený čas %d:%d", hod, min);
            Toast.makeText(getApplicationContext(), strVysledek,
                                        Toast.LENGTH_LONG).show();
      }
};

// vytvorit instanci dialogu
TimePickerDialog timePickerDialog = new TimePickerDialog(this, casListener,
                                        iHodina, iMinuta, true);
// zobrazit dialog
timePickerDialog.show();
```

Výsledek pokusů by měl vypadat přibližně jako na obrázku 16.5 – dialog pro zadání data a času. Vizuálně můžou být ve vzhledu dialogu rozdíly podle použité verze systému.



Obrázek 16.5: Alert dialog, rozšířený o volbu data a času. Zobrazeno v aplikaci spuštěné na emulátoru.

Dialogy DatePickerDialog a TimePickerDialog mají v konstruktoru ještě jeden zajímavý parametr, který jsme dosud přecházeli. Je jím themeResId, typu integer, do kterého se může pomocí některé z konstant Androidu, zadat barevné schéma a vizuální formát ve kterém se má zobrazit.

Základní témata jsou tato:

Theme_DeviceDefault_Dialog_Alert – standardní formát dialogu a barevné schéma, které má uživatel v zařízení nastavené

Theme_DeviceDefault_Light_Dialog_Alert. - zjednodušený formát dialogu a barevné schéma, které má uživatel v zařízení nastavené

Kromě těchto dvou existuje velké množství dalších formátů a kombinací barevných schémat, které je možné využít. Takto je možné dosáhnout i na nové verzi Androidu vizuálního vzhledu dialogů ze starších verzí.

> Jelikož další témata stále přibývají, je vhodné při jejich volbě sáhnout po online dokumentaci na adrese https://developer.android.com/reference/android/R.style.html

Nastavení defaultního tématu a jeho použití při zobrazení dialogu DatePickerDialog:

```
int iTema = android.R.style.Theme_DeviceDefault_Dialog_Alert;
DatePickerDialog datePickerDialog =
                  new DatePickerDialog(this, iTema, null, iRok, iMesic, iDen);
datePickerDialog.show();
```

Kromě témat definovaných Androidem existuje rovněž možnost nadefinovat a použít styl vlastní. Stačí v okně projektu najít soubor se styly: app → res → values → styles.xml (pokud zde není, tak vytvořit přes kontextové menu → New → XML → Values XML File) a v něm si definovat nový, vlastní. Například:

```
<style name="datepicker" parent="Theme.AppCompat.Light.Dialog.Alert">
    <item name="colorAccent">#00ff00</item>     <!-- zahlavi -->
    <item name="colorControlActivated">#00ff00</item>   <!-- zvoleny den -->
    <item name="colorControlHighlight">#009900</item>   <!-- oznaceny dne -->
    <item name="android:selectableItemBackgroundBorderless">
      @color/material_dynamic_neutral40</item>  <!-- barva posuvniku mesice -->
</style>
```

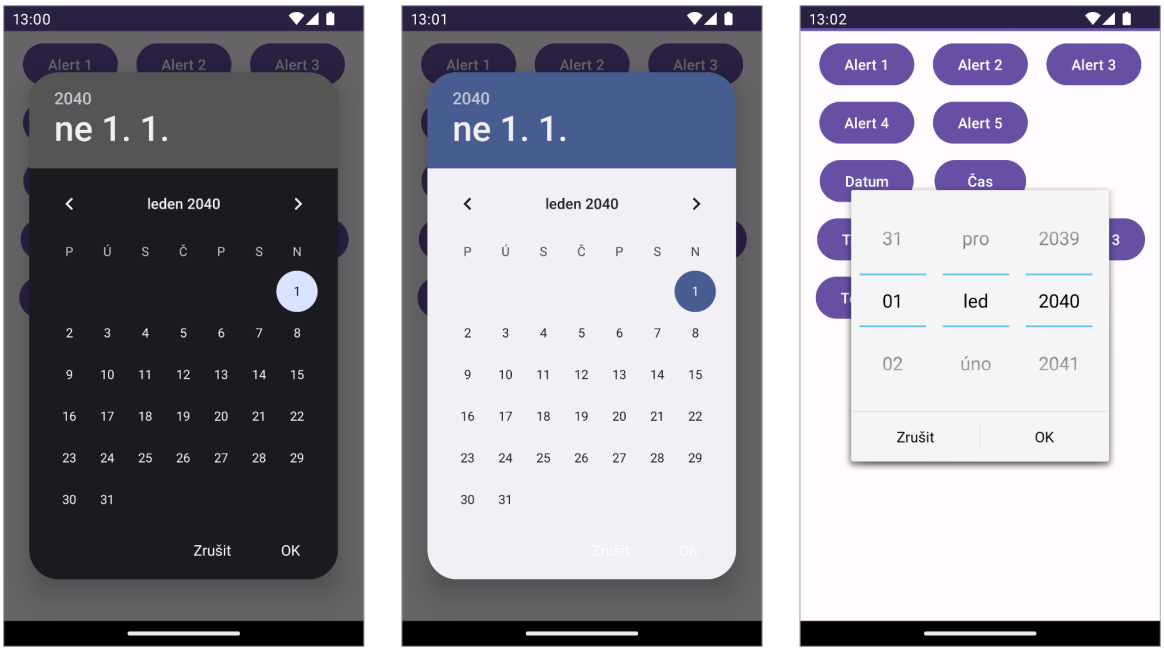
Takto definovaný styl nastaví barvy kalendáře do zelena. Aby nebylo nutné definovat všechny atributy, lze použít parametr parent, kterým se dá odkázat na definici stylu, která již v systému Android existuje a pouze změnit ty položky, které se mají lišit. Také pozor na poslední atribut, který nepoužívá kód barvy, ale odkazuje na její definici v souboru colors.xml.

Barvy se v Javě nastavují pomocí hexadecimální hodnoty složené z RGB složek (R=červená, G= zelená, B=modrá). Neboli – jde o tři hexadecimální čísla z intervalu 00-ff, která následují po sobě a před kterými je uvedený znak #. Například tedy #ff0000 je vyjádření červené barvy, #00ff00 modré, atd. Skládáním jednotlivých složek vznikají další barvy. Například #333333 je tmavě šedá.

Předpokládejme, že máme formulář se čtyřmi tlačítky (R.id.Button30, 31, 32 a 40). Tlačítka volají při stisknutí metodu alertDialogTema1(). Ta zobrazí kalendáře v různých tématech, s vyznačením aktuálního data:

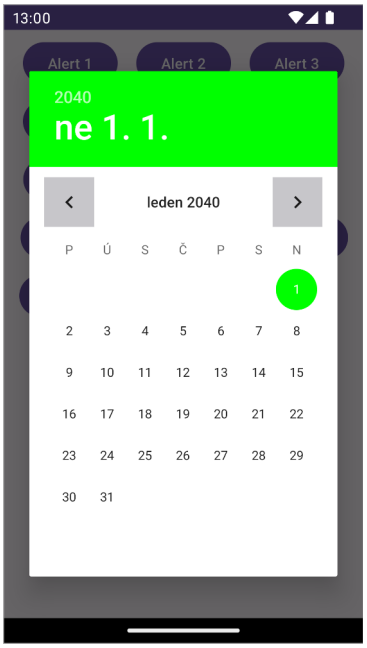
```
public void alertDialogTema1(View sender) {

    int iTema = android.R.style.Theme_DeviceDefault_Dialog_Alert;


    // volba stylu, podle tlacitka, na které uzivatel kliknul
    int butId = sender.getId();
    if (butId == R.id.button30)
        iTema = android.R.style.Theme_DeviceDefault_Dialog_Alert;
    else if (butId == R.id.button31)
        iTema = android.R.style.Theme_DeviceDefault_Light_Dialog_Alert;
    else if (butId == R.id.button32)
        iTema = android.R.style.Theme_Holo_Light_Panel;
    else if (butId == R.id.button40)
        iTema = R.style.datepicker;

    // nastavit datum, napr. 1.1.2040
    int iRok = 2040;
    int iMesic = 0; // pozor, mesice se v Java kalendari cisluji od 0
    int iDen = 1;

    // vytvorit instanci dialogu
    DatePickerDialog datePickerDialog =
                    new DatePickerDialog(this, iTema, null, iRok, iMesic, iDen);

    // zobrazit dialog
    datePickerDialog.show();
}
```

Na zařízení nebo emulátoru bude zobrazení prvních tří témat vypadat jako na obrázku 16.6. Je ale nutné počítat s tím, že jde o zobrazení na zařízení s API verze 31, na vyšších nebo nižších verzích mohou dialogy vypadat odlišně.



Obrázek 16.6: Alert dialog, rozšířený o volbu data, provedený v různých grafických stylech. Zobrazeno v aplikaci spuštěné na emulátoru.

V případě vlastního stylu, bude výsledek vypadat jako na obrázku 16.7.

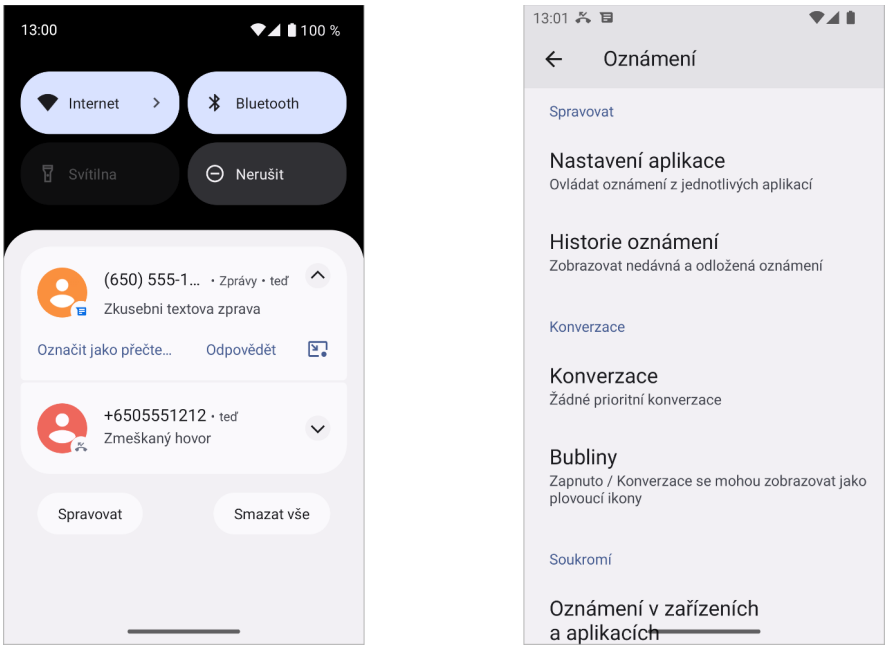


Obrázek 16.7: Alert dialog, rozšířený o volbu data, provedený ve vlastním grafickém stylu. Zobrazeno v aplikaci spuštěné na emulátoru

16.4 Oznámení

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 16_Dialogova_okna
projekt Notification

Oznámení, někdy také nazývaná upozornění či notifikace, se od předchozích dialogů trochu liší. A to především v tom, že se zobrazují mimo aplikaci, v uživatelském rozhraní systému Android. Bývají přístupné po přejetí prstem po displeji z jeho horní části směrem dolů, někdy můžou být také na zamykací obrazovce, nebo na ploše. Uživatelé mají možnost nastavit jakým způsobem chtějí být o oznámeních informováni a případně také která z nich se jim zobrazovat mají a která ne.



Obrázek 16.8: Vlevo oznámení o přijaté textové zprávě a textovém hovoru. Vpravo nastavení oznámení.

Základní způsob sestavení oznámení:

```
NotificationCompat.Builder builder =
                        new NotificationCompat.Builder(this, "NTF01");
builder.setContentTitle("Toto je notifikace");
builder.setContentText("Textový popis notifikace.");
builder.setSmallIcon(R.drawable.tlapka);
```

Pro vygenerování oznámení se používá třída NotificationCompat, od které je nutné vytvořit instanci a předat jí v parametrech kontext a unikátní identifikátor. Ten má formu textového řetězce a nejsou na něj kladeny žádné zvláštní nároky. Vývojář si ho může vytvořit v libovolné podobě, je potřeba dodržet pouze unikátnost. Identifikátor slouží k přiřazení oznámení ke kanálu oznámení, k tomu se dostaneme za chvíli.

Takto vytvořená instance poskytuje povinné metody setContentTitle() a setContentText() pro přiřazení textu titulku a obsahu oznámení. Povinné je i přiřazení ikonky pomocí setSmallIcon(). Ikonka musí být jednobarevná, s transparentním pozadím, systém si její obarvení řídí sám podle toho, jestli je zobrazena na světlém nebo tmavém pozadí. Velikost by měla být 24x24 nebo 48x48 obrazových bodů.

Google poskytuje na svých stránkách knihovnu ikonek, takže často není nutné vytvářet vlastní, stačí použít některou z takto předpřipravených a přizpůsobených systému. Knihovna je na internetové adrese https://fonts.google.com/icons

K zobrazení takto vytvořené notifikace se používá další třída, NotificationManager, resp. její novější verze NotificationManagerCompat. Zavoláním statické metody from() lze získat objekt, který poskytuje metodu notify() pro zobrazení, nebo cancel() pro ukončení oznámení. V parametru má jednak libovolně zvolený číselný identifikátor, který následně slouží jako reference na vyvolané oznámení a jako odkaz na samotnou definici oznámení:

```
NotificationManagerCompat manag = NotificationManagerCompat.from(this);
manag.notify(1, builder.build());
```

Již existující oznámení je možné zavřít obdobně, s použitím identifikátoru oznámení, zvoleného při jeho zobrazení. V našem případě je identifikátorem číslo 1:

```
NotificationManagerCompat manag = NotificationManagerCompat.from(this);
manag.cancel(1);
```

Zdrojový kód pro sestavení a zobrazení můžeme dát do metody, která se zavolá z formuláře, kliknutím na tlačítko. Takto to bude fungovat na starších verzích operačního systému, ale od Androidu verze 8 (API 26) je povinné zobrazovat oznámení prostřednictvím kanálu oznámení, ke kterému musí být přiřazené. Kanál zahrnuje vlastnosti a nastavení zobrazování zpráv, ta si uživatel může změnit. Kanálů může být v aplikaci definovaných víc, například jeden pro upozornění na aktualizace a druhý pro běžné aplikační informace. Kanály mohou mít rozdílná nastavení a uživatel si je může také rozdílným způsobem přizpůsobit. V každém případě všechna oznámení musí být systému předána prostřednictvím některého z těchto kanálů.

Kanál se vytvoří jako instance třídy NotificationChannel, přičemž prvním parametrem je unikátní identifikátor kanálu. Ten, který jsme použili už při vytváření oznámení. Druhým parametrem je název kanálu, pod kterým se zobrazí uživateli, pokud si ho bude chtít přizpůsobit, tedy nastavit jestli se mají oznámení předaná prostřednictvím tohoto kanálu vůbec zobrazit, jestli mají zavibrovat, pípnout atd. Třetím parametrem je důležitost oznámení, vysoká, standardní, nízká a minimální, viz tabulka 16.1.

Důležitost	Konstanta	Chování
vysoká	IMPORTANCE_HIGH	Zvukové oznámení, vibrace a zobrazení na displeji
standardní	IMPORTANCE_DEFAULT	Zvukové oznámení, vibrace
nízká	IMPORTANCE_LOW	Bez zvukového oznámení, bez vibrace
minimální	IMPORTANCE_MIN	Bez zvukového oznámení, bez vibrace a bez zobrazení v liště

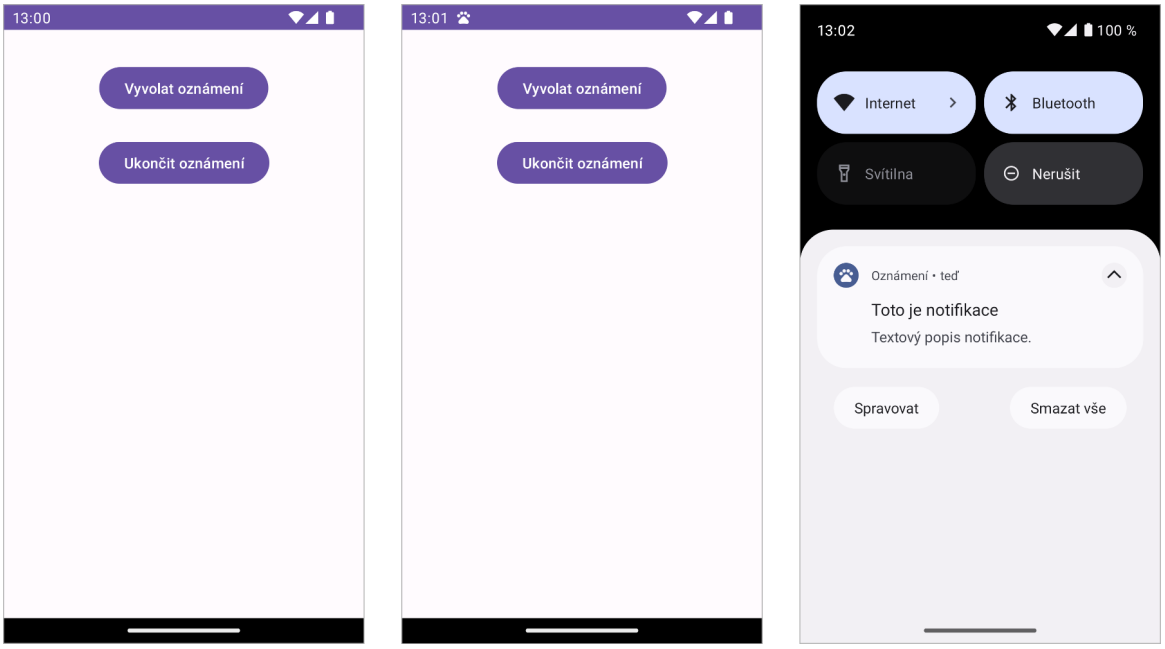
Tabulka 16.1: důležitost oznámení

Kanál se v systému aktivuje prostřednictvím metody createNotificationChannel() instance třídy NotificationManager:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    NotificationChannel notifKanal = new NotificationChannel("NTF01",
                        "Notifikace", NotificationManager.IMPORTANCE_DEFAULT);
    NotificationManager mNotificationManager = (NotificationManager)
                        getSystemService(Context.NOTIFICATION_SERVICE);
    mNotificationManager.createNotificationChannel(notifKanal);
}
```

Definice a aktivace kanálu může být umístěna do události onCreate(). Později už se na ni stačí pouze odkazovat pomocí identifikátoru.

Oznámení je možné dále široce konfigurovat. Dá se nastavit reakci na klepnutí do zprávy, přidat tlačítka, nebo vložit rovnou celý formulář s dalšími elementy uživatelského rozhraní. Dosud popsané oznámení bude na emulátoru nebo zařízení vypadat jako na obrázku 16.9 – vlevo formulář, uprostřed vyvolané oznámení, označené ikonou v horní liště a vpravo rozbalené oznámení.



Obrázek 16.9: Vlevo aplikace před vygenerováním oznámení, uprostřed otisk displeje s ikonu nového oznámení v horní liště a napravo rozbalené oznámení

17 Menu

Jedny z prvků uživatelského rozhraní, které usnadňují práci s aplikací, jsou menu. Operační systém Android podporuje několik typů menu:

Menu voleb (Options Menu) – otevírá se klepnutím na tři tečky v záhlaví okna. Obsahuje základní volby související se zobrazenou aktivitou.

Vyskakovací menu (PopUp Menu) - objeví se po klepnutí na prvek uživatelského rozhraní. Volby, které poskytuje, jsou zpravidla lokálního charakteru a vztahují se k položce, která menu vyvolala.

Kontextové menu (Context Menu) – otevře se po delším stisknutí prvku uživatelského rozhraní. Je to ekvivalent pravého tlačítka myši na počítači. Toto menu obvykle poskytuje speciální funkce vztažené k označenému objektu.

Vysouvací menu (Drawer Sliding Menu) – komplexní menu, které se vysouvá z levé strany displeje po stisknutí tlačítka v záhlaví.

Na jednotlivé typy menu se podíváme zvlášť, v samostatných kapitolách.

17.1 Menu voleb

V záhlaví aktivity ho obvykle symbolizují tři tečky nad sebou, ze kterých se po klepnutí otevře nabídka možných dalších činností. Toto menu se vkládá do části komponenty přidat do formuláře ToolBar, která tvoří jakési záhlaví formuláře. Má oficiální název Options Menu a vývojář musí vždy v editoru sestavit jeho design. Ať už vizuálně, nebo přímo v XML.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 17_Menu
projekt Menu1

Zkusíme si to na jednoduchém příkladu. Připravíme aplikaci, která bude obsahovat pouze menu a při zvolení položky o tom zobrazí informaci.

Co je potřeba udělat?

- vytvořit design menu

- přidat do formuláře ToolBar

- vložit menu do ToolBaru a napsat metodu, která zobrazí text po zvolení položky uživatelem

Vytvoříme soubor menu. Klikneme pravým tlačítkem na složku res a z kontextového menu zvolíme New → Android Resource File. Otevře se dialog, ve kterém nastavíme Resource type na hodnotu Menu. Soubor nazveme třeba menu_main. Otevře se designér, v němž jednoduše přidáme přetažením z palety položky Menu Item. Vytvoříme si jich několik a u každé vyplníme název a vhodné ID. V XML souboru to bude vypadat zhruba takto:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:id="@+id/nastaveni"
          android:title="Nastavení" />
    <item android:id="@+id/test1"
```

```
          android:title="Test 1" />
    <item android:id="@+id/test2"
          android:title="Test 2" />
    <item android:id="@+id/napoveda"
          android:title="Nápověda" />
    <item android:id="@+id/oaplikaci"
          android:title="O aplikaci" />
</menu>
```

Jelikož formulář nemá žádné vhodné záhlaví, ve kterém by mohlo být umístěné pole pro vyhledávání, tak do formuláře activity_main.xml vložíme z palety komponentu ToolBar. Postup je stejný jako v příkladu 12.5 Záhlaví formuláře. Nastavíme jí základní vlastnosti, umístění, velikost, barvu a vložený text:

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:minHeight="?attr/actionBarSize"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:titleTextColor="@android:color/white"
    app:title="@string/app_name"
    android:background="?attr/colorPrimary">
</androidx.appcompat.widget.Toolbar>
```

V MainActivity.java doplníme do události onCreate() dva řádky, z nichž první dohledá na formuláři ToolBar a druhý sdělí aktivitě, že s ním má pracovat jako s vlastním záhlavím:

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar)
```

Nyní lze k záhlaví ToolBar připojit menu. Učiníme tak předefinováním události onCreateOptionsMenu() v MainActivity:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}
```

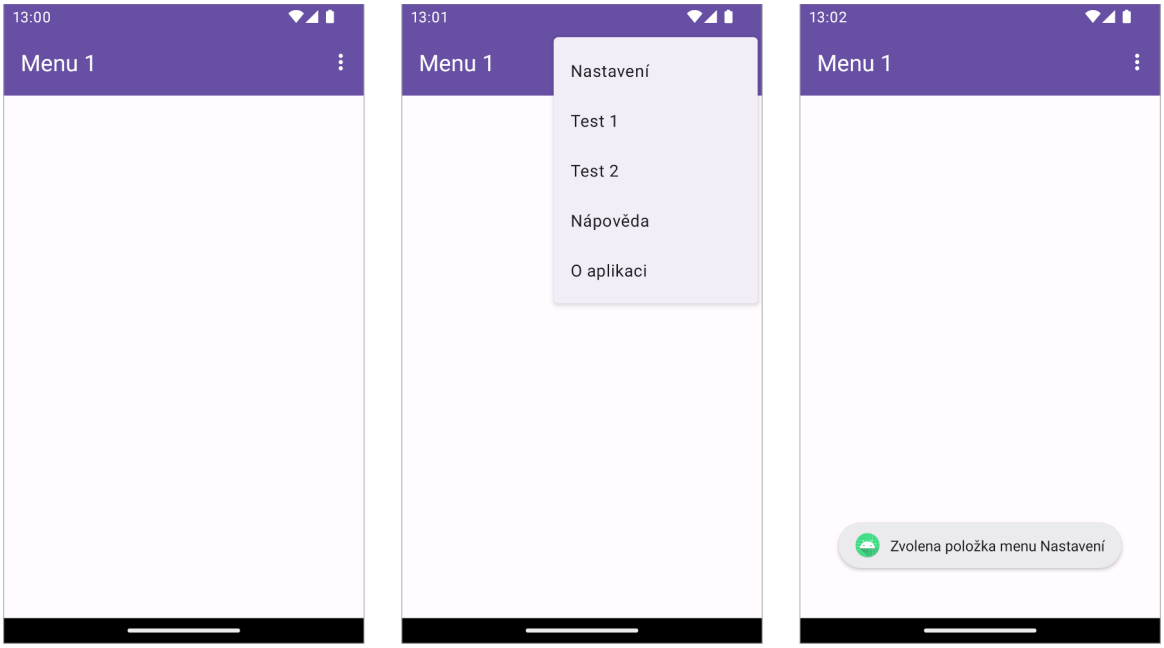
Po kompilaci a spuštění aplikace zobrazí menu v ToolBaru. Zbývá přidat událost, která bude čekat na volbu položky menu. Opět ji uděláme předefinováním události v MainActivity, tentokrát onOptionsItemSelected():

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.nastaveni) {
        Toast.makeText(this, "Zvolena položka menu Nastavení",
                                          Toast.LENGTH_SHORT).show();
        return true;
    }
    else {
```

```
            return super.onOptionsItemSelected(item);
        }
    }
```

Událost se spouští po volbě libovolné položky menu a my bychom měli následně, pomocí metody getItemId() zjistit, která položka byla zvolena. Pro zjednodušení ošetříme pouze položku Nastavení, všechny ostatní budeme ignorovat, resp. pouze zavoláme původní předefinovanou metodu, aby věděla, že se o zpracování nestaráme.

Výsledek by měl na emulátoru nebo koncovém zařízení vypadat jako na obrázku 17.1.



Obrázek 17.1: Aplikace s menu v záhlaví, označené třemi tečkami, spuštěná na emulátoru

17.2 Vyskakovací menu

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 17_Menu
projekt Menu2

Tento typ menu se v aplikacích objevuje po klepnutí na prvek uživatelského rozhraní, například na tlačítko. Pokud je na displeji dost místa, otevře se tak, aby levým horním rohem bylo přibližně na spodním levém rohu označeného prvku. Možnosti, které toto menu uživateli nabízí, jsou rozšířením základní funkce označeného prvku. Tedy například, když v galerii fotografií klikne na ikonku sdílení, tak se místo okamžitého nasdílení otevře vyskakovací menu s volbou typu sdílení – Bluetooth, E-mail atd.

Vyskakovací, neboli PopUp menu je podobně snadné, jako předchozí Menu voleb. Jeho použití si vyzkoušíme na aplikaci, která bude obsahovat jednu komponentu uživatelského rozhraní, při jejímž zvolení se zobrazí vyskakovací jednoduché menu.

Co je potřeba udělat?

- vytvořit formulář s prvkem uživatelského rozhraní, například tlačítkem

- vytvořit design menu

- svázat menu s prvkem uživatelského rozhraní

- přidat událost, která provede akci podle zvolené položky menu

Formulář vytvoříme obvyklým způsobem, přidáme na něj libovolně umístěné tlačítko a tím bude připravený.

Design menu připravíme, stejně jako v předchozím příkladě tak, že si založíme soubor menu v projektovém adresáři app/res/menu/ a nazveme ho například menu_popup.xml. Do menu vložíme jednotlivé položky, takže výsledný XML soubor bude vypadat zhruba takto:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/test1"
          android:title="Test 1" />
    <item android:id="@+id/test2"
          android:title="Test 2" />
    <item android:id="@+id/test3"
          android:title="Test 3" />
    <item android:id="@+id/napoveda"
          android:title="Nápověda" />
</menu>
```

V souboru MainActivity.java napíšeme metodu, kterou budeme volat po stisknutí tlačítka. Tato metoda připraví a následně zobrazí vyskakovací menu. Kromě toho ještě zaregistruje listener pro kliknutí na položku menu, který zajistí, aby v takovém případě byla zavolána metoda zvolenaPolozkaMenu().

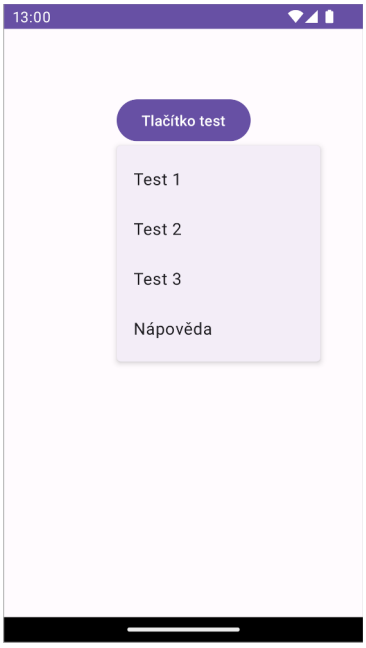
```
public void otevritPopUpMenu(View v) {
    // vytvorit menu
    PopupMenu popup = new PopupMenu(this, v);
    MenuInflater inflater = popup.getMenuInflater();
    inflater.inflate(R.menu.menu_popup, popup.getMenu());

    // kliknuti na polozku menu
    popup.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
        @Override
        public boolean onMenuItemClick(MenuItem item) {
            zvolenaPolozkaMenu(item);
            return true;
        }
    });

    // zobrazit menu
    popup.show();
}
```

Metoda zvolenaPolozkaMenu() už jen vykoná činnost, kterou požadujeme po volbě položky. My se nyní spokojíme se zobrazením informace, že položka menu byla zvolena:

```
private void zvolenaPolozkaMenu(MenuItem item) {
    Toast.makeText(this, "Zvolena položka menu " + item.getTitle(),
```

```
                                                Toast.LENGTH_SHORT).show();
}
```

Metodu otevritPopUpMenu() zavoláme z události tlačítka onClick(). Výsledek by měl na emulátoru nebo koncovém zařízení vypadat jako na obrázku 17.2.



Obrázek 17.2: Aplikace s vyskakovacím menu, spuštěná na emulátoru

17.3 Kontextové menu

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 17_Menu
projekt Menu3

Tento typ menu se podobá pravému tlačítku myši na operačních systémech počítačů. Android žádné pravé tlačítko nemá, přesto čas od času potřebuje uživateli dát možnost kontextové volby. Řeší se to delším stisknutím prvku uživatelského rozhraní. Android v takovém případě otevře kontextové menu, pokud je pro daný prvek k dispozici. Kontextová menu se, jak už jejich název napovídá, pro různé prvky liší.

Toto menu je podobné vyskakovacímu menu, rozdíl je pouze ve způsobu registrace události. Ale k tomu se ještě dostaneme. Nyní si vytvoříme jednoduchou aplikaci, která bude obsahovat dvě komponenty uživatelského rozhraní, při jejíchž zvolení se zobrazí vyskakovací jednoduché menu.

Co je potřeba udělat?

- vytvořit formulář se dvěma prvky uživatelského rozhraní, například obrázkem a tlačítkem
- vytvořit design dvou různých kontextových menu
- svázat menu s prvky uživatelského rozhraní
- přidat událost, která provede zvolenou položku menu

Formulář vytvoříme obvyklým způsobem, přidáme na něj libovolný obrázek do kontejneru ImageView s id „obrazek“, pod něj umístíme tlačítko s id „tlacitko“ a tím je formulář připravený.

Design menu připravíme, stejně jako v předchozím příkladu tak, že si založíme dva soubory menu v projektovém adresáři app/res/menu/ a nazveme je například menu_context1.xml a menu_context2.xml. Do obou menu vložíme odlišné položky tak, abychom je od sebe dokázali rozeznat. Postup vytváření souborů je stejný jako v minulém příkladu.

V souboru MainActivity.java v události onCreate() zaregistrujeme obě menu na prvky formuláře:

```
// zaregistrovat kontextove menu na obrazek
ImageView obrazek = (ImageView) findViewById(R.id.obrazek);
registerForContextMenu(obrazek);

// zaregistrovat kontextove menu na tlacitko
Button button = (Button) findViewById(R.id.tlacitko);
registerForContextMenu(button);
```

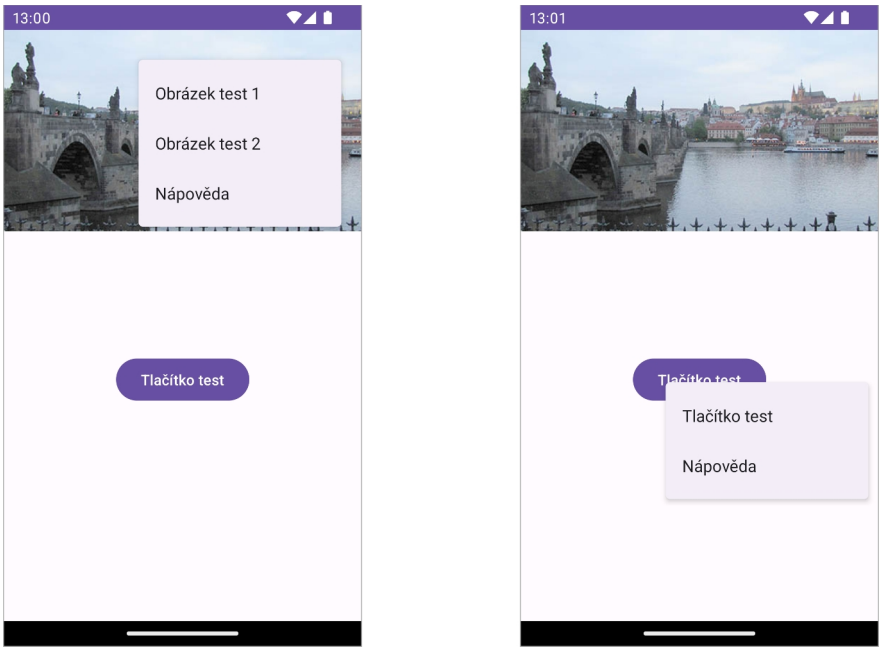
Tím jsme zajistili, že systém po delším stisknutí prvku vyvolá menu. Přesněji řečeno, zavolá událost onCreateContextMenu(). Tu musíme předefinovat a v ní otevření menu, v závislosti na objektu, který uživatel označil, zajistíme.

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenu.ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getMenuInflater();

    if(v.getId()==R.id.obrazek)
        inflater.inflate(R.menu.menu_context1, menu);
    else if(v.getId()==R.id.tlacitko)
        inflater.inflate(R.menu.menu_context2, menu);
}
```

Zbývá předefinovat událost pro provedení zvolené položky. Spokojíme se s vypsáním názvu položky, kterou si uživatel vybral.

```
@Override
public boolean onContextItemSelected (MenuItem item){
    Toast.makeText(this, "Zvolena položka menu " + item.getTitle(),
                                                 Toast.LENGTH_SHORT).show();
    return super.onContextItemSelected(item);
}
```

Výsledek by měl na emulátoru nebo koncovém zařízení vypadat jako na obrázku 17.3.



Obrázek 17.3: Aplikace s kontextovým menu, spuštěná na emulátoru a reagující různým způsobem nad rozdílnými objekty formuláře

17.4 Vysouvací menu

Nejnovějším a současně nejkomplexnějším typem menu, se kterým se lze u Androidu setkat, je takzvané Drawer Menu. Je to nabídka, která se vysouvá z levé strany displeje a obvykle je její přítomnost v aplikaci signalizovaná v záhlaví ikonkou takzvaného „hamburger menu“, neboli tří čárek nad sebou. S tímto menu se dá narazit zejména v aplikacích s rozsáhlejší strukturou, nebo s větším množstvím voleb, které by se nedaly obsloužit použitím Tab Layout kontejneru.

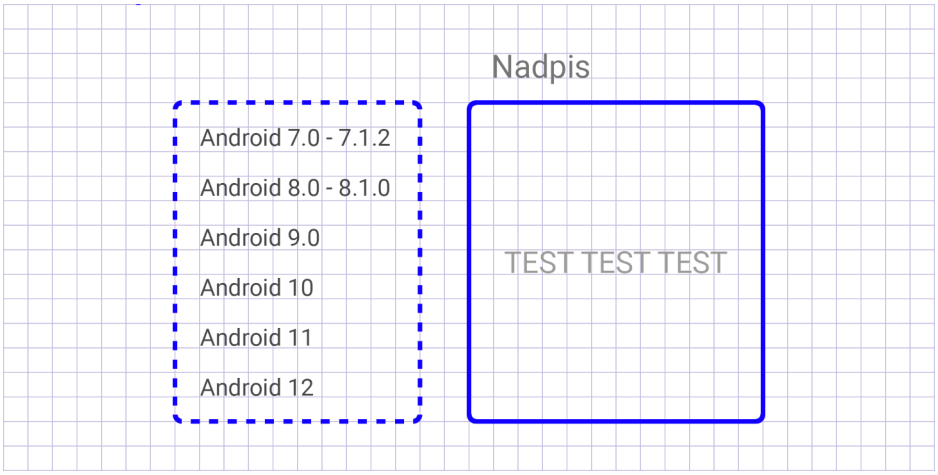
Navigaci pomocí menu Drawer lze relativně snadno vygenerovat s novým projektem, protože Android Studio obsahuje šablonu Navigation Drawer Activity v průvodci, jako jednu z možností pro vygenerování kostry aktivity. Takto vygenerované menu je poměrně robustní, univerzálně navržené, využívá fragmentů a dá se pomocí modifikací skutečně dosáhnout profesionálně vyladěné navigace. Pro jednodušší aplikace je však příliš komplikované a rozsáhlé, takže nezřídka bývá vhodnější vytvořit si vlastní verzi, která obsahuje jenom to, co je skutečně v aplikaci potřeba mít.

17.4.1 Vysouvací menu, 1. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 17_Menu
projekt MenuDrawer1

Zkusíme si udělat aplikaci, ve které zprovozníme menu s naprostým minimem kódu. Základem bude kontejner Drawer Layout, který může obsahovat dvě vnořené komponenty, z nichž jedna je formulář aplikace bez zobrazeného menu a druhou komponentou je samotné menu. Jelikož si se dvěma komponentami většinou nevystačíme, používají se místo nich buď formuláře nebo fragmenty, které už můžou obsahovat vše, na co jsme zvyklí.

Jak bude aplikace vypadat? Na základní obrazovce nebude nic zvláštního, pouze doprostřed formuláře umístíme libovolný text. Po přejetí prstem na displeji zleva doprava se vysune definované menu, které bude zobrazovat seznam starších verzí Androidu. Kliknutím na řádek v takto otevřeném menu už jenom zobrazíme zprávu na displeji. Menu zavřeme přejetím prstu po displeji v opačném směru.



Obrázek 17.4: Návrh formuláře s kontejnerem Drawer Layout

Co je potřeba udělat?

- přidat do projektu knihovnu Design Support,
- vytvořit formulář s kontejnerem DrawerLayout
- do kontejneru DrawerLayout umístit další kontejner, který bude obsahovat vzhled formuláře
- do kontejneru DrawerLayout umístit RecyclerView, který bude mít funkci menu
- do RecyclerView načíst položky menu, tj. seznam starších verzí Androidu
- zajistit zobrazování menu

Formulář s DrawerLayout kontejnerem můžeme udělat rovnou v activity_main.xml souboru. Samotný kontejner pravděpodobně nenajdeme v paletě komponent a budeme ho muset přidat do xml kódu. Není to naštěstí nijak složité, prostě jím nahradíme původní kontejner, který Android Studio do souboru vygenerovalo. Prázdný kontejner může vypadat nějak takto:

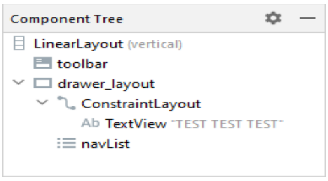
```
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

</androidx.drawerlayout.widget.DrawerLayout>
```

Do kontejneru DrawerLayout vložíme další kontejner, který umožní vytvořit vzhled aplikace. Řekněme že použijeme ConstraintLayout. A do něj umístíme libovolný obsah, v našem případě TextView s textem “TEST TEST TEST”. Toto všechno můžeme udělat i v grafickém editoru formulářů.

Kdybychom přidávali DrawerLayout do hotového projektu, tak bychom postupovali obdobně. Do formuláře bychom vložili DrawerLayout a až do něj umístili původní obsah.

Poté do DrawerLayoutu přidáme komponentu RecyclerView, která se postará o zobrazení jednotlivých položek menu. U RecyclerView vhodně nastavíme atribut barvy pozadí android:background, protože právě on rozhodne o tom, jakou barvu a případně průhlednost bude menu mít. Pokud použijeme hodnotu #aaffffff, tak říkáme, že má být menu bílé (R=0xFF, G=0xFF B=0xFF), s průhledností 170 (alfa kanál 0xAA, neboli dekadicky 170).

V hierarchické struktuře objektů v okně Component Tree, obrázek 17.5, je vidět vnoření prvků do kontejneru DrawerLayout.



Obrázek 17.5: Hierarchie objektů v layoutu, zobrazená v okně Component Tree

Výsledný XML kód bude vypadat přibližně takto:

```
<!-- DrawerLayout roztazeny pres cely formular -->
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- ConstraintLayout pro zobrazeni prvku formulare -->
    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <!-- textove pole -->
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="TEST TEST TEST"
            android:textSize="30sp"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>

    <!-- RecyclerView ktery tvori menu -->
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/navList"
        android:layout_width="200dp"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:background="#aaffffff"/>

</androidx.drawerlayout.widget.DrawerLayout>
```

Komponenta RecyclerView by se dala nahradit formulářem nebo fragmentem, pro naše účely bude ale zatím stačit, když položky menu zobrazí seznam RecyclerView.

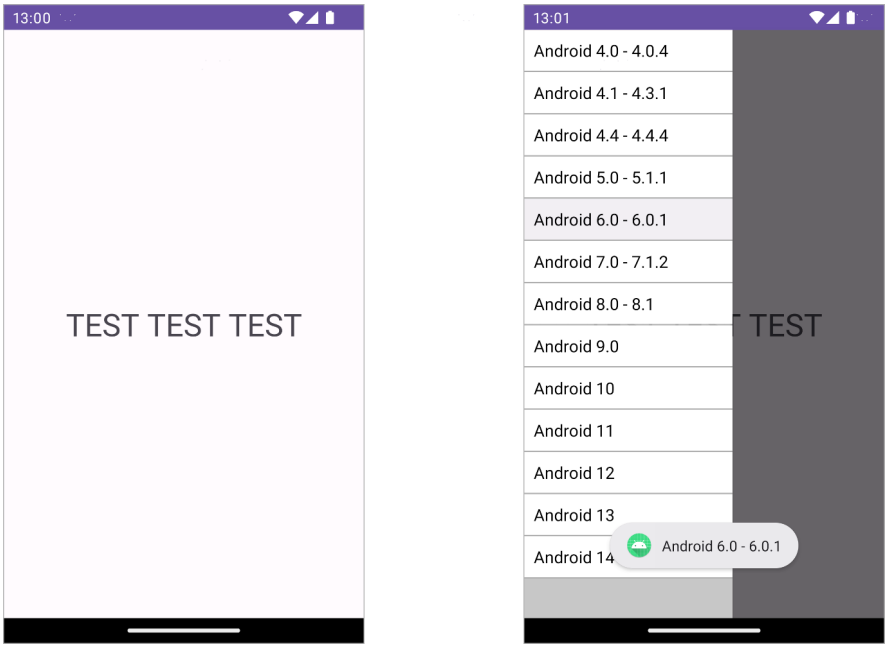
Aby bylo možné menu zobrazit, je potřeba přidat RecyclerAdapter, RecyclerHolder, položku menu, soubor itemcolors a udělat několik úprav v souboru MainActivity.java. Nové soubory budou identické s prvním příkladem komponenty RecyclerView. Jejich funkce bude shodná, prostě je bez úprav převezmeme tak, jak jsou.

V MainActivity je nutné načíst data do RecyclerView. Pro tento účel lze s výhodou použít metodu naplnitRecyclerView(), kterou jsme si připravili v kapitole o RecyclerView. Tuto metodu není potřeba nijak výrazně měnit, pouze v ní nahradíme ID samotného RecyclerView. Metoda načte data do komponenty a zajistí, aby po ťuknutí na některou z položek menu byla vypsána příslušná informace na displej. Metodu pouze zavoláme z události onCreate() třídy MainActivity.

Celá třída MainActivity bude vypadat takto:

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        naplnitRecyclerView();
    }

    private void naplnitRecyclerView()
    {
        ...
    }
}
```

Výsledek by měl na emulátoru nebo koncovém zařízení vypadat jako na obrázku 17.6.



Obrázek 17.6: Aplikace s kontejnerem typu Drawer Layout, spuštěná na emulátoru

17.4.2 Vysouvací menu, 2. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 17_Menu
projekt MenuDrawer2

Předchozí příklad ukázal, jak je možné udělat Drawer menu bez toho, aby bylo potřeba psát nějaký nový kód. Menu je v té nejjednodušší formě, nemá záhlaví s ikonkou a reaguje pouze na posunutí prstem po displeji do strany. Toto se teď pokusíme vylepšit. K menu připojíme záhlaví, ikonu, takzvaný „hamburger“, která bude reagovat na kliknutí a přidáme událost pro detekci otevření a zavření menu. Pomocí ní budeme měnit nadpis v záhlaví formuláře. Jako základ použijeme předchozí příklad, pouze ho rozšíříme o novou funkcionalitu.

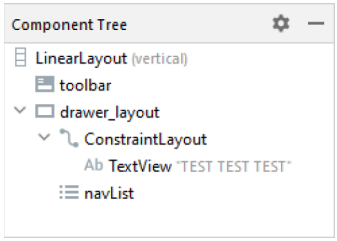
Co je potřeba udělat?

- přidat do formuláře záhlaví Toolbar

- napsat metodu, která nastaví vlastnosti Drawer menu

- přidat události pro obsluhu tlačítka

Zasazení záhlaví typu Toolbar, formulář trochu zkomplikuje. Vzhled formuláře může zůstat tak, jak jsme ho připravili v minulém příkladu, ale je ho potřeba obalit vhodným typem formuláře, do kterého půjde přidat záhlaví. Relativně snadným řešením je použít vertikální Linear Layout, do něj vložit nejdřív záhlaví Toolbar (postup je stejný jako v příkladu 12.5 Záhlaví formuláře) a pod něj celé Drawer Menu z minulého příkladu. V okně Component Tree bude vnoření objektů jako na obrázku 17.7.



Obrázek 17.7: Hierarchie objektů v layoutu, zobrazená v okně Component Tree

Následně napíšeme metodu, která umožní přepínání nadpisu a podoby ikonky při zavřeném, resp. otevřeném menu. Ikonka má při zavřeném menu podobu tří čárek nad sebou, při otevřeném zobrazuje šipku. Nejprve zajistíme přepínání nadpisu.

```
private ActionBarDrawerToggle drawerToggle;
private DrawerLayout drawerLayout;

private void nastavitDrawer () {
    drawerToggle = new ActionBarDrawerToggle(this, drawerLayout,
                            R.string.drawer_otevrit, R.string.drawer_zavrit) {

        // tato udalost se zavola jakmile se menu otevre
        public void onDrawerOpened(View drawerView) {
            super.onDrawerOpened(drawerView);
            getSupportActionBar().setTitle("Verze Androidu");
        }
```

```
        // tato udalost se zavola jakmile se menu zavre
        public void onDrawerClosed(View view) {
            super.onDrawerClosed(view);
            getSupportActionBar().setTitle(getTitle().toString());
        }
    };

    // nastaveni prepinace a udalosti
    drawerToggle.setDrawerIndicatorEnabled(true);
    drawerLayout.addDrawerListener(drawerToggle);
}
```

Metoda vytvoří instanci třídy ActionBarDrawerToggle(), určené k synchronizaci změny stavu Draweru s prvky v záhlaví. Metoda sama obsahuje dvě zpětně volané metody, onDrawerOpened() a onDrawerClosed(), které mohou obsahovat kód pro vyvolání dalších akcí. My pouze změníme text v záhlaví na „Verze Androidu“, pokud je menu otevřené a zpět vrátíme název aplikace, jakmile se menu zase zavře. Poslední dva řádky metody už jenom událost zaregistrují a aktivují.

Do události onCreate() přidáme nastavení Toolbaru, volání metody nastavitDrawer() a můžeme aplikaci spustit. Od předchozího příkladu se bude lišit pouze změnou nadpisu v záhlaví při otevírání a zavírání menu.

Pokud vše funguje, přidáme do události onCreate() instrukce setDisplayHomeAsUpEnabled() a setHomeButtonEnabled() pro nastavení tlačítka v záhlaví aplikace. Událost bude nyní vypadat takto:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // nastaveni Toolbaru
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    drawerLayout = (DrawerLayout)findViewById(R.id.drawer_layout);

    naplnitRecyclerView();
    nastavitDrawer();

    // nastaveni tlacitka
    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    getSupportActionBar().setHomeButtonEnabled(true);
}
```

To ale nestačí. Aby se tlačítko bylo schopné správně překreslovat, musíme ve třídě MainActivity ještě předefinovat události onPostCreate() a onOptionsItemSelected(). Nejprve onPostCreate():

```
@Override
protected void onPostCreate(Bundle savedInstanceState) {
    super.onPostCreate(savedInstanceState);
    drawerToggle.syncState();
}
```

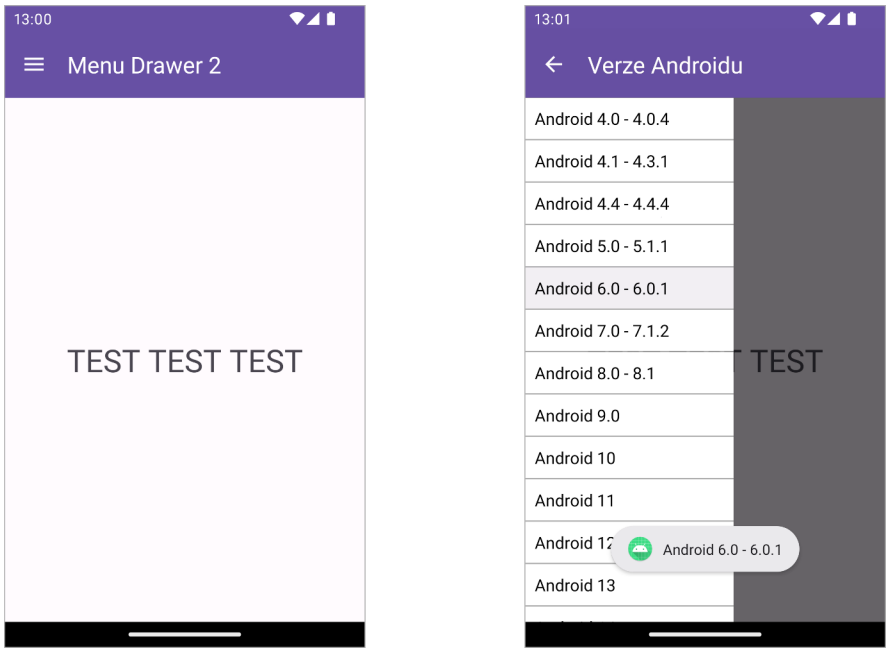
Událost zavolá metodu syncState(), která se postará o zobrazení tlačítka a jeho animaci, synchronizovanou na stav menu. Událost onPostCreate() volíme proto, že je volaná po ukončení události onStart() v životním cyklu aktivity. Jde o speciální událost, která byla v kapitole o životním cyklu zmíněna pouze okrajově.

Pokud teď aplikaci spustíme, tlačítko se objeví, bude animované a bude reagovat na otevření a zavření menu. Stále na něj ale nepůjde klikat. To se změní až s předefinováním události onOptionsItemSelected(). Opět jde o speciální událost, která je volaná, jakmile dojde ke kliknutí do záhlaví aktivity.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (drawerToggle.onOptionsItemSelected(item)) {
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Událost nejdřív zavolá metodu onOptionsItemSelected(), která zajistí otevření menu. Pokud volaná metoda vrátí true, můžeme událost ukončit se stejnou návratovou hodnotou. V opačném případě je ještě vhodné zavolat původní onOptionsItemSelected() z nadřazené třídy.

Výsledek by měl na emulátoru nebo koncovém zařízení vypadat jako na obrázku 17.8.



Obrázek 17.8: Aplikace s kontejnerem typu Drawer Layout, rozšířená o ikonu a titulek, spuštěná na emulátoru

17.4.3 Vysouvací menu, 3. příklad

V předchozím příkladu jsme sestavili drawer navigaci s dynamicky generovaným menu. Tím ale možnosti komponenty DrawerLayout zdaleka nekončí. Pokusíme se navigaci dále vylepšit.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 17_Menu
projekt MenuDrawer3

Změníme dynamicky generované menu na statické a přidáme záhlaví menu. To všechno opět na příkladu aplikace, která bude poskytovat informace o verzích systému Android. Vyjdeme z předchozího příkladu, který můžeme dále rozšiřovat, nebo si od něj pořídíme kopii a budeme pracovat na ní.

Co je potřeba udělat?

- vytvořit design menu a v něm definovat jednotlivé položky menu
- vytvořit formulář se záhlavím menu
- zrušit generování dynamického menu
- přidat do formuláře komponentu, která umožní zobrazit statické menu a záhlaví
- napsat událost, které bude reagovat na kliknutí na položku statického menu

Design menu připravíme, stejně jako v příkladech jednoduchých menu (vyskakovací, kontextové menu) tak, že si založíme soubor menu v projektovém adresáři app/res/menu/ a nazveme ho například menu_drawer.xml. Na rozdíl od zmíněných příkladů ho ale uděláme strukturovanější a aby vypadalo profesionálně, přidáme do něj ikony. Ikony si připravíme do složky app/res/drawable – pro začátek stačí jedna, kterou použijeme u všech položek menu. Do menu vložíme jednotlivé položky tak, že vždy uděláme jednu s názvem verze Androidu a do ní umístíme vnořené položky, reprezentující jednotlivé číselné verze. Do atributů každé položky připojíme připravenou ikonku a nezapomeneme položky unikátně pojmenovat. Podle jejich id budeme později rozlišovat, na kterou z nich uživatel klikl. Výsledný XML soubor bude vypadat zhruba takto:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:title="Nougat">
        <menu>
            <item
                android:id="@+id/nav_70"
                android:icon="@drawable/logo_android_male"
                android:title="Nougat 7.0" />
            <item
                android:id="@+id/nav_71"
                android:icon="@drawable/logo_android_male"
                android:title="Nougat 7.1" />
        </menu>
    </item>

    <item android:title="Oreo">
        <menu>
            <item
                android:id="@+id/nav_80"
                android:icon="@drawable/logo_android_male"
                android:title="Oreo 8.0" />
            <item
                android:id="@+id/nav_81"
                android:icon="@drawable/logo_android_male"
                android:title="Oreo 8.1" />
        </menu>
    </item>
```

```
    ...
</menu>
```

Takto sestavené menu si pouze zkontrolujeme v designéru a zatím ho necháme nevyužité, vrátíme se k němu hned po následujícím kroku. Tím je sestavení záhlaví menu. Záhlavím se rozumí oblast nad samotným menu, které bývá zpravidla podbarvené nebo tvořené obrázkem či barevným přechodem. V této oblasti jsou uvedené základní informace, případně obsahuje tematickou ikonku.

Záhlaví umístíme do samostatného xml souboru typu layout. Ten založíme v app/res/layout a nazveme ho například menu_drawer.xml. Aby záhlaví vypadalo elegantně, dáme mu do pozadí jednoduchý barevný přechod z tmavě šedé v levém dolním rohu do světle šedé v pravém horním rohu.

Přechod nemusíme kreslit v externím grafickém editoru, stačí když ho pomocí xml parametricky popíšeme. Pro tyto účely si založíme ještě xml soubor v adresáři app/res/drawable, s názvem menu_drawer_pozadí.xml a v něm přechod nadefinujeme jako lineární gradient pootočený v úhlu 45 stupňů (to proto, aby měl směr z levého dolního do pravého horního rohu), začínající tmavým a končící světlým odstínem šedé:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
                                                android:shape="rectangle">
<gradient
    android:startColor="#929292"
    android:endColor="#d1d1d1"
    android:type="linear"
    android:angle="45" />
</shape>
```

Ještě než se vrátíme k definici záhlaví, připravíme si ikonku která se hodí k tématu menu či aplikace. V našem případě to bude logo Androidu, které také uložíme do adresáře app/res/drawable.

Pro záhlaví zvolíme vhodný typ formuláře (například constraint layout), jako barvu pozadí nastavíme barevný přechod a dovnitř umístíme ikonku a textová pole s nezbytnými informacemi. Soubor bude vypadat přibližně takto (pole typu TextView jsou v kódu, pro přehlednost, sbalena):

```
<!-- constraint layout s barevnym prechodem na pozadi -->
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:background="@drawable/menu_drawer_pozadi"
    android:layout_width="match_parent"
    android:layout_height="145dp">

    <!-- ikona Android -->
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="97dp"
        android:layout_height="100dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:contentDescription="Android"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:srcCompat="@drawable/logo_android" />
```

```
    <!-- textove pole s nadpisem Verze Androidu -->
    <TextView...>

    <!-- textove pole s informaci o verzich -->
    <TextView...>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Máme tedy připravené jak menu, tak záhlaví. A můžeme dynamicky generované menu nahradit statickým. Ve třídě MainActivity smažeme metodu pro generování menu naplnitRecyclerView() a její volání v události onCreate(). Už je nebudeme potřebovat. Stejně tak můžeme smazat třídy Recycleradapter, RecyclerHolder a formulář polozka.xml.

Teď upravíme vzhled základního formuláře v souboru activity_main.xml. Komponentu RecyclerView už také nebudeme potřebovat. Namísto ní použijeme novou komponentu, NavigationView. Ta se používá téměř výhradně pro účely drawer menu a má v zásadě jediný účel - sloučit menu se záhlavím. Komponentu vložíme do formuláře buď ve formě xml, nebo v designeru z palety komponent. Nastavíme jí atributy headrerLayout a menu, kterými se odkážeme na xml soubory se záhlavím a designem menu.

```
<com.google.android.material.navigation.NavigationView
    android:id="@+id/nav_view"
    android:layout_width="255dp"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:fitsSystemWindows="true"
    app:headerLayout="@layout/zahlavi_drawer"
    app:menu="@menu/menu_drawer" />
```

Kromě záměny RecyclerView za NavigationView nic dalšího v souboru activity_main.xml měnit nepotřebujeme.

Poslední změny nás čekají ve třídě MainActivity. Musíme ji rozšířit o implementaci rozhraní posluchače NavigationView.OnNavigationItemSelectedListener, který dává možnost reagovat na kliknutí na položku menu:

```
public class MainActivity extends AppCompatActivity
        implements NavigationView.OnNavigationItemSelectedListener {
```

Potom přepíšeme událost onNavigationItemSelected(). Nastavíme, aby prozatím pouze zobrazila toast zprávu, potom rozlišíme o kterou položku menu jde a připravíme si kód pro její zpracování. Nakonec drawer menu zavřeme.

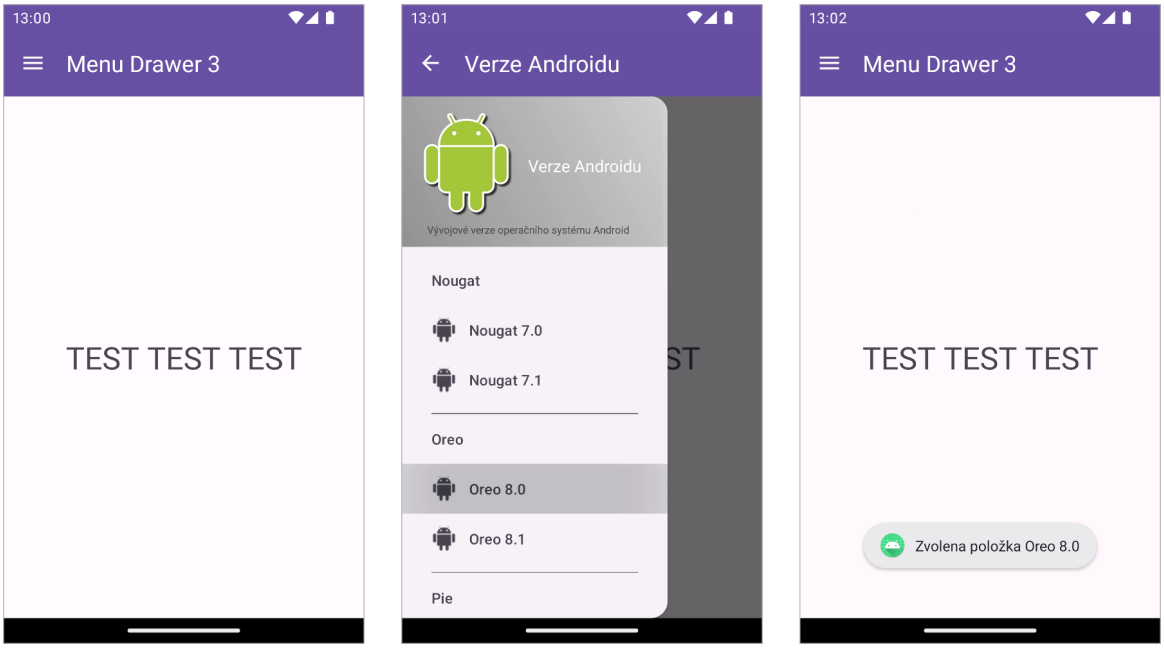
```
@Override
public boolean onNavigationItemSelected(MenuItem item) {
    // zobrazi text na displeji
    Toast.makeText(MainActivity.this, "Zvolena položka " + item.getTitle(),
                                                Toast.LENGTH_SHORT).show();

    // zjistime, ktera polozka byla zvolena
    int id = item.getItemId();
    if (id == R.id.nav_60) {
        // akce po volbe Androidu verze 6.0
    } else if (id == R.id.nav_70) {
```

```
        // akce po volbe Androidu verze 7.0
    }

    ...


    // zavrit drawer menu
    DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
    drawer.closeDrawer(GravityCompat.START);

    return true;
}
```

Aby posluchač události fungoval, musíme ho zaregistrovat. Což v praxi znamená, že přidáme do události onCreate() tyto dva řádky, z nichž první najde ve formuláři komponentu NavigationView a druhý ji přiřadí samotný listener:

```
NavigationView navigationView = (NavigationView) findViewById(R.id.nav_view);
navigationView.setNavigationItemSelectedListener(this);
```

Po kompilaci a spuštění na na emulátoru nebo koncovém zařízení, by měl výsledek vypadat jako na obrázku 17.9.



Obrázek 17.9: Aplikace s kontejnerem typu Drawer Layout, rozšířená o záhlaví, spuštěná na emulátoru

18 Ukládání dat

Při práci se systémem Android je často potřeba ukládat data, ke kterým se dá později vrátit a dále s nimi pracovat. Nejjednodušším příkladem je uložení nastavení aplikace a po opětovném spuštění její nastavení do původního stavu. Takto si lze zapamatovat prakticky cokoliv, co uživatel během práce s aplikací nastavil.

Android umožňuje uchování a následné čtení dat několika způsoby:

Sdílená nastavení, Shared Preferences – nejsnazší způsob uložení do souboru, omezený na kombinaci klíč a hodnota

Vnitřní úložiště, Internal Storage – vnitřní úložiště, data ukládaná do souboru umístěném v paměti zařízení

Externí úložiště, External Storage – vnější úložiště, data ukládaná do souboru umístěném do paměti, kterou je možné ze zařízení odstranit

Veřejně přístupné adresáře, Public Folders – může být jak externí tak interní, konkrétní místo si volí systém sám, aplikace pouze systému sdělí, s jakou složkou chce pracovat

Dočasné soubory, Cache – také mohou být interní i externí, slouží ke krátkodobému uložení malého množství dat

Soubory projektu, res, assets – soubory jsou součástí aplikace a dají se pouze číst

Databáze – komplexní uložení dat do SQL databáze, dostupné pouze z aplikace

Z hlediska systému Android existují pouze dvě úložiště – paměť zařízení a připojená karta. Je to dané historicky a přestože je dnes možné připojit i další úložiště, Android se stále chová jako by existovala pouze tato dvě.

Základní rozdíly mezi vnitřním a vnějším úložištěm:

Vnitřní úložiště:

- úložiště je vždy dostupné

- soubory, které aplikace umístí do interního úložiště, jsou přístupné pouze pro tuto aplikaci

- odinstalování aplikace způsobí smazání všech datových souborů aplikace

- interní úložiště využijeme zejména tehdy, když potřebujeme zajistit aby měla naše aplikace spolehlivý přístup ke svým datům

Vnější úložiště:

- není vždy dostupné, uživatel může toto úložiště odpojit nebo vyměnit za jiné

- uložená data jsou dostupná z vnějšku, takže nemáme spolehlivou kontrolu nad tím, kdo s nimi pracujete

- když uživatel aplikaci odinstaluje, smažou se pouze ta data na externím úložišti, která se nachází v adresáři určeném pro aplikaci; cestu lze zjistit pomocí metody getExternalFilesDir()

- externí úložiště je vhodné použít tehdy, když nezáleží na omezení přístupu k datům nebo když data chceme s ostatními aplikacemi sdílet

18.1 Sdílená nastavení, Shared Preferences

Tento způsob uložení dat připomíná zjednodušenou formu systémových registrů Windows. Používá se pro uchovávání malého množství dat, ukládá se vždy pouze klíč a k němu přiřazená hodnota. Tato data se ukládají do datového souboru ve formátu XML.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 18_Ukladani_dat
projekt SharedPreferences

Zápis i čtení jsou poměrně jednoduché operace. Nejprve je nutné otevřít soubor pro zápis sdílených referencí:

```
SharedPreferences sharedPreferences = this.getPreferences(Context.MODE_PRIVATE);
```

Parametr Context.MODE_PRIVATE říká, že půjde o zápis do souboru, který bude dostupný pouze pro naši aplikaci. Dříve bylo možné použít i parametr který soubor zpřístupnil dalším aplikacím, ale toto platilo pouze do API verze 17. V novějších byla tato alternativa z bezpečnostních důvodů zakázána, takže dnes ani jiná možnost než MODE_PRIVATE nedává smysl.

Jelikož se data zapisují až za běhu aplikace, není soubor uložený ve struktuře projektu, ale objeví se až v cílovém zařízení na místě: /data/user/0/*package*/databases/MyPrefsFile.xml

Package je název balíčku aplikace. Na emulátoru se dají data prohlédnout v okně „Device File Explorer“, které je pod záložkou na pravém okraji Android Studia, nebo v menu View → Tools Windows → . Na většině fyzických zařízení jsou systémové soubory znepřístupněny, takže tam si soubor nebude možné prohlédnout. V takovém případě nezbývá než věřit, že tam skutečně je. Umístění si však můžeme zjistit pomocí metody getDatabasePatch(), které v parametru předáme název souboru, složený z názvu aktivity a přípony .xml:

```
File f = getDatabasePath("NazevAktivity.xml");
String umisteniSouboru = f.getAbsolutePath();
```

Nestačí-li nám jeden společný soubor pro celou aplikaci, můžeme si jich vytvořit víc, maximálně však jeden pro každou aktivitu. V takovém případě bychom k souboru přistupovali takto:

```
SharedPreferences sharedPreferences = getSharedPreferences(nazev.xml,
      Context.MODE_PRIVATE);
```

Data budou uložena do souboru /data/user/0/*package*/databases/*nazev.xml*

Ukládání dat se dělá prostřednictvím objektu Editor. Ten se vytvoří na instanci preferencí a pomocí instrukce put se plní daty. Instrukce put má několik podob, pro různé datové typy: putString, putInt, putBoolean atd. Nakonec se celý objekt příkazem apply() uloží:

```
SharedPreferences.Editor editor = sharedPreferences.edit();
editor.putString("klic", "hodnota");
editor.apply();
```

Tím jsme uložili text „hodnota“ do klíče pojmenovaného „klic“. Metoda apply() uloží data okamžitě. Pokud by bylo potřeba data uložit asynchronně, je místo ní možné použít volání commit().

Načtení dat se provádí už jen voláním metody get daného datového typu:

```
String ulozenyText = sharedPreferences.getString("klic", "default");
```

Druhým parametrem metody get je defaultní hodnota, kterou vrátí v případě, že není klíč nalezen.

Jako příklad si uděláme jednoduchou aplikaci, která bude obsahovat formulář se dvěma textovými poli, jejichž obsah budeme pomocí tlačítka ukládat do souboru preferencí a pomocí druhého tlačítka je budeme načítat zpět do formuláře.

Co je potřeba udělat?

- vytvořit formulář se dvěma textovými poli a dvěma tlačítky

- napsat programový kód, který data uloží a zase načte

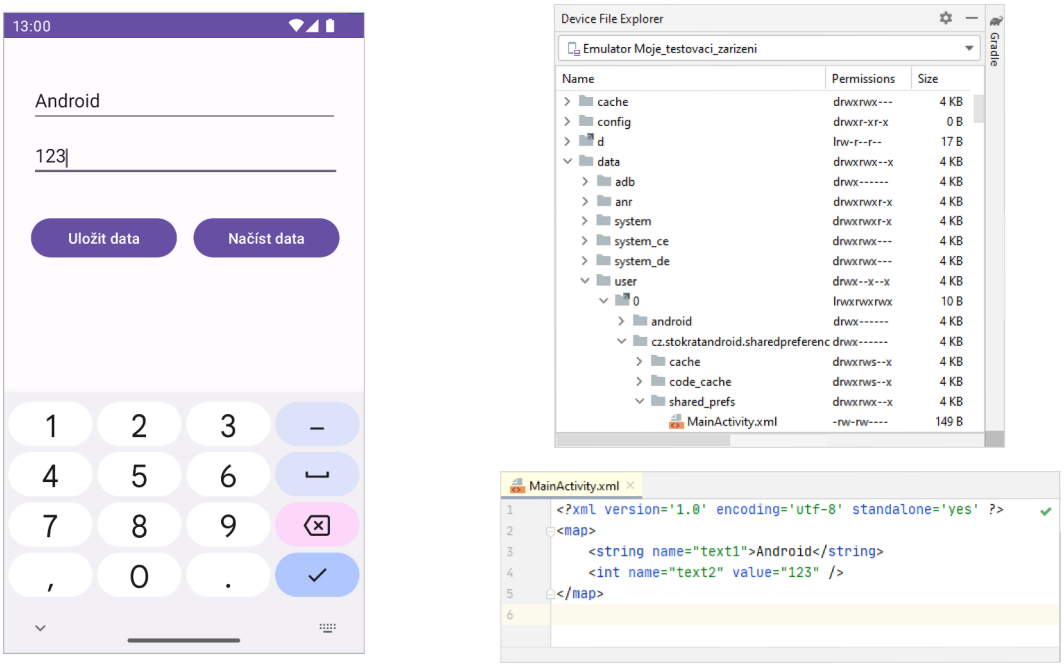
Sestavit formulář je jednoduché, použijeme například constraint layout, textová pole a tlačítka do něj umístíme na vhodné pozice. První textové pole “editText1” bude pracovat s textem, druhé “editText2” bude numerické. Tlačítka budou volat metody nacistData() a ulozitData().

V MainActivity.java vytvoříme metody nacistData() a ulozitData() a z události onLoad() metodu nacistData() zavoláme. Tím dosáhneme toho, že při spuštění aplikace budou ve formuláři vyplněna data která jsme naposledy uložili.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        nacistData(null);
    }

    public void ulozitData(View sender)
    {
        // nacist hodnoty z formulare
        EditText editText1 = (EditText)findViewById(R.id.editText1);
        String inputText = editText1.getText().toString();

        EditText editText2 = (EditText)findViewById(R.id.editText2);
        String text2 = editText2.getText().toString();
        int inputInt = Integer.parseInt(text2);

        // otevrit soubor pro zapis preferenci
        SharedPreferences sharedPreferences =
            this.getPreferences(Context.MODE_PRIVATE);

        // vytvorit objekt editor preferenci
        SharedPreferences.Editor editor = sharedPreferences.edit();

        // ulozit do editoru klic a hodnotu z formulare
        editor.putString("text1", inputText);
        editor.putInt("text2", inputInt);
```

```
        // ulozit data
        editor.apply();
    }

    public void nacistData(View sender)
    {
        // otevrit soubor pro cteni preferenci
        SharedPreferences sharedPreferences =
            this.getPreferences(Context.MODE_PRIVATE);

        // nacist ulozenou hodnotu
        String text1 = sharedPreferences.getString("text1", "default");
        String text2 = Integer.toString(sharedPreferences.getInt("text2", 0));

        // ulozit hodnoty do formulare
        EditText editText1 = (EditText)findViewById(R.id.editText1);
        editText1.setText(text1);

        EditText editText2 = (EditText)findViewById(R.id.editText2);
        editText2.setText(text2);
    }
}
```

V emulátoru nebo na koncovém zařízení by měla aplikace a uložená data vypadat jako na obrázku 18.1.



Obrázek 18.1: Aplikace pracující s daty v úložišti Shared Preferences

18.2 Vnitřní úložiště, Internal Storage

Internal Storage neboli interní úložiště je způsob uložení soukromých dat v paměti zařízení. Soukromými daty se myslí soubory, které patří pouze aplikaci a nikdo

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 18_Ukladani_dat
projekt InternalStorage

další k nim nemá přístup. Data nelze tímto způsobem ukládat kamkoliv, ale pouze do prostoru vyhrazeného aplikaci. Se smazáním aplikace jsou ze zařízení pochopitelně vymazána i tato data. Ve starších verzích Androidu bylo možné datové soubory sdílet i s jinými aplikacemi, ale od API verze 17 je tato praxe z bezpečnostních důvodů zakázána.

Aby bylo možné data zapsat, je nutné nejdřív otevřít soubor pro zápis, zavoláním metody openFileOutput(). Metoda vrací výstupní stream a má dva parametry – název otevíraného souboru a způsob přístupu. Součástí názvu není cesta k souboru, ta je daná systémovým nastavením, soubor se ukládá do úložiště vyhrazeného aplikaci. Můžeme si v něm ale vytvořit adresářovou strukturu a na soubory se potom odkazovat relativně v rámci této struktury. Způsoby přístupu připadají v úvahu v zásadě pouze dva:

MODE_PRIVATE – vytvoří nový, nebo přepíše existující soubor

MODE_APPEND – vytvoří nový, nebo připojí výstup na konec existujícího souboru

Do API verze 17 byly k dispozici ještě další parametry, umožňující zápis souboru, ke kterému měly přístup i další aplikace. Otevření souboru:

```
FileOutputStream outStream = openFileOutput(nazevSouboru, Context.MODE_PRIVATE);
```

Data zapisujeme pomocí instrukce write výstupního streamu, jejímž parametrem je zapisovaný text převedený do formátu pole bajtů:

```
outStream.write(inputText.getBytes());
```

Po takto předaných datech zbývá už jenom stream uzavřít instrukcí close a data jsou do souboru zapsána:

```
outStream.close();
```

Jelikož jde o vstupně-výstupní operaci při které může dojít k selhání, musí být volání openFileOutput() obaleno blokem try – catch, který zajistí zpracování v případě výjimky. K té může dojít například pokud není v paměti místo pro nový soubor, pokud má soubor stejné jméno jako již existující adresář atd. Obvykle se blokem try – catch neobaluje pouze operace otevření streamu, ale všechny instrukce od otevření až po jeho uzavření.

Čtení má stejná základní pravidla. Musíme otevřít vstupní stream, k čemuž poslouží volání metody openFileInput() s názvem souboru předaným v parametru:

```
FileInputStream inStream = openFileInput(NAZEV_SOUBORU);
```

Data se načítají pomocí readeru do bufferu, takže si vytvoříme jejich instance:

```
InputStreamReader reader = new InputStreamReader(inStream);
BufferedReader bufReader = new BufferedReader(reader);
```

Samotné čtení může proběhnout po řádcích pomocí volání metody bufReader.readLine(). V cyklu takto můžeme snadno načíst celý obsah souboru. Tímto způsobem můžeme však číst pouze textové soubory.

```
while ((radekTextu = bufReader.readLine()) != null) {
    nactenyText.append(radekTextu + "\n");
}
```

Druhou možností je načíst soubor po znacích. Tento postup je vhodný pro binární soubory, ale můžeme s ním načíst pochopitelně i text. Není však tak rychlý, jako načítání po řádcích.

```
while ((znak = inStream.read()) != -1) {
    nactenyText.append((char)znak);
}
```

Abychom zjistili, kam se soubory ukládají, použijeme metodu getFilesDir() z jejíhož výstupu získáme cestu do adresáře aplikace:

```
String umisteni = getFilesDir().getAbsolutePath();
```

Jako příklad si opět uděláme jednoduchou aplikaci, která bude obsahovat formulář s textovým polem a dvěma tlačítky, z nichž jedno bude obsah textového pole ukládat do souboru test.txt a druhé naopak obsah souboru přečte a vloží ho do textového pole. Při zápisu navíc zobrazíme umístění textového souboru v paměti zařízení.

Co je potřeba udělat?

- vytvořit formulář s víceřádkovým textovým polem a dvěma tlačítky

- napsat kód v jazyce Java, který data uloží a zase načte

Formulář obsahuje pouze známé základní prvky - textové pole a tlačítka. Použijeme například constraint layout, do kterého textové pole a tlačítka umístíme na vhodné pozice. Tlačítka budou volat metody nacistData() a ulozitData().

V MainActivity.java si vytvoříme metodu ulozitData() pro uložení dat do souboru ve vnitřním úložišti:

```
public void ulozitData(View sender)
{
    // nacist hodnoty z formulare
    EditText editText1 = (EditText)findViewById(R.id.editText1);
    String inputText = editText1.getText().toString();

    String NAZEV_SOUBORU = "test.txt";
    try {
        // otevrit soubor pro zapis
        FileOutputStream outStream = openFileOutput(NAZEV_SOUBORU,
                                        Context.MODE_PRIVATE);

        // zapsat text
        outStream.write(inputText.getBytes());

        // uzavrit soubor
        outStream.close();
    } catch (Exception e) {
        // zobrazit text, v pripade chyby
```

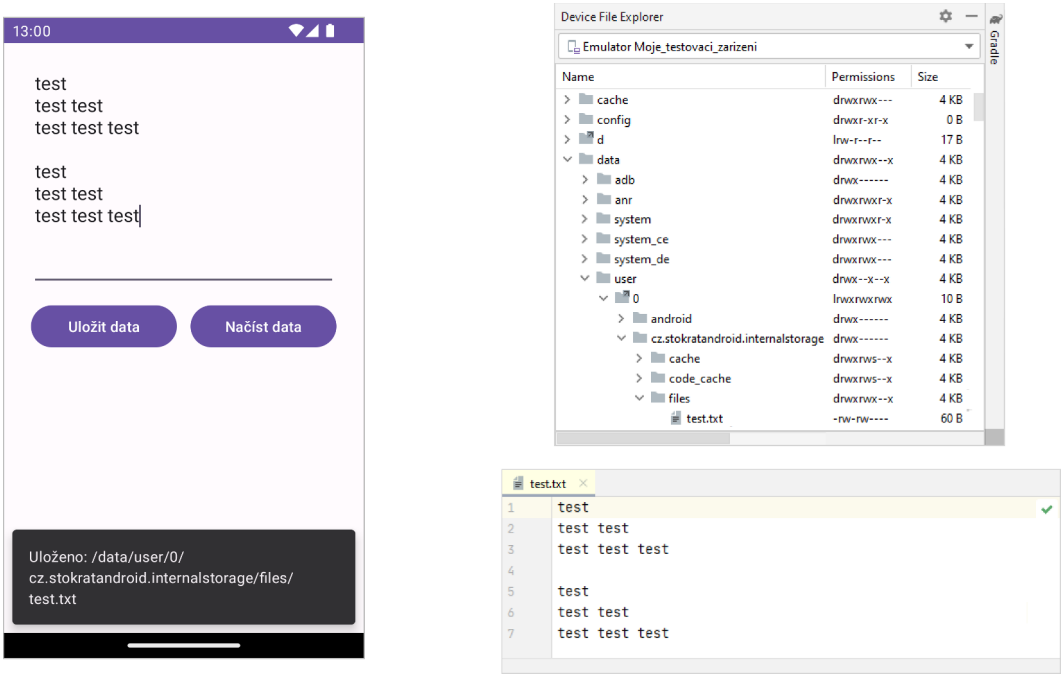
```
            Toast.makeText(MainActivity.this, "Nedaří se zapsat..." +
                           e.getMessage(), Toast.LENGTH_SHORT).show();
            return;
        }

        // zjistit misto ulozeni souboru
        String umisteni = "Uloženo: " + getFilesDir().getAbsolutePath() +
                          "/" + NAZEV_SOUBORU;

        // pro informaci zobrazit umisteni souboru
        Snackbar.make(findViewById(android.R.id.content), umisteni,
                          Snackbar.LENGTH_LONG).setTextMaxLines(5).show();
}
```

Metoda vyzvedne data z textových polí, vytvoří stream pro zápis a jeho prostřednictvím zapíše data do souboru. Stream následně uzavře. Nakonec zjistí umístění dat a pomocí Snackbar zprávy informuje uživatele.

Kromě zápisu do souboru potřebujeme i jeho čtení, takže napíšeme metodu nacistData(), která data obdobným způsobem ze souboru zase načte. Využije k tomu postupné načítání dat po jednotlivých znacích, výsledek doplní do polí na formuláři:

```
public void nacistData(View sender)
{
    String NAZEV_SOUBORU = "test.txt";
    try {
        // otevrit soubor pro cteni
        FileInputStream inStream = openFileInput(NAZEV_SOUBORU);

        // vytvorit instanci bufferu
        StringBuffer nactenyText = new StringBuffer();

        // soubor postupne, po znacich, nacist do bufferu
        int znak;
        while ((znak = inStream.read()) != -1) {
            nactenyText.append((char)znak);
        }

        // ulozit text do formulare
        EditText editText1 = (EditText)findViewById(R.id.editText1);
        editText1.setText(nactenyText);
    }
    catch (IOException e) {
        // zobrazit text, v pripade chyby
        String textChyby = "Nedaří se načíst.. \n" + e.getMessage();
        Toast.makeText(MainActivity.this, textChyby, Toast.LENGTH_SHORT).show();
    }
}
```

V emulátoru nebo na koncovém zařízení by měla aplikace a uložená data vypadat jako na obrázku 18.2.



Obrázek 18.2: Aplikace pracující s daty v úložišti Internal Storage, spuštěná na emulátoru

18.3 Externí úložiště, External Storage

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 18_Ukladani_dat
projekt ExternalStorage

Externím úložištěm se myslí paměť, disk, nebo karta, kterou je možné od zařízení odpojit, nebo která se chová, jako by ji odpojit bylo možné. Typickým příkladem je SD karta. Android standardně podporuje pouze jedno takové zařízení, což vývojáři situaci zjednodušuje, protože nemusí ověřovat, jaký druh přístupu má k danému kousku hardware. Existují sice zařízení, která umožňují připojit více SD karet, je to ale velmi výjimečné a k přepínání přístupu ke kartám se v takovém případě používají speciální techniky. Zde se spokojíme se situací, kdy má zařízení jedno nebo žádné externí úložiště.

Při práci s externím úložištěm nás bude trochu pronásledovat historie a s ní související změny v koncepci přístupu k souborům. Týká se to zejména nastavení oprávnění v manifestu. Do SDK verze 21 bylo nutné žádat zvlášť o oprávnění pro čtení, READ_EXTERNAL_STORAGE a zvlášť pro zápis, WRITE_EXTERNAL_STORAGE. Tato oprávnění byla v následujících verzích sloučena a žádalo se pouze o zápis s tím, že čtení bylo přidělené automaticky.

U Androidu verze 10 (API 29) došlo ke změně, začal být vyžadovaný speciální příznak s názvem requestLegacyExternalStorage, který signalizoval systému, že má aplikaci přístup k souborům umožnit. Od Androidu verze 11 (API 30) je nutné znovu žádat o čtení nebo zápis, ale nově přibylo oprávnění pro správu souborů, MANAGE_EXTERNAL_STORAGE.

Není se v tom úplně snadné orientovat, ale naštěstí je možné žádat i o nepotřebná oprávnění, přičemž systém ta nadbytečná prostě ignoruje. To znamená, že aplikace může žádat o čtení, o zápis, o správu a zároveň mít nastavený příznak výjimky, systém si z toho vybere pouze to, co je pro jeho verzi relevantní. Pro aplikace které nepotřebují být kompatibilní s verzemi před Androidem 11 je to jednodušší, stačí žádat o oprávnění ke čtení nebo zápisu a případně ke správě souborů.

Z hlediska uživatele je to tak, že do Androidu 10 schvaloval pouze přístup k úložišti, což zahrnovalo jak čtení, tak zápis. Od verze 11 už nepřiděluje práva k úložišti, ale k souborům a médiím. Pokud aplikace žádá o čtení nebo zápis, může uživatel povolit přístup k médiím. Když ale požádá také o správu souborů, uživatel si vybírá, zda jí povolí přístup jenom k médiím, nebo ke všem souborům.

Uděláme si jednoduchý příklad, aplikaci, která bude stejně jako v předchozí kapitole obsahovat formulář s textovým polem a dvěma tlačítky, z nichž jedno bude obsah textového pole ukládat do souboru test.txt a druhé obsah souboru přečte a do textového pole ho vloží. Při zápisu opět zobrazíme umístění textového souboru v paměti zařízení.

Co je potřeba udělat?

- vytvořit formulář s víceřádkovým textovým polem a dvěma tlačítky

- doplnit do souboru manifestu požadavek na oprávnění pro přístup k úložišti a souborům

- napsat metody, které data uloží a zase načtou

Formulář se nebude lišit od příkladu z předchozí kapitoly. Obsahuje pouze základní prvky - textové pole a tlačítka. Použijeme například constraint layout, do kterého textové pole a tlačítka umístíme na vhodné pozice. Tlačítka budou volat metody nacistData() a ulozitData().

Práci s externím úložištěm musí aplikaci vždy povolit uživatel. Proto je nezbytné umístit do manifestu žádost o povolení přístupu k externímu úložišti a k souborům:

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE" />
```

Pokud není potřeba kompatibilita s verzemi Androidu 9 a staršími, je možné druhý řádek vynechat. Pro správnou práci Androidu verze 10 je naopak nutné přidat ještě do bloku application příznak requestLegacyExternalStorage:

```
<application
    android:requestLegacyExternalStorage="true"
    ...
```

Při ladění nesmíme zapomenout aplikaci právo přístupu k souborům nebo k externímu úložišti přiřadit. Jelikož aplikaci neinstalujeme ale pouze spouštíme, může se stát, že nás o povolení nepožádá. V takovém případě je potřeba v Nastavení → Aplikace a oznámení → najít aplikaci → Oprávnění → přístup povolit. Pokud bychom toto neudělali, dostávali bychom během čtení a zápisu chybové hlášení „Permission denied“.

Od verze API 31 je nutné požádat uživatele o jednorázové povolení přístupu k externím souborům:

```
// metoda pozada o pravo pristupu k externim souborum
private void povoleniPristupu() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
        if (!Environment.isExternalStorageManager()){
            Intent intent = new Intent();
            intent.setAction(
```

```
                Settings.ACTION_MANAGE_APP_ALL_FILES_ACCESS_PERMISSION);
            Uri uri = Uri.fromParts("package", this.getPackageName(), null);
            intent.setData(uri);
            startActivity(intent);
        }
    }
}
```

Pokud je aplikace sestavená pro Android s minimálním API vyšším než 31, neboli Android 11 (označený písmenem R), je možné podmínku na verzi SDK vynechat. Metodu povoleniPristupu() je možné zavolat v onCreate() události aktivity, nebo před prvním pokusem o přístup do externího úložiště.

