UNIVERZITA J. E. PURKYNĚ V ÚSTÍ NAD LABEM
Přírodovědecká fakulta

Programování pro mobilní platformy

KI/PMP

Jiří Fišer

Ústí nad Labem 2020

Kurz: Programování pro mobilní platformy

Obor: Aplikovaná informatika

Klíčová slova: programování, Android

Anotace: Kurs je zaměřen na praktické programování pro mobilní platformy (mobilní telefony, tablety), přičemž pozornost je věnována typickým rysům těchto platforem – prodloužený životní cyklus aplikace, sandboxing, dynamičtější GUI a integrace se specifickými hardwarovými a softwarovými službami. Konkrétní platforma bude volena podle aktuálních požadavků (uplatnění na pracovním trhu, dostupnost hardwaru). V rámci kursu budou vytvářeny aplikace středního rozsahu ukazující klíčové aspekty zvolené platformy.

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

© Katedra informatiky, PřF, UJEP v Ústí nad Labem, 2016

Autor: Jiří Fišer

Obsah

Úvodní slovo

Výstupní znalosti

Absolvent kursu je připraven navrhovat a implementovat středně složité aplikace pro Android s využitím některých hardwarových a softwarových služeb tabletů a mobilních telefonů.

Seminární úkol

Seminárním úkolem je vytvoření aplikace pro mobilní platformu Android.

Aplikace musí splňovat tyto minimální požadavky:

- alespoň dvě aktivity
- služba na pozadí
- použití alespoň jednoho z klíčových témat: persistentní úložiště, přístup k webové službě, geolokace, využití sensorů (kombinace je samozřejmě také možná)

Výstupem seminárního úkolu je komentovaný zdrojový kód a UML třídní diagram uživatelských tříd.

Příklady seminárních úkolů z minulých let:

- jednoduchý simulátor pražského orloje
- rozhraní ke webové službě univerzitního systému STAG
- activity logger (využití senzoru zrychlení)

Klíčové pojmy

klíčový pojem

Pojmy uvedené na levém okraji textu (a v textu zvýrazněné tučně) jsou tzv. **klíčové pojmy**. Jejich správné a plné pochopení je nezbytné pro další studium. Je samozřejmou součástí zkoušek (a to i zkoušek navazujících předmětů resp. státní závěrečné) a jejich znalost se předpokládá i v rámci obhajoby seminární práce.

Většinu z nich můžete najít v doporučené literatuře a jsou popsány i v anglické *Wikipedii* (anglický překlad je uveden, s výjimkou termínů, kde je zřejmý). V oblasti informatiky jsou články anglické Wikipedii (ve většině případů) velmi kvalitní, s rozsahem přesahujícím popis uvedený v opoře a tak mohou být využity k dalšímu zpřesnění prohloubení znalostí. Navíc obsahují odkazy na další hodnotné zdroje.

1 Android z pohledu programátora

CÍLE KAPITOLY

Kapitola popisuje (ve stručnosti) různé možnosti vytváření aplikací na platformě Android (některé alternativní přístupy byste mohli i vyzkoušet). Hlavním cílem je nicméně stručný úvod do procesu vytváření nejběžnějších nativních aplikací — využití Javy a standardního API Androidu (SDK).

1.1 Jak lze v Androidu programovat

Android je moderní a komplexní operační systém, který nabízí několik možností tvorby aplikací prostřednictvím různých programovacích jazyků a platforem, a to na různých úrovních abstrakce

- aplikace pro webové prohlížeče (HTML 5 skriptování na straně klienta). Tyto aplikace jsou přenositelné i na jiné mobilní platformy či dokonce na platformy desktopové. Díky specializovaným knihovnám, jako je například *PhoneGap* mohou mít tyto aplikace přístup ke specializovanému hardwaru (kamera, akcelerometr, GPS) a mohou být lépe integrovány do infrastruktury operačního systému (notifikace, sítový přístup, kontakty, souborový přístup). Klíčová je i podpora limitovaného GUI s podporou dotykového vstupu (např. *jQuery Mobile*). Řešením je pouze tzv. rooting Androidu (tj. obejití bezpečnostního mechanismu, tím že je dovolen superuživatelský přístup) a doi
- použití obecných vysokoúrovňových skriptovacích jazyků (Python, Ruby, apod.). Tyto jazyky nabízejí své rozsáhlé univerzální knihovny, aby však byly využitelné pro tvorbu plnohodnotných aplikací musí opět nabízet alespoň částečnou integraci do infrastruktury OS (včetně podpory specializovaného hardwaru). Výhodou je i (pokud možno přenositelná) GUI knihovna s podporou moderního přístupu ke tvorbě GUI aplikací. Určité zkušenosti mám především s portem *Pythonu* (*QPython*) a knihovnou *Kivy*.
- programování nativních aplikací s využitím Javy a standardního API Androidu (Android SDK). Valná většina aplikací pro Android je naprogramována tímto způsobem. Javovský kód pro Android využívá relativně vysokou úroveň abstrakce a podporuje plnou integraci do infrastruktury Androidu, včetně spolupráce s ostatními aplikacemi. Je také jako jediný plně podporován firmou Google, tvůrcem a správcem Androidu. To mimo jiné znamená, že zajišťuje vysokou míru kompatibility s různými hardwarovými platformami. Na druhou stranu je Java relativně *rozvláčný* jazyk a v Androidu je tento rys ještě výraznější.
- na stejné úrovni abstrakce jsou i některé přenositelné platformy třetích stran, které přímo využívají Android API. Příkladem je knihovna *Xamarin* (využívající jazyk C# a překladač *Mono*) a *Qt*.
- pro vytváření kódu, který vyžaduje přímý přístup k hardwaru resp. efektivnější využití paměti lze využít *Android NDK* (jazyky C resp. C++). Tímto způsobem jsou však implementovány jen části aplikací (např. fyzikální výpočty, apod.). Přístup ke GUI je na této úrovni výrazně omezen.
- terminálově orientované unixovské aplikace (CLI) nelze v Androidu nativně používat (přestože Android využívá jádro Linuxu). Android však využívá zcela jiný model běhu aplikací (především bezpečnostní) a neposkytuje implicitně textový shell a četné standardní knihovny jazyka C. Řešením je pouze tzv. rooting Androidu (tj. obejití bezpečnostního mechanismu, tím že je dovolen