Na rozdíl od ukládání dat do paměti telefonu, v případě externího zařízení je potřeba stále kontrolovat, jestli je připojené a jestli má aplikace možnost zápisu. V MainActivity.java si uděláme metodu pro ověření dostupnosti externího zařízení kontrolaStavuExternihoMedia().

```
// metoda zjistí, jestli je mozne externi zarizeni pouzit ke cteni/zapisu
private String kontrolaStavuExternihoMedia() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        // data lze cist a zapisovat
        return "";
    } else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return "Data lze z externího média pouze číst.";
    } else {
        return "Data nelze na externí médium zapisovat ani z něj číst.";
    }
}
```

Následně v MainActivity.java vytvoříme metodu ulozitData() pro uložení dat do souboru na vnějším úložišti:

```
public void ulozitData(View sender)
{
    // overit, jestli je mozne externi uloziste pouzit
    String stavExternihoMedia = kontrolaStavuExternihoMedia();
    if(stavExternihoMedia.length()>0) {
        Toast.makeText(MainActivity.this, stavExternihoMedia,
                                    Toast.LENGTH_SHORT).show();
        return;
    }

    // nacteni hodnot z formulare
    EditText editText1 = (EditText)findViewById(R.id.editText1);
    String inputText = editText1.getText().toString();

    // nazev souboru a jeho umisteni
    String NAZEV_SOUBORU = "test.txt";

    String adresar = Environment.getExternalStorageDirectory()
                                    .getAbsolutePath();

    // zapis dat do souboru
    File file = new File(adresar, NAZEV_SOUBORU);
    FileWriter writer = null;
    try {
```

```
            writer = new FileWriter(file);
            writer.write(inputText);
            writer.close();
        } catch (IOException e) {
        String txtChyba = "Data se nepodařilo zapsat. " + e.getMessage();
        Toast.makeText(MainActivity.this, txtChyba, Toast.LENGTH_SHORT).show();
            return;
        }

        // zjistit misto ulozeni souboru
        String umisteni = "Uloženo: " + adresar + "/" + NAZEV_SOUBORU;
        // pro informaci zaobrazit umisteni souboru
        Snackbar.make(findViewById(android.R.id.content), umisteni,
                        Snackbar.LENGTH_LONG).setTextMaxLines(5).show();
}
```

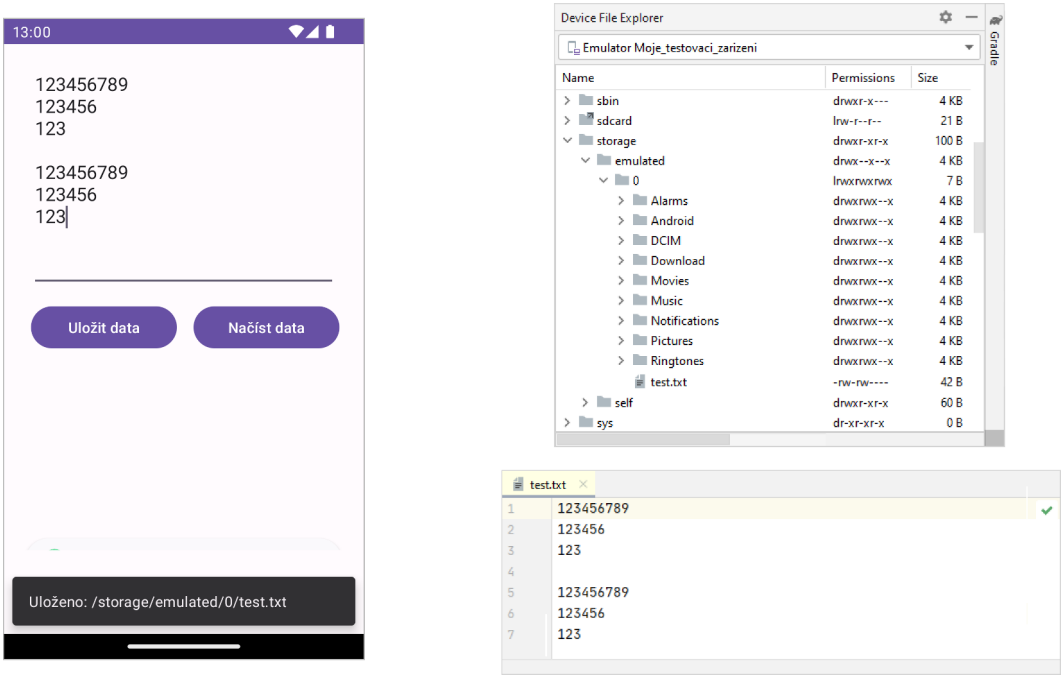
Metoda ověří dostupnost úložiště, vyzvedne data z textových polí, vytvoří stream pro zápis a jeho prostřednictvím zapíše data do souboru. Stream následně uzavře. Nakonec zjistí umístění dat a pomocí Snackbar zprávy o něm informuje uživatele.

Pro čtení dat napíšeme další metodu, nacistData(), která data obdobným způsobem ze souboru vyzvedne. Využije k tomu postupné načítání dat po jednotlivých znacích, výsledek doplní do polí na formuláři:

```
public void nacistData(View sender)
{
    // overit, jestli je mozne externi uloziste pouzit
    String stavExternihoMedia = kontrolaStavuExternihoMedia();
    if(stavExternihoMedia.length()>0) {
        Toast.makeText(MainActivity.this, stavExternihoMedia,
                                          Toast.LENGTH_SHORT).show();
        return;
    }

    // nazev souboru a jeho umisteni
    String NAZEV_SOUBORU = "test.txt";
    String adresar = Environment.getExternalStorageDirectory()
                                          .getAbsolutePath();

    try {
        // otevrit soubor pro cteni
        File soubor = new File(adresar, NAZEV_SOUBORU);
        FileInputStream inStream = new FileInputStream(soubor);

        // vytvorit instanci bufferu
        StringBuffer nactenyText = new StringBuffer();

        // soubor postupne, po znacich, nacist do bufferu
        int znak;
        while ((znak = inStream.read()) != -1) {
            nactenyText.append((char)znak);
        }

        // ulozit text do formulare
        EditText editText1 = (EditText)findViewById(R.id.editText1);
        editText1.setText(nactenyText);
    }
    catch (IOException e) {
        // zobrazit informaci o chybe
```

```
            String textChyby = "Nedaří se načíst..\n" + e.getMessage();
            Toast.makeText(MainActivity.this, textChyby, Toast.LENGTH_SHORT).show();
        }
}
```

V emulátoru nebo na koncovém zařízení by měla aplikace a uložená data vypadat jako na obrázku 18.4.



Obrázek 18.3: Aplikace pracující s daty v úložišti External Storage, spuštěná na emulátoru

18.4 Veřejně přístupné adresáře, Public Folders

Operační systém Android poskytuje několik tematicky členěných složek, ve kterých si může uživatel libovolným způsobem organizovat svá data. Typicky jde o složky fotografií, videí, hudby, stažených souborů a tak podobně. Soubory v těchto složkách zůstávají uložené i po smazání aplikace, která je vytvořila.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 18_Ukladani_dat
projekt PublicFolders

Tento způsob organizace souborů je výhodný nejen pro uživatele, ale i pro autory aplikací, protože není potřeba příliš složitě řešit, kam by se měly uložit vytvořené soubory. Prostě se nechá na Androidu, aby je umístil tam, kam patří.

Veřejně přístupné adresáře spadají do kategorie externích úložišť. Není však nutné ověřovat jestli je úložiště připojené, protože umístění složek si řídí Android sám a jakmile zjistí že není k dispozici požadovaná složka na externím úložišti, přesměruje operace do ekvivalentního adresáře v interní paměti. Je však nutné mít v manifestu aplikace nastavené právo přístupu k vnějšímu úložišti.

Pro přístup k veřejně přístupným adresářům se používají dvě základní metody:

getExternalStorageDirectory() - vrací cestu ke kořenovému adresáři, ve kterém se nachází jednotlivé veřejně přístupné složky

getExternalStoragePublicDirectory(String typPodadresare) - vrací cestu ke konkrétnímu podadresáři, který je reprezentovaný jednou z následujících konstant:

DIRECTORY_ALARMS - krátké zvukové soubory, které může uživatel nastavit jako zvuk budíku

DIRECTORY_DCIM - fotografie a videa pořízená zařízením

DIRECTORY_DOCUMENTS - uživatelem vytvořené soubory

DIRECTORY_DOWNLOADS - stažené soubory

DIRECTORY_MOVIES - filmy, videa

DIRECTORY_MUSIC - hudební soubory

DIRECTORY_NOTIFICATIONS - krátké zvukové soubory, které může uživatel nastavit jako zvuk upozornění

DIRECTORY_PICTURES - obrázky

DIRECTORY_PODCASTS - zvukové soubory zaslané uživateli

DIRECTORY_RINGTONES - krátké zvukové soubory, které může uživatel nastavit jako vyzváněcí melodii

Čtení a zápis se velmi podobá postupu popsaném v kapitole Externí úložiště. Proto si jako příklad pouze upravíme algoritmus ze zmíněné kapitoly. Formulář převezmeme v nezměněné podobě, jen přizpůsobíme metody ve třídě MainActivity.java.

V metodě ulozitData() stačí změnit umístění souboru, vše ostatní může zůstat téměř beze změny.

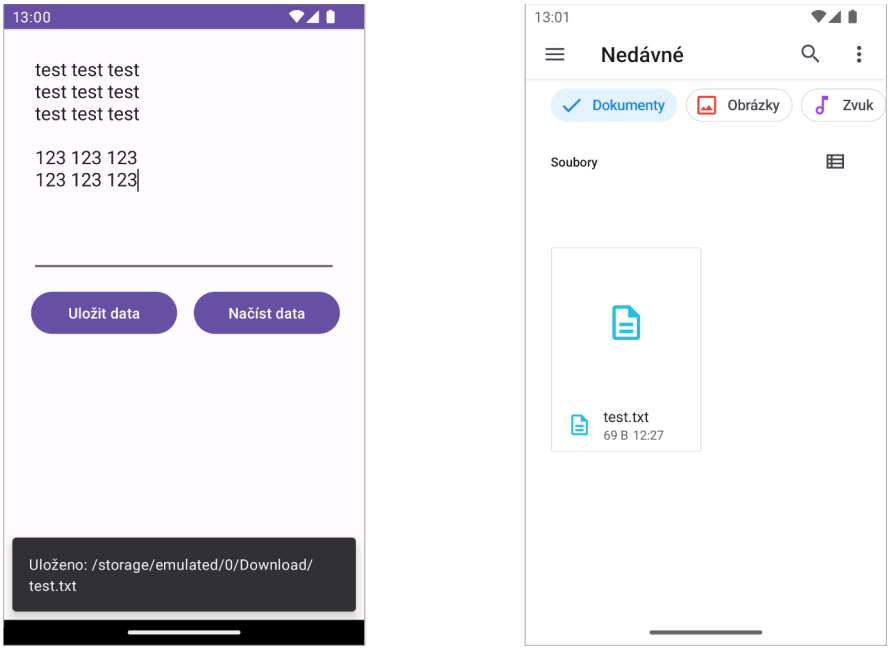
```
public void ulozitData(View sender)
{
    // nacteni hodnot z formulare
    EditText editText1 = (EditText)findViewById(R.id.editText1);
    String inputText = editText1.getText().toString();

    // nazev souboru a jeho umisteni
    String NAZEV_SOUBORU = "test.txt";
    File adresar = Environment
            .getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

    // zapis dat do souboru
    File file = new File(adresar, NAZEV_SOUBORU);
    try {
        FileWriter writer = new FileWriter(file);
        writer.write(inputText);
        writer.close();
    } catch (IOException e) {
        String txtChyba = "Data se nepodařilo zapsat. " + e.getMessage();
        Toast.makeText(MainActivity.this, txtChyba, Toast.LENGTH_SHORT).show();
            return;
    }

    // zjistit misto ulozeni souboru
    String umisteni = "Uloženo: " + adresar + "/" + NAZEV_SOUBORU;
```

```
    // pro informaci zaobrazit umisteni souboru
    Snackbar.make(findViewById(android.R.id.content), umisteni,
                  Snackbar.LENGTH_LONG).setTextMaxLines(5).show();
}
```

Stejným způsobem upravíme i metodu pro čtení nactiData(). Nesmíme zapomenout přidat do manifestu nastavení práva přístupu k vnějšímu úložišti a při prvním spuštění v zařízení nebo na emulátoru aplikaci tato oprávnění přidat. Jinak se, namísto uložení souboru, aplikace odvděčí strohým chybovým hlášením „Permission denied“.

V příkladu jsme zápis přesměrovali do DIRECTORY_DOWNLOADS, neboli do složky souborů stažených do zařízení z internetu, přes bluetooth atd. Výsledný soubor najdeme buď pomocí libovolné aplikace pro správu souborů, nebo přímo v zařízení v Nastavení → Úložiště → Dokumenty a další → v hamburger menu v levém horním rohu displeje vybrat položku Nedávné nebo Stahování. Tato cesta se však může lišit v závislosti na výrobci zařízení a na verzi Androidu.

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 18.4.



Obrázek 18.4: Aplikace pracující s daty ve veřejně přístupné složce, spuštěná na emulátoru

18.5 Dočasné soubory, Cache

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 18_Ukladani_dat
projekt Cache

Aplikace při své práci často potřebují zakládat dočasné, takzvané tmp (temporary) soubory, které mají význam v okamžiku vzniku, ale krátce poté, obvykle s ukončením aplikace, mohou být smazány.

Android má pro tyto účely dvě úložiště, takzvané cache. Jedna je interní, systém se u ní stará o promazávání starších souborů, pokud je aplikace nesmaže sama a druhá externí, kterou ale systém dále nespravuje. V případě smazání aplikace jsou však odstraněny všechny její dočasné soubory z obou typů úložišť.

Android poskytuje pro tyto účely dvě následující metody:

getCacheDir() - vrátí cestu k dočasným souborům aplikace, uloženým v interním úložišti

getExternalCacheDir() - vrátí cestu k dočasným souborům aplikace, uloženým v externím úložišti

Čtení a zápis se opět podobá postupu popsaném v kapitole Externí úložiště. Proto si jako příklad upravíme příklad ze zmíněné kapitoly. Formulář převezmeme v nezměněné podobě, pouze přizpůsobíme metody ve třídě MainActivity.java.

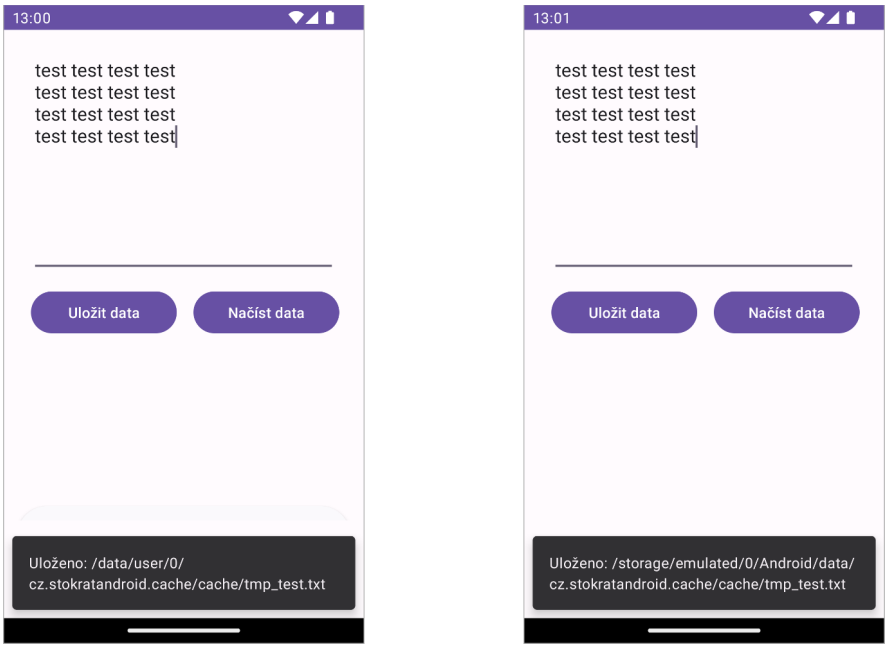
V metodě ulozitData stačí změnit umístění souboru, vše ostatní může zůstat téměř bez úprav.

```
public void ulozitData(View sender)
{
    // nacteni hodnot z formulare
    EditText editText1 = (EditText)findViewById(R.id.editText1);
    String inputText = editText1.getText().toString();

    // nazev souboru a jeho umisteni
    String NAZEV_SOUBORU = "tmp_test.txt";
    File adresar = getBaseContext().getCacheDir();

    // zapis dat do souboru
    File file = new File(adresar, NAZEV_SOUBORU);
    try {
        FileWriter writer = new FileWriter(file);
        writer.write(inputText);
        writer.close();
    } catch (IOException e) {
        String txtChyba = "Data se nepodařilo zapsat. " + e.getMessage();
        return;
    }

    // zjistit misto ulozeni souboru
    String umisteni = "Uloženo: " + adresar + "/" + NAZEV_SOUBORU;
    // pro informaci zaobrazit umisteni souboru
    Snackbar.make(findViewById(android.R.id.content), umisteni,
                  Snackbar.LENGTH_LONG).setTextMaxLines(5).show();
}
```

Stejným způsobem upravíme i metodu pro čtení nactiData().

V případě použití metody getExternalCacheDir(), resp. zápisu do externího úložiště, nemusíme kvůli dočasným souborům nastavovat v manifestu právo pro přístup k vnějším souborům. Tato starost tedy odpadá.

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 18.5. Na prvním otisku displeje je výsledek uložení souboru do interního a na druhém do externího úložiště.



Obrázek 18.5: Aplikace pracující s daty v dočasném souboru na interním (vlevo) a externím (vpravo) úložišti

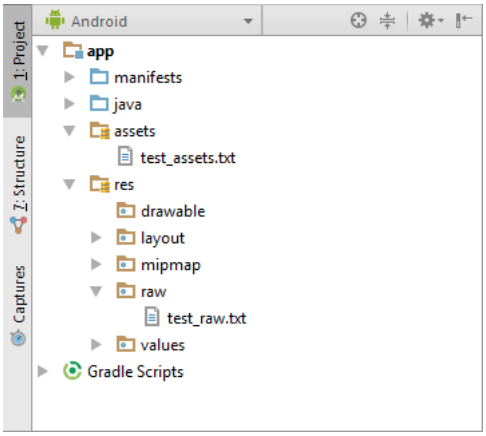
18.6 Soubory projektu, res, assets

V předchozích kapitolách jsme již narazili na soubory, které jsou součástí instalačního balíčku aplikace. Tyto soubory pocházejí z externích zdrojů a autor aplikace je přidá do složek v hierarchii projektu. Obecně se dá říct, že jde o soubory, které jsou k aplikaci pouze připojené, nekompilují se a aplikace může data uložená v těchto souborech pouze číst.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 18_Ukladani_dat
projekt SouboryProjektu

Existují dva druhy těchto souborů, každý má svoji zvláštní složku. První z nich je složka resources, neboli res/raw a druhá assets. Složku res známe už z dřívějška, jsou v ní uložené texty, nastavení barev atd. Tato složka je do projektu zařazena automaticky, vývojář si v ní pouze vytvoří adresář raw kliknutím na res pravým tlačítkem myši a volbou z kontextového menu New → Android resources directory. Do ní svůj soubor dostane tak, že ho jednoduše přidá do složky na disku počítače. Aby nebylo nutné složku pracně hledat, stačí v okně projektu kliknout pravým tlačítkem myši na adresář raw a v kontextovém menu zvolit Show in Explorer.

Naproti tomu složka assets není v projektu automaticky, musí se vytvořit ručně, pomocí kontextového menu v okně projektu New → Folder → Assets Folder. Do takto vzniklé složky se opět umístí vlastní soubory nakopírováním do složky na disku počítače.



Obrázek 18.6: Okno projektu, struktura aplikace

Obě složky, jak res/raw, tak assets jsou si velmi podobné a plní stejnou roli. Liší se však v tom, že soubory ve složce res mají podporu pro zobrazení na různých typech zařízení, s odlišnou orientací displeje, jazykovým nastavením atd. Důležité je také vědět, že při distribuci aplikace jsou soubory obou složek součástí instalačního balíčku, není se tedy potřeba obávat, že by nakonec některý ze souborů chyběl.

Na soubory v res se lze odkazovat přes ID. U assets nic takového není, tam jsou soubory organizované čistě podle uvážení autora aplikace a přistupuje se k nim prostřednictvím názvu souboru.

Načtení dat ze souboru ve složce res/raw pomocí InputStream:

```
Resources r = getResources();
InputStream inStream = r.openRawResource(R.raw.test_raw);
```

Načtení dat ze souboru ve složce assets pomocí InputStream :

```
Context context = getApplicationContext();
InputStream inStream = context.getAssets().open("test_assets.txt");
```

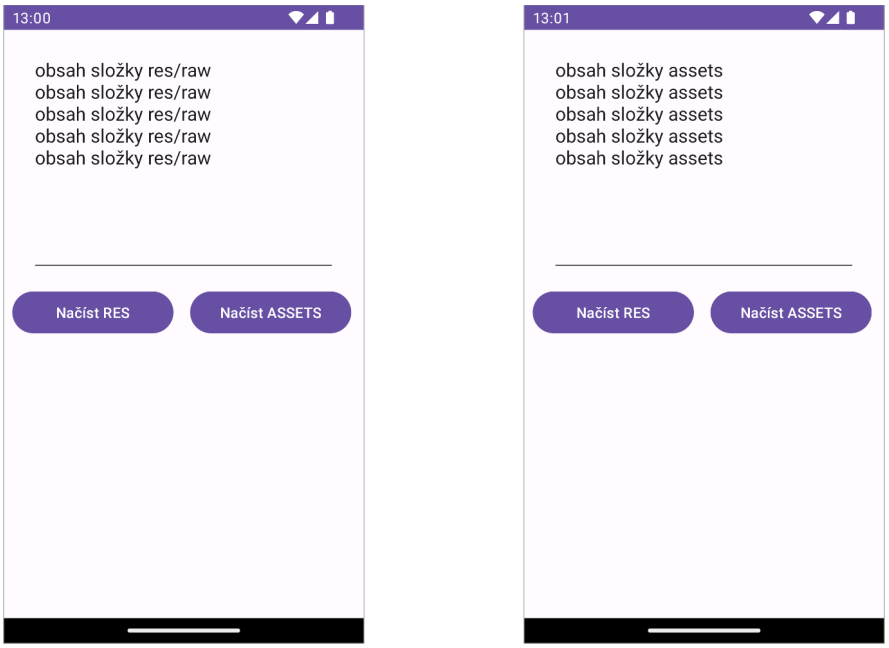
Tím je vytvořena instance inStream, ze které lze data načítat už známým způsobem po znacích, nebo pro změnu po řádcích:

```
BufferedReader buff = new BufferedReader(new InputStreamReader(inStream));
String radka = "";
while ((radka = buff.readLine()) != null) {
    nactenyText = nactenyText.concat(radka).concat("\n");
}
```

Potom už stačí načtená data vložit do formuláře a je hotovo:

```
EditText editText1 = (EditText)findViewById(R.id.editText1);
editText1.setText(nactenyText);
```

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 18.7.



Obrázek 18.7: Aplikace pracující s daty z adresáře res (vlevo) a assets (vpravo)

18.7 Databáze

V Androidu je zápis dat do souboru jednoduchý, přímočarý a relativně pohodlný. Ne vždy si s ním ale vývojář vystačí. Dříve nebo později se pravděpodobně dostane do situace, kdy bude pracovat s větším množstvím dat, s potřebou seskupovat, řadit, vyhledávat v nich, nebo je vyhodnocovat. Řešením takové situace je použití databáze.

Databáze obvykle představuje několik vzájemně logicky propojených tabulek, v nichž jsou strukturovaným způsobem uložena data, s nimiž se pracuje prostřednictvím příkazů databázového jazyka. Nejrozšířenějším typem jsou takzvané SQL databáze, neboli databáze, které poskytují přístup k datům pomocí SQL dotazů. SQL znamená Structured Query Language, čili strukturovaný dotazovací jazyk.

Systém Android obsahuje integrovanou knihovnu SQLite. Jde o malou, na systémové zdroje relativně nenáročnou databázi, která je šířená zdarma, pod volnou licencí. Původně byla napsaná pro použití na počítačích, kde ji mohli autoři software přilinkovat k aplikaci, takže uživatel nemusel nic dalšího instalovat a o existenci databáze často vůbec nevěděl. K vnitřní organizaci dat ji používaly nebo používají například prohlížeč Google Chrome, grafický editor Adobe Photoshop nebo antivirové programy McAfee.

Základní vlastností SQLite je to, že je lokální, čili nepotřebuje žádný server a odpadají tím záležitosti spojené s připojováním a ověřováním identity uživatele. Pro ukládání dat používá soubory umístěné do interního úložiště. A to tím způsobem, že si pro každou databázi založí samostatný soubor. Z hlediska Androidu je to výhodné, protože tak může být SQLite součástí systému, ale jednotlivé databáze už jsou přiřazeny aplikacím a v případě odinstalování aplikace se jednoduše smažou i databázové soubory které si založila. Z toho vyplývá, že každá aplikace může mít jednu svoji databázi, která je tvořena jedním datovým souborem v paměti zařízení.

K SQLite databázi poskytuje přístup třída SQLiteOpenHelper. Ta řeší dva základní problémy:

- Když je aplikace poprvé spuštěna a žádnou databázi zatím vytvořenou nemá, založí ji. Přitom vyvolá událost onCreate(), kde autor aplikace vytvoří tabulky, indexy a případně zapíše iniciační data.

- Pokud dojde k nahrání nové verze aplikace, zkontroluje, jestli se nezměnila také verze databázové struktury a pokud ano, vyvolá událost onUpgrade(). Do ní autor připraví sekvenci příkazů pro úpravu databáze na aktuální verzi.

Aby bylo možné spouštět SQL dotazy nad databází, je potřeba vytvořit instanci SQLiteDatabase. K tomu se využije buď příkaz getReadableDatabase() nebo getWritableDatabase():

```
SQLiteDatabase db = this.getReadableDatabase();
```

nebo

```
SQLiteDatabase db = this.getWritableDatabase();
```

První příkaz je primárně určený pouze pro čtení z databáze, druhý i pro zápis. Oba příkazy vrací stejný objekt a vývojáři v zásadě nic nebrání používat stále getWritableDatabase(). Pouze když už není v paměti zařízení dostatek volného místa, může systém zápis odmítnout. V takovém případě se může hodit otevřít databázi alespoň pro čtení.

Na takto získaném objektu typu SQLiteDatabase můžeme provádět databázové operace. Objekt poskytuje metody pro správu databáze, operace s daty a zejména pro spouštění SQL dotazů. Metoda execSQL() umožní v parametru předat SQL dotaz databázi, která ho následně provede:

```
db.execSQL("SQL dotaz");
```

Například vložení nového záznamu do tabulky VerzeAndroidu (k syntaxi SQL dotazu se ještě vrátíme):

```
db.execSQL("INSERT INTO VerzeAndroidu VALUES (NULL, 'KitKat', '4.4')");
```

Metodu execSQL() můžeme používat k předávání jakýchkoliv SQL dotazů které potřebujeme aby databáze provedla. Často se stává, že autoři aplikací pomocí SQL příkazů provádí i operace, které by mohli udělat pomocí metod objektu typu SQLiteDatabase. Důvodem je větší kontrola nad tím, co bude do databáze odeslané a obvykle také kratší zápis. Naproti tomu je u SQL dotazu větší riziko chyby, zejména co se týká použitých datových typů.

Nevýhodou execSQL() je absence návratové hodnoty. Nemůžeme tedy pomocí této metody získávat z databáze data a předávat je k dalšímu zpracování. K tomu slouží metoda rawQuery(), které také v parametru předáme SQL příkaz, ale ona navíc vrátí výsledek. Ještě se k ní dostaneme.

18.7.1 SQL syntaxe

SQLite podporuje naprostou většinu příkazů obecného jazyka SQL. Popis by vydal na samostatnou publikaci, my se prozatím spokojíme s úplnými základy. S příkazy pro vytvoření a smazání tabulky, načtení záznamů, jejich vkládání, úpravou a mazáním.

Založení tabulky

Aby měla databáze smysl, musí obsahovat jednu nebo více datových tabulek. Tabulku založíme pomocí dotazu CREATE TABLE, ve kterém specifikujeme názvy sloupců a datové typy. Tedy například:

```
CREATE TABLE VerzeAndroidu (_id INTEGER PRIMARY KEY, Nazev TEXT, Verze TEXT)
```

Tento dotaz založí tabulku s názvem VerzeAndroidu, určenou k evidenci jednotlivých verzí Androidu. Tabulka bude obsahovat tři sloupce:

_id – identifikátor záznamu, typu integer, který bude primárním klíčem tabulky. Jinými slovy, v tomto sloupci bude mít každý záznam svoje číslo, pod kterým ho bude možné dohledat.

Nazev – sloupec, s volným textem. Využijeme ho pro název verze Androidu.

Verze – sloupec, opět s volným textem. Využijeme ho pro číslo verze Androidu. Text volíme proto, abychom mohli číslo verze psát volně, s oddělovači v podobě teček. Pokud bychom se spokojili s celočíselnou hodnotou, použili bychom zápis `Verze INTEGER`.

První sloupec jsme nazvali _id. Mohl by se jmenovat jakkoliv jinak, ale v Androidu je výhodné pojmenovávat primární klíč právě takto, protože s názvem _id počítají datové adaptéry pomocí nichž data zobrazujeme v datových kontejnerech. Ušetříme si tak práci s přejmenováním názvu sloupce před předáním dat adaptéru.

Zkoušení funkčnosti SQL dotazů je v kódu aplikace poměrně komplikované. Můžeme ale využít některé z online služeb, která nám umožní připojení ke zkušební SQLite databázi, kde si syntaxi dotazů vyzkoušíme a do aplikace je zabudujeme již odladěné. Jedna z nich se nachází na internetové adrese sqliteonline.com

Smazání tabulky

Podobně jako jsme tabulku vytvořili, tak ji můžeme také smazat:

```
DROP TABLE IF EXISTS VerzeAndroidu
```

Tento SQL dotaz říká, že má být smazaná tabulka VerzeAndroidu, pokud již existuje. Smažou se jak data, tak celá struktura tabulky.

Načtení dat

Pro získávání dat z databáze se používá příkaz SELECT, který může mít mnoho podob. Nejjednodušší je tato:

```
SELECT * FROM VerzeAndroidu
```

Tento dotaz vrátí všechny záznamy z tabulky VerzeAndroidu. Znak ‚*‘ říká, že mají být vráceny všechny sloupce, které tabulka obsahuje.

Můžeme pochopitelně načíst pouze některé sloupce:

```
SELECT _id, Nazev FROM VerzeAndroidu ORDER BY Nazev
```

Tím získáme všechny záznamy z tabulky, ale pouze se sloupci s identifikátorem a názvem. Číslo verze zůstane nenačtené. Navíc pomocí klauzule ORDER zařídíme, aby byly záznamy vrácené abecedně seřazené podle názvu.

Příkaz SELECT je možné rozšiřovat o podmínky, vazby na další tabulky, agregační funkce, seskupování, o přejmenovávání sloupců atd. Pro naše účely se zatím spokojíme se dvěma uvedenými, základními dotazy.

Vkládání dat

Data do tabulky vkládáme pomocí příkazu INSERT. Stejně jako v předchozím případě se i zde seznámíme pouze se základní podobou dotazu:

```
INSERT INTO VerzeAndroidu ('Nazev', 'Verze') VALUES ('KitKat', '4.4')
```

Dotaz v této podobě zapíše do tabulky VerzeAndroidu nový záznam, který bude obsahovat ve sloupci Nazev hodnotu KitKat a ve sloupci Verze hodnotu 4.4. Do sloupce _id nic nezapisujeme, databáze vygeneruje klíč (číslo záznamu) sama. Pomocí INSERT můžeme vyplňovat pouze některé sloupce a jiné nechat prázdné, je však potřeba dát pozor na to, aby nezůstal prázdný sloupce s povinnou hodnotou (byl by při vytváření označený jako NOT NULL). Takový dotaz by skočil chybou. Naše tabulka VerzeAndroidu má však všechny sloupce nepovinné, takže se není čeho bát.

Lze použít i jednodušší zápis:

```
INSERT INTO VerzeAndroidu VALUES (NULL, 'KitKat', '4.4')
```

Tento dotaz vloží úplně stejná data. Jelikož ale nemáme specifikované sloupce, které vyplňujeme, musíme poslat hodnoty pro všechny které tabulka obsahuje. Pro _id odešleme NULL, neboli prázdnou hodnotu. Databáze si doplní identifikátor sama, stejně jako v předchozím případě. Zmíněné NULL je tam jenom z toho důvodu, aby odpovídal počet hodnot počtu sloupců. V tom se také skrývá riziko této zkrácené formy zápisu – jakmile do tabulky přidáme sloupec, dotaz přestane fungovat. Bezpečnější je tedy používat delší zápis, který s novým sloupcem, samozřejmě pokud není definovaný jako povinný, nemá nejmenší problém.

Úprava dat

Již vložená data můžeme podle potřeby aktualizovat pomocí příkazu UPDATE. Vždy potřebujeme znát identifikátor záznamu který měníme. Základní formát dotazu:

```
UPDATE VerzeAndroidu SET Nazev='Nougat', Verze='7.0' WHERE _id='4'
```

Dotazem sdělíme databázi, že má v tabulce VerzeAndroidu vyhledat záznam s identifikátorem 4 a změnit v něm hodnotu Nazev na Nougat a Verze na 7.0. Měnit můžeme samozřejmě libovolné sloupce, například jenom číslo verze:

```
UPDATE VerzeAndroidu SET Verze='7.0' WHERE _id='4'
```

Je potřeba dát pozor na klauzuli WHERE, kde je specifikované číslo upravovaného záznamu. Pokud bychom ji vynechali a dotaz vypadal takto:

```
UPDATE VerzeAndroidu SET Verze='7.0'
```

V takovém případě by databáze změnila hodnotu ve sloupci Verze u všech záznamů v tabulce VerzeAndroidu na 7.0.

Mazání dat

Stejně jako můžeme záznamy vkládat, tak je můžeme i smazat. K tomu slouží příkaz DELETE. Základní formát:

```
DELETE FROM VerzeAndroidu WHERE _id='4'
```

Tento dotaz po odeslání způsobí, že databáze vymaže z tabulky VerzeAndroidu záznam s identifikátorem 4. Opět zde má velký význam podmínka WHERE, která omezí operaci na konkrétní záznam. Podoba bez podmínky:

```
DELETE FROM VerzeAndroidu
```

Tímto dotazem vymažeme z tabulky VerzeAndroidu všechny záznamy. Tabulka bude po provedení dotazu zcela prázdná.

18.7.2 Metody třídy SQLiteDatabase

Třída SQLiteDatabase poskytuje sadu metod, určených ke správě databáze SQLite. Díky tomu je možné, s její pomocí, provádět všechny běžné databázové operace.

Spouštění SQL dotazů

Základní databázovou operaci, předání SQL dotazu databázi pomocí metody execSQL(), jsme si už popsali. Jenom zopakujme, že pomocí ní můžeme odeslat do databáze jakýkoliv SQL dotaz, který SQLite databáze podporuje. Z těch které jsme probrali, můžeme zavolat například aktualizaci názvu v záznamu s klíčem 4:

```
db.execSQL("UPDATE VerzeAndroidu SET Nazev='test' WHERE _id='4'");
```

Načtení dat

Pro načtení dat použijeme dotaz typu SELECT, ale voláme jej pomocí metody rawQuery(), která má implementovanou, na rozdíl od execSQL(), návratovou hodnotu. Jinými slovy, metoda dotaz nejen vykoná, ale také zpátky pošle výsledek.

Například získání dat z tabulky VerzeAndroidu:

```
Cursor data = db.rawQuery("select Nazev from VerzeAndroidu", null);
```

Prvním parametrem je SQL dotaz pro získání dat, do druhého parametru můžeme dát argumenty, které se do dotazu doplní. Nebo můžeme nechat dotaz bez argumentů, tedy s null. Metoda rawQuery() vrátí do proměnné data výsledek dotazu, neboli všechna získaná data v objektu typu Cursor.

Proměnnou data můžeme buď napojit na adaptér a zobrazit, nebo z ní můžeme jednotlivé záznamy získat pomocí sekvence příkazů:

```
data.moveToFirst();
do {
      String nazev = data.getString(c.getColumnIndex("Nazev"));
      // zde lze provadet operace se ziskanou hodnotou
} while (data.moveToNext());
data.close();
```

Vkládání dat

Krom toho, že objekt typu SQLiteDatabase umožňuje předávat do databáze SQL dotazy, tak také poskytuje možnosti přístupu k datům pomocí vlastních metod. Operace vkládání, úpravu a mazání dat lze provést voláním příslušných metod.

```
ContentValues vkladaneHodnoty = new ContentValues();
vkladaneHodnoty.put("Nazev", "KitKat");
vkladaneHodnoty.put("Verze", "4.4");
db.insert("VerzeAndroidu", null, vkladaneHodnoty);
```

Použijeme objekt typu ContentValues, který naplnime hodnotami jež chceme do tabulky zapsat a potom už jenom zavoláme nad objektem SQLiteDatabase metodu insert(), které předáme v parametrech název tabulky a vkládané hodnoty. Výsledek bude stejný jako bychom pomocí db.execSQL odeslali již známý SQL dotaz:

```
INSERT INTO VerzeAndroidu VALUES (NULL, 'KitKat', '4.4')
```

Úprava dat

Stejně postupujeme i v případě aktualizace dat. Naplníme si objekt typu ContentValues hodnotami které chceme upravit a zavoláme metodu update():

```
ContentValues vkladaneHodnoty = new ContentValues();
vkladaneHodnoty.put("Verze", "7.0");
db.update("VerzeAndroidu", vkladaneHodnoty, "_id=‘4‘", null);
```

Výsledek bude stejný jako bychom použili SQL dotaz:

```
UPDATE VerzeAndroidu SET Verze='7.0' WHERE _id='4'
```

Mazání dat

Pro smazání záznamu nemusíme plnit objekt typu ContentValues. Pouze zavoláme metodu delete() s příslušnými parametry:

```
db.delete("VerzeAndroidu", "_id=‘4‘", null);
```

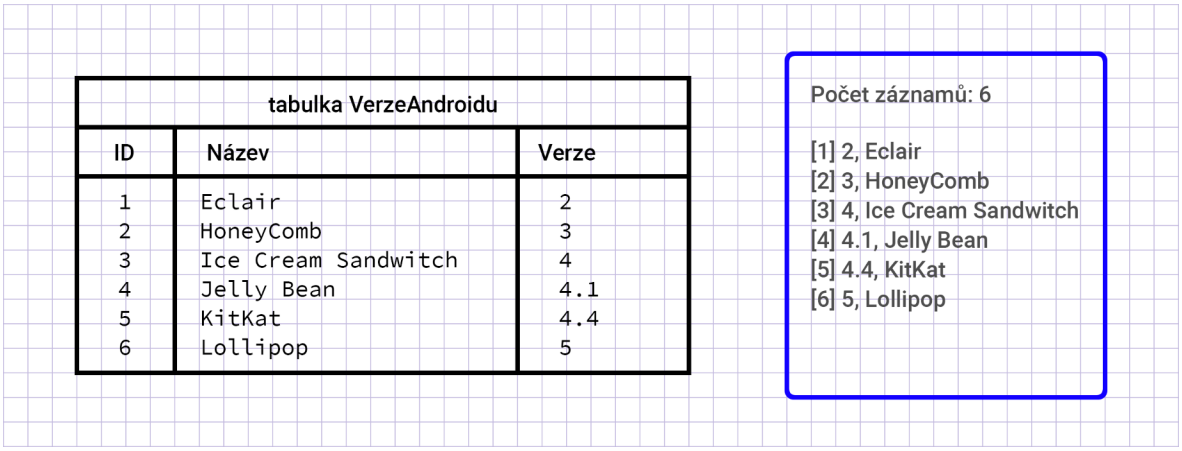
Výsledek se nebude lišit od dopadu SQL dotazu:

```
DELETE FROM VerzeAndroidu WHERE _id=4;
```

18.7.3 SQLite, 1. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 18_Ukladani_dat
projekt SQLite1

V prvním příkladu se omezíme na připojení k databázi, vytvoření tabulky, její naplnění pomocí programového kódu a následné načtení a zobrazení dat komponentou TextView, zasazenou do kontejneru ScrollView tak, aby mohla komfortně zobrazit větší množství textu. Pro práci s databází použijeme SQL dotazy předané databázi prostřednictvím metod execSQL() a rawQuery() objektu SQLiteDatabase.



Obrázek 18.8: Databázová tabulka a návrh zobrazení dat

Co je potřeba udělat?

- vytvořit formulář s komponentou TextView v kontejnerem ScrollView
- napsat jednoduchou knihovnu pro práci s databází
- založit databázovou tabulku a naplnit ji daty
- načíst data z tabulky a zobrazit je v komponentě TextView

Otevřeme soubor activity_main.xml a do formuláře vložíme kontejner Linear Layout. Do něj umístíme kontejner ScrollView. Oba kontejnery by měly být roztažené na celou plochu displeje. Do ScrollView vložíme komponentu TextView a opět jí nastavíme jak maximální šířku, tak i výšku.

Protože je přístup k databázi specifickou částí kódu, který vyžaduje samostatnou údržbu, není vhodné mít příslušné metody roztroušené v různých třídách. Řeší se to založením samostatné třídy, oddděné od třídy SQLiteOpenHelper. Nazvěme ji třeba DatabaseHandler. V této naší nové třídě přepíšeme události onCreate() a onUpgrade() a přidáme si metody vlastní, které zprostředkují všechny potřebné databázové operace.

```
public class DatabaseHandler extends SQLiteOpenHelper {

    public SQLiteDatabase db;

    // konstruktor se jmenem databaze a jeji verzi
    // zde dojde k zalozeni databaze aplikace
    public DatabaseHandler(Context context) {
        super(context, "VerzeAndroidu", null, 1);
```

```
    }

    // metoda volana pri vzniku databaze
    // zde zalozime databazove tabulky
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE VerzeAndroidu (ID INTEGER PRIMARY KEY,
                            Nazev TEXT, Verze TEXT)");
    }

    // metoda volana pri zmene verze databaze
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // smazat starou tabulku, pokud existuje
        db.execSQL("DROP TABLE IF EXISTS VerzeAndroidu");

        // znovu vytvorit tabulku
        onCreate(db);
    }

    // pripojeni k databazi
    public void pripojitDB() {
        db = getWritableDatabase();
    }

    // nacte a vrati vsechna data z tabulky VerzeAndroidu
    public Cursor nacistData() {
        return db.rawQuery("select * from VerzeAndroidu", null);
    }

    // do tabulky VerzeAndroidu vlozi zaznam
    public void vlozitZaznam(String nazev, String verze) {
        db.execSQL("INSERT INTO VerzeAndroidu ('Nazev', 'Verze')
                            VALUES ('" + nazev + "', '" + verze + "')");
    }

    // zmeni zaznam v tabulce VerzeAndroidu
    public void upravitZaznam(String id, String nazev, String verze) {
        db.execSQL("UPDATE VerzeAndroidu SET Nazev='" + nazev + "',
                            Verze='" + verze + "' WHERE id='" + id + "'");
    }

    // smaze obsah tabulky VerzeAndroidu
    public void smazatData() {
        db.execSQL("DELETE FROM VerzeAndroidu");
    }

    // odpojeni se od databaze
    public void odpojitDB(){
        db.close();
    }
}
```

Knihovna obsahuje kromě předefinovaných událostí onCreate() a onUpdate() vlastní metodu pro připojení k databázi pripojitDB(), kterou je nutné zavolat před první databázovou operací. Metoda nejen že připojí databázi, ale zároveň tím vyvolá událost onCreate() a pokud došlo ke změně verze databáze, tak i onUpgrade().

Verzi databáze si nastavujeme sami v konstruktoru třídy DatabaseHandler. Zvyšujeme ji pokaždé, když uděláme úpravu struktury databáze. U aplikace, která již byla distribuována

k uživatelům je nutné rozlišit, jakou mají předchozí verzi a pomocí skriptů udělat takové úpravy, aby se struktura dostala na verzi aktuální. Předchozí verzi zjistí Android sám a předá ji v parametru oldVersion. Aktuální verzi sice také předá, v parametru newVersion, jeho hodnotu ale předtím zjistí z konstruktoru třídy, kam ji můžeme sami, podle vlastního uvážení, zapsat. Z hlediska vývoje aplikace, dokud ještě databáze neobsahuje ostrá data, je výhodné nechat při upgrade databázi celou smazat a znovu ji vytvořit v aktuální verzi. Tak nemusíme udržovat skripty pro zvyšování jejich verzí. Pokud uděláme úpravu ve struktuře databáze, stačí když v konstruktoru naší třídy DatabaseHandler jednoduše zvýšíme verzi. Databáze se po restartu aplikace sama aktualizuje vyvoláním události onUpgrade().

Další metody ve třídě vytváří a mažou tabulku VerzeAndroidu, načítají z ní data, zapisují je a máme i metodu pro aktualizaci dat, kterou sice nevyužijeme, ale kdybychom chtěli nějaké záznamy modifikovat, jistě by se hodila.

Poslední metodou je odpojitDB(), která ukončí spojení s databází. Tu je vhodné zavolat po poslední databázové operaci, abychom systému zbytečně neubírali systémové zdroje. V každém případě databázi takto odpojujeme před ukončením aplikace. Odpojením se od databáze totiž zároveň dojde k jejímu uzavření a data jsou lépe chráněna před případným poškozením.

Ve třídě MainActivity si v události onCreate() vytvoříme instanci naší knihovny. Odkážeme se na ní pomocí proměnné dbHelper a jejím prostřednictvím už jenom budeme volat potřebné metody. Připojíme se tak k databázi, čímž vyvoláme při prvním spuštění událost onCreate(), která založí tabulku VerzeAndroidu. Pro jistotu z ní vymažeme stará data, což se uplatní při dalších spuštěních a tabulku naplníme vlastními záznamy.

Následně data z databáze načteme, zjistíme kolik záznamů v tabulce je, jejich počet vypíšeme do textového pole. Potom už jenom v cyklu jednotlivé záznamy načítáme a každý přidáme na konec textového pole.

```
public class MainActivity extends AppCompatActivity {

    DatabaseHandler dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // vytvorit instanci tridy DatabaseHandler
        dbHelper = new DatabaseHandler(this);

        // pripojeni se k databazi
        dbHelper.pripojitDB();

        // vymazat z tabulky stara data
        dbHelper.smazatData();

        // vlozit do tabulky iniciacni data
        dbHelper.vlozitZaznam("bez kódového označení", "1.0");
        dbHelper.vlozitZaznam("bez kódového označení", "1.1");
        dbHelper.vlozitZaznam("Cupcake", "1.5");
        dbHelper.vlozitZaznam("Donut", "1.6");

        ...

        // nacist data z tabulky
        Cursor data = dbHelper.nacistData();
```

```
        // zjistit pocet nalezenych zaznamu
        int pocetZaznamu=data.getCount();

        // najit textove pole ve formulari
        TextView dbDataTextView = (TextView) findViewById(R.id.dbData);

        // kontrola, jestli je v tabulce alespon jeden zaznam
        if(pocetZaznamu < 1) {
            dbDataTextView.setText("V databázi nebyl nalezen žádný záznam");
            return;
        }
        else {
            dbDataTextView.setText("Počet záznamů v tabulce: " +
                                                pocetZaznamu + "\n\n");
        }

        // data postupne nacist a zapsat do textoveho pole
        while (data.moveToNext()) {
            dbDataTextView.append("[" + data.getString(0) + "]");
            dbDataTextView.append(" Android " + data.getString(2));
            dbDataTextView.append(", " + data.getString(1) + "\n");
        }

        // odpojeni se od databaze
        dbHelper.odpojitDB();
    }
}
```

Při práci s databází si můžeme pomoct použitím nástroje Database Inspector, který lze spustit z menu View → Tool Windows → Database Inspector. Pomocí něj je možné prohlížet tabulky a spouštět nad nimi SQL příkazy. Umí se připojit k běžící aplikaci, zobrazit strukturu tabulek a otevřít okno pro zadání SQL příkazu. Pozor, při připojování nesmí být aplikace zastavena na breakpointu a musí mít otevřené připojení k databázi. Což v našem případě znamená, že nesmí být zavolaná metoda dbHelper.odpojitDB(). Stačí ji zakomentovat a aplikaci spustit, Database Inspector by se k ní měl bez problému připojit.

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 18.9.

VerzeAndroidu (27 rows)

```
SELECT * FROM 'VerzeAndroidu' LIMIT 0,10
```

Execute

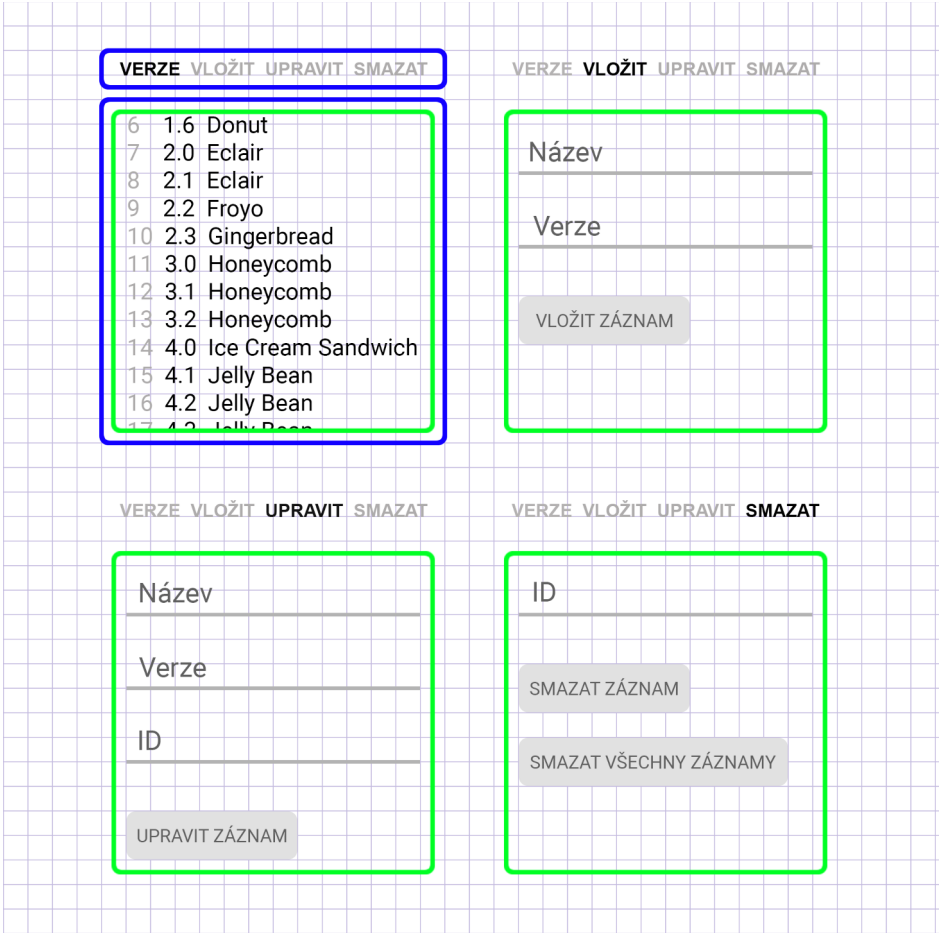
ID	Nazev	Verze
1	bez kódového označení	1.0
2	bez kódového označení	1.1
3	Cupcake	1.5
4	Donut	1.6
5	Eclair	2.0
6	Eclair	2.1
7	Froyo	2.2
8	Gingerbread	2.3
9	Honeycomb	3.0
10	Honeycomb	3.1

Obrázek 18.9: Obsah databázové tabulky (prvních 10 záznamů) a aplikace s komponentou Text View, která tato data zobrazuje

18.7.4 SQLite, 2. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 18_Ukladani_dat
projekt SQLite2

Ve druhém příkladu se pokusíme o komplexnější přístup k databázi. Uděláme si formulář s kontejnery Tab Layout a View Pager, kde na první záložce bude přehled dat uložených v databázi a na dalších záložkách budou jednotlivé operace – vložení, úprava a smazání záznamu. Struktura databáze zůstane stejná jako v předchozím příkladu. Pro operace s daty nebudeme tentokrát používat SQL dotazy, ale metody objektu SQLiteDatabase.



Obrázek 18.10: Návrh formuláře s komponentou View Pager, jejíž záložky obsahují prvky základních databázových operací

Co je potřeba udělat?

- vytvořit formulář s komponentami Tab Layout a View Pager
- nadefinovat jednotlivé záložky a prvky na nich
- vytvořit jednoduchou knihovnu pro práci s databází
- založit databázovou tabulku
- načítat data z tabulky na první záložku, prostřednictvím adaptéru a komponenty Grid View

Otevřeme soubor activity_main.xml a do formuláře vložíme pod sebe kontejnery Tab Layout a View Pager. Kontejnery nastavíme tak, aby vyplnily plochu celého displeje. Vytvoříme čtyři fragmenty pro jednotlivé záložky a vhodně si je pojmenujeme. Do prvního fragmentu vložíme komponentu gridView, která bude zobrazovat data a do dalších fragmentů umístíme prvky nezbytné pro jejich editaci, viz návrh formuláře.

I v tomto příkladu si připravíme samostatnou třídu, oděděné od třídy SQLiteOpenHelper, která bude mít funkci rozhraní pro přístup k databázi. V naší nové třídě přepíšeme události onCreate() a onUpgrade() a přidáme si metody vlastní, které zprostředkují všechny potřebné databázové operace, realizované tentokrát pomocí volání metod objektu SQLiteDatabase.

```
public class DatabaseHandler extends SQLiteOpenHelper {
```

```
public SQLiteDatabase db;

// konstruktor se jmenem databaze a jeji verzi
// zde dojde k zalozeni databaze aplikace
public DatabaseHandler(Context context) {
    super(context, "VerzeAndroidu", null, 1);
}

// metoda volana pri vzniku databaze
// zde zalozime databazove tabulky
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE VerzeAndroidu (_id INTEGER PRIMARY KEY,
                                Nazev TEXT, Verze TEXT)");
}

// metoda volana pri zmene verze databaze smerem nahoru
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    // smazat starou tabulku, pokud existuje
    db.execSQL("DROP TABLE IF EXISTS VerzeAndroidu");

    // znovu vytvorit tabulku
    onCreate(db);
}

// metoda volana pri zmene verze databaze smerem dolu
@Override
public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    // smazat starou tabulku, pokud existuje
    db.execSQL("DROP TABLE IF EXISTS VerzeAndroidu");

    // znovu vytvorit tabulku
    onCreate(db);
}

// pripojeni k databazi
public void pripojitDB() {
    db = getWritableDatabase();
}

// nacte a vrati vsechna data z tabulky VerzeAndroidu
public Cursor nacistData() {
    return db.rawQuery("select _id, Verze, Nazev from VerzeAndroidu", null);
}

// do tabulky VerzeAndroidu vlozi zaznam
public void vlozitZaznam(String nazev, String verze) {
    ContentValues vkladaneHodnoty = new ContentValues();
    vkladaneHodnoty.put("Nazev", nazev);
    vkladaneHodnoty.put("Verze", verze);
    db.insert("VerzeAndroidu", null, vkladaneHodnoty);
}

// zmeni zaznam v tabulce VerzeAndroidu
public void upravitZaznam(String id, String nazev, String verze) {
    ContentValues vkladaneHodnoty = new ContentValues();
    vkladaneHodnoty.put("Nazev", nazev);
    vkladaneHodnoty.put("Verze", verze);
    db.update("VerzeAndroidu", vkladaneHodnoty, "_id=" + id, null);
}
```

```
    // smaze zaznam z tabulky VerzeAndroidu
    public void smazatZaznam(String id) {
        db.delete("VerzeAndroidu", "_id=" + id, null);
    }

    // smaze obsah tabulky VerzeAndroidu
    public void smazatData() {
        db.delete("VerzeAndroidu", null, null);
    }

    // odpojeni se od databaze
    public void odpojitDB(){
        db.close();
    }
}
```

Knihovna opět obsahuje kromě předefinovaných událostí onCreate() a onUpdate() vlastní metodu pro připojení k databázi pripojitDB, kterou je nutné zavolat před každou databázovou operací. Metoda nejen že připojí databázi, ale zároveň tím vyvolá událost onCreate() a pokud došlo ke změně verze databáze, tak i onUpgrade().

V předchozím příkladu jsme mohli upravovat databázi a s každou změnou jsme jednoduše zvýšili číslo verze databáze. Potom stačilo nechat aplikaci, ať po restartu vyvolá událost onUpgrade(), která starou verzi databáze smaže a založí verzi s aktuální strukturou. Toto mělo nevýhodu, že jsme mohli verzi databáze sice kdykoliv zvyšovat, ale její číslo logicky stále rostlo. Jelikož jsme nedělali žádnou distribuci aplikace, tak bylo stálé zvyšování čísla verze databáze zbytečné. Kdybychom se pokusili číslo verze snížit, obdrželi bychom pouze chybové hlášení, které by nás informovalo, že toto není možné.

Nově máme předefinovanou událost onDowngrade(), která má chybové hlášení, při snižování verze, na svědomí. Tím že tuto událost přepíšeme, tak nejen že umožníme snížit verzi, ale také můžeme definovat SQL příkazy, které návrat ke starší verzi zajistí. Koncoví uživatelé aplikace toto asi nevyužijí, protože obvykle není nutné snižovat verzi aplikace, ale pro účely vývoje je výhodné, když můžeme nastavit verzi databáze nižší, než jsme ji měli dříve. Proto nám také stačí skript, který databázi prostě smaže a znova vytvoří.

V každém případě je potřeba dbát na to, aby verze databáze byla v okamžiku publikování aplikace vyšší (nebo stejná, pokud nedošlo k úpravě struktury databáze) než byla verze v předchozím vydání aplikace.

Další metody třídy vytváří a mažou tabulku VerzeAndroidu, načítají z ní data, zapisují je, aktualizují a případně odstraní jednotlivé nebo všechny záznamy.

Poslední je metoda pro odpojení se od databáze. Tu je vhodné zavolat nejpozději při ukončení aplikace, abychom zbytečně neubírali systémové zdroje a zároveň zvýšili zabezpečení dat.

Metody pro založení, aktualizaci a mazání dat jsou realizované pomocí volání metod objektu SQLiteDatabase. Je tomu tak pouze z důvodů demonstrace odlišného řešení, metody bychom mohli realizovat stejně jako v předchozím příkladu čistě pomocí volání SQL dotazů.

Ve třídě MainActivity si vytvoříme instanci naší knihovny. Odkážeme se na ní pomocí proměnné dbHelper a jejím prostřednictvím už jenom budeme volat potřebné metody. V předefinované události onCreate() zavoláme metodu pro načtení dat, která se připojí k databázi a prostřednictvím adaptéru naplní data do gridu na první záložce.

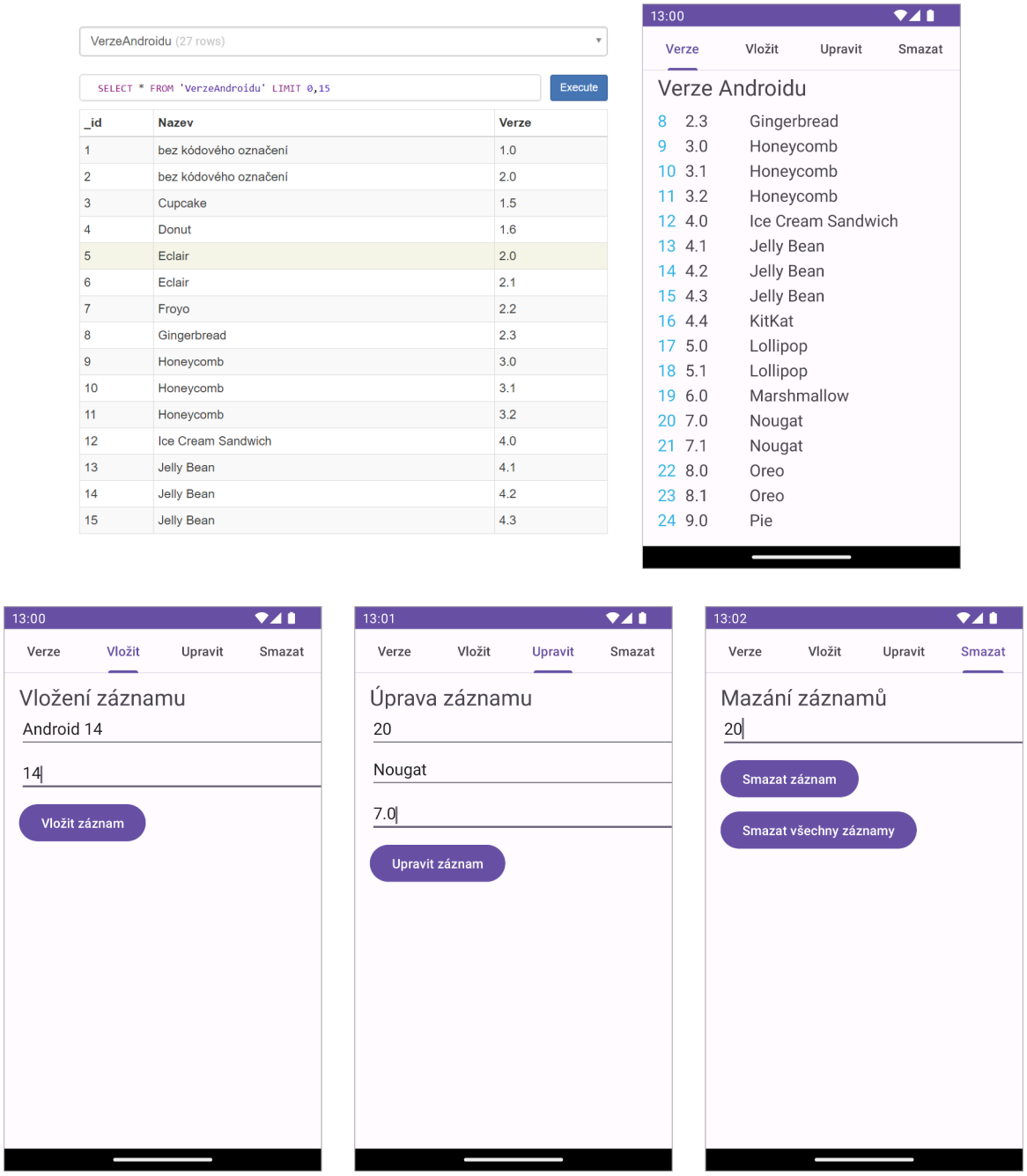
Připravíme si metody pro operace vložení, aktualizace a mazání dat. Tyto metody voláme z tlačítek na dalších záložkách. Každá z těchto metod si z formuláře vyzvedne potřebná data a zavolá příslušnou metodu ze třídy DatabaseHandler.

```
public class MainActivity extends AppCompatActivity {

    // proměnná s odkazem na fragment se seznamem verzi
    public View fragmentZalozkaVerze;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

    // nova instance adapteru
    TabAdapter mAdapter = new TabAdapter(this);

    // najdeme na formulari TabLayout a ViewPager
    TabLayout tabLayout = (TabLayout) findViewById(R.id.tabLayout);
    ViewPager2 viewPager = (ViewPager2) findViewById(R.id.viewPager);

    // propojeni ViewPager s adapterem
    viewPager.setAdapter(mAdapter);

    // nazvy zalozek
    final String[] zalozky = new String[]{"Verze", "Vložit",
                                          "Upravit","Smazat"};

    // propojeni TabLayoutu s ViewPager
    new TabLayoutMediator(tabLayout, viewPager,
            new TabLayoutMediator.TabConfigurationStrategy() {
                @Override public void onConfigureTab(TabLayout.Tab tab,
                                          int position) {
                    tab.setText(zalozky[position]);
                }
            }).attach();
    }

    // nacist data z DB tabulky, vytvorit adapter a propojit ho s gridView
    private void nacistData () {
        // pripojeni se k databazi
        dbHelper.pripojitDB();

        // test, jestli uz existuje zalozka verzi
        if (fragmentZalozkaVerze == null)
            return;

        // instance tridy DatabaseHandler
        DatabaseHandler dbHelper = new DatabaseHandler(
                                          fragmentZalozkaVerze.getContext());

        // pripojeni se k databazi
        dbHelper.pripojitDB();

        // nacteni dat z tabulky
        Cursor data = dbHelper.nacistData();

        // vytvorit instanci adapteru
        String[] arrFrom = new String[] {"_id", "Verze", "Nazev"};
```

```
        int[] arrTo = new int[] {R.id.polozka_ID, R.id.polozka_verze,
                                                 R.id.polozka_nazev};
        try {
            adapter = new SimpleCursorAdapter(this, R.layout.polozka_gridu,
                                                 data, arrFrom, arrTo, 0);
        }
        catch (Exception ex) {
            Toast.makeText(this, "Chyba: " + ex.getMessage(),
                                                 Toast.LENGTH_LONG).show();
        }

        // propojeni adapteru s kontejnerem GridView
        GridView gridView = (GridView) findViewById(R.id.gridViewVerzeAndroidu);
        gridView.setAdapter(adapter);

        // odpojeni se od databaze
        dbHelper.odpojitDB();
    }

    // udalost tlacitka Vlozit zaznam
    public void butVlozitZaznam(View view) {
        // pripojeni se k databazi
        dbHelper.pripojitDB();

        TextView txtVerze = (TextView) findViewById(R.id.textVlozitVerze);
        TextView txtNazev = (TextView) findViewById(R.id.textVlozitNazev);
        dbHelper.vlozitZaznam(txtNazev.getText().toString(),
                                                 txtVerze.getText().toString());
        Toast.makeText(this, "Záznam byl zapsaný do databáze",
                                                 Toast.LENGTH_SHORT).show();

        // znovunacteni dat do gridu a odpojeni se od DB
        nacistData();
        dbHelper.odpojitDB();
    }

    // udalost tlacitka Upravit zaznam
    public void butUpravitZaznam(View view) {
        // pripojeni se k databazi
        dbHelper.pripojitDB();

        TextView txtId = (TextView) findViewById(R.id.textUpravitId);
        TextView txtVerze = (TextView) findViewById(R.id.textUpravitVerze);
        TextView txtNazev = (TextView) findViewById(R.id.textUpravitNazev);
        dbHelper.upravitZaznam(txtId.getText().toString(),
                  txtNazev.getText().toString(), txtVerze.getText().toString());

        Toast.makeText(this, "Záznam byl zapsaný do databáze",
                                                 Toast.LENGTH_SHORT).show();

        // znovunacteni dat do gridu a odpojeni se od DB
        nacistData();
        dbHelper.odpojitDB();
    }

    // udalost tlacitka Smazat zaznam
    public void butSmazatZaznam(View view) {
        // pripojeni se k databazi
        dbHelper.pripojitDB();

        TextView txtId = (TextView) findViewById(R.id.textSmazatId);
```

```
        dbHelper.smazatZaznam(txtId.getText().toString());

        Toast.makeText(this, "Záznam byl vymazaný z databáze",
                                                Toast.LENGTH_SHORT).show();

        // znovunacteni dat do gridu a odpojeni se od DB
        nacistData();
        dbHelper.odpojitDB();
    }

    // udalost tlacitka Smazat zaznamy
    public void butSmazatZaznamy(View view) {
        // pripojeni se k databazi
        dbHelper.pripojitDB();

        dbHelper.smazatData();

        Toast.makeText(this, "Záznamy byly vymazané z databáze",
                                                Toast.LENGTH_SHORT).show();

        // znovunacteni dat do gridu a odpojeni se od DB
        nacistData();
        dbHelper.odpojitDB();
    }
}
```

V emulátoru nebo na koncovém zařízení by měl výsledek po kompilaci vypadat jako na obrázku 18.11.



Obrázek 18.11: Obsah databázové tabulky (prvních 15 záznamů) a aplikace, která má na každé záložce prvky pro jednu databázovou operaci

19 Média

Práce se zvukem a obrazem je jednou z uživatelsky nejatraktivnějších a často i nejvděčnějších. Pořizování fotografií a videí, nezřídka kombinovaných s jejich online publikováním, je ostatně fenoménem dnešní doby, který se díky mobilním zařízením stal dostupným doslova každému.

Z hlediska tvorby aplikací jde však o dost náročnou disciplínu, která rozhodně není vhodná pro psaní první aplikace. Na mikrofon a kameru je zkrátka potřeba mít už nějaké ostruhy. Jde o fyzická zařízení, která bývají zasažena nejen invencí a určitou tvůrčí rozmanitostí výrobců, ale také častým vylepšováním v nových verzích API. Během práce dochází ke kolizím při simultánních přístupech, je nutné využívat buffery, různé datové formáty a množství rozličných nastavení. Ke všemu nezřídka bývá obtížné identifikovat příčinu případných chyb. Toto všechno však bohatě vyvažují možnosti, které tato miniaturní zařízení vývojářům nabízejí.

Obecně se dá říct, že nejsnazší cestou jak pořídit zvuková nebo obrazová data je vyvolání vhodné aplikace pomocí Intentu (k tomu se podrobněji dostaneme v kapitole 24 Intent). Tento způsob ale prakticky neumožňuje programátorské přizpůsobení ani dostatečnou integraci do vlastní aplikace, takže se mu nyní nebudeme věnovat.

19.1 Zvuk

Prakticky všechna Android zařízení mají obvody pro zvukový výstup. Ať už do sluchátek, vestavěného, nebo externího reproduktoru. Systém podporuje několik formátů zvukových souborů, dává možnost přehrávat separátně systémové zvuky, upozornění a zvuky aplikací. Programátorovi umožňuje pracovat s nastavením hlasitosti a s událostmi připojení/odpojení sluchátek.

Jako příklad si uděláme jednoduchou aplikaci pro přehrávání zvuků. Budeme tedy potřebovat založit nový projekt, do kterého postupně, podle potřeby, přidáme jednotlivé komponenty.

19.1.1 Zvuk, 1. příklad

Základním typem jsou takzvané systémové zvuky. Slovo systémové znamená, že nepřicházíme do styku se zvukovým souborem, ale pouze s odkazem na melodie, které si uživatel volí sám v nastavení systému. Typicky vyzváněcí melodie na telefonu. Tu lze přehrát i z vlastní aplikace, pomocí knihovny RingtoneManager.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 19_Media
projekt MediaPlayer

Přidáme do aplikace první tlačítko a pomocí něj zavoláme metodu, například prehratZvuk1():

```
private Ringtone vyzvaneni;

public void prehratZvuk1(View sender) {
    if (vyzvaneni == null) {
        Uri vyzvaneniUri = RingtoneManager.getDefaultUri(
                                    RingtoneManager.TYPE_RINGTONE);
        vyzvaneni = RingtoneManager.getRingtone(
                                    getApplicationContext(), vyzvaneniUri);
        vyzvaneni.play();
    }
    else {
        vyzvaneni.stop();
```

```
            vyzvaneni = null;
        }
    }
```

Metoda využívá proměnnou vyzvaneni typu Ringtone, do které si ukládá odkaz na přehrávanou melodii. Nenechme se zmást názvem Ringtone, ve skutečnosti nemusí jít nutně o vyzvánění, ale třeba o zvuk upozornění nebo budíku. Metoda si nejprve ověří zda proměnná vyzvaneni není naplněna a pokud ne, tak do ní uloží odkaz získaný metodou getDefaultUri() knihovny Ringtone Manager. Který konkrétní typ zvuku nás zajímá, to jí sdělíme pomocí parametru – v našem případě jde o vyzvánění, tedy o konstantu TYPE_RINGTONE. Pro zvuk budíku je to TYPE_ALARM, pro upozornění TYPE_NOTIFICATION.

Ve chvíli kdy máme odkaz na vyzvánění, zavoláme metodu getRingtone(), které v parametru předáme kontext a zmíněný odkaz. Od toho okamžiku je v proměnné vyzvaneni vše potřebné a můžeme pomocí volání play() nebo stop() přehrávání spustit a zase zastavit. Pochopitelně jsou zohledněny i další systémová nastavení, v tomto případě zejména hlasitost. Pokud je hlasitost vyzvánění nastavena na nulu, tak samozřejmě není nic slyšet ani při přehrávání vyzvánění pomocí aplikace.

Jako jsme pro systémové zvuky použili knihovnu Ringtone, tak pro vlastní zvuky (ale také pro video) je k dispozici knihovna přehrávače MediaPlayer. Audio soubory využijeme kdykoliv, kdy chceme, aby aplikace disponovala vlastním zvukovým výstupem. A to od nejběžnějších zvuků typu kliknutí na ovládací prvek, které můžeme uplatnit prakticky v každé aplikaci, až po přehrávání hudby na pozadí, které najde uplatnění především v hrách.

Android podporuje hned několik formátů zvukových souborů. Určitě se neztratíme s typy AAC, AMR, MIDI nebo WAVE. Nejčastěji se ale zřejmě budeme setkávat s velmi rozšířeným formátem MP3. Pro naše potřeby se v Android Studiu ukládají tyto soubory do projektové složky app/res/raw.

Abychom mohli takový soubor přehrát, umístíme ho do zmíněné složky (v našem případě soubor nazvaný kliknuti.mp3), do formuláře přidáme tlačítko a vytvoříme metodu, kterou kliknutím na tlačítko zavoláme:

```
public void prehratZvuk2 (View sender) {
    // spustit prehravani zvuku
    MediaPlayer prehravac = MediaPlayer.create(this, R.raw.kliknuti);
    prehravac.start();

    Toast.makeText(getApplicationContext(), "Stisknuto tlačítko",
                                                Toast.LENGTH_SHORT).show();
}
```

Metoda nejprve vytvoří proměnnou prehravac typu MediaPlayer. Zároveň ji iniciuje pomocí volání metody create(), které předá v parametrech kontext a odkaz na zvuk. Tím získá knihovna MediaPlayer všechny potřebné informace. Je to jednoduché a funkční. Stačí už jenom zvuk přehrát pomocí volání start(). Jelikož jde o operaci asynchronní, neboli zvuk se přehrává na pozadí, zatímco program je zpracovávaný dál, je za instrukcí start() vyvolání toast zprávy, která se zobrazí ještě během přehrávání zvuku. Toto můžeme snadno ověřit zejména pokud zvolíme delší zvukový soubor. Při zkoušení je opět potřeba dát pozor na nastavení hlasitosti.

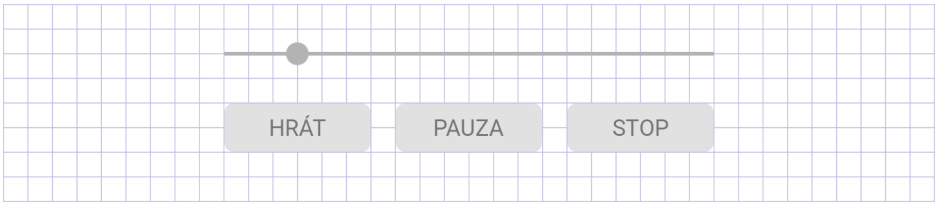
V některých případech potřebujeme vědět, že bylo přehrávání zvuku ukončeno. Například pokud chceme pokračovat přehráváním dalšího zvukového souboru. Jelikož jde o asynchronní operaci, řeší MediaPlayer tento problém vyvoláním události definované listenerem. Vyzkoušíme si

na dalším příkladu. Do adresáře raw umístíme vhodný soubor, v našem případě zvuk.mp3. Přidáme na formulář ještě jedno tlačítko a zavoláme jím následující metodu:

```
public void prehratZvuk3(View sender) {
    // spustit prehravani zvuku
    MediaPlayer prehravac = MediaPlayer.create(this, R.raw.zvuk);
    prehravac.start();

    // nastavit listener pro konec prehravani
    prehravac.setOnCompletionListener(new MediaPlayer.OnCompletionListener() {
        @Override
        public void onCompletion(final MediaPlayer mp) {
            Toast.makeText(getApplicationContext(), "Konec přehrávání",
                                                    Toast.LENGTH_SHORT).show();
        }
    });
}
```

Stejně jako v předchozím příkladu i nyní metoda vytvoří přehrávač, který spustí přehrávání zvukového souboru. Navíc si ale zaregistruje listener, který bude čekat na událost konce přehrávání. Jakmile k události dojde, zobrazí informaci ve formě toast zprávy. Zatímco v předchozím příkladu byla zpráva zobrazena hned po kliknutí na tlačítko, zde se s jejím zobrazením čeká až na konec přehrávání.

Na základě předchozích pokusů už není příliš těžké sestavit jednoduchý přehrávač. Pokusíme se o to. Měl by vypadat zhruba takto:



Obrázek 19.1: Očekávaný vzhled hudebního přehrávače

Budeme potřebovat hudební soubor, na formulář přidáme komponentu SeekBar a pod ni tři tlačítka – hrát, pauza a stop.

Co je potřeba udělat?

- pořídit si vhodný hudební soubor
- umístit do formuláře potřebné komponenty – tři tlačítka a SeekBar
- připravit tři metody které budou volány jednotlivými tlačítky
- zařídit, aby se SeekBar průběžně nastavoval podle pozice v přehrávané skladbě
- zařídit, aby se pozice přehrávání změnila, pokud uživatel nastaví na SeekBaru novou hodnotu

Jako hudební soubor poslouží prakticky libovolný, nejlépe MP3 audio soubor. Umístíme ho do složky app/res/raw. Formulář můžeme použít stávající, ten na kterém jsme si dělali tlačítka pro přehrávání. Pouze na něj přidáme SeekBar a pod něj tři další tlačítka.

Ve třídě MainActivity.java založíme tři metody, pojmenované například prehravacHrat(), prehravacPauza() a prehravacStop(). Každou z těchto metod budeme volat jedním z tlačítek na událost onClick(). Kromě již známé metody MediaPlayeru start(), použijeme nově ještě tyto metody:

prehravac.pause() - přeruší přehrávání skladby. Pokračovat lze pomocí prehravac.start()

prehravac.reset() - ukončí přehrávání skladby

prehravac.isPlaying() - vrátí true nebo false, podle toho jestli se skladba právě přehrává

prehravac.getDuration() - vrátí délku skladby v milisekundách

prehravac.getCurrentPosition() - vrátí pozici ve skladbě, opět v milisekundách

Nejdříve si ve třídě MainActivity založíme tři proměnné, které budeme následně využívat:

```
private MediaPlayer prehravac;
private SeekBar posuvnik;
private Handler handler = new Handler();
```

V proměnné prehravac si budeme ukládat ukazatel na MediaPlayer, tak abychom s ním mohli pracovat z různých metod. Proměnná posuvník bude obsahovat ukazatel na SeekBar, abychom ho nemuseli pokaždé zdlouhavě hledat na formuláři a proměnná handler ukazatel na časovač. K tomu se ještě dostaneme.

Metoda prehravacHrat() nastaví do MediaPlayeru skladbu a spustí ji. Postup již známe z předchozích příkladů. Mimo to nastaví na SeekBaru atribut max na počet milisekund trvání skladby. To proto, abychom SeekBaru mohli jednoduše předáním milisekundy přehrávané skladby nastavit polohu ukazatele, odpovídající přehrávané části.

Kromě toho musíme zařídit, aby se v pravidelných intervalech upravovala pozice ukazatele. To uděláme tak, že si nastavíme časovač, který bude v pravidelných intervalech volat metodu nastavitPosuvnik(). Ta bude kontrolovat pozici ve skladbě a nastavovat ji na SeekBaru pomocí atributu Progress.

Obě metody budou vypadat takto:

```
public void prehravacHrat(View sender) {
    if (prehravac != null)
        prehravac.reset();
    // nastavit zvukovy soubor a spustit prehravac
    prehravac = MediaPlayer.create(this, R.raw.android_latest_2013);
    prehravac.start();

    // iniciace posuvniku
    SeekBar posuvnik = (SeekBar) this.findViewById(R.id.seekBar);
    posuvnik.setMax(prehravac.getDuration());

    MainActivity.this.runOnUiThread(new Runnable() {
        @Override
        public void run() {
            nastavitPosuvnik();
            handler.postDelayed(this, 100);
        }
    });
```

```
}

// nastavi polohu posuvniku podle pozice v prehravane skladbe
private void nastavitPosuvnik() {
    if(prehravac != null)
        posuvnik.setProgress(prehravac.getCurrentPosition());
    else
        posuvnik.setProgress(0);
}
```

Metoda pro přerušení přehrávání bude již o poznání jednodušší. Pouze zjistíme jestli se skladba právě přehrává a pokud ano, zavoláme metodu pause(). A pokud se nepřehrává, tak naopak pomocí start() její přehrávání obnovíme.

```
public void prehravacPauza(View sender) {
    if (prehravac == null)
        return;

    if (prehravac.isPlaying()) {
        prehravac.pause();
    } else {
        prehravac.start();
    }
}
```

Složitá není ani metoda pro ukončení přehrávání. Nejprve ověří jestli je přehrávač iniciovaný a pokud ano, tak zastaví přehrávání pomocí reset() a proměnnou prehravac uvolní.

```
public void prehravacStop(View sender) {
   if (prehravac == null)
        return;

    prehravac.reset();
    prehravac = null;
}
```

Tím máme pohromadě téměř vše. Zbývá jediná drobnost. Zařídit, aby při změně pozice SeekBaru změnil svoji pozici ve skladbě i přehrávač. Toho docílíme pomocí listeneru který připojíme k SeekBaru a který bude čekat na událost změny polohy posuvníku. Jakmile k ní dojde, listener zavolá metodu onProgressChanged(), která zjistí jestli změnu udělal uživatel (tuto informaci dostane v parametru události) a přenastaví pozici ve skladbě pomocí metody přehrávače seekTo().

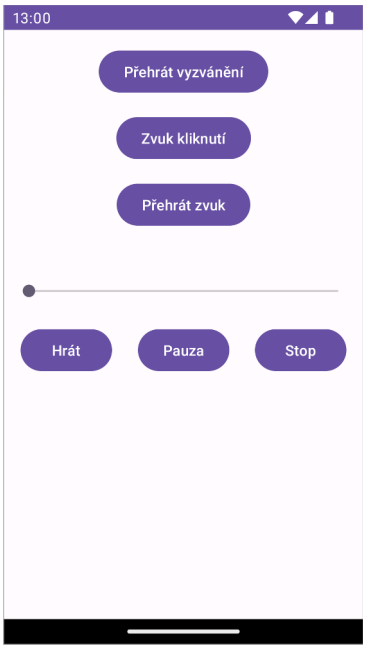
Při vytváření listeneru musíme předefinovat všechny tři jeho události - onProgressChanged(), kterou využijeme pro zachycení události změny posuvníku, onSartTrackingTouch(), která detekuje začátek kliknutí na posuvník a onStopTrackingTouch(), která detekuje jeho konec. Nyní využijeme pouze první událost, ale kdybychom chtěli například pípnout nebo přerušit přehrávání v době pohybu posuvníku, hodily by se i zbývající dvě.

Toto všechno dáme do metody kterou si nazveme například inicializacePosuvniku() a tu zavoláme v události onCreate() třídy MainActivity. To proto, aby se volala vždy přesně jednou, při startu aktivity.

```
private void inicializacePosuvniku() {
    // najdeme posuvnik ve formulari
    posuvnik = (SeekBar) this.findViewById(R.id.seekBar);
```

```
    // vytvorime listener ktery ceka na zmenu polohy ukazatele
    posuvnik.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
        @Override
        public void onStartTrackingTouch(SeekBar seekBar) {
        }

        @Override
        public void onStopTrackingTouch(SeekBar seekBar) {
        }

        @Override
        public void onProgressChanged(SeekBar seekBar, int pozice,
                                                    boolean odUzivatele) {
            if (prehravac == null)
                return;
            if(odUzivatele)
                prehravac.seekTo(pozice);
        }
    });
}
```

Výsledná aplikace by měla vypadat přibližně jako na obrázku 19.2. Klikáním na tlačítka by měla přehrávat systémové zvuky a zvukové soubory.



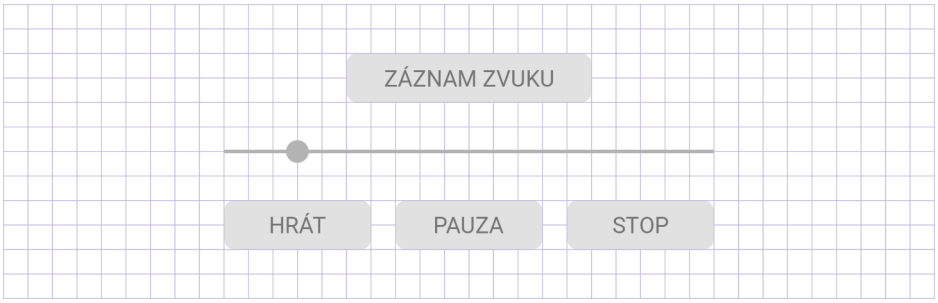
Obrázek 19.2: Hudební přehrávač, spuštěný na emulátoru.

19.1.2 Zvuk, 2. příklad

Tak jako knihovna MediaPlayer umožňuje zvuky přehrávat, tak lze pomocí knihovny MediaRecorder zvukový záznam také pořídit. Pro nahrávání jsou ale o něco omezenější možnosti, co se týká zvukových formátů. Zatímco přehrát MP3 není žádný problém, pro záznam je potřeba zvolit jiný formát, například oblíbený amr.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 19_Media
projekt MediaRecorder

Vše potřebné si ukážeme na aplikaci, která bude obsahovat stejný přehrávač jako v předchozím příkladu, jen nebude načítat zvukový soubor, ale soubor který si sama vytvoří a uloží do externího úložiště. Navíc přibude tlačítko pro záznam zvuku, které spustí MediaRecorder a zaznamená zvuk z mikrofonu. Ten pak ve formě amr souboru uloží do zmíněného externího úložiště.



Obrázek 19.3: Očekávaný vzhled hudebního přehrávače se záznamem zvuku.

Budeme potřebovat zdrojové kódy přehrávače z předchozího příkladu, včetně komponent na formuláři. K těm přidáme tlačítko Záznam zvuku.

Co je potřeba udělat?

- do nového projektu dát přehrávač z předchozího příkladu
- na formulář přidat tlačítko Záznam zvuku
- upravit přehrávač tak, aby načítal soubor z externího úložiště
- přidat do manifestu oprávnění pro nahrávání z mikrofonu a pro přístup do externího úložiště
- napsat rutinu pro nahrávání přes mikrofon a uložení zvuku do souboru

Přenést zdrojové kódy přehrávače z předchozího příkladu je poměrně jednoduché, stačí zkopírovat obsah souborů activity_main.xml a MainActivity.java a vložit je do nového projektu. V designeru přidáme na formulář tlačítko a zavoláme jím zatím neexistující metodu zaznamZvuku().

Kód, který načítal mp3 soubor z raw adresáře, musíme upravit tak, aby použil amr soubor z externího úložiště. Do metody prehravacHrat() přidáme instrukci pro získání cesty k souboru a tu potom vložíme ve formě vstupního parametru do přehrávače:

```
Uri souborUmisteni = Uri.parse(Environment.getExternalStorageDirectory()
                                                .getPath()+"/Music/test.amr");
prehravac = MediaPlayer.create(this, souborUmisteni);
```

Pro ověření základní funkčnosti to stačí. V praxi bychom však kvůli robustnosti aplikace měli přidat ještě ověření, jestli soubor opravdu existuje.

Jelikož je přístup k externímu úložišti a také k mikrofonu omezen uživatelským oprávněním, musíme do manifestu, tedy do souboru app/manifests/AndroidManifest.xml, vložit žádost o přístup k těmto zařízením. Jelikož bude aplikace ukládat data do externího úložiště, je dobré zohlednit také verzi systému na kterém poběží a raději dát všechna oprávnění, tak, jak je popsáno v kapitole 18.3 Externí úložiště, External Storage:

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE" />
```

Po spuštění, ať už na emulátoru nebo ve fyzickém zařízení, nesmíme zapomenout tato práva aplikaci nastavit, bez toho by nahrávání skončilo okamžitě chybou.

Nejdříve si ve třídě MainActivity založíme dvě nové proměnné, které budeme následně využívat:

```
private MediaRecorder zaznamnik;
private Button tlacitkoNahravani;
```

V proměnné zaznamnik se bude ukládat ukazatel na MediaRecorder a do proměnné tlacitkoNahravani uložíme v události onCreate() ukazatel na nové tlačítko, ve kterém budeme podle potřeby měnit text:

```
tlacitkoNahravani = (Button)this.findViewById(R.id.button1);
```

Založíme si novou metodu zaznamZvuku(), na kterou se už odkazuje událost onClick() z tlačítka na formuláři. Nejdřív otestujeme, jestli už je vytvořena instance zvukového záznamníku. Pokud ne, tak to znamená že záznamník není spuštěný a my mu musíme instanci vytvořit. Zároveň jí nastavíme zdroj zvuku, tedy AudioSource.MIC pro mikrofon, formát výstupního zvukového souboru a místo kam má být uložený.

Předtím, než nahrávání spustíme, musíme pro ověření nastavení zavolat metodu prepare(). Tato metoda se volá zásadně uvnitř bloku try – catch, abychom mohli ošetřit případnou výjimku, ke které práce s periferií může snadno vést.

Pokud je vše v pořádku, můžeme zavolat metodu start() a zvuk se začne nahrávat. Hned poté změníme text na tlačítku tak, aby bylo zřejmé že záznam běží a že je ho možné tím samým tlačítkem zase zastavit.

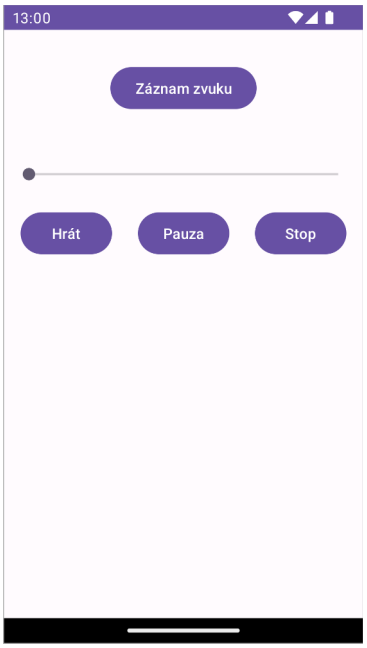
Záznam se zastaví zavoláním metody stop() a následným nastavením null hodnoty do proměnné zaznamnik, aby při příštím kliknutí bylo jasné, že je potřeba vytvořit novou instanci MediaRecorderu. Také se změní text na tlačítku, které tak bude uživateli signalizovat připravenost k pořízení dalšího zvukového souboru. Nakonec můžeme pomocí toast zprávy uživateli sdělit, že bylo nahrávání ukončeno.

Metoda bude vypadat takto:

```
public void zaznamZvuku(View sender) {
    if (zaznamnik == null) {
        // zahajit zaznam zvuku z mikrofonu
        zaznamnik = new MediaRecorder();
        zaznamnik.setAudioSource(MediaRecorder.AudioSource.MIC);
        zaznamnik.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
        zaznamnik.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
        zaznamnik.setOutputFile("/sdcard/Music/test.amr");
        try {
            zaznamnik.prepare();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
            zaznamnik.start();

            //zmenit text tlacitka
            tlacitkoNahravani.setText("Konec nahrávání");
        }
        else {
            // ukoncit zaznam zvuku z mikrofonu
            zaznamnik.stop();
            zaznamnik = null;
            //zmenit text tlacitka
            tlacitkoNahravani.setText("Záznam zvuku");

            // informace o ukonceni nahravani
            Toast.makeText(getApplicationContext(), "Konec záznamu.",
                                                Toast.LENGTH_SHORT).show();
        }
}
```

Výsledná aplikace bude vypadat přibližně jako na obrázku 19.4. Kliknutím na tlačítko Záznam zvuku by měla zvuk nahrát a uložit do souboru a po kliknutí na tlačítko Hrát by ho měla začít přehrávat.



Obrázek 19.4: Hudební přehrávač se záznamem zvuku, spuštěný na emulátoru.

19.2 Video

Naprostá většina Android zařízení bývá vybavena alespoň jednou kamerou, která se dá kromě pořizování fotografií použít také pro záznam videosekvencí. Android API proto podporuje práci s několika kamerami a množstvím funkcí, které hardwarově poskytují.

Pro zobrazení videa jsou k dispozici tři základní komponenty, SurfaceView, která je sama o sobě vlastně takovou malou aplikací, jež se spustí nad uživatelským rozhraním ve vlastním (byť neviditelném) okně. Takže opticky vypadá jako součást aplikace, ale bohužel tím pádem neumožňuje libovolné zásahy do přehrávaného videa a také je komplikované její využití

v RecyclerView nebo ScrollView. Výborně se však dá použít, spolu s MediaPlayerem, jako spolehlivý přehrávač videosekvencí.

Další komponentou je VideoView, která je oděděná od SurfaceView a rozšířená o MediaPlayer. Takže má všechny základní vlastnosti SurfaceView a k přehrání videa samostatný MediaPlayer nepotřebuje. Představuje asi nejjednodušší způsob, jak v aplikaci videa přehrávat. Pro vývojáře však opět s mnoha omezeními.

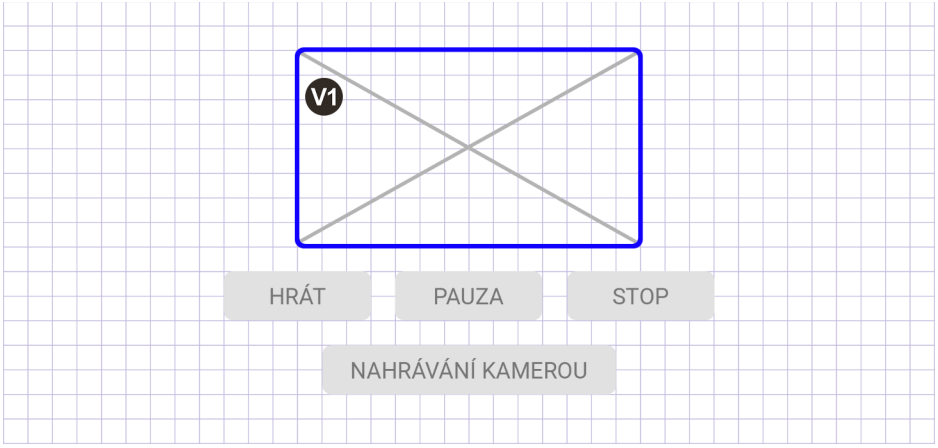
Omezení předchozích dvou komponent řeší TextureView. Ta je regulérním prvkem uživatelského rozhraní, umí vše co SurfaceView a navíc využívá hardwarovou akceleraci, dokáže zobrazit videostreamy, podporuje nejrůznější zásahy do obrazu, od vkládání vlastních prvků po kompletní transformace. Hodí se pro zkušenější programátory, kteří potřebují mít nad zobrazovanými sekvencemi kontrolu a chtějí využít bohatých možností zásahů do zobrazovaného videa.

19.2.1 Video, 1. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 19_Media
projekt Video1

Napíšeme si jednoduchou aplikaci, která nahraje videosekvenci z hlavní kamery zařízení a potom ji, stejně jako v příkladu s nahráváním zvuku, uživateli přehraje. Prozatím se, kvůli jednoduchosti, nebudeme zabývat zobrazováním preview, přepínáním kamer, otáčením obrazu, ani nastavováním dalších parametrů. Nahrané video uložíme do externího úložiště, ve formátu mp4. Pro záznam videa použijeme knihovnu MediaRecorder a pro zobrazení videa, z důvodu maximální jednoduchosti, komponentu VideoView. Mediaplayer nebudeme potřebovat, je integrovaný do VideoView.

Výsledná aplikace by měla vypadat zhruba takto:



Obrázek 19.5: Očekávaný vzhled videopřehrávače. V1 je komponenta VideoView.

Co je potřeba udělat?

- do manifestu přidat žádost o příslušná oprávnění
- vytvořit jednoduchý formulář s ovládacími prvky a komponentou pro zobrazení videa
- kvůli uložení videa požádat o povolení přístupu k externím souborům
- nastavit vlastnosti kamery tak, aby byla schopna záznam pořídit a uložit

Nejprve tedy přidáme do manifestu povolení pro přístup ke kameře a k nahráváni zvuku a obrazu:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.RECORD_VIDEO" />
```

To ale není všechno. Abychom mohli ukládat zaznamenané video na paměťové zařízení, musíme požádat ještě o právo zapisovat do externí paměti. Je dobré zohlednit verzi systému na kterém aplikace poběží a raději dát všechna oprávnění, tak, jak je popsáno v kapitole 18.3 Externí úložiště, External Storage:

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE" />
```

Způsob získání povolení přístupu k externím souborům je stejný jako v příkladu 18.3 Externí úložiště, External Storage. Použijeme metodu, která uživatele o přístup jednorázově požádá. Může se jmenovat třeba povoleniPristupuKExternimSouborum() a volat ji můžeme hned při vytvoření aktivity MainActivity, v události onCreate().

V activity_main.xml vytvoříme formulář, který bude obsahovat komponentu VideoView pro zobrazení přehrávaného videozáznamu, tři tlačítka, Přehrát, Stop a Pauza pro přehrávání záznamu a jedno tlačítko, které spustí záznam videa. VideoView je k dispozici v paletě komponent, v kategorii Widgets, stačí ji tedy do formuláře jednoduše přetáhnout a případně dokonfigurovat.

Připravíme si, v MainActivity.java metodu inicializace kamery:

```
private void inicializaceKamery() {

    // nova instance MediaRecorderu
    kamera = new MediaRecorder();
    kamera.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
    kamera.setVideoSource(MediaRecorder.VideoSource.CAMERA);

    // nova instance bufferu
    SurfaceTexture sft = new SurfaceTexture(0);
    Surface sf = new Surface(sft);
    kamera.setPreviewDisplay(sf);

    // nastaveni profilu kamery
    CamcorderProfile cpHigh = CamcorderProfile.get(0,
                                        CamcorderProfile.QUALITY_HIGH);
    kamera.setProfile(cpHigh);

    // soubor s ulozenymi daty
    kamera.setOutputFile(Environment.getExternalStorageDirectory().getPath()
                                        + "/Movies/test.mp4");
}
```

První akcí je tedy nová instance třídy MediaRecorder, nastavení zdroje zvuku a obrazu, následuje nová instance SurfaceTexture, která se stará o bufferování nahrávaného toku dat, získání profilu s nastavením vysoké kvality záznamu a nakonec nasměrování nahraných dat do souboru.

Aby bylo možné metodu zavolat, potřebujeme deklaraci proměnné třídy „kamera“. A také událost kameraNahravani(), kterou vyvolá tlačítko pro nahrávání:

```
private MediaRecorder kamera;

public void kameraNahravani (View sender) {
    if (kamera == null) {
        try {
            inicializaceKamery();
            kamera.prepare();
        } catch (IOException e) {
            e.printStackTrace();
            Toast.makeText(this, "Chyba inicializace: " + e.getMessage(),
                                                    Toast.LENGTH_LONG).show();
        }
        try {
            kamera.start();
        } catch (Exception e) {
            e.printStackTrace();
            Toast.makeText(this, "Chyba při startu záznamu: " + e.getMessage(),
                                                    Toast.LENGTH_LONG).show();
        }
    }
    else {
        kamera.stop();
        kamera = null;
    }
}
```

Tlačítko se používá jak pro začátek záznamu, tak pro jeho ukončení. Jestli už byl záznam spuštěn, se pozná podle stavu proměnná třídy kamera. Obsahuje-li hodnotu null, provede se nastavení kamery a spuštění nahrávání. Pokud proměnná obsahuje cokoliv jiného než prázdnou hodnotu, neboli pořizuje záznam, tak se nahrávání ukončí.

Předtím, než nahrávání spustíme, musíme pro ověření nastavení zavolat metodu prepare(). Tato metoda se volá zásadně uvnitř bloku try – catch, abychom mohli ošetřit případnou výjimku, ke které práce s periferií může snadno vést. Pokud je vše v pořádku, můžeme zavolat metodu start() a video se začne nahrávat. Metoda start() nemusí být nutně ošetřená pro případ výjimky, ale vzhledem k tomu, že je kamera periférií která může z různých důvodů vykazovat chybu, je lepší výjimky zachytávat a zpracovávat.

Události tlačítek pro ovládání přehrávače videa jsou stejné jako u příkladu hudebního přehrávače. Každé má svoji událost a ta buď přehrávání spouští, pozastavuje nebo ukončí.

```
private VideoView prehravac;

// zahajit prehravani
public void prehravacPrehrat (View sender) {
    prehravac = (VideoView)findViewById(R.id.videoView);

    Uri souborUmisteni = Uri.parse(Environment.getExternalStorageDirectory()
                                        .getPath()+ "/Movies/test.mp4");
    prehravac.setVideoURI(souborUmisteni);
    prehravac.requestFocus();
    prehravac.start();
}
```

```
// prerusit/obnovit prehravani
public void prehravacPauza (View sender) {

    if (prehravac == null)
        return;

    if (prehravac.isPlaying()) {
        prehravac.pause();
    } else {
        prehravac.start();
    }
}

// ukoncit prehravani
public void prehravacStop (View sender) {

    if (prehravac == null)
        return;

    prehravac.stopPlayback();
    prehravac = null;
}
```

Před prvním spuštěním nesmíme zapomenout nastavit aplikaci příslušná oprávnění. Při ladění aplikace může dojít k mnoha chybám. Pokud není jasné co vlastně nefunguje, může se hodit ladit zvlášť část přehrávání, ukládání a nahrávání. Přehrávání se dá zjednodušit vhodným krátkým videosouborem, třeba staženým z internetu, který se umístí do projektové složky app/res/raw a přehrávači se nastaví jako zdroj dat k přehrání. Podobně je možné odladit ukládání dat do externího úložiště a až obě tyto části prokazatelně fungují, tak teprve doladit nejcitlivější část, samotný záznam.

Výsledná aplikace by měla vypadat přibližně jako na obrázku 19.6. Kliknutím na tlačítko Nahrávání kamerou by měla nahrát videosekvenci a uložit ji do souboru. Tlačítkem Přehrát by měla začít video přehrávat.

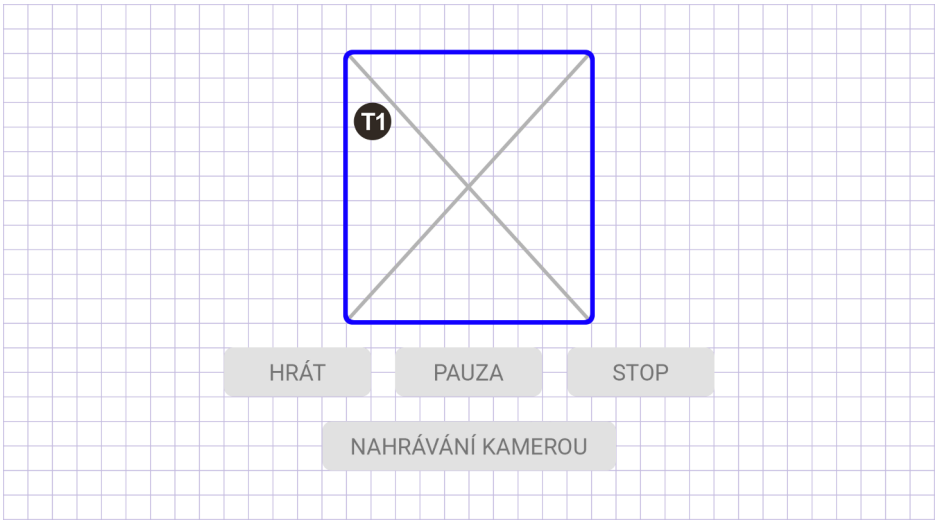
Obrázek 19.6: Přehrávač se záznamem videa, spuštěný na emulátoru.

19.2.2 Video, 2. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 19_Media
projekt Video2

Zatímco v předchozí kapitole jsme pro zobrazení videa použili relativně jednoduchou a bezproblémovou komponentu VideoView, nyní zkusíme video zobrazit v kontejneru TextureView. Základní rozdíl je v možnostech které tyto prvky uživatelského rozhraní nabízejí. Zatímco VideoView je poměrně jednoduchá komponenta, která ochotně zobrazí téměř jakýkoliv zdroj bez velkého nastavování, ale současně bez možnosti přizpůsobení, tak řízení TextureView je výrazně složitější, ovšem s jasným benefitem – video budeme mít mnohem více ve vlastních rukou. Můžeme do něj vkládat ikony, masky, dělat animace, upravovat obraz nebo ho transformovat do rozličných podob. Zaplatíme za to mnohem složitějším řízením, využívajícím události a zpětně volané (callback) metody.

Aplikace bude vypadat podobně jako ta z předchozího příkladu a bude mít i stejné základní funkce, pouze postavené na komponentě TextureView.



Obrázek 19.7: Očekávaný vzhled videopřehrávače. V1 je komponenta TextureView.

Co je potřeba udělat?

- do manifestu přidat žádost o příslušná oprávnění
- vytvořit jednoduchý formulář s ovládacími prvky a komponentou pro zobrazení videa
- kvůli uložení videa požádat o povolení přístupu k externím souborům
- nastavit vlastnosti kamery tak, aby byla schopna záznam pořídit a uložit

Do manifestu tedy musíme, stejně jako minule, vložit žádosti o povolení přístupu ke kameře, zvuku, videu a externímu úložišti:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.RECORD_VIDEO" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE" />
```

V activity_main.xml vytvoříme formulář, do kterého vložíme z kategorie Widgets v paletě komponent prvek TextureView pro zobrazení videozáznamu, tři tlačítka, Přehrát, Stop a Pauza pro přehrávání záznamu a jedno tlačítko pro záznam videa.

Povolení přístupu k externí souborům je stejné jako v předchozím příkladu. Opět použijeme metodu povoleniPristupuKExternimSouborum(), která uživatele o přístup jednorázově požádá. Volat ji opět budeme hned při vytvoření aktivity MainActivity, v události onCreate().

Začneme funkcí nahrávání videa. To se prakticky neliší od předchozího příkladu. Připravíme si deklaraci lokální proměnné třídy mediaRecorder a metodu, která se postará o inicializaci kamery:

```
private MediaRecorder mediaRecorder;

private void inicializaceKamery() {
    // nova instance MediaRecorderu + jeho nastaveni
    mediaRecorder = new MediaRecorder();
```

```
    mediaRecorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
    mediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
    mediaRecorder.setOrientationHint(90);

    // nova instance bufferu
    SurfaceTexture sft = new SurfaceTexture(0);
    Surface sf = new Surface(sft);
    mediaRecorder.setPreviewDisplay(sf);

    // nastaveni profilu kamery
    CamcorderProfile cpHigh = CamcorderProfile.get(0,
                                        CamcorderProfile.QUALITY_HIGH);
    mediaRecorder.setProfile(cpHigh);

    // soubor s ulozenymi daty
    mediaRecorder.setOutputFile(Environment.getExternalStorageDirectory()
                                        .getPath()+"/Movies/test.mp4");
}
```

Proměnnou mediaRecorder budeme používat k ovládání nahrávání. Samotnou inicializací se rozumí vytvoření instance MediaRecorderu, který se o nahrávání bude starat, instance bufferu, do kterého se nahraná data budou ukládat, nastavení profilu kamery a souboru do kterého bude video na závěr uloženo.

Pro spuštění a ukončení pořizování nahrávky potřebujeme metodu, pojmenujme si ji opět kameraNahravani(), kterou budeme volat stisknutím tlačítka na formuláři:

```
// zahaji/ukonci zaznam videa
public void kameraNahravani (View sender) {

    // odkaz na tlacitko
    Button butNahravani = (Button) findViewById(R.id.butNahravani);

    // spusteni nahravani
    if (mediaRecorder == null) {
        try {
            inicializaceKamery();
            mediaRecorder.prepare();
        } catch (IOException e) {
            e.printStackTrace();
            Toast.makeText(this, "Chyba inicializace: " + e.getMessage(),
                                                Toast.LENGTH_LONG).show();
        }
        try {
            mediaRecorder.start();
            butNahravani.setText("Konec nahrávání");
        } catch (Exception e) {
            e.printStackTrace();
            Toast.makeText(this, "Chyba při startu záznamu: " + e.getMessage(),
                                                Toast.LENGTH_LONG).show();
        }
    }
    else {
        mediaRecorder.stop();
        mediaRecorder = null;
        butNahravani.setText("Nahrávání kamerou");
    }
}
```

Metoda ověří, jestli už byla kamera inicializována a spuštěna. Pokud dosud nebyla, zavolá metodu inicializaceKamery() a kameru spustí. Jestliže už předtím inicializovaná byla, tak ji naopak zastaví. V obou případech změní také adekvátním způsobem text na tlačítku. Tím je nahrávání videa zajištěné.

Nyní nás čeká přehrávač. Přidáme dvě další proměnné třídy, jednu pro uchování instance MediaPlayeru a druhou pro instanci Surface. Surface je třída, která má na starosti vykreslení obrazu na displeji zařízení.

```
private MediaPlayer prehravac;
private Surface sface;
```

Připravíme si, v MainActivity.java metodu inicializace přehrávače:

```
private void inicializacePrehravace () {
    // kontrola souboru s ulozenym videem
    File file = new File(Environment.getExternalStorageDirectory()
                                        .getPath()+"/Movies/test.mp4");
    if(!file.exists()) {
        Toast.makeText(this, "Soubor s videem neexistuje",
                                        Toast.LENGTH_SHORT).show();
        return;
    }

    // instance prehravace a jeho spusteni
    try {
        prehravac = new MediaPlayer();
        prehravac.setDataSource(Environment.getExternalStorageDirectory()
                                        .getPath()+"/Movies/test.mp4");
        prehravac.setSurface(sface);
        prehravac.prepareAsync();
        prehravac.setAudioStreamType(AudioManager.STREAM_MUSIC);

        prehravac.setOnPreparedListener(new MediaPlayer.OnPreparedListener() {
            @Override
            public void onPrepared(MediaPlayer mp) {
                prehravac.start();
            }
        });
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (SecurityException e) {
        e.printStackTrace();
    } catch (IllegalStateException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Metoda nejprve zkontroluje, jestli je k dispozici soubor s videem. Pokud ano, vytvoří novou instanci MediaPlayeru, nastaví mu soubor jako zdroj a prostřednictvím posluchače události spustí přehrávání. Spuštění přehrávače je v bloku try-catch, protože může vyvolat celou řadu výjimek, které bude potřeba identifikovat.

Abychom mohli s přehrávačem pracovat, musíme rozšířit MainActivity o implementaci rozhraní posluchače TextureView:

```
public class MainActivity extends AppCompatActivity
                        implements TextureView.SurfaceTextureListener {
```

V události onCreate() přidáme na komponentu TextureView registraci posluchače Surface. Takže událost bude nyní vypadat takto:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    povoleniPristupuKExternimSouborum();

    TextureView textureView = (TextureView) findViewById(R.id.textureView);
    textureView.setSurfaceTextureListener(this);
}
```

S implementací posluchače TextureView budeme potřebovat předefinovat jeho události:

```
@Override
public void onSurfaceTextureAvailable(SurfaceTexture surface, int width,
                                                    int height) {
}

@Override
public void onSurfaceTextureSizeChanged(SurfaceTexture surfaceTexture, int i,
                                                    int i1) {
}

@Override
public boolean onSurfaceTextureDestroyed(SurfaceTexture surfaceTexture) {
    return false;
}

@Override
public void onSurfaceTextureUpdated(SurfaceTexture surfaceTexture) {
}
```

Z těchto událostí nás ale nyní bude zajímat pouze onSurfaceTextureAvailable(), která je vyvolaná v okamžiku, kdy je Surface připraven k použití. Doplníme ji tedy o volání inicializace kamery:

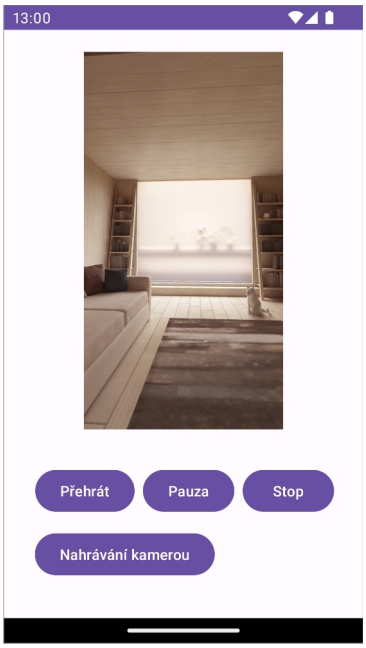
```
@Override
public void onSurfaceTextureAvailable(SurfaceTexture surface, int width,
                                                        int height) {
    sface = new Surface(surface);
    inicializacePrehravace ();
}
```

Události tlačítek pro ovládání přehrávače videa jsou stejné jako u předchozího příkladu. Každé má svoji událost a ta buď přehrávání spouští, pozastavuje nebo ukončí.

```
public void prehravacPrehrat (View sender) {
    if (prehravac != null) {
        prehravac.release();
        prehravac = null;
    }
    inicializacePrehravace();
}

public void prehravacPauza (View sender) {
    if (prehravac == null)
        return;

    if (prehravac.isPlaying()) {
        prehravac.pause();
    } else {
        prehravac.start();
    }
}

public void prehravacStop (View sender) {
    if (prehravac == null)
        return;

    prehravac.stop();
    prehravac.release();
    prehravac = null;
}
```

MediaPlayer má základní instrukce start(), pause(), stop() a release(), které k ovládání využijeme. Metoda prehravacPrehrat() spustí inicializaci přehrávače a ta zároveň spouští samotné přehrání. Pokud přehrávač v okamžiku volání běží, tak ho nejdřív ukončí a až poté znovu spustí. Metoda prehravacPauza() přehrávač v závislosti na aktuálním stavu buď zastavuje nebo obnovuje přehrávání. A nakonec metoda prehravacStop() video zastaví a uvolní instanci Mediaplayeru.

Výsledná aplikace by měla vypadat přibližně jako na obrázku 19.8. Kliknutím na tlačítko Záznam zvuku by měla zvuk nahrát a uložit do souboru a po kliknutí na tlačítko Hrát by ho měla začít přehrávat.



Obrázek 19.8: Videopřehrávač, využívající komponentu TextureView, spuštěný na emulátoru.

19.2.3 Video, 3. příklad

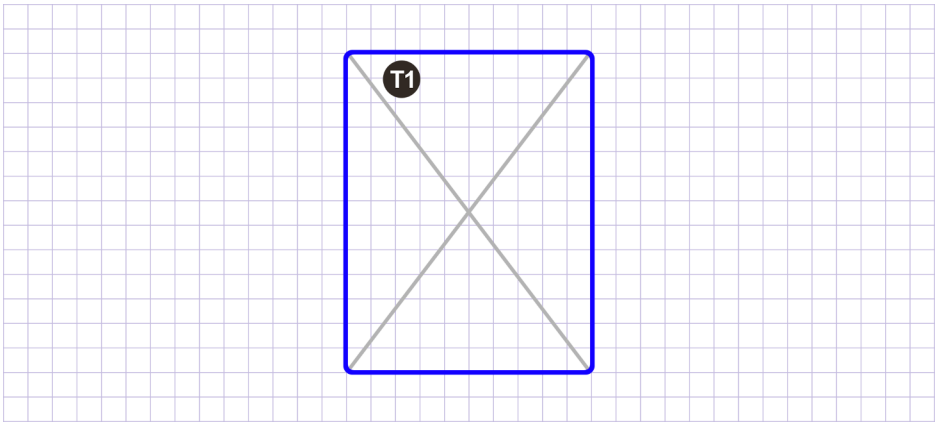
Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 19_Media
projekt Video3

V předchozích příkladech jsme úsilí zaměřili na přehrávání videosekvencí. Nahrávání za nás řešil MediaRecorder. Pro lepší uživatelský komfort se ale hodí, aby byla data poskytovaná kamerou převáděna na obraz okamžitě, ve formě preview. Tj. před, nebo během nahrávání. Účel je zřejmý, uživatel chce vidět scénu ještě předtím, než začne záznam pořizovat a během něj potřebuje kontrolovat nahrávaný obsah.

Preview se přitom dá využít nejen pro video, ale také pro focení, kde uživatel rovněž potřebuje sledovat scénu před pořízením snímku.

Sestavíme si aplikaci, která bude ve formě preview zobrazovat na displeji zařízení data aktuálně poskytovaná kamerou. Pro zobrazení preview použijeme opět komponentu TextureView.

Výsledná aplikace by měla vypadat zhruba takto:



Obrázek 19.9: Očekávaný vzhled videopřehrávače. Jedinou komponentou je T1, TextureView.

Co je potřeba udělat?

- do manifestu přidat žádost o příslušná oprávnění
- vytvořit jednoduchý formulář s ovládacími prvky a komponentou pro zobrazení preview
- nastavit vlastnosti kamery tak, aby byla schopna předávat data pro zobrazení preview

Manifest upravíme stejně jako v předchozích dvou příkladech. Vložíme do něj žádosti o povolení přístupu ke kameře, zvuku a videu. Jelikož nebudeme záznam nikam ukládat, právo na zápis do externího úložiště tentokrát nebude potřeba.

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.RECORD_VIDEO" />
```

V activity_main.xml vytvoříme formulář, do kterého vložíme z kategorie Widgets v paletě komponent prvek TextureView pro zobrazení videozáznamu. Žádné další ovládací prvky nebudou potřeba.

Základem bude metoda spustitKameru(), která se postará o inicializaci a spuštění kamery:

```
private Size velikostObrazu;

private void spustitKameru() {

    // test pristupu ke kamere
    if (ActivityCompat.checkSelfPermission(this,
            Manifest.permission.CAMERA) != PackageManager.PERMISSION_GRANTED) {
        Toast.makeText(this, "Aplikace nemá  přístup ke kameře",
            Toast.LENGTH_LONG).show();
        return;
    }

    CameraManager manager =
            (CameraManager) getSystemService(Context.CAMERA_SERVICE);
    try {
        // zjistit ID hlavni kamery
```

```
            String kameraId = manager.getCameraIdList()[0];
            CameraCharacteristics characteristics =
                manager.getCameraCharacteristics(kameraId);
            StreamConfigurationMap vlastnostiKamery = characteristics
                .get(CameraCharacteristics.SCALER_STREAM_CONFIGURATION_MAP);

            // zjistit rozliseni kamery
            velikostObrazu = vlastnostiKamery
                .getOutputSizes(SurfaceTexture.class)[0];

            // spusteni kamery
            manager.openCamera(kameraId, stateCallback, null);

        } catch (CameraAccessException e) {
            e.printStackTrace();
            Toast.makeText(this, "Chyba: " + e.getMessage(),
                Toast.LENGTH_LONG).show();
        }
    }
```

Metoda používá proměnnou třídy velikostObrazu, do které uloží rozměry snímaného obrazu v obou osách. Tyto hodnoty využijeme až později.

Prvním krokem je kontrola nezbytných oprávnění pro přístup ke kameře. Potom získáme odkaz na službu CameraManager, která slouží k detekci kamer v zařízení, popisu jejich vlastností a připojení k nim. Pomocí CameraManageru načteme seznam kamer, vezmeme ID první z nich (budeme předpokládat, že zařízení vždy alespoň jednu kameru obsahuje) a uložíme ho do lokální proměnné kameraId. K této kameře získáme seznam jejich vlastností a z nich potom vybereme velikost obrazu. Tu budeme později potřebovat pro nastavení velikosti vyrovnávací paměti při čtení dat z kamery.

Kameru spustíme zavoláním metody CameraManageru openCamera. Předáme ji ID požadované kamery a zejména ukazatel na událost stateCallback, která bude volaná při každé změně stavu kamery.

```
private CameraDevice kamera;

private CameraDevice.StateCallback stateCallback =
            new CameraDevice.StateCallback() {
    @Override
    public void onOpened(CameraDevice camera) {
        // udalost vyvolana pri spusteni kamery
        kamera = camera;
        kameraPreviewStart();
    }

    @Override
    public void onDisconnected(CameraDevice camera) {
        kamera.close();
    }

    @Override
    public void onError(CameraDevice camera, int error) {
        kamera.close();
        kamera = null;
        Toast.makeText(getApplicationContext(),
            String.format("Chyba kamery:%i", error), Toast.LENGTH_LONG).show();
```

```
    }
};
```

Instance stateCallback obsahuje tři předefinované události. Nejdůležitější je onOpened(), vyvolaná při spuštění kamery. V ní si uložíme ukazatel na kameru do proměnné třídy a zavoláme metodu pro zahájení zobrazování preview kameraPreviewStart(). Událost onDisconnected() je vyvolaná, pokud kamera přestane být k dispozici. V takovém případě s ní pouze ukončíme spojení. Událost onError() se vyvolá v případě závažné chyby kamery. Chybu uživateli zobrazíme a rovněž ukončíme spojení s kamerou.

```
private CaptureRequest.Builder captureRequestBuilder;
private CameraCaptureSession cameraSession;

private void kameraPreviewStart() {
    try {
        SurfaceTexture texture = textureView.getSurfaceTexture();
        texture.setDefaultBufferSize(velikostObrazu.getWidth(),
                  velikostObrazu.getHeight());

        Surface surface = new Surface(texture);
        captureRequestBuilder =
                  kamera.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW);
        captureRequestBuilder.addTarget(surface);

        kamera.createCaptureSession(Arrays.asList(surface), new
                  CameraCaptureSession.StateCallback(){
            @Override
            public void onConfigured(@NonNull CameraCaptureSession
                  cameraCaptureSession) {
                // pokud je session pripravena, zahajit preview
                cameraSession = cameraCaptureSession;
                updatePreview();
            }
            @Override
            public void onConfigureFailed(@NonNull CameraCaptureSession
                  cameraCaptureSession) {
                Toast.makeText(getApplicationContext(), "Chyba konfigurace",
                  Toast.LENGTH_SHORT).show();
            }
        }, null);

    } catch (CameraAccessException e) {
        e.printStackTrace();
        Toast.makeText(this, "Chyba: " + e.getMessage(),
                  Toast.LENGTH_LONG).show();
    }
}
```

K zachytávání snímků odesílaných kamerou slouží knihovna SurfaceTexture. Pomocí ní snímky zpracováváme, renderujeme (vykreslujeme), ukládáme do vyrovnávací paměti a postupně předáváme komponentě TextureView, která se stará o jejich zobrazení. Při startu preview tedy nejdřív z textureView získáme ukazatel na třídu SurfaceTexture, ten si uložíme do lokální proměnné texture a jejím prostřednictvím nastavíme velikost vyrovnávací paměti, odpovídající velikosti snímaného obrazu.

Kromě toho, že potřebujeme vyrovnávací paměť na vstupu obrazu, potřebujeme kvůli plynulosti přehrávání vyrovnávací paměť i na TextureView. Tu zajistí třída Surface. Vytvoříme si její instanci, uložíme ji do lokální proměnné surface a předáme jako cíl pro captureRequestBuilder.

API poskytuje pro kameru několik různých předdefinovaných nastavení pro různé typy zpracování obrazu. Tato nastavení se liší pro záznam videa, pořizování fotografických snímků nebo právě pro preview bez záznamu. K nastavení parametrů kamery se používá builder třídy CaptureRequest.

Data z CaptureRequest se odesílají prostřednictvím CameraCaptureSession, která má dvě předefinované události, onConfigured() a onConfigureFailed(). První je vyvolaná v okamžiku úspěšného nakonfigurování, druhá pokud konfigurace selže.

```
private void updatePreview() {
    captureRequestBuilder.set(CaptureRequest.CONTROL_AE_MODE,
                              CaptureRequest.CONTROL_AE_MODE_ON);

    captureRequestBuilder.set(CaptureRequest.CONTROL_AE_TARGET_FPS_RANGE,
                              new Range(15,15));

    try {
        cameraSession.setRepeatingRequest(captureRequestBuilder.build(),
                              null, backgroundHandler);
    } catch (CameraAccessException e){
        e.printStackTrace();
        Toast.makeText(this, "Chyba: " + e.getMessage(),
                              Toast.LENGTH_LONG).show();
    }
}
```

Zatímco jsme v metodě kameraPreviewStart() nastavili kameře blok přednastavených parametrů, v updatePreview() prostřednictvím captureRequestBuilder nastavíme další parametry už jednotlivě, pole konkrétní potřeby. Zde se spokojíme se dvěma základními – zapnutí automatického nastavování expozice a přizpůsobení počtu snímku aktuální expozici. Jsou to parametry, bez jejichž nastavení by výstup z kamery mohl být příliš tmavý. Kamera si se zapnutou automatickou expozicí bude řídit světlost snímku sama, ale pokud jí necháme snímat např. 30 snímků za vteřinu, tak ve špatně osvětleném prostředí bude výstup tmavý, protože na delší expozici nebude mít mezi snímky jednoduše dost času. Pro preview ale nepotřebujeme tolik snímků, takže si můžeme dovolit nechat kameru zredukovat jejich počet až na polovinu. To už by mělo stačit pro dostatečně zřetelný výstup i ve tmavém prostředí. Povolený rozsah se obvykle dopočítává automaticky, zde ho ale můžeme kameře stanovit napevno. Pro účely pokusu to bude stačit.

Nakonec parametry odešleme, pro jistotu v bloku try-catch na kameru. V tomto případě jsme si zvolili dva parametry automatické expozice, ale nastavit jich můžeme velké množství, z nichž některé jsou obecné a jiné závislé na hardwarovém vybavení kamery. Můžeme takto nastavit ostření, přiblížení, optickou stabilizaci, použití blesku, otočení obrazu, redukci šumu a mnoho dalšího.

Tím máme připravené metody a události, které se budou postupně volat tak, že metoda spustitKameru() iniciuje zpětné volání CameraDevice.StateCallback(), to zavolá metodu kameraPreviewStart() a ta z události onConfigured() metodu updatePreview(). Zbývá zavolat onu první metodu, spustitKameru(). Toho docílíme zaregistrováním posluchače SurfaceTextureListener, který bude reagovat na dostupnost SurfaceTexture, změnu velikosti, ukončení a aktualizaci. Stačit bude první z událostí – onSurfaceTextureAvailable(). Jakmile bude SurfaceTexture k dispozici, zavoláme spustitKameru() a zobrazení preview může začít.

```
TextureView.SurfaceTextureListener textureListener =
                        new TextureView.SurfaceTextureListener() {

    @Override
    public void onSurfaceTextureAvailable(SurfaceTexture surface, int width,
                        int height) {
        spustitKameru();
    }

    @Override
    public void onSurfaceTextureSizeChanged(SurfaceTexture surface,
                        int width, int height) {
    }

    @Override
    public boolean onSurfaceTextureDestroyed(SurfaceTexture surface) {
        return false;
    }

    @Override
    public void onSurfaceTextureUpdated(SurfaceTexture surface) {
    }
};
```

Výsledná aplikace by měla vypadat přibližně jako na obrázku 19.10. Jsou-li nastavena potřebná oprávněná, měla by hned po spuštění začít zobrazovat preview snímané hlavní kamerou.

Obrázek 19.10: Preview realizované komponentou TextureView, spuštěné na emulátoru.

20 Připojení k internetu

Jednou z nejsilnějších stránek moderních zařízení je schopnost komunikovat prostřednictvím internetu. Android zařízení jsou většinou technicky vybavena ke komunikaci pomocí WiFi připojení nebo prostřednictvím sítě telekomunikačního operátora. Kromě těchto nejčastějších způsobů je ještě často možné připojit zařízení k internetu pomocí USB kabelu, Bluetooth, nebo jinými alternativními způsoby.

20.1 XML nebo JSON?

V následujících kapitolách se budeme zabývat přenosem dat mezi Android zařízením a internetovými servery. Přitom vznikne potřeba použití vhodného formátu předávaných dat. Android podporuje dva základní formáty, XML (Extensible Markup Language) a JSON (JavaScript Object Notation).

Při rozhodování, který formát použít, často pomůže server, se kterým bude naše aplikace komunikovat. Pokud podporuje pouze XML nebo pouze JSON, je volba jednoduchá. V případě, že podporuje oba formáty, nebo si budeme server sami zakládat, je dobré zohlednit následující rozdíly obou formátů:

- XML zpráva je okem lépe čitelná, struktura je snáz zobrazitelná
- XML struktura je obecně rozšířenější, má výrazně širší použití než jen na předávání dat
- JSON je kratší a tím pádem i rychleji přenesený
- XML se pomaleji čte a zapisuje, dekódování je výrazně náročnější a pomalejší
- JSON nepodporuje přenos binárních dat
- JSON podporuje pole
- JSON nepoužívá koncovou značku

Kromě těchto rozdílů mají oba formáty společné znaky:

- oba se dají číst pouhým okem (i když XML o něco lépe)
- podporují hierarchii (data můžou být obsažena uvnitř dat)
- mají podporu v mnoha programovacích jazycích, takže serverovou část je snadné implementovat

Jednoduchá zpráva ve formátu XML, která obsahuje záhlaví a textový atribut, může vypadat takto:

```
<?xml version="1.0" encoding="UTF-8" ?>
<message>
   <header>
      <info>zahlavi zpravy</info>
   </header>
   <body>
      <txt>text zpravy</txt>
   </body>
</message>
```

Stejná zpráva ve formátu JSON:

```
{"message": {"header": {"info": "zahlavi zpravy"},"body": {"txt": "text
zpravy"}}}
```

Dekódování (parsování) XML

Pro účely dekódování XML se v Androidu používá především knihovna XmlPullParser. Práce s ní je vcelku jednoduchá. Nejprve je potřeba vytvořit instanci parseru:

```
XmlPullParserFactory parserFactory = XmlPullParserFactory.newInstance();
XmlPullParser parser = parserFactory.newPullParser();
```

Potom parseru přiřadíme zdroj, naši XML zprávu v textové proměnné:

```
parser.setInput(xmlData);
```

Od této chvíle můžeme procházet XML dokumentem po jednotlivých značkách pomocí metody next():

```
int eventType = parser.next();
```

Každé její zavolání způsobí přechod na další značku a do proměnné eventType přiřadí jeho typ:

XmlPullParser.START_DOCUMENT – začátek XML dokumentu

XmlPullParser.END_DOCUMENT – konec XML dokumentu

XmlPullParser.START_TAG – začátek atributu

XmlPullParser.END_TAG – koncová značka atributu

XmlPullParser.TEXT – obsah atributu

Na výšeuvedeném příkladu XML tedy bude posloupnost následující:

START_DOCUMENT (začátek XML dokumentu)

START_TAG (značka začátku atributu message)

START_TAG (značka začátku atributu header)

START_TAG (značka začátku atributu info)

END_TAG (značka konce atributu info)

END_TAG (značka konce atributu header)

a tak dále.

V každém atributu navíc můžeme použít metodu getText(), která vrátí jeho obsah:

```
String hodnotaAtributu = parser.getText();
```

Pokud jsme se tedy pomocí next() dostali až k atributu txt, vrátí getText() jeho obsah, tj. text „text zpravy“.

Na základě toho si můžeme napsat jednoduchý parser, který projde XML dokument, najde požadovaný atribut a vrátí jeho hodnotu:

```
private String parserXml (XmlPullParser parser, String atribut) {
    try {
        // dokud není konec dokumentu…
        int typ = parser.getEventType();
        while (typ != XmlPullParser.END_DOCUMENT) {
            // jsme-li na pocatecni znacce hledaneho atributu
            if(typ == XmlPullParser.START_TAG &&
                                        parser.getName().equals(atribut)) {
                // nacist a vratit text atributu
                typ = parser.next();
                if (typ == XmlPullParser.TEXT) {
                    return parser.getText();
                }
            }
            // prejit na dalsi znacku
            typ = parser.next();
        }
    }
    catch (Exception e) {
        Toast.makeText(MainActivity.this, "Chyba při dekódování XML zprávy " +
                                    e.getMessage(), Toast.LENGTH_SHORT).show();
    }
    return "";
}
```

Metoda parserXml() prochází v cyklu jednotlivé značky obsažené v XML zprávě a jakmile narazí na počáteční značku hledaného atributu, načte jeho obsah a ten předá v návratové hodnotě. Pokud atribut nenajde, nebo zjistí, že neobsahuje textovou hodnotu, vrátí prázdný řetězec.

Dekódování (parsování) JSON

Začíná se tak, že si vytvoříme JSONObject:

```
JSONObject jsonObject = new JSONObject(jsonData);
```

Proměnná jsonData obsahuje zprávu v textovém formátu. Od této chvíle je v proměnné jsonObject celá struktura JSON zprávy. Pokud bychom z ní chtěli dostat pouze záhlaví, získali bychom ho takto:

```
String zahlavi = jsonObject.getString("header");
```

A opět z něj můžeme udělat JSON objekt:

```
JSONObject jsonZahlavi = new JSONObject(zahlavi);
```

Následně bychom zjistili hodnotu uloženou v atributu info:

```
String zahlaviInfo = jsonZahlavi.getString("info");
```

Nyní je v proměnné zahlaviInfo uložena hodnota „zahlavi zpravy“. Tímto způsobem se dá dostat k libovolnému prvku v hierarchii JSON zprávy. Kromě metody getString() pro získání řetězcové hodnoty jsou k dispozici také metody getBoolean(), getInt(), getLong(), getDouble() a getJSONArray() pro odpovídající datové typy.

Můžeme si udělat jednoduchý parser, který nebude univerzální a který se bude hodit pouze pro náš typ zprávy. Když si zprávu naformátujeme tak, aby byla vidět hierarchie, zjistíme, že se k atributu txt dostaneme poměrně snadno přes značky „message“ a následně „body“:

```
{
  "message":
  {
    "header":
    {
      "info": "zahlavi zprávy"
    },
    "body":
    {
      "txt": "text zprávy"
    }
  }
}
```

Parser, který má ve vstupním parametru text JSON zprávy a na výstupu hodnotu atributu „txt“, může vypadat takto:

```
// vrati obsah atributu nalezeneho v JSON zprave
private String parserJson (String jsonZprava) {
    try {
        // vytvorit instanci JSONObject
        JSONObject jObj = new JSONObject(jsonZprava);

        // projit hierarchicky pres message > body
        jObj = new JSONObject(jObj.getString("message"));
        jObj = new JSONObject(jObj.getString("body"));

        // vratit hodnotu atributu txt
        return jObj.getString("txt");
    }
    catch (Exception e) {
        Toast.makeText(MainActivity.this, "Chyba při dekódování JSON zprávy " +
                                    e.getMessage(), Toast.LENGTH_SHORT).show();
    }

    return "";
}
```

Takový parser nepůjde použít na nic jiného než na typ zprávy, který bude mít strukturu jako zpráva v našem příkladu. Ale to nemusí vadit, pro většinu zpráv ze serveru je struktura pevně daná a s jednoduchým parserem se dá vystačit.

20.2 Připojení

Ať už se zařízení připojuje jakkoliv, Android umožňuje komunikaci se servery pomocí HTTP nebo HTTPS protokolu. Dva základní způsoby komunikace jsou tyto:

- Web Sockets – pro komunikaci s okamžitou odezvou

- HttpClient – pro asynchronní komunikaci

Web Sockets používají hry, aplikace pro chatování nebo datové hovory. Aplikace vytvoří komunikační kanál, takzvaný TCP stream, který jí umožní přenášet data v reálném čase. Pro realizaci se používají knihovny třetích stran. Nejznámější je zřejmě open source knihovna jWebSocket.

HttpClient se používá tam, kde můžeme odeslat požadavek a na odpověď čekáme. Například při načítání dat do formulářů nebo přenos jakýchkoliv externích dat. Je to takzvaná REST (Representational State Transfer) komunikace, u které je předávání dat řešené odesíláním požadavků (request) a vracením odpovědí (response). Používají se tyto metody komunikace:

- GET – vhodná metoda pro získávání dat ze serveru

- POST – pro odesílání dat na server

- OPTIONS – vrátí seznam metod, které server podporuje

- HEAD – podobně jako GET, vrací ale pouze hlavičku zprávy

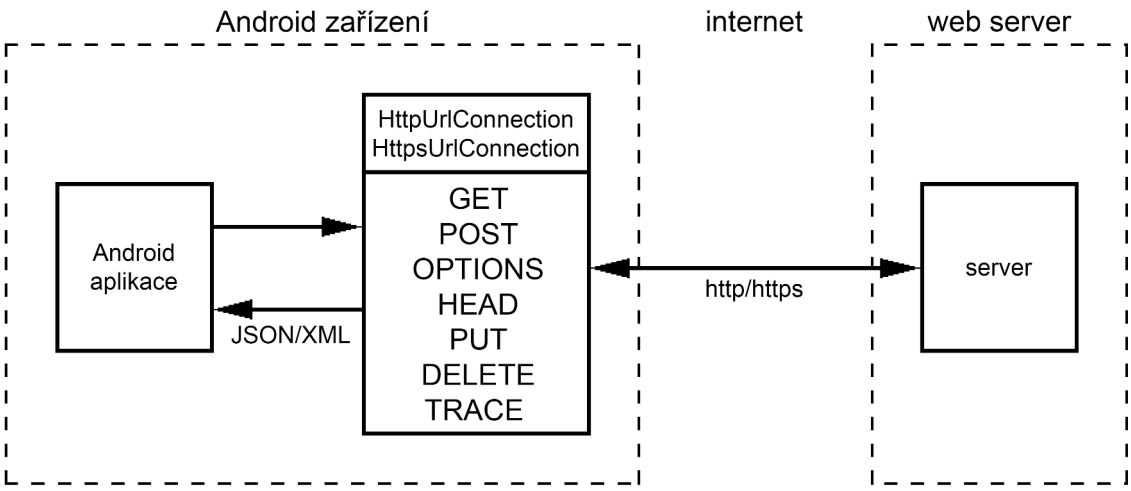
- PUT – pro aktualizaci dat

- DELETE – pro mazání dat

- TRACE – pro ladění a testování

V případě HttpClient komunikace lze využít buď knihovny třetích stran, nebo jednu ze dvou standardních Android knihoven, HttpsUrlConnection či HttpUrlConnection. Obě poskytují stejnou funkcionalitu, první na zabezpečeném, druhá na nezabezpečeném spojení. Tyto knihovny podporují komunikaci všemi uvedenými metodami (metoda se nastaví parametrem setRequestMethod), bez nutnosti cokoliv dalšího do Android Studia doinstalovávat.

Obecně se doporučuje používat pouze zabezpečené spojení. Starší verze Androidu sice nezabezpečené spojení neomezovaly, ale od API verze 28 (Android 9, Pie) je možné volně použít pouze SSL/TLS připojení. Přestože existuje možnost pomocí výjimky v manifestu povolit připojení na nezabezpečený server, tak je lepší tuto možnost nevyužívat. A to nejen z bezpečnostních důvodů. Schvalování aplikací na Google Play, využívajících nezabezpečené spojení, trvá déle a v budoucnu se může stát, že Google tuto možnost zakáže úplně. Nic ale nebrání v použití http připojení pro účely testování.



Obrázek 20.1: Komunikace Android zařízení se serverem pomocí http nebo https protokolu.

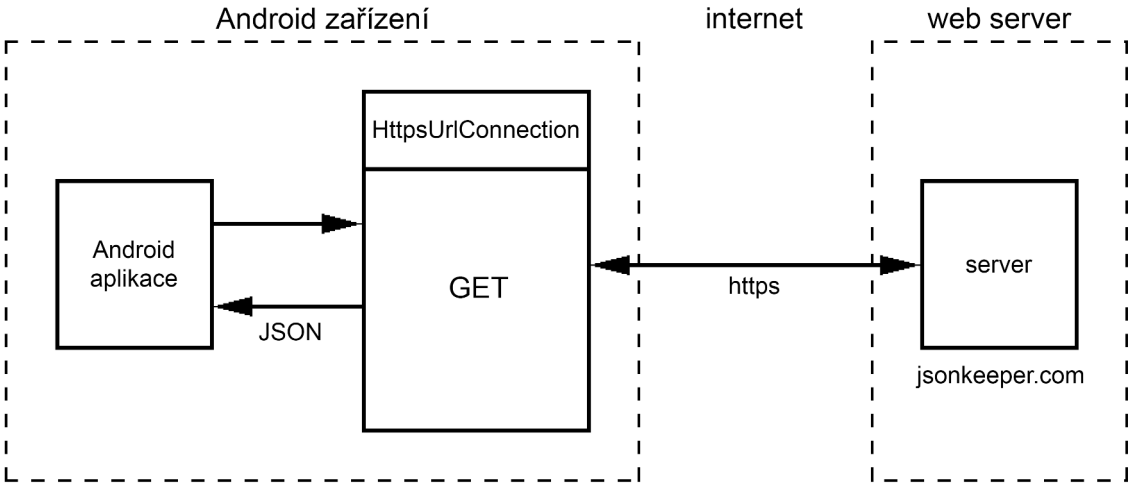
Pro předávání dat se používají dva již známé formáty. XML, který je strukturovaný a lze ho snadno číst, nebo JSON, který sice není tak snadno čitelný, ale je výrazně úspornější z hlediska objemu předávaných dat.

20.3 Připojení k internetu, 1. příklad

Uděláme si jednoduchý příklad načtení dat ze serveru do Android aplikace pomocí REST komunikace. Použijeme standardní knihovnu HttpsUrlConnection, o data požádáme metodou GET a necháme si je předat ve formátu JSON.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 20_Pripojeni_k_internetu
projekt Internet1

Schematicky znázorněné by to mělo být takto:



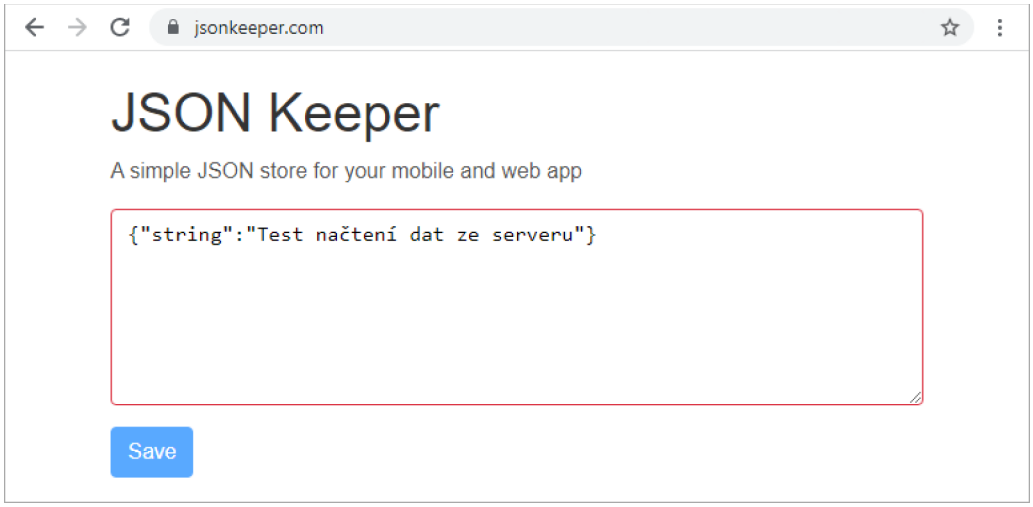
Obrázek 20.2: Předání dat ve formátu JSON mezi Android zařízením a serverem, pomocí https protokolu a metody GET.

Co je potřeba udělat?

- zajistit server, který bude poskytovat data
- navrhnout jednoduchý formulář aplikace
- povolit aplikaci přístup k datovým přenosům
- napsat metodu která odešle metodou GET požadavek na server
- zobrazit přijatou zprávu

Abychom nemuseli internetový server programovat, použijme pro pokusy některou ze zdarma dostupných stránek, které poskytují prostor pro uložení malých statických dat. Takových služeb je k dispozici celá řada, jsonkeeper.com, jsonserve.com, mocki.io, extendsclass.com a mnoho dalších. Většinou nejsou vhodné pro větší projekty, ale na jednoduché pokusy jsou plně dostačující. Každá z těchto služeb umožní nadefinovat si, jakou odpověď od serveru očekáváme a zpřístupní API pomocí kterého si lze o data požádat. Vezměme třeba první z nich, jsonkeeper.com. Rozhraní serveru není nijak komplikované, v prohlížeči otevřeme stránku www.jsonkeeper.com a hned na titulní straně zadáme očekávanou odpověď. Stačí, aby byla úplně jednoduchá, například takto:

```
{"string":"Test načtení dat ze serveru"}
```



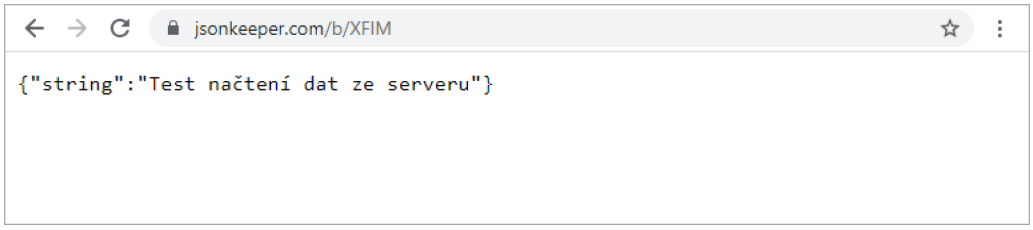
Obrázek 20.3: Online služba jsonkeeper.com

Je to zpráva ve formátu JSON o jediném textovém atributu.

Po uložení server vygeneruje a sdělí URI adresu (takzvaný endpoint), na kterém očekává žádost, neboli request. Adresa vypadá nějak takto:

https://www.jsonkeeper.com/b/XFIM

Tuto adresu můžeme zadat do prohlížeče a zkontrolovat, že na ní server skutečně vrací očekávanou JSON zprávu.



Obrázek 20.4: Připojení se internetovým prohlížečem k api rozhraní serveru jsonkeeper.com

Stejnou zprávu načteme i do Android aplikace. Zatím ji nebudeme nijak upravovat, pouze ji zobrazíme tak, jak ji server předá. Zde je potřeba říct, že standardní server tyto zprávy generuje dynamicky a v závislosti na požadavku vrací různá data, jak z hlediska obsahu tak i velikosti odpovědi. Server, který jsme použili slouží pouze pro účely otestování technologie a pochopitelně vrací zprávu stále stejnou – tak, jak jsme si ji zadali.

Založíme projekt v Android Studiu. Formulář navrhneme jednoduchý, bude stačit tlačítko pro odesílání požadavku a textové pole pro zobrazení výsledku. Můžeme přidat ještě další tlačítko, pro mazání obsahu textového pole.

Abychom mohli komunikovat prostřednictvím datových přenosů, musíme rozšířit manifest o příslušné oprávnění:

```
<uses-permission android:name="android.permission.INTERNET" />
```

V rámci tohoto příkladu si ještě ukážeme, jak ověřit, jestli jsou data zapnutá. K tomu budeme potřebovat další oprávnění:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Jelikož je odesílání a příjem dat pomocí knihovny HttpUrlConnection asynchronní, hodí se pro komunikaci se serverem založit vlastní třídu. Nazveme ji třeba GetData.java. V ní založíme metodu startAsync(), ve které využijeme knihovnu ExecutorService. Ta umí jednoduchým způsobem, předefinováním run(), definovat a spustit asynchronní operaci a po jejím skončení zpracovat výsledek.

Třída GetData bude vypadat takto:

```
public class GetData {
    String nactenaData = "";

    // operace, která bude spuštěna na pozadí
    protected Void startAsync() {

        ExecutorService executor = Executors.newSingleThreadExecutor();
        Handler handler = new Handler(Looper.getMainLooper());

        executor.execute(new Runnable() {
            @Override
            public void run() {

                try
                {
                    URL url= new URL("https://www.jsonkeeper.com/b/XFIM");
                    HttpsURLConnection conn = (HttpsURLConnection)
```

```
                                    url.openConnection();
                    InputStream stream = conn.getInputStream();
                    BufferedReader buff = new BufferedReader(
                                    new InputStreamReader(stream));
                    String radka = "";
                    do {
                        nactenaData = nactenaData + radka;
                        radka = buff.readLine();
                    } while (radka != null);
                }
                catch(Exception e) {
                    nactenaData = "Chyba: " + e.getMessage();
                }
                handler.post(new Runnable() {
                    // udalost volaná po skončení operace
                    @Override
                    public void run() {
                        MainActivity.jsonTextView.setText(nactenaData);
                    }
                });
            }
        });
    }
}
```

Metoda startAsync() nejdřív pošle přes HttpsURLConnection požadavek na server jsonkeeper.com. Že jde o metodu GET není nutné knihovně říkat, pokud nemá určenou jinou metodu, odesílá data právě pomocí GET. Parametry nejsou potřeba, co má server vrátit, ví podle kódu 19errd, který služba vygenerovala, když jsme nadefinovali požadovanou odpověď serveru. Knihovna odešle požadavek https protokolem na server a ten stejnou cestou vrátí odpověď. Tu nejprve načteme do bufferu a z něj se potom v cyklu naplní proměnná nactenaData.

Odesílání a příjem musí být, vzhledem k tomu, že jde o vstupně/výstupní operaci, v bloku try. Pokud dojde k chybě, nakopírujeme ji do proměnné nactenaData namísto zprávy. Tím bude zajištěno, že se následně zobrazí na displeji a bude vidět, že se něco nepodařilo.

Po skončení zpracování metody startAsync() je vyvolaná metoda run() v bloku handler.post, která už jenom předá obsah proměnné nactenaData do textového pole ve formuláři.

Ve třídě MainActivity je potřeba zařídit spuštění asynchronní metody startAsync() po stisknutí tlačítka. Než to ale uděláme, připravíme si v MainActivity ještě jednu pomocnou metodu. Bude se jmenovat datovePripojeni() a bude zjišťovat, jestli jsou na Android zařízení zapnuté datové přenosy:

```
// Overi dostupnost datoveho pripojeni
private boolean datovePripojeni() {
    Context context = getApplicationContext();
    ConnectivityManager cm = (ConnectivityManager)
                    context.getSystemService(Context.CONNECTIVITY_SERVICE);
    if (cm == null)
        return false;

    // API verze 21 a vyssi
    if (android.os.Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        NetworkCapabilities capabilities =
                    cm.getNetworkCapabilities(cm.getActiveNetwork());
        if (capabilities != null &&
      (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR) ||
```

```
      capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) ||
      capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI_AWARE))
        )
        return true;
    }
    // API verze nizsi nez 21
    else {
        NetworkInfo netInfo = cm.getActiveNetworkInfo();
        return (netInfo != null && netInfo.isConnected());
    }
    return false;
}
```

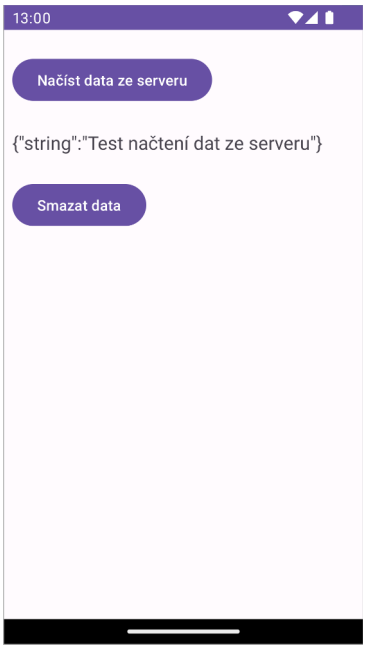
Metoda pomocí ConnectivityManageru otestuje stav datového připojení a podle výsledku vrátí buď true nebo false. Způsob zjištění se liší podle verze API. Do 21 bylo možné zjistit pouze, jestli je připojení aktivní. Od verze 21 je možné rozlišit také jednotlivé typy připojení. Metoda ověří pouze mobilní a WiFi připojení, k dispozici je ale ještě mnoho dalších.

Přidáme metodu, která bude volaná stisknutím tlačítka:

```
public void nacistDataZeServeru(View view) {
    if (datovePripojeni() == true) {
        GetData getData = new GetData();
        getData.startAsync();
    }
}
```

Metoda nacistDataZeServeru() pouze zavolá metodu pro ověření datového připojení a pokud je vše v pořádku, tak spustí asynchronní proces ve třídě GetData.

Výsledná aplikace by, po načtení dat ze serveru, měla vypadat přibližně jako na obrázku 20.5.



Obrázek 20.5: Aplikace, zobrazující data z web serveru, spuštěná na emulátoru.

20.4 Připojení k internetu, 2. příklad

Předchozí příklad nyní rozšíříme tak, aby získal praktičtější uplatnění. Na internetu jsou dostupné služby, které poskytují informace o počasí. A některé z nich zpřístupňují i API, takže je možné data získat pomocí HTTP volání z aplikace. Toho využijeme a napíšeme aplikaci, která se připojí k serveru, získá z něj základní data o počasí, ta dekóduje a zobrazí na displeji.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 20_Pripojeni_k_internetu
projekt Internet2

Co je potřeba udělat?

- najít vhodnou službu, která přes API poskytuje data o počasí

- sestavit řetězec pro GET metodu, tak aby získal základní data pro konkrétní místo a v požadovaném formátu (XML nebo JSON)

- stejně jako v předchozím příkladu, načíst data pomocí volání GET

- načtená data dekódovat a získat z nich informaci o počasí

- informaci o počasí zobrazit na displeji

Služeb, které dávají bezplatně k dispozici základní informace o počasí prostřednictvím API je několik. Většinou se liší v licenčních podmínkách, způsobu zadání vstupních údajů nebo ověření uživatele, který o data žádá.

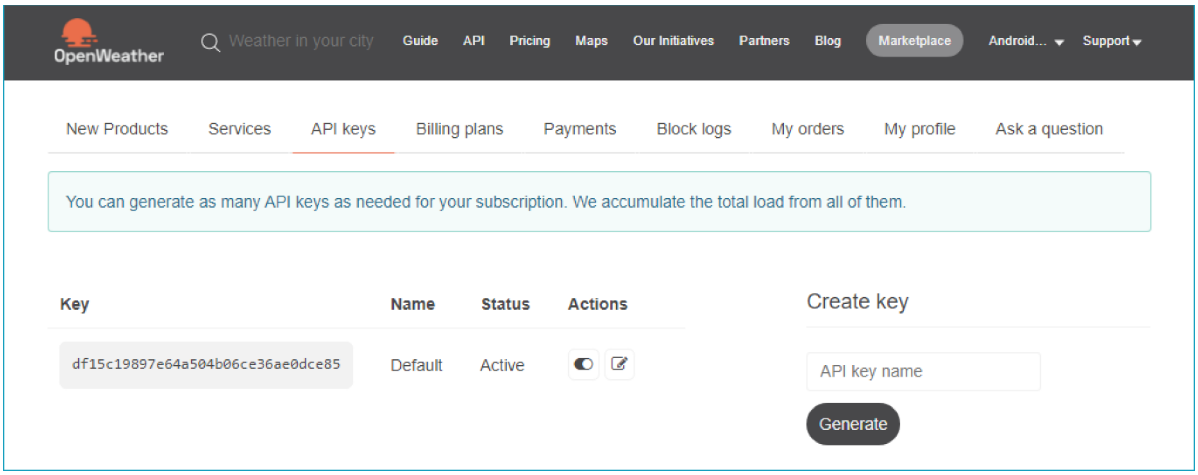
openweathermap.org – podrobná data o počasí, v bezplatné verzi omezení na 60 volání za minutu

weatherapi.com – velké množství volání zdarma, podmínkou je zpětný odkaz na službu v aplikaci a a bezplatné obnovování registrace každý měsíc

Weatherbit.io – v bezplatné verzi omezené na tisíc volání denně

Existuje mnoho dalších služeb a většinou je nutné volit podle omezení která mají a které vývojáři buď budou nebo nebudou vyhovovat. U většiny služeb je nutná registrace, k dispozici bývají jak bezplatné tak placené verze, obvykle je nutné do volání vkládat vygenerovaný API klíč. Pro náš pokus zvolíme openweathermap.org.

První, co je nutné udělat, je získat klíč pomocí kterého se budeme při dotazování na počasí identifikovat. Na stránce api.openweathermap.org klikneme na odkaz Sign Up. Projdeme si krátkou registrací a to je v podstatě všechno co po nás provozovatelé serveru chtějí. Klíč vygenerují automaticky, stačí si ho vyzvednout pod záložkou API keys:



Obrázek 20.6: Stránka OpenWeatherMap s vygenerovaným klíčem.

Tento klíč budeme přidávat jako parametr při každé žádosti o data. Použijeme opět volání API pomocí metody GET. Parametry tedy budeme předávat pomocí URL. Adresa pro přístup k API je http://api.openweathermap.org/data/2.5/weather

Vše ostatní zadáváme pomocí zmíněných parametrů. Přehled základních parametrů, které se při sestavování dotazu mohou hodit:

q – místo. Stačí zadat název města, server se ho pokusí identifikovat. Tento parametr je povinnný

mode – formát odpovědi. Pokud parametr není uveden, je zpráva ve formátu JSON, další možné hodnoty jsou xml a html

units – použité jednotky, zejména pro teplotu. Neuvedený parametr znamená, že teplota bude ve stupních Kelvina. Hodnota metric nastaví teplotu ve stupních Celsia, imperial ve stupních Fahrenheita.

lang – jazyk textových popisů. Hodnoty jsou nabíledni – cz pro češtinu, sk pro slovenštinu, en (nebo neuvedený parametr) pro angličtinu.

appid – API klíč. Povinný parametr

Další parametry jsou popsané v online dokumentaci.

Základní dotaz má formu URL a vypadá takto:

```
api.openweathermap.org/data/2.5/weather?q=brno
&appid=df15c19897e64a504b06ce36ae0dce85
```

Můžeme ho dát do prohlížeče a pokochat se výsledkem který zobrazí aktuální počasí v Brně, ve formátu JSON:

```
{"coord":{"lon":16.61,"lat":49.19},"weather":[{"id":800,"main":"Clear",
"description":"clear sky","icon":"01n"}],"base":"stations","main":
{"temp":286.72,"pressure":1003,"humidity":82,"temp_min":284.26,"temp_max":289.26
},"visibility":10000,"clouds":{"all":0},"dt":1558304094,"sys":{"type":1,
"id":6851,"message":0.0066,"country":"CZ","id":3078610,"name":"Brno","cod":200}
```

A rovnou dotaz pomocí dalších parametrů trochu vylepšíme:

```
http://api.openweathermap.org/data/2.5/weather?q=brno&units=metric&mode=xml
&lang=cz&appid=df15c19897e64a504b06ce36ae0dce85
```

Takto získáme opět informaci o brněnském počasí, ale zobrazenou ve výrazně čitelnějším formátu XML, s popisem v češtině a teplotou ve stupních Celsia. Dozvíme se z ní řadu informací, mimo jiné, že v Brně je právě jasno, 16,6 stupňů Celsia:

```
<current>
      <city id="3078610" name="Brno">
            <coord lon="16.6" lat="49.19"/>
            <country>CZ</country>
      </city>
      <temperature value="13.17" min="10" max="16.11" unit="celsius"/>
      <humidity value="82" unit="%"/>
      <pressure value="1003" unit="hPa"/>
      <clouds value="0" name="jasno"/>
      <visibility value="10000"/>
      <precipitation mode="no"/>
      <weather number="800" value="jasno" icon="01n"/>
</current>
```

Pro účely pokusné aplikace zvolíme kompromis. Adresa bude obsahovat všechny uvedené parametry kromě mode. Získáme tedy stejné informace, ale ve formátu JSON.

http://api.openweathermap.org/data/2.5/weather?q=brno&units=metric&lang=cz
&appid=df15c19897e64a504b06ce36ae0dce85

Endpoint, neboli adresa s parametry, kterou voláme, nevypadá ani příliš hezky ani není dobře čitelný. To nás ale nemusí nijak zvlášť zajímat. Bude stále stejný, v aplikaci budeme pouze nahrazovat název města, který se nachází zhruba uprostřed.

Vytvoříme si novou aplikaci, nebo upravíme předchozí tak, abychom na formuláři měli vstupní textové pole pro zadání města, tlačítko pro načtení počasí a textové pole které počasí zobrazí.

Nesmíme zapomenout na manifest, který stejně jako minule rozšíříme o oprávnění pro přístup k internetu a případně i ke kontrole stavu připojení:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Třídu MainActivity.java upravíme tak, aby poskytovala přístup k polím pro zadání města a zobrazení výsledku. Metodu nacistDataZeServeru() rozšíříme o smazání pole s informací o počasí, abychom ho nemuseli mazat samostatným tlačítkem jako v předchozím příkladu.

```
public class MainActivity extends AppCompatActivity {
    public static TextView txtMesto;
    public static TextView txtPredpoved;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        // pole pro zadani mesta a informaci o pocasi priradit do promennych
        txtMesto = (TextView) findViewById(R.id.txtMesto);
        txtPredpoved = (TextView) findViewById(R.id.txtPredpoved);
    }

    // nacte data ze serveru a zobrazi je v textovem poli
    public void nacistDataZeServeru(View view) {
        // smazat predchozi informaci o počasí
        txtPredpoved.setText("");
        // spustit nacitani dat o počasí
        if (datovePripojeni() == true) {
            GetData getData = new GetData();
            getData.startAsync();
        }
    }
}
```

Do třídy GetData přidáme konstruktor, abychom mohli předat název města a upravíme blok try – catch v metodě pro spuštění na pozadí, startAsync():

```
String mesto = "Karlovy Vary";
String data = "";
Context context;

// konstruktor tridy
public GetData (Context cont, String nazevMesta) {
    context=cont;
    mesto = nazevMesta.replace(" ", "%20"); // nahradit pripadne mezery
}

protected Void startAsync() {

    ...

    try
    {
        URL url= new URL("https://api.openweathermap.org/data/2.5/weather?q="
        + mesto
        + "&units=metric&lang=cz&appid=df15c19897e64a504b06ce36ae0dce85");

        HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
        conn.setRequestMethod("GET");    // pro predani dat pouzita metoda GET
        conn.setReadTimeout(10000);      // timeout v milisekundach
        conn.setConnectTimeout(15000);   // timeout v milisekundach
        conn.setDoOutput(false);         // priznak - telo zpravy neodesilat

        InputStream stream = conn.getInputStream();
        BufferedReader buff = new BufferedReader(new
                                    InputStreamReader(stream));

        // nacteni dat po jednotlivych radcich
        String radka = "";
        while (radka != null)
        {
            radka = buff. ReadLine();
            data = data + radka;
        }
    }
```

```
    catch(Exception e) {
        MainActivity.txtPredpoved.setText(e.getMessage());
    }

    ...

}
```

Do proměnné mesto předáme hodnotu ze vstupního textového pole formuláře. Ale jelikož text názvu města budeme vkládat do URL parametru, je v něm potřeba nahradit mezery. Potom obsah proměnné vložíme do vygenerovaného endpointu a ten si dáme do proměnné url. Připojení se na server, načtení dat a jejich uložení do proměnné data bude stejné jako v předchozím příkladu. Pouze si pro názornost přidáme k připojení conn další parametry:

setRequestMethod("GET") – tímto parametrem řekneme že chceme použít metodu GET. Toto není nutné uvádět, ale pokud bychom odesílali data jiným způsobem, museli bychom ho v tomto parametru uvést.

setReadTimeout(10000) – nastavení maximální doby pro zpracování požadavku. Můžeme tím ovlivnit, jak dlouho dobu budeme čekat než server požadavek zpracuje a vrátí odpověď. Pokud tuto dobu překročí, nebudeme na odpověď čekat a zobrazíme chybové hlášení. Uplatní se u serverů, které například kvůli přetížení odpovídají až po delší době nebo neodpoví vůbec.

setConnectTimeout(15000) – nastavení maximální doby pro čekání na připojení k serveru. Pokud se během uvedené doby nepodaří k serveru připojit, zobrazíme chybové hlášení.

setDoOutput(false) – příznak že posíláme pouze hlavičku zprávy. U metody GET zasíláme vždy jen hlavičku, u POST posíláme i zprávu samotnou.

Do metody run() v bloku handler.post, která se automaticky spustí po načtení dat, přidáme stejně jako v minulém příkladu instrukci pro jejich zobrazení v textovém poli na displeji:

```
@Override
public void run() {
    MainActivity.txtPredpoved.setText(data);
}
```

Toto poslouží pouze pro ověření, že vše funguje. Zobrazená data jsou ve formátu JSON a nevypadají moc hezky. Pokud se ale při zkušebním spuštění zobrazila, znamená to, že vše funguje, jak má. Data bude nutné rozparsovat, neboli dekódovat, abychom si z nich mohli vybrat pouze hodnoty, které nás zajímají, ty si zformátujeme a následně zobrazíme. Vytvoříme si k tomu pomocnou metodu dekodovatJson(), jejímž vstupním parametrem bude přijatá zpráva ve struktuře JSON a výstupem text s informací o počasí, získaný ze zprávy.

Struktura načtených dat je, po naformátování, následující:

```
{
  "coord":
  {
    "lon":16.61,"lat":49.19
  },
  "weather":
  [{
    "id":800,"main":"Clear",
    "description":"jasno",
    "icon":"01n"
```

```
  }],
  "base":"stations",
  "main":
  {
    "temp":13.45,
    "pressure":1014,
    "humidity":67,
    "temp_min":10,
    "temp_max":17
  },
  "visibility":10000,
  "wind":
  {
    "speed":2.1,
    "deg":320
  },
  "clouds":
  {
    "all":0
  },
  "dt":1558821490,
  "sys":
  {
    "type":1,
    "id":6851,
    "message":0.0065,
    "country":"CZ",
    "sunrise":1558753175,
    "sunset":1558809699
  },
  "timezone":7200,
  "id":3078610,
  "name":"Brno",
  "cod":200
}
```

Ve zprávě je zakódovaných mnoho informací, ale naštěstí v relativně jednoduché struktuře. Nás budou zajímat pouze dva údaje, teplota a textový popis počasí v češtině. V případě teploty se k hodnotě dostaneme relativně jednoduše. Načteme si vnoření „main“ a z toho vyseparujeme hodnotu parametru „temp“. O něco složitější je textový popis. Opět načteme první vnoření, tj. „weather“, ale tím namísto atributu získáme pole. To je označené hranatými závorkami. Pravděpodobně se počítá s tím, že se zde data můžou vyskytnout vícekrát. Jelikož jde pouze o základní funkčnost, tak se spokojíme s tím, že vezmeme jednoduše první prvek pole. A ten už obsahuje parametr „description“, ze kterého si načteme potřebnou hodnotu.

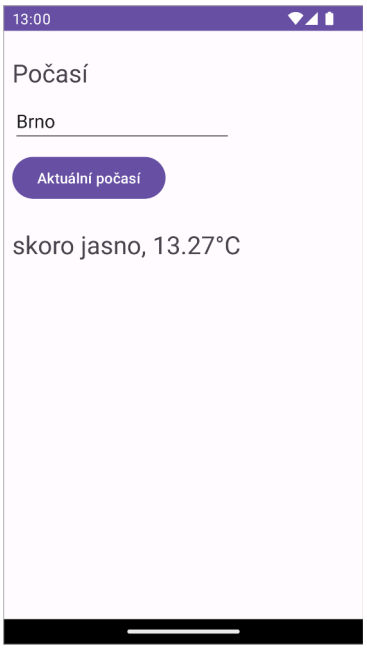
```
// rozparsovani dat z formatu JSON
private String dekodovatJson(String data) {
    try {
        JSONObject jObj = new JSONObject(data);
        jObj = new JSONObject(jObj.getString("main"));
        String teplota = jObj.getString("temp");

        jObj = new JSONObject(data);
        JSONArray jArray = new JSONArray(jObj.getString("weather"));
        jObj = new JSONObject(jArray.getString(0));
        String text = jObj.getString("description");

        return String.format("%s, %s°C", text, teplota);
```

```
    } catch (JSONException e) {
        return e.getMessage();
    }
}
```

Nakonec ještě upravíme metodu run() v bloku handler.post tak, aby zavolala pomocnou metodu dekodovatJson():

```
@Override
public void run() {
    String dekodovanaData = dekodovatJson (data);
    MainActivity.txtPredpoved.setText(dekodovanaData);
}
```

Výsledná aplikace by, po spuštění a načtení dat ze serveru, měla vypadat přibližně jako na obrázku 20.7.



Obrázek 20.7: Aplikace načítající data o počasí, spuštěná na emulátoru

21 Mapy

Díky API Google Maps lze do Android aplikací přidávat mapové podklady, kreslit do nich značky, čáry, polygony a vkládat obrázky. Samotná práce s mapou není příliš složitá, vyžaduje ale menší přípravu, spojenou se zabezpečením komunikace vůči serverům Google.

Pro zobrazení map je možné použít základní komponentu MapView. Práce s ní je ale poměrně těžkopádná. Musí se totiž předefinovat několik událostí životního cyklu komponenty, z nichž většinu nakonec stejně nejspíš vůbec nevyužijeme.

Existují ale další dvě komponenty, které práci zjednodušují. Tvoří samostatný fragment, v němž je zabalené vše, co je potřeba pro zobrazení mapy. Jedna se jmenuje MapFragment a druhá SupportMapFragment. Obě umožňují velmi jednoduché zobrazení mapy, SupportMapFragment má navíc tu vlastnost, že funguje i na starších verzích Androidu, které fragmenty nepodporovaly nativně. Na rozdíl od MapView se ale obě komponenty musí vložit do designeru v podobě jednoduchého XML kódu, protože paleta komponent pamatuje pouze na MapView.

Zjednodušený kód pro vložení komponenty MapFragment do designeru formuláře:

```
<fragment class="com.google.android.gms.maps.MapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Zjednodušený kód pro vložení komponenty SupportMapFragment do designeru formuláře:

```
<fragment class="com.google.android.gms.maps.SupportMapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Kdy tedy kterou komponentu použijeme:

- MapView, pokud chceme mít maximální kontrolu nad mapou a nevadí nám více kódu. Pokud potřebujeme dát mapovou komponentu do fragmentu, je toto správná volba. Následující komponenty by způsobovaly zbytečné vnořování fragmentů.

- MapFragment, pokud upřednostníme jednoduchou práci a budeme podporovat pouze novější verze Androidu (od API verze 12). Toto je asi nejčastější volba. Zejména pokud vkládáme mapu přímo do aktivity a tím pádem nevadí, že tvoří samostatný fragment.

- SupportMapFragment, pokud upřednostníme jednoduchou práci a budeme podporovat i staré verze Androidu (do API verze 11). Ostatní vlastnosti jsou shodné s vlastnostmi komponenty MapFragment.

Při volbě správné komponenty je vhodné také zohlednit rizika plynoucí z příštích verzí těchto elementů. Zatímco budoucí MapView nemusí být kompatibilní, ale může přinášet další možnosti, u MapFragment a SupportMapFragment je díky systémovému zabalení do fragmentu minimální pravděpodobnost, že by se rozhraní změnilo - upgrade za nás tedy udělají autoři komponent, nám bude všechno fungovat jako doposud.

21.1 Mapy – registrace

Servery Google Maps komunikují výhradně zabezpečeným způsobem a pouze s aplikacemi, které jsou autorizované. V praxi to znamená, že vývojář musí mít u Google účet, stačí obyčejný na

Gmail nebo Google Drive. Pomocí tohoto účtu si vygeneruje klíč, který vloží do aplikace a zároveň v registraci Google uvede klíč lokálního certifikátu. Díky těmto dvěma klíčům může vzniknout zabezpečené spojení, kdy budou mít obě strany jistotu v tom, s kým komunikují.

Je to tak proto, že některé účty jsou limitované, například co se týká počtu zobrazení. A Google si chce být jistý, že jeho služby nejsou zneužívány někým jiným. Samozřejmě mu to umožňuje nabízet prémiové služby nebo případnou nadměrnou zátěž zpoplatnit. To se map momentálně netýká, k dispozici jsou volně, nicméně nutnost zabezpečeného spojení zůstává.

Ke zlepšení zabezpečení přispívá, svými možnostmi, i sám Google. Využívá k tomu vlastní vyhledávač a pokud někde na internetu objeví takto vygenerovaný mapový klíč, považuje ho za kompromitovaný a vývojáře upozorní e-mailem.

Co je tedy potřeba pro úspěšnou komunikaci s API Google map udělat?

- zjistit otisk lokálního SHA certifikátu

- zjistit namespace aplikace

- zaregistrovat namespace a klíč na Google a získat tak klíč k mapám

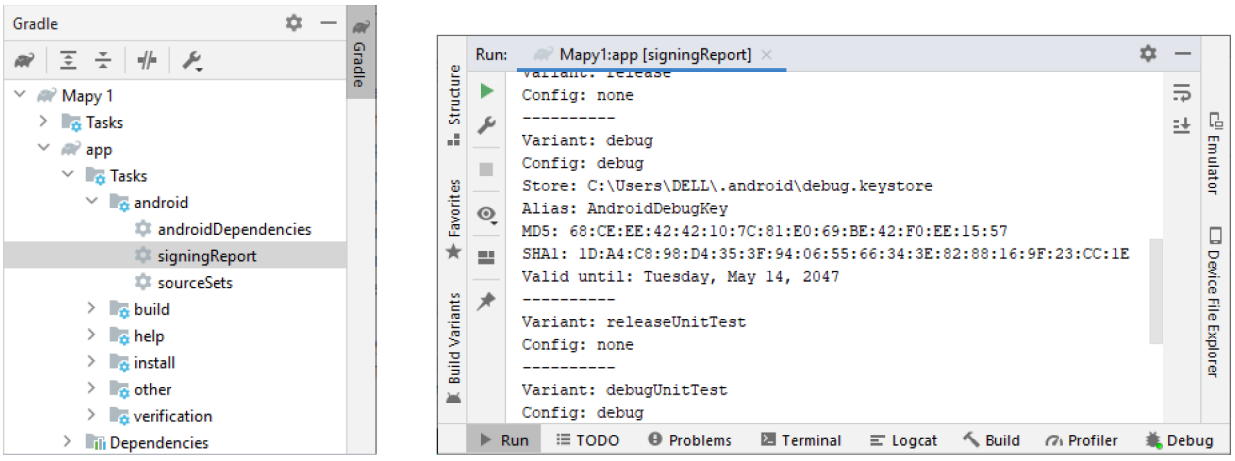
Zní to komplikovaně? Není se čeho bát, složitější je to jenom napoprvé, jakmile budeme jednou mít klíč k mapám, nebude nutné postup znovu opakovat, pro všechny další aplikace už stačí pouze upravit konfiguraci.

Otisk certifikátu je sekvence dvaceti hexadecimálních čísel, unikátní pro každou instalaci Javy. Jak tedy zjistíme ten zmiňovaný otisk lokálního SHA certifikátu? Existuje několik způsobů. Tím snazším je založení nového projektu v Android Studiu a v průvodci zvolit na stránce „Add an activity“ volbu Google Maps Activity. Při generování aktivity Android Studio zjistí náš otisk certifikátu a přidá ho do komentáře v souboru app/res/values/google_maps_api.xml. Po vygenerování projektu si odtud stačí otisk vykopírovat a projekt můžeme zase smazat.

Obrázek 21.1: Otisk lokálního SHA certifikátu získaný z vygenerovaného souboru google_maps_api.xml

Druhá možnost je otevřít v Android Studiu na pravé straně záložku Gradle. V ní najít položku signingReport, dvojklikem ji spustit a v Run okně se zobrazí informace o všech certifikátech v úložišti. Najdeme SHA1 a otisk si zkopírujeme. Pozor, vypsalo se víc konfigurací, zajímá nás „Variant: debug“.

Může se stát, že ve stromové struktuře, v záložce Gradle, položka signingReport nebude k nalezení. V takovém případě by mělo stačit kliknout na ikonku Execute Gradle Task (první ikonka v řadě, v záložce Gradle), otevře se okno Run Anything a v něm lze jednoduše spustit příkaz „gradle signingReport“. Výsledek by měl být stejný, jako při spuštění dvojklikem.

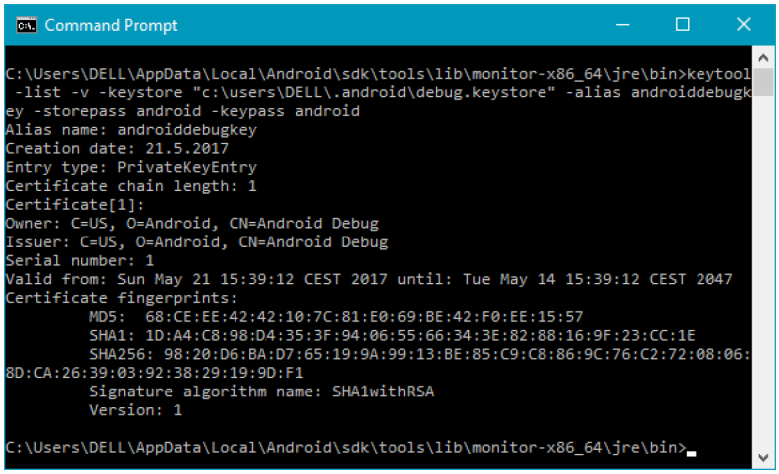


Obrázek 21.2: Otisk lokálního SHA certifikátu získaný z Gradle

Kdo by se chtěl dostat otiskům více pod kůži, může si ho zkusit zjistit pomocí nástroje keytool, který je součástí SDK Javy. Nejdřív je nutné najít aplikaci keytool.exe. Bude pravděpodobně někde v adresáři \AppData\Local\Android\sdk\tools\lib\monitor-x86_64\jre\bin\ Pomoci by mohla funkce hledání souborů ve Windows. Jakmile keytool.exe máme, spustíme ho v command okně s následujícími parametry:

```
keytool -list -v -keystore "c:\users\uzivatelske_jmeno\.android\debug.keystore"
-alias androiddebugkey -storepass android -keypass android
```

Text *uzivatelske_jmeno* nahradíme reálným uživatelským jménem ve Windows. Soubor debug.keystore je úložiště certifikátů. Můžeme se ho také pokusit najít ručně a cestu podle potřeby upravit. Každopádně po úspěšném spuštění získáme obsah úložiště, včetně otisku certifikátu SHA1:

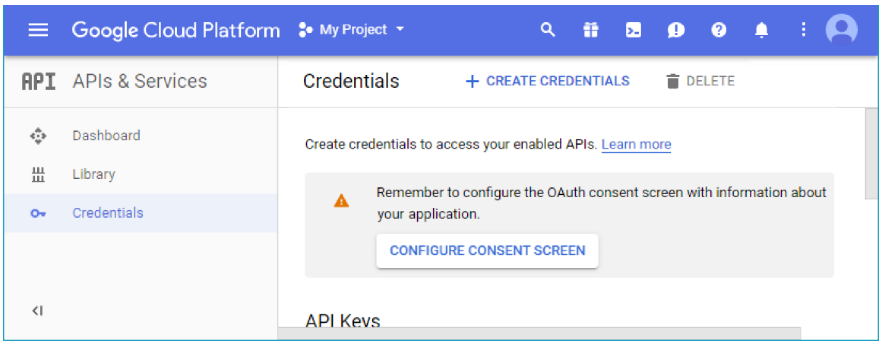


Obrázek 21.3: Otisk lokálního SHA certifikátu získaný pomocí Java nástroje keytool

Pozor, popsanými způsoby jsme získali otisk certifikátu pro účely vývoje, neboli pro „Variant: debug“. Tento otisk zatím bude stačit, pro finální aplikaci, tedy „Variant: release“ je potřeba vygenerovat samostatný otisk. První postup v tom již nepomůže, druhý postup vyžaduje pouze nalezení správné varianty po kompilaci projektu. Pro poslední způsob je nutné upravit cestu k úložišti tak, aby obsahovala keystore pro release.

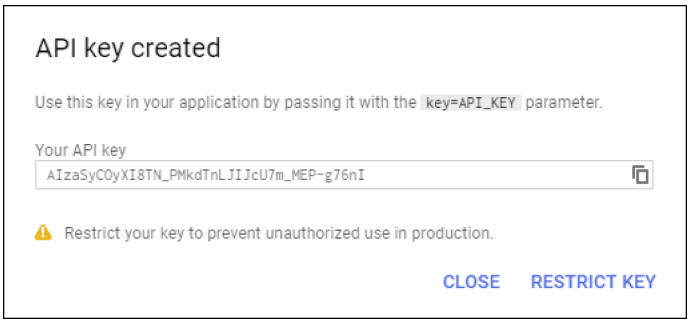
Jakmile máme otisk certifikátu, musíme ho zaregistrovat na stránkách Google API. K tomu potřebujeme ještě znát namespace vyvíjené aplikace. Pokud si ho nepamatujeme, najdeme ho v souboru manifestu app/manifests/AndroidManifest.xml, kde je zapsaný v atributu package.

V prohlížeči otevřeme stránku Google Cloud Platform na adrese https://console.cloud.google.com/, přihlásíme se svým Google uživatelským jménem a přejdeme do APIs & Services → Credentials.



Obrázek 21.4: Registrace otisku certifikátu

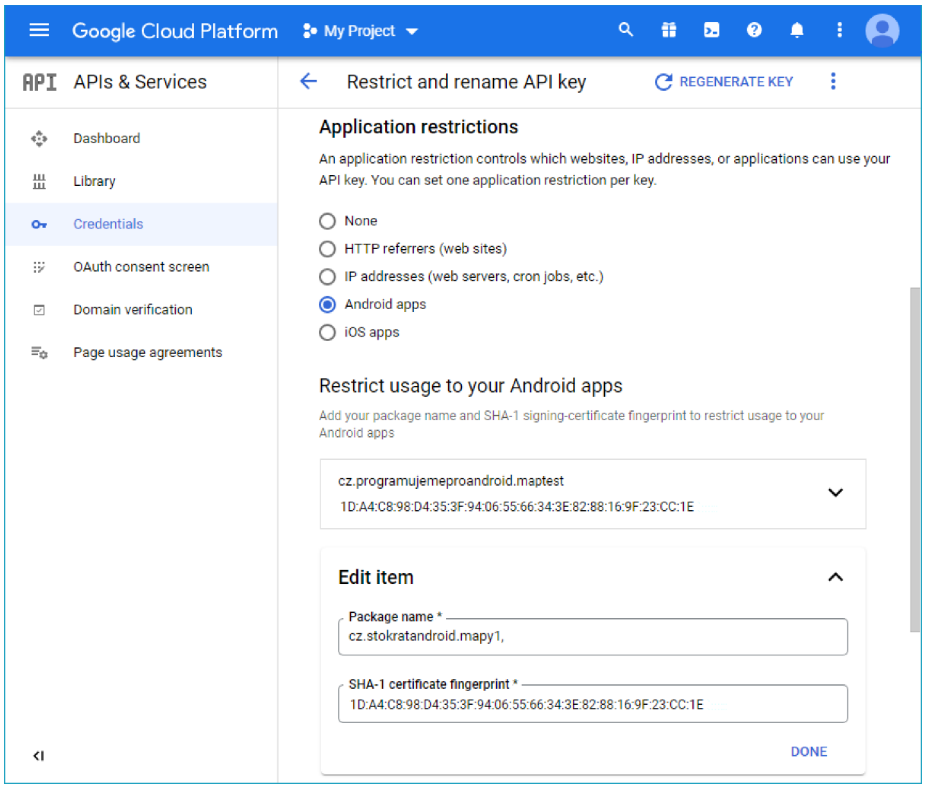
Klikneme na tlačítko Create credentials → API key. Google vygeneruje klíč, který si zkopírujeme. Toto je náš nový klíč k mapovým podkladům, kvůli kterému jsme to celé dělali. Ještě si k němu nastavíme omezení, buď hned po vygenerování tlačítkem Restrict key, nebo později pomocí editace API klíče.



Obrázek 21.5: Klíč k mapovým podkladům, získaný registrací na stránkách Google

Nastavení restrikcí je nutné pro to, aby nebylo možné náš klíč zneužít. Pro účely vývoje Android aplikací se nastavuje omezení na konkrétní aplikaci. Neboli nikdo jiný než vyjmenované aplikace nemůže klíč použít. Aplikace je identifikována pomocí namespace, který jsme zjistili z manifestu. Dobrou zprávou je, že pro všechny naše aplikace už bude stačit jediný API klíč, pouze k němu budeme přiřazovat namespace dalších aplikací.

Přiřazení provedeme takto:



Obrázek 21.6: Přiřazení aplikace ke klíči mapových podkladů

V nastavení API klíče zvolíme Application restrictions na Android apps, klikneme na tlačítko Add package name and fingerprint a do políček vložíme namespace aplikace a vedle otisk SHA1 certifikátu. Pokud připravujeme víc aplikací, můžeme si rovnou zaregistrovat i další namespace.

21.2 Mapy – první zobrazení

Uděláme si základní aplikaci pro ověření funkčnosti. Nebudeme na ni mít nijak vysoké nároky, půjde nám pouze o zobrazení mapy. Jako malý bonus si mapu nastavíme na zvolené souřadnice a přidáme si do ní značku. Žádné další požadavky mít nebudeme.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 21_Mapy
projekt Mapy1

Co je potřeba udělat?

- přidat do manifestu klíč k mapám
- vytvořit formulář s mapovou komponentou
- pokud není nainstalována knihovna Google Play Services, nainstalovat ji
- nastavit v komponentě mapu na zvolené souřadnice a přidat značku

Na úpravě manifestu tentokrát záleží víc než kdy jindy. Je do něj totiž potřeba přidat, do bloku application, mapový klíč, který jsme získali při registraci na stránkách Google.

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="AIzaSyBtij1Qkk0hdX5L7NOrh5m-H9PeUpmXNg0" />
```

Samotný klíč je v parametru value. Musí být aktivovaný pro namespace aplikace, jinak se mapy nezobrazí a může být poměrně těžké zjistit proč.