superuživatelský přístup) a doinstalování potřebného softwaru (což je pro provozování jednoduchých GUI aplikací pověstný kanón na vrabce). Klasické textové aplikace lze portovat za pomoci NDK a emulátoru terminálu (což není bohužel triviální)

Tento výukový materiál se zaměřuje jen na využití standardního SDK za použití jazyka *Java*. To přirozeně znamená, že ostatní přístupy (především využití HTML5 a skriptovacích jazyků) jsou horší. Ve skutečnosti mají mnohé výhody, avšak rozsah tohoto materiálu neumožňuje popsat všechny alternativy.

1.2 Java a Android

Nejdůležitějším programovacím jazykem na Androidu je *Java*. Na první pohled se může zdát, že se jedná o klasickou Javu známou z dalších platforem. Na úrovni syntaxe tomu tak opravdu je (pro Android lze využít jakoukoliv modernější verzi standardní Javy od verze 7 včetně).

Jinak je tomu na úrovni knihoven. Android využívá nejen své vlastní knihovny, ale i zcela jiný programovací model. Zcela rozdílná je například realizace GUI vrstvy (Android nepoužívá AWT, SWING nebo JavaFX).

Jen relativně malá číst knihoven z platformy Java SE je dostupná i na Androidu (tím spíše knihovny jiných javovských platforem jako je Java ME). Naštěstí do této omezené podmnožiny spadají často používané třídy kolekcí a proudů.

ART

Ještě hlubší rozdíly existují ve fázi vykonávání bytového kódu, který se získá překladem javovského kódu. Android používá vlastní běhové prostředí označované jako **ART**, který využívá zcela odlišný bytový kód (tento kód je společný s původním virtuální strojem s JIT kompilací, jenž byl označován jako *Dalvik*). Tento bytový kód je registrově orientovaný (standardní bytový kód JVM využívá primárně zásobník podobně jako je tomu na platformě .NET). Dalvik bytový kód by měl být optimalizován pro limitovaná zařízení (je například o něco kompaktnější), ale existuje k němu jen minimum dokumentace (i ve vyhledávači Google je původní stroj *Dalvik* málem předběhnut islandskou obcí Dalvík s 1454 obyvateli). ART nepoužívá strategii JIT (just-in-time kompilace), ale AOT (*ahead-of-time compilation*) tj. bytový kód se do strojového přeloží již při instalaci (výsledkem je běžný unixový spustitelný soubor).

Tento rozdíl se však navenek příliš neprojevuje. Zesložiťuje procesní řetězec, neboť vkládá další krok do procesu překladu javovského kódu. Javovský kód je nejdříve přeložen do JVM bytového kódu (přípona *.class*) a pak do kódu Dalviku (přípona *.dex*). To se však navenek příliš neprojevuje, neboť překlad v *Android Studiu* (resp. jiném vyspělém IDE) je prováděn automaticky na pozadí při spuštění emulátoru a zajistí všechny nezbytné kroky (kromě překladu *Java* $\longrightarrow$ *JVM* $\longrightarrow$ *Dalvik*, je to i zabalení do instalačního balíku s příponou APK) .

OTÁZKY

1. Jaké části knihoven jsou společné pro Android a standardní Javu (Java SE)?
2. Jaký je rozdíl mezi JIT a AOT kompilací?

OTÁZKY K ZAMYŠLENÍ

1. Jaké jsou výhody a nevýhody použití bytového kódu a virtuálního stroje na mobilní platformě?
2. Jakou přidanou hodnotu nabízí specializované HTML mobilní frameworky oproti běžným knihovnám (např. jQuery Mobile oproti jQuery)?

2 Vývojové prostředí Android Studio

CÍLE KAPITOLY

Tato kapitola popisuje stručně postup instalace vývojového prostředí *Android Studio*. Toto prostředí je k dispozici volně pro všechny hlavní desktopové platformy včetně Linuxu.