Formulář bude typu SupportMapFragment. Jelikož tento typ formuláře není v paletě nástrojů, otevřeme si soubor activity_main.xml a vložíme do něj následující kód:

```
<fragment
    android:id="@+id/map"
    class="com.google.android.gms.maps.SupportMapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Jde o typ fragmentu určený pro jednoduché zobrazení mapy. Parametry jsou celkem prosté, id a rozměry (layout_width, layout_height) můžeme podle potřeby změnit. Cílem je mapu pouze zobrazit, takže zatím nepotřebujeme formulář nijak výrazně zkrášlovat.

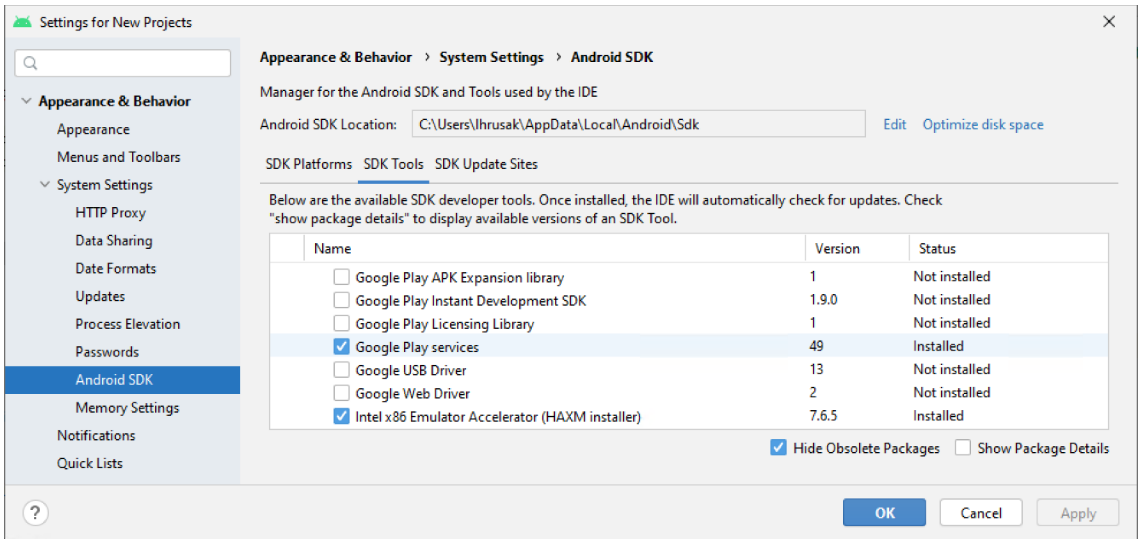
Mapový fragment patří do knihovny Google Play Services, která není standardně součástí Android Studia. A to může být kámen úrazu. Jestliže nebude knihovna nainstalovaná, bude Android Studio zobrazovat červeně podtrženou část namespace třídy SupportMapFragment.

Obrázek 21.7: Zvýrazněný namespace třídy SupportMapFragment signalizuje chybějící knihovnu Google Play Services

Pokud se toto stane, je potřeba zajistit doinstalování Google Play Services. Možností, jak toho dosáhnout je několik. Žádná ale není úplně spolehlivá, je proto potřeba trochu experimentovat.

První a nejjednodušší způsob je, přepnout se do designéru formuláře, vzít v paletě komponent mapové view (složka Google → MapView) a hodit ho do formuláře. Android Studio by mělo nabídnout doinstalování Google Play Services a pokud se to podaří, je záležitost vyřízena. Mapové view můžeme zase z formuláře odstranit, nebudeme ho tam potřebovat.

Zůstává-li nadále část namespace červeně podtržená, signalizuje tím Android Studio, že se instalace nepodařila a je to potřeba zkusit nějak jinak. Další cestou je v Android Studiu menu Tools → SDK Manager, okno Appearance & Behavior → System Settings → Android SDK, záložka SDK Tools. Zde najdeme Google Play Services a nainstalujeme je.



Obrázek 21.8: Doinstalování knihovny Google Play Services do projektu v Android Studiu

Potom se doporučuje ještě Android Studio restartovat a dát překompilovat projekt (menu Build → Rebuild Project). Nyní může a nemusí být vyhráno. Pokud je namespace stále červené, zbývá poslední možnost. Otevřít v okně projektu GradleScripts → build.gradle (Module app), v něm najít část dependencies a přidat do ní řádek:

```
implementation 'com.google.android.gms:play-services-maps:18.1.0'
```

Android Studio případně samo napoví, že existuje novější verze, takže ji ještě upravíme. Potom projekt znovu překompilujeme a pokud Gradle nabídne synchronizaci souborů, potvrdíme mu ji. Namespace by po této proceduře mělo být v pořádku.

Téhož lze dosáhnout přes uživatelské rozhraní Android Studia, v menu File → Project Structure. Podrobněji je postup popsaný v kapitole 11 Závislé zdroje – Dependencies.

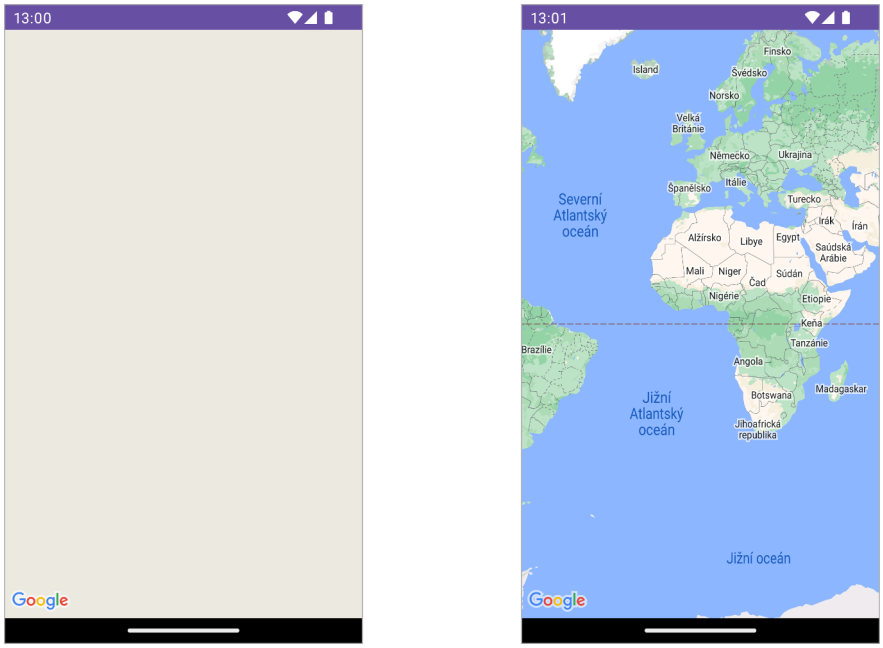
Tímto máme připravené vše potřebné k zobrazení mapy. Můžeme aplikaci zkompilovat a spustit na emulátoru nebo na koncovém zařízení. Měla by se zobrazit mapa světa, víceméně náhodně nastavená. Pokud se nic nezobrazuje a mapová komponenta je prázdná, je nutné zkontrolovat nastavení klíče v manifestu. Je-li v pořádku, zkontrolujeme ještě registraci mapového klíče na stránkách Google. Zejména zaregistrovaný namespace aplikace. Jestliže nic nepomáhá, můžeme v registraci zkusit dočasně změnit omezení z Android apps na None. Tím se přestane kontrolovat jestli má oprávnění právě tato aplikace a mapy jsou posílány každé aplikaci, která se identifikuje zaregistrovaným klíčem. Není to bezpečné, ale dočasně, pro účely identifikace příčiny problému se takové nastavení může hodit.

Aby bylo možné mapy zobrazit, je nutné mít datové připojení. Ať už přes síť nebo přes WiFi. Pokud zařízení nebude moci komunikovat se servery Google, zobrazí se stejná prázdná obrazovka s logem, jako bychom měli špatně nastavenou registraci.

Když se mapa stále nechce zobrazit, hodí se také sledovat informace v konzoli Run Android Studia. Na to, že chybí klíč v manifestu, nebo že není zaregistrovaný u Google, nás může upozornit.

Pokud se na koncovém zařízení objeví mapa, kterou je možné posouvat a přibližovat, je vše v pořádku. Jestliže se ale mapa nenačetla a je zobrazeno pouze logo Google, znamená to, že je chyba někde v registraci aplikace k mapovému serveru. Pokud už aplikace mapu zobrazila, může

být načtena z vyrovnávací paměti. Že je něco v nepořádku však poznáme, jakmile se pokusíme mapu přiblížit.



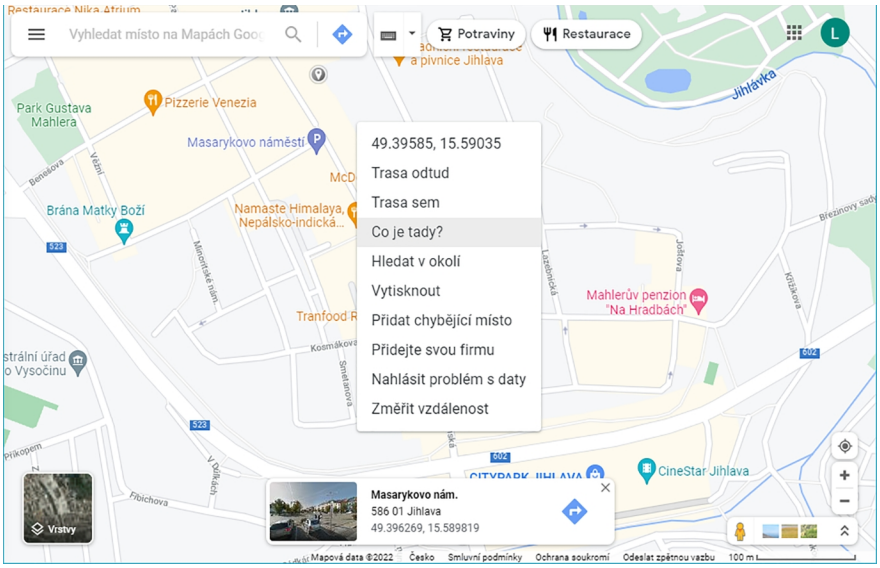
Obrázek 21.9: Zobrazení mapových podkladů v aplikaci. Na levém obrázku je případ, kdy se podklady nepodařilo zobrazit, což může mít několik příčin. Na pravém obrázku je všechno v pořádku

Jakmile se mapa zobrazí, nastavíme její umístění, zvětšení a vložíme do ní značku. Ve třídě MainActivity přidáme do události onCreate() iniciaci fragmentu:

```
SupportMapFragment mapFragment = (SupportMapFragment)
                        getSupportFragmentManager().findFragmentById(R.id.map);
mapFragment.getMapAsync(this);
```

První instrukce najde na formuláři fragment a uloží ho do proměnné mapFragment a druhá nastaví volání události onMapReady(), jakmile je instance mapové komponenty připravena k použití.

Operace s mapou umístíme do události onMapReady(), kterou si pro tento účel přepíšeme. Potřebujeme znát souřadnice místa, na které má být mapa nastavena. Souřadnice potřebujeme v desetinném formátu podporovaném Google mapami. To znamená, zeměpisnou šířku v intervalu -90 až 90 a délku v rozsahu -180 až 180. Zjistíme je snadno pomocí internetového prohlížeče přímo v Google mapách na adrese maps.google.com. Stačí si najít místo, které nás zajímá, kliknout pravým tlačítkem myši do mapy a v menu zvolit „Co je tady?“. Zobrazí se malé okno se souřadnicemi.



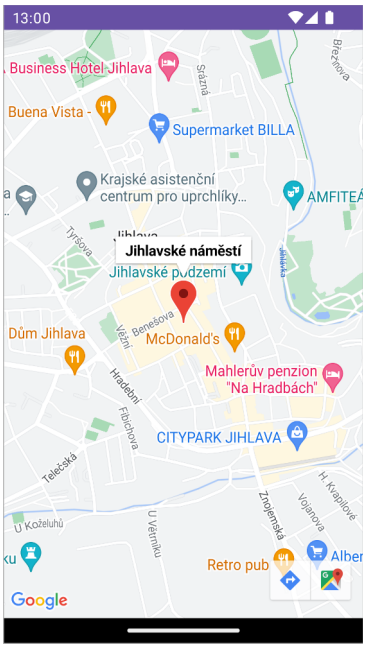
Obrázek 21.10: Způsob získání souřadnic ze stránek maps.google.com

Řekněme, že budeme chtít zobrazit jihlavské náměstí. Nalezené souřadnice jsou přibližně 49.396 a 15.589. Souřadnice si dáme do proměnné datového typu LatLng. Do mapy na tyto souřadnice umístíme pomocí metody addMarker() třídy GoogleMap značku. Potom budeme potřebovat nastavit zvětšení mapy. To je v rozsahu 2 až 20, přičemž platí, že čím vyšší hodnota, tím větší přiblížení. Správnou hodnotu je potřeba vyzkoušet, začněme třeba s přiblížením 15. Metoda moveCamera() mapu zvětší a nastaví na požadované souřadnice.

Událost onMapReady() může tedy vypadat třeba takto:

```
@Override
public void onMapReady(GoogleMap map) {
    LatLng souradnice = new LatLng(49.396, 15.589);
    map.addMarker(new MarkerOptions().position(souradnice).title(
                                                    "Jihlavské náměstí"));
    map.moveCamera(CameraUpdateFactory.newLatLngZoom(souradnice,15));
}
```

Výsledek by měl na emulátoru nebo koncovém zařízení vypadat jako na obrázku 21.11.



Obrázek 21.11: Aplikace s mapovou komponentou SupportMapFragment, spuštěná na emulátoru

21.3 Mapy – komponenta MapView

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 21_Mapy
projekt Mapy2

V předchozím příkladu jsme použili jednoduchou komponentu SupportMapFragment, která díky univerzálnosti a jednoduchosti použití způsobuje při zobrazení map nejméně problémů. Můžeme se ale dostat do situace, kdy budeme mapovou komponentu sami vkládat do vlastního fragmentu. V takovém případě je lepší využít MapView a nadbytečnému vnořování fragmentů se tak vyhnout.

Následující aplikace zobrazí, stejně jako v minulém příkladu, mapu, tentokrát ale pomocí komponenty MapView. I v tomto případě mapu posuneme na zvolené souřadnice, zvětšíme a přidáme do ní značku.

Co je potřeba udělat?

- přidat do manifestu klíč k mapám

- vytvořit formulář s mapovou komponentou

- nastavit v komponentě mapu na zvolené souřadnice a přidat značku

Úprava manifestu se neliší od předchozího příkladu. Prostě do něj vložíme stejný kód s klíčem k mapovým podkladům. Jenom je potřeba dát pozor na to, abychom nezapomněli v registraci na stránkách Google API přidat do omezení také namespace této nové aplikace.

Formulář vytvoříme snadno a rychle. Jednoduše do něj z palety komponent vložíme prvek MapView, nastavíme mu vhodné id a velikost match_parent v obou osách.

Nejvíce práce bude ve třídě MainActivity. Nejdřív musíme přidat implementaci interface OnMapReadyCallback, aby byla volána událost onMapReady() ve chvíli, kdy je mapová komponenta připravena:

```
public class MainActivity extends AppCompatActivity
                                          implements OnMapReadyCallback {
```

Přidáme proměnnou, která bude udržovat ukazatel na komponentu MapView a upravíme událost aktivity onCreate() tak, aby v ní nová proměnná mapView na komponentu odkazovala a aby ji inicializovala.

```
private MapView mapView;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mapView = findViewById(R.id.mapView);
    mapView.onCreate(savedInstanceState);
    mapView.getMapAsync(this);
}
```

Musíme předefinovat události komponenty MapView:

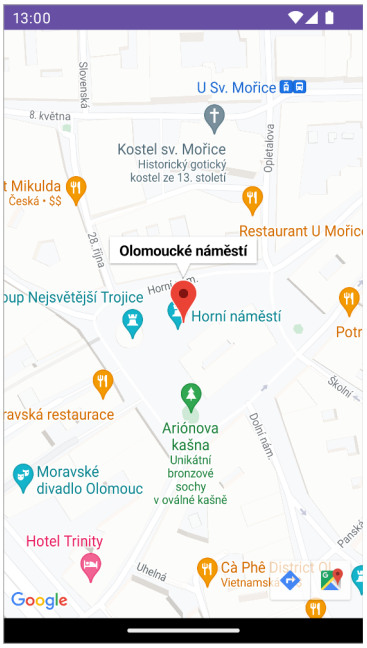
```
@Override
protected void onResume() {
    super.onResume();
    mapView.onResume();
}

@Override
protected void onStart() {
    super.onStart();
    mapView.onStart();
}

@Override
protected void onStop() {
    super.onStop();
    mapView.onStop();
}

@Override
protected void onPause() {
    mapView.onPause();
    super.onPause();
}

@Override
protected void onDestroy() {
    mapView.onDestroy();
    super.onDestroy();
}

@Override
```

```
public void onLowMemory() {
    super.onLowMemory();
    mapView.onLowMemory();
}
```

A nakonec poslední událost, ta která je pro nás nejdůležitější, onMapReady(). V ní nastavíme souřadnice, zvětšení a vytvoříme značku. Postup se neliší od předchozího příkladu, snad jen souřadnice můžeme zvolit jiné.

```
@Override
public void onMapReady(GoogleMap map) {
    LatLng souradnice = new LatLng(49.594, 17.251);
    map.addMarker(new MarkerOptions().position(souradnice)
                                                    .title("Olomoucké náměstí"));
    map.moveCamera(CameraUpdateFactory.newLatLngZoom(souradnice,17));
}
```

Po kompilaci a spuštění by měl být výsledek k nerozeznání podobný od příkladu s komponentou SupportMapFragment. Na emulátoru nebo koncovém zařízení bude aplikace vypadat jako na obrázku 21.12.



Obrázek 21.12: Aplikace s mapovou komponentou MapView, spuštěná na emulátoru

21.4 Mapy – značky

Zobrazení map jsme zvládli, je tedy na čase vyzkoušet některé prakticky využitelné prvky, které mapy nabízí. Zkusíme rozšířit příklad s mapou postavenou na fragmentu SupportMapFragment, o vlastní značku. Ke značce přidáme vlastní informační okno a po ťuknutí do okna vyvoláme událost, která provede definovanou akci.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 21_Mapy
projekt Mapy3

Co je potřeba udělat?

- přidat do manifestu klíč k mapám
- vytvořit formulář s mapovou komponentou
- připravit si ikonu značky a předdefinovanou značku pomocí ní předefinovat
- vytvořit formulář informačního okna a připojit ho ke značce
- udělat událost která bude volaná po ťuknutí do okna

Pokud zakládáme nový projekt, přidáme do manifestu klíč stejně jako v předchozích příkladech. Opět nezapomeneme v registraci na stránkách Google API přidat do omezení namespace této nové aplikace. Ani formulář se neliší od příkladu s mapou postavenou na fragmentu SupportMapFragment. Uděláme ho tedy stejný.

Ve třídě MainActivity přidáme do události onCreate() iniciaci fragmentu:

```
SupportMapFragment mapFragment = (SupportMapFragment)
                        getSupportFragmentManager().findFragmentById(R.id.map);
mapFragment.getMapAsync(this);
```

Dále si musíme pořídit ikonku, která bude nahrazovat mapovou značku. Můžeme si ji nakreslit, nebo stáhnout z internetu některou z volně použitelných ikon. Měla by mít velikost asi tak do 50 obrazových bodů v každé ose. Ikonku uložíme do projektového adresáře app/res/drawable/.

Třídu MainActivity rozšíříme o implementaci interface OnMapReadyCallback, tak, jak to už známe:

```
public class MainActivity extends AppCompatActivity
                                          implements OnMapReadyCallback {
```

Díky tomu můžeme předefinovat událost onMapReady(), která bude Androidem zavolaná ve chvíli, kdy bude mapa připravena. Do události umístíme nastavení značky a oproti předchozím příkladům nastavení značky rozepíšeme do proměnné vlastnostiZnacky. Do ní potom vložíme, pomocí vlastnosti icon, také vlastní ikonku.

```
@Override
public void onMapReady(GoogleMap map) {
    LatLng souradnice = new LatLng(49.939, 14.188);

    // nastavit vlastnosti znacky
    MarkerOptions vlastnostiZnacky = new MarkerOptions().position(souradnice);
    vlastnostiZnacky.icon(
                BitmapDescriptorFactory.fromResource(R.drawable.znacka));

    Marker znacka1 = map.addMarker(vlastnostiZnacky);
    znacka1.setTag("znacka1");

    // -- sem pozdeji vlozime adapter informacniho okna a udalost kliknuti --

    // nastavit pozici mapy a jeji zvetseni
    map.moveCamera(CameraUpdateFactory.newLatLngZoom(souradnice,12));
}
```

Jelikož v mapě může být více značek, budeme potřebovat každou z nich později identifikovat, abychom v souvislosti s ní mohli zobrazovat správný obsah. Dá se to udělat několika způsoby. Buď si budeme držet v proměnné ukazatel na značku – k tomu by stačilo udělat z lokální proměnné znacka1 proměnnou třídy a pak bychom se na ni mohli odkazovat. Nebo využít metody getId() značky, která vrátí jednoznačný identifikátor od libovolné značky. Nevýhodou je, že se ID generují automaticky při založení značky a vývojář je nemá možnost nastavit. Takže bychom se museli smířit s tím, že jsou v takové podobě, jak si ji stanoví Android. Tato id se můžeme ukládat například do pole a později podle nich zjistíme, o kterou značku jde. Asi nejelegantnější možností je ale použití metody setTag(), která umožní každou značku označit textovým řetězcem, podle kterého ji pak lze identifikovat. My si takto naši značku označíme například textem znacka1. Později se nám to bude hodit.

Projekt můžeme zkompilovat a spustit na emulátoru nebo koncovém zařízení. Měla by se zobrazit mapa s naší vlastní značkou.

Dalším krokem je vytvoření uživatelsky definovaného, informačního okna, které se otevře po ťuknutí na ikonku. Založíme si nový formulář v projektovém adresáři app/res/layout/. Nazveme ho třeba marker_info.xml. Můžeme do něj dát textové pole a případně obrázek, podle vlastního uvážení.

Abychom mohli informační okno zobrazit, musíme si vytvořit adaptér typu InfoWindowAdapter a ten přiřadit k mapě pomocí metody setInfoWindowAdapter(). Tento adaptér vyžaduje předefinování dvou metod getInfoWindow() a getInfoContents(). První z nich slouží k definici kompletního vzhledu informačního okna, takže nemusí vypadat jako obdélník, ale třeba jako ovál nebo kulatá bublina. Druhá metoda umožňuje jednoduše vložit obsah do okna, přičemž jeho vzhled je daný – je to prostý obdélník. Jednu z těchto dvou metod musíme využít, druhou stačí pouze ukončit pomocí instrukce return null. Pro jednoduchost použijeme druhou metodu. Adaptér bude vypadat takto:

```
map.setInfoWindowAdapter(new GoogleMap.InfoWindowAdapter() {
    // definice kompletniho okna (nevyuzijeme)
    @Override
    public View getInfoWindow(Marker znacka) {
        return null;
    }

    // definice obsahu informacniho okna
    @Override
    public View getInfoContents(Marker znacka) {
        // ukazatel na formular marker_info
        View v = getLayoutInflater().inflate(R.layout.marker_info, null);

        // ziskar reference na objekty ve formulari
        TextView txtNadpis = (TextView) v.findViewById(R.id.txtNadpis);
        ImageView imgObrazek = (ImageView) v.findViewById(R.id.imgObrazek);
        TextView txtPopis = (TextView) v.findViewById(R.id.txtPopis);

        if (znacka.getTag()=="znacka1") {
            txtNadpis.setText("Hrad Karlštejn");
            imgObrazek.setImageResource(R.drawable.karlstejn);
            txtPopis.setText("Středověký hrad, založený králem Karlem IV...");
        }
        return v;
    }
});
```

Obsah adaptéru není nijak složitý. Pomocí inflateru si připravíme formulář, najdeme v něm jednotlivé objekty a ty vyplníme požadovaným obsahem. Zde využijeme text určený pro identifikaci značky, získaný pomocí getTag(). Nakonec připravený formulář odešleme v návratové hodnotě. Celý tento adaptér umístíme do události onMapReady(), hned za definici značky.

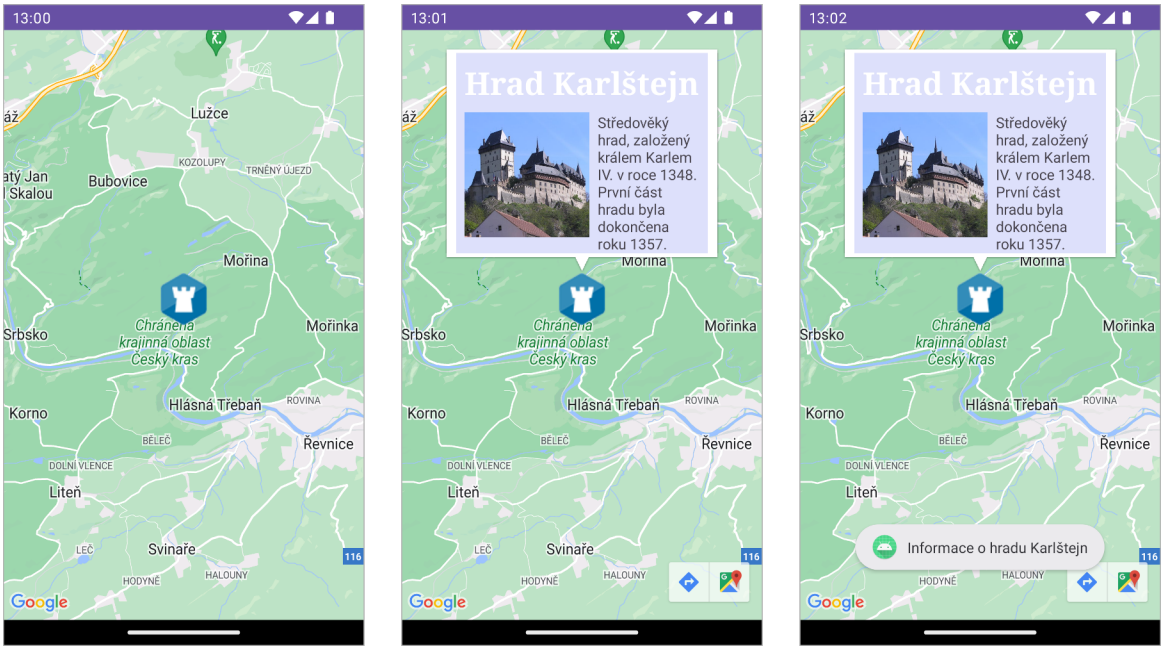
Opět můžeme zkompilovat a spustit. Aplikace by měla zobrazit naši značku a po ťuknutí na ni otevřít okno s doplněnými informacemi.

Předpokládejme, že koncový uživatel bude chtít do zvoleného informačního okna ťuknout. Například aby získal další informace. Událost kliknutí se jmenuje onInfoWindowClick() a můžeme ji samozřejmě předefinovat.

```
map.setOnInfoWindowClickListener(new GoogleMap.OnInfoWindowClickListener() {
    @Override
    public void onInfoWindowClick(Marker znacka) {
        if (znacka.getTag()=="znacka1") {
            Toast.makeText(MainActivity.this, "Informace o hradu Karlštejn",
                                                Toast.LENGTH_SHORT).show();
        }
    }
});
```

Zde opět využijeme identifikace značky pomocí metody getTag(). V události bychom mohli například otevřít aktivitu s dalším formulářem. Spokojíme se ale v tomto případě s pouhým zobrazením informace pomocí Toast.

Po kompilaci a spuštění by měl být výsledek vypadat jako na následujícím obrázku. První je mapa s vlastní značkou, uprostřed je s otevřeným informačním oknem a vpravo po kliknutí do informačního okna.



Obrázek 21.13: Aplikace s mapovou komponentou zobrazující mapový podklad, vlastní značku a definované informační okno

21.5 Kreslení do map

Při práci s mapovými podklady si pravděpodobně nevystačíme se značkami. Má-li aplikace zobrazovat trasu nebo objekty, budeme potřebovat zakreslit do mapy spojnice, polygony, kružnice. Díky knihovně GoogleMap k tomu máme všechny nezbytné prostředky.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 21_Mapy
projekt Mapy4

Zkusíme si udělat příklad zakreslení reálné trasy do mapy. Začátek i konec trasy označíme.

Co je potřeba udělat?

- přidat do manifestu klíč k mapám
- vytvořit formulář s mapovou komponentou
- připravit si souřadnice trasy
- zakreslit trasu do mapy
- připravit si ikonku počátku a konce trasy
- ikonky umístit na začátek a konec trasy

Klíč přidáme stejně jako v předchozích příkladech. Ani formulář pro zobrazení mapy se nijak neliší, opět použijeme některý ze tří známých způsobů – MapView, MapFragment nebo SupportMapFragment.

Pro účely příkladu bude stačit, když souřadnice vložíme přímo do kódu. K tomuto účelu si na maps.google.com, nebo v jiné mapové aplikaci, najdeme reálné souřadnice. Potřebovat budeme alespoň čtyři body.

Zakreslení souřadnic do mapy není nijak složité. Nejprve zobrazíme známým způsobem mapu - v události onMapReady() třídy MainActivity zadáme souřadnice středu a pomocí metody moveCamera() nastavíme zobrazení mapy na požadovaných souřadnicích a s požadovaným zvětšením. Potom vytvoříme objekt typu PolylineOptions, který ponese všechny informace o trase. Tedy zejména barvu a tloušťku čáry, tvar konce spojnice, zakřivení, jednotlivé souřadnice a tak dále. Do mapy přidáme spojnice metodou addPolyline() třídy GoogleMap.

Zdrojový kód bude vypadat nějak takto:

```
@Override
public void onMapReady(GoogleMap map) {

    // nastavit pozici mapy a jeji zvetseni
    LatLng souradnice = new LatLng(49.9453, 14.1871);
    map.moveCamera(CameraUpdateFactory.newLatLngZoom(souradnice,13));

    // trasa
    PolylineOptions trasaVlastnosti = new PolylineOptions();
    trasaVlastnosti.geodesic(true);
    trasaVlastnosti.color(Color.GRAY);
    trasaVlastnosti.add(new LatLng(49.9323, 14.1753));
    trasaVlastnosti.add(new LatLng(49.9328, 14.1799));
    ...
    trasaVlastnosti.add(new LatLng(49.9583, 14.2067));
    trasaVlastnosti.add(new LatLng(49.9611, 14.2036));
```

```
    map.addPolyline(trasaVlastnosti);
}
```

Kromě souřadnic jsme nastavili příznak geodesic na true. To znamená, že má být při zobrazení bráno v potaz zakřivení Země. Neboli spojnice mezi dvěma body ve 2d prostoru není přímka, ale křivka. Pokud bychom nastavili příznak na false, vykreslila by se pokaždé přímka. Zakřivení se projeví pouze při zobrazení spojnice mezi body, které jsou vzdáleny alespoň stovky kilometrů. V případě blízkých bodů na tomto parametru příliš nezáleží, zakřivení není zřetelné. Dalším parametrem je barva spojnice a mohli bychom použít také width pro nastavení tloušťky čáry v pixelech. Zde ponecháme přednastavenou hodnotu, kterou neměníme. Souřadnice přidáváme běžným způsobem pomocí instancí LatLng. Nakonec celý objekt trasaVlastnosti přiřadíme pomocí metody addPolyline() k mapě a mapová komponenta se postará o vykreslení trasy.

Dalšími zajímavými metodami třídy PolylineOptions, které můžeme použít, je dvojice startCap() a endCap(). Pomocí nich můžeme definovat, jak má vypadat začátek první a konec poslední spojnice. Zdánlivě nic zajímavého. Jednoduše můžeme před zobrazením trasy říct, že má být konec (nebo začátek) zaoblený a je to vyřízené:

```
trasaVlastnosti.endCap(new RoundCap());
```

Jenže! Zakončení lze definovat také uživatelsky, pomocí instance třídy CustomCap. Toho můžeme využít pro definici zakončení pomocí vlastní ikonky, kterou umístíme do složky app/res/drawable/. Ikonku přiřadíme tímto způsobem:

```
trasaVlastnosti.startCap(new CustomCap
                  (BitmapDescriptorFactory.fromResource(R.drawable.start)));
```

Takto přiřazená ikonka se bude chovat poněkud jinak než značka. Zatímco ta je vždy ve svislé poloze, pevně svázaná se souřadnicí, ikonka CustomCap se natáčí ve směru první spojnice. Aniž bychom museli jakkoliv dál nastavovat její pootočení, sama si upraví pozici vždy do stejného úhlu jako má spojnice. A to i tehdy, když mapu zmenšíme natolik, že se první spojnice stane neviditelnou. Ikonka CustomCap se jednoduše otočí ve směru první spojnice, která je ještě vidět.

Pro srovnání můžeme na konec trasy umístit ikonku pomocí značky tak, jak to již umíme. Na displeji uvidíme rozdíl v chování obou ikonek.

```
LatLng souradniceCil = new LatLng(49.9611, 14.2036);
MarkerOptions vlastnostiZnacky = new MarkerOptions().position(souradniceCil);
vlastnostiZnacky.icon(BitmapDescriptorFactory.fromResource(R.drawable.cil));
map.addMarker(vlastnostiZnacky);
```

Kromě kreslení spojnic můžeme kreslit také polygony. Toho využijeme kdykoliv, kdy potřebujeme označit nějaký objekt nebo zvýraznit oblast. Způsob práce s polygonem je podobný práci se spojnicemi. Pomocí PolygonOptions nastavíme jeho vlastnosti, přidáme souřadnice a mapová komponenta se postará o vykreslení.

```
@Override
public void onMapReady(GoogleMap map) {

    // nastavit pozici mapy a jeji zvetseni
    LatLng souradnice = new LatLng(49.9453, 14.1871);
```

```
    map.moveCamera(CameraUpdateFactory.newLatLngZoom(souradnice,13));

    // polygon
    PolygonOptions polygonVlastnosti = new PolygonOptions();
    polygonVlastnosti.add(new LatLng(49.9415, 14.1919));
    polygonVlastnosti.add(new LatLng(49.9411, 14.1873));
    ...
    polygonVlastnosti.add(new LatLng(49.9385, 14.1918));

    polygonVlastnosti.strokeColor(Color.DKGRAY);
    polygonVlastnosti.fillColor(0x40808080);

    Polygon polygon = map.addPolygon(polygonVlastnosti);
}
```

Metodou strokeColor() nastavujeme barvu ohraničení, fillColor() slouží k nastavení barvy výplně. Nakonec pomocí addPolygon() přidáme polygon do mapy.

Zde se zastavíme u nastavení barvy, která je v příkladu zvolena jako „0x40808080“. Zápis 0x znamená, že budou následovat hexadecimální hodnoty. Tedy hodnoty 0x00 až 0xFF, neboli dekadicky 0-255. Tyto hodnoty jsou v kódu barvy celkem čtyři. První (0x40, čili dekadicky 64) je hodnota alfa kanálu, neboli definice průhlednosti výplně. Čím nižší hodnota, tím je výplň průhlednější. Následují tři hodnoty jednotlivých barevných složek RGB. Zde jsou všechny tři hodnoty stejné, 0x80, což dá dohromady světle šedou barvu. Dá se tedy říct, že takto použitá definice barvy „0x40808080“ znamená světle šedou, průhlednou výplň polygonu. Stejnou definici, včetně průhlednosti, bychom mohli použít i při nastavení barvy ohraničení. Nebo v případě kreslení spojnic Polyline. Ale u výplní to dává větší smysl.

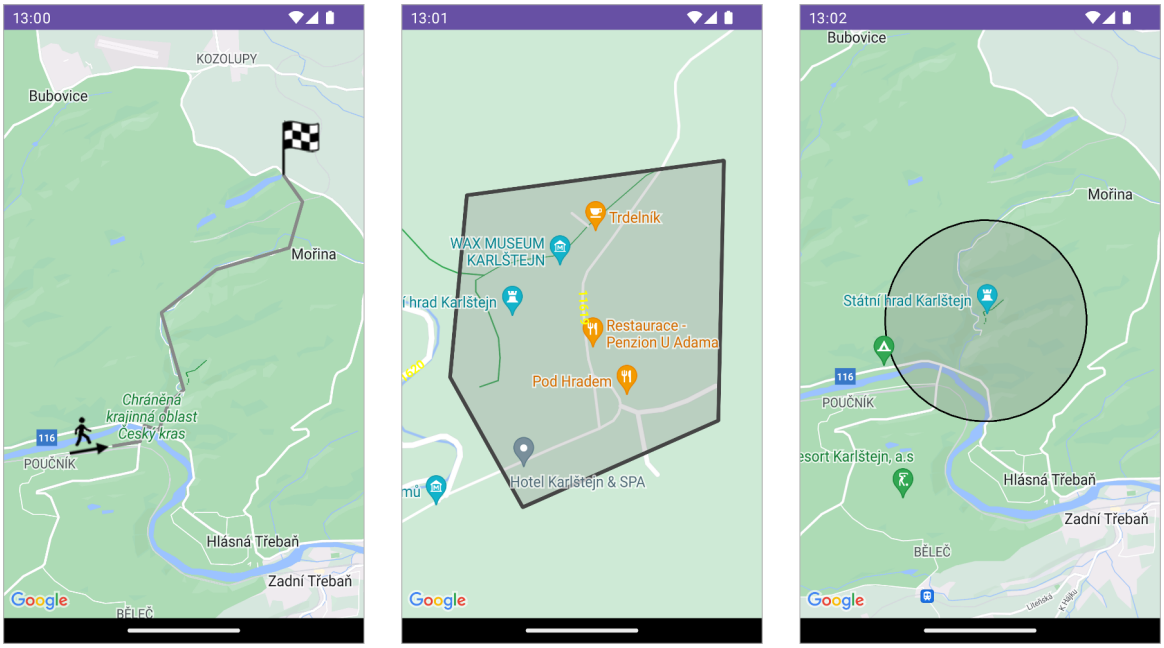
Dalším objektem, který můžeme upotřebit je kružnice. Práce s ní je podobná. Její vlastnosti nastavíme pomocí CircleOptions. Zajímat nás bude zejména střed, průměr, barva a tloušťka ohraničení a samozřejmě barva výplně.

```
@Override
public void onMapReady(GoogleMap map) {

    // nastavit pozici mapy a jeji zvetseni
    LatLng souradnice = new LatLng(49.9453, 14.1871);
    map.moveCamera(CameraUpdateFactory.newLatLngZoom(souradnice,13));

    // kruh
    LatLng souradniceKruhu = new LatLng(49.9392, 14.1879);
    CircleOptions circleOptions = new CircleOptions();
    circleOptions.center(souradniceKruhu);
    circleOptions.radius(1200);
    circleOptions.strokeColor(Color.BLACK);
    circleOptions.fillColor(0x44888888);
    circleOptions.strokeWidth(5);

    map.addCircle(circleOptions);
}
```

Kromě vlastností, které jsme si ukázali, můžeme nastavovat u většiny objektů také jejich Z-index, neboli umístění v ose Z. To se hodí zejména tehdy, je-li zakresleno více překrývajících se objektů. U polygonu je možné také nastavit pomocí setHoles() jakýsi vnitřní polygon, který už ale nemá výplň. Tím se dá docílit inverzního zobrazení polygonu, jakéhosi ztmavení celé mapy, kromě oblasti dané vnitřním polygonem.

Objekty zakreslené v mapě mohou být klikací, čili po ťuknutí na objekt lze vyvolat událost. Toho se dá využít například, pokud při zakreslení trasy chceme uživateli zobrazit podrobné údaje o každém úseku, který si na displeji označí.

Po kompilaci a spuštění by měl být výsledek vypadat jako na následujícím obrázku. První je mapa s trasou, uprostřed polygon a vpravo mapa s kružnicí.



Obrázek 21.14: Aplikace s mapovou komponentou a zakreslenou trasou, polygonem a kružnicí

22 Systémová data

Aby mohl Android fungovat, má ve svých vnitřních strukturách celou řadu ukazatelů aktuálního stavu a docela velké množství provozních dat, která jsou systémem průběžně využívána. Některá z nich jsou interní, ale jiná jsou uživateli přístupná prostřednictvím GUI. Například indikátor stavu baterie, intenzity signálu a tak podobně. Část dat pořizuje sám uživatel, například tak, že napíše textovou zprávu, která se uloží do vnitřní paměti. K některým datům mají prostřednictvím API přístup i aplikace. Těmi se budeme dále zabývat, jednak coby stavovými atributy systému a potom i prací se systémem průběžně ukládanými daty.

22.1 Stav zařízení

Aplikace často potřebují znát stavové atributy systému. Někdy proto, že samy slouží jako monitorovací nástroje, jindy z toho důvodu, že je některá z jejich funkcí na stavu zařízení závislá, nebo se podle stavu zařízení rozhodne, v jakém režimu bude fungovat. Například pokud je k dispozici WiFi připojení, může aplikace stahovat větší množství dat, při mobilním připojení fungovat úsporněji, nebo případně přejít do režimu offline. Stavových ukazatelů má Android velké množství, často daných uživatelským nastavením, jako je třeba intenzita osvětlení displeje, nebo postavených na vnitřních senzorech, jako je teplota baterie. Řadu z nich je možné číst a hodnoty následně vývojářsky využít.

22.1.1 Stav baterie

U některých aplikací, zejména pokud provádějí časté operace na pozadí operačního systému, je užitečné znát aktuální stav baterie. Můžou se tak v případě potřeby samy přepnout do úsporného módu, nebo varovat uživatele. Systém poskytuje rozhraní, které umožňuje zjistit jak okamžitý stav baterie, tak i průběžně monitorovat změny úrovně jejího nabití. Zkusíme si to na příkladu.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 22_Systemova_data
projekt StavBaterie

Co je potřeba udělat?

- připravit formulář, který bude zobrazovat stav baterie
- napsat metodu pro zjištění stavu baterie
- napsat událost pro zaznamenávání změn stavu baterie

Formulář bude úplně jednoduchý. Ze začátku postačí, když zobrazí v TextView prvku stav nabití baterie. K tomuto účelu slouží třída BatteryManager, která dokáže úroveň baterie nejen monitorovat, ale také zjistit spoustu jejich zajímavých parametrů – kromě hodnoty nabití i její celkový ‚zdravotní' stav, jestli právě není nabíjena a případně jakým způsobem, jakou má teplotu, jestli se nepřehřívá, umí spočítat za přibližně jakou dobu se baterie vybije, nebo naopak jak dlouho se bude ještě nabíjet do sta procent. My se pro začátek spokojíme se zjištěním aktuální úrovně nabití.

K tomu si připravíme metodu zjistitStavBaterie(), která pomocí volání getSystemService(), s parametrem BATTERY_SERVICE založí instanci třídy BatteryManager. Potom zavoláme její metodu getIntProperty() pro načtení hodnoty typu integer a v parametru specifikujeme konstantou

BATTERY_PROPERTY_CAPACITY, že nás zajímá aktuální úroveň nabití. Získanou hodnotu vypíšeme do TextView na formuláři.

```
private void zjistitStavBaterie () {
    BatteryManager bm = (BatteryManager) getSystemService(BATTERY_SERVICE);
    int stavBaterie =
                bm.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY);


    TextView txtStav = (TextView)findViewById(R.id.txtStavSpusteni);
    txtStav.setText(String.format("Stav při spuštění: %d%%", stavBaterie));
};
```

Metodu můžeme volat přímo z události onCreate() aktivity MainActivity:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    zjistitStavBaterie();
});
```

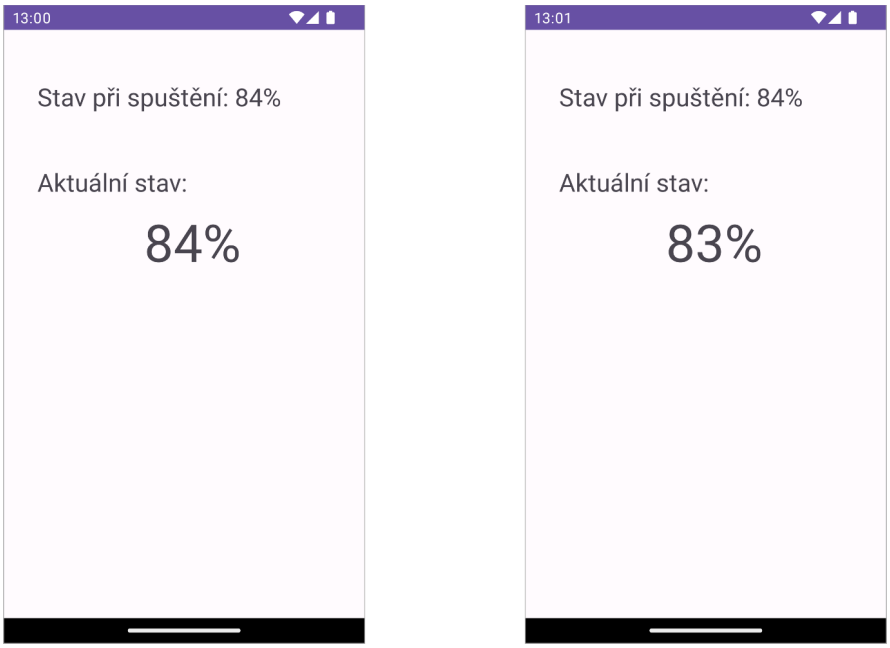
Tím ale získáme pouze stav baterie v okamžiku volání metody. Pro průběžné monitorování stavu je potřeba, aby systém při každé změně úrovně zavolal událost, která zjistí aktuální stav a vypíše ho na displej. K tomu slouží BroadcastReceiver, třída, která umí přijímat systémové události. Stav baterie se v takovém případě vypočítává z předaných hodnot LEVEL a SCALE, z nichž první udává hodnotu nabití a druhý maximální možnou hodnotu nabití. Rozsah bývá obvykle nastavený na 0-100.

```
private TextView txtStavAktualni;
private final BroadcastReceiver mBatInfoReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        int level = intent.getIntExtra(BatteryManager.EXTRA_LEVEL, 100);
        int scale = intent.getIntExtra(BatteryManager.EXTRA_SCALE, 100);
        int stavBaterie = level * 100 / scale;
        txtStavAktualni.setText(String.valueOf(stavBaterie) + "%");
    }
};
```

Ještě je potřeba v události onCreate() nastavit ukazatel na textové pole na formuláři a zaregistrovat přijímač události změny nabití baterie:

```
txtStavAktualni = (TextView) this.findViewById(R.id.txtStavAktualni);
this.registerReceiver(this.mBatInfoReceiver, new
                        IntentFilter(Intent.ACTION_BATTERY_CHANGED));
```

Aplikaci je lepší ladit na reálném zařízení, emulátor Android Studia sice umožňuje nastavit úrovně nabití baterie, ale ne vždy funguje volání událostí při změně přes nastavení parametrů emulátoru. Výsledná aplikace by měla vypadat přibližně jako na obrázku 22.1.



Obrázek 22.1: Zobrazení stavu baterie při spuštění aplikace a po jejím vybití o jedno procento

22.1.2 Datové služby

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 22_Systemova_data
projekt DatovePripojeni

Pokud aplikace potřebuje přistupovat k online datům, je důležité znát stav připojení. V zásadě rozlišujeme tři základní stavy, offline režim, režim s mobilními daty (pokud je zařízení podporuje) a režim s jiným přístupem k datům. Do poslední kategorie patří zejména datové připojení prostřednictvím Wi-Fi, ale také přes Bluetooth nebo ethernetový kabel. Systém opět podporuje jak zjištění okamžitého stavu, tak i monitorování jeho změn. Vyzkoušíme na příkladu.

Co je potřeba udělat?

- připravit formulář, který bude zobrazovat stav datových služeb
- přidat do manifestu žádost o potřebné oprávnění
- napsat metodu pro zjištění stavu datových služeb
- napsat událost pro zaznamenávání změn stavu datových služeb

I v tomto případě si vystačíme s jednoduchým formulářem. Obsahovat bude TextView, které textově zobrazí stav datových služeb.

Aby mohla aplikace detekovat zapnutí datových přenosů, je nutné upravit oprávnění v manifestu. Přidáme tedy do souboru manifest.xml tuto žádost o povolení:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Ke zjišťování stavu datových služeb slouží třída ConnectivityManager, která dokáže zjistit aktuální parametry připojení. Rozhraní třídy zpřístupňuje, mimo jiné, metodu getActiveNetwork(). Ta vrací v objektu datového typu Network informace o aktuálním datovém připojení. Nebo null, pokud není aktivní žádné připojení. Pomocí další metody, getNetworkCapabilities() je možné z tohoto objektu získat informaci o typu datového připojení. To potom srovnáním s konstantami jednotlivých typů připojení stačí k tomu, aby bylo možné určit aktuálně použitý typ datového připojení. Takto si sestavíme metodu, která zjistí, jaký typ připojení je právě aktivní:

```
private String zjistitStavPripojeni(Context context) {
    ConnectivityManager cm = (ConnectivityManager)
                             context.getSystemService(CONNECTIVITY_SERVICE);

    Network pripojeni = cm.getActiveNetwork();
    if (pripojeni == null)
        return "Bez připojení";

    NetworkCapabilities capabilities = cm.getNetworkCapabilities(pripojeni);
    if (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR))
        return "Mobilní síť";
    else if (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI))
        return "WiFi";
    else if (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_BLUETOOTH))
        return "Bluetooth";
    else if (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_ETHERNET))
        return "Ethernet";
    else
        return "Jiné";
};
```

Metodu lze volat přímo z události onCreate() aktivity MainActivity a výsledek rovnou zapsat do formuláře:

```
String stavPripojeni = zjistitStavPripojeni(this);
txtStav.setText(stavPripojeni);
```

Tím získáme stav připojení platný v okamžiku volání metody. Pro průběžné monitorování stavu je potřeba, aby systém při každé změně typu připojení zavolal událost, která zjistí aktuální stav a vypíše ho na displej. K tomu slouží třída BroadcastReceiver, která umí přijímat systémové události. Změna se projeví vyvoláním události onReceive(). Do té umístíme kód, který jsme před chvílí využili v metodě zjistitStavPripojeni() a kterou tímto nahradíme:

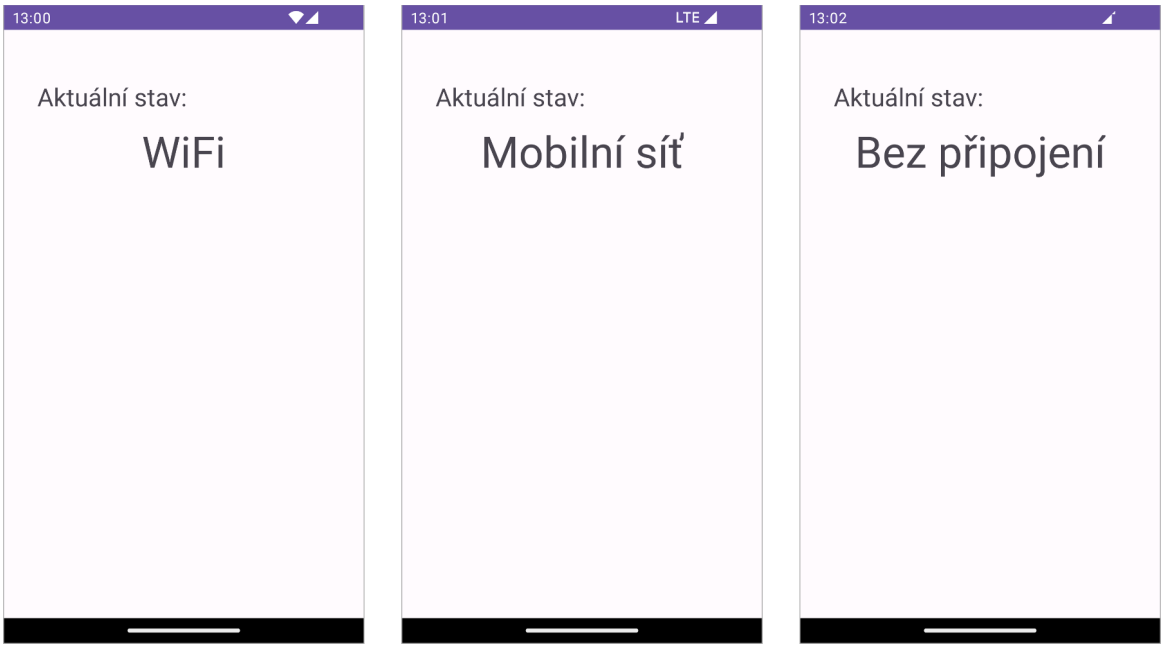
```
boolean broadcastAktivni = false;
static TextView txtStav;

// zachytavani zmeny typu datoveho pripojeni
private static final BroadcastReceiver broadcastReceiver =
                             new BroadcastReceiver()
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        ConnectivityManager cm = (ConnectivityManager)
                             context.getSystemService(CONNECTIVITY_SERVICE);

        Network pripojeni = cm.getActiveNetwork();
```

```
        if (pripojeni == null) {
            txtStav.setText("Bez připojení");
            return;
        }

        NetworkCapabilities capabilities = cm.getNetworkCapabilities(pripojeni);
        if (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR))
            txtStav.setText("Mobilní síť");
        else if (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI))
            txtStav.setText("WiFi");
        else if (capabilities.hasTransport
                              (NetworkCapabilities.TRANSPORT_BLUETOOTH))
            txtStav.setText("Bluetooth");
        else if (capabilities.hasTransport
                              (NetworkCapabilities.TRANSPORT_ETHERNET))
            txtStav.setText("Ethernet");
        else
            txtStav.setText("Jiné");
    }
};
```

Aby BroadcastReceiver mohl události přijímat, je ho ještě nutné zaregistrovat a nastavit mu filtr na akci CONNECTIVITY_ACTION. Jelikož by mohlo dojít k vícenásobné registraci, uložíme do pomocné proměnné broadcastAktivni příznak, že už byl BroadcastReceiver jednou zaregistrován. To celé, ještě spolu s inicializací proměnné txtStav, odkazující na prvek TextView na formuláři, umístíme do události onCreate().

```
txtStav = (TextView) findViewById(R.id.txtStav);

if (!broadcastAktivni) {
    IntentFilter intentFilter = new IntentFilter();
    intentFilter.addAction(ConnectivityManager.CONNECTIVITY_ACTION);
    registerReceiver(broadcastReceiver, intentFilter);
    broadcastAktivni = true;
}
```

Na zařízení, nebo na emulátoru, by se aplikace měla chovat jako na obrázku 22.2. Při každé změně typu datového připojení vyvolá systém událost onReceive(), která aktualizuje zobrazenou informaci. Pokud dojde k pádu během volání metody getActiveNetwork(), znamená to, že nejsou v manifestu správně nastavena práva aplikace pro detekování typu datového připojení.



Obrázek 22.2: Zobrazení stavu datového připojení

22.1.3 Velikost paměti a datového úložiště

Stav paměti je u mobilních zařízení poměrně zásadní informací. A to jak velikost volné a obsazené operační paměti, která je přidělována aplikacím, tak stav datového úložiště, kam si aplikace můžou odkládat provozní data. Jak si ukážeme, ke všem těmto hodnotám se můžeme prostřednictvím API dostat.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 22_Systemova_data
projekt VelikostPameti

Co je potřeba udělat?

- připravit formulář, který bude zobrazovat velikost operační paměti a datového úložiště
- napsat kód pro zjištění aktuálního stavu

Formulář bude kromě popisků zahrnovat čtyři pole TextView, pro zobrazení celkové velikosti a volného místa, jak pro paměť, tak pro úložiště.

Ke zjišťování stavu paměti slouží třída ActivityManager, která dokáže zjistit její aktuální parametry. Rozhraní třídy zpřístupňuje, mimo jiné, metodu getMemoryInfo(). Ta vrací v objektu datového typu MemoryInfo informace o volné paměti a o její celkové velikosti:

```
ActivityManager actManager = (ActivityManager)
                                    getSystemService(ACTIVITY_SERVICE);
ActivityManager.MemoryInfo memInfo = new ActivityManager.MemoryInfo();
actManager.getMemoryInfo(memInfo);
float operacniPametCelkem = (float) memInfo.totalMem / (1000*1000*1000);
float operacniPametVolna = (float) memInfo.availMem / (1000*1000*1000);
```

Parametry totalMem a availMem vrací velikost paměti v bytech, což není ideální pro čtení. Proto ji pomocí konstanty 1000*1000*1000 převedeme na GB. Zde je potřeba poznamenat, že zatímco ve světě IT se pro vyjádření velikosti paměti běžně používají násobky 1024, neboli 2^10, vycházející z dvojkové soustavy, u mobilních zařízení výrobci užívají desítkovou soustavu a tedy násobky 1000.

Jakmile jsou hodnoty vypočítané, můžeme je formátovaně, na dvě desetinná místa, zobrazit na displeji:

```
TextView txtPametCelkem = (TextView) findViewById(R.id.txtPametCelkem);
TextView txtPametVolna = (TextView) findViewById(R.id.txtPametVolna);

txtPametCelkem.setText(String.format("%.02f GB", operacniPametCelkem));
txtPametVolna.setText(String.format("%.02f GB", operacniPametVolna));
```

Pro stanovení velikosti datového úložiště můžeme využít hned několika způsobů. Asi nejjednodušší je třída StatFs, která má původ v Unixu a která umí vrátit celkovou velikost úložiště a jeho volné místo podle cesty v souborovém systému, předané ve formě parametru. Výsledek je v bytech, čili v Androidu pro převod na GB opět dělíme pomocí násobků tisíce:

```
File path = Environment.getDataDirectory();
StatFs stat = new StatFs(path.getPath());
float ulozisteCelkem = (float)stat.getTotalBytes() / (1000*1000*1000);
float ulozisteVolne = (float)stat.getFreeBytes() / (1000*1000*1000);
```

Podobnou, jenom techničtější alternativou stanovení prostoru v úložišti je zjistit pomocí třídy StatFs velikost paměťových bloků a jejich počet. To potom stačí vynásobit a měl by vyjít stejný výsledek jako v předchozím případě:

```
File cesta = Environment.getDataDirectory();
StatFs stat = new StatFs(cesta.getPath());
long velikostBloku = stat.getBlockSizeLong();
long celkemBloku = stat.getBlockCountLong();
long volnychBloku = stat.getAvailableBlocksLong();
float ulozisteCelkem = (float)celkemBloku * velikostBloku / (1000*1000*1000);
float ulozisteVolne = (float)volnychBloku * velikostBloku / (1000*1000*1000);
```

Třetím způsobem je vytvoření virtuálního souboru v datovém úložišti a následné načtení celkového a volného místa pomocí metod třídy File:

```
File soubor = new File(getFilesDir().getAbsoluteFile().toString());
float ulozisteCelkem = (float)soubor.getTotalSpace() / (1000*1000*1000);
float ulozisteVolne = (float)soubor.getFreeSpace() / (1000*1000*1000);;
```

Ve všech třech případech platí, že celkové místo není totéž jako velikost úložiště v zařízení. To se totiž dělí na část vyhrazenou systému a část vyhrazenou pro data. Do systémové části nemají aplikace přístup, takže i výsledky metod pro zjištění kapacity úložiště jsou vztažené pouze k aplikační části. Neboli k celkové velikosti úložiště mínus velikosti vyhrazené pro systém.

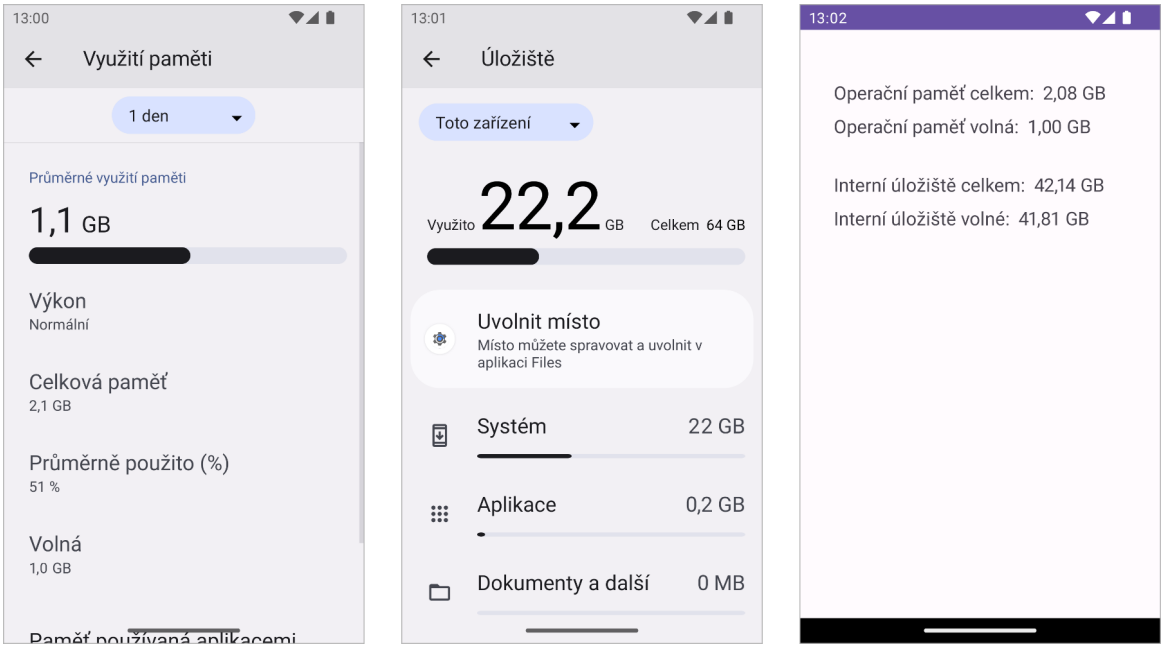
Zjištěné hodnoty opět naformátujeme a zobrazíme na displeji:

```
TextView txtUlozisteCelkem = (TextView) findViewById(R.id.txtUlozisteCelkem);
TextView txtUlozisteVolne = (TextView) findViewById(R.id.txtUlozisteVolne);

txtUlozisteCelkem.setText(String.format("%.02f GB", ulozisteCelkem));
txtUlozisteVolne.setText(String.format("%.02f GB", ulozisteVolne));
```

Pro ladění je výhodné využít namísto fyzického zařízení emulátor Android Studia. V jeho nastaveních lze totiž podle potřeby měnit jak velikost paměti, tak i úložiště.

Velikost úložiště v systému zjistíme zpravidla v nastaveních, pod položkou Úložiště.
Velikost paměti je v Nastavení → Systém → Rozšířená nastavení → Pro vývojáře → Paměť. Je ale nutné, mít aktivované vývojářské menu.
Pozor, toto se může u jednotlivých zařízení nebo verzí systému lišit.

Na obrázku 22.3 je v levém otisku displeje zaokrouhlená velikost operační paměti zařízení 2,1GB, z toho 1,1 GB nevyužitých. Na prostředním je celková kapacita úložiště 64 GB, ale 22 GB zabírá systém, čili pro aplikace zbývá 42 GB. Z toho je obsazeno jenom minimum, takže volné místo je také téměř 42 GB. Tomu všemu odpovídají údaje v aplikaci, na otisku vpravo.



Obrázek 22.3: Zobrazení velikosti paměti a prostoru v datovém úložišti

22.2 Interní databáze

Android zařízení potřebuje ke své práci velké množství dat, která systém používá a která si ukládá ve svých vnitřních strukturách. Některá z nich mohou aplikace využívat, nemají ale možnost přímého přístupu, musí využít API rozhraní. Jde například o záznamy z kalendáře, telefonní seznam, textové zprávy, seznam blokovaných čísel a podobně.

Pro získávání dat z vnitřního úložiště dat systému Android se používá třída ContentResolver. Ta slouží jako jakési rozhraní mezi aplikací a úložištěm. Na základě požadavku předaného pomocí dotazu (query) vrátí požadovaná data ve formě databázového kurzoru, neboli množiny záznamů, kterou je možné postupně procházet. Úložiště je ve své podstatě databázová tabulka, takže i přístup připomíná databázové dotazování.

Syntaxe dotazu vypadá takto:

```
ContentResolver contentResolver = getContentResolver();
cursor = contentResolver.query(
    OBSAH_URI,
    sloupce,
    podminka,
    podminkaHodnoty,
    razeni);
```

První parametr musí být uveden vždy. Určuje, do které tabulky v úložišti budeme takto přistupovat. Jde o jednotný identifikátor zdroje, neboli URI (Uniform Resource Identifier). Pro jednotlivé zdroje dat je nutné znát konkrétní Uri, ta vypadají například takto:

```
Uri KALENDARE = Uri.parse("content://calendar/calendars");
Uri UPOZORNENI = Uri.parse("content://calendar/calendar_alerts");
Uri UDALOSTI = Uri.parse("content://calendar/instances/when");
Uri KONTAKTY = Uri.parse("content://contacts/contacts");
Uri SMS_PRICHOZI = Uri.parse("content://sms/inbox");
Uri SMS_ODESLANE = Uri.parse("content://sms/sent");
Uri SMS_ROZEPSANE = Uri.parse("content://sms/draft");
```

Naštěstí jsou pro většinu zdrojů zavedeny konstanty, takže si není nutné Uri pamatovat. Například Uri pro seznam dostupných kalendářů můžeme získat i takto:

```
Uri KALENDARE = CalendarContract.Calendars.CONTENT_URI;
```

Druhým parametrem jsou sloupce, které má dotaz vracet. Tento parametr není povinný, pokud místo něj předáme hodnotu null, vrátí dotaz všechny sloupce. Obvykle to ale není úplně šťastné řešení, sloupců potřebujeme jenom několik a přenášet nadbytečná data zbytečně zatěžuje systém. Počet sloupců je přitom u většiny zdrojů až překvapivě vysoký.

Seznam požadovaných sloupců se předává v poli textových řetězců a i zde, aby si nebylo nutné pamatovat co ve kterém sloupci je, můžeme použít konstanty. Do pole si je uložíme například takto:

```
String[] SLOUPCE = new String[] {
        CalendarContract.Calendars._ID,
        CalendarContract.Calendars.IS_PRIMARY,
```

```
        CalendarContract.Calendars.CALENDAR_DISPLAY_NAME
};
```

V tomto případě jde o atributy dotazu na dostupné kalendáře. Dotaz s takto definovanými sloupci by vrátil v každém záznamu ID kalendáře, příznak, jestli jde o hlavní kalendář a jeho název.

Třetí a čtvrtý parametr definují podmínku, kterou musí záznam splňovat, aby byl zahrnut mezi výsledky. Jde o analogii k příkazu WHERE v jazyce SQL. Tyto parametry jsou opět nepovinné, první je řetězec obsahující text podmínky a druhý pole řetězců, které reprezentují hodnoty podmínky. Pokud budou oba parametry zastoupeny hodnotou null, tak se podmínka jednoduše nevyužije a vráceny budou všechny dostupné záznamy. Jak podmínku s hodnotou sestavit?

```
String PODMINKA = CalendarContract.Calendars.IS_PRIMARY + " = ?";
String[] PODMINKA_HODNOTY = new String[]{"0"};
```

První řádek říká, že nás budou zajímat pouze kalendáře s příznakem IS_PRIMARY, čili příznakem hlavního kalendáře a druhý řádek doplňuje hodnotu 0. Neboli záznam musí mít ve sloupci IS_PRIMARY hodnotu 0. Což znamená, že dotaz vrátí pouze záznamy pomocných kalendářů, hlavní vynechá. Podmínka může být složená, podobně jako v SQL, dílčí podmínky lze snadno spojit pomocí klauzule AND nebo OR. Pokud tím vznikne potřeba více hodnot, jednoduše se doplní do pole hodnot.

Posledním parametrem je způsob řazení získaných záznamů. I tento parametr je nepovinný a také zde můžeme využít konstant, abychom snadno nastavili dejme tomu řazení kalendářů podle libovolného sloupce. Například podle názvu (ten je zároveň přednastavený jako hlavní sloupec pro řazení):

```
String RAZENI = CalendarContract.Calendars.DEFAULT_SORT_ORDER;
```

Když toto vše spojíme do uceleného dotazu, získáme kód, který zajistí, že Content Resolver vrátí ID a název všech pomocných kalendářů, seřazený abecedně podle názvu. Dotaz bude vypadat takto:

```
Uri KALENDARE = CalendarContract.Calendars.CONTENT_URI;

String[] SLOUPCE = new String[] {
        CalendarContract.Calendars._ID,
        CalendarContract.Calendars.CALENDAR_DISPLAY_NAME
};

String PODMINKA = CalendarContract.Calendars.IS_PRIMARY + " = ?";
String[] PODMINKA_HODNOTY = new String[]{"0"};

String RAZENI = CalendarContract.Calendars.DEFAULT_SORT_ORDER;

ContentResolver contentResolver = getContentResolver();
Cursor cursor = contentResolver.query(
                  CalendarContract.Calendars.CONTENT_URI,
                  SLOUPCE,
                  PODMINKA,
                  PODMINKA_HODNOTY,
                  RAZENI);
```

Tímto způsobem získané záznamy můžeme v cyklu procházet a dál s nimi pracovat. V tomto případě je pouze vypíšeme do okna Console v Andoid Studiu:

```
if(cursor.getCount() > 0)
{
    while (cursor.moveToNext()) {
        System.out.println(String.format("Id: %s, název: %s",
                           cursor.getString(0), cursor.getString(1)));
    }
}
cursor.close();
```

Popsaným způsobem je možné data načíst. A toho se podržíme i v následujících příkladech. Načítání dat ale není jediným typem operací, které ContentResolver poskytuje. Analogickým způsobem je možné dosáhnout také založení záznamu, aktualizaci nebo jeho smazání.

22.2.1 Kalendář, upozornění

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 22_Systemova_data
projekt Kalendar

Jak už z předchozího textu vyplývá, Android umožňuje práci s více kalendáři. Za základní se zpravidla bere Google Kalendář, který má uživatel propojený se svým Google účtem. Jakmile má Android příležitost, neboli datové připojení, tak svůj vnitřní kalendář synchronizuje s Google Kalendářem. K tomuto kalendáři je možné připojit další, pomocné, které bude Android také synchronizovat.

Vytvoříme aplikaci, která načte seznam všech dostupných kalendářů a potom načte upozornění ze všech dostupných kalendářů. Nebudeme se příliš zabývat estetikou jejich zobrazení, ze systému získáme pouze data a ta vypíšeme do textového pole na displeji.

Co je potřeba udělat?

- propojit Android zařízení s kalendářem, obsahujícím upozornění
- vytvořit aplikaci s formulářem, který umožní zobrazit seznam kalendářů a upozornění
- ověřit oprávnění aplikace číst kalendář
- načíst a zobrazit seznam dostupných kalendářů
- načíst a zobrazit seznam upozornění z hlavního kalendáře

Propojení zařízení s kalendářem pravděpodobně nebude způsobovat žádné potíže. Většina zařízení je propojená už od registrace a kalendář automaticky synchronizuje. Pouze je potřeba ověřit, jestli jsou v kalendáři nějaké události s nastaveným upozorněním a pokud ne, tak jich několik vytvořit.

Založíme novou aplikaci a na hlavní formulář umístíme jedno malé a druhé větší TextView pro zobrazení výsledků. Můžeme přidat také tlačítka pro načtení kalendářů a upozornění.

Do manifestu je potřeba přidat žádost o čtení z kalendáře:

```
<uses-permission android:name="android.permission.READ_CALENDAR"/>
```

Ve třídě MainActivity si můžeme udělat jednoduchou kontrolu, jestli má uživatel přístup ke kalendáři skutečně povolený a pokud ne, tak o oprávnění požádat:

```
int OPRAVNENI_ZADOST = 1;
private void testOpravneni() {
    if (ContextCompat.checkSelfPermission(this,
                Manifest.permission.READ_CALENDAR) !=
                                    PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.READ_CALENDAR},
                OPRAVNENI_ZADOST);
    }
}
```

Pro načtení seznamu kalendářů využijeme třídu ContentResolver. Ta bude potřebovat v prvním parametru určit systémovou tabulku, tedy CalendarContract.Calendars.CONTENT_URI a ve druhém seznam sloupců systémové tabulky, které chceme načíst. Spokojíme se s ID a názvem kalendáře, pro výpis víc nepotřebujeme. Poslední tři parametry, tedy dva pro podmínku a jeden pro řazení zatím nevyužijeme, předáme pouze null.

```
public void nacistKalendare(View view) {

    // sloupce s daty, ktera nas zajimaji
    final String[] SLOUPCE = new String[] {
            CalendarContract.Calendars._ID,
            CalendarContract.Calendars.CALENDAR_DISPLAY_NAME
    };

    List<String> kalendare = new ArrayList<String>();

    // nacteni dat
    ContentResolver contentResolver = this.getContentResolver();
    Cursor cursor = contentResolver.query(
                        CalendarContract.Calendars.CONTENT_URI,
                        SLOUPCE,
                        null,
                        null,
                        null);

    // prochazeni dat a jejich ulozeni do seznamu
    if(cursor.getCount() > 0)
    {
        while (cursor.moveToNext()) {
            kalendare.add( String.format("%s: %s",
                            cursor.getString(0), cursor.getString(3)) );
        }
    }

    cursor.close();

    // vypis ziskanych dat
    String seznamKalendaru = "";
    for (String kalendar : kalendare) {
        seznamKalendaru = seznamKalendaru.concat(kalendar).concat("\n");
    }

    TextView txtKalendar = (TextView) findViewById(R.id.txtKalendare);
```

```
    txtKalendar.setText(seznamKalendaru);
}
```

Data vrácená ve formě kurzoru v cyklu projdeme a uložíme si je do seznamu kalendare. Potom kurzor zavřeme, abychom uvolnili systémové zdroje a obsah seznamu kalendare vypíšeme do TextView txtKalendar.

Mohli bychom samozřejmě plnit TextView už během procházení kurzoru a v tomto případě by to fungovalo. Pokud bychom ale chtěli s daty dál pracovat, například je propojit s RecyclerView, potřebovali bychom je mít uložené v nějaké vlastní struktuře, tedy například v seznamu řetězců ArrayList<String>.

Načtení seznamu upozornění se příliš neliší od načtení kalendářů. Opět využijeme volání třídy ContentResolver, v parametrech předáme URI upozornění a seznam vhodných sloupců. ContentResolver vrátí data opět ve formě kurzoru, ta v cyklu projdeme a uložíme si je do seznamu lstUpozorneni. Nakonec kurzor zavřeme a obsah seznamu lstUpozorneni vypíšeme do TextView txtUpozorneni.

```
public void nacistUpozorneni(View view) {
    final String[] SLOUPCE = new String[] {
            CalendarContract.CalendarAlerts.EVENT_ID,
            CalendarContract.CalendarAlerts.BEGIN,
            CalendarContract.CalendarAlerts.TITLE,
            CalendarContract.CalendarAlerts.ORGANIZER
    };


    ContentResolver contentResolver = this.getContentResolver();
    Cursor cursor = contentResolver.query(
                        CalendarContract.CalendarAlerts.CONTENT_URI,
                        SLOUPCE,
                        null,
                        null,
                        null);
    List<String> lstUpozorneni = new ArrayList<String>();

    // prochazeni dat, jejich vypis do console a ulozeni do seznamu
    while (cursor.moveToNext()) {
        long upozorneniID = cursor.getLong(0);
        long zacatek = cursor.getLong(1);
        String nazev = cursor.getString(2);
        String kalendar = cursor.getString(3);

        String calendarInfo = String.format(
                        "Upozor. ID: %s\nZacatek: %s\nNazev: %s\nKalendar: %s",
                        upozorneniID, zacatek, nazev, kalendar);
        System.out.println(calendarInfo);

        DateFormat df = new SimpleDateFormat("dd.MM. HH:mm");
        lstUpozorneni.add(String.format(
                        "[%s] %s %s",
                        upozorneniID, df.format(zacatek), nazev));
    }

    cursor.close();

    String seznamUpozorneni = "";
    for (String upozorneni : lstUpozorneni) {
```

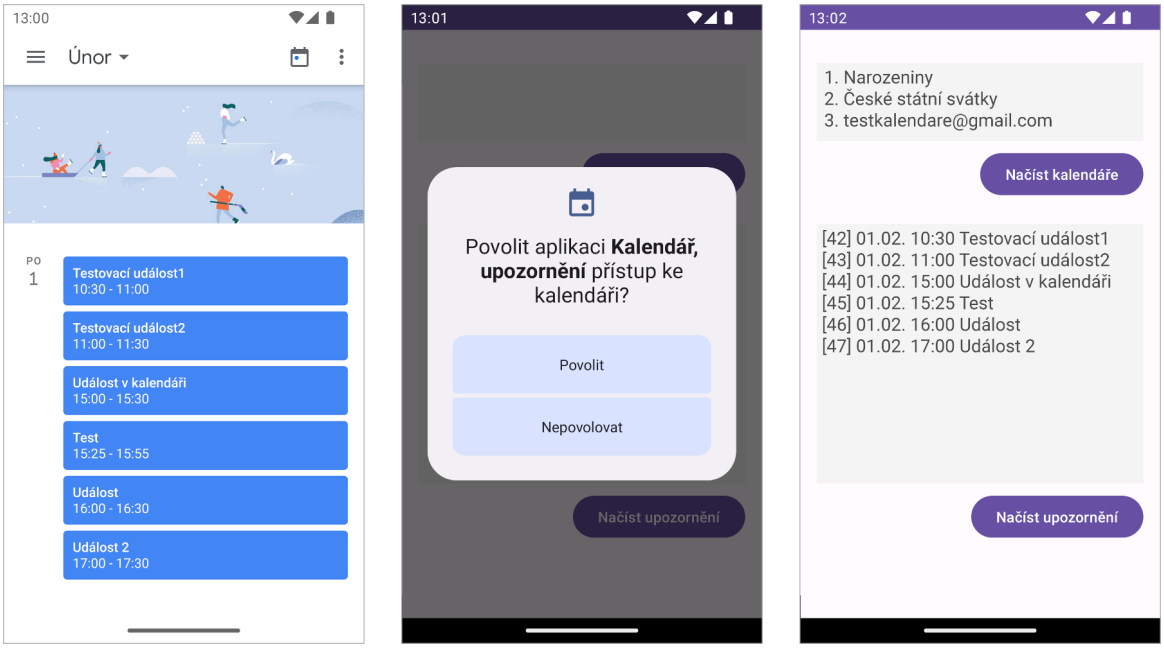
```
            seznamUpozorneni = seznamUpozorneni.concat(upozorneni).concat("\n");
        }

        TextView txtUpozorneni = (TextView) findViewById(R.id.txtUpozorneni);
        txtUpozorneni.setText(seznamUpozorneni);
    }
```

Podobným způsobem by bylo možné vypsat taky seznam všech událostí ve zvoleném kalendáři. K tomu je potřeba použít příslušný zdroj dat:

```
Uri UDALOSTI = Uri.parse("content://calendar/instances/when");
```

Zvláštností tohoto zdroje je, že musí být definováno omezení na datový rozsah, aby se ContentResolver nedostal do situace, kdy bude muset vrátit nepřiměřeně velké množství záznamů.

Aplikaci spustíme na emulátoru nebo v zařízení. Výsledek by měl vypadat jako na obrázku 22.4. Na prvním otisku displeje jsou události v Google kalendáři, uprostřed je žádost aplikace o povolení přístupu k datům, vpravo je seznam dostupných kalendářů a z nich načtená upozornění.



Obrázek 22.4: Vlevo kalendář, uprostřed žádost o přístup k datům a vpravo data načtená aplikací z kalendáře

22.2.2 Telefonní seznam, kontakty

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 22_Systemova_data
projekt TelefonniSeznam

Na rozdíl od kalendářů je seznam kontaktů v systému Android jednodušší. Existuje pouze jeden seznam, který je k dispozici hned po instalaci systému a není ho nutné se službami Google nijak synchronizovat. Mírnou komplikací je uložení kontaktů ve více datových zdrojích, resp. Tabulkách. V první je identifikace kontaktu, jméno, příjmení atd. a v ostatních samotný kontaktní údaj, například e-mailová adresa, telefonní číslo, webová stránka, poznámka a tak podobně. Každý typ kontaktu je uložený v jiné tabulce. Mezi

kontaktem a kontaktním údajem je vazba 1:n, neboli ke každému kontaktu může být přiřazeno několik kontaktních údajů.

Zkusíme načíst ze seznamu kontaktů všechny záznamy, které obsahuje a ty zobrazíme v textovém okně na displeji. Grafickou stránkou se ani teď nebudeme příliš zabývat, zajímají nás pouze data a ta si zpřístupníme v nejjednodušším textovém formátu.

Co je potřeba udělat?

- přidat do seznamu kontaktů několik záznamů
- vytvořit aplikaci s formulářem, který umožní zobrazit seznam kontaktů a kontaktních údajů
- ověřit oprávnění aplikace číst kontakty
- načíst a zobrazit seznam kontaktů

Přidání záznamů do telefonního seznamu nebude problémem. Při práci s emulátorem je seznam prázdný, ale stačí otevřít aplikaci Telefonní seznam a nějaké kontakty do něj doplnit. Na fyzickém zařízení pravděpodobně bude kontaktů dost z běžného používání.

Založíme novou aplikaci a na hlavní formulář umístíme TextView pro zobrazení výsledků. Můžeme přidat také tlačítko pro načtení kontaktů.

Do manifestu je potřeba přidat žádost o čtení kontaktů:

```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
```

I zde můžeme udělat jednoduchou kontrolu, jestli má uživatel přístup ke kontaktům skutečně povolený a pokud ne, tak o oprávnění požádat. Kód bude prakticky stejný jako v předchozím příkladu.

Pro načtení seznamu kontaktů opět využijeme třídu ContentResolver. Systémová tabulka, která potřebná data obsahuje, má URI v konstantě ContactsContract.Contacts.CONTENT_URI. Jakmile budou kontakty načteny, budeme je postupně procházet a ke každému budeme ve vnořeném cyklu dohledávat jednotlivé kontaktní údaje. Řekněme, že nás budou zajímat pouze telefonní čísla. Ta jsou uložena v datovém zdroji, který lze identifikovat pomocí další URI konstanty. Seznam základních konstant viz tabulka 22.1.

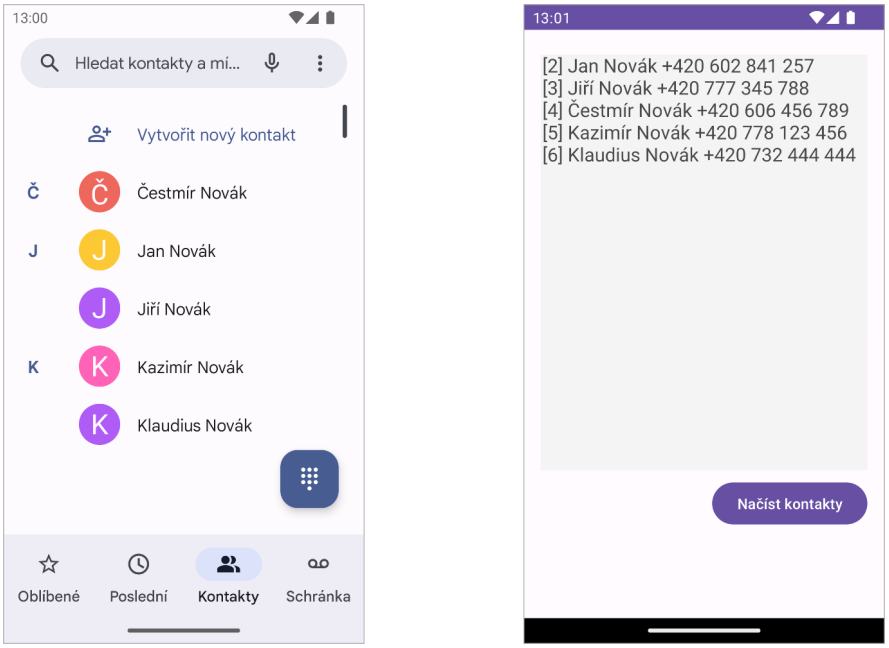
Typ kontaktu	URI konstanta
základní kontakt	ContactsContract.Contacts.CONTENT_URI
telefonní číslo	ContactsContract.CommonDataKinds.Phone.CONTENT_URI
e-mailová adresa	ContactsContract.CommonDataKinds.Email.CONTENT_URI
poštovní adresa	ContactsContract.CommonDataKinds.StructuredPostal.CONTENT_URI
webová stránka	ContactsContract.CommonDataKinds.Website.CONTENT_URI
fotografie	ContactsContract.CommonDataKinds.Photo.CONTENT_URI
poznámka	ContactsContract.CommonDataKinds.Note.CONTENT_URI

Tabulka 22.1: základní URI konstanty pro načítání kontaktů a kontaktních údajů

Pro kontakty i kontaktní údaje je vhodné definovat sloupce, které nás zajímají. V případě kontaktu je to pouze ID a jméno. U kontaktního údaje nás zajímá telefonní číslo. Potom pomocí třídy ContentResolver získáme všechny kontakty a ve vnořeném cyklu k těmto údajům dohledáme telefonní čísla.

```
public void nacistKontakty(View view) {
    final String[] SLOUPCE_KONTAKT = new String[] {
            ContactsContract.Contacts._ID,
            ContactsContract.Contacts.DISPLAY_NAME
    };

    final String[] SLOUPCE_TEL = new String[] {
            ContactsContract.CommonDataKinds.Phone.NUMBER
    };

    List<String> kontakty = new ArrayList<String>();

    ContentResolver cr = getContentResolver();
    Cursor cur1 = cr.query(
                 ContactsContract.Contacts.CONTENT_URI,
                 SLOUPCE_KONTAKT,
                 null,
                 null,
                 null);

    while (cur1.moveToNext()) {
        String id = cur1.getString(0);
        String jmeno = cur1.getString(1);
        System.out.println("Jmeno : " + jmeno + ", ID : " + id);

        // nacteni kontaktnich udaju (tel. cisel)
        Cursor cur2 = cr.query(
            ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
            SLOUPCE_TEL,
            ContactsContract.CommonDataKinds.Phone.CONTACT_ID +" = ?",
            new String[]{id}, null);

        // prochazeni nalezenych kontaktnich udaju (tel. cisel)
        while (cur2.moveToNext()) {
            String phone = cur2.getString(0);
            kontakty.add(String.format("[%s] %s %s", id, jmeno, phone));
        }
        cur2.close();
    }

    String sKontakty = "";
    for (String kontakt : kontakty) {
        sKontakty = sKontakty.concat(kontakt).concat("\n");
    }

    TextView txtKontakty = (TextView) findViewById(R.id.txtKontakty);
    txtKontakty.setText(sKontakty);

    cur1.close();
}
```

Aplikaci spustíme na emulátoru nebo v zařízení. Výsledek by měl vypadat jako na obrázku 22.5. Na prvním otisku displeje jsou kontakty v telefonním seznamu na druhém je načtený seznam kontaktů s telefonním číslem.



Obrázek 22.5: Vlevo kontakty v systému Android, vpravo tytéž kontakty, načtené aplikací

22.2.3 Textové zprávy

Co se týká struktury, je v případě textových zpráv situace jednodušší než v předchozích příkladech. Zprávy jsou uloženy v několika tabulkách, zvlášť příchozí, odeslané, rozepsané. Má to tu výhodu, že není potřeba dělat mezi zdroji vazbu, stačí použít ten, který nás aktuálně zajímá.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 22_Systemova_data
projekt TextoveZpravy

V příkladu načteme všechny příchozí zprávy, uložené v zařízení. Grafickou stránkou se opět nebudeme zabývat, zobrazíme pouze zprávy ve formě prostého textu, umístěného do textového pole na displeji.

Co je potřeba udělat?

- zajistit v zařízení nějaké přijaté zprávy
- vytvořit aplikaci s formulářem, který umožní zobrazit seznam zpráv
- ověřit oprávnění aplikace číst zprávy
- načíst a zobrazit seznam textových zpráv

Při spuštění aplikace na fyzickém zařízení, asi nebude se záznamy v tabulce příchozích zpráv problém. V emulátoru bude tato tabulka naopak pravděpodobně prázdná. Příchozí zprávy je ale možné v nastaveních emulátoru nasimulovat a tím tabulku naplnit alespoň několika pokusnými zprávami.

Založíme novou aplikaci a na hlavní formulář umístíme TextView pro zobrazení výsledků. Můžeme přidat také tlačítko pro načtení zpráv.

Do manifestu je potřeba přidat žádost o čtení textových zpráv:

```
<uses-permission android:name="android.permission.READ_SMS"/>
```

Může se hodit přidat kontrolu oprávnění pro přístup k textovým zprávám. Postup je obdobný jako v předchozích příkladech.

Záznamy budeme opět načítat pomocí třídy ContentResolver. V tomto případě ale narazíme na jednu komplikaci, pro systémové tabulky nejsou u textových zpráv k dispozici konstanty s potřebným URI. Musíme tedy URI napsat přesně tak, aby odpovídalo umístění zdrojových tabulek. Nás budou nyní zajímat pouze příchozí zprávy, ale později se mohou hodit i ostatní:

Uri SMS_PRICHOZI = Uri.parse("content://sms/inbox");

Uri SMS_ODESLANE = Uri.parse("content://sms/sent");

Uri SMS_ROZEPSANE = Uri.parse("content://sms/draft");

Stejná situace je s definicí sloupců. Ani tady nejsou konstanty k dispozici, je nutné znát jejich fyzické názvy. Naštěstí se dají snadno najít na internetu. My se nyní spokojíme se základními, id záznamu, telefonním čísle odesílatele a textem zprávy. Definice sloupců bude vypadat takto:

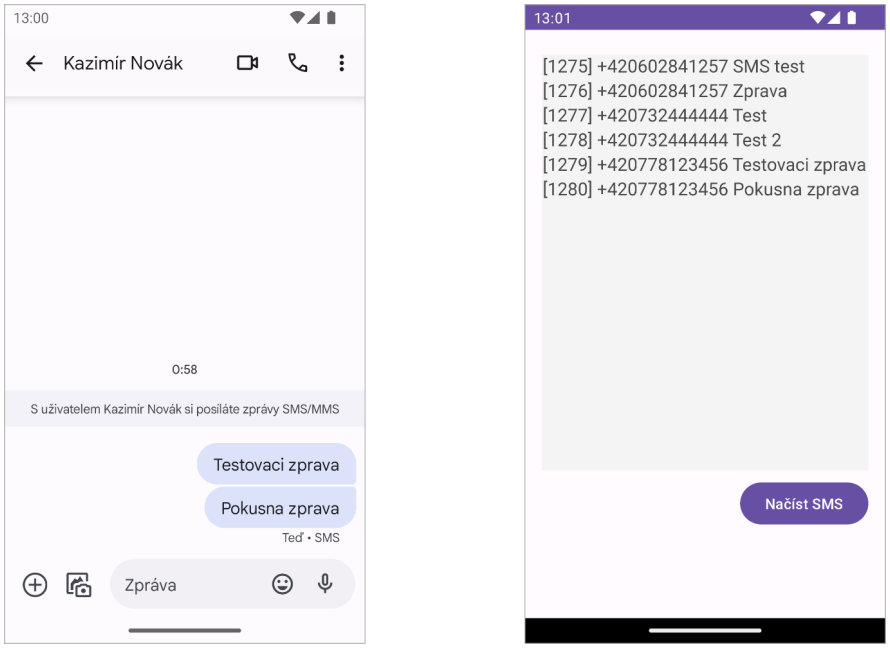
```
final String[] SLOUPCE = new String[] {
        "_id",
        "address",
        "body"
};
```

Další postup je stejný jako v předchozích příkladech. Třídě ContentResolver předáme URI zdroje, a seznam sloupců které nás zajímají. Ostatní parametry zatím necháme null. ContentResolver vrátí v kurzoru všechny záznamy z tabulky příchozích zpráv. Záznamy postupně projdeme, uložíme do seznamu a na závěr je zapíšeme do textového pole na displeji.

```
public void nacistSMS(View view) {

    final String[] SLOUPCE = new String[] {
            "_id",
            "address",
            "body"
    };

    ContentResolver cr = getContentResolver();
    Cursor cur = cr.query(
                        Uri.parse("content://sms/inbox"),
                        SLOUPCE,
                        null,
                        null,
```

```
                        null);

    List<String> zpravySMS = new ArrayList<String>();


    while (cur.moveToNext()) {
        zpravySMS.add( String.format("[%s] %s %s",
                                cur.getString(0),
                                cur.getString(1),
                                cur.getString(2) ));

    }

    cur.close();

    String sSMS = "";
    for (String zpravaSMS : zpravySMS) {
        sSMS = sSMS.concat(zpravaSMS).concat("\n");
    }

    TextView txtSMS = (TextView) findViewById(R.id.txtSMS);
    txtSMS.setText(sSMS);
}
```

Aplikaci spustíme na emulátoru nebo v zařízení. Výsledek by měl vypadat jako na obrázku 22.6. Na prvním otisku displeje jsou zprávy v systému Android, na druhém výpis všech záznamů, které jsou uložené v seznamu příchozích zpráv.



Obrázek 22.6: Vlevo textové zprávy v systému Android, vpravo zprávy načtené aplikací

22.3 Inzertní ID (Advertising ID)

Internetové prohlížeče používají k ukládání malých dat soubory cookies, které jim mimo jiné umožňují jednoznačně identifikovat uživatele a dodávat mu personalizovaný obsah. V Android

zařízení takové soubory neexistují, sdílení dat mezi aplikacemi je problematické a s odinstalováním aplikace se mažou i její dočasné soubory. Knihovny Google Play jsou však vybaveny jednoznačným identifikátorem, který podobnou personalizaci umožňuje. Je to sekvence typu GUID, tedy 128 bitový, neboli 32 znakový unikátní řetězec hexadecimálních číslic. Tento řetězec je vygenerovaný při instalaci Androidu na zařízení a teoreticky by měl být po celou dobu jeho života neměnný. Každý takto nainstalovaný systém má jiné ID a toto je přístupné z aplikací. Ty ho můžou předat třeba na server, na kterém udržují uživatelův profil. Toho se dá využít například při zobrazení reklam, kdy systém pozná, že se reklama zobrazí na zařízení uživatele, o kterém má už jakési informace – například jaké reklamy ho zajímají, nebo které už se mu zobrazily a on je nechal bez povšimnutí. Využít se dá pochopitelně, kromě reklamy, i mnoha jinými způsoby.

Kvůli ochraně soukromí to ale se zmíněnou neměnností ID není úplně stoprocentní. Uživatel má možnost svoje přidělené ID v systému resetovat, tedy vytvořit si nové a tím v podstatě donutit aplikaci, aby mu vytvořila nový profil. Nebo může předávání ID úplně zakázat a aplikace získá pouze sekvenci sestavenou ze samých nul. Ačkoliv toho uživatelé často nevyužívají, je potřeba s takovou eventualitou počítat.

Jednoduchým příkladem zkusíme načíst Inzertní ID zařízení, na kterém aplikace poběží. Grafická stránka nás ani tentokrát nebude příliš zajímat, zobrazíme pouze v textovém poli zjištěné ID nebo chybový text.

Co je potřeba udělat?

- zjistit jaké má zařízení přiděleno Inzertní ID

- vytvořit aplikaci s formulářem, který bude obsahovat tlačítko a textové pole s výsledkem

- načíst a zobrazit Inzertní ID

Zjistit Inzertní ID zařízení a případně ho přenastavit nedá nijak velkou práci. Stačí najít v Nastaveních část Ochrana soukromí, položku Reklamy a tam si ho prostě přečíst.

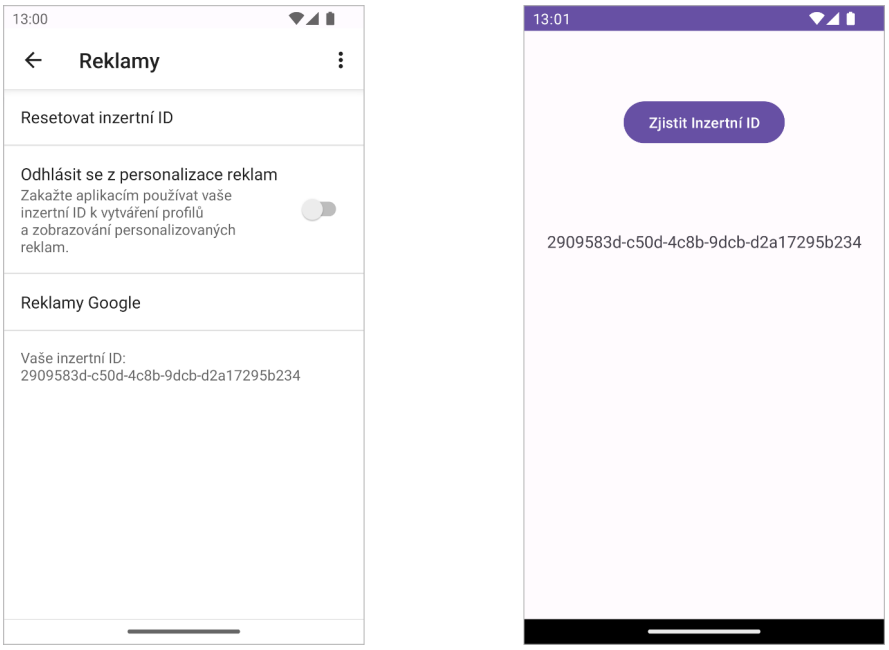
Založíme novou aplikaci, na hlavní formulář umístíme tlačítko a pole typu TextView, pro zobrazení výsledku, pojmenované například txtAdvertisingId. Tím je formulář připraven.

Ve třídě MainActivity vytvoříme metodu, která se o načtení ID postará. Zde je nutné poznamenat, že načítání musí proběhnout asynchronně. K tomu využijeme standardní Java knihovnu ExecutorService, která běh na pozadí zařídí. Voláním execute() se spustí metoda, kterou má v parametru:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
    @Override
    public void run() {
        TextView txtAdvertisingId = (TextView)
                  findViewById(R.id.txtAdvertisingId);
        try {
            AdvertisingIdClient.Info adInfo =
                  AdvertisingIdClient.getAdvertisingIdInfo(MainActivity.this);

            String appId = adInfo != null
                  ? adInfo.getId() : "Advertising ID nelze načíst.";

            txtAdvertisingId.setText(appId);
```

```
            } catch (Exception e) {
                txtAdvertisingId.setText("chyba");
            }
        }
});
```

Metoda getAdvertisingIdInfo() třídy AdvertisingIdClient, která je součástí Google Play Services, načte informace o Inzertním ID a jeho nastavení. Pokud není metoda k dispozici, znamená to, že zřejmě nejsou nainstalované Google Play Services a nezbývá, než je doplnit podle postupu popsaného v kapitole 11 Závislé zdroje – Dependencies. Samotné ID potom už stačí jenom získat pomocí metody getId() a výsledek zobrazit.



Obrázek 22.7: Vlevo Inzertní ID v Nastavení systému Android, vpravo načtené ID

23 Senzory a periférie

Android zařízení bývají zpravidla vybavena velkým množstvím senzorů, které umožňují sledovat fyzikální veličiny jak uvnitř zařízení, tak mimo něj. Detekovat lze teplotu, tlak, zrychlení, intenzitu osvětlení, vzdálenost zařízení od uživatele, krokoměr a mnoho dalších. Systém podporuje několik desítek různých typů senzorů, je však na výrobci, kterými zařízení vybaví. Většina senzorů je hardwarových, ale některé můžou být nahrazené programovým algoritmem. Například senzor pro detekci polohy (označený konstantou TYPE_ORIENTATION) vypočítává polohu zařízení čistě softwarově, jako vstup ale potřebuje hodnoty z fyzických senzorů přitažlivosti a magnetického pole (TYPE_GRAVITY, TYPE_MAGNETIC_FIELD).

Senzory pohybu	
TYPE_ACCELEROMETER	Akcelerometr, měří zrychlení ve třech osách se zohledněním gravitace, v m/s^2. Použití k detekci pohybu, třesení, naklánění.
TYPE_ACCELEROMETER_UNCALIBRATED	Jako TYPE_ACCELEROMETER, ale výsledné hodnoty vrací nekalibrované a kalibrované (celkem 6 hodnot).
TYPE_GRAVITY	Přitažlivost ve třech osách, v m/s^2. Použití k detekci pohybu, třesení, naklánění.
TYPE_GYROSCOPE	Měření rychlosti rotace ve třech osách, v radiánech za sekundu.
TYPE_GYROSCOPE_UNCALIBRATED	Jako TYPE_GYROSCOPE, ale výsledné hodnoty vrací nekalibrované a kalibrované (celkem 6 hodnot).
TYPE_LINEAR_ACCELERATION	Zrychlení ve třech osách se zohledněním gravitace. Použití k detekci pohybu, třesení, naklánění.
TYPE_ROTATION_VECTOR	Měření změn orientace zařízení.
Senzory polohy	
TYPE_MAGNETIC_FIELD	Intenzita okolního magnetického pole, ve třech osách.
TYPE_MAGNETIC_FIELD_UNCALIBRATED	Jako TYPE_MAGNETIC_FIELD, ale výsledné hodnoty vrací nekalibrované a kalibrované (celkem 6 hodnot).
TYPE_ORIENTATION	Úhel natočení zařízení ve třech osách. Slouží k detekci polohy zařízení.
TYPE_PROXIMITY	Pomocí infračerveného paprsku měří vzdálenost objektů od displeje. Používá se zejména pro zhasnutí displeje při telefonování.
Senzory okolního prostředí	
TYPE_AMBIENT_TEMPERATURE	Vnější teplota ve stupních Celsia.
TYPE_LIGHT	Intenzita osvětlení okolí v luxech.
TYPE_PRESSURE	Okolní tlak, v hektopascalech nebo milibarech.
TYPE_RELATIVE_HUMIDITY	Měření okolní vlhkosti, v procentech.
Tělesné senzory	
TYPE_HEART_BEAT	Detekce srdečního tepu (vyvolá událost při každém tepu)

Senzory pohybu	
TYPE_HEART_RATE	Frekvence srdečního tepu
TYPE_STEP_DETECTOR	Detekce kroků (vyvolá událost při kroku)
TYPE_STEP_COUNTER	Krokoměr

Tabulka 23.1: Konstanty, označující základní typy senzorů podporované Androidem

Práce s jednotlivými senzory je v zásadě dost podobná. Je nutné získat přístup ke službě senzor manageru, nastavit typ požadovaného senzoru, ověřit, jestli je daný senzor v zařízení k dispozici, vytvořit si posluchač (listener) čekající na změnu hodnoty senzoru a potom už jenom využívat data, která senzor průběžně poskytuje.

23.1 Senzor intenzity osvětlení

Uděláme si příklad detekce intenzity okolního osvětlení. Na formulář umístíme dvě textová pole, do nichž budeme vypisovat aktuální hodnoty zjištěné na senzoru TYPE_LIGHT. V MainActivity nejprve rozšíříme třídu o interface SensorEventListener:

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 23_Senzory_a_periferie
projekt Osvetleni

```
public class MainActivity extends AppCompatActivity
                         implements SensorEventListener {
```

Dále využijeme předefinování dvou událostí třídy Activity k iniciaci a ukončení čtení ze senzoru, onCreate() a onPause(). Bylo by sice možné použít pouze onCreate(), ale způsobili bychom, že by načítání ze senzorů probíhalo i když by aplikace nebyla aktivní. Zbytečně bychom tím zařízení zatěžovali. Nejprve tedy v onResume() získáme SensorManager a uložíme si ho do pomocné proměnné, protože pomocí něj budeme registrovat a odregistrovávat listener. Nastavíme typ senzoru na TYPE_LIGHT a otestujeme, jestli je senzor k dispozici. Pokud ano, listener na něj zaregistrujeme.

Na závěr této inicializační části ještě uložíme do pomocných proměnných ukazatele na textová pole do nichž budeme zapisovat výsledky a zavoláme přepisovanou událost onResume().

```
SensorManager sensorManager;
private TextView txtIntenzita, txtPresnost;

@Override
protected void onResume() {
    sensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
    Sensor senzorOsvetleni = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
    if (senzorOsvetleni == null) {
        Toast.makeText(getApplicationContext(), "Senzor osvětlení
                              není k dispozici.", Toast.LENGTH_LONG).show();
    }
    else {
        sensorManager.registerListener(this, senzorOsvetleni,
                              SensorManager.SENSOR_DELAY_NORMAL);
    }

    txtIntenzita = (TextView) this.findViewById(R.id.txtIntenzita);
```

```
    txtPresnost = (TextView) this.findViewById(R.id.txtPresnost);
    super.onResume();
}
```

Jakmile se tedy v životním cyklu aktivity dostaneme k její aktivaci, zavolá se událost onResume(), která iniciuje sensorManager, předá ho do proměnné a následně v dalších dvou proměnných předá textová pole pro zápis výsledků.

V okamžiku deaktivace aktivity bude zavolána událost onPause(). Toho využijeme k zastavení listeneru. Pouze ho ukončíme a zavoláme původní událost onPause().

```
@Override
protected void onPause() {
    sensorManager.unregisterListener(this);
    super.onPause();
}
```

V tuto chvíli už máme připravený listener a zbývá jenom implementace metod, které s vracenými hodnotami nějak naloží. Pro tento účel přepíšeme události onSensorChanged() a onAccuracyChanged(). První se bude volat při změně hodnoty intenzity osvětlení a druhá při změně přesnosti měření.

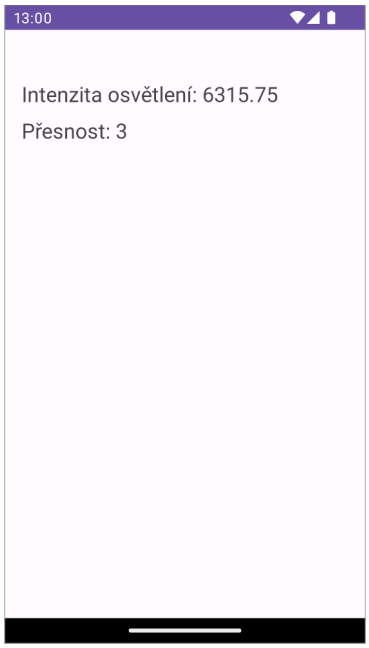
Přesnost měření je udávaná v rozmezí 0 – 3, kde 0 znamená nutnost kalibrace, 1 je nízká, 2 střední a 3 vysoká dosažitelná přesnost. Pokud nás přesnost nezajímá, můžeme ji ignorovat, nicméně i tak je nutné událost onAccuracyChanged() předefinovat, byť s prázdným tělem, abychom vyhověli interface SensorEventListener.

```
@Override
public void onSensorChanged(SensorEvent event) {
    if(event.sensor.getType() == Sensor.TYPE_LIGHT){
        txtIntenzita.setText("Intenzita osvětlení: " + event.values[0]);
    }
}

@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
    if(sensor.getType() == Sensor.TYPE_LIGHT){
        txtPresnost.setText("Přesnost: " + accuracy);
    }
}
```

Ověření typu senzoru v tomto případě není nutné, protože vše jde ze senzoru intenzity osvětlení, ale kdybychom četli z více senzorů najednou, tak se bude hodit vědět, který z nich hodnotu právě vrátil.

Úplně stejně bychom postupovali i při čtení z jiných senzorů, pouze bychom v SensorManageru nastavili příslušný typ senzoru a v události onSensorChanged() bychom v případě že by vracel více hodnot vypisovali každou z nich z pole event.values[]. Například TYPE_ACCELEROMETER_UNCALIBRATED vrací šest hodnot, takže bychom je získávali z polí event.values[0] až event.values[5].

Výsledná aplikace by měla vypadat přibližně jako na obrázku 23.1. Při změně polohy nebo zakrytí senzoru intenzity osvětlení by se hodnota na displeji měla měnit.



Obrázek 23.1: Aplikace zobrazující data ze senzoru osvětlení, spuštěná na emulátoru.

23.2 Určování polohy, GPS

Při vývoji aplikací se často setkáváme s potřebou geolokace, neboli zjištění místa, kde se uživatel právě nachází. Jelikož jsou polohové přijímače stále menší a levnější, dostávají se do stále většího počtu zařízení. Zároveň vznikají další polohové systémy podporované přijímači, takže se zvyšuje přesnost a dostupnost signálu nezbytného pro zjištění polohy uživatele.

Je třeba říct, že termín GPS (Global Positioning System) je názvem amerického polohového systému, provozovaného od roku 1994 americkým Ministerstvem obrany. Jeho fungování zajišťuje minimálně 24 (ve skutečnosti však spíše o desítku víc) družic obíhajících kolem Země ve výšce cca 20 tisíc kilometrů. Družice neustále vysílají údaje o přesném čase a svojí poloze. Zachycením těchto informací alespoň se čtyř družic současně je možné, pomocí matematických algoritmů, zjistit přesnou polohu přijímače. GPS umožňuje dosažení přesnosti v řádu centimetrů, ale pro civilní účely je přesnost omezena na několik metrů.

V době svého vzniku byl americký GPS unikátním projektem. A svoji výlučnost si udržel po mnoho následujících let. Až v roce 2011 dosáhl globálního pokrytí další systém – ruský Glonass. Následně vznikly další polohové systémy s globální ambicí, evropský Galileo a čínský Beidou, všechny založené na stejném principu jako GPS. Kromě toho vznikly regionální projekty – japonský QZSS a indický IRNSS, které operují s malým počtem družic a pokrývají pouze omezené území.

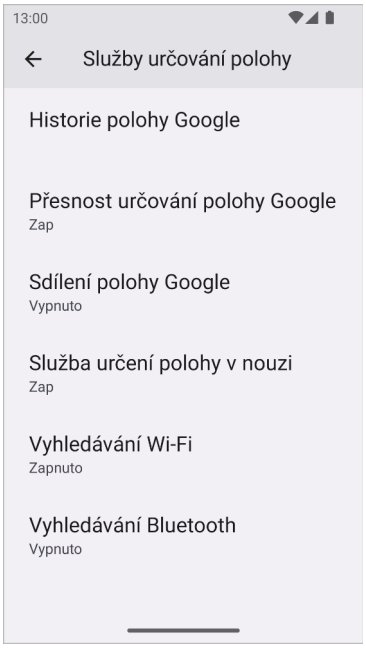
Díky historickému náskoku amerického systému se zkratka GPS zažila natolik, že se často používá jako obecný termín pro polohový systém. Uživatelé často ani necítí potřebu zjišťovat, který systém ve skutečnosti jejich přijímač využívá. A to obzvlášť tehdy, když přijímač podporuje více systémů a získaná data kombinuje s cílem získání vyšší přesnosti. Správně by se však měla používat obecnější zkratka GNSS neboli Global Navigation Satellite System, která zahrnuje všechny dostupné polohové systémy. S ní se setkáme u knihoven Androidu, kde v novějších verzích

nahrazuje původní zkratku GPS. U uživatelů však bude zřejmě dál vítězit již po několik desetiletí zažitá zkratka GPS.

Satelitní polohové systémy nejsou jediným způsobem, jak můžou zařízení zjistit svoji polohu. Podobné informace jako GNSS totiž poskytují také základnové stanice mobilních operátorů. Zařízení, které umožňuje komunikaci jejich prostřednictvím, typicky mobilní telefony, tak mají možnost stanovit polohu triangulací, podle signálu ze stanic BTS (Base Transceiver Station, základnová stanice pro rádiovou komunikaci s mobilními telefony). Přesnost je nižší než v případě GNSS, přibližně desítky metrů. Ale jelikož telefony komunikují s BTS už ze svého principu, není nutné kvůli určení polohy napájet další zařízení, takže je zjištění polohy tímto způsobem výhodnější z hlediska spotřeby energie. Na rozdíl od satelitních systémů tato metoda funguje i v uzavřených prostorách.

Dalším možným způsobem určení polohy je detekce sítí v okolí zařízení. K tomu se využívá zejména WiFi a případně i Bluetooth signál detekovaný zařízením. Data o zjištěných sítích se odešlou na servery Google (v případě Androidu) a ty vrátí přibližnou polohu zjištěnou pomocí triangulace. Přesnost je závislá na množství sítí v okolí. A ačkoliv lze v systému povolit vyhledávání sítí i při jejich vypnutí, tak fyzicky musí WiFi nebo Bluetooth zařízení fungovat, mít dostupný signál a umožněné datové přenosy.

Novější verze Androidu pracují jak s GNSS tak i s určováním polohy podle bezdrátových sítí a nechávají na uživateli, aby zvolil jedno, druhé nebo kombinaci obojího.



Obrázek 23.2: Nastavení způsobu určování polohy v Android zařízení.

Základní zjišťování polohy aplikací není příliš složité. S využitím knihoven Androidu stačí zaregistrovat listener, který bude volat událost při každé detekované změně polohy a zbytek můžeme nechat na zařízení. Nejen že si může zvolit Mnohem víc práce je s ošetřením nastavení uživatele. Nejen, že si může zvolit způsob určování polohy, ale může také zjišťování polohy úplně zakázat, nemusí být v dosahu signálu, ať už satelitního nebo mobilního a má samozřejmě možnost vypínat jednotlivé periferie potřebné k určení polohy. To všechno může nastat kdykoliv během práce s aplikací a vývojář s tím musí počítat.

23.2.1 Určování polohy, 1. příklad

Uděláme si jednoduchou aplikaci, která bude pracovat pouze se satelitním signálem a zjištěné údaje vypíše v jednoduché formě na displeji zařízení. Zajímat nás bude zeměpisná šířka, délka, výška nad mořem a tolerance měření.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 23_Senzory_a_periferie
projekt GPS1

Co je potřeba udělat?

- do manifestu přidat právo na zjišťování polohy zařízení
- vytvořit jednoduchý formulář, který zobrazí získané údaje
- předefinovat událost která bude volaná při změně polohy
- nastavit LocationManager, který se bude starat o volání události

Nejprve tedy přidáme do manifestu povolení pro zjišťování polohy zařízení:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

V activity_main.xml vytvoříme jednoduchý formulář, který bude obsahovat čtyři textová pole, pojmenovaná například txtSirka, txtDelka, txtVyska a txtPresnost.

V MainActivity.java rozšíříme třídu o interface LocationListener, který umožní práci s knihovnou pro určení polohy aniž bychom pro posluchač museli vytvářet samostatnou třídu:

```
public class MainActivity extends AppCompatActivity
                                  implements LocationListener {
```

Zjišťování polohy funguje asynchronně, na principu volání událostí. Jakmile dojde ke změně některého ze sledovaných parametrů (změna polohy zařízení, zapnutí/vypnutí lokační služby atd.), je zavolaná k tomu určená událost, v níž se nová situace programově vyhodnotí a zpracuje. Interface vyžaduje předefinování následujících metod:

onLocationChanged() – událost volaná při změně polohy zařízení

onProviderDisabled() – událost volaná při vypnutí detekce polohy uživatelem

onProviderEnabled() – událost volaná při zapnutí detekce polohy uživatelem

onStatusChanged() – událost volaná při změně stavu detekce polohy z vnějších příčin (například ztráta signálu)

Všechny tyto metody si vytvoříme a postupně do nich budeme doplňovat kód. Kromě těchto metod ale přepíšeme i další dvě události třídy MainActivity, onResume() a onPause(). Budeme je potřebovat proto, abychom mohli zapínat a vypínat zjišťování polohy podle stavů aktivity. Kdybychom polohu zjišťovali pořád, docházelo by zbytečně k úbytku systémových zdrojů a zejména k rychlejšímu vybíjení baterie. Nám stačí zobrazovat polohu pouze v té části životního

cyklu aktivity, kdy je formulář zobrazený uživateli. Po jeho skrytí polohu zjišťovat a vypisovat přestaneme.

Nejdřív tedy onResume(), neboli zobrazení formuláře. Je vhodné ověřit, že má uživatel povolené zobrazování polohy. Pokud ne, dáme mu to na vědomí toast zprávou. Je-li vše v pořádku, aktivujeme LocationManager:

```
locationManager.requestLocationUpdates(
                        LocationManager.GPS_PROVIDER, 5000, 5, this);
```

První parametr nastaví zjišťování polohy pomocí GPS přijímače. Alternativně bychom mohli použít konstantu NETWORK_PROVIDER, která by stanovila zjišťování polohy podle sítí. Druhý parametr říká, jak často bude poloha kontrolovaná. Hodnota je v milisekundách, takže v našem případě otestujeme polohu každých 5 vteřin. Třetí parametr je v metrech a stanovuje, jak velká vzdálenost se má považovat za změnu. Posledním z parametrů je umístění listeneru, v našem případě třída MainActivity.

Nastavení tedy říká, že LocationManager zkontroluje každých 5 vteřin polohu a pokud se liší alespoň o 5 metrů od minule zjištěné polohy, zavolá událost onLocationChanged(). Periodicitu i minimální vzdálenost lze libovolně měnit, je ale potřeba mít na paměti, že čím kratší interval a menší vzdálenost, tím větší je spotřeba energie a potažmo rychlejší vybíjení baterie.

Podobně jako jsme v onResume() zjišťování polohy zapnuli, tak ho v onPause() zase vypneme:

```
locationManager.removeUpdates(this);
```

Celý kód třídy MainActivity bude vypadat takto:

```
public class MainActivity extends AppCompatActivity implements LocationListener
{
    protected LocationManager locationManager;
    private TextView txtSirka, txtDelka, txtVyska, txtPresnost;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        txtSirka = (TextView)findViewById(R.id.txtSirka);
        txtDelka = (TextView)findViewById(R.id.txtDelka);
        txtVyska = (TextView)findViewById(R.id.txtVyska);
        txtPresnost = (TextView)findViewById(R.id.txtPresnost);
        locationManager = (LocationManager)
                            getSystemService(Context.LOCATION_SERVICE);
    }

    @Override
    protected void onResume() {
        int permissionCheck = ContextCompat.checkSelfPermission(this,
                            Manifest.permission.ACCESS_FINE_LOCATION);

        if (permissionCheck == PackageManager.PERMISSION_DENIED) {
            Toast.makeText(this, "Aplikace nemá povolené zjišťování polohy.",
                            Toast.LENGTH_SHORT).show();
        }
```

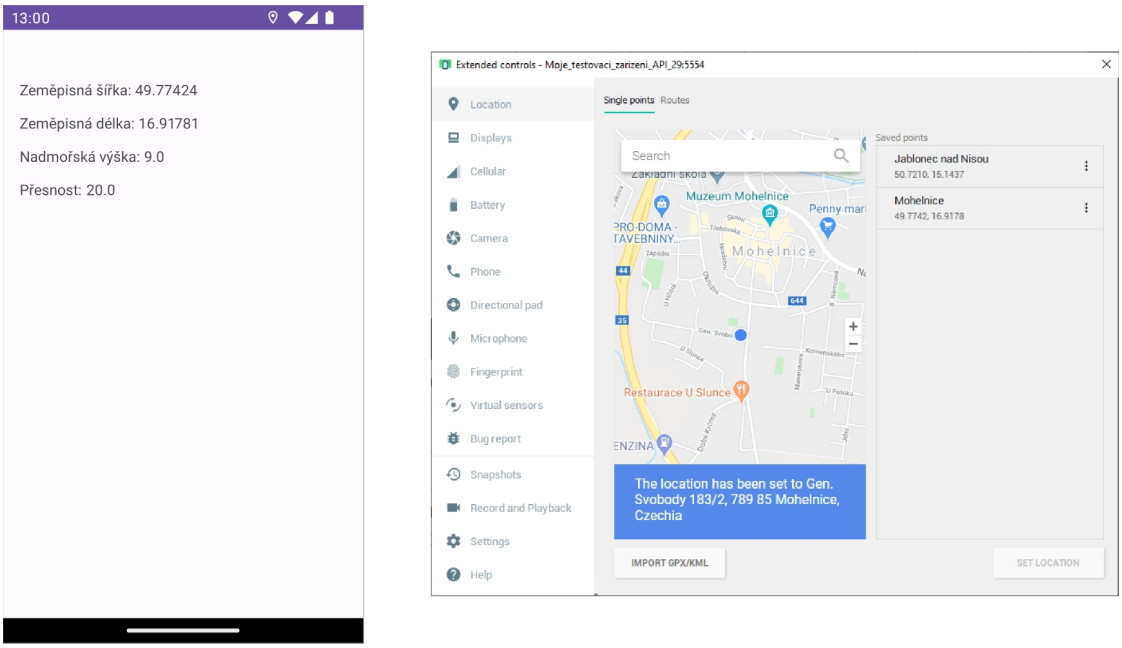
```
        else {
            locationManager.requestLocationUpdates(
                                LocationManager.GPS_PROVIDER, 2000, 1, this);
        }
        // zavolat puvodni onResume udalost
        super.onResume();
    }

    @Override
    protected void onPause() {
        // konec zjistovani polohy
        locationManager.removeUpdates(this);

        // zavolat puvodni onPause udalost
        super.onPause();
    }

    @Override
    public void onLocationChanged(Location location) {
                                    zobrazitSouradniceGps(location);
    }

    @Override
    public void onProviderDisabled(String provider) {
        Toast.makeText(getBaseContext(), "GPS je vypnuta!",
                                    Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onProviderEnabled(String provider) {
        Toast.makeText(getBaseContext(), "GPS je zapnuta.",
                                    Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onStatusChanged(String provider, int status, Bundle extras) {
    }

    // zobrazit ziskane souradnice
    private void zobrazitSouradniceGps (Location loc) {
        txtSirka.setText("Zeměpisná šířka: " + loc.getLatitude());
        txtDelka.setText("Zeměpisná délka: " + loc.getLongitude());
        txtVyska.setText("Nadmořská výška: " + loc.getAltitude());
        txtPresnost.setText("Přesnost: " + loc.getAccuracy());
    }
}
```

V událostech onProviderDisabled() a onProviderEnabled() pouze pomocí toast zprávy zobrazujeme informaci o zapnutí nebo vypnutí GPS. Pro výpis získaných hodnot máme pomocné proměnné a výsledky zapisujeme pomocí metody zobrazitSouradniceGps().

Událost onProviderDisabled() bychom mohli ještě rozšířit o otevření systémového formuláře nastavení zjišťování polohy, aby mohl uživatel tuto funkcionalitu rovnou zapnout:

```
Intent intent = new Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS);
startActivity(intent);
```

Výsledná aplikace by měla na emulátoru vypadat přibližně jako na obrázku 23.3.



Obrázek 23.3: Aplikace zobrazující data ze senzoru polohy, spuštěná na emulátoru. Vpravo je nastavení parametrů simulace senzoru polohy.

Při testování je potřeba dát pozor na to, aby měla aplikace povolené zjišťování polohy a měla zapnutou GPS. V emulátoru můžeme nastavit souřadnice a tím nasimulovat libovolnou polohu, ale na fyzickém zařízení musíme mít výhled na oblohu a počkat až přijímač zachytí satelitní signál z alespoň minimálního počtu družic.

23.2.2 Určování polohy, 2. příklad

Předchozí příklad byl vlastně tím nejjednodušším, co můžeme pro zjištění polohy udělat. Pokusíme se ho trochu vylepšit. Napíšeme aplikaci, která z něj vyjde, ale kromě základních informací ze satelitní navigace bude současně vypisovat souřadnice zjištěné podle okolních sítí. Umožní tak porovnat přesnost a rychlost reakce na změnu. Zároveň přidáme ještě funkcionalitu zjištění adresy.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 23_Senzory_a_periferie
projekt GPS2

Co je potřeba udělat?

- rozšířit formulář z předchozího příkladu o další pole
- přidat LocationManager pro zjišťování polohy podle sítí
- upravit metody událostí tak, aby poznaly zdroj dat o poloze
- nastavit LocationManager, který se bude starat o volání události

Do formuláře activity_main.xml přidáme stejná pole jaká máme pro GPS, ještě jednou pro navigaci podle sítí. A na konec ještě přidáme jedno velké textové pole pro adresu.

Tak jako jsme minule přidali LocationManager do události onResume(), tak tam přidáme další, ale tentokrát s konstantou NETWORK_PROVIDER:

```
locationManagerNet.requestLocationUpdates(
                        LocationManager.NETWORK_PROVIDER, 5000, 5, this);
```

Nesmíme zapomenout v události onPause() ani tento nový LocationManager vypnout, abychom zbytečně nezatěžovali zařízení pokud uživatel nemá aplikaci zobrazenou na displeji:

```
locationManagerNet.removeUpdates(this);
```

Dále nás čeká úprava metody zobrazitSouradniceGps(). Zobecníme ji, aby zobrazovala jak GPS tak NET souřadnice, takže ji můžeme přejmenovat na zobrazitSouradnice(). Parametr typu Location může zůstat, je pro oba případy společný. Uvnitř metody budeme potřebovat vědět, která pole vyplnit. Souřadnice dostaneme ve vstupním parametru. A dostaneme v něm zároveň i informaci o původu. Toho využijeme. Podmínka bude vypadat takto:

```
if (loc.getProvider().equals(LocationManager.GPS_PROVIDER)) {
```

Pokud tedy přijdou souřadnice ze satelitního přijímače, bude podmínka splněna a vyplníme pole s GPS souřadnicemi. Jestliže splněna nebude, jde o souřadnice, které byly prostřednictvím Location Manageru získány ze sítě, takže vyplníme pole určené pro souřadnice typu NETWORK. Metoda bude vypadat následovně:

```
private void zobrazitSouradnice (Location loc) {
    if (loc.getProvider().equals(LocationManager.GPS_PROVIDER)) {
        txtGpsSirka.setText("Zeměpisná šířka: " + loc.getLatitude());
        txtGpsDelka.setText("Zeměpisná délka: " + loc.getLongitude());
        txtGpsVyska.setText("Nadmořská výška: " + loc.getAltitude());
        txtGpsPresnost.setText("Přesnost: " + loc.getAccuracy());
    }
    else {
        txtSitSirka.setText("Zeměpisná šířka: " + loc.getLatitude());
        txtSitDelka.setText("Zeměpisná délka: " + loc.getLongitude());
        txtSitVyska.setText("Nadmořská výška: " + loc.getAltitude());
        txtSitPresnost.setText("Přesnost: " + loc.getAccuracy());
    }
}
```

Tím bychom mohli být hotovi. Ale připravili jsme si ještě jedno pole – pro adresu. Android zpřístupnil knihovnu Geocoder, která dokáže podle adresy stanovit souřadnice (takzvaný geocoding) a naopak adresu podle souřadnic (reverzní geocoding). A toho by byla škoda nevyužít. Takzvaný Geocoder potřebuje komunikovat se servery Google, takže datové přenosy musí být zapnuté. Komunikace je v tomto případě ale hodně úsporná.

Jak to funguje? Založíme si instanci třídy GeoCoder:

```
Geocoder gc = new Geocoder(context, Locale.getDefault());
```

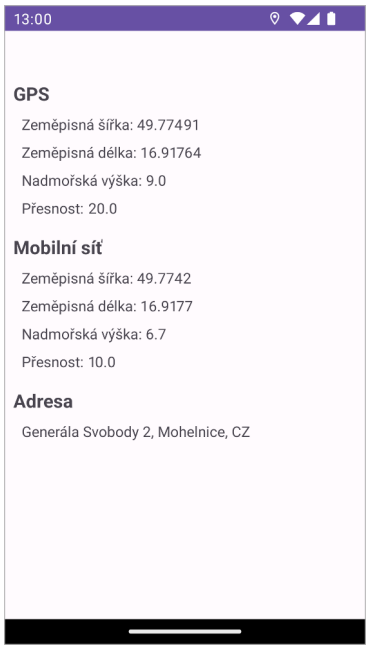
A metodou getFromLocation() vyvoláme reverzní geocoding. Jako první dva parametry zadáme souřadnice. Třetím parametrem je maximální počet adres, které má metoda vrátit. To má význam v případě, kdy je na daných souřadnicích více objektů. Proto je také nutné výstup uložit do seznamu typu List<Address>. Situaci si ale můžeme zjednodušit tak, že třetí parametr bude konstanta 1, v takovém případě ze seznamu následně budeme vybírat pouze první položku:

```
List<Address> adresy = myLocation.getFromLocation(
                            loc.getLatitude(), loc.getLongitude(), 1);
```

Potom už stačí jenom ověřit, jestli seznam adresu obsahuje a pokud ano, zobrazíme ji. Jelikož je celá operace poměrně citlivá a snadno může dojít k výjimce, musí být obalena pomocí try-catch. Větev catch přitom nemusí obsahovat žádný kód. Pokud se adresu z nějakého důvodu nepodaří zjistit, prostě ji jenom nezobrazíme. Na konec metody zobrazitSouradnice() tedy přidáme tento kód:

```
try {
    Geocoder myLocation = new Geocoder(
                        getApplicationContext(), Locale.getDefault());

    List<Address> adresy = myLocation.getFromLocation(
                        loc.getLatitude(), loc.getLongitude(), 1);

    if (adresy.size() > 0) {
        TextView txtMisto = (TextView) findViewById(R.id.txtMisto);
        Address adresa = adresy.get(0);
        String misto = adresa.getThoroughfare() + " "
                        + adresa.getFeatureName() + ", "
                        + adresa.getLocality() + ", "
                        + adresa.getCountryCode();
        txtMisto.setText(misto);
    }
}
catch (IOException e){
}
```

Výsledná aplikace by měla vypadat přibližně jako na obrázku 23.4. Musíme si opět dát pozor, abychom měli zapnuté zjišťování polohy, v nastaveních pro satelitní i síťové zjišťování polohy, aby aplikace měla povolenou komunikaci a lokalizaci a abychom měli viditelnost na oblohu. Potom by se měla vyplnit všechna pole formuláře.



Obrázek 23.4: Aplikace zobrazující data ze senzoru polohy, spuštěná na emulátoru.

23.3 Čtečka otisku prstu

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 23_Senzory_a_periferie
projekt OtiskyPrstu

Některá Android zařízení jsou vybavena malou zajímavou periférií, senzorem pro načtení otisku prstu. Tato komponenta může mít formu malé lesklé plošky, může být zabudována v tlačítku, může se nacházet jak na přední, tak na zadní straně zařízení, na jeho boční straně nebo může být dokonce ukryta pod displejem. A ačkoliv se vyskytuje téměř výhradně v telefonech a to ještě ne příliš často, tak Android práci s ní podporuje jak na úrovni zabezpečení přístupu do zařízení, tak i prostřednictvím API.

Obecně se dá říct, že ochrana tímto způsobem není příliš spolehlivá a doporučuje se ji používat pouze tam, kde není nutné silné zabezpečení. Na druhou stranu umožňuje relativně jednoduchý a rychlý přístup, bez nutnosti zapamatování kódu nebo hesla.

Pro vývojáře aplikací má ještě jednu zásadní nevýhodu – čtečku je možné využít pouze tehdy, pokud je jejím prostřednictvím nastaveno zabezpečení přístupu k zařízení jako celku. V opačném případě nemá operační systém k dispozici vzory otisků prstu a nemá tedy jak vyhodnotit oprávněnost přístupu. Vzory otisků se totiž s vypnutým zabezpečením vymažou.

Ideální podoba senzoru se stále hledá a její budoucnost je tedy poměrně nejistá. Výrobci mohou od jejího používání kvůli malé spolehlivosti a z důvodu úspory místa úplně ustoupit, stejně jako se čtečka může rozšířit, zvýšit svoji spolehlivost a oblibu u uživatelů. Je možné, že ji Android zpřístupní aplikacím bez nutné vazby na nastavení zabezpečení celého zařízení. Nebo také ne. Ukáže až budoucnost. V každém případě nic nebrání použití takového zabezpečení ve vlastních aplikacích jako doplňkovou alternativu například k zadávání hesla nebo PINu tam, kde si lze dovolit preferovat rychlost před spolehlivostí.

Připravíme si příklad ověření uživatele pomocí otisku prstu.

Co je potřeba udělat?

- udělat jednoduchý formulář s tlačítky pro otestování čtečky a pro pro spuštění načítání otisku
- zpřístupnit knihovnu pro práci s biometrickými údaji
- napsat rutinu pro test stavu čtečky
- napsat rutinu pro ověření otisku

Formulář bude úplně jednoduchý, bude obsahovat dvě tlačítka a jedno malé pole pro zobrazení výsledku ověření otisku. První tlačítko bude volat metodu testCtecky(), která ověří, jestli je čtečka v zařízení obsažena a připravena k použití. Druhé tlačítko zavolá metodu overeniOtisku(), která provede vše potřebné pro ověření uživatele.

Ve třídě MainActivity.java tedy připravíme metodu testCtecky() se dvěma testy. První ověří, zdali zařízení obsahuje hardware pro čtení otisků. Druhý test zjistí, jestli je v systému nastavené odemykání a zamykání pomocí otisku prstu v systému, neboli jestli jsou definované vzory otisků pro ověřování. Pro oba testy využijeme metodu canAuthenticate() třídy BiometricManager.

Metoda canAuthenticate() ověří, jestli je možné použít biometrické zabezpečení na úrovni, dané konstantou ve vstupním parametru. Jsou možné tři konstanty, určující, jestli je vyžadováno silné, slabé nebo nebiometrické zabezpečení, viz tabulka 23.2. Pro účely snímače otisku prstu si vystačíme s ověřením možnosti slabého biometrického zabezpečení.

Konstanta	Význam
BIOMETRIC_STRONG	Silné zabezpečení. Biometrické senzory splňující definici zabezpečení stupně 3 (Class 3). Tj. obvykle čtečky duhovky, obličeje nebo otisku prstu. Zařazení senzorů do třídy 3 je definované normou Android CDD.
BIOMETRIC_WEAK	Slabé zabezpečení. Biometrické senzory splňující definici zabezpečení stupně 2 (Class 2). Tj. obvykle čtečky duhovky, obličeje nebo otisku prstu. Splňují ho automaticky i všechny prostředky stupně 3. Zařazení senzorů do třídy 2 je definované normou Android CDD.
DEVICE_CREDENTIAL	Nebiometrické zabezpečení pomocí hesla, PINu, nebo gesta.

Tabulka 23.2: Úrovně zabezpečení

Na výstupu vrací opět konstantu, která informuje o stavu senzoru, viz tabulka 23.3. Podle této konstanty se rozhodneme, jestli uživateli ověření umožníme nebo jestli mu pouze sdělíme, že se tímto způsobem ověřit nelze.

Konstanta	Význam
BIOMETRIC_SUCCESS	Senzor biometrického ověření je připraven.
BIOMETRIC_ERROR_UNSUPPORTED	Senzor aktuálně není k dispozici.
BIOMETRIC_ERROR_HW_UNAVAILABLE	Hardwarová chyba senzoru.
BIOMETRIC_ERROR_NONE_ENROLLED	Ověření pomocí senzoru nelze použít, protože není nakonfigurované v nastavení zařízení.
BIOMETRIC_ERROR_NO_HARDWARE	Zařízení není senzorem vybavené.

Tabulka 23.3: Stav senzoru

Třída BiometricManager je součástí knihovny androidx.biometric, kterou je nutné k projektu připojit. Je třeba v okně projektu otevřít GradleScripts → build.gradle (Module app), v něm najít část dependencies a přidat do ní řádek

```
implementation 'androidx.biometric:biometric:1.2.0'
```

Pokud existuje novější verze, Android Studio na to upozorní. Gradle by měl po editaci souboru nabídnout synchronizaci, potvrdíme mu ji. Potom je možné v projektu začít používat třídu BiometricManager a její metody.

```
public void testCtecky (View view) {

    BiometricManager biometricManager=BiometricManager.from(this);

        switch (biometricManager.canAuthenticate(
                            BiometricManager.Authenticators.BIOMETRIC_WEAK)){

        case BiometricManager.BIOMETRIC_SUCCESS:
            Toast.makeText(this, "Čtečka otisku prstu je připravena.",
                            Toast.LENGTH_LONG).show();
            break;
```

```
        case BiometricManager.BIOMETRIC_ERROR_NO_HARDWARE:
            Toast.makeText(this, "Zařízení neobsahuje čtečku otisku prstu.",
                           Toast.LENGTH_LONG).show();
            break;

        case BiometricManager.BIOMETRIC_ERROR_NONE_ENROLLED:
            Toast.makeText(this, "Zamykání není v Nastaveních aktivované.",
                           Toast.LENGTH_LONG).show();
            break;

        default:
            Toast.makeText(this, "Jiná chyba.",
                           Toast.LENGTH_LONG).show();
    }
}
```

Samotné ověření je možné naprogramovat s použitím třídy BiometricManager (od API verze 29), nebo tříd FingerprintManager a KeyguardManager (pro API 23 až 27). Dá se ale také výhodně použít třída BiometricPrompt, dostupná od API 28, která zpřístupní systémové rozhraní ve formě uživatelského dialogu, pro zadání a ověření otisku. Tato třída je pro účely ověření otisku vůbec nejjednodušším řešením z hlediska programátorských nastavení. Toho využijeme.

Připravíme lokální proměnné, kterým v konstruktoru přiřadíme referenci na textové pole s výsledkem ověření, s referencí na instanci třídy BiometricPrompt a s definicí obsahu dialogu ve struktuře PromptInfo.

```
private TextView txtVysledek;
private BiometricPrompt biometricPrompt;
private BiometricPrompt.PromptInfo promptInfo;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    txtVysledek = findViewById(R.id.txtVysledek);
    Executor executor = ContextCompat.getMainExecutor(this);

    biometricPrompt = new BiometricPrompt(this, executor,
                      new BiometricPrompt.AuthenticationCallback(){
        @Override
        public void onAuthenticationError(int errorCode, CharSequence errString)
        {
            super.onAuthenticationError(errorCode, errString);
            txtVysledek.setText("Nepodařilo se ověřit");
        }

        @Override
        public void onAuthenticationSucceeded(
                       BiometricPrompt.AuthenticationResult result) {
            super.onAuthenticationSucceeded(result);
            txtVysledek.setText("Ověřeno");
        }

        @Override
        public void onAuthenticationFailed() {
            super.onAuthenticationFailed();
            txtVysledek.setText("Chyba – neověřeno");
```

```
        }
    });

    promptInfo = new BiometricPrompt.PromptInfo.Builder()
            .setTitle("Otisk prstu")
            .setNegativeButtonText("Zrušit")
            .build();
}
```

Třídě BiometricPrompt je při vytváření nové instance nutné, pro účely zpětného volání, předat v parametru Executor, neboli objektu, který spravuje asynchronně běžící úlohy. Jelikož nepotřebujeme předávat zpracování samostatnému paralelnímu vláknu, použijeme ukazatel na hlavní vlákno aplikace. Ten se získá voláním metody getMainExecutor().

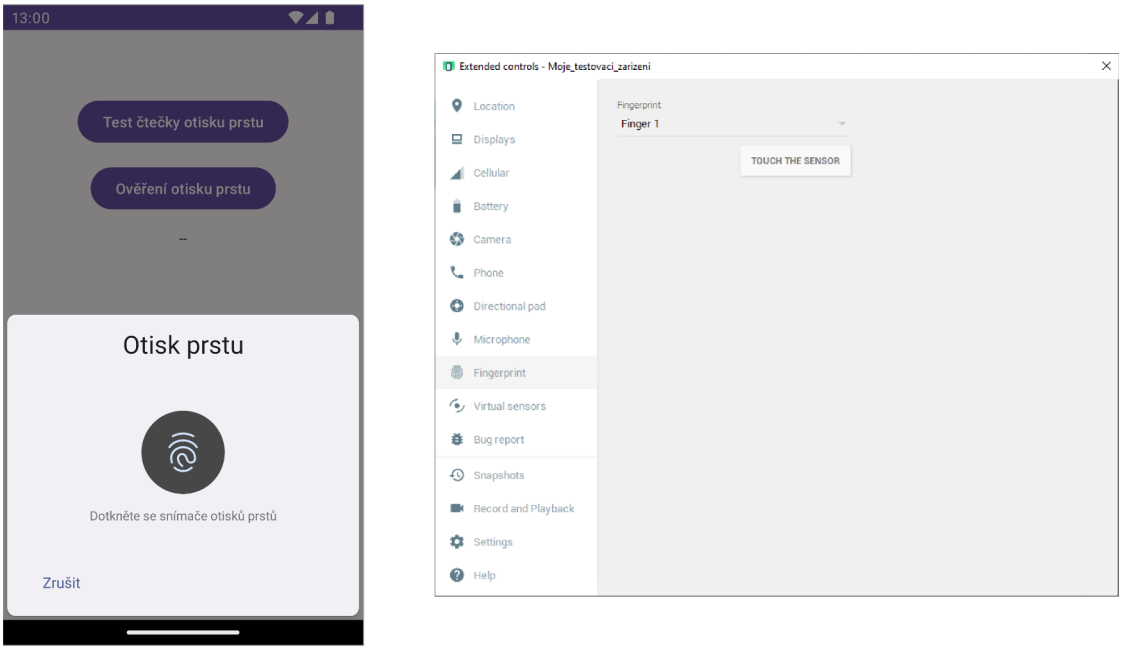
Dalším parametrem je AuthenticationCallback, objekt, který zpracovává události ověření otisku prstu. V něm je potřeba přepsat události pro chybu, pro úspěšné a pro neúspěšné ověření.

Instanci BiometricPrompt je potřeba doplnit základní parametry, které sice nejsou povinné, ale přinejmenším titulek dialogu je vhodné nastavit. Instanci sestavíme a přiřadíme do proměnné.

Pro zahájení procesu ověřování bude sloužit metoda overeniOtisku(), kterou voláme druhým tlačítkem na formuláři:

```
public void overeniOtisku(View view) {
    biometricPrompt.authenticate(promptInfo);
}
```

Aplikaci lze komfortně ladit v emulátoru. K dispozici je deset předdefinovaných otisků prstů:



Obrázek 23.5: Aplikace ověřující uživatele pomocí čtečky otisku prstu, spuštěná na emulátoru

23.4 Blesk fotoaparátu

Od API verze 21 je možné ovládat LED blesk samostatně, nezávisle na fotoaparátu. Od té doby ho aplikace využívají pro různé další účely, zejména pro notifikace nebo jako příruční svítilnu.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 23_Senzory_a_periferie
projekt Blesk

Jelikož je ale blesk, hardwarově i logicky, stále součástí fotoaparátu, stará se o jeho funkci třída CameraManager. Pro základní práci s bleskem si vystačíme s jedinou metodou – setTorchMode(). Stačí, když ji v parametru předáme ID fotoaparátu (to pro případ, že by bylo v zařízení fotoaparátů s bleskem více) a příznak, jestli blesk rozsvítit nebo zhasnout. To je vše. O zbytek se postará sám CameraManager.

```
CameraManager cameraManager;
cameraManager.setTorchMode(String idFotoaparatu, Boolean priznakRozsviceni);
```

Drobnou komplikací je určení id fotoaparátu u kterého je nainstalovaný blesk. K tomu můžeme přistoupit několika způsoby. Buď si řekneme, že jde o nejčastější případ, tj. že blesk má hlavní fotoaparát, který má standardně id = 0. pro účely experimentování si s tím snadno vystačíme:

```
idFotoaparatu = "0";
```

Nebo na věc můžeme jít sofistikovaněji a najít id hlavního fotoaparátu takto:

```
try {
    idFotoaparatu = cameraManager.getCameraIdList()[0];
} catch (CameraAccessException e) {
    Toast.makeText(getApplicationContext(), "Chyba: " + e.getMessage(),
                                                Toast.LENGTH_LONG).show();
}
```

Toto řešení je vhodnější, protože eliminuje teoretickou možnost, že id hlavního fotoaparátu nebude „0“, ale bude mít vyšší hodnotu. Není to příliš pravděpodobné, ale pokud bychom na takový případ narazili, asi by nás pěkně potrápil.

Celá záležitost se ale dá vyřešit i opravdu robustně tak, že napíšeme metodu, která projde všechny připojené fotoaparáty, u každého ověří jestli má blesk a vrátí opravdu správné id, nebo případně vypíše chybu:

```
private String idFotoaparatuSBleskem () {
    String cameraId = "";
    // projit postupne vsechny fotoaparaty dostupne v zarizeni..
    try {
        String[] fotoaparaty = cameraManager.getCameraIdList();
        if (fotoaparaty.length == 0) {
            Toast.makeText(getApplicationContext(),
                  "Zařízení neobsahuje fotoaparát.", Toast.LENGTH_LONG).show();
            return "";
        }

        for (String camId : fotoaparaty) {
            CameraCharacteristics characteristics =
                  cameraManager.getCameraCharacteristics(camId);
```

```
            // zjistit, jestli ma blesk
            boolean maBlesk = characteristics.get(
                  CameraCharacteristics.FLASH_INFO_AVAILABLE);
            // pokud ma blesk, zapamatovat si jeho id
            if (maBlesk)
                cameraId = camId;
        }
    }
    catch (CameraAccessException ex) {
        Toast.makeText(getApplicationContext(), "Chyba: " + ex.getMessage(),
                  Toast.LENGTH_LONG).show();
    }
    if (cameraId.isEmpty()) {
        Toast.makeText(getApplicationContext(), "Zařízení neobsahuje blesk.",
                  Toast.LENGTH_LONG).show();
    }
    return cameraId;
}
```

Návratovou hodnotou je id fotoaparátu s bleskem, nebo prázdný řetězec, pokud blesk nebyl nalezen.

Můžeme si zkusit udělat jednoduchou aplikaci, která pomocí uvedené metody najde v zařízení fotoaparát s bleskem a nechá uživatele, aby ho zapínal a vypínal.

Co je potřeba udělat?

- vytvořit formulář s tlačítky Zapnout a Vypnout
- přidat do manifestu oprávnění pro práci s kamerou a bleskem
- vložit do třídy activity_main metodu pro nalezení id fotoaparátu
- upravit ve třídě activity_main událost onCreate() a napsat události pro vypnutí a zapnutí blesku

Základ projektu vytvoříme snadno a rychle známým způsobem. Do activity_main v designéru vložíme dvě tlačítka a nastavíme jim na onClick() volání metod rozsvitit_click() a zhasnout_click().

Do manifestu přidáme žádost o oprávnění pro práci s kamerou a bleskem a zároveň oznámíme, pomocí atributu uses-feature, použití těchto dvou hardwarových prostředků.

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.FLASHLIGHT" />
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.flash" />
```

Způsoby nalezení id kamery jsme si popsali. Použijeme jeden z nich, ideálně metodu idFotoaparatuSBleskem(), kterou vložíme do třídy MainActivity. V této třídě ještě upravíme událost onCreate(). Rozšíříme ji o detekci blesku a vyplníme lokální proměnné cameraManager a cameraId:

```
private CameraManager cameraManager;
private String cameraId;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    setContentView(R.layout.activity_main);

    // test, jestli zarizeni obsahuje blesk
    Boolean jeBleskPodporovan = getApplicationContext().getPackageManager()
                        .hasSystemFeature(PackageManager.FEATURE_CAMERA_FLASH);
    if (!jeBleskPodporovan) {
        Toast.makeText(getApplicationContext(), "Zařízení nepodporuje blesk.",
                        Toast.LENGTH_LONG).show();
    }

    cameraManager = (CameraManager) getSystemService(Context.CAMERA_SERVICE);
    cameraId = idFotoaparatuSBleskem();
}
```

Ve formuláři jsou tlačítka, pro která musíme připravit události. První bude pro zapnutí blesku:

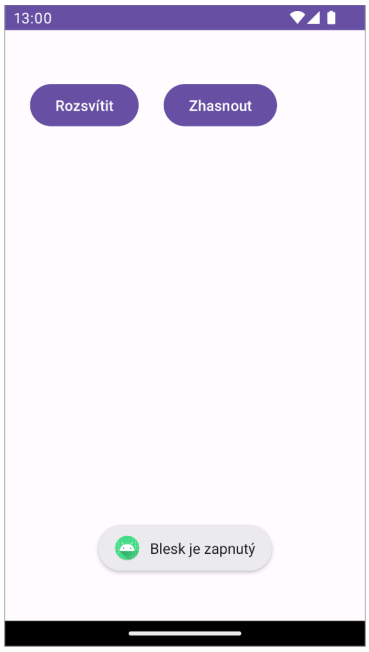
```
public void rozsvitit_click (View view) {
    try {
        cameraManager.setTorchMode(cameraId, true);
        Toast.makeText(getApplicationContext(), "Blesk je zapnutý",
                        Toast.LENGTH_SHORT).show();
    } catch (Exception e) {
        Toast.makeText(getApplicationContext(), "Chyba: " + e.getMessage(),
                        Toast.LENGTH_LONG).show();
    }
}
```

Událost pomocí metody setTorchMode() zapne blesk fotoaparátu s id, které je uchované v proměnné cameraId a zobrazí textovou informaci o provedené akci. Celá operace je pro jistotu uzavřena v bloku try-catch, pro případ že by se něco nepodařilo.

Metoda pro vypnutí blesku je téměř identická, rozdíl je pouze ve druhém parametru metody setTorchMode() a v textu informace o provedené akci.

```
public void zhasnout_click (View view) {
    try {
        cameraManager.setTorchMode(cameraId, false);
        Toast.makeText(getApplicationContext(), "Blesk je vypnutý",
                        Toast.LENGTH_SHORT).show();
    } catch (Exception e) {
        Toast.makeText(getApplicationContext(), "Chyba: " + e.getMessage(),
                        Toast.LENGTH_LONG).show();
    }
}
```

Po kompilaci a spuštění by se na displeji měl zobrazit formulář hlavní aktivity. Pokud zařízení neobsahuje fotoaparát s bleskem, objeví se uživateli chybová hlášení. Je-li blesk přítomen, může ho uživatel pomocí tlačítek zapínat a vypínat.



Obrázek 23.6: Aplikace ovládání blesku fotoaparátu, spuštěná na emulátoru

23.5 Bluetooth

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 23_Senzory_a_periferie
projekt Bluetooth

Bluetooth je technologie pro bezdrátové předávání dat mezi zařízeními na krátkou vzdálenost. Krátkou vzdáleností se myslí jednotky, nebo ve zvláštních případech desítky metrů. Využití je poměrně široké, nejčastěji se asi setkáme s případy, kdy pomocí Bluetooth komunikuje telefon se sluchátky nebo s chytrými hodinkami. Bluetooth modul ale můžeme najít i v autech, televizích, v počítačích, tabletech, kamerách, přehrávačích, tiskárnách a v moderní elektronice vůbec. Objevuje se všude tam, kde je potřeba snadno a rychle předávat data bez nutnosti zapojování kabelu.

Každý Bluetooth modul má svoji vlastní unikátní adresu skládající se ze šesti hexadecimálních číslic. Krom toho má přiřazený textový, neunikátní název, který je možné libovolně měnit. Každé Bluetooth zařízení může vyhledávat zařízení ve svém okolí a získávat informace o jejich adrese a textovém názvu.

Komunikace mezi dvěma zařízeními je zabezpečena pomocí šifrování. Proto je nutné zařízení, která spolu mají komunikovat, nejdřív spárovat. To je proces, při kterém si zařízení společně vygenerují šifrovací klíč a ten při další komunikaci oboustranně využívají pro ověření, že to druhé zařízení je oprávněno k předávání dat. Smazání šifrovacího klíče na kterémkoliv zařízení způsobí, že je spárování zrušeno.

Bluetooth má také možnost zůstat sice zapnutý, ale neviditelný pro další nespárovaná zařízení. Android této vlastnosti využívá a z bezpečnostních důvodů umožňuje dočasné ‚zviditelnění‘ zařízení, zpravidla na dvě minuty. Během této doby je nutné provést spárování s jiným zařízením.

Komunikační protokoly Bluetooth nejsou úplně jednoduché, uděláme si tedy alespoň příklad aplikace, která umožní základní práci s Bluetooth – zapnutí, vypnutí, zobrazení stavu, aktivaci viditelnosti, zobrazení seznamu spárovaných zařízení a vyhledání dalších zařízení v okolí.

Co je potřeba udělat?

- vytvořit formulář
- přidat oprávnění do manifestu
- napsat metody pro zapnutí, vypnutí, viditelnost, test a vyhledávání

Založíme novou aplikaci a vytvoříme nový formulář, který bude obsahovat tlačítka pro zapnutí, vypnutí, test, vyhledávání a dvě textová pole pro zobrazení stavu a výpis okolních zařízení.

Upravíme manifest tak, že do něj přidáme sadu následujících oprávnění:

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

<uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
<uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
```

První tři oprávnění jsou potřebná kvůli kompatibilitě se staršími API, do verze 30. Oprávnění BLUETOOTH je základní a umožňuje obecně přístup k modulu Bluetooth. BLUETOOTH_ADMIN je nezbytné pro vyhledávání zařízení a párování. ACCESS_COARSE_LOCATION je nutný pro vyhledávání – tím totiž zároveň, alespoň teoreticky, umožňujeme lokalizaci místa, kde se zařízení nachází. Pozor, při ladění aplikace musíme toto třetí oprávnění také uživatelsky potvrdit v nastavení telefonu. Jinak zařízení nebude schopné vyhledávat a nedá to nijak na vědomí, pouze nic v okolí nenajde.

Druhá trojice oprávnění je využita pro zařízení s API verze 31 a vyššími. Oprávnění BLUETOOTH_SCAN je nezbytné pro vyhledávání zařízení v okolí. BLUETOOTH_ADVERTISE je potřeba pokud má být zařízení viditelné pro ostatní zařízení v okolí. Poslední oprávnění, BLUETOOTH_CONNECT, slouží pro povolení komunikace se spárovanými zařízeními.

Připravíme si základní metody, které budou volané událostmi po stisknutí tlačítek. Pro práci s Bluetooth modulem se používá Android třída BluetoothAdapter, která obsahuje vše potřebné. V MainActivity.java tedy nejprve získáme ukazatel na Bluetooth adaptér pomocí metody getDefaultAdapter():

```
private BluetoothAdapter btAdapter = BluetoothAdapter.getDefaultAdapter();
```

Od tohoto okamžiku můžeme dělat elementární test – jestli zařízení obsahuje Bluetooth modul. Pokud je v proměnné btAdapter hodnota null, tak modul není v zařízení k dispozici. S tím se setkáme zejména u emulátoru v Android Studiu, který modul neemuluje, takže jsme odkázáni na ladění na reálném zařízení, nebo případně na emulátorech od jiných výrobců.

Hned v první metodě, zapnoutBluetooth(), využijeme ověření zdali je modul nainstalován. Potom pomocí metody adaptéru isEnabled() ověříme, jestli je zapnutý a pokud ne, voláním typu Intent Bluetooth spustíme. Android se pravděpodobně ještě uživatele zeptá jestli je aplikace

oprávněna něco takového udělat a pokud jí to povolíme, tak modul skutečně zapne, což okamžitě uvidíme na indikátoru na stavovém řádku.

```
public void zapnoutBluetooth (View view) {
    // test jestli ma zarizeni modul Bluetooth
    if (btAdapter == null) {
        Toast.makeText(this, "Zařízení nemá k dispozici Bluetooth.",
                                                Toast.LENGTH_SHORT).show();
        return;
    }
    // test, jestli je Bluetooth zapnuty
    if (!btAdapter.isEnabled()) {
        // zapnout Bluetooth
        Intent zapnoutBtIntent = new
                            Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
        startActivity(zapnoutBtIntent);
    }
}
```

Ještě jednodušší je vypnutí Bluetooth. Stačí zavolat metodu adaptéru disable():

```
btAdapter.disable();
```

Rovněž pomocí intentu vyřešíme zviditelnění zařízení. Prostřednictvím číselného parametru EXTRA_DISCOVERABLE_DURATION předáme čas po který má být zařízení viditelné. Jednotkou jsou sekundy, defaultní hodnota je 120 a maximální 300.

```
public void udelatViditelne (View view) {
    // zapnout viditelnost zarizeni na dve minuty
    Intent viditelneBtIntent = new
                        Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
    viditelneBtIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION,
                        120);
    startActivity(viditelneBtIntent);
}
```

Připravíme metodu pro otestování stavu Bluetooth modulu. Bude obsahovat řadu jednoduchých testů, které řeknou, jaký je aktuální stav modulu a nakonec ještě vypíše seznam spárovaných zařízení.

Prvním testem bude zjištění, jestli je modul v zařízení dostupný. To už víme že je snadné, pouze otestujeme hodnotu v proměnné btAdapter a výsledek zapíšeme do připraveného textového pole na formuláři:

```
if (btAdapter == null) {
    txtTestBt.setText("Zařízení neobsahuje Bluetooth modul");
    return;
}
else {
    txtTestBt.setText("Bluetooth modul OK");
}
```

Dalším testem bude stav Bluetooth modulu. Metoda adaptéru getState() vrátí hodnotu kterou porovnáme s konstantami stavu:

STATE_ON – Bluetooth je zapnutý

STATE_TURNING_ON – zapíná se

STATE_TURNING_OFF – vypíná se

STATE_OFF – je vypnutý

```
if (btAdapter.getState() == BluetoothAdapter.STATE_ON) {
    txtTestBt.setText(txtTestBt.getText() + "\n" + "Bluetooth je zapnutý.");
}
else if
...
```

Metodou adaptéru getScanMode() ověříme, jestli je zařízení pro ostatní viditelné:

```
if(btAdapter.getScanMode()==BluetoothAdapter.SCAN_MODE_CONNECTABLE_DISCOVERABLE)
    txtTestBt.setText(txtTestBt.getText() + "\n"
                                    + "Zařízení je pro okolí viditelné.");
else
    txtTestBt.setText(txtTestBt.getText() + "\n"
                                    + "Zařízení není pro okolí viditelné.");
```

Dále nás bude zajímat, jestli modul zrovna nevyhledává zařízení v okolí. K tomu poslouží metoda adaptéru isDiscovering():

```
if (btAdapter.isDiscovering())
    txtTestBt.setText(txtTestBt.getText() + "\n"
                                    + "Probíhá vyhledávání okolních zařízení.");
```

Informace o stavu modulu doplníme seznamem spárovaných zařízení. Ten je umístěný v zařízení a my ho získáme pomocí metody adaptéru getBondedDevices(). Bude stačit, když v cyklu od každého zařízení načteme název a jeho adresu. Ty potom zobrazíme v textovém poli na formuláři.

```
Set<BluetoothDevice> pairedDevices = btAdapter.getBondedDevices();
if (pairedDevices.size() == 0) {
    txtTestBt.setText(txtTestBt.getText() + " žádné zařízení");
}
else {
    for (BluetoothDevice device : pairedDevices) {
        String zarizeniJmeno = device.getName();
        String zarizeniAdresa = device.getAddress();
        txtTestBt.setText(txtTestBt.getText() + "\n" + "  "
                            + zarizeniJmeno + "[" + zarizeniAdresa + "]");
    }
}
```

Nejošidnější operací bude průzkum okolí a vypsání seznamu zařízení, které se nachází v dosahu našeho Bluetooth modulu. K tomu účelu si vytvoříme instanci třídy BroadcastReceiver, kterou si zaregistrujeme tak, aby byla zavolaná v případě, že dojde k očekávané události, kterou je v tomto případě nalezení zařízení. V instanci přepíšeme událost onReceive() tak, že nejdřív podmínkou ověříme zda jde o volání ACTION_FOUND, neboli že bylo nalezeno zařízení. Potom z intentu získáme informace o nalezeném zařízení a ty zobrazíme v textovém poli na formuláři.

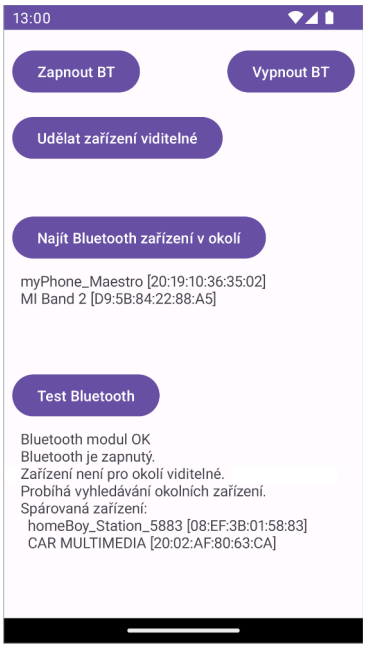
Receiver se musí zaregistrovat v metodě hledatBluetooth() jako intent s filtrem na vyhledávaná Bluetooth zařízení. Potom už zbývá pouze zavolat metodu adapteru startDiscovery() a čekat, až bude receiver zavolaný.

```
public void hledatBluetooth (View view) {
    // registrace udalosti nalezeni zarizeni v okoli
    IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
    registerReceiver(receiver, filter);

    // zahajit vyhledavani zarizeni
    btAdapter.startDiscovery();
}

// vytvorit instanci BroadcastReceiver pro hledani BT zarizeni
private final BroadcastReceiver receiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (BluetoothDevice.ACTION_FOUND.equals(action)) {
            BluetoothDevice device =
                        intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            String zarizeniJmeno = device.getName();
            String zarizeniAdresa = device.getAddress();

            TextView txtOkolniBt=(TextView)findViewById(R.id.txtOkolniBt);
            txtOkolniBt.setText(txtOkolniBt.getText() + zarizeniJmeno
                                        + " [" + zarizeniAdresa + "]\n");
        }
    }
};
```

Receiver zbytečně spotřebovává systémové zdroje, takže je potřeba ještě na závěr, například v události onDestroy() zavolat metodu k odregistrování receiveru. Jelikož neexistuje žádný vhodný způsob jak zjistit, jestli je receiver aktuálně zaregistrovaný, musíme buď udržovat příznak ve kterém si budeme evidovat každé zaregistrování a odregistrování, nebo můžeme použít pohodlnější metodu, odregistrování zavolat v bloku try, který v případě chyby prostě necháme jít dál:

```
try {
    unregisterReceiver(receiver);
}
catch (IllegalArgumentException ex) {
}
```

Výsledná aplikace by, po spuštění a zapnuti Bluetooth, měla vypadat přibližně jako na obrázku 23.7.



Obrázek 23.7: Aplikace s nastavením Bluetooth, spuštěná na emulátoru

24 Intent

Při vytváření aplikací nezřídka dochází k případům, kdy vývojář potřebuje zavolat ze své aplikace aplikaci jinou a v té spustit nějakou akci. Například když je potřeba pořídit fotografii, není nutné programovat fotoaparát, stačí zavolat ten, který už v systému je, předat mu požadavek a případně také parametry a potom už jenom počkat, až Android aplikace pro pořizování fotek vrátí výsledek.

Jde o asynchronní operaci mezi aplikacemi, kterou má na starost třída nazývaná Intent. Umožňuje vyvolání funkcionality sdílené mezi aplikacemi, předání parametrů a vrácení výsledku ve formě zavolání události, kterou si vývojář při volání zaregistroval.

Kromě interakce mezi aplikacemi se Intent dokáže postarat o totéž i mezi aktivitami uvnitř aplikace. Takže například při přechodu z jedné aktivity na druhou můžeme využít jeho služeb a vcelku jednoduše ho nechat, aby se o spuštění druhé aktivity sám postaral.

Intentu vždy říkáme, jakou akci má provést. Můžeme (a nemusíme) mu předat data, nad kterými akci provede. A nakonec stanovíme, jak má akci provést. Používají se dva základní způsoby provedení resp. volání - explicitní a implicitní. Explicitní způsob určuje, která komponenta a jakým způsobem má být použita. Implicitní pouze řekne, jakou akci chce provést a o způsobu už rozhodne operační systém.

24.1 Explicitní intent

Explicitní intent je ten, u nějž programátor stanoví, která komponenta operačního systému akci vykoná a jakým způsobem. Nejčastějším využitím tohoto typu intentu je přechod mezi aktivitami uvnitř aplikace, neboli z uživatelského pohledu přechod mezi formuláři.

```
Intent intent = new Intent(this, Activity2.class);
startActivity(intent);
```

Tato sekvence vytvoří novou instanci třídy Intent a spustí aktivitu Activity2. Jde o typický postup používaný při přechodu z jedné aktivity na jinou. Podobně bychom mohli spustit například službu:

```
Intent intent = new Intent(this, Sluzba.class);
startService(intent);
```

Zde pouze vezměme jako fakt, že lze tímto způsobem vyvolat různé typy akcí. Konkrétně ke službám se dostaneme později, zatím zůstaňme u aktivit.

24.1.1 Explicitní intent, příklad

Napíšeme aplikaci, která bude obsahovat jednu hlavní aktivitu (neboli uživatelsky hlavní formulář) a dvě vedlejší. Zkusíme z hlavního formuláře přejít kliknutím tlačítka na druhý formulář. Následně příklad ještě vylepšíme a z hlavního přejdeme na další formulář, kterému předáme parametry. Jejich hodnotu pouze zobrazíme.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 24_Intent
projekt ExplicitniIntent

Co je potřeba udělat?

- vytvořit pomocnou aktivitu, na kterou se z té hlavní budeme přepínat
- na hlavní aktivitu přidat dvě tlačítka, která budou volat pomocné aktivity
- z hlavní aktivity prostřednictvím intentu zavolat a zobrazit aktivitu pomocnou
- vytvořit další pomocnou aktivitu, která bude umět zobrazit hodnoty parametrů
- z hlavní aktivity volat druhou pomocnou aktivitu a předávat jí parametry

Založíme pomocnou aktivitu. Můžeme ji nazvat třeba Activity2. Nemusí obsahovat nic, kromě tlačítka ‚zpět'. To bude volat metodu zpet():

```
public void zpet (View view) {
    finish();
}
```

Volání finish() způsobí ukončení aktivity a návrat na předchozí, tedy na hlavní aktivitu.

Do formuláře hlavní aktivity přidáme tlačítko, které bude volat metodu intent1(). Tato metoda vytvoří novou instanci třídy Intent a spustí aktivitu Activity2:

```
public void intent1 (View view) {
    Intent intent = new Intent(this, Activity2.class);
    startActivity(intent);
}
```

Po spuštění na emulátoru máme aplikaci, v níž můžeme přecházet mezi dvěma aktivitami tam a zpět. Při vyvolání aktivit nezřídka potřebujeme volanému objektu současně předat parametry. Pro tyto účely má Intent implementovanou metodu putExtra():

```
Intent intent = new Intent(this, Activity2.class);
intent.putExtra("param1", "test");
intent.putExtra("param2", 10);
startActivity(intent);
```

Tím spolu se spuštěním aktivity Activity2 odešleme jeden textový a jeden číselný parametr. Volaná aktivita si může tyto parametry načíst:

```
Intent mujIntent = getIntent();
String parametr1 = mujIntent.getStringExtra("param1");
int parametr2 = mujIntent.getIntExtra("param2", 0);
```

Metoda getIntent() vrátí ukazatel na poslední vytvořený intent a potom už jen stačí pomocí příslušných metod načíst parametry jednotlivých datových typů.

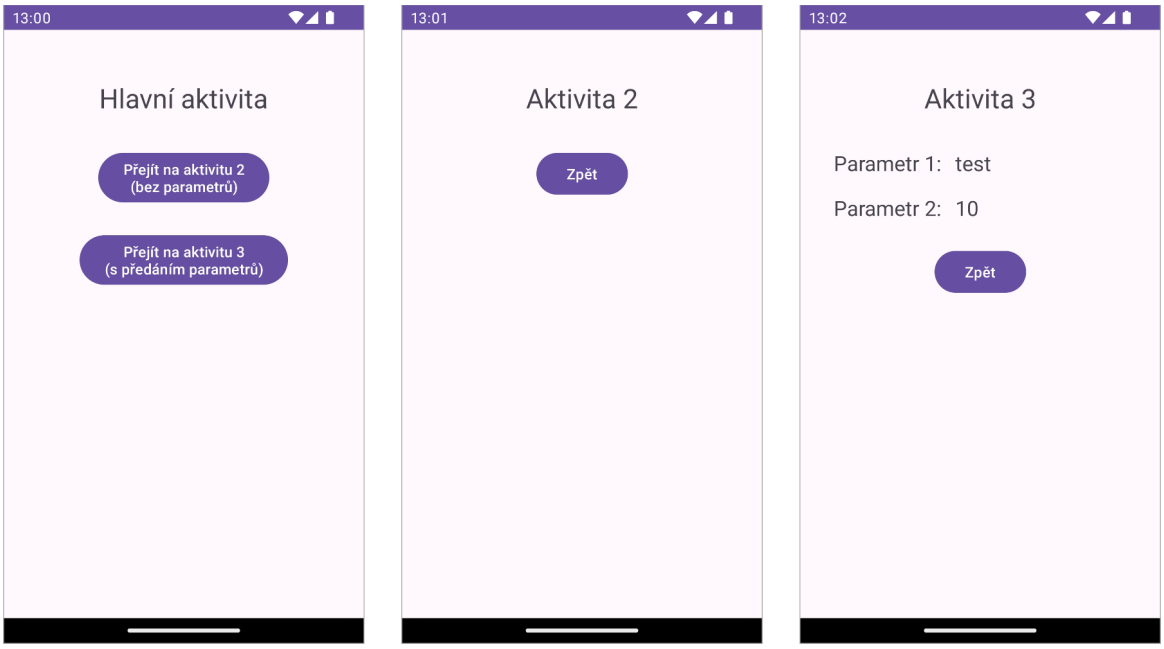
Využijeme toho tak, že si vytvoříme další pomocnou aktivitu Activity3, která bude tentokrát obsahovat také dvě pole typu TextView pro zobrazení předaných parametrů. Jedno se bude jmenovat txtView1, druhé txtView2. Událost onCreate() aktivity Activity3 rozšíříme o výpis předaných parametrů:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity3);

    // nacteni intentu
    Intent mujIntent = getIntent();
    // nacteni prvniho, textoveho, parametru
    String parametr1 = mujIntent.getStringExtra("param1");
    // nacteni druheho, ciselneho, parametru
    int parametr2 = mujIntent.getIntExtra("param2", 0);

    // zapis nactenych parametru do textovych poli
    TextView txtView1 = (TextView)findViewById(R.id.textView1);
    txtView1.setText(parametr1);
    TextView txtView2 = (TextView)findViewById(R.id.textView2);
    txtView2.setText(Integer.toString(parametr2));
}
```

Do MainActivity přidáme tlačítko, které zavolá metodu intent2(). V ní opět zavoláme prostřednictvím instance třídy Intent pomocnou aktivitu a navíc jí předáme číselný a textový parametr:

```
public void intent2 (View view) {
    Intent intent = new Intent(this, Activity3.class);
    intent.putExtra("param1", "test");
    intent.putExtra("param2", 10);
    startActivity(intent);
}
```

Výsledek by měl vypadat jako na obrázku 24.1. Z hlavní aktivity můžeme přejít na aktivitu pomocnou a zpět. V prvním případě prostým zavoláním, ve druhém s předáním parametrů.



Obrázek 24.1: Explicitní intent, použitý uvnitř aplikace pro volání aktivit.

24.2 Implicitní intent

Zatímco pro explicitní intent bylo nutné říct, kterou aktivitu voláme, v případě implicitního intentu nic takového není potřeba. Co se má stát volí vývojář z množiny typů intentů, které jsou v Androidu k dispozici. Tato množina je poměrně rozsáhlá, systém umožňuje spojit hovor, napsat SMS, e-mail, otevřít obchod Google Play Store, ukázat polohu v Google mapách a mnoho dalšího. Způsob volání je většinou potřeba najít v dokumentaci.

24.2.1 Implicitní intent, 1. příklad

Napsat aplikaci, která zavolá akci systému, není nic složitého. Zkusíme zavolat fotoaparát a pokud je jich v systému víc, necháme na uživateli, aby si zvolil, kterým chce akci provést.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 24_Intent
projekt ImplicitniIntent1

Co je potřeba udělat?

- vytvořit aktivitu s tlačítkem, které zavolá metodu s intentem

- pomocí intentu vyvolat aplikaci pro focení

Do MainActivity přidáme tlačítko, které bude volat metodu spustitKameru():

```
public void spustitKameru (View view) {
    Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    startActivity(intent);
}
```

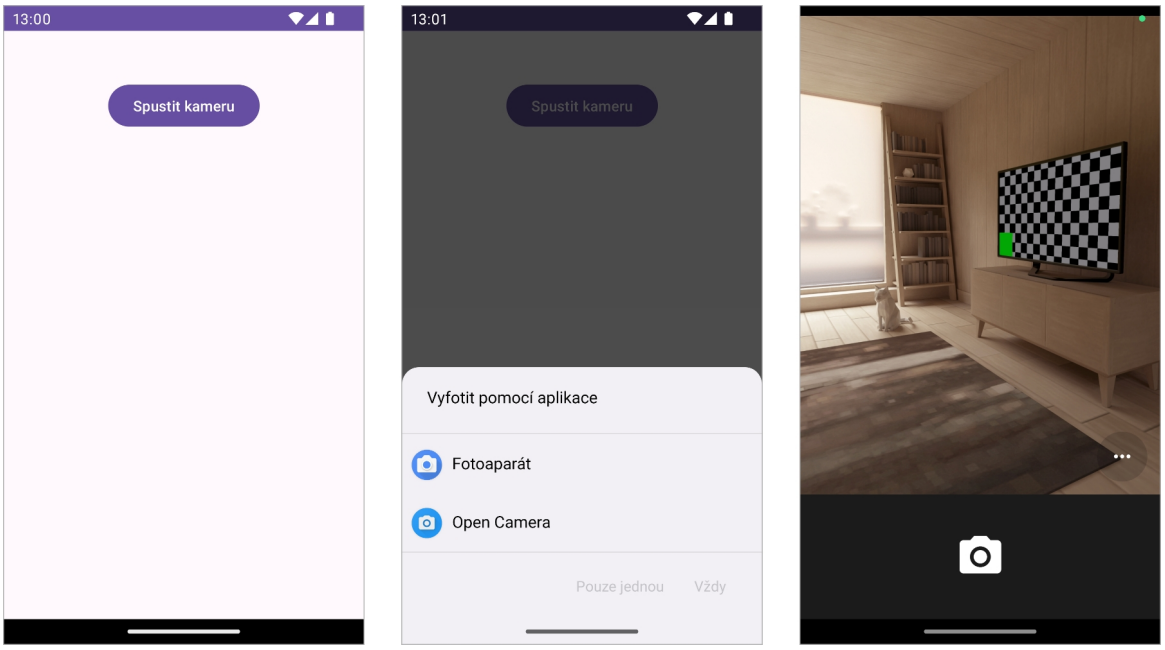
Je to takto jednoduché. Nejprve vytvoříme novou instanci třídy Intent, které v parametru sdělíme, že jde o pořízení fotky a potom aktivitu jednoduše zavoláme. Nic víc není potřeba, o zbytek se postará sám operační systém.

Výsledek by měl vypadat jako na obrázku 24.2. Po stisknutí tlačítka Android zjistí, jaké má k dispozici aplikace zpracovávající akci typu ACTION_IMAGE_CAPTURE. Pokud nějakou najde, tak ji spustí. Najde-li jich víc, dá uživateli prostřednictvím dialogu vybrat, kterou by chtěl použít.

Metodu spustitKameru() můžeme dále vylepšit. Pokud by v systému nebyla nainstalovaná aplikace pro pořizování fotek, skončilo by volání intentu chybou. Tomu můžeme předejít tak, že otestujeme, jestli je pro daný typ intentu aplikace k dispozici. Upravená metoda by vypadala takto:

```
public void spustitKameru (View view) {
    Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    if (intent.resolveActivity(getPackageManager()) != null) {
        startActivity(intent);
    }
    else {
        Toast.makeText(getApplicationContext(),
                       "Není nainstalovaná aplikace pro focení.",
                       Toast.LENGTH_SHORT).show();
    }
}
```

Ve zdokonalování aplikace by se dalo pokračovat. Další úpravou, která se nabízí, by mohlo být předání parametru s cestou k souboru, do kterého má kamera výsledek uložit. K parametrům implicitního intentu se ale dostaneme až v dalším příkladu.



Obrázek 24.2: Implicitní intent, volající aplikaci pro pořízení fotografie. Na prostředním obrázku je vidět, že jsou v zařízení nainstalované dvě takové aplikace, uživatel si zvolí kterou chce použít.

24.2.2 Implicitní intent, 2. příklad

Na dalším příkladu si ukážeme volání pomocí implicitního intentu s předáním dat v parametru. Z aplikace zavoláme prohlížeč webových stránek a necháme zobrazit vyhledávač Google.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 24_Intent
projekt ImplicitniIntent2

Co je potřeba udělat?