Instalace je snadná a tak se popis zaměřuje jen na klíčové konfigurační volby. Popsán je stav ve verzi 1.5. V novějších verzích se může nastavení lišit (i když základní principy jako je volba API zůstanou s vysokou pravděpodobností zachovány)

Hlavním Vaším cílem je úspěšná instalace a vytvoření testovací aplikace, včetně jejího ověření v emulátoru či reálné zařízení. Pokud při tomto procesu k chybě, zkuste nejdříve najít informaci na Internetu (problémy jsou často omezeny jen na menšinu systémů a vyučující s nimi nemusí mít žádnou zkušenost).

2.1 Instalace

Aplikace pro Android lze vytvářet v libovolném vývojovém prostředí poskytujícím, alespoň minimální podporu pro jazyk Java a spouštění externích nástrojů. Tvorba aplikací v Androidu vyžaduje minimálně tyto nástroje:

1. *JAVA SE SDK* (typicky od Oraclu), minimální verze Java 7, základní knihovny a překladače
2. *Android SDK* (od Googlu) — obsahující knihovny, překladače a emulátory
3. sestavovací nástroj (zajistí správné provedení celého kompilačního řetězce) — minimálně make, ale vhodnější je vyspělejší nástroj s přímou podporou Android nástrojů jako např. Gradle

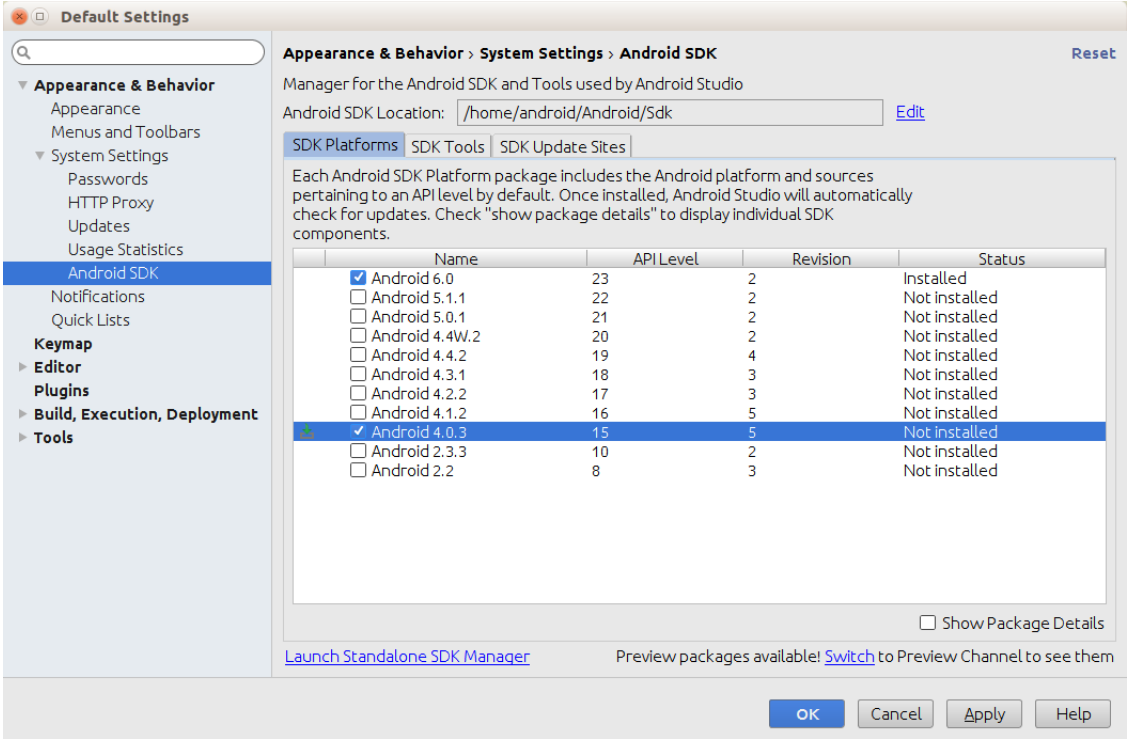
Celý vývojový systém je dnes již dost komplexní a jeho udržování není snadné: Proto se doporučuje využít IDE s přímou podporou vývoje pro Android. My budeme využívat IDE Android Studio, což je původní editor IntelliJ IDEA upravený a přizpůsobený pro Android projekty. Toto vývojové prostředí vytvořila a spravuje firma Google. Je relativně nový (prosinec 2014) avšak již od svých počátků byl navrhován jako základní vývojový nástroj pro Android (nahradil tak původní plugin pro *Eclipse*).

Instalace je v zásadě jednoduchá a provádí se v těchto krocích:

1. stažení Java SE JDK (minimálně verze 1.7, nestačí JRE), nejlépe přímo od firmy Oracle
2. kontrola zde je v PATH odkaz na překladač *javac*
3. stažení *Android Studio*
4. rozbalení (jedná se o ZIP, TGZ, instalační EXE)
5. první spuštění (v rámci, něhož se stahuje i Android SDK)

Upozornění: Celkově se stahuje více než 1GiB dat a to i v minimální konfiguraci (reálně spíše ke 1,5 GiB). Také ostatní hardwarové nároky nejsou malé. Procesor spíše třídy Core i-5 a minimálně 4GiB paměti (pro běh emulátoru je lepší 8GiB).