- vytvořit aktivitu s tlačítkem, které zavolá metodu s intentem
- pomocí intentu vyvolat aplikaci pro zobrazení webové stránky

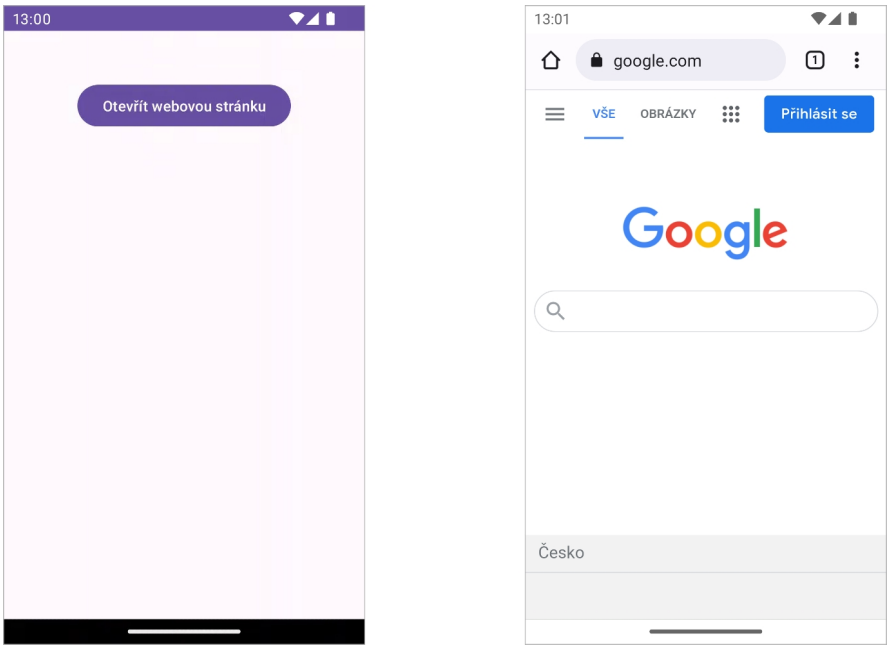
Do MainActivity přidáme tlačítko, které bude volat metodu otevritStranku():

```
public void otevritStranku (View view) {
    Intent intent = new Intent(Intent.ACTION_VIEW,
                                Uri.parse("https://www.google.com"));
    startActivity(intent);
}
```

Postup volání je téměř identický s předchozím příkladem. Teď jsme navíc využili jedné z vlastností třídy Intent, že jí můžeme prostřednictvím přetíženého konstruktoru předat v parametru adresu.

Výsledek by měl vypadat jako na obrázku 24.3. Po stisknutí tlačítka Android spustí prohlížeč, předá mu adresu a ten se už postará o její načtení a zobrazení.

Toto je ten nejjednodušší způsob předání parametru. U implicitního intentu můžeme ale také používat předávání hodnot pomocí parametrů putExtra, stejně jako jsme si to ukázali už u explicitního intentu. Toho se využívá zejména v případech, kdy je potřeba předat více parametrů, například při odesílání SMS zprávy nebo e-mailu, kde je kromě samotného textu potřeba předat také adresáta a případně předmět odesílané zprávy.



Obrázek 24.3: Implicitní intent, volající internetový prohlížeč. Adresa zobrazené stránky je předaná parametrem

24.3 Intent aplikace

V příkladu implicitního intentu jsme si ukázali, jak lze vyvolat aktivitu, o které rozhodne Android nebo uživatel. Jak ale dosáhnout toho, aby Android při volání implicitního intentu vybral naši aktivitu? Stejně jako všechno kolem intentů, ani toto není příliš složité. V manifestu aplikace musíme pro danou aktivitu zaregistrovat intent-filter. Tím operačnímu systému sdělíme, jaké typy volání jsou pro naši aktivitu akceptovatelné a on ji v případě potřeby zařadí do výběru.

24.3.1 Intent aplikace, příklad

Zkusíme si zaregistrovat naši vlastní aplikaci jako responzivní pro volání intentem typu VIEW, neboli budeme předstírat, že je regulérním prohlížečem webových stránek. Ve skutečnosti ale nebudeme webovou stránku zobrazovat, pouze uživateli vypíšeme předanou adresu webové stránky.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 24_Intent
projekt IntentAplikace

Co je potřeba udělat?

- vytvořit aktivitu, která bude simulovat webový prohlížeč

- zaregistrovat v manifestu aktivitu pro implicitní intent typu prohlížeč

- zavolat z jiné aplikace přes implicitní intent prohlížeč

Založíme novou aplikaci, MainActivity použijeme libovolným způsobem. Nebo ji můžeme nechat prázdnou. Přidáme další aktivitu. Nazvěme ji třeba IntentActivity. Do ní vložíme pole typu TextView, které bude zobrazovat jeden řádek textu. Nebudeme se pokoušet vytvářet celý prohlížeč, pouze při zavolání intentem necháme aplikaci zobrazit adresu, kterou mu volající aplikace předá. A k tomu bude sloužit právě toto textové pole.

Aktivita IntentActivity musí obsahovat také rutinu pro zpracování předávaného intentu. Vložíme ji do události onCreate():

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_intent);

    // ziskat predavany intent
    Intent mujIntent = getIntent();

    // zajima nas pouze intent typu VIEW
    if (mujIntent.getAction().equals(Intent.ACTION_VIEW)) {

        // nacist predana data
        Uri data = mujIntent.getData();

        // data zobrazit
        TextView txtView = (TextView)this.findViewById(R.id.textView);
        txtView.setText(data.toString());
    }
}
```

V podmínce testujeme, jestli je předávaný intent typu VIEW, neboli webový prohlížeč. Tato podmínka není nutná, pokud budeme aktivitu volat pouze jako prohlížeč. Data z intentu načteme, jde o adresu stránky, která se má v prohlížeči zobrazit a tuto adresu vyplníme do pole na formuláři. Tím je simulace prohlížeče připravena.

V manifestu musíme zaregistrovat aktivitu jako responzivní pro volání typu VIEW. Najdeme blok aktivity IntentActivity a doplníme do něj filtr s atributem action typu VIEW:

```
<activity android:name=".IntentActivity" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="https"/>
    </intent-filter>
</activity>
```

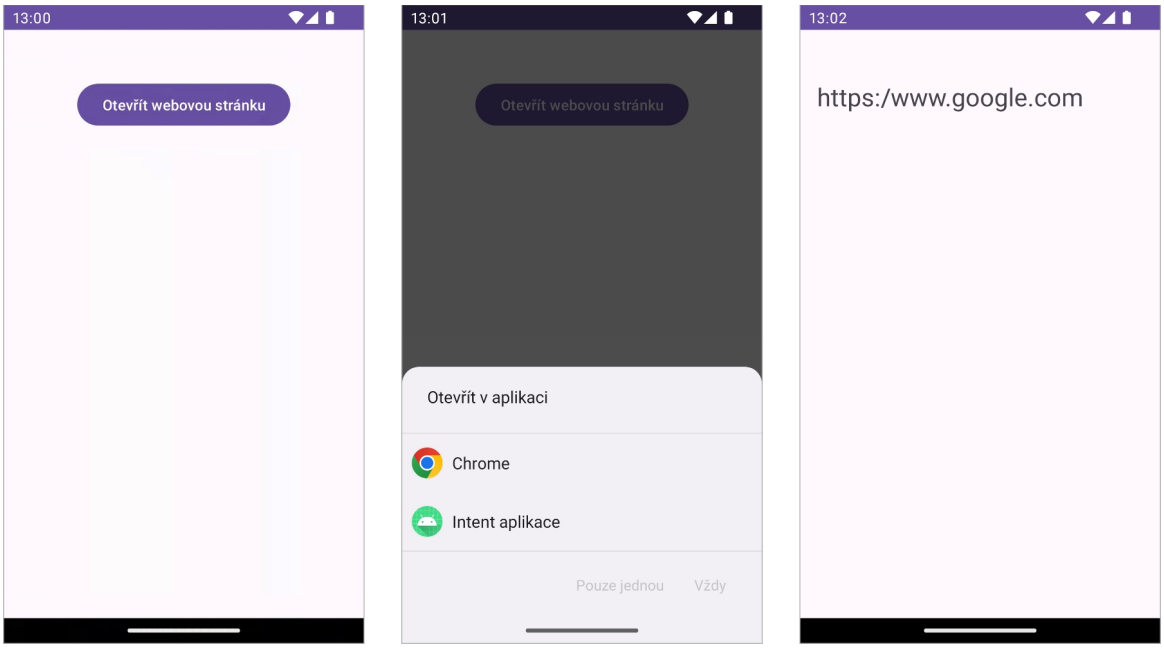
Konstanta „https“ říká, že budeme reagovat pouze na volání využívající zabezpečený protokol. Neboli na webové stránky přístupné přes http nebudeme reagovat.

Kdybychom se nespokojili s voláním aktivity coby prohlížeče, můžeme ji současně zaregistrovat třeba i jako prohlížeč obrázků. V takovém případě by popis aktivity v manifestu vypadal takto:

```
<activity android:name=".IntentActivity" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="https"/>
    </intent-filter>
    <intent-filter>
        <action android:name="android.media.action.IMAGE_CAPTURE" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

S tímto nastavením bude aktivita reagovat na dva různé intenty. V případě prohlížeče internetu a obrázků to možná není úplně praktické, ale v některých případech se může takové spojení hodit.

Pro zavolání implicitního intentu můžeme použít aplikaci, kterou jsme vytvořili v kapitole 24.2.2 Implicitní intent, 2. příklad. Ta umí zavolat všechny nainstalované prohlížeče a předat jim adresu https://www.google.com. Na zařízení nebo emulátoru by mělo volání prohlížeče z aplikace vypadat jako na obrázku 24.4.



Obrázek 24.4: Volání vlastní aplikace, zaregistrované jako prohlížeč. K volání je použita aplikace Implicitní intent 2.

25 Operace na pozadí

Operace na pozadí, neboli takzvané paralelní zpracování úloh, je technologie, která umožňuje zpracování programového kódu bez toho, aby byl zablokován běh hlavního vlákna aplikace. Uživatelsky to tedy znamená, že zatímco s aplikací můžeme pracovat běžným způsobem, tak současně kdesi vzadu v zařízení probíhají časově náročné výpočty, periodicky se opakující akce, nebo třeba obsluha periférií.

Způsobů, kterými můžeme paralelní zpracování realizovat, je v Androidu hned několik. Ukážeme si tři základní - vlákna neboli threads, která poskytuje samotná Java a která umožní vytvořit jakousi samostatnou větev aplikace, ve které běží program nezávisle na hlavní aktivitě. Druhou metodou je časovač, neboli třída CountDownTimer, která v pravidelných intervalech spouští připravenou programovou rutinu. A opět je nezávislá na hlavní aktivitě. Ta nezávislost má ovšem v obou případech jednu společnou výjimku a tou je ukončení aktivity. Jakmile přestane aktivita existovat, ukončí se také vlákno nebo časovač. Pro účely úplné nezávislosti na hlavní aktivitě bychom museli použít službu, která je třetím způsobem, na který se podíváme podrobněji.

25.1 Vlákno (thread)

Vlákna jsou součástí jazyka Java a způsob implementace je zjednodušeně následující:

```
new Thread(new Runnable(){
  public void run() {
    // rutina bezici ve vlakne
  }
}).start();
```

Napíšeme si jednoduchou aplikaci, která po stisknutí tlačítka vytvoří vlákno, to spustí připravenou rutinu, která každou sekundu pípne a zapíše informaci do logu.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 25_Operace_na_pozadi
projekt Vlakno

Co je potřeba udělat?

- připravit formulář se dvěma tlačítky, pro spuštění a ukončení běhu vlákna
- napsat metodu která definuje a spustí vlákno
- přidat metodu pro ukončení vlákna

Formulář vytvoříme obvyklým způsobem. Budou stačit dvě tlačítka, která budou po stisknutí volat metody spustitVlakno() a ukoncitVlakno().

Do MainActivity.java přidáme proměnnou vlakno typu Thread, která bude udržovat ukazatel na samotné vlákno. To využijeme jednak k tomu, abychom věděli, jestli už vlákno existuje a potom také proto, abychom ho mohli zastavit:

```
private Thread vlakno = null;
```

Do metody spustitVlakno() přidáme kód, který založí vlákno, definuje činnost kterou má vlákno vykonávat a hned ho spustí:

```
public void spustitVlakno(View view) {
    if (vlakno != null) {
        Toast.makeText(this, "Vlákno je již spuštěno",
                                        Toast.LENGTH_SHORT).show();
        return;
    }

    vlakno = new Thread(new Runnable() {
        @Override
        public void run() {
            MediaPlayer prehravac = MediaPlayer.create(getApplicationContext(),
                                        R.raw.pipnuti);
            try {
                for (int i = 0; i < 100; i++) {

                    // spustit prehravani zvuku
                    if (prehravac.isPlaying())
                        prehravac.stop();
                    prehravac.start();

                    // zapsat informaci do logu
                    Log.i("Vlakno", "Vlakno bezi, krok " + i);

                    // zastavit vlakno na 1 sekundu
                    Thread.sleep(1000);
                }
            }
            catch (Exception e) {
                Log.i("Sluzba test", "Vyjimka: " + e.getMessage());
            }
        }
        });
     vlakno.start();
}
```

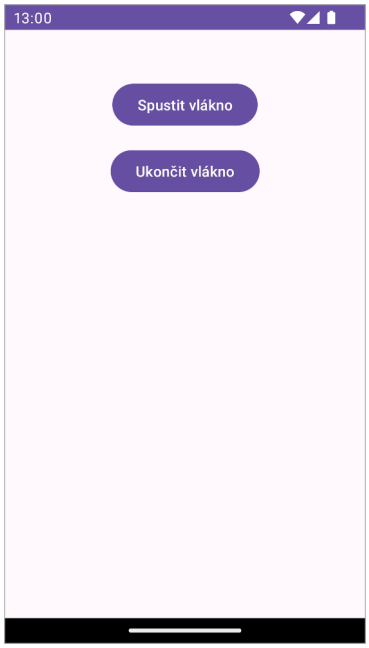
Nejprve je nutné zjistit, jestli už vlákno existuje. Pokud ano, znamená to, že je spuštěné a že se nemáme pokoušet o vytváření další instance. Ne že by to nebylo možné, vláken můžeme vytvořit hned několik, ale hůř by se s nimi pracovalo. Proto zatím vystačíme pouze s jedním vláknem a dokud jeho běh neskončí, tak uživateli nedovolíme vytvořit další. Pouze mu v takovém případě toastem vypíšeme informaci, že vlákno už existuje.

Abychom mohli přehrát zvuk, uložíme si zvukový soubor do adresáře app/res/raw a ve vláknu vytvoříme instanci přehrávače MediaPlayer. Potom necháme ve smyčce zvuk přehrávat, přičemž po každém přehrání zastavíme běh vlákna pomocí instrukce Thread.sleep, na jednu sekundu. Zároveň zapíšeme do logu informaci o právě proběhlé iteraci. Zbývá vlákno spustit pomocí volání metody start(). Výsledkem bude pípnutí každou sekundu.

Aby bylo možné vlákno zastavit, přidáme metodu ukoncitVlakno(), která nejprve zkontroluje, jestli vlákno existuje a pokud ano, tak jeho běh instrukcí interrupt() přeruší:

```
public void ukoncitVlakno(View view) {
    if (vlakno != null) {
        vlakno.interrupt();
        vlakno = null;
    }
}
```

Aplikace samotná není vizuálně příliš zajímavá, důležité je to, co se stane po stisknutí tlačítek pro spuštění a zastavení vlákna.



Obrázek 25.1: Aplikace, která spustí a zastaví vlákno běžící na pozadí operačního systému zařízení

25.2 Časovač (timer)

Zatímco vlákno po spuštění jednoduše běželo a vykonávalo jednotlivé instrukce kontinuálně, časovač se spouští v pravidelných intervalech, zpravidla na relativně krátkou dobu. Android má knihovnu CountDownTimer, která slouží právě k tomuto účelu. Základní způsob implementace je následující:

```
CountDownTimer casovac = new CountDownTimer(TRVANI, INTERVAL) {
    @Override
    public void onTick(long millisUntilFinished) {
        // rutina spustena casovacem
    }
    @Override
    public void onFinish() {
        // rutina spustena po skonceni behu casovace
    }
}.start();
```

Předefinovaná metoda onTick() spustí po uplynutí požadované doby programovou rutinu. Metoda onFinish() spustí další připravenou rutinu, jakmile časovač ukončí svoji činnost. Předčasně je možné zastavit čítač pomocí volání metody casovac.cancel(). Pochopitelně jeho činnost skončí také s ukončením aktivity nebo aplikace.

Konstruktor třídy CountDownTimer požaduje dva parametry typu long. Konstanta TRVANI, v prvním parametru, obsahuje hodnotu, která říká, jak dlouho má časovač běžet. Udává se v milisekundách, takže například hodnota 10 000 znamená, že časovač skončí svoji činnost po deseti sekundách a poté vyvolá událost onFinish(). Konstanta INTERVAL říká, po jaké době se má

opakovaně volat metoda onTick(). Je také v milisekundách, takže například hodnota 500 spustí onTick() každé půl vteřiny.

Někdy se může hodit nekonečný časovač. Tedy takový, který poběží tak dlouho, jak bude spuštěná aktivita nebo celá aplikace. Toto nastavení třída CountDownTimer bohužel neumožňuje, vždy je nutné zadat přesnou dobu běhu čítače. Je však možné použít jako parametr konstruktoru, udávající dobu běhu, konstantu Long.MAX_VALUE. Ta reprezentuje poměrně vysoké 64 bitové číslo, 9 223 372 036 854 775 807. Neboli v přepočtu na milisekundy přibližně 292 milionů let. A to už pro většinu aplikací stačí.

Vytvoříme aplikaci s časovačem, který poběží jednu minutu a po spuštění vždy ve vteřinových intervalech pípne. Bude tedy dělat to samé, co vlákno v předchozím příkladu, ale nebude potřebovat cyklus s čekáním v každé iteraci, pouze při vyvolání rutiny pípne a hned ji ukončí. O další spuštění se postará časovač sám, v definovaném okamžiku, čili každou vteřinu.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 25_Operace_na_pozadi
projekt Casovac

Co je potřeba udělat?

- připravit formulář se dvěma tlačítky, pro spuštění a ukončení časovače
- napsat metodu která definuje a spustí časovač
- přidat metodu pro ukončení časovače

Formulář připravíme snadno a rychle, pouze do něj umístíme dvě tlačítka, která budou po stisknutí volat metody spustitCasovac() a ukoncitCasovac().

Do MainActivity.java přidáme proměnnou casovac typu CountDownTimer, která bude udržovat ukazatel na timer. To využijeme jednak proto, abychom věděli, jestli už časovač existuje a potom pro to, abychom ho mohli zastavit:

```
private CountDownTimer casovac = null;
```

Do metody spustitCasovac() přidáme kód, který založí timer, definuje činnost kterou má vykonávat a hned ho spustí:

```
public void spustitCasovac(View view) {
    if (casovac != null) {
        Toast.makeText(this, "Časovač je již spuštěn",
                                                Toast.LENGTH_SHORT).show();
        return;
    }

    casovac = new CountDownTimer(60000, 1000) {
        MediaPlayer prehravac = MediaPlayer.create(getApplicationContext(),
                                                R.raw.pipnuti);
        public void onTick(long millisUntilFinished) {

            // spustit prehravani zvuku
            if (prehravac.isPlaying())
                prehravac.stop();
            prehravac.start();
```

```
            // zapsat informaci do logu
            Log.i("Casovac", "Vyvolana udalost onTick");
        }

        public void onFinish() {
            // konec
        }
    };

    casovac.start();
}
```

Metoda nejdřív zjistí, jestli už časovač neběží a pokud ano, tak pouze vypíše uživateli informaci. Poté založí instanci časovače CountDownTimer, v našem případě s prvním parametrem, který říká, že má běžet 60 sekund (resp. 60 000 milisekund) a z toho každou sekundu (resp. každých 1 000 milisekund) spustit rutinu.

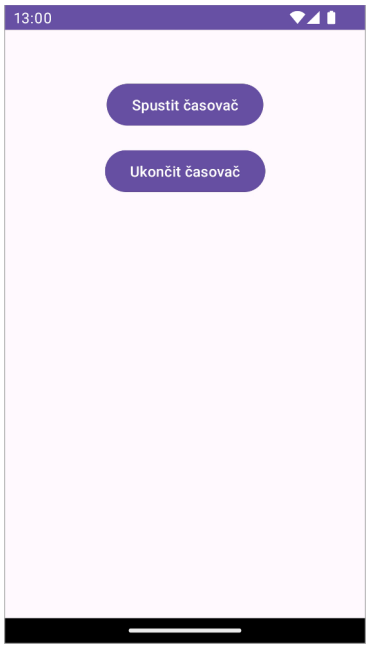
Samotná rutina je umístěna v události onTick() a opět dělá to, že pomocí MediaPlayeru přehraje zvuk pípnutí a zapíše informaci do logu. Potom skončí. Opět ji vyvolá až časovač po uplynutí nastavené jedné sekundy.

Jako malá prémie je k dispozici událost onFinish(), která se volá po uplynutí předepsaného času běhu timeru.

Abychom mohli časovač zastavit, přidáme metodu ukoncitCasovac(). Ta nejprve zkontroluje, jestli časovač existuje a pokud ano, tak jeho běh instrukcí cancel() ukončí:

```
public void ukoncitCasovac(View view) {
    if (casovac != null) {
        casovac.cancel();
        casovac = null;
    }
}
```

Aplikace opět není vizuálně příliš zajímavá, důležité je to, co se stane po stisknutí tlačítek pro spuštění a zastavení časovače.



Obrázek 25.2: Aplikace s časovačem, spuštěná na emulátoru

25.3 Služba (service)

Služba je druh aplikace, nebo samostatná komponenta aplikace, která běží v operačním systému Android na pozadí. Nemá uživatelské rozhraní, takže sama o sobě není vidět. Může ale vyvolávat různé akce, například přehrát zvuk, zobrazit upozornění, spustit aplikaci a tak podobně. Služba běží i tehdy, když uživatel pracuje s úplně jinou aplikací – příkladem je přehrávač hudby, který hraje na pozadí a jehož součástí je jak běžná aplikace, kde si uživatel nastaví co se má přehrávat, tak služba, která následně přehrávání zajišťuje. Aplikace přehrávače může být zastavena nebo ukončena, ale služba běží a hraje dál, dokud činnost nedokončí, nebo dokud nedostane signál (opět z aplikace), že došlo k nějaké změně.

Existují dva základní typy služeb, z nichž ta první se dělí na další dvě:

- Unbound služba – běží nezávisle na aktivitě a ta s ní nemá možnost komunikovat. Aktivita může službu pouze spouštět a ukončovat. Spouští se metodou startService() a ukončí se buď sama, nebo pomocí metody stopService().
    - Foreground – služba, jejíž běh je nějakým způsobem zaznamenatelný uživatelem. Například již zmíněný přehrávač hudby. Uživatel její výstup slyší, takže není žádoucí, aby mohl systém její běh přerušit nebo zastavit. Foreground služba má stejnou prioritu jako aktivita na popředí. Služba v tomto režimu musí mít ve stavovém řádku systému ikonku, pomocí níž je možné otevřít související aktivitu, která obsahuje uživatelské rozhraní.
    - Background – služba, která běží na pozadí a uživatel ji žádným způsobem nevnímá. Jsou to například služby, které defragmentují soubory.
- Bound – služba, která je svázána s aktivitou. Svojí strukturou připomíná architekturu klient - server, kde služba je serverem a připojená aktivita (nebo aktivity) klientem. Spouští a ukončuje se spolu s aktivitou a má možnost s ní komunikovat prostřednictvím zasílání požadavků a přijímání odpovědí. Služba se spouští pomocí metody bindService() a ukončuje metodou

unbindService(). Jakmile se ukončí aktivita (resp. navázané aktivity), končí i služba samotná. Používá se například pro dlouhotrvající výpočty, které se zpracovávají na pozadí aplikace, zatímco samotnou aplikaci má uživatel k dispozici pro běžné operace.

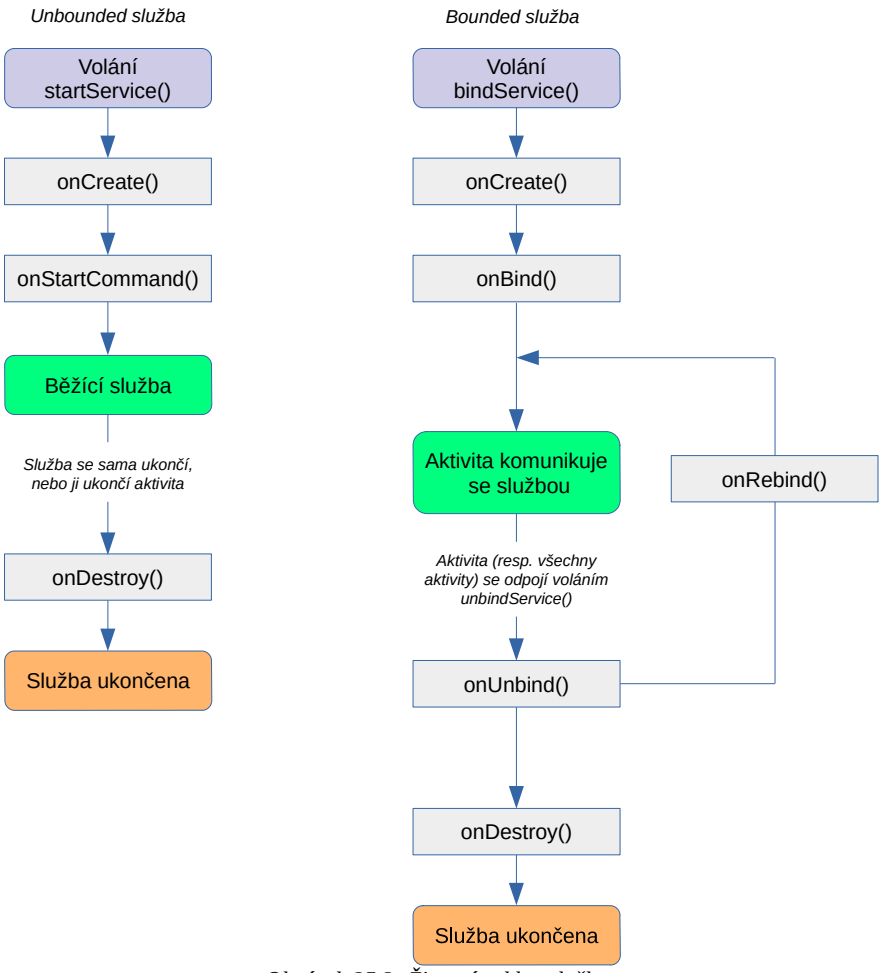
Životní cyklus služby

Životní cyklus se mírně liší pro bounded a unbounded služby. V každém případě ale začíná jejím spuštěním, ať už zavoláním metody startService() nebo bindService(). Podle volané metody se také pozná, jestli se službou pracujeme v režimu unbounded nebo bounded. Jako první se zavolá událost onCreate(), kterou můžeme využít pro inicializaci služby.

Poté je u unbounded služby vyvolána událost onStartCommand(), ve které zajistíme spuštění jádra služby, neboli kódu, který má služba zpracovávat. Tato událost může být vyvolána opakovaně, takže je v ní, hned na začátku, vhodné definovat, jak se má chovat, pokud je znovu spuštěna ještě předtím, než dokončila svůj předchozí běh.

Bound služba vyvolá událost onBind(), která poskytuje rozhraní iBinder, s jehož pomocí můžeme zajistit komunikaci aktivity se službou. Pokud se aktivita od služby odpojí, je navíc vyvolána událost onUnbind() a pokud služba není ukončena (to je tehdy, když jsou na ni připojeny ještě další aktivity), tak s připojením další aktivity vyvolá událost onRebind().

Jakmile unbounded služba ukončí zpracování, nebo je ukončena z aktivity, vyvolá se událost onDestroy() a služba zaniká. Bounded služba skončí odpojením poslední připojené aktivity, rovněž vyvolá událost onDestroy() a zaniká.



Obrázek 25.3: Životní cyklus služby

Základní metody, které použijeme při práci se službou

Unbounded služba:

startService() – z aktivity spustí službu. Pokud už služba běží, vyvolá událost onStartCommand().

stopService() – z aktivity ukončí službu, vyvolá událost onDestroy().

stopSelf() – služba sama ukončí svoji činnost. Například v okamžiku, kdy dokončila svoji práci.

Bounded služba:

bindService() – z aktivity spustí službu. Pokud už služba běží, vyvolá událost onBind().

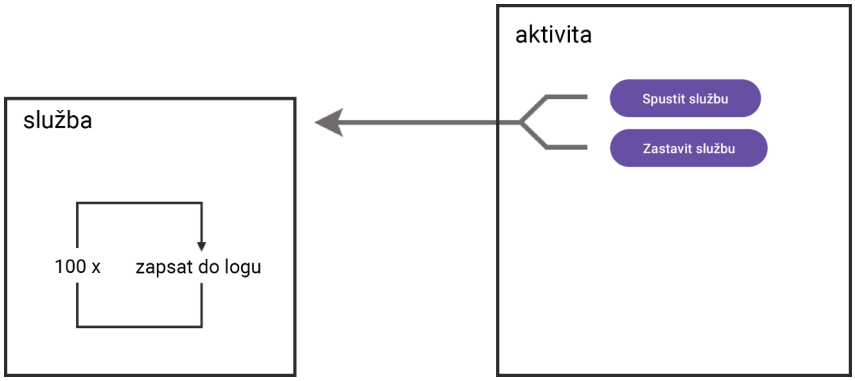
unboundService() – odpojí aktivitu od služby. Vyvolá událost onUnbind() a pokud je poslední připojenou aktivitou, tak také onDestroy().

Služba sama o sobě není aplikací, je spíše jenom cosi jako obálka, která určuje způsob jakým se vykoná její programový obsah. Proto se zpravidla realizuje tak, že se definuje několik základních událostí služby, zejména tedy onStartCommand() resp. onBind() a onDestroy() (viz životní cyklus služby) a samotný výkonný kód běží ve vláknu, které tyto události spustí a ukončí.

25.3.1 Služba, 1. příklad

Jako příklad si uděláme jednoduchou unbounded foreground službu, která bude spouštěna stisknutím tlačítka na aktivitě. Služba bude po spuštění opakovaně vypisovat záznamy do logu.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 25_Operace_na_pozadi
projekt Sluzba1



Obrázek 25.4: Jednoduchá služba ovládaná tlačítky na formuláři (aktivitě)

Co je potřeba udělat?

- upravit manifest, aby aplikace podporovala běh služby
- vytvořit v samostatné třídě službu
- ve službě spouštět a ukončovat vlákno s výkonným kódem
- připravit formulář s tlačítkem pro spuštění a ukončení služby
- službu z hlavní aktivity spouštět a ukončovat

Manifest upravíme jednoduše. Dovnitř, do bloku application, vložíme záznam o službě:

```
<service android:name=".Sluzba"/>
```

Záznam může mít v parametrech kromě názvu nastavenou taky ikonu, popis, příznak jestli má služba komunikovat s jinými aplikacemi a tak dále. Zatím se spokojíme s pouhým názvem.

Založíme novou třídu a pojmenujeme ji například Sluzba. Tuto třídu, aby mohla být službou, musíme oddědit od třídy Service:

```
public class Sluzba extends Service {
```

Tím vznikne potřeba předefinovat některé základní metody. Do onCreate(), onBind() a onDestroy() pouze vložíme logovací informaci, nic víc teď potřebovat nebudeme. Příklad logování pro událost onCreate():

```
Log.i("Sluzba test", "Service onCreate()");
```

Zajímat nás ale bude především metoda onStartCommand(), která obsahuje hlavní řídící kód. Nejdřív rozhodneme, jak se má služba chovat pokud je volána opakovaně, ještě předtím než byl ukončen běh po předchozím vyvolání. Buď můžeme spustit další vlákno a nechat běžet proces vícekrát paralelně – v takovém případě nemusíme nyní dělat nic. S každým dalším vyvoláním služby vznikne nová instance. Nebo můžeme předchozí vlákno zastavit a spustit nové. A nebo třetí možnost, kterou v příkladu využijeme, zkontrolujeme, jestli služba běží a pokud ano, další spuštění budeme ignorovat. Přesněji řečeno, zapíšeme do logu informaci o neúspěšném pokusu o spuštění a původní vlákno necháme doběhnout.

Takto bude vypadat kód služby:

```
public class Sluzba extends Service {
    private boolean sluzbaBezi = false;

    @Override
    public void onCreate() {
        Log.i("Sluzba test", "Service onCreate()");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {

        // test jestli je sluzba uz spustena, pokud ano, znovu nespoustet
        if (sluzbaBezi) {
            Log.i("Sluzba test", "Sluzba je uz spustena.");
            return Service.START_STICKY;
        }
        else {
            sluzbaBezi = true;
            Log.i("Sluzba test", "Sluzba - onStartCommand()");
        }

        // Vytvorit nove vlakno a sluzbu spustit
        new Thread(new Runnable() {
            @Override
            public void run() {
                // na toto misto umistime kod, ktery ma sluzba zpracovavat

                // v sekundovych intervalech vypisovat do logu zaznamy
                try {
                    for (int i = 0; i < 100; i++) {
                        // vlakno na sekundu uspat
                        Thread.sleep(1000);
                        Log.i("Sluzba test", "Sluzba bezi, krok " + i);
                    }
                }
                catch (InterruptedException e) {
                    Log.i("Sluzba test", "Vyjimka InterruptedException");
                    stopSelf();
                }
                catch (Exception e) {
                    Log.i("Sluzba test", "Vyjimka: " + e.getMessage());
                }

                // Sluzba ukoncila svoji cinnost, bude zastavena
                stopSelf();
            }
        }).start();
        return Service.START_STICKY;
```

```
    }

    @Override
    public IBinder onBind(Intent arg0) {
        Log.i("Sluzba test", "Sluzba - onBind");
        return null;
    }

    @Override
    public void onDestroy() {
        sluzbaBezi = false;
        Log.i("Sluzba test", "Sluzba - onDestroy");
    }
}
```

Je důležité poznamenat, že zatímco událost onStartCommand() poměrně průhledným způsobem vytvoří a spustí vlákno, ve kterém bude běžet cyklus s výpisem do logu, ukončení proběhne vyvoláním události onDestroy(). Ta zkontroluje jestli je vlákno aktivní a pokud ano, tak se ho pokusí ukončit vyvoláním výjimky InterruptedException. To je důvod, proč ve vlákně čekáme na tuto výjimku a pokud k ní dojde, tak vlákno sami ukončíme. Pozor, služba to za nás neudělá. Pokud bychom na tento typ výjimky nečekali, aplikace by skončila v lepším případě chybou, v horším by spadla do větve obecné výjimky Exception, ta by zalogovala, ale vlákno by běželo dál.

Služba po spuštění vypisuje v sekundových intervalech do logu záznamy a po stém záznamu se sama ukončí. Logy můžeme sledovat v Android Studiu pod záložkou Logcat. Aplikaci spouštíme v režimu Debug (Shift + F9). Záznamy logujeme na úrovni Info (Log.i – i je metoda třídy Log, která označuje úroveň Info). Díky tomu si můžeme v okně Logcat přepnout filtrování na tuto úroveň a tím omezit vypisování informací pouze na ty, které nás zajímají.

Zde je ještě nutné upozornit na jeden zásadní parametr a tím je návratová hodnota metody onStartCommand(). Jakmile metoda proběhne a vytvoří vlákno které bude generovat výpisy do logu, tak vrátí konstantu START_STICKY. Tato konstanta říká, že pokud by musel systém službu z nějakého důvodu předčasně ukončit, tak ji má opět co nejdříve znovu spustit.

K dispozici jsou tyto základní konstanty:

START_STICKY – po předčasném ukončení systémem službu znovu spustit

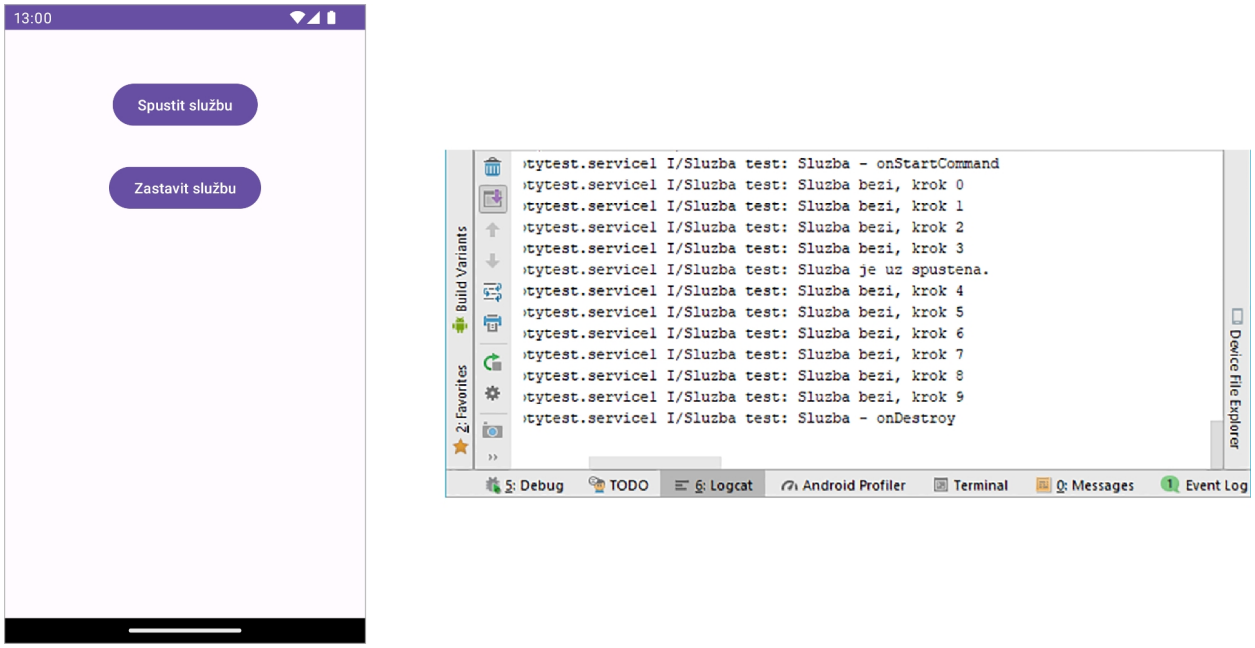
START_REDELIVER_INTENT – jako předchozí, ale navíc zachrání instanci služby

START_NOT_STICKY – po předčasném ukončení systémem už službu nespouštět

Spuštění služby z aktivity zajistíme pomocí metody startService(), které předáme instanci třídy Intent. Stejným způsobem službu ukončíme.

```
public void spustitSluzbu (View view) {
    Intent intent = new Intent(this, Sluzba.class);
    startService(intent);
}

public void zastavitSluzbu (View view) {
    Intent intent = new Intent(this, Sluzba.class);
    stopService(intent);
}
```

Výsledná aplikace by po spuštění a stisknutí tlačítka měla generovat záznamy do logu, přibližně jako na obrázku 25.5. Po spuštění služby můžeme aplikaci přepnout na pozadí, spustit si jinou aplikaci, nebo telefon zamknout, ale služba bude nerušeně dál vypisovat záznamy do logu, dokud ji neukončíme nebo dokud se neukončí sama. Sekundovou přesnost při vypisování logu je potřeba brát s rezervou. Zařízení sice bude výpisy generovat přibližně v pravidelných vteřinových intervalech, ale přenos informace do Android Studia a samotné zobrazení jsou rovněž asynchronní operace, takže se může stát, že se jednotlivé záznamy budou vypisovat s určitým zpožděním, případně že budou vypsány po více záznamech najednou. V závislosti na zatížení počítače.



Obrázek 25.5: Aplikace s tlačítky pro ovládání služby, která vypisuje informace do logovacího okna Logcat

Výpis do logu je sice zajímavý z programátorského úhlu pohledu, ale jinak není uživatelsky příliš atraktivní. Službu můžeme snadno vylepšit tak, aby s každým krokem ještě navíc pípla. Stačí dát do složky res/raw/ požadovaný krátký zvuk, řekněme že se bude jmenovat třeba pipnuti.mp3 a ten před spuštěním cyklu ve vláknu přiřadit MediaPlayeru:

```
MediaPlayer prehravac = MediaPlayer.create(getApplicationContext(),
                                                    R.raw.pipnuti);
```

V místě, kde se ve smyčce vypisuje logový záznam, přidáme přehrání zvuku:

```
if (prehravac.isPlaying())
    prehravac.stop();
prehravac.start();
```

Tím dosáhneme toho efektu, že služba každou vteřinu pípne, i když je telefon zamčený nebo máme spuštěnou úplně jinou aplikaci. Přestane až po stém pípnutí, nebo až dostane z formuláře tlačítkem povel k ukončení. Pokud nás nezajímá logování, tak v tomto případě není potřena aplikaci spouštět v režimu ladění, ale stačí použít Run app, resp. klávesu F9.

Výhodou je, že tentokrát uslyšíme mnohem přesněji kdy dochází k jednotlivým iteracím a můžeme si udělat představu, jak přesně se službě bude dařit přehrávat zvuk v závislosti na zatížení zařízení.

25.3.2 Služba, 2. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 25_Operace_na_pozadi
projekt Sluzba2

Ve většině případů je nutné se službou alespoň nějakým základním způsobem komunikovat. Obvykle to znamená nastavení hodnot v aktivitě, které službě předáme a naopak zasílání dat ze služby do aktivity, která je nějakým způsobem zobrazí. Na příkladu hudebního přehrávače je komunikace od aktivity ke službě například v možnosti posunutí přehrávání o několik sekund dopředu. Uživatel posune jezdec symbolizující pozici v přehrávané skladbě, takto nastavená hodnota se předá službě a ta přejde na požadovanou část skladby.

Stejně tak v opačném směru – služba může předávat aktivitě informaci o přehrávané skladbě, například místo, kde se zrovna nachází a aktivita umístí posuvník do příslušné pozice.

V praxi se používá několik způsobů předávání zpráv mezi aktivitou a službou, nejpraktičtější je však zřejmě třída BroadcastReceiver, která zajišťuje obecné přijímání zpráv systému Android. Tedy i zpráv, které vygeneruje aktivita nebo služba pomocí sendBroadcast() jako Intent.

Příklad odeslání zprávy:

```
Intent broadcastIntent = new Intent();
broadcastIntent.setPackage("cz.stokratandroid.sluzba2");
broadcastIntent.setAction("test");
broadcastIntent.putExtra("data", i);
sendBroadcast(broadcastIntent);
```

Odesílání zprávy funguje tak, že si vytvoříme instanci třídy Intent, do ní si nadefinujeme vlastní akci – prostě si vybereme jakýkoliv vlastní unikátní název, v daném případě „test“, pomocí metody putExtra() přidáme data, u nichž si opět zvolíme unikátní název (např. „data“) a hodnotu. Ta je v našem případě v proměnné i. Potom takový Intent už jenom odešleme a systém se postará o to, aby byl doručen aktivnímu příjemci. Tím je událost onReceive() ve třídě odvozené od BroadcastReceiver:

```
class DataUpdateReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals("test")) {
            int data = intent.getIntExtra("data", -1);
            prijataZprava.setText("Přijatá zpráva: " + Integer.toString(data));
        }
    }
}
```

Tato nová třída, která je rozšířením bázové třídy BroadcastReceiver, bude vyvolaná při odeslání zprávy Intent. Proto jako první zkontroluje, jestli jde o zprávu kterou očekává, tj. v našem případě, jestli je název akce „test“. Teoreticky můžeme tuto událost spouštět i pomocí dalších typů zpráv. Jde-li tedy o očekávaný typ zprávy, načte z ní pomocí metody getIntExtra() data a z těch potom například sestaví zprávu pro uživatele, nebo je použije k libovolnému jinému účelu.

Aby mohla být událost onReceive() vyvolána, musíme třídu zaregistrovat jako receiver:

```
DataUpdateReceiver dataUpdateReceiver = new DataUpdateReceiver();
IntentFilter intentFilter = new IntentFilter("test");
registerReceiver(dataUpdateReceiver, intentFilter, RECEIVER_NOT_EXPORTED);
```

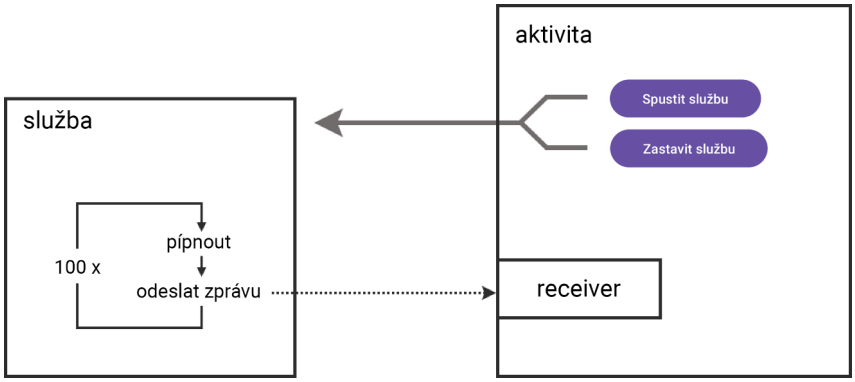
Do receiveru jsme přidali IntentFilter nastavený na zprávy typu „test“, takže se dá říct že rozlišení na typ zprávy už v samotné třídě DataUpdateReceiver není nutné. Je ho tam ale přesto vhodné nechat, protože v budoucnu můžeme tuto událost vyvolávat i pomocí jiných zpráv. Konstanta ve třetím parametru, RECEIVER_NOT_EXPORTED má bezpečnostní charakter. Omezuje přijímání zpráv pouze z vlastní aplikace. Pokud by měl mít receiver možnost přijímat zprávy i z jiných aplikací, musel by to mít povolené konstantou RECEIVER_EXPORTED. Příklad si posílá zprávy v rámci aplikace, bude tedy fungovat s první i druhou konstantou. Povolovat externí zprávy je ale nadbytečné.

Pokud potřebujeme činnost receiveru ukončit, typicky při přerušení nebo ukončení aktivity, stačí ho jednoduše odregistrovat:

```
if (dataUpdateReceiver != null)
   unregisterReceiver(dataUpdateReceiver);
```

Toto je základní princip předávání dat mezi aktivitou a službou, pomocí odesílání intent zpráv a jejím přijímáním DataUpdateReceiverem. Rozšíříme si předchozí příklad tak, aby kromě logování pípnutí každou vteřinu ještě služba odesílala informaci aktivitě a ta ji zobrazila na formuláři.

Jako příklad si uděláme jednoduchou unbounded foreground službu, která bude spouštěna stisknutím tlačítka na aktivitě. Služba bude po spuštění opakovaně vypisovat záznamy do logu. Vyjdeme z předchozího příkladu.



Obrázek 25.6: Služba která komunikuje s aktivitou (formulářem)

Co je potřeba udělat?

- upravit formulář tak, aby obsahoval nové TextView pole, s informací od služby
- přidat do služby odesílání Intent zprávy
- přidat do aktivity třídu, která bude přijímat zprávy
- zaregistrovat a odregistrovat třídu, která bude přijímat zprávy

Úprava formuláře bude úplně jednoduchá, prostě přidáme pod tlačítka pole typu TextView vhodné velikosti.

Úprava služby také nebude nijak komplikovaná. Přidáme do ní, do místa kde se služba spustí kvůli pípnutí MediaPlayer, ještě odeslání zprávy. To bude vypadat přesně stejně, jako jsme si ukázali na začátku této kapitoly.

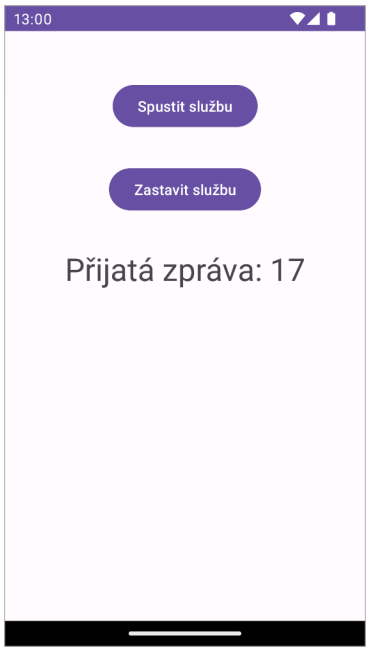
Do třídy MainActivity nejdřív přidáme dvě lokální proměnné, z nichž první bude ukazatelem na nové textové formuláři a druhá ukazatelem na instanci třídy DataUpdateReceiver. Zároveň si v události onCreate() první z uvedených proměnných naplníme. Potom do třídy vložíme třídu DataUpdateReceiver, opět přesně v té podobě, jak jsme si ji popsali.

Potom už jenom musíme přidat zaregistrování a odregistrování receiveru, k čemuž využijeme událostí onResume() a onPause(). Takto bude vypadat celý kód třídy:

```
public class MainActivity extends AppCompatActivity {
    private TextView prijataZprava;
    private DataUpdateReceiver dataUpdateReceiver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        prijataZprava=this.findViewById(R.id.txtPrijataZprava);
    }

    // kliknuti na tlacitko Spustit sluzbu
    public void spustitSluzbu (View view) {
        Intent intent = new Intent(this, Sluzba.class);
        startService(intent);
    }

    // kliknuti na tlacitko Zastavit sluzbu
    public void zastavitSluzbu (View view) {
        Intent intent = new Intent(this, Sluzba.class);
        stopService(intent);
    }

    // udalost onResume, volana pri spusteni aktivity
    @Override
    public void onResume() {
        super.onResume();

        if (dataUpdateReceiver == null)
            dataUpdateReceiver = new DataUpdateReceiver();
        IntentFilter intentFilter = new IntentFilter("test");
        registerReceiver(dataUpdateReceiver, intentFilter,
                                                RECEIVER_NOT_EXPORTED);
    }

    // udalost onPause, volana pri preruseni aktivity
    @Override
    public void onPause() {
        super.onPause();

        if (dataUpdateReceiver != null)
            unregisterReceiver(dataUpdateReceiver);
    }
```

```
    // trida prijimajici data odeslana sluzbou
    class DataUpdateReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
            if (intent.getAction().equals("test")) {
                int data = intent.getIntExtra("data", -1);
                prijataZprava.setText("Přijatá zpráva: " +
                                                Integer.toString(data));
            }
        }
    }
}
```

Výsledná aplikace by po spuštění měla vypadat přibližně jako na obrázku 25.7. Pokud aktivitu ukončíme, přestane přijímat zprávy od služby která stále poběží. Když se následně k aktivitě vrátíme, měla by zprávy zase začít přijímat. Což znamená, že by měla následující sekundu aktualizovat obsah textového pole.



Obrázek 25.7: Aplikace s tlačítky pro ovládání služby, která zároveň přijímá zprávy odeslané službou

25.3.3 Služba, 3. příklad

V minulém příkladu služba odesílala zprávy do aktivity, ta je přijímala a zpracovávala. V praxi ale budeme stejně tak potřebovat opačný proces, tedy aby zprávu odeslala aktivita, služba ji přijala a zpracovala.

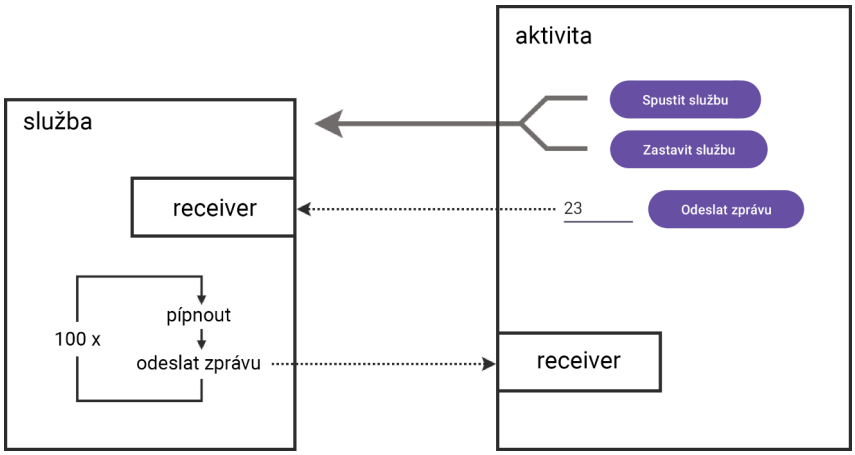
Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 25_Operace_na_pozadi
projekt Sluzba3

Způsob, jak dosáhnout předávání zpráv od aktivity ke službě je prakticky stejný jako při předávání v opačném směru. Opět použijeme k odeslání zprávy typu Intent metodu sendBroadcast() a pro přijetí událost onReceive() třídy BrodcastReceiver.

Vylepšíme předchozí příklad o možnost odeslání číselné hodnoty, zadané uživatelem, do služby. Služba bude stejně jako předtím opakovaně pípat, ale tentokrát navíc po přijetí zprávy

upraví hodnotu iterace podle toho, co jí uživatel pošle. Pokud tedy bude u dvacátého pípnutí ze sta a dostane zprávu s hodnotou 90, nastaví si iteraci na danou hodnotu a pípne už pouze desetkrát.

Jako příklad si uděláme jednoduchou unbounded foreground službu, která bude spouštěna stisknutím tlačítka na aktivitě. Služba bude po spuštění opakovaně pípat a odesílat zprávy.



Obrázek 25.8: Služba která obousměrně komunikuje s aktivitou (formulářem)

Co je potřeba udělat?

- upravit formulář tak, aby obsahoval nové numerické pole a tlačítko pro odeslání zadané hodnoty
- přidat do aktivity odesílání Intent zprávy
- přidat do služby třídu, která bude přijímat zprávy
- zaregistrovat a odregistrovat třídu, která bude přijímat zprávy

Úprava formuláře není složitá. Prostě do něj pole a tlačítko vložíme, zarovnáme vhodným způsobem na střed, v MainActivity založíme metodu poslatZpravu() a tlačítku nastavíme aby ji v události onClick() zavolalo.

V metodě poslatZpravu() načteme hodnotu ze vstupního pole, tu převedeme na datový typ integer a sestavíme intent ve kterém hodnotu odešleme. Musíme zvolit vhodný název akce, aby byl unikátní, pozor tedy při případném kopírování intentu z předchozího příkladu.

```
public void poslatZpravu(View view) {
    TextView txtZpravaKOdeslani = findViewById(R.id.txtZpravaKOdeslani);
    int intZpravaKOdeslani =
                    Integer.parseInt(txtZpravaKOdeslani.getText().toString());

    Intent broadcastIntent = new Intent();
    broadcastIntent.setPackage("cz.stokratandroid.sluzba3");
    broadcastIntent.setAction("zpravaZAktivity");
    broadcastIntent.putExtra("data", intZpravaKOdeslani);
    sendBroadcast(broadcastIntent);
}
```

Přejdeme ke třídě Sluzba a přidáme si do ní dvě nové proměnné. Jednu pro uchování instance třídy DataUpdateReceiver a druhou pro předání načtené zprávy.

```
private DataUpdateReceiver dataUpdateReceiver;
private int prijataZprava = -1;
```

Třída pro příjem zprávy je principiálně stejná jako v předchozím příkladu. Opět bude rozšiřovat bázovou třídu BroadcastReceiver, přepíšeme v ní událost onReceive() a v té se pouze přesvědčíme, jestli je přijatá zpráva očekávaného typu a hodnotu uložíme do připravené proměnné prijataZprava.

```
class DataUpdateReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals("zpravaZAktivity")) {
            prijataZprava = intent.getIntExtra("data", -1);
        }
    }
}
```

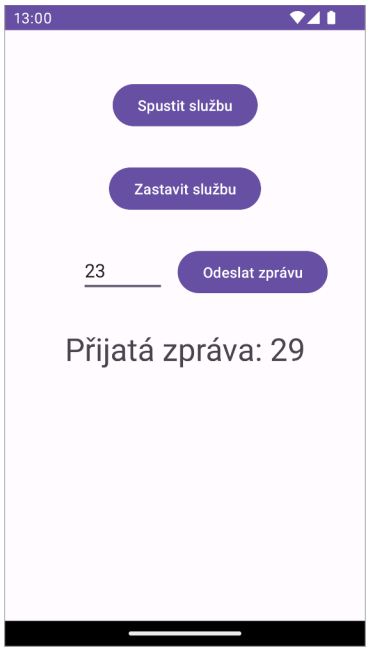
Receiver, který je tvořený naší třídou DataUpdateReceiver, zaregistrujeme při spuštění služby tak, že v již existující události onStartCommand() vytvoříme novou instanci třídy, přidáme ji filtr na náš typ zprávy a zavoláme metodu registerReceiver(). Příjem můžeme omezit pouze na zprávy z vlastní aplikace, tedy s konstantou RECEIVER_NOT_EXPORTED ve třetím parametru.

```
if (dataUpdateReceiver == null)
    dataUpdateReceiver = new DataUpdateReceiver();
IntentFilter intentFilter = new IntentFilter("zpravaZAktivity");
registerReceiver(dataUpdateReceiver, intentFilter, RECEIVER_NOT_EXPORTED);
```

Podobně budeme postupovat při odregistrování receiveru. K tomu využijeme událost onDestroy(), která už ve třídě také je a pouze přidáme volání metody unregisterReceiver().

```
if (dataUpdateReceiver != null)
        unregisterReceiver(dataUpdateReceiver);
```

Výsledná aplikace by po spuštění měla vypadat přibližně jako na obrázku 25.9.



Obrázek 25.9: Aplikace obousměrně komunikující se službou

26 Ikona aplikace

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 26_Ikona_aplikace
projekt Ikona1-4

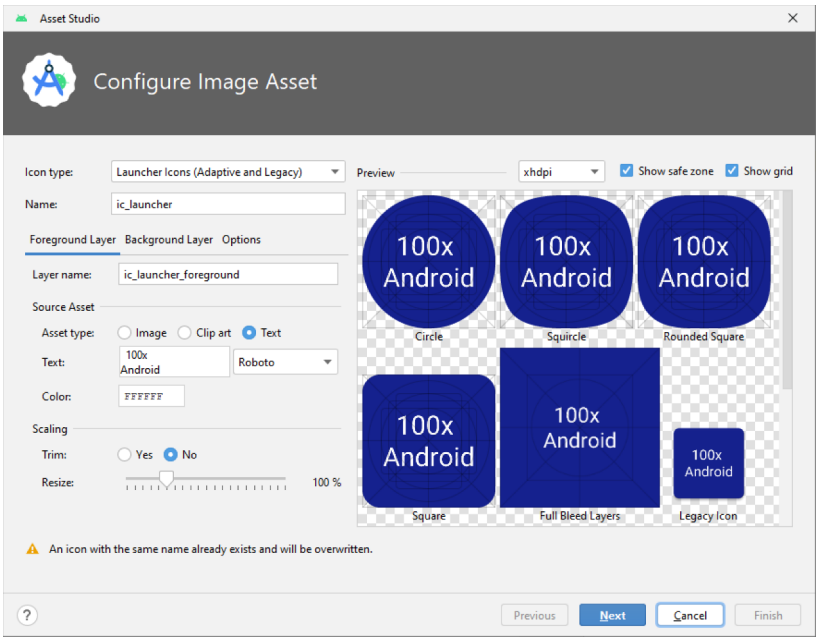
Dosud jsme u všech příkladů přecházeli jeden důležitý prvek aplikace. Její ikonku. Ačkoliv nemá na samotnou funkci pražádný vliv, je tím prvním, co má uživatel při spouštění aplikace na očích. Kvalitní ikona může pozornost uživatele přitáhnout, stejně jako ho špatně udělaná ikonka dokáže odradit dřív, než se dostane k samotné aplikaci. Proto bychom ji měli věnovat patřičnou pozornost.

Android přiřazuje aplikacím bez ikonky svoji vlastní. S tou jsme se již mnohokrát setkali. Teď si zkusíme udělat ikonu sami, zařadíme ji do projektu a vygenerujeme aplikaci, která ji bude využívat.

Nejjednodušší možností je použít již existující ikonu, kterých je na internetu k dispozici zdarma ke stažení obrovské množství. Pravděpodobně ale zjistíme, že žádná volně dostupná ikona není dost originální a naši aplikaci dostatečně nevystihuje. V takovém případě je nutné zapojit vlastní invenci, ikonku si navrhnout a vyrobit. Kdo má dostatečné tvůrčí nadání, může ji prostě nakreslit. Nebo se dá využít vhodně upravené fotografie.

V každém případě ale bude nutné vyjít z jedné ze tří základních typů ikon. První a nejjednodušší je textová ikona. Tedy textová by mělo být spíše v uvozovkách, jde o ikonu, která obsahuje barvu pozadí a krátký text, převedený do vektorových i bitmapových souborů tak, aby si instalátor mohl vybrat. Taková ikona nebude příliš nápaditá, ale pro aplikace určené omezenému okruhu uživatelů může stačit. Je to asi nejsnazší a nejrychlejší způsob, jak pořídit aplikaci ikonku. Druhou možností je vektorová ikona. Vytvořit ji zpravidla bývá dost pracné, ale vzhledem k tomu, že se u ní dá zaručit malá velikost a prakticky neomezená škálovatelnost co do rozměrů, tak je u profesionálních aplikací bezkonkurenčně nejrozšířenější. Poslední možností je bitmapová ikona, která může vzniknout třeba z otisku obrazovky, z fotky, nebo z naskenovaného obrázku.

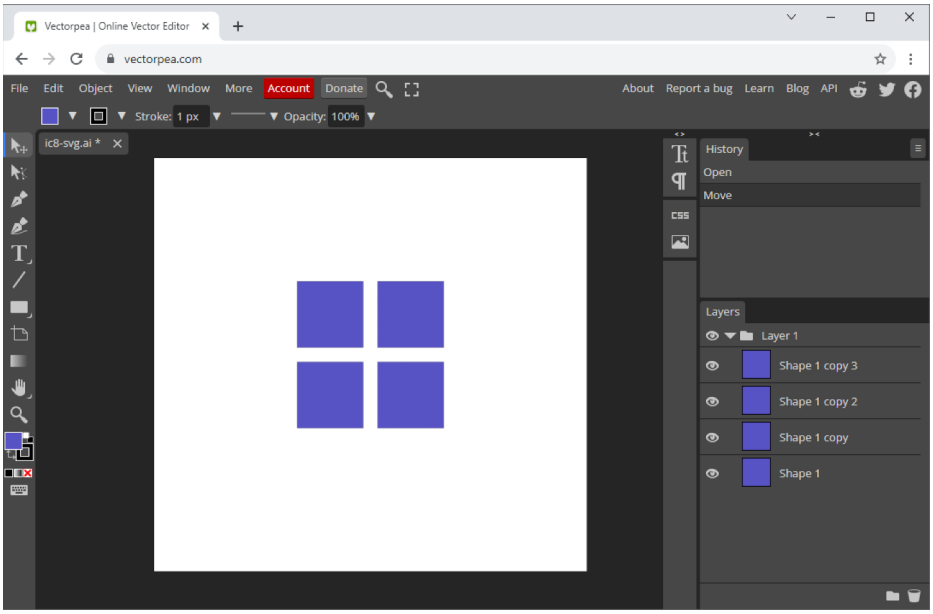
Textová ikona se vytváří přímo v Android Studiu, v okně projektu New → Image Asset. Otevře se editor, ve kterém se nastaví velikost a barva textu, barva pozadí a text se usadí do různých typů ikon. Průvodce následně ikonu zkonvertuje do vektorového xml souboru a také do bitmapových souborů typu WebP nebo PNG, pro různé velikosti ikony. Všechno je vcelku intuitivní a rychlé. Na vytvoření tak stačí doslova několik minut.



Obrázek 26.1: Textová ikona v editoru Android Studia

Vektorovou ikonu je nutné vytvořit v externím editoru. Musí být uložena ve formátu SVG nebo PSD. Pro tyto účely se dá použít některý z bezplatných online editorů na internetu, například Vectorpea. Nakreslit v něm vhodný obrázek nemusí být úplně jednoduché, chce to trochu cviku, ale lze si vypomoct už existujícími volně dostupnými vektorovými obrázky, takzvanými cliparty, staženými z internetu.

Pokud si ale vystačíme s jednoduchým motivem, neměl by být problém dosáhnout výsledku během pár minut. Jenom je potřeba dát pozor na umístění a okraje, ilustrace by měla být vystředěná v obou osách a kolem by mělo zůstat dost prostoru, aby byla čitelná jak na čtvercové, tak na zaoblené nebo kulaté ikoně, na kterých se okraje ořezávají.



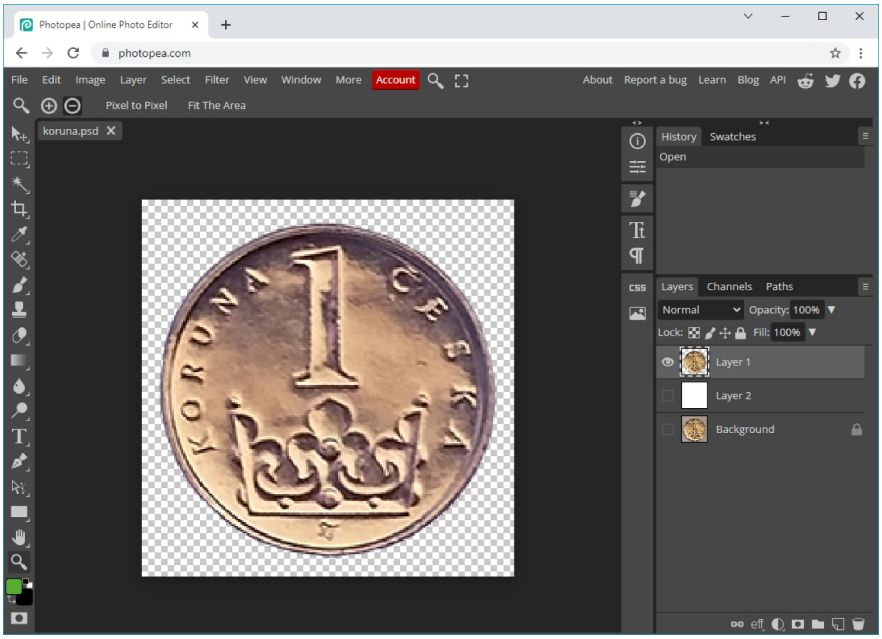
Obrázek 26.2: Online grafický editor Vectorpea

Po dokončení vyexportujeme ilustraci přes menu File → Export as → SVG do souboru formátu SVG. V Android Studiu, v projektovém okně, klikneme pravým tlačítkem myši do hierarchie projektu a volbou New → Vector Asset otevřeme okno průvodce pro import vektorové ikony. Zadáme cestu k souboru, nastavíme parametry a průvodce vytvoří ve složce app/res/drawable/ xml soubor s ikonou. A rovnou na něj udělá odkazy z konfiguračních souborů ../ic_launcher/ic_launcher.xml a ../ic_launcher_round/ic_launcher_round.xml.

Bitmapová ikonka je vhodná pro případy, kdy má obsahovat reálný obrázek, který nejde převést do vektorů. Pokud převést jde, tak je lepší dát přednost vektorové ikoně, s níž se Androidu pracuje přece jenom lépe.

Nejprve je nutné promyslet, co by ikonka vlastně měla obsahovat. Řekněme, že pro účely pokusu bude vyhovovat ikona zobrazující korunovou minci. Jednoduše si ji vyfotíme a v grafickém editoru upravíme tak, aby odpovídala potřebným parametrům. Jako grafický editor můžeme použít libovolný bitmapový editor, na který jsme zvyklí. Může to být jak profesionální Photoshop, tak zdarma dostupný Gimp, nebo třeba online editor Photopea. Obrázek, který bude předlohou pro ikonu by měl mít velikost alespoň 192x192 pixelů, což je rozměr největší Android ikony. Lepší je ale mít zdrojový obrázek několikanásobně větší a potom ho podle potřeby zmenšovat.

Příprava předlohy v online aplikaci Photopea:

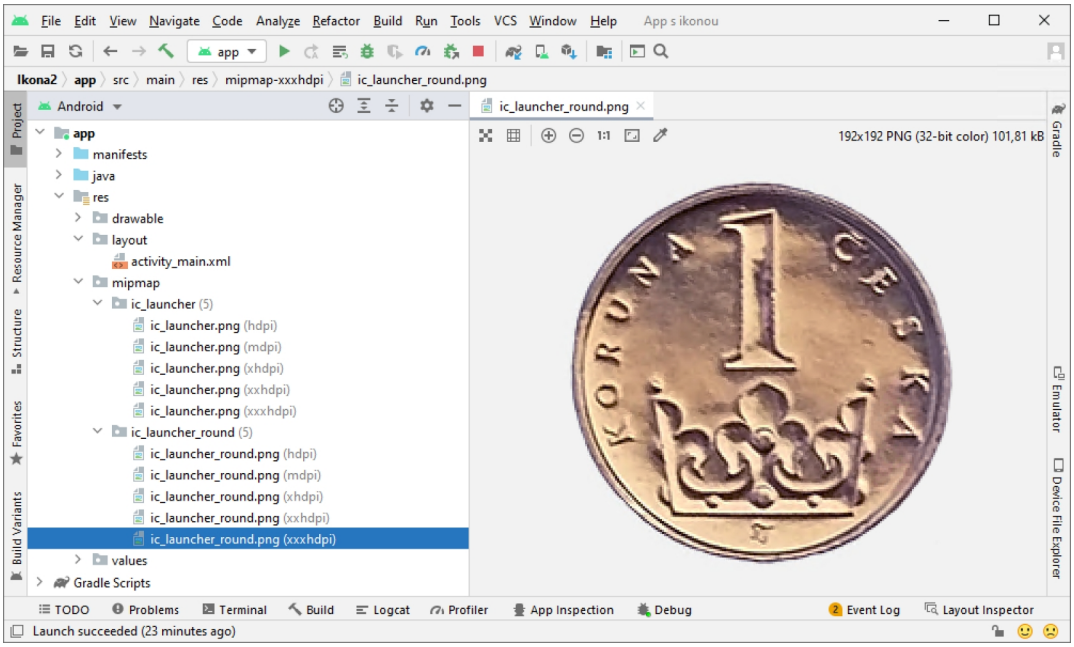


Obrázek 26.3: Online grafický editor Photopea

Jakmile je obrázek připravený, máme dvě možnosti, jak dál postupovat. První je jeho import do projektu, přičemž si Android Studio upraví velikost a umístění na ikonce. Udělá se to v projektovém okně, kliknutím pravým tlačítkem myši do hierarchie projektu a volbou New → Image Asset. Otevře se okno průvodce ve kterém přepneme Icon Type na ‘Launcher Icons’, nastavíme cestu k souboru a podle náhledu doladíme zbývající parametry ikony. Po zavření průvodce se výsledek objeví v okně projektu app → res → mipmap.

Druhá možnost je pracnější, ale poskytuje větší kontrolu nad výsledkem. Otevřeme si v okně projektu soubory ikon app → res → mipmap a klikneme pravým tlačítkem myší na složku mipmap. V kontextovém menu zvolíme Show in Explorer. Otevře se Windows Průzkumník se strukturou ikonek zařazených podle velikosti do jednotlivých složek. Tyto ikony v grafickém editoru

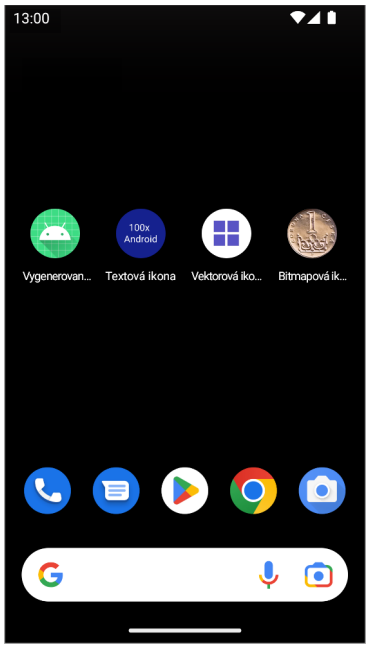
překreslíme našim motivem. Takto by měla ve struktuře projektu vzniknout skupina ikon, které můžeme zkontrolovat v Android Studiu:



Obrázek 26.4: Ikonky v Android Studiu

Pozor, je dobré upravit všechny ikony, které Android Studio nabízí. Některé verze Androidu podporují hranaté a jiné kulaté, na tabletech jsou ikonky větší než na telefonu atd. Pokud překreslíme všechny, nemusíme se bát, že se na některém zařízení nakonec ikona správně nezobrazí.

Potom už stačí projekt zkompilovat a spustit:



Obrázek 26.5: Ikonky aplikací na emulátoru. První ikona (v horní polovině displeje) je přednastavená Android Studiem, druhá je textová, třetí vektorová a čtvrtá bitmapová.

27 Lokalizace, překlad aplikace

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 27_Lokalizace
projekt Lokalizace

Jelikož Play Store poskytuje aplikace uživatelům z více než 150 zemí světa, je nutné při jejich programování myslet také na překlady do různých jazyků. Zpravidla je vše připraveno tak, že se aplikace sama spustí v jazyce, který je nastavený v Android zařízení jako primární. Pochopitelně pouze tehdy, pokud je příslušný překlad k dispozici. Jinak použije jazyk, který je v aplikaci nastavený jako základní.

Jazyk se nastavuje ve většině Android zařízení v menu Nastavení → Jazyky a zadávání → Jazyky. Po přepnutí se změní jak jazyk Androidu, tak i aplikací, pokud to umožňují.

Jak je to celé zařízené? Kouzlo se skrývá v projektové složce app/res/values. Jak jsme si již řekli, texty aplikace by měly být ukládány zásadně do souboru app/res/values/strings.xml. Odtud by si aplikace měla brát všechny textové řetězce, které uživateli zobrazuje, nebo, které kamkoliv zapisuje.

Text v resources souboru string.xml vypadá nějak takto:

```
<resources>
    <string name="nazevTextu">Příklad lokalizovatelného textu</string>
</resources>
```

Jenom pro jistotu si připomeňme, že k textům se dostáváme ve formuláři pomocí odkazu @string/nazevTextu a k obrázkům @drawable/nazevObrazku.

Příklad přístupu k textu z formuláře:

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/nazevTextu" />
```

Tato zjednodušená forma TextView na formuláři zobrazí textové pole s předvyplněným textem z resources @string/nazevTextu. V kódu se k lokalizovanému textu dostaneme přes kontext:

```
String text = context.getResources().getString (R.string.nazevTextu);
```

V případě textů můžeme v resources využít možnost snadného formátování. Využívají se k tomu pomocné identifikátory v textu, které zastupují obsah dodatečně doplněný pomocí parametrů. Formát je například %1$s - vždy za znakem % je pořadové číslo parametru, následované znakem $ a datovým typem. Příklad takového resource:

```
<resources>
    <string name="testovaciZprava">Seznam %1$s obsahuje %2$d záznamů</string>
</resources>
```

Text obsahuje dva formátovací parametry – první „%1$s“ je zástupce typu string a druhý „%2$d“ je zástupcem celého čísla. Použití:

```
String nazev = "xxx";
int počet = 10;
String zprava = getString(R.string.testovaciZprava, nazev, pocet);
```

V proměnné zprava bude text „Seznam xxx obsahuje 10 záznamů“. Výhodou tohoto formátování je, že při překladu do cizího jazyka lze použít jiný slovosled, aniž bychom se museli obávat, že se hodnoty doplněné do textu vzájemně prohodí nebo zařadí na nesprávné místo.

Pokud jsou tedy texty připravené v resources, je překlad poměrně snadnou záležitostí. Otevřeme si soubor strings.xml a v pravém horním rohu okna je text ve formě odkazu Open editor. Klikneme na něj a otevře se okno pro překlady.

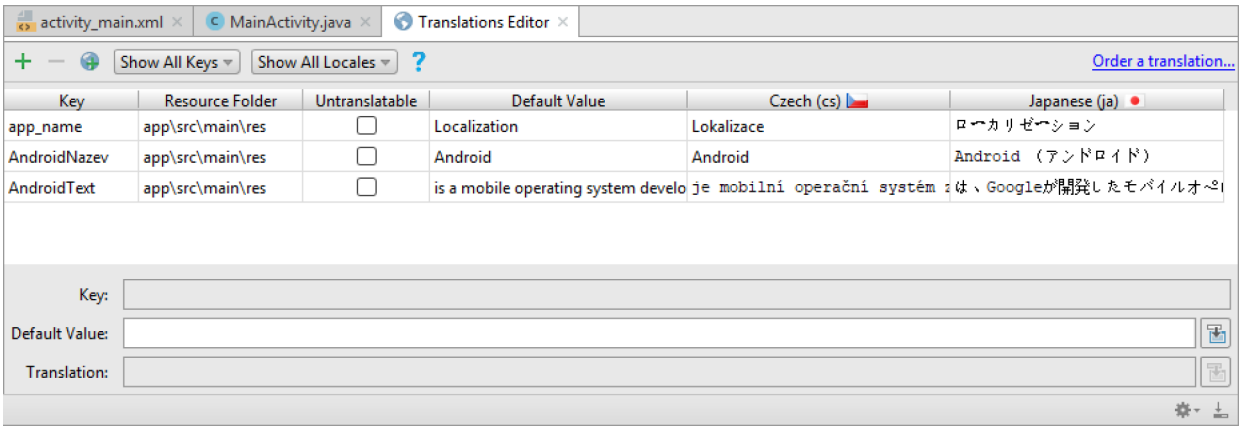
V okně pro překlady je nejdříve jenom jeden jazyk, ve sloupci Default Value. Je dobré rozmyslet, jaký jazyk to má být. Pokud bude aplikace používána pouze v českém prostředí, může to být čeština, ale při jakékoliv vyšší ambici se doporučuje mít jako základní jazyk angličtinu. Nejen že je nativním jazykem Play Store, ale je současně bezkonkurenčně nejrozšířenější mezi uživateli Androidu. Tento přednastavený jazyk bude použit pokaždé, když nebude k dispozici překlad pro konkrétní jazyk.

Zde je ještě nutné poznamenat, že překlady mají svoje identifikátory, které se zapisují do názvu složky, v níž se nachází soubor s texty. Identifikátory vychází z normy ISO 639-1, angličtina má přidělený kód en, čeština cs, němčina de a tak dále. Čili český překlad bude umístěný v souboru app/res/values-cs/string.xml. Výchozí texty přitom zůstanou uložené v souboru app/res/values/strings.xml.

Kromě kódu jazyka můžeme připojit ještě kód oblasti. V případě češtiny je to jednoduché, používá se jenom v jedné úřední verzi, ale taková angličtina, ta už je pro Kanadu jiná než pro Británii. Proto využijeme další normu, ISO 3166-2. Ta říká, že se pro region Česká republika používá zkratka CZ. Pro Kanadu CA, pro Spojené království GB a tak dále. Kód jazyka včetně regionu je pro Českou republiku tedy cs-CZ, v Androidu ještě pro jistotu rozšířený na cs-rCZ, aby byl zřejmý rozdíl mezi jazykem a regionem.

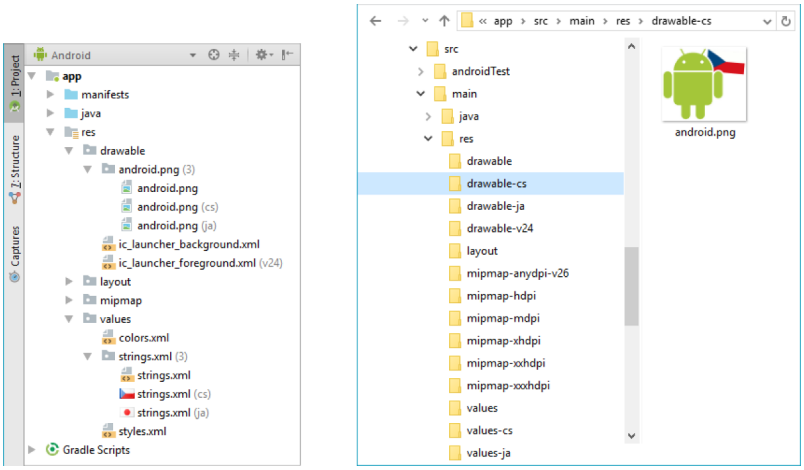
Texty se tedy uloží do souboru app/res/values-cs-rCZ/string.xml. Pro kanadskou angličtinu by to byl soubor app/res/values-en-rCA/string.xml.

Překlad vytvoříme kliknutím na ikonku zeměkoule, kde vybereme jazyk a případně i region. Editor automaticky přidá sloupec, do kterého můžeme vepsat překlad.



Obrázek 27.1: Editor Android Studia pro lokalizaci textů

Struktura souborů v okně projektů se rozšíří o verze s překlady, je to ale zjednodušené znázornění, na disku bude struktura tak, jak jsme si popsali. Výhoda ukládání do jednotlivých souborů je zřejmá – vždycky můžeme vzít xml soubor daného jazyka a předat ho někomu k překladu, aniž bychom posílali celý balík jazyků.

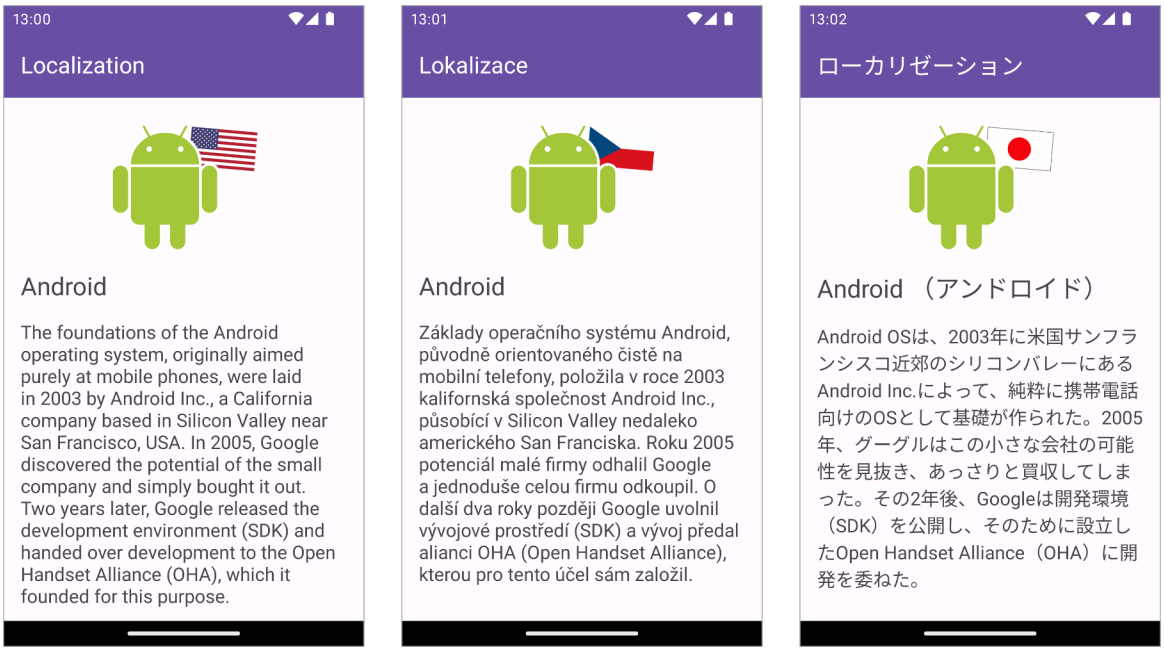


Obrázek 27.2: Struktura lokalizovaných souborů v okně projektu a na disku počítače

Takto si snadno, bez jakéhokoliv programování, pouze klikáním, můžeme vytvořit aplikaci, která bude akceptovat překlady. Totéž, co jsme udělali s texty můžeme udělat i s obrázky, ale tam už nemáme k dispozici editor a proto musíme pracovat přímo se soubory. Systém je stejný, přednastavené obrázky ukládáme do složky app/res/drawable/, české do app/res/drawable-cs/. Obrázky musí mít stejný název, Android Studio je opět zobrazí ve zjednodušené formě, stejně jako tomu bylo v případě textů.

Stejným způsobem můžeme lokalizovat rovněž barvy, styly, ikony, nebo celé formuláře.

Po přepnutí jazyka zařízení se automaticky překreslí text a obrázky aplikace. Na prvním obrázku je angličtina, přednastavená pro všechny nepodporované jazyky. Na druhém a třetím jsou jazyky, do kterých byla aplikace přeložena.



Obrázek 27.3: Aplikace s lokalizovaným obrázkem a texty, přepnutá do různých jazyků

28 Styly a témata

V Androidu má vývojář několik možností, jak nastavit vzhled jednotlivých komponent uživatelského rozhraní. Tím nejjednodušším je přímé nastavení vlastnosti použité komponenty pomocí jejích vlastních atributů. Tento způsob konfigurace komponent má ale svá úskalí. Pro větší množství komponent roste pracnost a množství kódu, které je k jejich popsání potřeba. Kód se navíc často zbytečně opakuje. Ke všemu je potom dost obtížné jednorázově upravit vzhled komponent, natož ho dynamicky přizpůsobovat potřebám uživatele.

Naštěstí existují způsoby, jak měnit vzhled komponent, aktivit, nebo rovnou celé aplikace hromadně. Pro tyto účely se používají styly a témata. S jejich pomocí je možné, bez použití velkého množství kódu, parametricky přiřadit prvku požadované vizuální vlastnosti nebo barevnou kombinaci. V případě stylu je k dispozici kolekce atributů, jako velikost, řez písma, okraje, barva popředí a pozadí, odsazení a mnoho dalších. Styl potom stačí komponentě přiřadit a ta má rázem vzhled, který bychom jinak museli nastavovat každému prvku pomocí mnoha samostatných atributů.

Podobně je to s tématy. Ta se nevztahují pouze ke komponentě, ale rovnou ke vzhledu aktivity nebo celé aplikace. Pochopitelně s dopadem i na jednotlivé komponenty, které bychom jinak mohli mít popsané stylem.

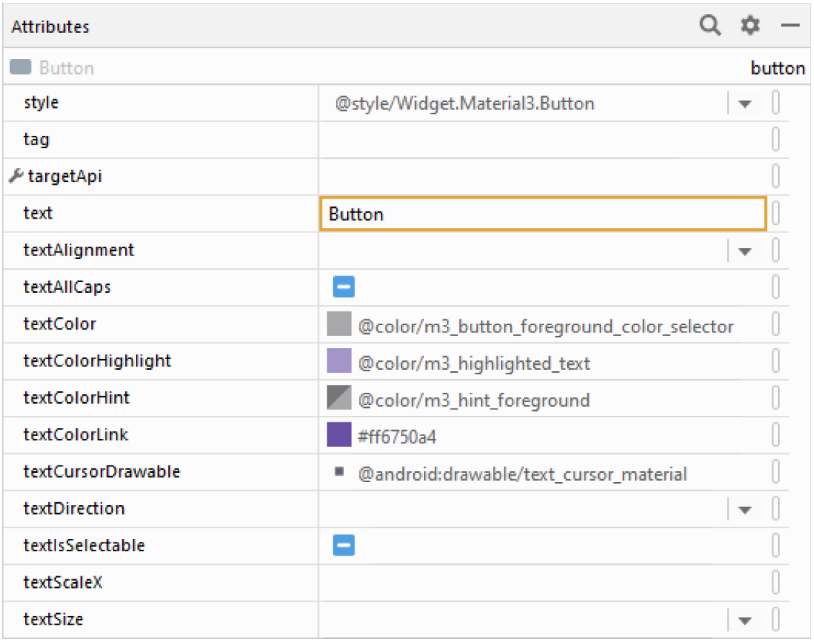
Styly a témata se vytváří podobně. V obou případech jde o skupinu vybraných atributů, které mají název a k němu přiřazenou hodnotu. Název každého atributu je dopředu daný, hodnotu si nastavuje vývojář. Které atributy budou v definici stylu nebo tématu použity, určuje rovněž vývojář.

Obecně platí, že vzhled aplikace nebo aktivity lze nastavit tématem. Pokud je navíc na některé komponentě použitý styl, má před tématem přednost. A když má komponenta navíc přímo nastavené atributy vzhledu, mají tyto přednost jak před tématem, tak i před stylem.

28.1 Styly

Vývojář se snadno dostane do situace, kdy má na formuláři pod sebou několik textových polí, z nichž každé má spoustu parametrů. A kdy se tyto parametry pro jednotlivá pole téměř neliší, pouze se pořád dokola opakují. Ve vizuálním editoru Android Studia to ještě není takový problém, ale při editování přímo v XML kódu formuláře už je situace o poznání horší. Velké množství atributů nesvědčí přehlednosti a nepřispívá ani k rychlosti vykreslování hotového layoutu. Případná změna vzhledu jednoho typu komponenty potom znamená všechny je projít a u každé změnit ten stejný atribut. Tyto problémy pomáhají řešit styly.

Styl je skupina vlastností, popisujících vzhled komponenty. Každá komponenta uživatelského rozhraní má přiřazený styl. Ten si lze v XML formátu prohlédnout v SDK. Snadněji se k němu ale dá dostat v designéru Android Studia, u konkrétní komponenty v okně vlastností. Parametry, včetně názvu použitého stylu jsou tam zobrazeny světlým písmem, viz obrázek 28.1.



Obrázek 28.1: Vlastnosti tlačítka, nastavené stylem

Na obrázku vlastností tlačítka je nastavený pouze text „Button“. Všechny ostatní, světlým písmem uvedené vlastnosti, jsou převzaté ze stylu Widget.Material3.Button (viz řádek „style“). Vývojář může samozřejmě tyto parametry přenastavit, pokud je ale nechá beze změny, budou použity právě tyto vlastnosti dané stylem.

V Android Studiu jsou styly uložené v souboru app/res/values/styles.xml. Umístění a pojmenování není povinné, teoreticky by šlo dát styl do souboru s vlastním názvem nebo do kteréhokoliv jiného existujícího souboru v resources, třeba do colors.xml. Ale historicky se zažila konvence ukládání stylů právě ve styles.xml.

Jak styl použít? Existují dva základní způsoby. Buď se vytvoří úplně nový styl a ten se přiřadí komponentě, nebo se nový styl oddědí od již existujícího, upraví a opět se přiřadí konkrétní komponentě. Obě varianty si ukážeme na příkladech.

28.1.1 Styly, 1. příklad

V Android Studiu vytvoříme nový styl, který bude definovat vzhled komponenty uživatelského rozhraní, konkrétně tlačítka. Poté ho přiřadíme tlačítkům na formuláři. Následně budeme mít možnost měnit vzhled tlačítek pouhou úpravou parametrů stylu.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 28_Styly_a_temata
projekt Styly1

Co je potřeba udělat?

- vytvořit aplikaci s tlačítkem na formuláři
- vytvořit nový styl, který bude popisovat vzhled tlačítka
- přiřadit nový styl tlačítku na formuláři

Založíme si v Android Studiu nový projekt s prázdnou aktivitou. Můžeme použít Linear Layout, který umožní umístit prvek bez velkého množství parametrů, takže bude později lépe vidět dopad použitého stylu. Do formuláře vložíme tlačítko. Jeho XML popis může být minimalistický, stačí, aby mělo nastavenou dynamickou velikost a text:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Tlačítko 1" />
```

Takové tlačítko samozřejmě nebude nijak zajímavé. Bude se krčit v horním rohu formuláře, provedené v barvách prostředí a celkově na něm nebude nic pozoruhodného. Můžeme si pod něj umístit další, které vylepšíme nastavením jeho jednotlivých parametrů. Ty najdeme buď v online dokumentaci, nebo v Android Studiu, v okně vlastností. Tlačítko může v XML vypadat třeba takto:

```
<Button
    android:layout_width="200dp"
    android:layout_height="70dp"
    android:layout_gravity="center"
    android:layout_marginStart="30dp"
    android:layout_marginTop="30dp"
    android:layout_marginEnd="30dp"
    android:layout_marginBottom="30dp"
    android:backgroundTint="#FF000000"
    android:text="Tlačítko 2"
    android:textColor="#FFFFFF"
    android:textSize="20sp" />
```

Teď jsou na formuláři dvě tlačítka, první, které je v systémovém provedení bez dalších úprav a druhé, s nastavenou velikostí, odsazením, barvou textu i pozadí.