Při prvním spuštění stačí zvolit standardní instalaci, která proběhne téměř bez interakce. Výjimkou je volba verze SDK. Předvoleno je nejvyšší aktuální SDK (to je určeno verzí Androidu, na obrázku 6.0 a číslem tzv. API na obrázku je to 23). Moderní SDK podporují i vytváření aplikací pro nižší verze (menší API). Pro ověření běhu v nativním prostředí (především v emulátoru) *je možné* přidat i nižší API (na obrázku je to API 15, což je API mého mobilního telefonu).



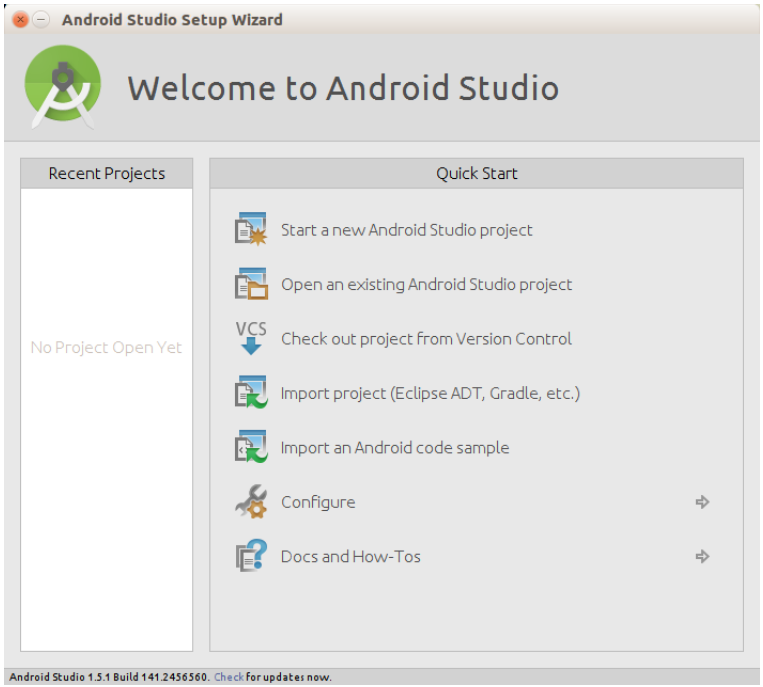
SDK lze samozřejmě přidávat resp. aktualizovat i poté (ve *File* | *Settings*).

Po dokončení instalace se zobrazí hlavní menu průvodců a my jsme připraveni vytvořit první aplikaci pro Android.

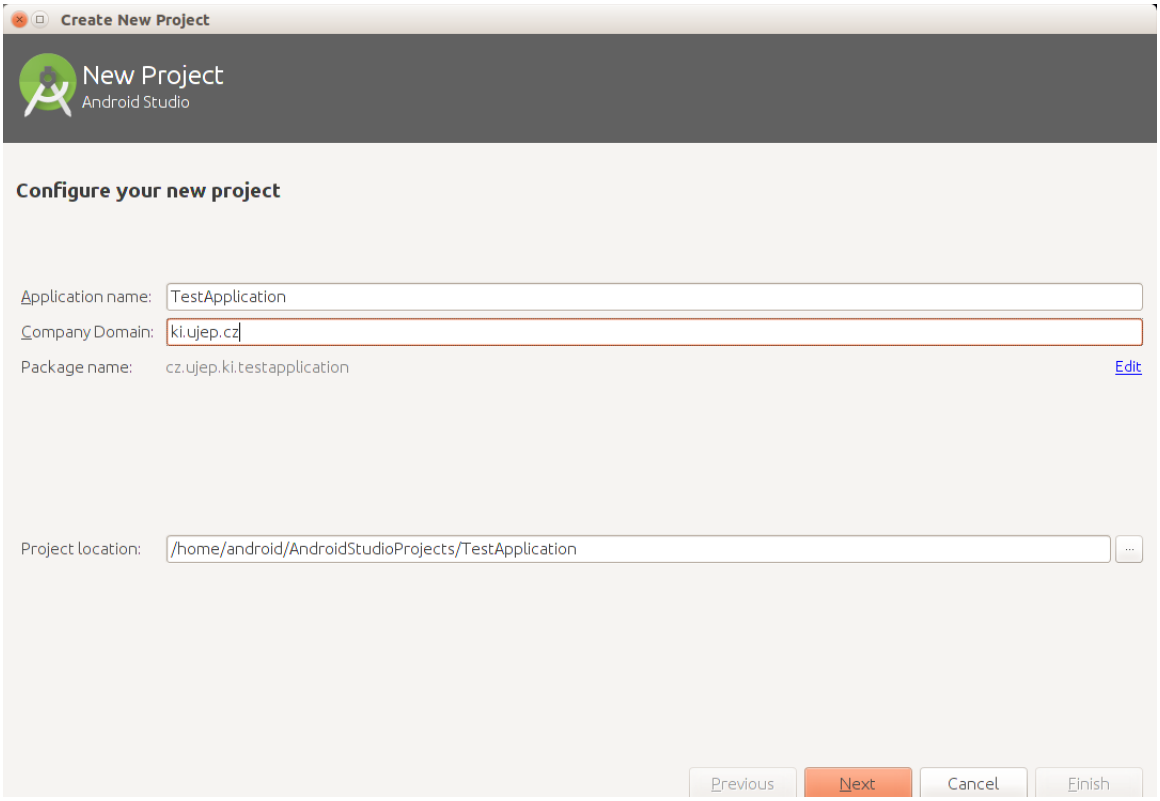
2.2 Vytvoření nové aplikace (projektu)

Pro ověření funkčnosti aplikace (a stažení obrazů jádra, pokud používáme emulátor) je dobré vytvořit a spustit testovací aplikaci.

V hlavním menu průvodců zvolte *Start a new Android Studio project.*



Nastavení není složité. Většina nastavení se provede v prvním ze čtyř formulářů.

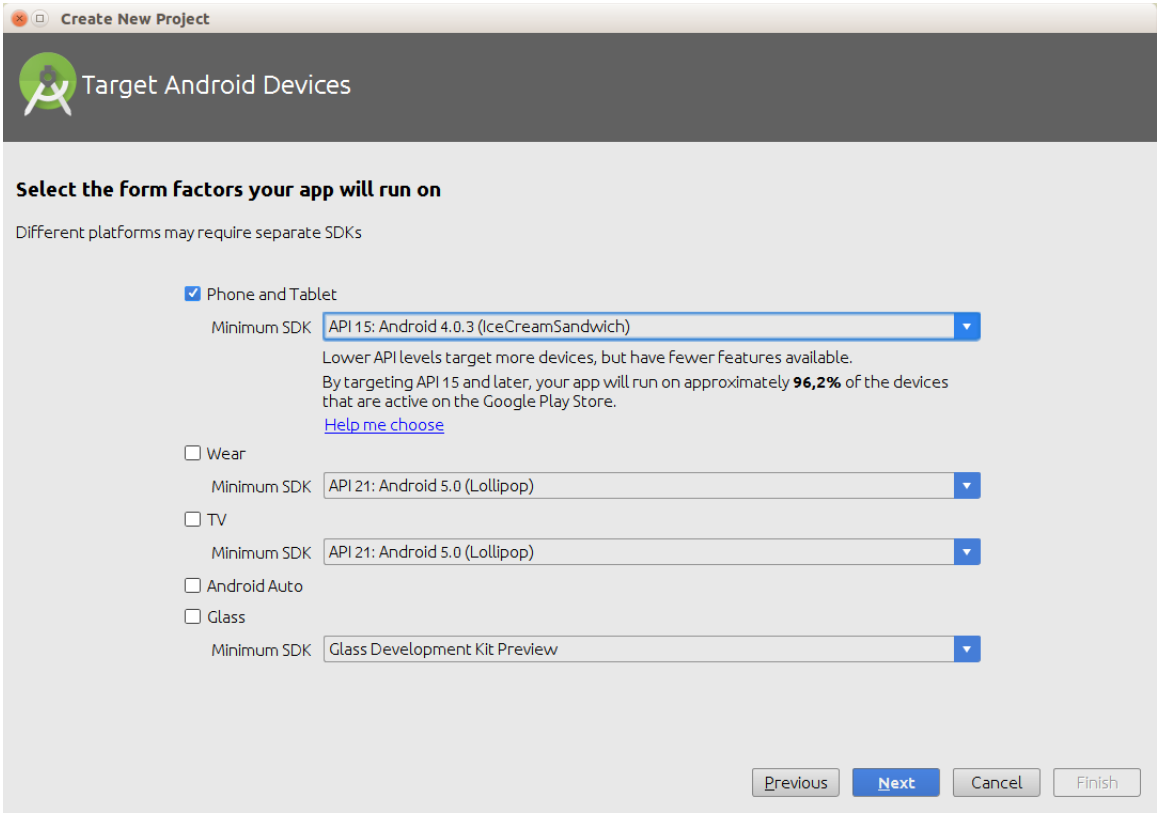


Aplication name je jméno aplikace, tak jak bude vidět koncový uživatel. Jeho pozdější změna je velmi obtížná, tj. je nutné volit uvážlivě.

Company domain je doménová část adresy tvůrce, sloužící (v opačném gardu) jako prefix balíku (jmenného prostoru). Můžete uvést jakoukoliv doménu, u níž můžete zajistit jedinečnost jmen všech balíků (tj. danou doménu spravujete).

U *project location* zkontrolujte, zda cesta ukazuje rozumné umístění (základem je tzv. workspace adresář). Umístění lze samozřejmě změnit.

Druhý formulář průvodce slouží k nastavení typu API a minimálního SDK.