Pokud Android Studio samo nevygenerovalo soubor app/res/values/styles.xml, tak si ho stejně jako v předchozím příkladu vytvoříme pomocí kontextového menu v okně projektu New → Values Resource File. Vložíme do něj element <resources> a do něj další element, <style...>, s vhodným názvem stylu, například „StylTlacitko“. Do elementu <style...> vložíme položky, které budou pojmenovány stejně jako standardní atributy tlačítka. Položek bude tolik, kolik jich budeme potřebovat k popisu vzhledu tlačítka. Můžeme použít prakticky všechny parametry doplněné do druhého tlačítka na formuláři a případně i cokoliv dalšího, co uznáme za užitečné. Můžeme nastavit i text tlačítka.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <style name="ButtonStyle">
        <item name="android:layout_width">200dp</item>
        <item name="android:layout_height">70dp</item>
        <item name="android:layout_gravity">center</item>
        <item name="android:layout_marginStart">30dp</item>
        <item name="android:layout_marginTop">30dp</item>
        <item name="android:layout_marginEnd">30dp</item>
        <item name="android:layout_marginBottom">30dp</item>
        <item name="android:backgroundTint">#FF000000</item>
        <item name="android:text">Tlačítko</item>
        <item name="android:textColor">#FFFFFF</item>
        <item name="android:textSize">20sp</item>
```

```
    </style>

</resources>
```

Tím máme vytvořený styl ButtonStyle. Do XML popisu tlačítka na hlavním formuláři můžeme nyní přidat třetí tlačítko. U toho už nebudeme upravovat jednotlivé parametry, zadáme pouze zobrazovaný text a pomocí atributu style přiřadíme nově vytvořený styl:

```
<Button
    android:text="Tlačítko 3"
    style="@style/ButtonStyle" />
```

Takto jednoduše popsané tlačítko bude mít stejné vizuální vlastnosti jako to nad ním. Dokonce ani není nutný atribut *text*, i ten můžeme vynechat. V takovém případě se použije text ze stylu, tedy „Tlačítko“. Nyní už není problém vyrobit další tlačítka, která budou mít i bez nastavování parametrů stejný vzhled. Ten navíc můžeme jednotně měnit úpravou stylu. Stylů můžeme mít pro tyto účely víc a vždy použít ten, který do vizuální stránky formuláře lépe zapadá.

Výsledná aplikace by po spuštění měla vypadat přibližně jako na obrázku 28.2. Nahoře je tlačítko bez vlastního stylu, pouze s vlastním textem a rozměry. Pod ním je tlačítko s podrobně nastavenými vlastnostmi. A nakonec třetí tlačítko, které má nastavený pouze text a připojený styl.

Obrázek 28.2: Dvě tlačítka nakonfigurovaná pomocí parametrů a třetí, vizuálně nastavené pomocí stylu

28.1.2 Styly, 2. příklad

Styly mají ještě jednu důležitou vlastnost a tou je dědičnost. V definici stylu lze použít definici jiného stylu a pouze doplnit další parametry. Může tak vzniknout určitá hierarchie stylů, kdy lze volbou vhodného stylu kombinovat grafické provedení komponent na formuláři.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 28_Styly_a_temata
projekt Styly2

Vyzkoušíme si to opět na příkladu s tlačítky. Nadefinujeme jednoduchý základní styl. Další styly oddědí jeho vlastnosti a přidají nové. Tlačítka potom budou mít různý vzhled, ale některé jejich vlastnosti půjde ovlivňovat úpravou hierarchicky nejvýše umístěného stylu a jiné pomocí dílčích stylů.

Co je potřeba udělat?

- vytvořit základní styl, který bude popisovat vzhled tlačítka

- vytvořit další styly, které zdědí vlastnosti základního stylu

- na formulář umístit tlačítka s přiřazenými styly

Založíme v Android Studiu nový projekt s prázdnou aktivitou. Pro jednoduchost opět použijeme Linear Layout.

Najdeme nebo vytvoříme soubor se styly app/res/values/styles.xml a definujeme základní styl, který si pojmenujeme třeba „StylTlacitko“. Nastavíme v něm všechny atributy, které by tlačítko mít mělo, tj. rozměry, umístění, okraj, velikost textu:

```
<style name="StylTlacitko">
    <item name="android:layout_width">200dp</item>
    <item name="android:layout_height">70dp</item>
    <item name="android:layout_gravity">center</item>
    <item name="android:layout_margin">30dp</item>
    <item name="android:textSize">20sp</item>
</style>>
```

Když tento styl přiřadíme tlačítku, bude vizuálně upravené, ale v barvách daných přednastaveným Android stylem. Do formuláře activity_main.xml umístíme tlačítko a hned mu pomocí atributu style přiřadíme nový styl „StylTlacitko“. Už v grafickém designéru Android Studia uvidíme, jak se vzhled tlačítka změnil oproti tlačítku bez stylu.

Barevné provedení tlačítka můžeme upravit buď přímým nastavením jeho vlastností, nebo můžeme použít další styl, který zdědí vlastnosti předchozího a rozměry s umístěním k němu doplní:

```
<style name="StylTlacitkoCervene" parent="@style/StylTlacitko">
    <item name="android:backgroundTint">#FFFF0000</item>
    <item name="android:textColor">#FFFFFFFF</item>
</style>
```

Klíčová je vlastnost atributu „parent“. Ta říká, že má styl převzít, neboli oddědit, všechny vlastnosti přiřazeného stylu, tedy stylu „StylTlacitko“. Tlačítko s tímto stylem tedy bude mít definovaný rozměr, umístění, okraj, velikost textu a navíc červenou barvu a bílý text.

Zde je nutné poznamenat, že je možné oddědit pouze jeden styl. Nelze tedy v jednom stylu nastavit rozměry, ve druhém umístění a toto třetím stylem sloučit.

Můžeme hned vytvořit další styl, který zdědí vlastnosti stylu „StylTlacitka“ a přidá k nim novou barvu:

```
<style name="StylTlacitkoZlute" parent="@style/StylTlacitko">
    <item name="android:backgroundTint">#FFFFFF00</item>
```

```
    <item name="android:textColor">#FF000000</item>
</style>
```

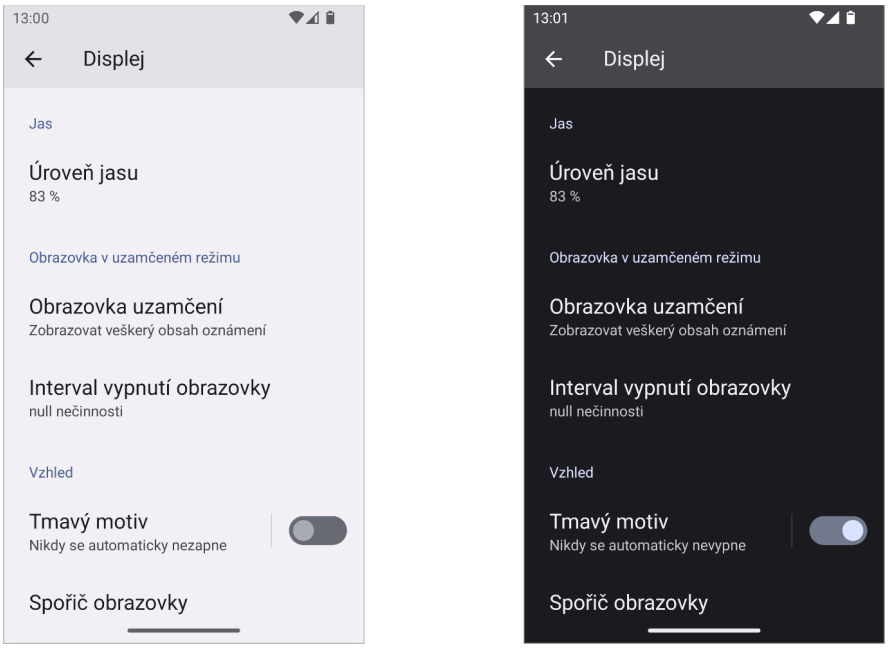
Tlačítko s tímto stylem bude mít opět definovaný rozměr, umístění, okraj, velikost textu a navíc žlutou barvu a tmavý text.

Do formuláře vložíme další dvě tlačítka a přiřadíme jim nové styly „StylTlacitkoCervene“ a „StylTlacitkoZlute“. Výsledná aplikace by po spuštění měla vypadat jako na obrázku 28.3. Nahoře je tlačítko se základním stylem a pod ním dvě tlačítka se styly definujícími barevné provedení a dědícími ostatní parametry ze základního stylu.

Obrázek 28.3: Tlačítko s přiřazeným základním stylem a dvě tlačítka se styly které dědí a rozšiřují základní styl

28.2 Témata

S tématy (někdy se také používá označení motiv) se uživatel setká už v nastavení systému Android. V části „Displej“ může přepínat prostředí Androidu na světlé nebo na tmavé, viz obrázek 28.4. Podle toho, co vybere, je následně nastaven nejen vzhled systému, ale také vzhled všech aplikací, které tmavé a světlé téma podporují.



Obrázek 28.4: Přepínání světlého a tmavého tématu (motivu)

Nastavení témat najdeme v Android Studiu ve složce resources. Už při založení aplikace tam Android Studio vytvoří dva soubory – themes.xml a themes.xml (night). Ty odpovídají světlému a tmavému tématu. Pokud je využijeme a nepřepíšeme barvy komponent jiným způsobem, bude se aplikace barevně přizpůsobovat přepínači v nastaveních displeje, resp. tomu, co je v souborech nastavené.

Souborů s tématy může být v projektu víc. Které z nich se použije je uvedeno v manifestu aplikace. Téma může být tímto způsobem nastavené pouze jedno. Přesněji řečeno – jedno, ale ve dvou verzích, světlé a tmavé. Příslušný parametr, theme, je v manifestu součástí definice aplikace:

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.nazev_aplikace" />
```

Některé atributy, které můžeme pomocí témat nastavovat, jsou uvedeny v tabulce 28.1. Atributy, které v názvu obsahují „On” (colorOnPrimary, colorOnError,..) jsou barvy použité na podkladu barvy ekvivalentního atributu (colorPrimary, colorError,..). Například text na tlačítku je colorOnPrimary, podklad tlačítka colorPrimary. Tyto dvojice by tedy měly být vůči sobě kontrastní. Téma nemusí nutně obsahovat všechny existující atributy. Ten, který není v tématu použit, zůstává nastavený na systémovou hodnotu.

Název atributu	Význam
colorPrimary	Základní a nejčastěji použitá barva, vyskytuje se u většiny komponent
colorPrimaryVariant	Tmavší (u tmavého tématu naopak světlejší) alternativa colorPrimary

Název atributu	Význam
colorSecondary	Doplňková barva. Spolu s colorPrimary tvoří barevný ‚brand‘.
colorSecondaryVariant	Tmavší (u tmavého tématu naopak světlejší) alternativa colorSecondary
colorBackground	Barva pozadí, za všemi ostatními prvky na displeji
colorSurface	Barva povrchu prvků, jako jsou záložky nebo menu
colorError	Barva varovných a chybových hlášení
colorOnPrimary	Barva, která se použije na podkladu colorPrimary. Obvykle barva textu nebo ikon.
colorOnSecondary	Barva, která se použije na podkladu colorSecondary
colorOnBackground	Barva prvku umístěného na pozadí
colorOnSurface	Barva prvků umístěných na povrch prvků, jako jsou záložky nebo menu
colorOnError	Barva, která se použije na podkladu colorError. Obvykle ikony a text hlášení.
android:statusBarColor	barva stavového řádku

Tabulka 28.1: Základní atributy, které je možné použít v definici tématu

28.2.1 Témata, 1. příklad

Zkusíme vytvořit novou aplikaci, které změníme její přednastavené téma. Na formulář aplikace vložíme několik komponent, na nichž bude vidět, co lze změnou tématu ovlivnit.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 28_Styly_a_temata
projekt Temata1

Co je potřeba udělat?

- vytvořit aplikaci s přednastaveným tématem
- přidat do souboru colors.xml definice vlastních barev
- upravit v xml souboru nastavení tématu

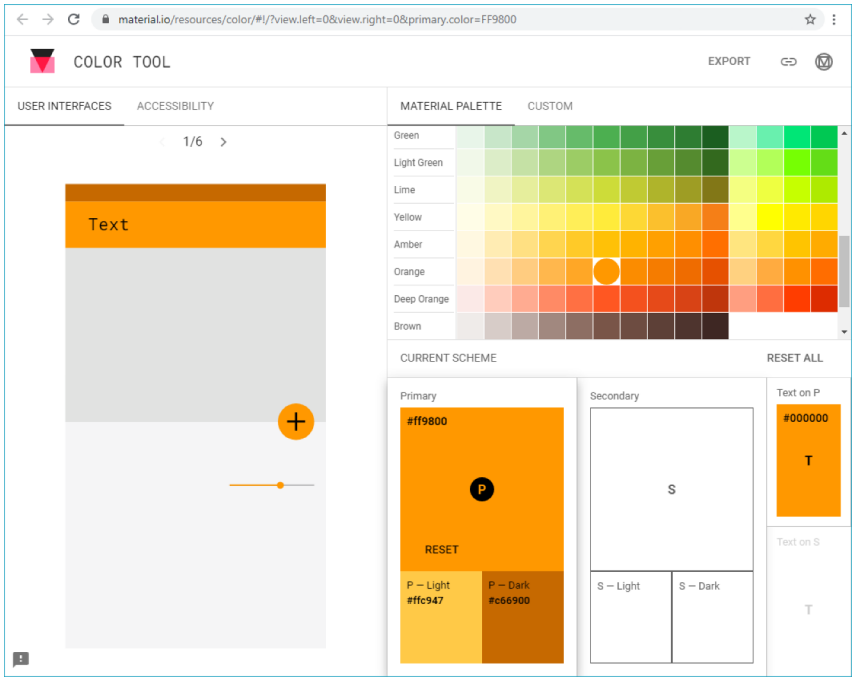
Založíme si v Android Studiu nový projekt s prázdnou aktivitou. Android Studio se samo postará o to, aby v resources byly dva soubory s tématy, na které odkazuje atribut android:theme v manifestu. Do formuláře vložíme prvky, u kterých se dá očekávat změna vizuálního provedení: toolbar, tlačítko, přepínač, posuvník a případně další.

Definici témat najdeme v projektovém adresáři app/res/values/themes/ ve dvou souborech themes.xml. První XML soubor otevřeme, obsahuje přednastavené hodnoty v sedmi základních atributech. Každý z nich se odkazuje na definici barvy v souboru colors.xml.

Abychom mohli použít nové barevné schéma, musíme si definovat vhodné barvy. Mohli bychom je nastavit intuitivně, ale pravděpodobně by výsledek na první pokus nebyl dokonalý. Na internetu se dají najít nástroje, kterým stačí nastavit pouze základní barvu a ony dopočítají všechny ostatní tak, aby byl výsledek barevně vhodně sladěný a v souladu s definicí barevných schémat Android.

Jednou z takových internetových aplikací, vytvořenou přímo Google vývojáři, je Color Tool na adrese https://material.io/resources/color/

Ať už v Color Tool nebo v jiné aplikaci, zvolíme hlavní barvu prostředí a necháme si najít ostatní pomocné barvy. Viz obrázek 28.5.



Obrázek 28.5: Vyhledání kódů barev pro oranžové téma v aplikaci Color Tool

Pro začátek si vykopírujeme kódy barev Primary, Primary – Light a Primary – Dark. Z nich si v souboru colors.xml vytvoříme nové barvy, které si vhodně pojmenujeme. A k tomu přidáme ještě černou a bílou, ty se nikdy neztratí a hodí se zejména pro kontrastní barvu textu ve světlém nebo naopak tmavém tématu:

```
<color name="oranzova">#FFFF9800</color>
<color name="oranzovatmava">#FFC66900</color>
<color name="oranzovasvetla">#FFFFC947</color>
<color name="cerna">#FF000000</color>
<color name="bila">#FFFFFFFF</color>
```

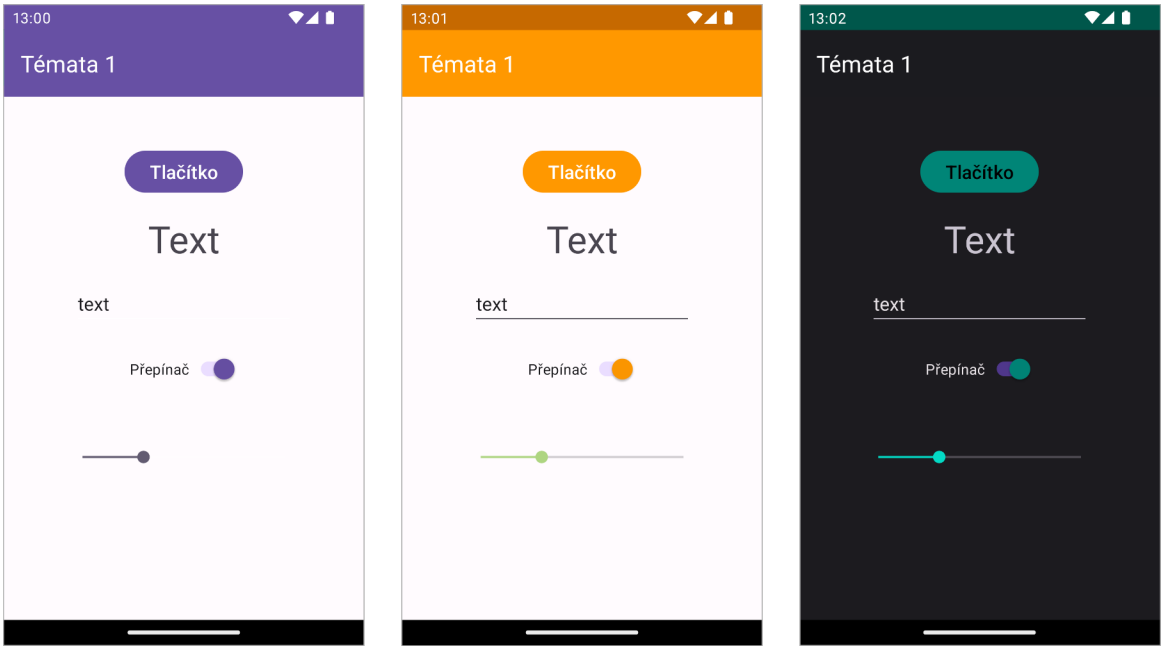
Původní barvy v souboru není nutné mazat ani přepisovat, stačí když svoje tři nové jednoduše přidáme na konec. Tím je soubor colors.xml připraven. Otevřeme themes.xml a nastavíme tři nejdůležitější barvy. Základní colorPrimary, její tmavší variantu colorPrimaryVariant a barvu která se použije přes základní, tj. zejména pro text, colorOnPrimary:

```
<item name="colorPrimary">@color/oranzova</item>
<item name="colorPrimaryVariant">@color/oranzovatmava</item>
<item name="colorOnPrimary">@color/bila</item>
```

Abychom dosáhli komplexní změny, měli bychom upravit také colorSecondary atributy. Ty ale zatím pro jednoduchost, stejně jako ostatní atributy, nechme v původním nastavení.

Aplikaci spustíme na emulátoru nebo v zařízení. Výsledek by měl vypadat jako na obrázku 28.6. Na prvním otisku displeje je vzhled komponent před úpravou themes.xml (v některých verzích Android Studia může být vygenerovaný jako modrý, v jiných zelený), na druhém je po úpravě

themes.xml a na třetím je po přepnutí systému do tmavého režimu. Vzhled pro tmavý režim bychom mohli stejným způsobem změnit v souboru themes.xml (noční).



Obrázek 28.6: Standardní téma, upravené téma a standardní tmavé téma

28.2.2 Témata, 2. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 28_Styly_a_temata
projekt Temata2

Ne vždy si vývojář vystačí s přepínáním dvou barevných témat podle nastavení systému. Někdy se hodí dát uživateli zvolit barevné téma podle jeho vlastních preferencí. Na příkladu si ukážeme jak se dají přepínat přednastavená témata v aplikaci. Cílem není udělat komplexní barevná témata, pouze nastavíme základní atributy pro několik denních témat, která bude uživatel moct přepínat.

Co je potřeba udělat?

- založit aplikaci s vhodnými prvky na formuláři hlavní aktivity
- přidat do souboru colors.xml definice barev pro další témata
- přidat nová témata
- nastavit v hlavní aktivitě přepínání témat

Založíme si v Android Studiu nový projekt s prázdnou aktivitou. Android Studio se opět samo postará o to, aby v resources byly dva soubory s tématy na které odkazuje atribut android:theme v manifestu. Do formuláře vložíme prvky u kterých se dá očekávat změna vizuálního provedení: tlačítko, přepínač, posuvník a případně další.

V souboru app/res/values/colors.xml nadefinujeme základní barvy pro několik různých témat. Stejně jako v předchozím příkladu, i zde je vhodné použít aplikaci která s vhodným sladěním barev pomůže. Zde jsou, kromě obvyklé černé a bílé, definice barev pro tři témata v různých odstínech šedé:

```
<color name="seda1">#FFE0E0E0</color>
<color name="seda1svetla">#FFFFFFFF</color>
<color name="seda1tmava">#FFAEAEAE</color>

<color name="seda2">#FF9E9E9E</color>
<color name="seda2svetla">#FFCFCFCF</color>
<color name="seda2tmava">#FF707070</color>

<color name="seda3">#FF616161</color>
<color name="seda3svetla">#FF8E8E8E</color>
<color name="seda3tmava">#FF373737</color>

<color name="cerna">#FF000000</color>
<color name="bila">#FFFFFFFF</color>
```

Takto definované barvy použijeme v souborech s definicemi témat. Ve složce app/res/values/ vytvoříme nové resources soubory, například s názvy themes1.xml, themes2.xml a themes3.xml. Pro zjednodušení můžeme do každého z nich nakopírovat hodnoty z themes.xml, který vygenerovalo Android Studio. V nových souborech je potřeba změnit název stylu. Postupně v nich také nastavíme vždy jednu z barevných kombinaci. V prvním souboru to můžou být tyto hodnoty:

```
<style name="Theme.Tema1"
                  parent="Theme.MaterialComponents.DayNight.DarkActionBar">
    <item name="colorPrimary">@color/seda1</item>
    <item name="colorPrimaryVariant">@color/seda1tmava</item>
    <item name="colorOnPrimary">@color/cerna</item>

    ...

</style>
```

Doplňkové barvy teď nastavovat nemusíme, takže ostatní atributy zatím necháme v původním stavu. Analogicky upravíme i zbývající soubory. Kdybychom se rozhodli vytvořit také nová noční témata, museli bychom na disku počítače rozdělit soubory s tématy do složek values a values-night.

Tím jsou témata připravena. Není těžké vymyslet hned několik způsobů, jak je přepínat. Základem je volání systémové metody setTheme(), která se o vše podstatné postará. Důležité je, aby byla zavolána ještě před vykreslením formuláře, tedy před voláním setContentView() v události onCreate().

```
@Override
protected void onCreate(Bundle savedInstanceState) {

    setTheme(R.style.Theme_Tema1);

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```

Se změnou stylu před vykreslením si vystačíme pouze když otevíráme nový formulář. Změna tématu je ale potřeba i v okamžiku kdy uživatel mění jeho nastavení. Toto se nejsnáze řeší voláním instrukce pro nové nahrání formuláře recreate().

Způsobů, jak témata přepínat, lze vymyslet mnoho. Jedna z dalších možností - v aktivitě MainActivity si vytvoříme pomocnou statickou proměnnou iTema, ve které bude uloženo číslo aktuálně použitého tématu. Statickou proto, aby si uchovala hodnotu i po opakovaném načtení formuláře. Podle této hodnoty potom v onCreate() pomocí setTheme() témata přepínáme. Prostřednictvím tlačítek na formuláři voláme události onClick(), které nastaví proměnnou iTema a znovu načtou formulář.

```
private static int iTema=1;

@Override
protected void onCreate(Bundle savedInstanceState) {

    if (iTema == 1) {
        setTheme(R.style.Theme_Tema1);
    }
    else if (iTema == 2) {
        setTheme(R.style.Theme_Tema2);
    }
    else {
        setTheme(R.style.Theme_Tema3);
    }

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}

public void onClickTema1 (View view) {
    iTema = 1;
    this.recreate();
}

public void onClickTema2 (View view) {
    iTema = 2;
    this.recreate();
}

public void onClickTema3 (View view) {
    iTema = 3;
    this.recreate();
}
```

Aplikaci spustíme na emulátoru nebo v zařízení. Výsledek by měl vypadat jako na obrázku 28.7. Klikáním na tlačítka se mění nastavené téma aplikace.

Obrázek 28.7: Přepínání barevných témat (motivů)

29 Reklamy

Jedním ze způsobů monetizace aplikace je použití reklam ve formulářích. Google dává vývojářům k dispozici stránky AdMob, kde si můžou zaregistrovat a nastavit zobrazování reklam v komponentě AdView. Za zobrazování potom vývojářům platí podíl ze zisku, vygenerovaném danou aplikací.

Prvotní zprovoznění reklam v aplikaci není úplně snadné. Lehce se může stát, že ačkoliv je zdánlivě všechno správně nastavené, komponenta reklamu ne a ne zobrazit. Projděme si proces registrace na AdMob a nastavení komponenty tak, aby se pokud možno všechno podařilo na první pokus nebo abychom alespoň měli možnost případnou chybu odhalit a napravit.

29.1 Služba AdMob

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 29_Reklamy
projekt Reklama1

Na samém začátku se musíme prokousat přes několik základních konfigurací a nastavení. Celý proces trochu připomíná překážkovou dráhu, je to ale jenom napoprvé, jakmile se na zařízení zobrazí první reklama, bude v podstatě vyhráno. Všechny další kroky budou už jenom o drobných úpravách a o přidávání dalších reklamních ploch.

Co tedy musíme udělat?

- zaregistrovat se do služby AdMob

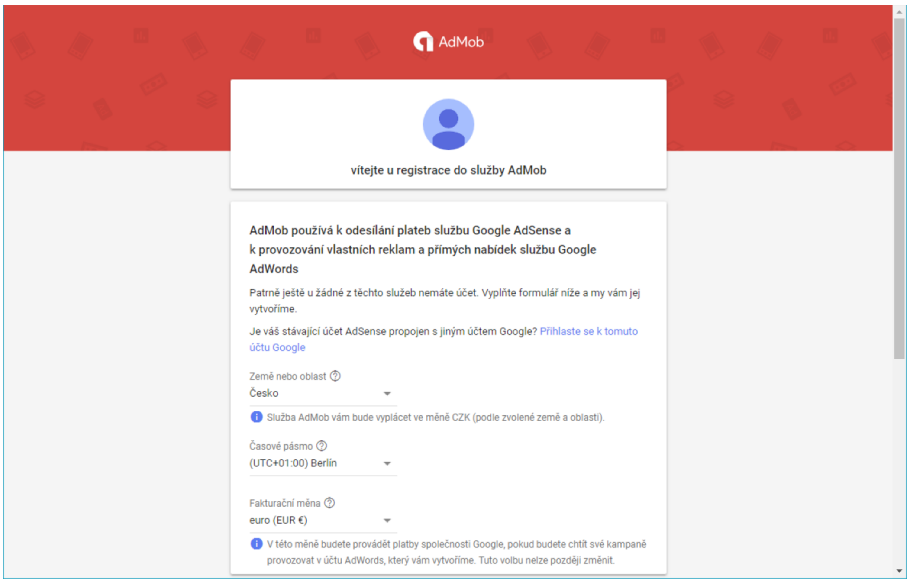
- vygenerovat ID pro aplikaci a pro reklamní plochu

- založit aplikaci s komponentou AdView

- načíst do komponenty reklamu

Začneme na internetové stránce www.admob.com. Google má pro účely mobilní reklamy zřízenou speciální doménu admob, ale v podstatě jde o stejnou stránku, na kterou se lze dostat zadáním ryze googlovské adresy www.google.com/admob/. Zvolíme Sign up for AdMob, neboli registraci do služby AdMob. Pro registraci můžeme použít stávající účet, který používáme pro ostatní služby Google.

Následná registrace je společná pro AdMob (služba pro zobrazování reklam) a AdWords (služba pro zadavatele reklam). Musíme tedy zadat údaje pro obě platformy, ačkoliv tu druhou nijak nevyužijeme.

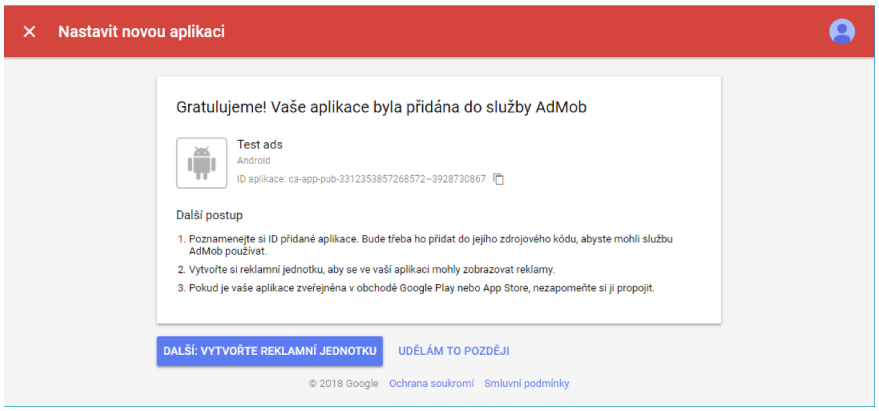


Obrázek 29.1: Registrace na stránkách služeb AdMob a AdWords

Po registraci ve službě je potřeba zaregistrovat aplikaci. To už je snadné a rychlé. Zadáme její jméno a obratem obdržíme klíč aplikace.

Pomocí tohoto klíče se budeme z aplikace hlásit ke službě AdMob. Je to důležité kvůli připisování vydělaných peněz na účet.

Není potřeba obávat se registrace aplikace, která dosud není na Google Play Store. AdMob se profiluje jako samostatná služba, která poskytuje reklamu bez ohledu na to, jestli je aplikace pro Android, iOS, nebo jestli je šířena přes alternativní distribuční kanály. Nevadí tedy ani vývojová verze aplikace, která zatím nebyla publikována vůbec nikde.



Obrázek 29.2: Registrace aplikace ve službě AdMob

Každá aplikace může mít více reklamních ploch, dokonce různého typu. Základní druhy ploch jsou tyto:

Banner – reklamní proužek, který obsahuje text nebo obrázek. Zobrazuje se na ploše k tomu vyhrazené. Je nejčastějším a nejméně rušivým způsobem zobrazení reklamy v aplikaci.

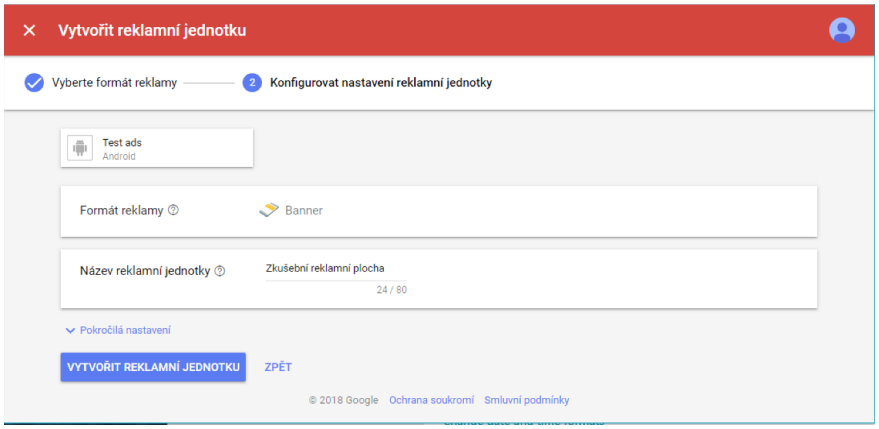
Vsunutá (Interstitial) – reklama, která se zobrazí přes celou plochu aplikace a čeká, až ji uživatel zavře. Použít ji lze v přirozeně oddělených fázích práce s aplikací, například po dosažení jedné úrovně hry. Jelikož jde o dost nepříjemný způsob zobrazení reklamy, je ji vhodné využívat spíše výjimečně, velmi opatrně a pouze tam, kde se dá očekávat, že ji bude uživatel akceptovat.

S nabídkou odměny (Rewarded Video) – zobrazí reklamní video přes celou plochu aplikace. Uživatel může za shlédnutí klipu získat odměnu. Tuto reklamu je vhodné zobrazit až poté, co si ji uživatel vyžádá, například kliknutím na tlačítko.

Přizpůsobená (Native) – reklama, kterou je možné přizpůsobit vzhledu aplikace. Vývojář může definovat vzhled titulků, barev a dalších vlastností zobrazovaných reklam, takže tyto do aplikace lépe zapadnou.

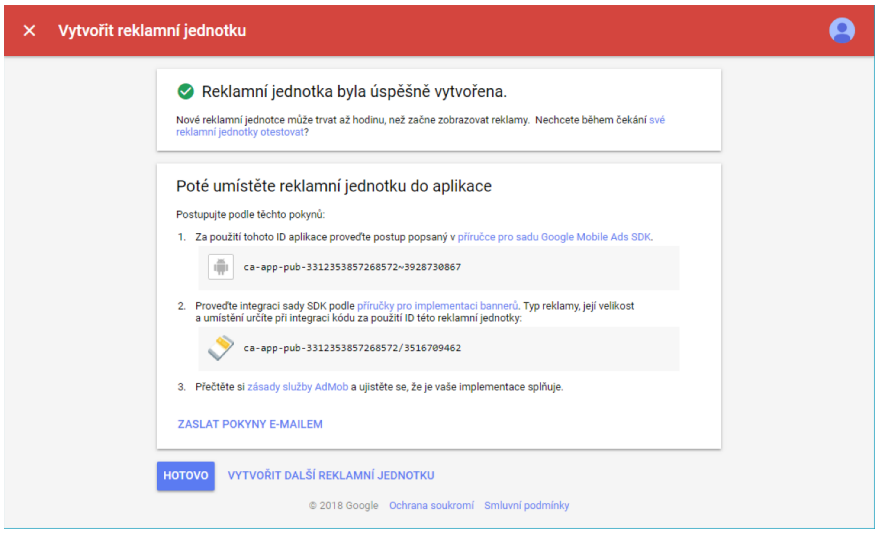
Spuštěná aplikace (App Open) – okno, které překryje obrazovku při spouštění aplikace a zobrazí reklamu. Uživatel ho může zavřít a přejít do aplikace. Tento typ reklamy je v podstatě variací na vsunutou reklamní plochu (Interstitial).

Musíme tedy zaregistrovat ještě každou reklamní plochu zvlášť. V terminologii AdMob se reklamním plochám říká Reklamní jednotky neboli Ad units.



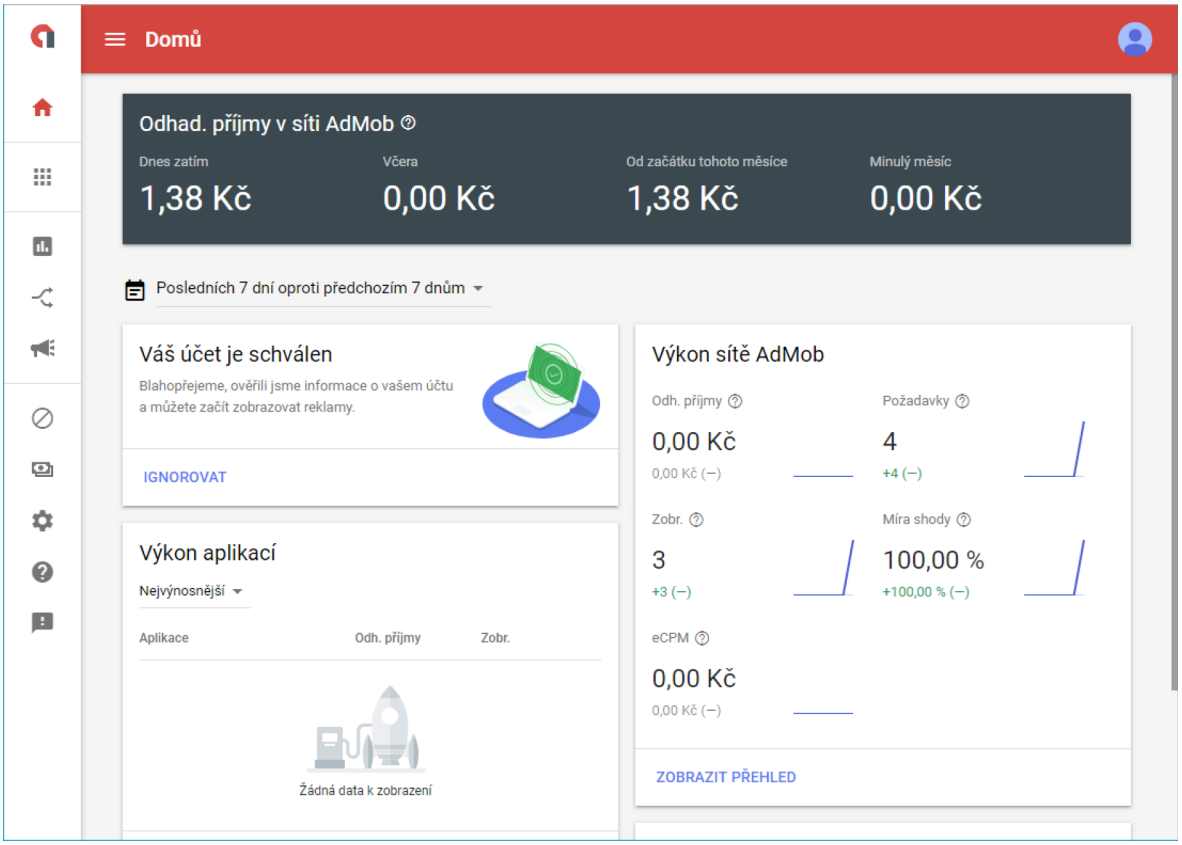
Obrázek 29.3: Registrace reklamní plochy ve službě AdMob

Po nastavení reklamní jednotky získáme druhý klíč – klíč reklamní jednotky. Pozor, oba klíče mají na svém začátku identifikační číslo uživatele, takže jsou si dost podobné. Nesmíme je splést. Pro rozlišení si stačí zapamatovat, že číslo aplikace uvnitř obsahuje vlnovku, zatímco číslo reklamní jednotky lomítko.



Obrázek 29.4: Vygenerování klíče reklamní jednotky ve službě AdMob

Tím máme připravené všechno potřebné pro to, abychom mohli zabudovat svoji první reklamní jednotku do aplikace. Aktuální stav účtu budeme sledovat a upravovat na stránce apps.admob.com.



Obrázek 29.5: Stav účtu na stránkách služby AdMob

Nyní si můžeme založit novou aplikaci, které do formuláře vložíme komponentu Google → AdView. Můžeme se přitom dostat do stejných nesnází jako při vkládání mapového fragmentu.

AdView potřebuje mít nainstalované knihovny Google Play Services Ads. V ideálním případě si je Android Studio doinstaluje samo při vkládání komponenty. Že je vše v pořádku se dá poznat po otevření souboru Gradle Scripts → build.gradle (Module: app). Musí v něm být, v části dependencies, záznam o Play Services Ads:

```
dependencies {
    ...
    implementation 'com.google.android.gms:play-services-ads:22.3.0'
}
```

Pokud tam není, je jej nutné přidat ručně. Stačí do bloku dependencies nakopírovat chybějící řádek implementation.

Otevřeme soubor Scripts → build.gradle (Project: *název projektu*) a do bloku allprojects – repositories přidáme odkaz na SDK v úložišti google. Výsledek by měl vypadat takto:

```
allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

Po úpravách gradle souborů oba sesynchronizujeme. Android Studio by mělo synchronizaci nabídnout samo. Pokud se tak nestane, stačí kliknout pravým tlačítkem myši postupně na oba soubory a zvolit položku Synchronize ‘build.gradle’. Pro jistotu ještě můžeme smazat starší soubory aplikace, menu Build → Clean Project.

Přestože se budou reklamy stahovat přes internetové připojení, není nutné v tomto případě aplikaci zvlášť dávat právo na datovou komunikaci. V manifestu tedy nemusíme žádat o žádné nové oprávnění. Musíme do něj ale přidat App ID, které je přiřazené aplikaci. Toto App ID jsme získali během registrace na stránkách admob.com.

```
<application
    ....>

    ...

    <meta-data
        android:name="com.google.android.gms.ads.APPLICATION_ID"
        android:value="ca-app-pub-3312353857268572~5812959490"/>

</application>
```

Dalším krokem bude inicializace Ads SDK, spojená s předáním ID aplikace. Otevřeme třídu MainActivity a do události onCreate() vložíme instrukci pro inicializaci reklamy:

```
MobileAds.initialize(this, new OnInitializationCompleteListener() {
    @Override
    public void onInitializationComplete(
                              InitializationStatus initializationStatus) {
    }
});
```

Tato inicializace musí alespoň jednou proběhnout, nejlépe hned při startu aplikace. Proto se ji hodí mít v onCreate(). Předefinovaná událost onInitializationComplete() se zavolá, jakmile je inicializace úspěšně dokončena, nebo vyprší 30 sekundový timeout.

Ještě než dojde k prvnímu načtení, je potřeba doplnit také ID reklamní jednotky. Otevřeme v textovém režimu soubor activity_main.xml, v němž je definice komponenty AdView a přidáme do ní parametry ads:adSize a ads:adUnitId.

```
<com.google.android.gms.ads.AdView
    xmlns:ads="http://schemas.android.com/apk/res-auto"
    android:id="@+id/adView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    ads:adSize="BANNER"
    ads:adUnitId="ca-app-pub-3312353857268572/3516709462"
    ads:layout_constraintEnd_toEndOf="parent"
    ads:layout_constraintStart_toStartOf="parent"
    ads:layout_constraintTop_toTopOf="parent" />
```

Parametr adSize nese informaci o typu reklamní jednotky, resp. o její velikosti. Tou je v našem případě ‚BANNER‘, neboli plocha 320 x 50 obrazových bodů. UnitId obsahuje identifikátor reklamní plochy, získaný během registrace na stránkách admob.com.

K dispozici jsou tyto velikosti reklamních jednotek:

BANNER: 320x50 bodů, určený pro telefony a tablety

LARGE_BANNER: 320x100 bodů, určený pro telefony a tablety

MEDIUM_RECTANGLE: 300x250 bodů, určený pro telefony a tablety

FULL_BANNER: 468x60 bodů, určený pro tablety

LEADERBOARD: 728x90 bodů, určený pro tablety

SMART_BANNER: přizpůsobí se velikosti displeje, určený pro telefony a tablety

Nastavíme-li jinou velikost než BANNER, automaticky se v designeru upraví velikost reklamní plochy.

Teď už zbývá jenom reklamní plochu zobrazit. Vrátíme se ještě ke třídě MainActivity a do události onCreate() vložíme instrukci pro načtení reklamy:

```
AdView adView = findViewById(R.id.adView);
AdRequest adRequest = new AdRequest.Builder().build();
adView.loadAd(adRequest);
```

Aplikaci zkompilujeme a spustíme na koncovém zařízení. Krátce po startu by se měla reklamní plocha zobrazit.

Pokud se reklama nezobrazí, nahradíme v souboru activity_main.xml ID reklamní plochy vývojářským klíčem:

```
ads:adUnitId="ca-app-pub-3940256099942544/6300978111"
```

Tento klíč je univerzální a zobrazuje namísto reklam pouze vývojářskou informaci. Jelikož se na něj nevztahují tak přísná pravidla zabezpečení jako na reálné reklamy, je jeho zobrazení snazší. Univerzální vývojářský klíč je určený pro reklamní jednotku typu BANNER, bez ohledu na jeho velikost. Další typy reklamních jednotek ale mají svoje vlastní vývojářské klíče, viz tabulka 29.1.

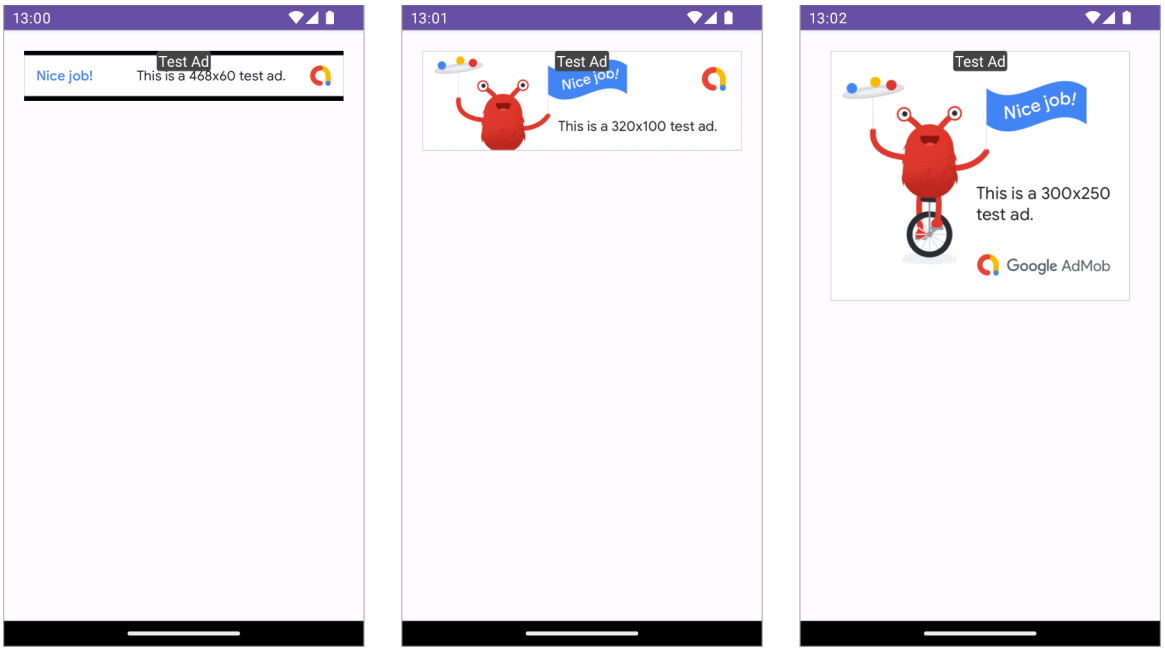
Reklamní jednotka	Typ	Vývojářský klíč
Banner	Všechny velikosti	ca-app-pub-3940256099942544/6300978111
Vsunutá (Interstitial)	Základní	ca-app-pub-3940256099942544/1033173712
	Video	ca-app-pub-3940256099942544/8691691433
S nabídkou odměny (Rewarded Video)	Všechny typy	ca-app-pub-3940256099942544/5224354917
Přizpůsobená (Native)	Základní	ca-app-pub-3940256099942544/2247696110
	Video	ca-app-pub-3940256099942544/1044960115
Spuštěná aplikace (App Open)	Všechny typy	ca-app-pub-3940256099942544/1033173712

Tabulka 29.1: Univerzální vývojářské klíče pro reklamní jednotky AdMob

Jestliže se ani s vývojářským klíčem reklamní jednotka nezobrazila, je nutné znovu projít nastavení. A když se něco změní v Gradle, nesmíme ho zapomenout opět sesynchronizovat a udělat nový build. Také je nutné zohlednit rozlišení displeje, jestli na zobrazení jednotky stačí. Problémy by mohl způsobovat i nástroj na blokování reklam, je-li na zařízení nainstalovaný. Další nápovědu můžeme nalézt v Android Studiu v oknech Run a Logcat, kam se vypisují informace během spouštění aplikace.

Pokud se vývojářská reklamní jednotka zobrazila, ale reálná je stále neviditelná, bude problém v nastavení jednotky. Je tedy nutné hledat v nastaveních na admob.com.

Když se vše podaří, bude výsledek na koncovém zařízení vypadat přibližně jako na obrázku. Vlevo je nejmenší dostupná reklamní jednotka, typ BANNER, uprostřed LARGE_BANNER a vpravo MEDIUM_RECTANGLE. Všechny jsou navázané na ID vývojářské reklamní jednotky:



Obrázek 29.6: Aplikace s reklamními jednotkami typu BANNER, LARGE_BANNER a MEDIUM_RECTANGLE, spuštěná na emulátoru

29.2 Testovací reklamy

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 29_Reklamy
projekt Reklama2

Pro účely vývoje není příliš vhodné používat reálné reklamy. Neobvyklé aktivity, jako časté zobrazování nebo klikání na reklamní plochy by mohl systém AdMob vyhodnotit jako pokus o podvod a zobrazování zablokovat. Pro tyto účely existuje konfigurační nastavení, pomocí kterého dá aplikace systému na vědomí, že na něm probíhají vývojářské aktivity a že má tedy tolerovat nezvyklé chování a zároveň nemá za takto zobrazené reklamy vyplácet odměny a potažmo ani účtovat poplatky jejich zadavatelům.

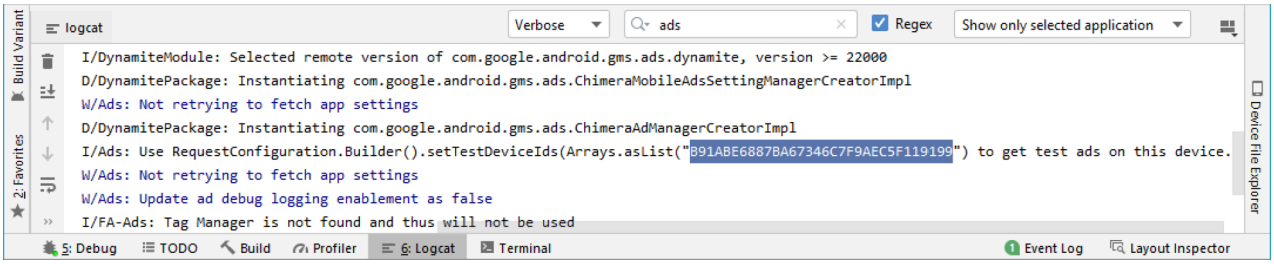
Reklamy, které má systém AdMob připravené k distribuci rozděluje podle řady kritérií, tak aby zasáhl pokud možno co nejvhodnější cílovou skupinu uživatelů. V případě, že pro zařízení nemá dostatek reklam, tak prostě nic nezobrazí. Takové situaci se lze pomocí testovacích reklam také vyhnout, systém bude reklamy dodávat přednostně, bez toho, aby mu ‚došly'.

V případě, že aplikaci ladíme na emulátoru, je konfigurace pro testovací reklamy nastavena automaticky. Není tedy nutné nic dělat, aplikace bude zobrazovat reklamy pouze v testovacím režimu.

U ladění na fyzickém zařízení je to složitější, tam je nutné dát systému vědět, že bude aplikace pracovat v testovacím režimu. K tomu potřebujeme znát ID zařízení, které reklamnímu systému předáme. To se dá nejsnáze zjistit při ladění aplikace, která reklamy již zobrazuje. A mohou to být i pouhé vývojářské reklamní plochy, viz předchozí kapitola. Stačí v Android Studiu otevřít okno Logcat, aplikaci spustit a zachytit řádek, který bude obsahovat text:

```
I/Ads: Use RequestConfiguration.Builder().setTestDeviceIds(Arrays.asList("xxxx")
to get test ads on this device.
```

V uvozovkách bude namísto „xxxx“ identifikátor zařízení. A právě ten nás zajímá. Z okna Logcat si ho stačí následně vykopírovat.



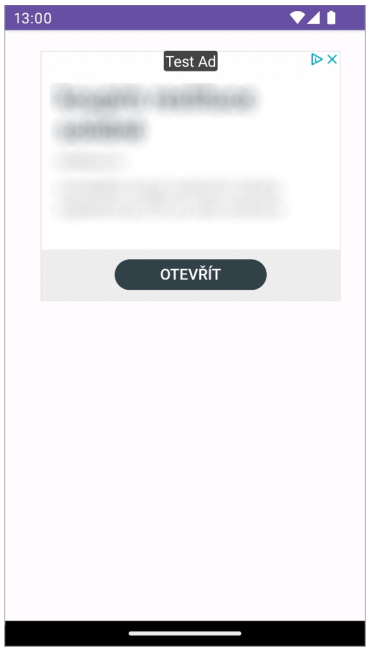
Obrázek 29.7: Identifikátor připojeného zařízení v okně Logcat Android Studia

Samotnou konfiguraci je vhodné přidat do třídy MainActivity, do události onCreate(), nejlépe na její začátek, v každém případě před instrukce pro inicializaci a načtení reklamy:

```
List<String> testDeviceIds = Arrays.asList("B91ABE6887BA67346C7F9AEC5F119199");
RequestConfiguration configuration = new
      RequestConfiguration.Builder().setTestDeviceIds(testDeviceIds).build();
MobileAds.setRequestConfiguration(configuration);
```

Pokud je komponenta AdView nastavena tak, aby zobrazovala vývojářské reklamy, je v ní potřeba nahradit vývojářský klíč klíčem vlastní reklamní jednotky služby AdMob. Aplikace tedy bude, s výjimkou konfigurace pro testovací reklamy, nastavena úplně stejně, jako by měla zobrazovat běžné komerční reklamy.

Aplikaci zkompilujeme a spustíme na koncovém zařízení. Měla by zobrazit standardní komerční reklamu, která však bude označena malým nápisem Test Ad ve své horní části. Vzhledem k tomu, že tato reklama není určena k vyplacení odměny, tak se její zobrazení neprojeví ve statistikách služby AdMob.



Obrázek 29.8: Aplikace s reklamní jednotkou typu MEDIUM_RECTANGLE, v režimu zobrazování testovacích reklam

30 Widgety

Malé aplikace, trvale usazené na displejích zařízení, zobrazující základní informace o čase, počasí nebo čemkoliv jiném, co uživatele zajímá. Neexistuje pro ně ustálený český název, takže je lepší zůstat u zkomoleniny z angličtiny – widgety. Jsou zřejmě nejviditelnějším typem aplikací, které se dají na Android zařízeních najít. Někdy, kromě toho, že informace zobrazují, umožňují také minimalistickou interakci s uživatelem.

Při návrhu vzhledu widgetu se postupuje podobně jako při návrhu běžné aplikace. Na výběr je ale omezená množina komponent a formulářů, které lze použít. Prvky, se kterými si vývojář musí vystačit, jsou uvedeny v tabulce 30.1.

Formuláře	Komponenty
FrameLayout	AnalogClock
LinearLayout	Button
RelativeLayout	Chronometer
GridLayout	ImageButton
	ImageView
	ProgressBar
	TextView
	ViewFlipper
	ListView
	GridView
	StackView
	AdapterViewFlipper

Tabulka 30.1: Prvky, které je možné použít při návrhu widgetu

Tyto prvky můžou být, ještě ke všemu, použity pouze v základní formě, žádné odvozené typy ani potomci nejsou povoleny. Naštěstí Google množství dostupných komponent čas od času rozšiřuje.

V některých případech ale dochází k nepříjemné situaci, že jsou prvky použitelné ve widgetu již zastaralé a v API jsou nahrazeny jinými, modernějšími. Například ListView je v novějších API označený jako ‚deprecated' a Google místo něj doporučuje použít RecyclerView. Ten ale bohužel pro widgety povolený není. Před návrhem nového widgetu je tedy vhodné zkontrolovat aktuální stav odpovídající verzím API, pro které má být widget dostupný. Může se stát, že mezitím přibyly nové formuláře nebo komponenty, které ještě nedávno s widgety kompatibilní nebyly.

Použití nepovolených prvků se neprojeví při kompilaci widgetu, jak by se dalo očekávat, ale až při jeho použití v Android zařízení. Takový widget se v různých verzích Androidu chová různě. Někdy prostě nejde umístit na plochu, jindy se ho tam umístit podaří, ale s ne úplně výstižným výpisem chyby ‚Při načítání došlo k potížím'.

Widgety mají několik specifických vlastností, které je třeba, při jejich návrhu, vzít v potaz. Velikost zobrazeného formuláře je, na rozdíl od běžných aplikací, menší než plocha celého displeje. Displej je pro tento účel rozdělený na malé bloky, do nichž se widget musí vejít. Počet bloků je daný velikostí displeje. Vývojář si může zvolit, kolik bloků má naprogramovaný widget zabírat. Minimum je jeden blok, maximum není omezené, ale je dobré počítat s tím, že displeje mají celkem běžně k dispozici 4 x 3 bloky a uživatel rozhodně nebude chtít, abychom mu je zabrali všechny. Volme proto co nejmenší počet bloků, který nám ještě stačí. Uživatelé také ocení, když se widget bude umět dynamicky přizpůsobit pro různou velikost a on si tak bude moct sám vybrat, kolik z dostupné plochy má widgetem zabrat.

Další specifickou vlastností je aktualizace údajů. Jelikož je widget na displeji stále, tak má nároky na systémové prostředky. A widgetů může mít na displeji uživatel hned několik. Proto je potřeba přistupovat k obnovování informací widgetu uvážlivě. Obecně se dá říct, že čím méně operací, tím lépe. Pokud má widget zobrazovat aktuální čas, je vhodné udělat obnovení pouze každou minutu. Rozhodně ne každou sekundu, takový widget by zatěžoval procesor příliš a více podobných widgetů by tak mohlo způsobit zvýšenou spotřebu baterie. Ideální je nechat obnovení závislé na uživateli – pokud například ťukne na tlačítko, tak informaci obnovit, jinak ji aktualizovat v dlouhých časových intervalech, třeba jednou za hodinu.

Widgety jsou zpravidla svázány s plnohodnotnou aplikací. Ta ale nemusí dělat žádné významné operace, sloužit může jenom pro uživatelské nastavení základních parametrů widgetu. Jak aplikace, tak samotný widget bývají součástí jednoho projektu.

30.1 Widgety, 1. příklad

Na příkladu si ukážeme, jak vytvořit jednoduchý widget. Prozatím si dáme za cíl pouze, aby se přidal do systému, aby ho bylo možné umístit na plochu a aby zobrazil jednoduchý text. K widgetu bude patřit aplikace, která by se jinak použila pro nastavení parametrů widgetu. Náš widget ale žádné parametry potřebovat nebude, takže aplikace bude mít prázdný formulář.

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 30_Widgety
projekt Widget1

Co je potřeba udělat?

- vytvořit formulář widgetu
- vytvořit xml soubor s parametry widgetu
- upravit manifest tak, aby výsledkem byla aplikace s widgetem
- napsat jednoduchý provider, který bude widget řídit

Založíme si v Android Studiu nový projekt s prázdnou aktivitou. Tuto aktivitu nemusíme nijak upravovat, stane se z ní aplikace, která bude spouštěna při instalaci widgetu. Jelikož widget nebude potřebovat žádná uživatelská nastavení, může aktivita zůstat prázdná. Do projektu ale přidáme nový formulář. Nazveme ho třeba widget.xml a umístíme ho do obvyklé složky app/res/layout/. Formulář musí být typu Frame, Linear, Relative nebo Grid Layout. Pro jednoduchost použijeme Relative Layout, který umožní snadno a rychle umístit doprostřed formuláře jednoduché textové pole. XML kód formuláře bude vypadat takto:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#30000000">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="Widget test"
        android:textAlignment="center"
        android:textColor="#E0E0E0"
        android:textSize="28sp" />
</RelativeLayout>
```

Máme tedy formulář s Relative Layoutem, který má barvu pozadí #30000000, neboli černou, ale s průhledností #30, díky níž je formulář mírně průhledný a opticky světle šedý. Na tmavém pozadí nebude vidět, na světlém bude mírně ztmavený. Vertikálně i horizontálně uprostřed má umístěné textové pole s velikostí písma 28 sp a barvou textu #E0E0E0, neboli světle šedou. Tím je formulář připraven.

Protože je widget speciálním druhem aplikace, vyžaduje také některá speciální programátorská nastavení ve formě metadat. Ta se definují v xml souboru, na který se později odkážeme v manifestu. Soubor bude uložený v projektové struktuře ve složce app/res/xml/ a může se jmenovat třeba widget_info.xml. Jak může obsah souboru vypadat:

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="120dp"
    android:minHeight="60dp"
    android:updatePeriodMillis="1800000"
    android:initialLayout="@layout/widget"
    android:resizeMode="horizontal|vertical"
    android:widgetCategory="home_screen|keyguard">
</appwidget-provider>
```

První dva parametry určují minimální velikost widgetu. Podle toho potom Android, při umístění widgetu na plochu, rozhodne o tom, kolik bloků má widget zabrat. Je vhodné použít násobky 60, protože velikost bloků zpravidla bývá 60 x 60 dp.

Parametr updatePeriodMillis říká, v jakém intervalu má být volána událost onUpdate() provideru, která aktualizuje zobrazení widgetu. Hodnota je v milisekundách, ale je potřeba upozornit, že minimální dobou pro aktualizaci je 30 minut. Čili zadávat méně než 1 800 000 milisekund nedává příliš smysl, k aktualizaci dojde stejně až po půl hodině. Pokud by bylo potřeba aktualizovat widget častěji, řeší se to pomocí události alarmu. Ale k tomu se ještě dostaneme. Zatím budeme zobrazovat statický text, takže v tomto případě půlhodina nepředstavuje problém.

Na formulář widgetu odkazuje parametr initialLayout. Sem prostě zadáme cestu k formuláři. V následujícím parametru resizeMode nastavíme, jestli může uživatel widget zvětšovat a to ve vodorovné nebo svislé ose.

Posledním je widgetCategory, parametr, který říká, jestli jde o widget, který jde umístit na plochu, nebo na odemykací obrazovku. Případně na obojí. Odemykací obrazovka se týká starších verzí Androidu (do verze 4.4, tj. API 19), novější podporují pouze umístění na plochu a druhou

možnost jednoduše ignorují. Takže na nastavení tohoto parametru záleží pouze částečně, je v něm nutné mít zmíněnou hodnotu home_screen. Druhá možná hodnota, keyguard, tam být může a nemusí, efekt bude mít pouze na starých zařízeních.

Máme tedy konfigurační soubor s metadaty widgetu, který odkazuje na připravený formulář. Aby to celé fungovalo, musíme rozšířit manifest o informaci o konfiguračním souboru. Uděláme to tak, že v manifestu do sekce application přidáme následující sekvenci:

```
<receiver android:name=".WidgetProvider"
          android:exported="true">
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
    <meta-data android:name="android.appwidget.provider"
        android:resource="@xml/widget_info" />
</receiver>
```

Tím Androidu sdělíme, že jde o widget a nasměrujeme ho na soubor s metadaty, kde najde potřebné informace pro jeho instalaci a spuštění.

Pozor, blok activity z manifestu neodstraňujeme, obsahuje informace o aktivitě, která slouží k uživatelským nastavením a je povinná.

Ještě zbývá napsat provider, neboli třídu, která se bude starat o řádné zobrazení widgetu. Založíme soubor WidgetProvider.java, který bude rozšiřovat třídu AppWidgetProvider. Rozšíření s sebou nese především nutnost předefinovat událost onUpdate(), která se volá jednak po umístění widgetu na plochu, po restartu zařízení a po uplynutí intervalu definovaného v metadatech widgetu (ona zmiňovaná minimálně půlhodina). V události onUpdate() tedy musíme zařídit všechno, co se týká zobrazení a kdybychom ve widgetu měli nějaké komponenty pro interakci s uživatelem, musíme zde také definovat příslušné handlery. To ale zatím nepotřebujeme, widget nebude na uživatele nijak reagovat.

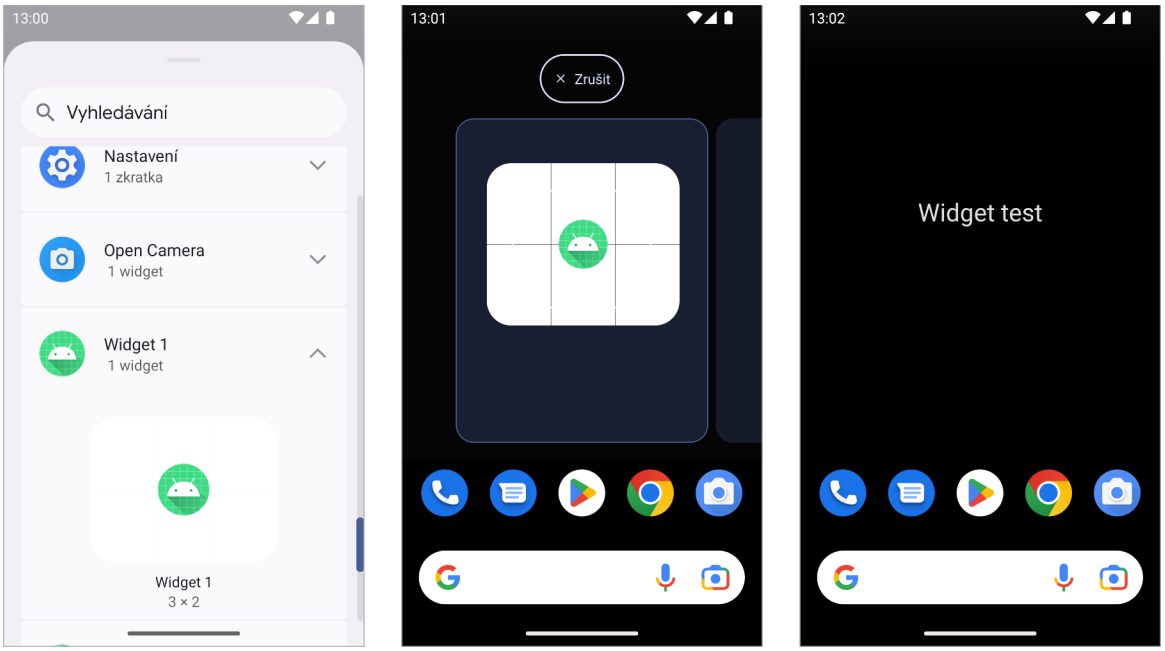
Kód provideru:

```
public class WidgetProvider extends AppWidgetProvider {
    @Override
    public void onUpdate(Context context, AppWidgetManager appWidgetManager,
                                                    int[] appWidgetIds) {
        final int count = appWidgetIds.length;
        for (int i = 0; i < count; i++) {
            int widgetId = appWidgetIds[i];
            RemoteViews remoteViews = new RemoteViews(context.getPackageName(),
                                                    R.layout.widget);
            remoteViews.setTextViewText(R.id.textView, "Widget test");
            appWidgetManager.updateAppWidget(widgetId, remoteViews);
        }
    }
}
```

V události je zajímavé zejména pole appWidgetIds[]. Obsahuje informaci o všech instancích widgetu, které má uživatel na displeji. Widget totiž nemusí být na ploše jednou, ale uživatel si ho tam může dát kolikrát chce. A díky tomuto poli má provider přehled o všech takto umístěných instancích a může je snadno aktualizovat. Nejsnazší je v takovém případě projít instance v cyklu a hodnotu, v našem případě „Widget test“ vypsat v TextView na všech widgetech, které jsou

aktuálně na ploše. Mohli bychom samozřejmě zjišťovat, který widget je který a na každý vypisovat jiné informace. V tomto případě to ale není třeba, všechny instance se budou chovat stejně.

Po kompilaci a spuštění by se na displeji měl objevit formulář hlavní aktivity, určený pro uživatelská nastavení. Ten zavřeme a přejdeme do instalace widgetů. Náš widget by měl být v seznamu tak, jako na obrázku 30.1 vlevo. Vezmeme ho a umístíme na plochu, viz prostřední obrázek. Na pravém vidíme, jak by měl náš widget na ploše vypadat.



Obrázek 30.1: Instalace vlastního widgetu

30.2 Widgety, 2. příklad

Zdrojové soubory ke stažení
www.github.com
repozitář 100xAndroid/Java
složka 30_Widgety
projekt Widget2

Po jednoduchém testovacím widgetu můžeme přejít k praktičtější aplikaci. S použitím kódu a postupů z předchozích kapitol, vytvoříme widget, který bude v textové formě zobrazovat aktuální počasí, bude mít grafické tlačítko pro otevření aktivity s nastavením a tlačítko pro načtení aktuálního stavu počasí. Kromě toho se bude počasí každou půlhodinu načítat automaticky. V aktivitě s nastavením zadáme místo, pro které se má počasí zjišťovat.

Využijeme znalosti z předchozích kapitol:

- z kapitoly 20.4 Připojení k internetu, 2. příklad použijeme celou třídu GetData, pro načítání a parsování informací o počasí ze serveru openweathermap.org

- z kapitoly 18.1 Sdílená nastavení, Shared Preferences využijeme postup pro uložení a načtení nastavení prostřednictvím sdílené paměti

- postupy z kapitoly 26 Ikona aplikace, zúročíme při tvorbě grafických tlačítek a zejména ikony widgetu

Data poskytovaná serverem openweathermap.org jsou dostupná v mnoha jazycích. Asi by je nebyl problém při načítání dynamicky přepnout do jazyka zařízení, nebo je zobrazit graficky, pro jednoduchost se do toho ale nebudeme pouštět. Informace na displeji zobrazíme textově v češtině, jak je openweathermap.org dodá.

Co je potřeba udělat?

- pořídíme kopii předchozího příkladu
- upravíme formulář s nastavením activity_main.xml
- ve třídě MainActivity zařídíme, aby se nastavení ukládalo do sdílené paměti
- upravíme formulář widget.xml
- přidáme ikonky tlačítek
- do projektu přidáme třídu GetData pro načítání počasí
- rozšíříme manifest o oprávnění pro přístup k datům
- ve WidgetProvider.java napíšeme metody pro načítání nastavení, pro výpis dat na displej a pro volání metod třídy GetData
- přidáme ikonu aplikace

Protože aplikace bude mít úplně stejný základ jako předchozí příklad, můžeme pokračovat v jejím rozšiřování. Nebo si uděláme její kopii a změníme název. Další funkcionalitu tedy budeme pouze přidávat.

Jelikož bude widget potřebovat vědět, pro které místo má informace o počasí načítat, využijeme tentokrát i hlavní aktivitu. Upravíme soubor activity_main.xml tak, aby formulář obsahoval textové pole pro zadání místa a tlačítko pro jeho uložení. Na typu formuláře příliš nezáleží, nic nezkazíme například použitím ConstraintLayoutu.

Tlačítko pro ukládání bude volat metodu ulozitNastaveni(), která do sdíleného souboru Nastaveni uloží obsah textového pole. Využití sdíleného souboru je v tomto případě výhodné, protože jak aktivita tak widget jsou součástí jedné aplikace a díky tomu mají k souboru přístup. Základní myšlenka je taková, že aktivita bude soubor zakládat a editovat a widget z něj bude číst pokaždé, když bude chtít stáhnout aktuální počasí.

Předpokládejme, že se textové pole ve formuláři aktivity bude jmenovat txtMisto. Metoda bude vypadat nějak takto:

```
public void ulozitNastaveni (View view) {
    EditText editText1 = (EditText)findViewById(R.id.txtMisto);
    String inputText = editText1.getText().toString();

    SharedPreferences sharedPreferences = getSharedPreferences(
                                          "Nastaveni", Context.MODE_PRIVATE);

    SharedPreferences.Editor editor = sharedPreferences.edit();
    editor.putString("misto", inputText);

    editor.apply();
```

```
    Toast.makeText(getApplicationContext(), "Nastavení uloženo.",
                                            Toast.LENGTH_LONG).show();
}
```

Metoda načte hodnotu z textového pole txtMisto, založí sdílený soubor, uloží do něj hodnotu. Toto všechno instrukcí apply() uloží a potom už dá jenom uživateli pomocí toast zprávy vědět, že je hotovo.

Jelikož se v aktivitě hodnota pole txtMisto bude nejen pořizovat, ale také editovat, je ji vhodné uživateli předvyplnit na základě toho, co zadal minule. Kvůli tomu je třeba rozšířit událost onCreate() o načtení hodnoty ze sdíleného souboru. Uděláme to takto:

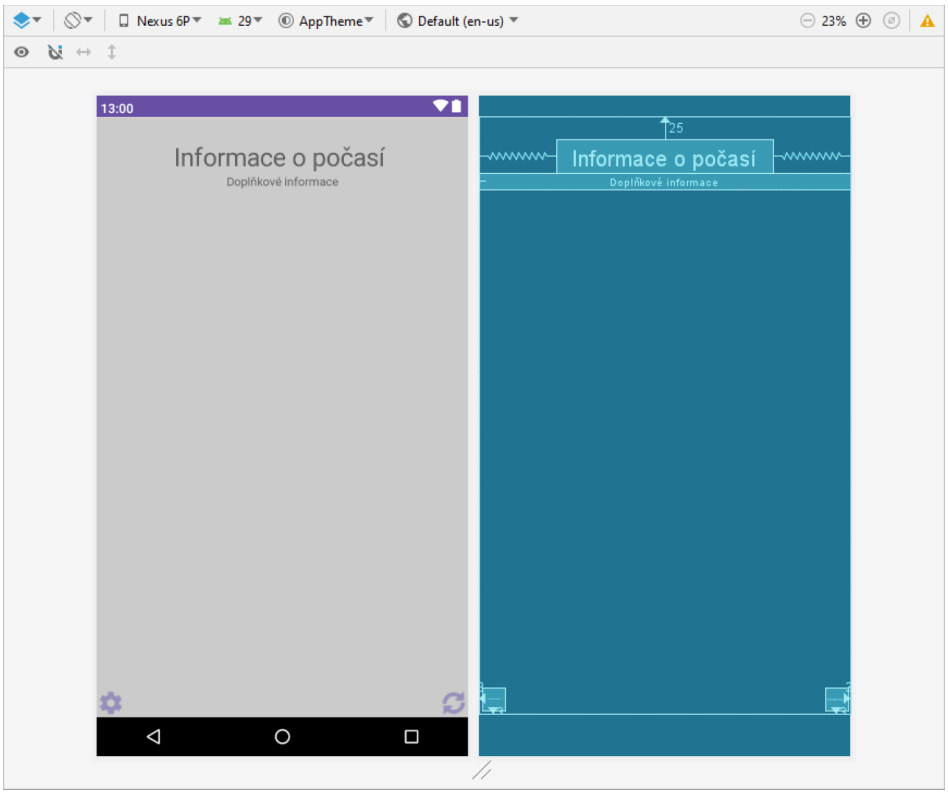
```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    SharedPreferences sharedPreferences = getSharedPreferences("Nastaveni",
                                                Context.MODE_PRIVATE);
    // nacteme ulozenou hodnotu
    String misto = sharedPreferences.getString("misto", "Karlovy Vary");
    // zapiseme hodnotu do pole
    EditText editText = (EditText)findViewById(R.id.txtMisto);
    editText.setText(misto);
}
```

Co se při vyvolání události stane? Nejprve se otevře soubor pro čtení, potom se z něj uložená hodnota načte a nakonec se zapíše do textového pole na formuláři.

Jakmile je aktivita nastavení hotová, vrhneme se na vzhled widgetu. Zde jsme omezeni několika typy rozložení formuláře, viz minulá kapitola. Na formuláři budou potřeba dvě textová pole, jedno s velkým textem, informujícím o počasí, druhé malé, s informací o místě a poslední aktualizaci. Nalevo a napravo budou grafická tlačítka, jedno pro otevření nastavení a druhé pro okamžité načtení dat o počasí. Pokud zůstaneme u formuláře typu Relative Layout, bude stačit umístit do horní části vycentrované velké textové pole a na ně navázat pole malé, také vystředěné. Grafická tlačítka můžou být v levém a pravém dolním rohu.

V grafickém editoru připravíme ikonky pro nastavení a pro opakované načtení dat. Pokud má být widget vizuálně částečně průhledný, je vhodné ikonkám před uložením nastavit alfa kanál, neboli průhlednost, aby i přes ně mohlo prosvítat pozadí.

V designéru to bude vypadat takto:



Obrázek 30.2: Widget v designéru Android Studia

Je potřeba si uvědomit, že náhled v designeru je trochu zavádějící. Zobrazuje klasickou aktivitu, ale xml soubor převedený do podoby widgetu bude obsahovat pouze vložené prvky bez záhlaví a sražené na malou plochu.

Z projektu z kapitoly 20.4 Připojení k internetu, 2. příklad, si vezmeme soubor GetData.java. Jeho kopii vložíme do projektového adresáře na stejnou úroveň, ve které je MainActivity.java. Díky tomu se nebudeme muset zdržovat psaním metod pro načítání informací o počasí a jejich parsování.

Aby mohla aplikace přistupovat k internetu, což je pro načítání informací o počasí nezbytné, je nutné upravit oprávnění v manifestu. Přidáme tedy tyto žádosti o povolení:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

První oprávnění umožní datovou komunikaci a druhé kontrolu zapnutí datových přenosů. Jenom pro úplnost dodejme, že manifest je už z dřívějška rozšířený také o odkaz na konfigurační soubor widgetu.

Postupně jsme se dostali k jádru aplikace, ke třídě WidgetProvider. Ta bude řídit chod celého widgetu. K jejímu fungování bude potřeba několik pomocných metod. Ty se budou volat ze dvou hlavních událostí – onUpdate(), která je spuštěna při instalaci a aktualizacích widgetu a nově onReceive(), vyvolané v reakci na události uživatelského rozhraní, v našem případě zejména při stisknutí jednoho ze dvou grafických tlačítek. Pojďme se podívat nejprve na pomocné metody.

Aby bylo možné zjistit, jaké místo má uživatel nastavené, bude potřeba metoda, která načte ze sdíleného souboru atribut misto:

```
private static String nastaveniMista () {
    SharedPreferences sharedPreferences = context.getSharedPreferences(
                                    "Nastaveni", context.MODE_PRIVATE);
    return sharedPreferences.getString("misto", "Karlovy Vary");
}
```

Metoda stejným způsobem, jaký jsme použili ve třídě MainActivity, načte nastavení místa. Pokud ho nebude moct načíst, použije přednastavenou hodnotu „Karlovy Vary“.

Dále bude potřeba metoda, která zobrazí požadovaný text na displeji. Může se jmenovat zobrazitText() a bude dělat to, že si nejprve vytvoří instanci typu RemoteView formuláře widgetu, potom na něm najde pole s velkým textem a do něj napíše řetězec předaný v parametru. Potom si najde pole s malým textem a do něj zapíše informaci o místě, kterou získá pomocí metody nastaveniMista(). K tomu doplní formátovanou informaci o aktuálním datu a čase. Nakonec instanci RemoteView zapíše na všechny aktivní widgety aplikace.

```
public static void zobrazitText (String text){
    RemoteViews remoteViews = new RemoteViews(context.getPackageName(),
                                        R.layout.widget);
    remoteViews.setTextViewText(R.id.txtPocasi, text);
    String aktualniCas=DateFormat.format("dd.MM. HH:mm", new Date()).toString();
    remoteViews.setTextViewText(R.id.txtDatum, nastaveniMista() + ", " +
                                                        aktualniCas);

    final int count = appWidgetIds.length;
    for (int i = 0; i < count; i++) {
        int widgetId = appWidgetIds[i];
        appWidgetManager.updateAppWidget(widgetId, remoteViews);
    }
}
```

Aby mohla aplikace spolehlivě datově komunikovat, bude potřebovat metodu pro ověření dostupnosti dat:

```
// Overi dostupnost datoveho pripojeni
private boolean datovePripojeni(Context context) {
    ConnectivityManager cm = (ConnectivityManager)
                        context.getSystemService(Context.CONNECTIVITY_SERVICE);
    if (cm == null)
        return false;

    // API verze 21 a vyssi
    if (android.os.Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        NetworkCapabilities capabilities =
                        cm.getNetworkCapabilities(cm.getActiveNetwork());
        if (capabilities != null &&
      (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR) ||
      capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) ||
      capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI_AWARE))
        )
        return true;
    }
    // API verze nizsi nez 21
    else {
        NetworkInfo netInfo = cm.getActiveNetworkInfo();
        return (netInfo != null && netInfo.isConnected());
    }
```

```
    zobrazitText("Datové připojení nedostupné.");
    return false;
}
```

Její princip asi není nutné příliš rozebírat, podobnou metodu jsme měli už v kapitole 20.3 Připojení k internetu, 1. příklad. Řekněme tedy pouze, že metoda vrací hodnotu true, pokud je datová komunikace možná, jinak na displeji zobrazí chybu a vrátí false.

Tlačítko na widgetu vlevo bude sloužit k otevření aktivity nastavení, aby uživatel mohl změnit místo i jindy, než jenom při instalaci widgetu. K tomu bude potřeba metoda volaná tímto tlačítkem:

```
private void otevritNastaveni() {
    // intent na tridu MainActivity a jeho spusteni
    Intent intent = new Intent(Intent.ACTION_MAIN, null);
    intent.addCategory(Intent.CATEGORY_LAUNCHER);
    ComponentName cn = new ComponentName(context, MainActivity.class);
    intent.setComponent(cn);
    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    context.startActivity(intent);
}
```

Metoda vytvoří intent typu CATEGORY_LAUNCHER, který se používá pro spouštění aktivit, přiřadí mu třídu MainActivity a aktivitu spustí.

Tlačítko pro aktualizaci údajů způsobí volání metody nacistDataZeServeru(), která je paradoxně jednodušší. Pouze vymaže obsah textových oken tak, že zavolá metodu zobrazitText(), zkontroluje datové připojení a pokud je k dispozici, tak vytvoří instanci třídy GetData s parametrem obsahujícím uživatelem zadané místo. Následně zavolá metodu execute() a o zbytek už se postará třída GetData.

```
private void nacistDataZeServeru() {
    // smazat predchozi informaci o počasí
    zobrazitText("");
    // spustit nacitani dat o počasí
    if (datovePripojeni() == true) {
        GetData getData = new GetData(nastaveniMista());
        getData.execute();
    }
}
```

Žádnou z obou popsaných metod, otevritNastaveni() ani nacistDataZeServeru(), ale nemůžeme volat přímo, musíme pro ně mít zaregistrovanou událost. Dostaneme se k tomu za chvíli.

Hlavní metodou třídy WidgetProvider je událost onUpdate(), volaná při instalaci a při aktualizaci widgetu:

```
private static AppWidgetManager widgetManager;
private static int[] widgetIds;

@Override
public void onUpdate(Context context, AppWidgetManager appWidgetManager,
                                                int[] appWidgetIds) {
    final int count = appWidgetIds.length;

    widgetManager = appWidgetManager;
```

```
        widgetIds = appWidgetIds;

        // cyklus - pro kazdou existujici instanci widgetu,
        // pripojenou k tomuto provideru
        for (int i = 0; i < count; i++) {
            int widgetId = widgetIds[i];

            // vytvorit instanci layoutu widgetu
            RemoteViews views = new RemoteViews(context.getPackageName(),
                            R.layout.widget);

            // vytvorit intent na tuto tridu a prida parametry
            Intent clickIntent = new Intent(context, WidgetProvider.class);
            clickIntent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, widgetId);
            clickIntent.setAction("btnNastaveniKlik");

            // vytvorit pending intent, ktery bude vyvolany po kliknuti na
            // tlacitko Nastaveni
            PendingIntent pendingIntent = PendingIntent.getBroadcast(context,
                            widgetId, clickIntent, PendingIntent.FLAG_IMMUTABLE);
            views.setOnClickPendingIntent(R.id.btnNastaveni, pendingIntent);

            // jako predchozi, ale pro tlacitko Obnovit
            Intent clickIntent2 = new Intent(context, WidgetProvider.class);
            clickIntent2.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
                            widgetId);
            clickIntent2.setAction("btnObnovitKlik");
            PendingIntent pendingIntent2 = PendingIntent.getBroadcast(context,
                            widgetId, clickIntent2, PendingIntent.FLAG_IMMUTABLE);
            views.setOnClickPendingIntent(R.id.btnObnovit, pendingIntent2);

            // zmeny odeslat widget manageru widgetManager
            widgetManager.updateAppWidget(widgetId, views);

        }
        nacistDataZeServeru();
    }
```

Událost nejdřív zjistí počet nainstalovaných instancí widgetů a dá si ho do proměnné count. Potom postupně projde všechny nainstalované instance a pro každou vytvoří instanci formuláře typu RemoteViews. Další kroky se točí kolem tlačítek. Založí intent třídy WidgetProvider, přidá mu parametr action s hodnotou btnNastaveniKlik – tu jsme si vymysleli, může být jakákoliv jiná, ale různá pro každé tlačítko. Později podle ní budeme detekovat, které z tlačítek bylo stisknuté a tuto hodnotu k tomu využijeme. Nato zaregistrujeme pomocí btnNastaveniKlik událost pro levé tlačítko. Jakmile ho uživatel stiskne, bude vyvolána událost onReceive(), ke které se dostaneme za chvíli.

Stejnou operaci uděláme i pro druhé tlačítko, změna je pouze v názvu parametru podle kterého se bude tlačítko identifikovat. Potom už jenom všechny tyto změny pomocí metody updateAppWidget() odešleme WidgetManageru. Nakonec zavoláme připravenou metodu pro načtení dat, nacistDataZeServeru().

Pro zpracování událostí vyvolaných widgetem je důležitá již zmíněná metoda onReceive(). Zavolá se při každé akci uživatelského rozhraní, pro nás je významná zejména kvůli tlačítkům. Jakmile uživatel tlačítko stiskne, událost si pomocí intent.getAction() zjistí obsah parametru action a na jeho základě zjistí, které tlačítko bylo stisknuté a zavolá buď metodu otevritNastaveni() nebo nacistDataZeServeru().

```
@Override
public void onReceive(Context context, Intent intent) {
    super.onReceive(context, intent);

    // podle nazvu udalosti rozlisit tlacitko
    String nazevUdalosti = intent.getAction();
    if (nazevUdalosti.equals("btnNastaveniKlik")) {
        otevritNastaveni();
    }
    else if (nazevUdalosti.equals("btnObnovitKlik")) {
        nacistDataZeServeru();
        Toast.makeText(context, "Počasí bylo aktualizováno.",
                                                Toast.LENGTH_SHORT).show();
    }
}
```

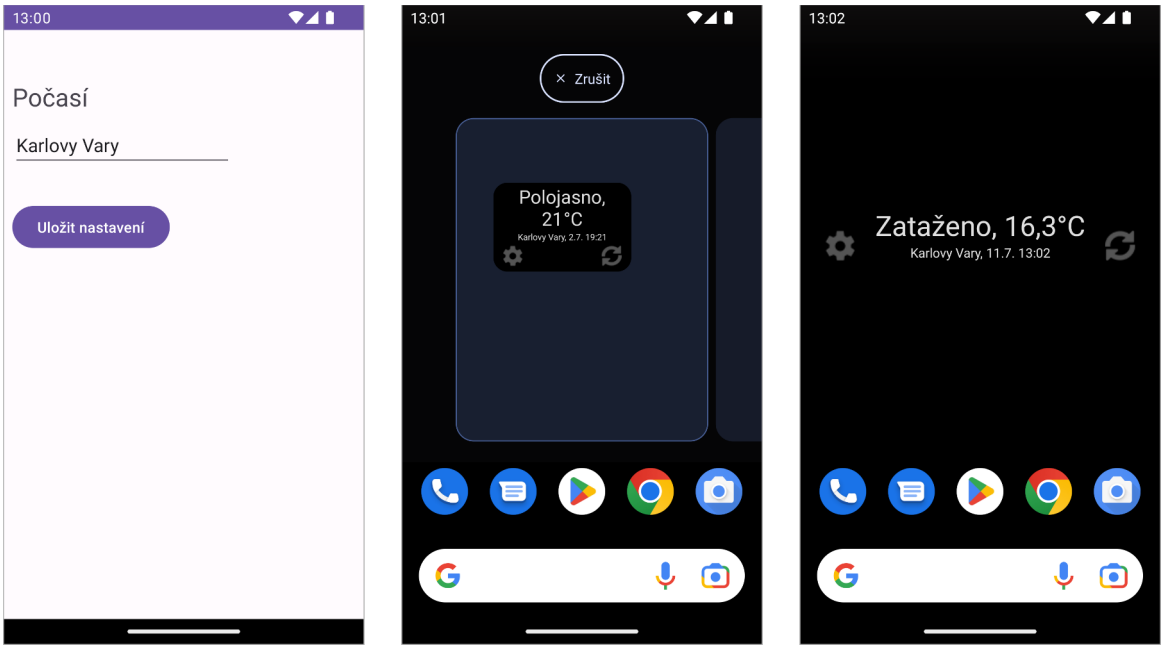
Aplikaci bychom mohli přiřadit standardní ikonu, tak, jak je popsané v kapitole 26 Ikona aplikace. A všechno by fungovalo. Ikona by byla jak na samotné aplikaci, tak i v seznamu widgetů a stejně tak i na náhledu widgetu během jeho instalace. Widget má nicméně možnost, pro účely instalace, přiřadit widgetu obrázek, který bývá nejčastěji náhledem reálného widgetu. Uživateli usnadní orientaci, když během instalace uvidí, jak bude widget reálně vypadat.