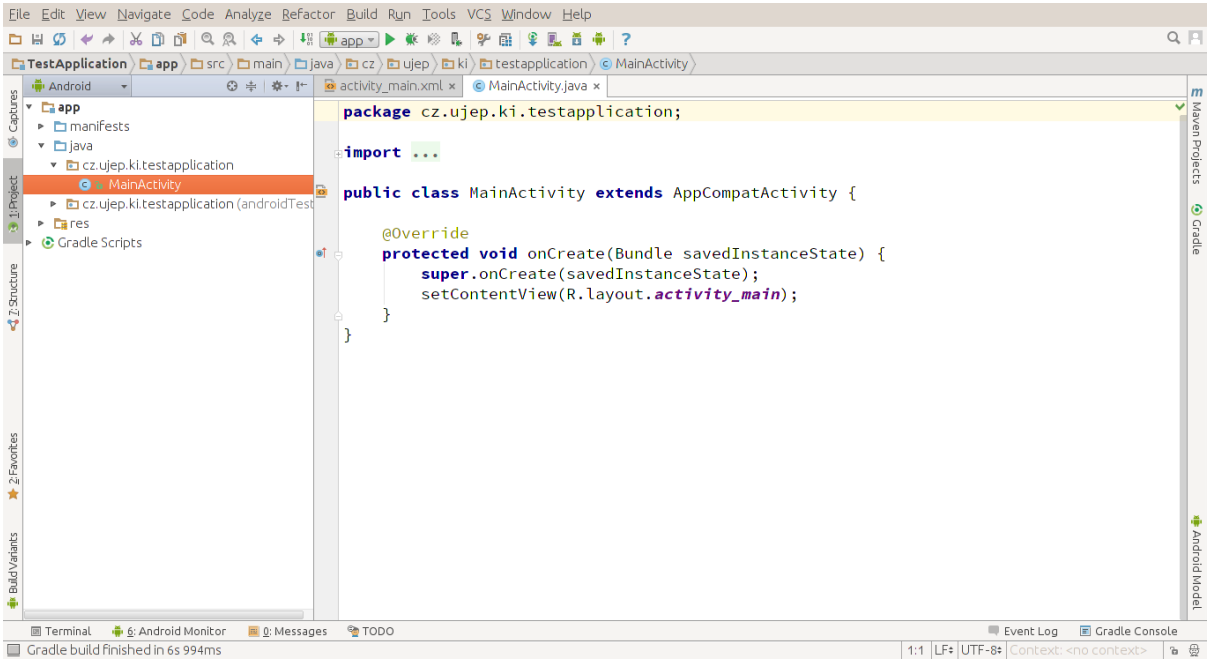


V rámci této opory budeme používat pouze klasické Android API pro telefony a tablety. Volba minimálního SDK ovlivňuje podíl Android zařízení s nimiž bude aplikace kompatibilní. Čím nižší API tím větší počet cílových zařízení, ale také méně použitelných GUI prvků (i když mnohé nové prvky jsou v současnosti dostupné i na starších API díky knihovnám pro zpětnou kompatibilitu).

V rámci kurzu budeme používat minimální API 15 (tj. verze 4.0.3 a vyšší).

V dalším formuláři zvolíme vzhled tzv. hlavní aktivity (tj. vzhled centrálního okna aplikace). Zvolte „Empty activity“ (zcela prázdná aktivita). V dalším okně můžete třídu aktivity přejmenovat, ale to je prozatím zbytečné, tj. stačí průvodce ukončit tlačítkem „*Finish*“.

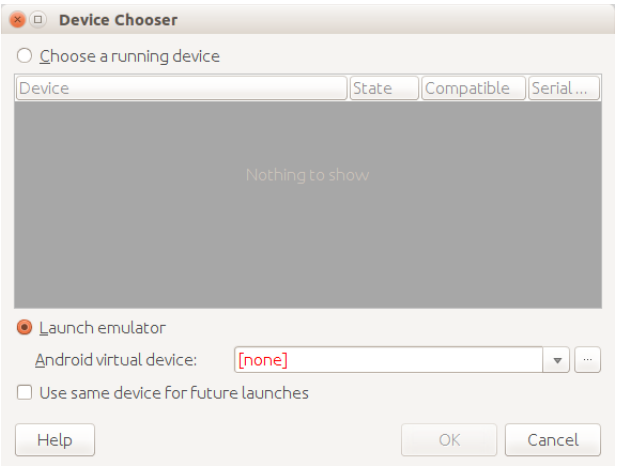
Po určité době se vytvoří nový projekt a zobrazí grafický návrhář rozložení. Pro nás je teď zajímavější kód aktivity a tak projektovém editoru (vlevo, záložka project), klikněte ve složce *app/java/cz.ujep.ki.testapplication* na položku *MainActivity* (pozor nikoliv ve složce označené navíc *androidTest*!). Otevře se editor Javy nad kódem dané třídy. Prostředí by mělo vypadat podobně jako na následujícím snímku obrazovky (všimněte si, že zobrazen je celý třídy, veškerá další funkčnost je zděděna).