Jak takový náhled pořídit? Nejsnazší je widget spustit na emulátoru, z něj pořídit otisk obrazovky, oříznout a dát do projektu do složky app/res/drawable\. Totéž se dá udělat i na koncovém zařízení – widget nainstalovat, pořídit otisk obrazovky (postup se pro každý typ telefonu či tabletu liší), stáhnout do počítače, oříznout a uložit do projektové složky drawable. Nebo si lze obrázek v grafickém editoru nakreslit. Fantazii se meze nekladou. Vhodné je pouze dodržet poměr stran obrázku, měl by odpovídat poměru stran základní velikosti widgetu.

Přiřazení náhledu se udělá v konfiguračním xml souboru widgetu (název a umístění souboru je uvedené v manifestu v parametru meta-data) přidáním následujícího parametru do bloku appwidget-provider (‚nahled‘ je název obrázku):

```
android:previewImage="@drawable/nahled"
```

Po kompilaci a spuštění by se na displeji měl objevit formulář hlavní aktivity, určený pro uživatelská nastavení, viz obrázek vlevo. Zadáme v něm název místa, uložíme a formulář zavřeme. Nainstalujeme widget. Prostřední obrázek zachycuje instalaci widgetu s náhledem, který je upraveným otiskem nainstalovaného widgetu. A konečně na obrázku vpravo vidíme, jak by měl widget na ploše vypadat.



Obrázek 30.3: Nastavení widgetu a jeho instalace

31 Testování aplikací

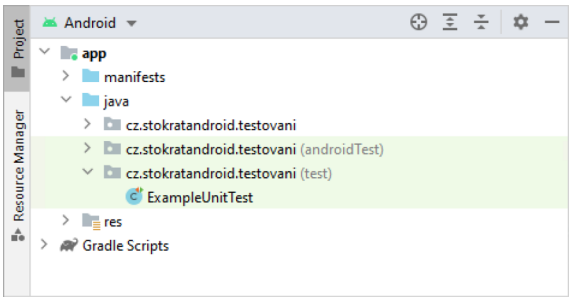
Nedílnou součástí vývoje aplikace, která je určena ke zveřejnění, je její zevrubné testování. O možnostech distribuce aplikace k uživatelům, kteří se na testování budou podílet, se zmíníme v kapitole 34.4 Google Play Store, nyní se zaměříme na programátorské testování kódu pomocí takzvaných unit a instrumented testů.

31.1 Unit testy

Termínem Unit test se rozumí naprogramovaný mechanismus pro rychlé a automatizované otestování stávající funkčnosti některé části aplikace. Unit test je obvykle prvním stupněm testování, píše si je sám programátor, kterému slouží k ověření základní jednotky (unity) kódu. Tou může být metoda, třída, nebo i celá knihovna. Obecně však platí, že je takto vhodné testovat co nejmenší izolované celky.

Zjednodušeně řečeno, vývojář si napíše testovací rutinu, která zavolá například metodu třídy. Té předá vstupní data a následně zkontroluje výsledek. A to všechno automaticky, pomocí programového kódu. Hlavním cílem je kontrola, že s postupným vývojem nedošlo k narušení původně zamýšlené funkčnosti metody.

Unit testy je nejlepší psát už během, nebo hned po vytvoření programového celku, u kterého se předpokládá, že bude potřeba navzdory budoucím zásahům udržovat původně předpokládanou funkcionalitu. Unit testy by měly být pokud možno jednoduché, ale zároveň též komplexně pokrývající primární funkce testovaného celku.



Obrázek 31.1: Projektové okno se souborem Unit testu ExampleUnitTest

Android Studio k tomu dává základní nástroje. V okně projektu je ve složce java kromě zdrojových kódů vygenerovaná také složka, která obsahuje vzorovou třídu ExampleUnitTest. Do této složky si lze přidávat vlastní třídy s testovacími rutinami. Složka není zahrnuta do finálního distribučního balíčku aplikace, svůj význam má pouze pro vývoj a kód v ní obsažený zůstává čistě v projektovém adresáři, nikam se nepřenáší.

Jestliže vývojář neplánuje využívat Unit testy, může jejich složku z projektu beztrestně odstranit. Ovšem pozor. Její návrat, pokud si to programátor rozmyslí, je sice možný, ale o poznání komplikovanější.

Příklad zjednodušené statické metody, kterou máme ve třídě MainActivity a kterou budeme moci následně pomocí Unit testu spouštět a ověřovat její funkčnost:

```
public class MainActivity extends AppCompatActivity {

      public static int testovaciMetoda (int param1, int param2) {
          int soucet = param1 + param2;
          return soucet;
       }
}
```

Metoda testovaciMetoda() nedělá nic komplikovaného. Vezme dvě čísla předaná v parametrech a prostě je sečte. Výsledek odevzdá v návratové hodnotě. Reálně taková metoda může být obsáhlá, ale pro naše účely teď postačí pouhé sečtení dvou čísel.

Ve třídě ve složce Unit testu napíšeme testovací rutinu:

```
public class ExampleUnitTest {

    @Test
    public void testovaciMetoda_isCorrect() {
        int vysledekTestu = MainActivity.testovaciMetoda(2, 3);
        assertEquals(5, vysledekTestu);
    }
}
```

Tím jsme založili testovací rutinu testovaciMetoda_isCorrect(). Ta zavolá připravenou statickou metodu třídy MainActivity s parametry ‚2' a ‚3'. Volaná metoda parametry sečte, vrátí výsledek a ten je porovnán s očekávanou hodnotou ‚5'.

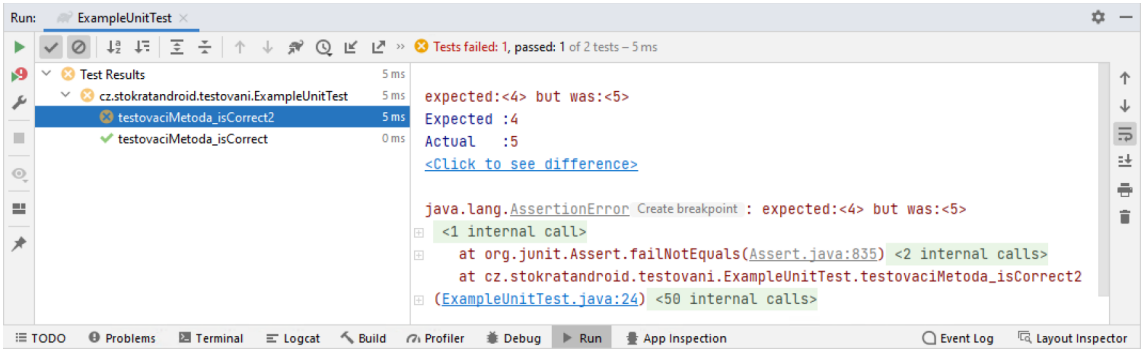
K tomu je potřeba říct, že pro vyhodnocení testu používáme klíčová slova typu assert. Například assertEquals znamená, že očekáváme výsledek rovný připravené hodnotě, v našem případě číslu ‚5'. Ale dá se použít také assertNotEquals, který zařídí, že bude očekávaný jakýkoliv jiný výsledek než je zadaná hodnota. Klíčových slov pro vyhodnocení je k dispozici celá řada, stačí si vybrat, co pro daný typ testu nejlépe vyhovuje.

Samotný test se spustí kliknutím pravého tlačítka myši na soubor v projektovém okně. Tedy například na ExampleUnitTest. Zvolíme buď Run nebo Debug, podle toho, jestli chceme testy jenom nechat proběhnout nebo je i ladit. Postupně jsou spuštěny všechny testy ve zvolené třídě. Výsledky se vypíší v okně Android Studia.

Můžeme přidat další Unit test, testovaciMetoda_isCorrect2(), stejný jako předchozí, pouze budeme úmyslně předpokládat, že má metoda s parametry 2 a 3 vrátit číslo 4:

```
@Test
public void testovaciMetoda_isCorrect2() {
    int vysledekTestu = MainActivity.testovaciMetoda(2, 3);
    assertEquals(4, vysledekTestu);
}
```

Po spuštění Unit testu dostaneme výsledek jako na obrázku 31.2. V jednom případě test proběhl podle očekávání, ve druhém jsme získali varování, že je někde něco v nepořádku. Po rozkliknutí lze zjistit podrobnosti. Pozor na nastavení výsledků, aby se zobrazily úspěšné i neúspěšné testy, je potřeba mít označeno filtrovací tlačítko v levém horním rohu okna.



Obrázek 31.2: Výsledek Unit testu. Jeden z testů proběhl bez chyby, druhý zjistil, že metoda vrací jiný než očekávaný výsledek

Jednou z nezanedbatelných výhod Unit testů je jejich rychlost. Start je po kompilaci téměř okamžitý a celkový čas provedení závisí pouze na testovaném kódu.

Další výbornou vlastností je, že běží na JVM (Java Virtual Machine) počítače, ke své funkci tedy nepotřebují mít připojené koncové zařízení ani spuštěný emulátor. Toho se dá mimo jiné využít i pro vývoj bloků kódu, jejichž ladění by bylo kvůli přenosu APK balíčků na zařízení zdlouhavé (byť se o to postará Android Studio samo). Pomocí Unit testu se jednoduše zavolá vyvíjený blok kódu a pokud ten sám o sobě nevyžaduje interakci s Androidem, tak jej lze hned začít ladit.

31.2 Instrumented testy

Instrumented testy jsou ve své podstatě Unit testy, které jsou ale spuštěné na koncovém zařízení nebo na emulátoru. Tím pádem jsou také mnohem pomalejší. Používají se pro komplexnější otestování aplikace. Kód testu má po spuštění k dispozici kontext aplikace. V projektovém okně se nachází ve složce hned vedle Unit testů a Android Studio pro ně standardně generuje do každého projektu vzorovou třídu ExampleInstrumentedTest.

Vzhledem k tomu, že je test spuštěný na zařízení, proběhne v podstatě stejně jako když se aplikace na zařízení spustí nebo ladí. Jen s tím rozdílem, že na konci opět výsledky porovná s očekávanými hodnotami. V zásadě bychom mohli jako instrumented test spustit i metodu pro sčítání, kterou jsme použili v minulé kapitole. A zpracovala by se úplně stejně. Ale nezískali bychom tím nic navíc. Zkusíme ji rozšířit o volání, které vyžaduje přítomnost API. Zeptáme se na verzi Androidu v zařízení. Vstupní parametry tentokrát nebudeme potřebovat:

```
public class MainActivity extends AppCompatActivity {

    public static int testovaciMetoda () {
        int verze = android.os.Build.VERSION.SDK_INT;
        return verze;
    }
}
```

Metoda by měla vrátit verzi API Androidu. Pokud bychom ji spustili Unit testem, tak by vzhledem k nepřítomnosti operačního systému koncového zařízení vrátila pouze hodnotu 0. V instrumented testu ale bude metoda spuštěna na zařízení, takže si verzi systému zjistit může. Předpokládejme, že test budeme spouštět na zařízeních, která budou mít vyšší verzi API než 20 a za správný výsledek testu tedy považujme návratovou hodnotu větší než 20.

```
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
    @Test
    public void testovaciMetoda_isCorrect() {

        Context appContext = InstrumentationRegistry.getTargetContext();

        int vysledekTestu = MainActivity.testovaciMetoda();
        assertTrue(vysledekTestu > 20);
    }
}
```

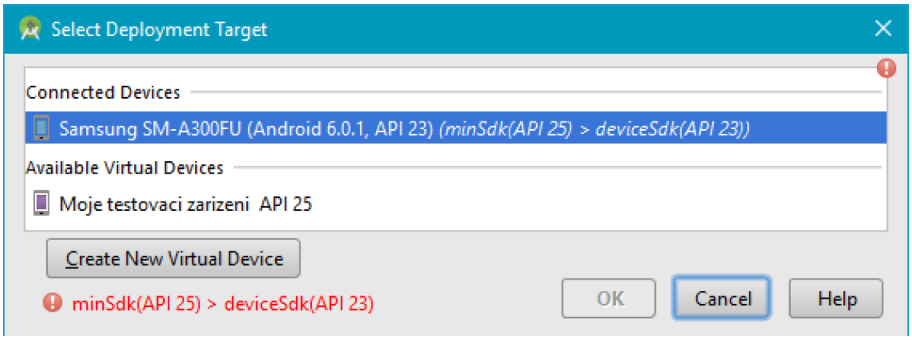
V testu zůstal kontext. Mohli bychom ho využít, ale pro účely takto jednoduchého příkladu není potřeba. Vyhodnocení uděláme pomocí assertTrue, kdy splnění podmínky, tedy v našem případě vrácená hodnota vyšší než 20, znamená úspěšně provedený test. Výsledek průběhu testu bude v Android Studiu vypadat podobně jako v případě Unit testu.

32 Několik praktických rad a postupů

Během práce se zdrojovými kódy je nezřídka potřeba měnit názvy aplikací, čísla verzí, přesouvat hotový kód na disku, nebo ho distribuovat. Tyto zdánlivě jednoduché operace však můžou narazit na nepochopení Android Studia, které potřebuje mít nad aplikací z mnoha důvodů alespoň částečnou kontrolu. Při dodržení jednoduchých postupů se však dá kolizním situacím vcelku snadno vyhnout.

32.1 Změna verze API

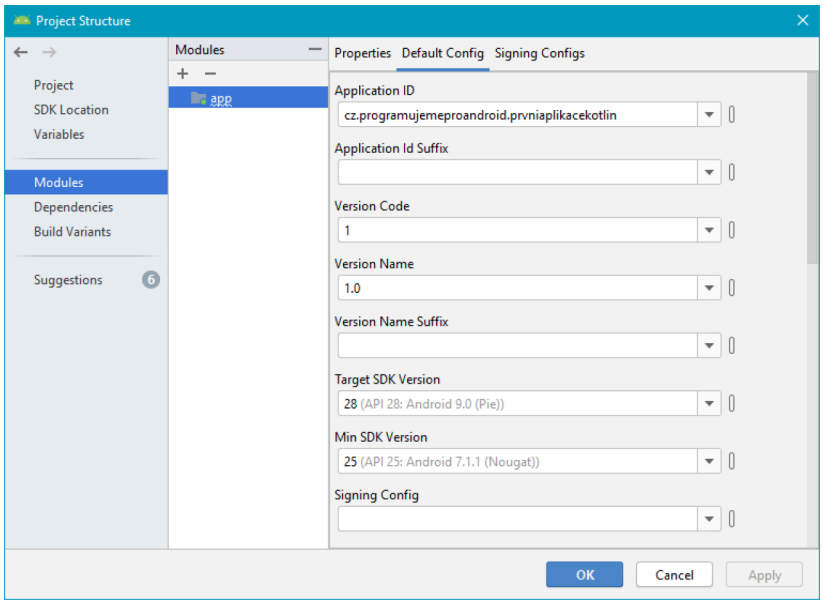
Během programování se můžou stát dvě nemilé věci, související s verzí API nastavenou při založení projektu. Vývojář zjistí, že potřebuje použít vyšší API než kterou si nastavil jako minimální, nebo naopak potřebuje spustit aplikaci na zařízení s nižším API, než je ono minimální.



Obrázek 32.1: Upozornění Android Studia na nesoulad verze API při spouštění aplikace

V obou případech může buď oželet možnosti, kterých se mu nedostává a zůstat u nastavené verze, nebo může verzi API změnit. Změna není složitá, náročnější je pouze hledání nastavení v prostředí Android Studia.

Otevřeme okno Project Structure, v menu File → Project Structure, zvolíme Modules, označíme modul app a vybereme záložku Default Config. Na ní se dá nastavit jak minimální, tak cílová verze API. Potom už jen stačí projekt znovu přeložit.

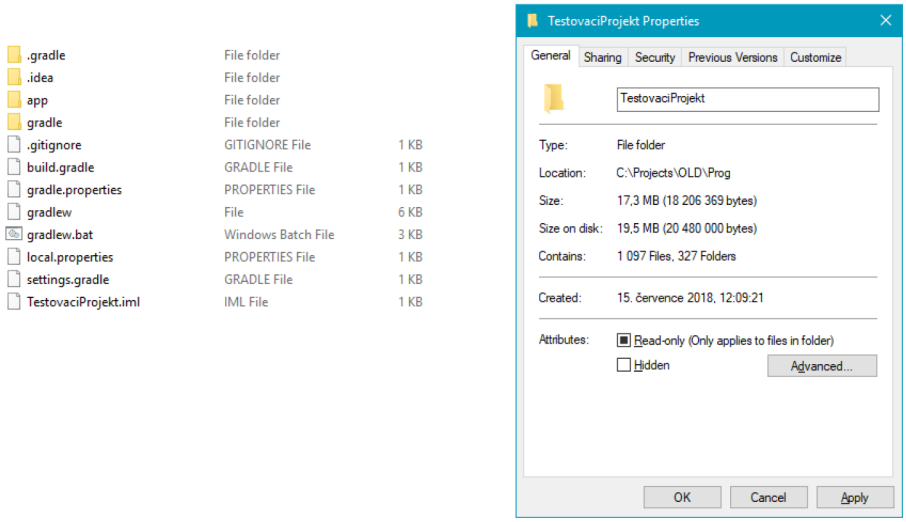


Obrázek 32.2: Nastavení minimální a cílové verze API v Android Studiu

32.2 Distribuce zdrojových souborů

Adresář aplikace na disku počítače obsahuje často tisíce souborů, které mají nezanedbatelnou velikost. Pokud je potřeba zdrojová data někam přenést, nebo je dát ve formě balíčku k dispozici dalším vývojářům, není úplně vhodné přenášet celou strukturu projektu. Většinu obsahu totiž tvoří různé binární a konfigurační soubory, které jsou po přenosu tak jako tak nepoužitelné a musí se přegenerovat.

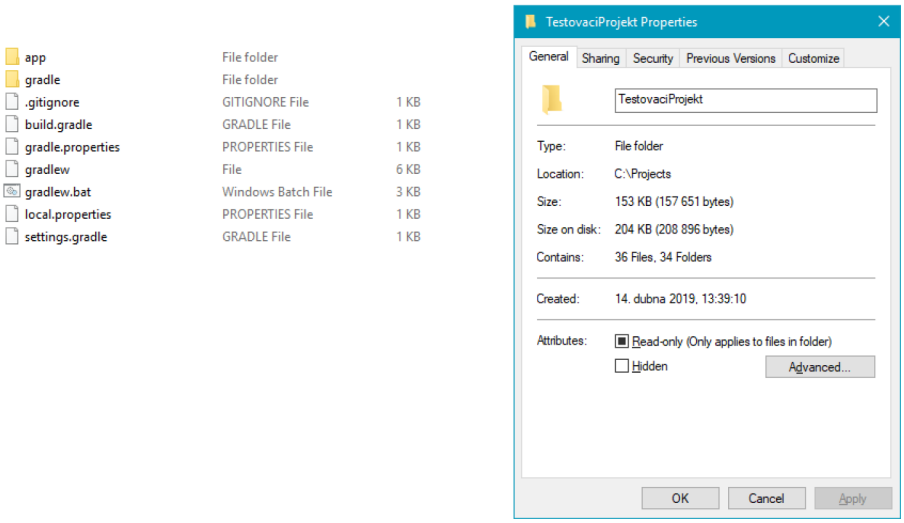
Co tedy projektový adresář obsahovat musí a co z něj můžeme odstranit aniž bychom projektu ublížili? Úplně jednoduchý projekt, který obsahuje minimum zdrojových kódů, vypadá jako na obrázku . Úhrnnou velikost má kolem 20 MB.



Obrázek 32.3: Projektová složka čerstvě založené aplikace. Obsahuje kolem tisíce souborů a má velikost cca 20 MB.

Distribuovat není nutné systémové soubory Gradle a Idea, tj. obsah dvou adresářů, které začínají tečkou. Můžeme je celé smazat. Dále zaměříme svoji pozornost na složku app. Uvnitř najdeme podadresář build, který obsahuje soubory potřebné k sestavení a samotnou sestavenou aplikaci. To ovšem se zdrojovými soubory také nepotřebujeme nikam přenášet. Smažeme tedy vnořený adresář build.

A to je vše. Tím jsme jednoduchým způsobem získali strukturu jako na obrázku 32.4, ušetřili jsme místo a zredukovali množství souborů na rozumné minimum. Teď už lze projektovou složku zabalit, zkomprimovat a přenést kam potřebujeme. Po jejím rozbalení a otevření v Android Studiu bude nutné udělat nový build aplikace. To už by ale mělo proběhnout bez dalších komplikací.



Obrázek 32.4: Zredukovaná projektová složka. Obsahuje už jenom několik desítek souborů o velikosti stovek kilobajtů.

32.3 Přemístění projektu

Přemístěním projektu se myslí změna cesty k projektovým souborům. Tedy přesun projektu na disku počítače, nebo přejmenování složky, ve které se nachází.

Při takové operaci je pochopitelně nutné, aby projekt nebyl aktuálně otevřený v Android Studiu. Projekt podle potřeby přemístíme nebo složku přejmenujeme – musí se přitom ale dát pozor,

aby cesta neobsahovala mezery. Android Studio je na to citlivé a při zakládání nového projektu vývojáře upozorní, v tomto případě je ale vše v našich rukou. Stejně tak je nevhodné používat jiné než ASCII znaky, čili zejména znaky s diakritikou.

Poté můžeme projekt otevřít v Android Studiu a projekt znovu sestavit (menu Build → Rebuild Project). V ideálním případě je tím vše vyřízeno.

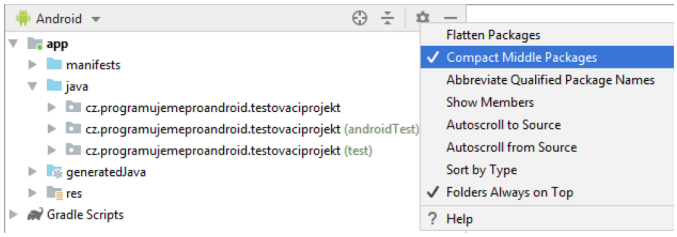
U rozsáhlejších projektů se ale může stát, že si Android Studio s přesunem samo neporadí a začne během sestavování vypisovat chyby, způsobené novým umístěním již dříve zkompilovaných souborů. V takovém případě ukončíme Android Studio, v projektové složce promažeme soubory tak, jak je popsáno v kapitole 32.2 Distribuce zdrojových souborů. Po znovuotevření projektu v Android Studiu a jeho novém sestavení by mělo být ve stejném stavu, v jakém byl před zahájením přesunu.

32.4 Přejmenování projektu

Během práce na projektu se může snadno stát, že je potřeba změnit jeho název nebo použitý jmenný prostor (namespace). Změna názvu aplikace je vcelku snadná. Stačí v manifestu najít řetězec s jejím jménem a ten přepsat. Pokud bude potřeba změnit současně název adresáře, ve kterém je projekt uložený, použijeme postup popsaný v kapitole 32.3 Přemístění projektu.

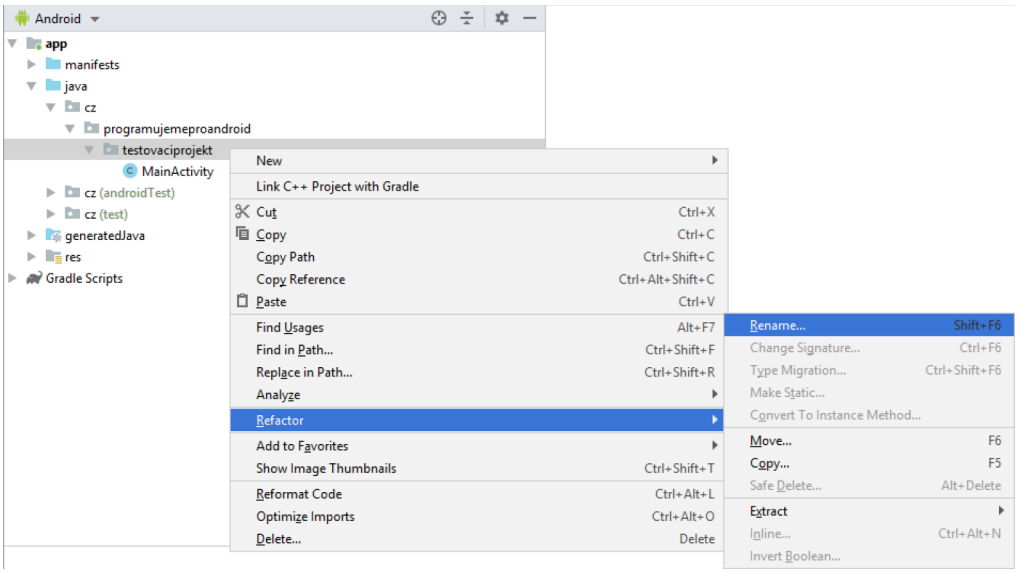
Trošku složitější je to se změnou namespace projektu. Ten je uložený nejen v manifestu, ale nachází se i v dalších projektových souborech. Pro tento účel má Android Studio funkci přejmenování, kterou najdeme v kontextovém menu.

Pokud je struktura projektu zobrazena kompaktně, tj. celý namespace pohromadě, tak je ho potřeba nejdřív přepnout do režimu zobrazení po samostatných úrovních. V okně projektu je pro přepínání určena položka pod ikonkou nastavení Compact Middle Packages, viz obrázek 32.5.



Obrázek 32.5: Přepnutí zobrazení projektu z kompaktního na zobrazení po úrovních.

V okně projektu si otevřeme strukturu projektu, označíme tu část namespace, kterou chceme změnit a v kontextovém menu zvolíme položku Refactor → Rename. Pokud nás Android Studio upozorní, že názvy složek souborů odpovídají názvu balíčku (package) a bude chtít vědět, jestli chceme změnit název složek nebo celého balíčku, volíme „Rename package“. Dosáhneme tím současného přejmenování všech částí projektu.



Obrázek 32.6: Změna názvu projektu pomocí refaktoringu.

Potom je potřeba zadat nové jméno, změnu potvrdit a Android Studio zavede nový namespace v projektových souborech. Bohužel to často nezafunguje úplně stoprocentně, Android Studio nezřídka vynechává některé méně významné soubory, například určené pro testování. Takže se doporučuje ještě nechat prohledat všechny soubory projektu na původní název a pokud někde zůstal, tak ho ještě opravit ručně. K tomu se dá využít nástroj pro hledání napříč soubory, vyvolaný kombinací kláves CTRL+SHIFT+F.

Jakmile jsou všechny použité namespace přejmenované, je ještě vhodné projekt znovu sestavit (menu Build → Rebuild Project). V případě nouze, kdyby něco navzdory novému sestavení nezafungovalo, může pomoci vypnutí Android Studia a smazání adresářů pro sestavení, tak, jak je popsáno v kapitole 32.2 Distribuce zdrojových souborů.

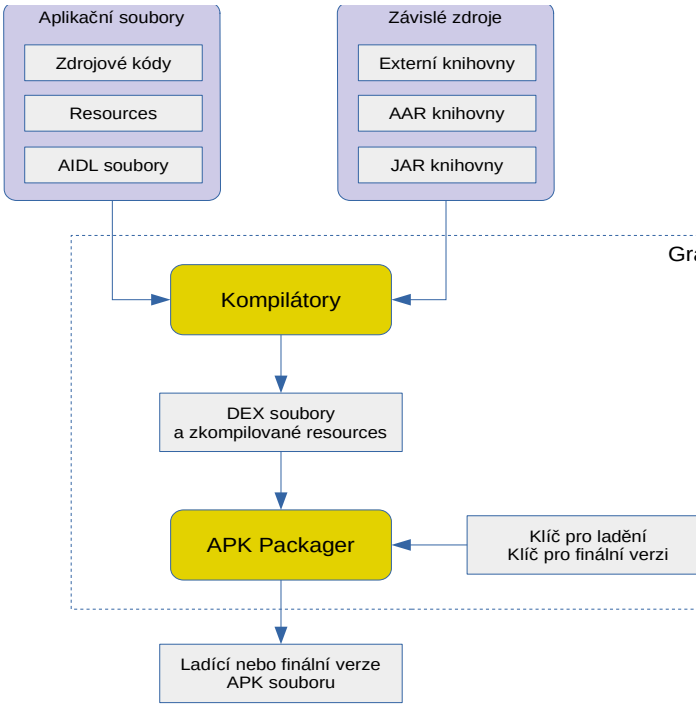
33 Gradle

V Android Studiu vývojáře provází na každém kroku Gradle (vyslovuje se greidl) – součást studia, která se stará o sestavení aplikace. Tento nástroj byl původně vyvinutý pro automatické sestavování velkých javových multiprojektových celků a díky tomu, že je šířený pod svobodnou licencí (Open Source) se později stal také standardní součástí Android Studia.

Obecně se umí Gradle postarat o kompilaci, sestavení, připojení knihoven, umí vytvářet několik různých verzí aplikace (například verze zdarma a placená) a to všechno na jedno kliknutí. Akce, které má Gradle vykonat jsou popsány pomocí skriptů, které může vývojář upravovat a tím proces sestavování přizpůsobovat svým potřebám. Je ale nutné dodat, že skripty jsou napsané v sice jednoduchém, přesto speciálním jazyku, Groovy (vyslovuje se gru:vy).

Pro nás je podstatné, že se Gradle postará i bez nastavování o vše potřebné sám a že stačí stisknout tlačítko Make project nebo Run app. Při tvorbě většiny jednodušších aplikací si lze vystačit s předdefinovaným nastavením, bez nutnosti jakýchkoliv úprav. Přesto se může hodit znát princip sestavování a alespoň základní možnosti konfigurace Gradle.

Jednoduché schéma sestavení aplikace pomocí Gradle:



Obrázek 33.1: Sestavení aplikace pomocí Gradle

Na samém začátku jsou zdrojové kódy v adresáři java, formuláře a další zdroje ve složce res a případně AIDL soubory (zdroje pro komunikaci mezi aplikacemi nebo mezi serverem a klientem). Čili vše, co programátor připraví v Android Studiu. K nim jsou připojené externí knihovny, které se vlastním zdrojovým souborům přidávají pomocí direktivy „import“ na začátku každé Java třídy. Projekt může obsahovat také AAR knihovny (Android ARchive - vlastní knihovny, v Android balíčku) a JAR knihovny (Java ARchive - vlastní knihovny, v Java balíčku).

Toto všechno jsou vstupní soubory pro Gradle. Ten je pomocí Java kompilátoru přeloží a vytvoří z nich zvlášť soubory se zdroji a zvlášť binární soubory, spustitelné na Android zařízeních. Ukládá je ve formátu DEX (Dalvik Executable) do struktur ve složce app\build\intermediates\transforms\.

Pokud se podaří vše řádně zkompilovat, nastává druhá fáze, příprava balíčku. Gradle všechny soubory zabalí do jednoho archivu, zkomprimovaného pomocí běžného zip algoritmu. Balíček je typu APK (Android Package Kit) a obsahuje všechny soubory, které jsou potřebné pro spuštění aplikace na Android zařízení.

Z bezpečnostních důvodů, aby se předešlo neoprávněným modifikacím, je balíček ještě nutné digitálně podepsat. Bez toho by jej Android zařízení nepřijalo. Gradle tedy použije, podle účelu kompilace, k podpisu buď klíč pro ladění nebo pro finální verzi aplikace.

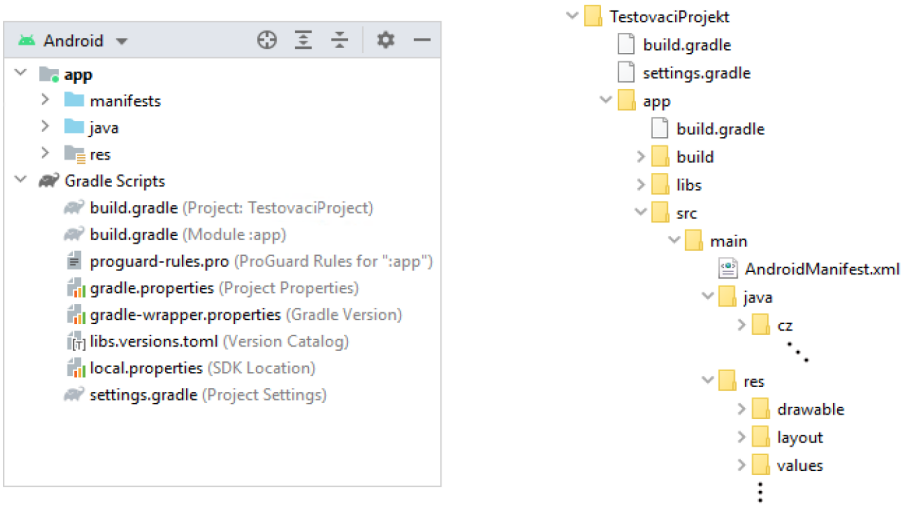
Klíč pro ladění má Android Studio vlastní, v úložišti digitálních klíčů zvaném Keystore a není se jím nutné nijak zabývat. Gradle ho použije, aniž by tím vývojáře nějak zatěžoval. Při generování finální verze aplikace, takzvaném release, je nutné mít v Keystore uložený také release klíč. Je-li tam už z dřívějška, Gradle ho jednoduše použije. A pokud tam není, řekne o jeho nahrání. Klíč se získá registrací u Google.

Posledním krokem je optimalizace balíčku pomocí nástroje zipalign, s cílem dosáhnout co nejnižších paměťových nároků při použití archivního souboru v zařízení.

Výsledkem procesu je soubor s koncovkou .apk, který Gradle uloží do složky app\build\outputs\apk\. Během ladění aplikace Gradle spustí AVD Manager (Android Virtual Devices Manager), který se postará o přenos a spuštění aplikace na virtuálním nebo fyzickém zařízení.

33.1 Konfigurace Gradle

Konfigurační soubory jsou přístupné v Android Studiu v okně projektu, takže působí dojmem, jako by byly umístěny v souborovém systému na jednom místě. Tak tomu ale není, na disku jsou zanořeny v projektových adresářích podle toho, kam logicky spadají. Pro snazší pochopení účelu je vhodné vědět, kde se ten který soubor fyzicky nachází.



Obrázek 33.2: Konfigurační soubory Gradle. Vlevo v okně projektu, vpravo na disku počítače.

Účel jednotlivých konfiguračních souborů:

build.gradle (Project) – v kořenovém adresáři projektu, je v něm definovaný způsob sestavení, který se vztahuje na všechny připojené moduly projektu.

build.gradle (Module) – v adresáři modulu (nejčastěji tedy app\), obsahuje nastavení pro konkrétní modul. Nastavení uvedená v tomto souboru mají vyšší váhu než nastavení v projektovém build.gradle nebo v manifestu. Pokud dojde ke kolizi, je použito nastavení z tohoto souboru.

proguard-rules.pro - v adresáři app\, nastavení nástroje proGuard, který slouží k optimalizaci velikosti zdrojových kódů.

gradle.properties – v kořenovém adresáři projektu, zde jsou technická nastavení, důležitá pro běh samotného Gradle.

gradle-wrapper.properties - v adresáři gradle\wrapper\, slouží k nastavení aktualizací nástroje Gradle.

libs.versions.toml – v adresáři gradle\, seznam verzí použitých knihoven. Samotné knihovny jsou k projektu připojeny v souboru build.gradle (Module)

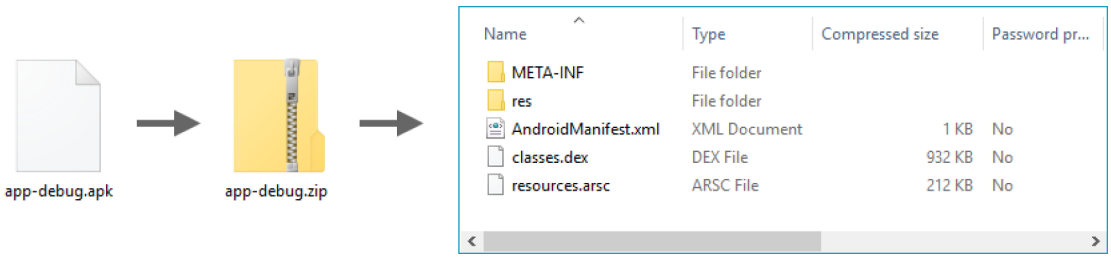
local.properties – v kořenovém adresáři projektu, obsahuje informace o prostředí. Obvykle pouze umístění SDK na disku počítače.

settings.gradle – v kořenovém adresáři projektu, říká, které moduly má Gradle při sestavování připojit. U jednoduchých projektů obsahuje obvykle pouze informaci o připojení modulu app, tedy:

```
include ':app'
```

33.2 APK balíček

Pokud by někoho zajímalo, co přesně Gradle vygeneroval, je možné APK balíček podrobit dalšímu zkoumání. Stačí změnit příponu z .apk na .zip a ze souboru se v mžiku stane archiv. Ten se dá otevřít a prohlédnout.



Obrázek 33.3: Obsah APK balíčku

Obsah balíčku se může v závislosti na struktuře projektu lišit, základní složení však bývá v zásadě stejné, viz tabulka 33.1.

Balíček APK (Android Package)		
složka metadat META-INF	soubor MANIFEST.MF	seznam souborů v balíčku (pozor, nejde o manifest který známe z Android Studia), u každého souboru je jeho hash (digitální otisk) vytvořený algoritmem SHA-1
	soubor CERT.SF	seznam souborů v balíčku; seznam je stejný jako v

Balíček APK (Android Package)		
		MANIFEST.MF, ale u každého souboru je uvedený jeho digitální podpis, vytvořený algoritmem SHA-1
	soubor CERT.RSA	certifikát aplikace
	další soubory s informací o verzích knihoven	
složka lib	zkompilovaný kód v samostatných verzích pro jednotlivé, podporované hardwarové platformy	
složka res	všechny zdroje projektu, s výjimkou předkompilovaných zdrojů umístěných v souboru resources.arsc	
soubor AndroidManifest.xml	doplňkový manifest aplikace ve formátu binárního XML (redukovaný XML formát, úspornější a strojově snadno dekódovatelný)	
soubor classes.dex	třídy projektu zkompilované do formátu dex	
soubor resources.arsc	předkompilované zdroje, jako jsou například binární XML	

Tabulka 33.1: Struktura balíčku APK

Znalost struktury balíčku má význam snad jen pro zpětnou identifikaci projektu nebo pro zjišťování, proč případně nezafungovalo něco v procesu jeho distribuce. Zasahovat do jeho obsahu však nedává příliš smysl, v podstatě se mu tím dá jenom ublížit.

34 Instalace a distribuce aplikací

Aplikace je možné do zařízení dostat několika způsoby. Dobře už známe instalaci prostřednictvím ADB (Android Debug Bridge), který je součástí Android Studia a aplikaci do zařízení umí dopravit automaticky. Dalším způsobem je distribuce APK balíčku a pochopitelně uživateli nejpoužívanější způsob, prostřednictvím virtuálního obchodu Play Store.

34.1 Verze aplikace

Aby v jednotlivých vydáních aplikace nebyl zmatek, je potřeba každý balíček, který opustí vývojové prostředí, řádně očíslovat. Je to mimořádně důležité zejména v okamžiku, kdy dojde na jeho distribuci prostřednictvím Play Store. Čísla verzí dávají pocit jistoty uživatelům, kteří z nich při aktualizaci můžou intuitivně odvodit, jestli jde o významnou nebo méně významnou změnu, při zpětné vazbě se dá díky číslu verze zjistit, kterého sestavení se vlastně připomínka týká a při update aplikace je možné nastavit konverzi uživatelských dat v závislosti na číslu původní verze. V případě komponent nebo aplikací, které vzájemně komunikují, slouží číslo verze také k ověření vzájemné kompatibility, resp. schopnosti předávat si data.

Android má dvě základní čísla verze. První je versionCode. Jde o prosté číslo typu integer, které stačí s každým vydáním jednoduše inkrementálně zvýšit. Při prvním vydání má hodnotu 1 a potom už se jen roste. Toto číslo je pro uživatele neviditelné, ale aplikaci umožňuje snadnou orientaci při update nebo při vzájemné komunikaci. Zároveň slouží jako vodítko při publikování na Google Play, kde se pomocí něj ověřuje, že je zveřejňovaná verze skutečně novější než všechny předchozí.

Číslo verze není bezpodmínečně nutné zvyšovat po jedné. V každém případě je ale potřeba dávat pozor aby nepřekročilo horní mez, která je nastavena na 2 100 000 000.

Druhé číslo verze je versionName. Je obvykle hierarchické, uložené v textovém formátu, uživatel ho vidí a je vhodné toto číslo udržovat v logické a snadno pochopitelné podobě. Kromě zobrazení uživateli však nemá žádný další zásadní význam.

Pro nastavení versionName existuje hned několik možných strategií. Ty můžou být buď prostým několikaúrovňovým číslováním, můžou nést informaci o minimálním API, o rozlišení displeje, o typu sestavení (alfa, beta, rc atd.) nebo o datu vydání.

Asi nejčastěji používanou technikou je MAJOR.MINOR.PATCH, neboli sekvence tří cifer, z nichž první (MAJOR) označuje zásadní verzi aplikace, obvykle závislou na verzi API. Když dojde ke změně minimální API, zvyšujeme i toto číslo. Druhá cifra (MINOR) označuje přidání funkcionality a třetí (PATCH) opravy chyb nebo přidání malé funkcionality, která nemá vliv na zpětnou kompatibilitu. Takže například verze 1.0.2 znamená první vydání, bez rozšiřujících funkcionalit, se dvěma opravami/vylepšeními.

Čísla verzí jsou uložena v souboru dostupném v okně projektu Gradle Scripts -> build.gradle (Module:app) a při sestavení se automaticky nakopírují do manifestu obsaženém v distribučním balíčku. Zápis v build.gradle může vypadat například takto:

```
android {
    defaultConfig {
```

```
        ...
        versionCode 3
        versionName "1.0.2"
        ...
    }
    ...
}
```

Což v tomto případě znamená, že jde o vydání číslo 3, s uživatelským označením 1.0.2. Verze můžou být zapsané také přímo v manifestu. V takovém případě mají přednost před zápisem v build.gradle. Tento postup ale není obecně doporučovaný.

Číslo verze lze zjistit pomocí třídy Package Manager následujícím způsobem:

```
PackageManager pm = getPackageManager();
Long cisloVerze = pm.getPackageInfo(getPackageName(), 0).getLongVersionCode();
String nazevVerze = pm.getPackageInfo(getPackageName(), 0).versionName;
```

34.2 Vygenerování APK balíčku

Každá aplikace, která se instaluje do Android zařízení, musí obsahovat digitální podpis. Android Studio má svoje vlastní úložiště podpisů, takzvaný KeyStore a v něm má podpisový klíč pro účely ladění – debug key. Kdykoliv ADB potřebuje nahrát aplikaci do zařízení, zkompiluje ji a podepíše pomocí tohoto klíče. Aby mohl vývojář aplikaci distribuovat, musí ji mít podepsanou vlastním distribučním klíčem - release key.

Cílem podepisování aplikací je zvýšení zabezpečení. A to jednak o možnost prokázání původu aplikace a identity zdroje, tedy autora nebo vydavatele. Druhým důvodem je zamezení neautorizovaným modifikacím vydané aplikace předtím, než se od autora dostane do koncového zařízení.

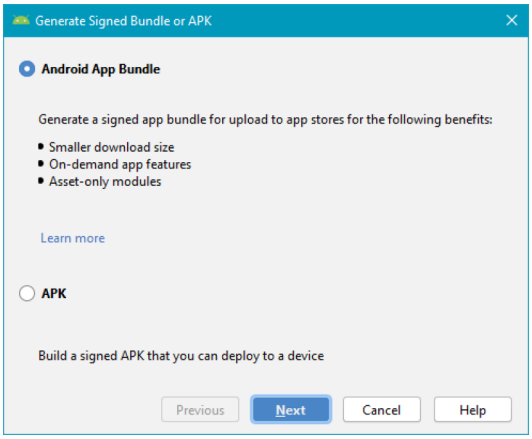
Stejně tak v případě aktualizace aplikace zařízení nejprve ověří, jestli se její podpis shoduje s podpisem v aplikaci a pokud ne, aktualizaci odmítne jako neautorizovanou. Z tohoto důvodu je nutné mít klíče bezpečně uložené, při jejich ztrátě vývojář přijde o možnost provádět aktualizace vlastních aplikací. V takovém případě mu nezbyde nic jiného, než vydat zcela novou aplikaci, podepsanou pomocí nového klíče.

Android Studio a Google podporují kromě standardních instalačních APK balíčků rovněž generování takzvaných AAB (Android App Bundles) balíčků. Je to speciální verze distribučního balíčku, který není určen k přímému nainstalování na zařízení, ale k přenosu do Play Store, kde se transformuje v několik verzí APK instalátoru pro hardwarově různá zařízení a lokalizace. AAB balíček obsahuje více dat, je proto větší. Z hlediska koncových uživatelů je ale stažený a instalovaný balíček naopak menší, s lepší optimalizací pro prostředí do kterého je instalován.

Klíče je možné nahrát na Google Play a tím snížit riziko jejich ztráty. Aplikace jsou potom podepisované až při přenosu do PlayStore. Balíčky se v takovém případě podepisují pouze pomocí nahrávacího klíče (upload key), který je možné v případě nouze nechat resetovat a nahradit novým.

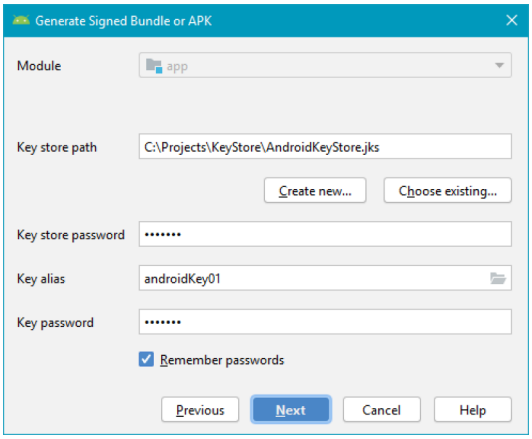
Pokud nemáme již z dřívějška založené vlastní úložiště a vygenerovaný distribuční klíč, povedou naše první kroky k jejich pořízení. V menu Android Studia zvolíme položku Build

→ Generate Signed Bundle or APK. Android Studio dá uživateli vybrat, jestli chce vygenerovat AAB nebo APK balíček.



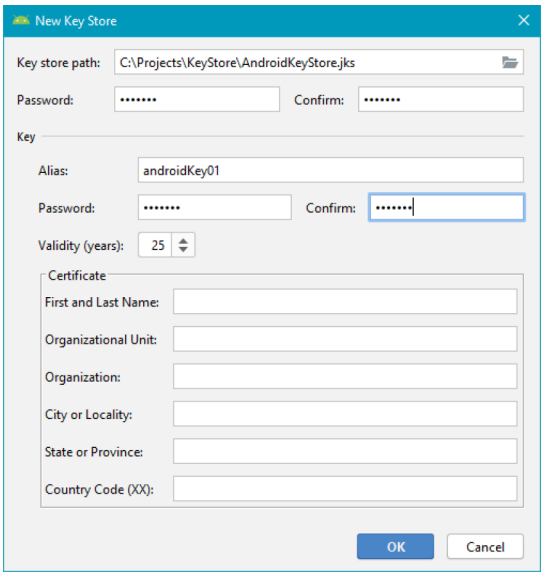
Obrázek 34.1: Generování AAB nebo APK balíčku

Ať už zvolíme AAB nebo APK, v dalším kroku musíme nastavit cestu k úložišti klíčů Key Store. Pokud ho zatím nemáme, nastal pravý čas pro jeho založení. Stejně tak, pokud zatím nemáme vlastní klíč. Z praktických důvodů se obojí zakládá pomocí tlačítka Create New.



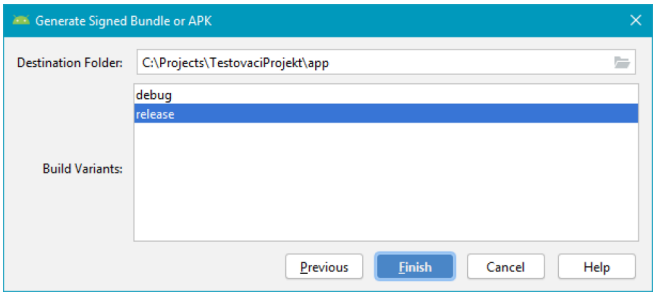
Obrázek 34.2: Nastavení veřejného klíče

Pro vytvoření nového úložiště stačí zadat cestu ke složce, kam bude uložen, název souboru s koncovkou .jks a nové heslo pro přístup do úložiště. Se samotným klíčem je to o něco složitější, kromě názvu a hesla je potřeba zadat také údaje o autorovi. Tyto údaje se stanou součástí certifikátu uloženého do balíčku, nebudou však zobrazeny uživatelům v Play Store.



Obrázek 34.3: Vytvoření úložiště klíčů a veřejného klíče

Posledním krokem je volba verze generované aplikace. Po kliknutí na Finish vytvoří Android Studio instalační nebo distribuční balíček a uloží ho do projektové složky.



Obrázek 34.4: Vygenerování balíčku

34.3 Instalace APK balíčku

Sestavení a strukturu APK balíčku, který je jakýmsi instalátorem Android aplikace, jsme prošli v předchozích kapitolách. Je nasnadě, že se takový balíček dá použít k distribuci vlastní aplikace, třeba za účelem testování, pro širší okruh zájemců.

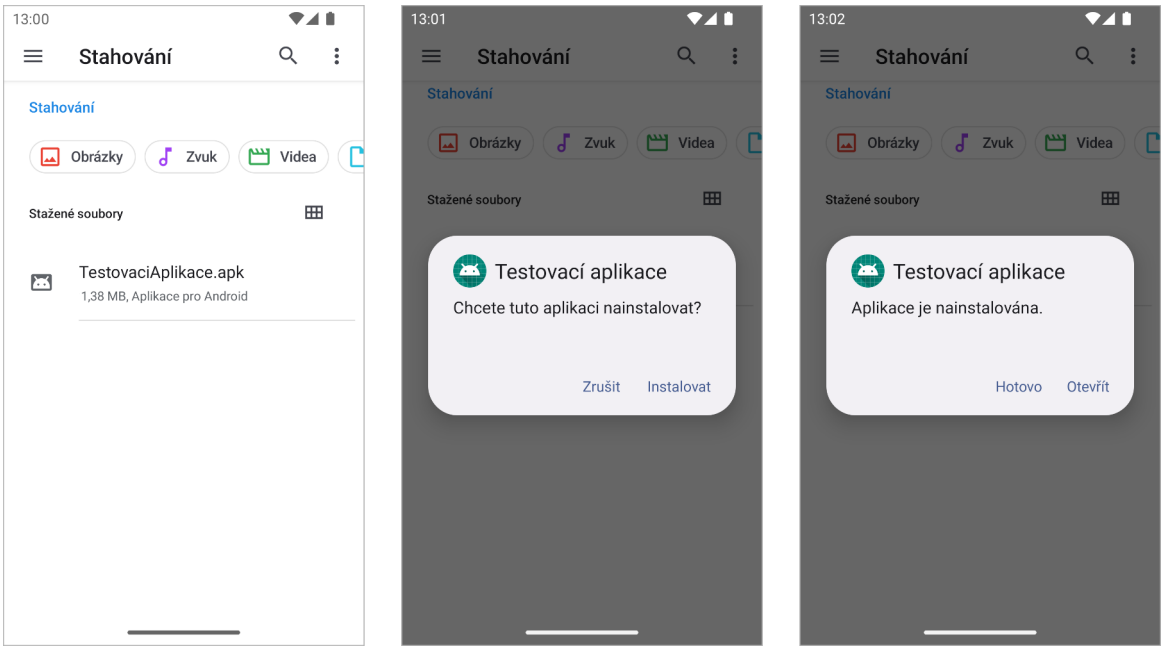
Přirozený a zároveň nejběžnější způsob distribuce aplikace přes Google Play Store prozkoumáme v příští kapitole. Kromě tohoto způsobu je totiž možné také dát APK balíček k dispozici zájemcům přímo, nebo prostřednictvím alternativních distributorů Android aplikací. Na internetu je není nijak těžké najít, k dispozici je jich celá řada. A ačkoliv se používání alternativních zdrojů aplikací z důvodu bezpečnosti příliš nedoporučuje, lze je k distribuci vlastního díla samozřejmě s výhodou využít.

Uživatel, který dostane, nebo si stáhne aplikaci ve formě APK balíčku musí mít v systému Android povolenou instalaci aplikací z externích zdrojů. Ta se nachází v Nastavení → Zabezpečení → Neznámé zdroje.

Potom je potřeba APK balíček umístit do úložiště v zařízení, buď stažením z alternativního internetového zdroje - v takovém případě bude balíček ve složce stažených souborů, odkud většinou jde instalaci rovnou spustit. Nebo lze nahrát balíček do zařízení z PC. V takovém případě je ho vhodné umístit do některé ze složek, které jsou snadno přístupné jak z počítače, tak z prostředí Androidu.

Kdo nemá v zařízení nainstalovanou aplikaci pro procházení souborového systému, může použít k nalezení APK balíčku a spuštění instalace vestavěnou aplikaci pro procházení úložiště v menu *Nastavení → Úložiště*. Nemusí to ale fungovat stejně ve všech verzích Androidu..

Instalace se spustí poklepáním na APK balíček, viz obrázek 34.5. Vše ostatní už proběhne víceméně automaticky. Stejným způsobem se dá později provést aktualizace aplikace na novější verzi.

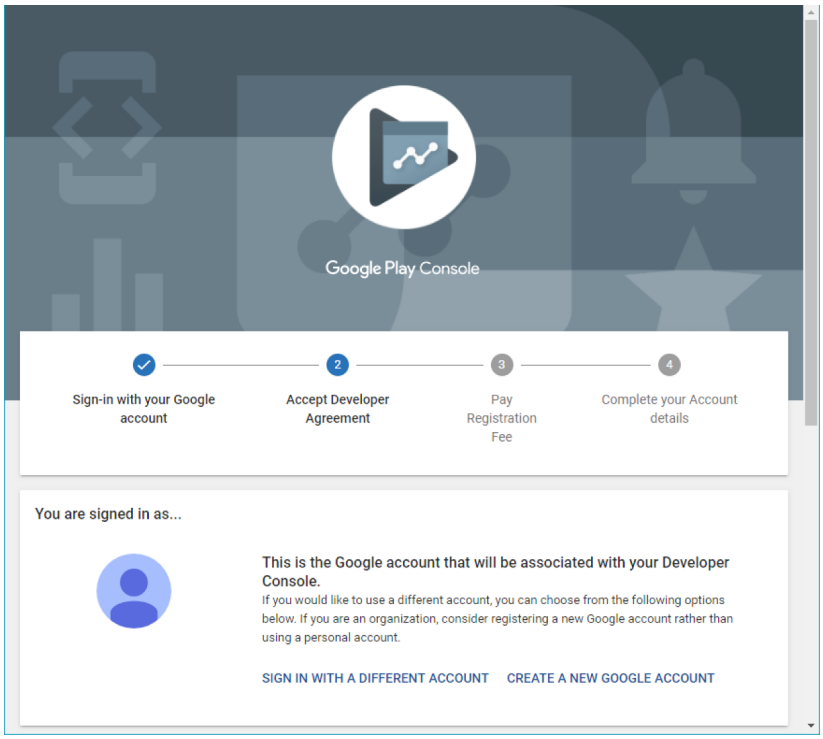


Obrázek 34.5: Instalace aplikace pořízené z alternativního zdroje

34.4 Google Play Store

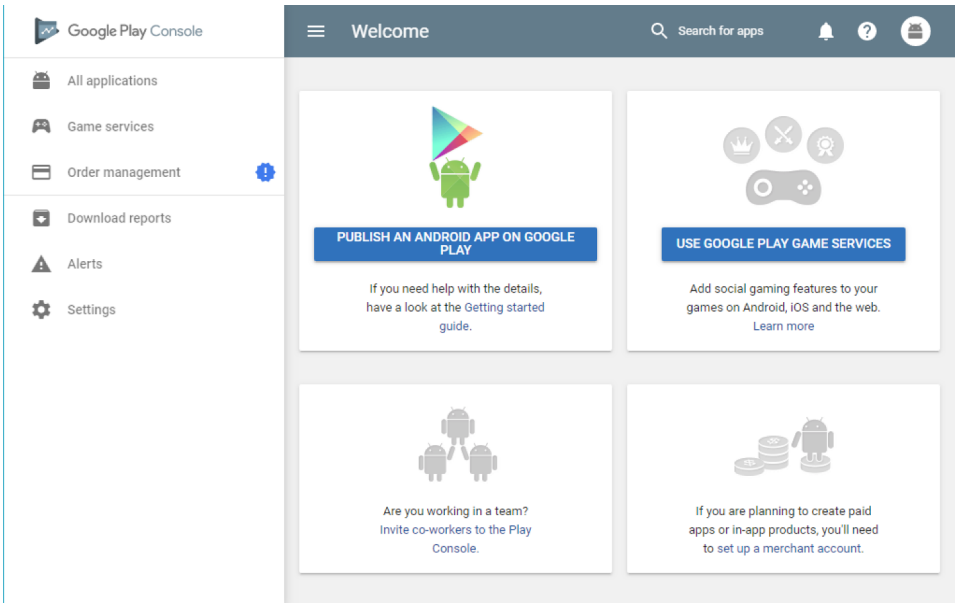
Umístění aplikace do Play Store je poměrně jednoduchou záležitostí. Podmínkou je registrace vývojářského účtu na Google Play, jeho propojení s obecným Google účtem a zaplacení registračního poplatku 25 USD. Poplatek je jednorázový a po jeho zaplacení je možné zveřejňovat vlastní aplikace bez dalších množstevních nebo časových omezení. Google se ale snaží zvyšovat kvalitu nabízených aplikací, proto podmínky vůči vývojářům často zpřísňuje a přizpůsobuje aktuálním potřebám. Je tedy vhodné se s momentálním stavem ještě před registrací seznámit.

Nejprve tedy registrace vývojářského účtu. Na stránce https://play.google.com/apps/publish/ viz obrázek 34.6, je potřeba nastavit vlastní Google registraci, zadat potřebné údaje a zaplatit poplatek.



Obrázek 34.6: Registrace na Google Play

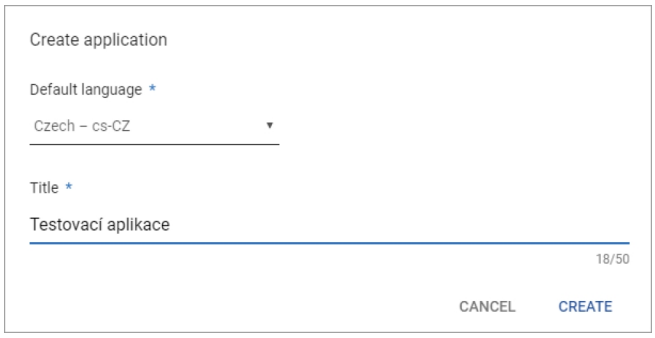
Registrací získá vývojář přístup do Google Play Console, ve které je podstatná zejména možnost „Publish an Android app on Google Play“, neboli zveřejnění aplikace na Google Play.



Obrázek 34.7: Google Play Console

V případě registrace aplikace, za kterou budou koncoví uživatelé platit, je nutné ještě zaregistrovat obchodní účet v Download Reports → Financial → Set up a merchant account now. Pro zveřejnění první aplikace se bez toho ale většina vývojářů zřejmě obejde.

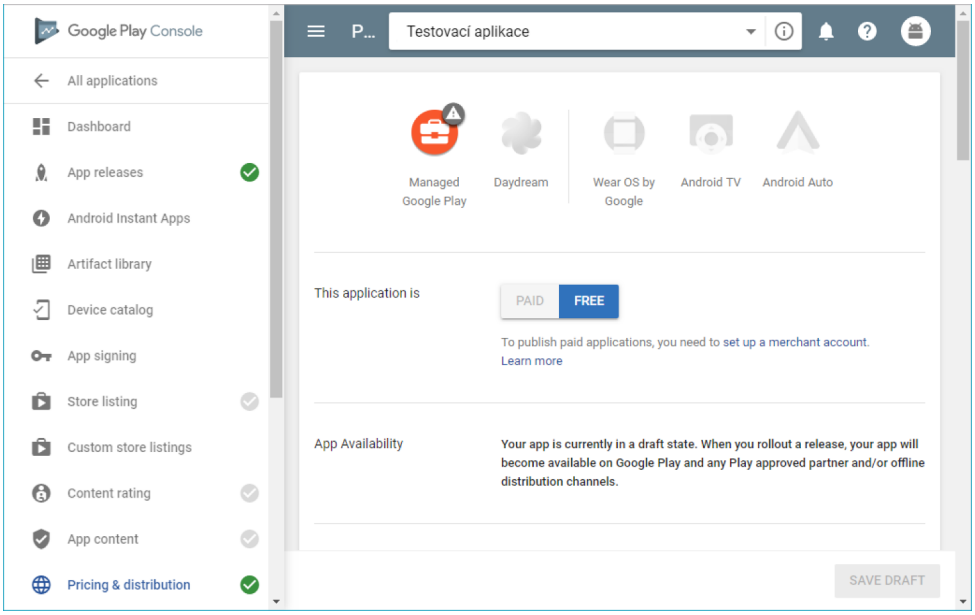
Po kliknutí na „Publish an Android app on Google Play“ nebo „Create Application“ (podle aktuálního zobrazení), můžeme nastavit jazyk a název aplikace. Název je důležitý, stane se z něj titulek v Google Play Store a je vhodné, aby se shodoval s fyzickým názvem aplikace.



Obrázek 34.8: Registrace aplikace na Google Play

Následuje vyplnění neméně důležitého stručného a komplexního popisu aplikace. Dál už je vše na autorovi – přidat lze náhledy aplikace, videosekvence, ikonu, kontaktní údaje, překlady atd. Prostě cokoliv z toho, co se dá běžně vidět na Play Store v průběhu hledání a instalace aplikací.

Tím je profil aplikace hotový a zbývá přidat aplikaci samotnou. Pro to je potřeba splnit řadu podmínek. Aplikace musí být sestavena v režimu release, musí být podepsaná, měla by mít formát spíše AAB (Android App Bundle) než APK (Android Package Kit) a vhodné číslo verze. Další podmínky se skrývají v menu Google Play Console. Pro zveřejnění aplikace je nutné mít všechny položky zeleně odškrtnuté. Pokud jsou odškrtnuté šedě, tak to znamená, že stále cosi chybí a bez doplnění nebude zveřejnění možné.



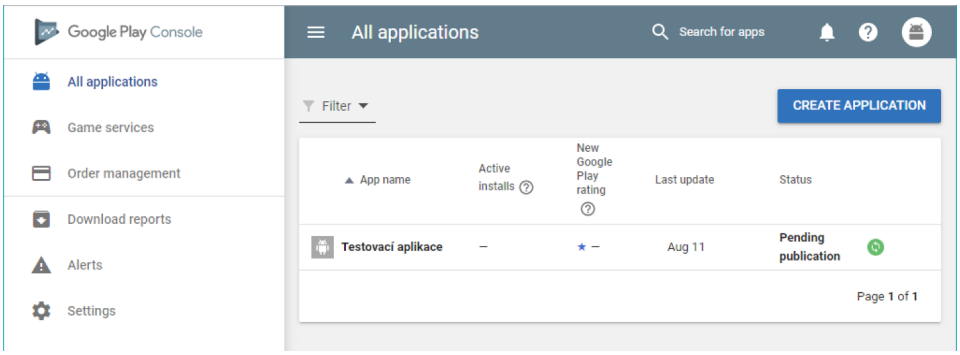
Obrázek 34.9: Příprava na zveřejnění aplikace

Dalším krokem před zveřejněním by mělo být testování. Zde je potřeba rozmyslet typ testů, kterým by se aplikace měla podrobit. Nabízí se interní testy, v jejichž rámci lze dát kolegům ve vývojovém týmu odkaz na její stažení. Vyšší úrovní je uzavřené testování neboli alfatesty, kdy aplikaci dostane k dispozici uzavřené množství testerů, které si vývojář sám vybere a identifikuje je e-mailovou adresou.

Po uzavřených testech přichází na řadu otevřené, neboli beta testování. Otevřené se mu říká proto, že autor může sestavit poměrně velkou skupinu testerů. Stačí, aby od autora dostali internetový odkaz.

Je jenom na autorovi, jestli se rozhodne využít možnosti testování aplikace nahrané do Google Play Console. V každém případě je to ale velmi silně doporučovaný postup.

Po úspěšném absolvování testů je na čase vydat první verzi aplikace. Postup je podobný jako u testování, pouze se namísto typu testu zvolí „Production“. Po předání do produkce se v seznamu aplikací objeví v poli Status příznak „Pending publication“, čekání na zveřejnění. Toto trvá u první vytvořené aplikace přibližně den, během té doby Google aplikaci zkontroluje a pokud je vše v pořádku, tak ji schválí. Ke zdržení až několika dní může dojít v případě, že je aplikace potenciálně podezřelá a nelze ji zpracovat automaticky (například pokud se má chovat jako prohlížeč cizích stránek, k čemuž stačí to, že obsahuje komponentu WebView, nebo se pokouší o nešifrovanou komunikaci prostřednictvím http protokolu). Update vyšších verzí už proběhne v řádu hodin. Poměrně účinným způsobem, jak proces urychlit je důsledné testování. Aplikace, které úspěšně prošly několika koly testování jsou zpracovány přednostně.



Obrázek 34.10: Čekání na schválení aplikace

Po schválení se aplikace objeví v Google Play Store a od toho okamžiku je úsilí vývojáře završeno. Aplikace je potenciálně k dispozici celému světu. Tedy asi třem miliardám aktivních uživatelů operačního systému Android.

35 Základní termíny, pro lepší orientaci v textu

SDK (Software Development Kit) - základní sada vývojářských nástrojů určených pro danou technologii, v našem případě se jí myslí zejména Android Studio.

API (Application Programming Interface) - rozhraní pro programování aplikací. Je to množina funkcí, procedur, knihoven, protokolů atd. které operační systém poskytuje vývojáři k využití. Například 'zavolání funkce operačního systému pro odeslání SMS' je terminologicky totéž jako 'volání API pro odeslání SMS'. API je postupně vylepšované, proto se v jednotlivých verzích Androidu liší. Aby byl v tom byl pořádek, má každá verze API svoje vlastní číslo verze (odlišné od čísla verze Androidu).

Java - jeden z nejpoužívanějších programovacích jazyků, původně vyvinutý firmou Sun Microsystems, dnes spravovaný společností Oracle. Zároveň je takto označovaná platforma pro spouštění programů. Nás zajímá zejména platforma Java ME (Mobile Edition), použitá v operačním systému Android.

Kotlin – moderní objektově orientovaný programovací jazyk, alternativní k Javě. Může používat Java knihovny, takže se v jednom projektu mohou sejít i oba programovací jazyky. Některé softwarové firmy, včetně Google, ho před Javou preferují.

Linuxové jádro (Linux kernel) - jádro je nejnižší vrstva operačního systému, která komunikuje přímo s hardware zařízení. Označení Linux znamená, že je použito jádro z operačního systému Linux, vyvíjeného pod licencí GNU (licence pro svobodný software), umožňující úpravy pro zařízení jednotlivých výrobců.

GUI (Graphical User Interface) - grafické uživatelské rozhraní aplikace. Jinými slovy to, co uživatel vidí na displeji nebo obrazovce a přes co zařízení ovládá - okna, ikony, menu atd.

XML (eXtensible Markup Language) - normalizovaný jazyk určený k serializaci dat a k rozšíření textu o formátovací informace. Jeho použití je poměrně široké. Lze pomocí něj zaznamenat data včetně struktury (výhodné například pro konfigurační soubory), popsat jím formulář, který je potom možné podle XML popisu graficky zobrazit, nebo lze pomocí XML uložit data a ta předat jiné aplikaci či systému.

GitHub – webová služba, která umožňuje verzování (udržování historie) a sdílení zdrojových kódů volně šiřitelných programátorských projektů. Zdrojové kódy jsou uloženy v repozitáři, neboli datovém úložišti. Adresa služby je www.GitHub.com

Průvodce (Wizzard) - je uživatelské rozhraní, které pomocí sekvence formulářů umožní uživateli provést základní nastavení. Setkáme se s ním například při zakládání nového projektu v Android Studiu.

Formulář (Layout) - graficky znázorněný návrh uživatelské obrazovky. Je to náhled na displej telefonu v režimu, ve kterém můžeme editovat jednotlivé grafické komponenty.

Emulátor - software který umožní spuštění aplikace na počítači tak, jako by byla spuštěna na cílovém zařízení. Například na telefonu nebo tabletu. Používá se k základnímu pohodlnému ladění a testování vytvářené aplikace.

Správce virtuálních zařízení / AVD Manager (Android Virtual Device Manager) - správce profilů pro emulátor Android zařízení. Umožňuje definovat parametry virtuálního zařízení, na kterém je následně možné ladit Android aplikace.

ASCII (American Standard Code for Information Interchange) - tabulka 128 základních znaků anglické abecedy a nejčastějších znaků používaných v informatice. Neobsahuje češtinu.

Bluetooth - technologie bezdrátové komunikace na krátkou vzdálenost. Lze ji využít k ladění aplikací pro hodinky a jinou nositelnou elektroniku. Většina Android zařízení tuto technologii podporuje, často ji má ale standardně neaktivní a je ji potřeba zapnout.

WiFi - technologie pro bezdrátové připojení zařízení k místní síti. Umožňuje připojení k internetu a lze ji použít také pro ladění aplikací, pokud nevyhovuje nebo není k dispozici propojení přes USB rozhraní.

USB (Universal Serial Bus) - sériová sběrnice, resp. způsob připojení periférie nebo jiného zařízení k počítači pomocí kabelu. Využijeme ji zejména pro ladění aplikací na fyzickém zařízení. Nezbytnou podmínkou je volný USB konektor jak na počítači tak na zařízení a pochopitelně propojovací kabel.

Třída (Class) - základní stavební prvek aplikace. Termín vychází z objektově orientovaného programování. Třída pomocí vlastností a metod popisuje objekt a jeho chování. Pro nás je to kus kódu, který obvykle odpovídá jednomu souboru Javy.

Metoda (Method) - v Javě je metoda součástí třídy, obsahuje kód který řídí chování objektu. V jiných jazycích je jejím ekvivalentem funkce.

Balíček (Package) - množina souborů potřebných pro běh aplikace. Z programátorského pohledu je to knihovna tříd organizovaná pomocí jmenné konvence (namespace). Název balíčku je uvedený vždy na začátku souboru. Ve struktuře projektu najdeme balíčky v adresáři *app → java*.

Jmenný prostor (Namespace) - způsob organizace tříd pomocí systému jmen. Každá třída spadá do nějakého namespace, čímž je zajištěno, že nenastane zmatek ve stejně pojmenovaných veřejných metodách a proměnných různých tříd.

Breakpoint - místo v kódu, kde dojde k přerušení vykonávání programu. Používá se při ladění – uživatel si nastaví breakpoint a aplikace na takto označeném místě zastaví. Do kódu je lze umísťovat podle potřeby, přímo v editoru Android Studia. Breakpointy mají efekt pouze v režimu ladění.

Zásobník volání (Call stack) - datová struktura do které se ukládají informace o zanořování při běhu programu. S každým zavoláním metody se do zásobníku uloží informace o vnoření a s každým opuštěním metody se tento záznam smaže. Takže když si zobrazíme zásobník při ladění, můžeme se podívat jak hluboko jsme zanořeni a jak byly jednotlivé metody vzájemně volány.

Instance třídy (Instance) – objekt zavedený do paměti, vytvořený z nějaké konkrétní třídy. Obsahuje metody které můžeme volat a atributy které můžeme číst nebo do nich zapisovat. V Javě se instance třídy vytváří pomocí klíčového slova ‚new'. Například NazevTridy instance = new NazevTridy(); V Kotlinu se new nepoužívá: val instance = NazevTridy()

Dědičnost (Inheritance) – je jedna ze základních vlastností objektově orientovaného programování. Slouží k vytvoření objektu na základě jiného již existujícího, který rozšíří a od kterého převezme všechny jeho metody a atributy. Odděděné třídy je možné dál dědit a vytvářet tak hierarchickou strukturu tříd. V Javě se dědění označuje klíčovým slovem `extends` následovaným názvem rozšiřované třídy.

Událost (Event) – je změna stavu. V systému Android se používá k asynchronnímu volání metod, jakmile dojde k nějaké definované změně stavu. Například pokud je aplikace spuštěna, systém vyvolá událost onCreate, což vede k zavolání metody onCreate() v hlavní aktivitě aplikace.

Stromová struktura – hierarchické uspořádání objektů, například složek a souborů operačního systému. Nejvyšším bodem je takzvaný kořen stromu (na počítači například disk C), který obsahuje další objekty – složky a soubory. Složky umožňují jeho další větvení.

Unit testing – nástroje pro automatické ověření funkčnosti aplikace. Vytváří je zpravidla sám tvůrce aplikace a jejich cílem je snadno a rychle ověřit, jestli během vývoje nebo následných úprav nedošlo k poškození

základní funkčnosti. Android Studio při zakládání projektu vytváří třídu pro Unit testy v adresáři app → java.

dp nebo **dip** (Density-independent Pixels) – abstraktní jednotka udávající šířku objektů na displeji bez ohledu na jemnost displeje. Odpovídá velikosti jednoho obrazového bodu (pixelu) na displeji s hustotou 160 bodů na palec. Díky tomu můžeme v Android Studiu snadno nastavit velikost objektů, u kterých potřebujeme konstantní rozměr bez ohledu na použité zařízení. Objekt široký 10dp má na displeji s hustotou 160 bodů na palec rozměr 10 pixelů. Na displeji s hustotou 320 bodů na palec je tvořen dvaceti pixely. Na obou displejích se zobrazí se stejnou šířkou.

sp (Scale-independent Pixels) – abstraktní jednotka udávající velikost textu na displeji, bez ohledu na to, jakou má uživatel nastavenou velikost písma. Tuto jednotku se doporučuje používat při nastavování velikosti písma všude tam, kde se nějaký text zobrazí na displeji.

Fragment – komponenta uživatelského rozhraní. Fragmenty jsou bloky obsahující prvky uživatelského rozhraní, které je možné umístit do formuláře podle velkosti displeje. Na malém displeji je umístěno méně fragmentů než na velkém. Například na displeji telefonu je zobrazený seznam kontaktů a po kliknutí se zobrazí jiná aktivita s jejich detailem. Na tabletu se stejný seznam zobrazí v levé části a na téže aktivitě je vedle seznamu rovnou k dispozici i detail vybraného kontaktu, bez nutnosti zobrazit jinou stránku (aktivitu).

Adaptér – komponenta, propojující dvě rozhraní, která by sama o sobě nedokázala vzájemně komunikovat. Používá se například k propojení seznamu RecyclerView se zdrojem dat.

Posluchač událostí (Listener) – objekt, který sleduje jestli nedošlo k vyvolání události. V Androidu je to například obslužná rutina, která čeká na kliknutí na tlačítko.

Předefinovat metodu (Method overriding) – nahradit virtuální metodu z nadřazené třídy metodou vlastní. Ta musí mít stejné jméno, parametry a shodný návratový typ. Pro označení předefinovávané metody se používá specifikátor @Override.

Konstruktor (Constructor) – metoda třídy bez návratové hodnoty, která je volaná při vzniku instance této třídy. Obvykle obsahuje iniciační příkazy, které se mají provést v okamžiku jejího vytvoření.

HTML – HyperText Markup Language, skriptovací jazyk, určený k vytváření webových stránek. Základní vlastností tohoto jazyka jsou hypertextové odkazy, které umožňují propojení stránek vzájemně mezi sebou.

Vyrovnávací paměť (Buffer) – paměť určená k dočasnému uchování informace předtím, než bude zpracována. Využívá se, když rychlost načítání a zpracování dat není stejná. Typickým příkladem je zápis na disk, který je příliš pomalý, proto aplikace zapisovaná data uloží do vyrovnávací paměti a zatímco provádí další činnost, data se na disk postupně zapíší.

Databáze – systém pro ukládání dat s možností vyhledávání, řazení a vyhodnocování. Android má integrovanou databázi SQLite, která je přístupná pomocí SQL dotazů.

SQL (Structured Query Language) – strukturovaný dotazovací jazyk, používaný pro přístup k datům uloženým v SQL databázích.

SQL dotaz (SQL Query) – příkaz nebo sekvence příkazů, které jsou odeslány databázi k jednorázovému provedení. Dotazy jsou určené buď k administraci databáze nebo k manipulaci s daty. Formátu dotazů se říká SQL syntaxe. Například dotaz ‚SELECT * FROM VerzeAndroidu ORDER BY Verze‘ říká, že má databáze najít všechny záznamy z tabulky VerzeAndroidu, seřadit je podle sloupce Verze a vrátit hodnoty ze všech sloupců, které tabulka obsahuje.

Kontext (Context) – část aplikace, ke které se vztahuje objekt. Například tlačítko má zpravidla jako kontext aktivitu. Jakmile k tlačítku potřebujeme přistoupit, musíme nejdřív získat přístup k aktivitě. Pokud aktivita zanikne, zanikne i tlačítko.

GPS, Globální polohový systém (Global Positioning System) – družicový systém, pomocí nějž je možné určit polohu GPS přijímače kdekoliv na světě. Systém je provozovaný americkým Ministerstvem obrany a je k dispozici i pro civilní použití. Dnes existuje podobných systémů, ať už regionálních nebo globálních, několik (Galileo, Glonass, Beidou, QZSS, IRNSS), ale zkratka GPS se natolik vžila a zobecnila, že se často používá souhrnně pro všechny polohové systémy.

GNSS, Globální družicový polohový systém (Global Navigation Satellite System) – služba, která umožňuje zjistit geografickou polohu přijímače kdekoliv na světě. Zahrnuje jednotlivé globální systémy, GPS, Glonass, Galileo atd.

Logování – postup při kterém aplikace zapisuje krátké zprávy o průběhu činnosti, neboli logy. Používají se zejména pro dohledávání a analýzu chyb.

Stack trace – strukturovaný výpis chyby, který umožňuje dohledat příčinu výjimky a způsob, jak byly vnořené metody postupně volány. Používá se při ladění aplikace.

Požadavek - odpověď (request - response) – způsob komunikace mezi dvěma objekty (v našem případě zpravidla Android zařízení a internetový server), realizovaný pomocí zasílání zpráv a odpovědí na ně.

REST (Representational State Transfer) – jednoduchý způsob předávání požadavků a dat mezi servery, pomocí HTTP volání. Základní operace, neboli metody, jsou GET, POST, PUT a DELETE.

GET metoda (GET method) – způsob předání dat při HTTP komunikaci se serverem. Data jsou součástí URL, mají formu parametru. Například *http://domena.cz?p=test* – je jednoduchý URL řetězec s parametrem p a hodnotou test.

POST metoda (POST method) – způsob předání dat při HTTP komunikaci se serverem. Data jsou uložena uvnitř odesílané zprávy.

JSON (JavaScript Object Notation) – způsob strukturovaného zápisu dat, vhodný k vzdálenému přenosu mezi komunikujícími zařízeními (v našem případě zpravidla Android zařízení a internetový server). Zpráva je platformně nezávislá, podobně jako XML má podobu čitelného textu, ale na rozdíl od XML je z hlediska objemu dat výrazně úspornější.

Yahoo! – internetový portál poskytující velké množství služeb, jako zpravodajství, vyhledávač freemail a podobně. Mimo jiné poskytuje také informace o počasí, k nimž zpřístupnila API, takže je můžou vývojáři využívat ve svých aplikacích.

Parsování, dekódování (parsing) – transformace textové informace na datovou strukturu. Jde o operaci, kdy textovou zprávu, například „1;Oreo;8“, převedeme na data podle předem daného pravidla. Pokud víme, že první hodnota je id, druhá název, třetí číslo verze a oddělovač je středník, tak můžeme uvedený text, pomocí parseru, rozparsovat do proměnných id=1; nazev=“Oreo“; verze=”8”;

Parser – programová rutina, která dokáže rozparsovat (viz termín Parsování) textovou informaci na datovou strukturu.

Služba (Service) – aplikace nebo komponenta aplikace, která běží na pozadí, nemá uživatelské rozhraní a je spuštěna i když uživatel pracuje s jinou aplikací. Například přehrávač hudby.

Zpětné volání (callback) – způsob, jak zajistit spuštění programové rutiny poté co je vykonána jiná operace, aniž by bylo nutné její dokončení aktivně ověřovat. V Androidu například aktivujeme čtečku otisku prstu, pomocí API předáme adresu rutiny která se má po načtení otisku spustit a dále se obsluze čtečky nevěnujeme. Jakmile je otisk prstu načtený, Android sám zavolá metodu na adrese kterou jsme mu předali a tato metoda například odemkne obrazovku.

Sandbox – způsob zabezpečení, který umožňuje aplikaci přístup pouze k prostředkům vlastního procesu. Ostatní procesy jsou oddělené, pro aplikaci fakticky neviditelné. Totéž se týká některých služeb systému, ke kterým aplikace z bezpečnostních důvodů nemá přístup. Je-li nutné, aby aplikace měla možnost používat prostředky mimo prostředí sandboxu, musí o ně požádat v souboru manifestu.

Widget – malá aplikace, kterou lze umístit trvale na displej zařízení. Zobrazuje uživateli důležité informace a někdy umožňuje minimalistickou interakci. Které widgety budou na displeji zobrazeny, si volí uživatel sám. Příkladem jsou hodiny, předpověď počasí, přehrávač hudby a tak podobně.

Gradle – standardní součást Android Studia, nástroj který se stará o sestavení aplikace.

Groovy – skriptovací jazyk, ve kterém jsou napsané skripty sestavovacího nástroje Gradle.

Vlákno (thread) – součást vícevláknového procesu, kdy rutina běží paralelně s ostatními rutinami (vlákny). Dochází tak k současnému zpracování více operací.

Časovač (timer) – implementace automatického spouštění programových rutin v definovaných okamžicích, zpravidla periodicky se opakujících.

Intent – prostředek pro komunikaci mezi komponentami aplikace. Explicitní intent – určuje jakým způsobem má Android operaci provést (například spustit konkrétní aktivitu), implicitní intent – pouze předá Androidu potřebné informace a ten sám určí jak operaci vykonat (například spuštěním prohlížeče). Android často, při použití implicitního intentu, nechává na uživateli aby si vybral z několika možných alternativ (například z několika typů prohlížečů).

Úložiště klíčů (keystore) – je to binární soubor uložený na disku počítače, který obsahuje soukromé (případně i veřejné) klíče. Soubor je vhodné udržovat na bezpečném místě. Úložiště je chráněno heslem.

Soukromý klíč (private key) – v kryptografii se takto označuje klíč, který má k dispozici pouze jeho držitel. Využívá se k dešifrování nebo podepisování. V případě vývoje Android aplikací se tyto klíče používají zejména pro podepisování aplikací před jejich nahráním do zařízení.

APK balíček (Android Package Kit) – komprimovaný soubor, který obsahuje všechna potřebná data pro instalaci aplikace na Android zařízení. APK balíček se používá pokaždé, kdy potřebujeme vyvíjenou aplikaci na zařízení spustit, nebo když ji potřebujeme předat uživatelům k instalaci – ať už přímo, nebo prostřednictvím Google Play Store.

AAB balíček (Android App Bundles) – komplexnější verze APK balíčku. Používá se pro nahrání vyvíjené aplikace do Google Play Store, kde se převede na několik verzí instalační aplikace. Každý z takto vytvořených instalátorů je přizpůsobený hardwarově jinému zařízení a jiné lokalizaci. Uživatel si k instalaci nahraje automaticky tu verzi, která odpovídá jeho potřebám. Výhodou oproti APK balíčku je zejména menší velikost instalované aplikace.

Unit testy (Unit testing) – způsob automatického testování již existujícího kódu, pomocí naprogramovaných postupů, s předepsanými vstupy a s porovnáním očekávaných výstupů. Cílem je ověřit funkčnost dříve napsaných bloků kódu.

Bundle – objekt, který se používá pro předávání dat mezi aktivitami. Hodnoty jsou do něj uloženy pod textovými názvy, objekt je předán cílové aktivitě a v ní jsou hodnoty opět pod stejnými názvy zase vyzvednuty.

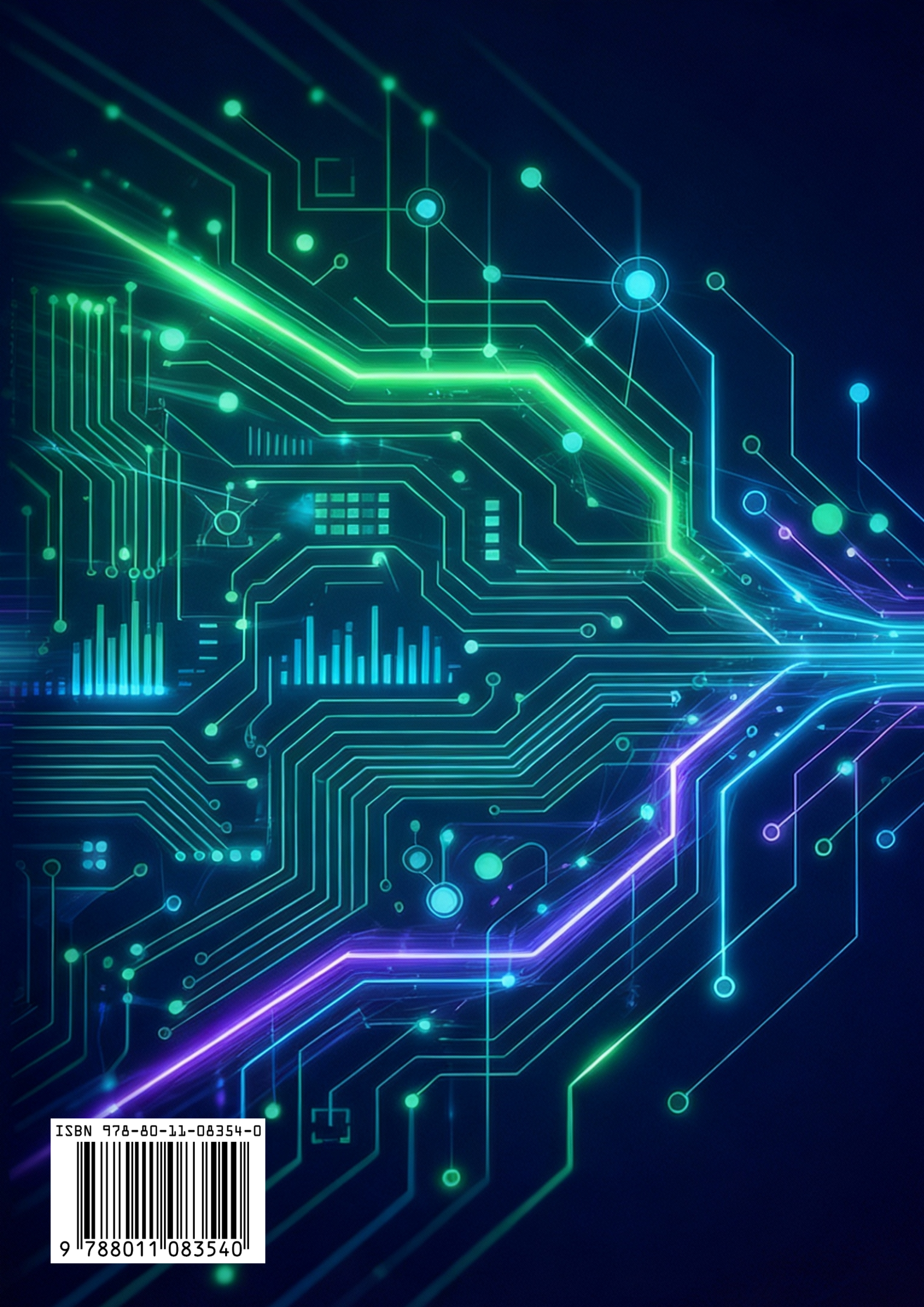
Kurzor (Cursor) – v jazyce SQL je kurzorem množina záznamů, které jsou výsledkem dotazu a které je možné postupně procházet. Má tři základní příkazy – otevření kurzoru, přečtení následujícího záznamu a zavření kurzoru. Obvykle obsahují smyčku, ve které se postupně zpracovávají všechny, dotazem získané záznamy.

Vývojářské menu (Developer options) – skrytá položka v Nastavení, kterou je možné uživatelsky aktivovat. Pod tímto menu jsou k dispozici podrobnější technické informace o zařízení a systému.

Material design – definice vzhledu a vizuálního chování grafického rozhraní. V Androidu je to soubor systémových knihoven, které ovlivňují to, jak jednotlivé komponenty aplikace vypadají a jak se chovají. Například vzhled tlačítka je na starším Androidu jiný než na novějších verzích. Základními verzemi jsou Material design a Material design 3.

GUID (Globally Unique Identifier) – 128 bitový identifikátor zapisovaný hexadecimálními číslicemi, který se generuje algoritmicky tak, aby byl jedinečný bez ohledu na systém, který ho

vytvořil. Jedinečnost není zaručena stoprocentně, ale pravděpodobnost vzniku dvou stejných GUID je tak nízká, že se v praxi může považovat za nulovou. Příklad GUID: „42ba71fc-26a2-4014-b364-5b369ad2f252“.



ISBN 978-80-11-08354-0

9 788011 083540