2.3 Spuštění

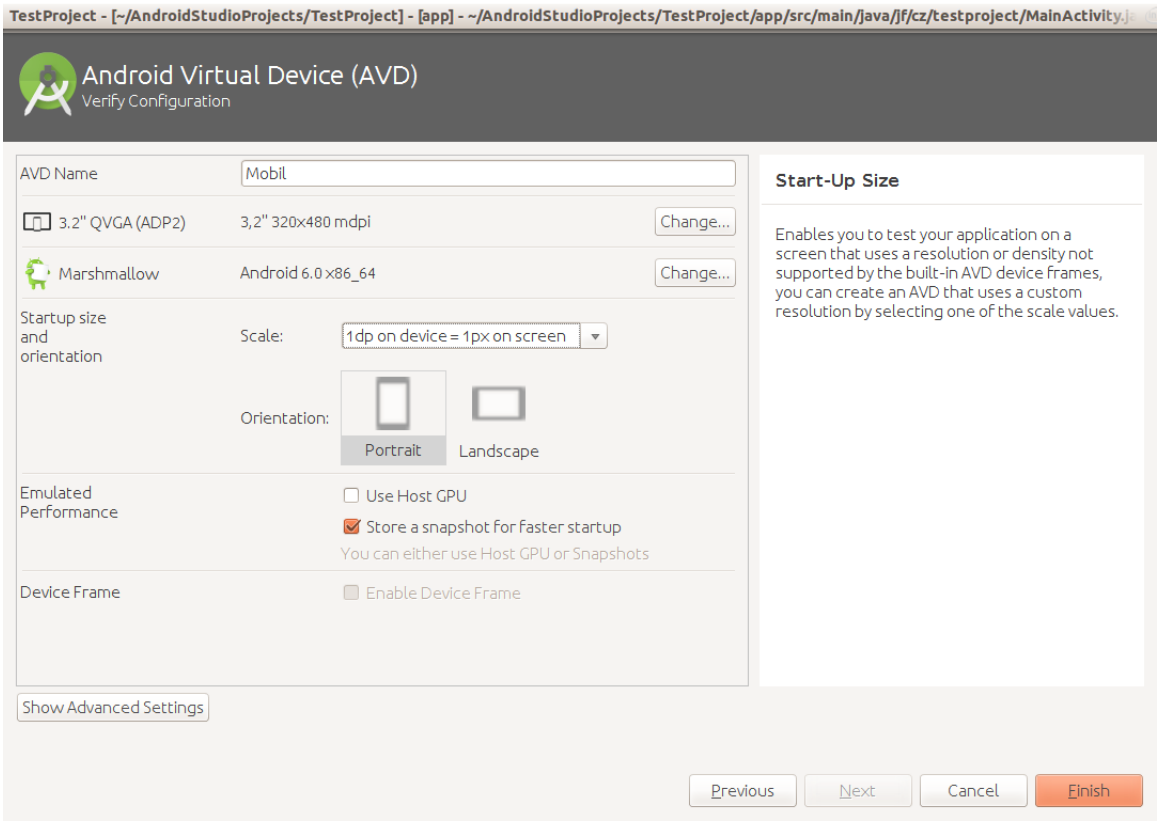
Pokud chcete aplikace ladit na svém zařízení s Androidem postačuje jeho propojení s počítačem pomocí USB kabelu a **povolení ladění** v nastavení Androidu na daném zařízení (sekce *Vývojářská nastavení*). Připojení proveďte ještě před spuštěním aplikace.

Ladící běh spustíte v menu *Run | Run App* (resp. odpovídajícím tlačítkem na nástrojové liště nebo stiskem Shift + F10). Po chvíli by se mělo objevit dialogový box určující zařízení, na němž program poběží.



Pokud máme připojeno Android zařízení a toto zařízení bylo rozpoznáno, tak je přednastaveno (v sekci *Choose a running device*). Pokud zařízení nemáte musíte nakonfigurovat a spustit emulátor. Pro vytvoření nové konfigurace emulátoru stiskněte tlačítko vpravo od rozbalovacího tlačítka „Android virtual device“ (při prvním spouštěné obsahuje text *[none]*). V první fázi se volí typ zařízení podle vzoru (pokud nemáte dostatek paměti a výkonu volte menší zařízení a nižším rozlišením) a následně verze Androidu, který na něm běží. Fungovat budou jen ty verze, pro něž máte nainstalováno i SDK. Proto je nejjednodušší zvolit obraz odpovídající nejvyššímu API (bez Google map), neboť toto SDK se instaluje automaticky. Z důvodů efektivity volte obraz pro procesory Intel (32 nebo 64 podle verze OS na počítači).

Výsledná konfigurace by může vypadat například takto (můžete samozřejmě vyzkoušet i další kombinace, a ani jméno není povinné).



Po návratu z konfigurace dané zařízení nastartujte (vhodné je zaškrtnout volbu *Use same device for future launches*) a po několika desítkách sekund se dočkáte okna s emulátorem, v němž bude po další chvíli spuštěna daná aplikace (prozatím prázdné okno). Spuštění emulátoru je pomalé i na rychlejších strojích a proto okno s emulátorem nechte otevřené. Při dalším testovacím spuštění aplikace se virtuální stroj nemusí startovat a spuštění je výrazně rychlejší.

3 Základní struktura programu a 2D grafika: Mandelbrotka

CÍLE KAPITOLY

Ukázková aplikace pro vykreslení fraktálu Mandelbrotovy množiny ilustruje následující mechanismy programování v Androidu:

1. životní cyklus aplikace
2. základní interakce s tzv. aktivitou (obdoba aplikačního okna u desktopových aplikací) — menu
3. základní 2D vykreslování (bez použití OpenGL ES)
4. využití asynchronních vláken
5. reakce na vstupní události (dotyky)

Tato aplikace není zcela klasickou Androidí aplikací, tvoří však dobrý přechod mezi běžnými desktopovými aplikacemi a komplexnějšími aplikacemi v Androidu. Navíc skvěle vypadá :)

3.1 Mandelbrotova množina

Mandlebrotova množina je jedním z nejznámějších fraktálů.

Madelbrodova množina je množina komplexních čísel c, pro která je posloupnost

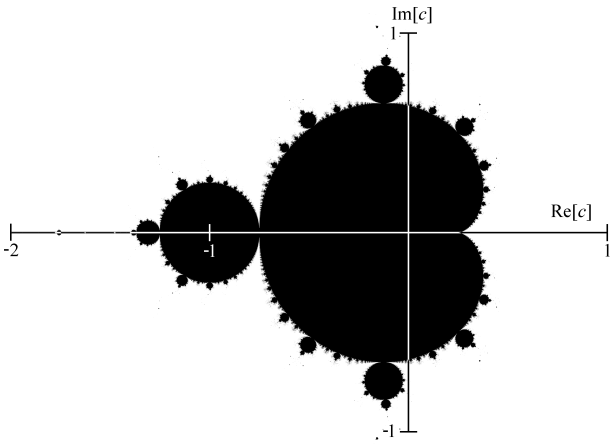
$z_0 = 0, \quad z_{n+1} = z_n^2 + c$

omezená, tj. že splňuje následující podmínku:

Existuje reálné číslo m takové, že pro všechna $n \in \mathbb{N}$ je $|z_n| < m$.

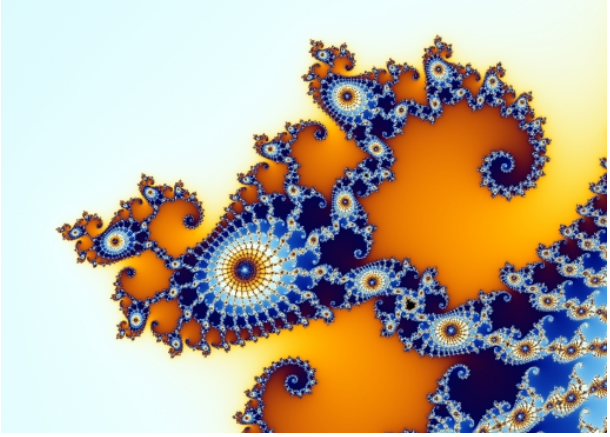
Velmi kvalitní popis tohoto fraktálu najdete na anglické Wikipedii (`http://en.wikipedia.org/wiki/Mandelbrot_set`, z tohoto článku jsou převzaty i ilustrativní obrázky).

Mandelbrotova množina je souvislá množina bodů, jejíž grafem je zvláštní 2D útvar s podivnými střapatým okrajem (avšak bez vnitřní struktury).



V počítačové grafice se však dává přednost representaci, v níž se zohledňuje i rychlost konvergence. Výsledný fraktál tak může být tvořen několika (barevnými) přechody. Tento přístup zvolíme i my, neboť jen tak využijeme krásně barevné displeje současných tabletů a mobilních telefonů.

Typická ukázka konvergentní representace Mandelbrotovy množiny tzv. mořský koník (vyskytuje se mimo jiné u styku jednotlivých kruhovitých a kardoidních útvarů).



Matematik by začal návrhem a programováním algoritmu pro vykreslení fraktálu, praktik však začne vytvářením kostry aplikace, neboť algoritmus může najít přímo na stránkách anglické *Wikipedie* (není v Javě, ale v tzv. pseudokódu, ale převod je snadný).

3.2 Aktivita — jádro Androidí aplikace

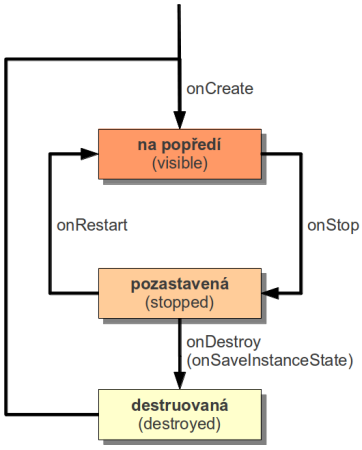
Struktura aplikace je v Androidu ovlivněna několika základními východisky (zjednodušeno):

- aplikace využívá grafické uživatelské rozhraní pro interakci s uživatelem (přičemž jako vstup jsou preferovány dotyky)
- aplikace je v zásadě nesmrtelná, je prováděna (alespoň se tak navenek jeví) od prvního spuštění až po (případnou) deinstalaci
- pouze jedna aplikace je tzv. na popředí tj. využívá displej pro interakci s uživatelem
- hardwarové prostředky jsou relativně silně omezené tj. běžící aplikaci může být odebrán nejen procesor a fyzická paměť, ale i proces a tím i regiony virtuální paměti

aktivita

Z tohoto důvodu je klíčovým prvkem každého interaktivního GUI programu tzv. **aktivita**. Aktivita se stará o grafickou interakci s uživatelem v těch okamžicích, kdy je aplikace na popředí. Pokud je však odsunuta do pozadí, pak se stává neaktivní a po určité době může být spolu s procesem, který ji vykonává, nemilosrdně zlikvidována (a uvolní tak prostředky potřebnějším procesům). V okamžiku, kdy je opět potřeba, se vytvoří nový proces a v něm je instanciována nová aktivita.

Aktivita tak prochází během své (takřka) nesmrtelnosti cyklicky mezi dvěma či čtyřmi stavy:



Při přechodu mezi stavy je jsou volány metody aktivity, v nichž je možno alokovat prostředky (metody *onCreate* resp. *onRestart*), příslušné prostředky uvolňovat (*onStop*, *onDestroy*) resp. ukládat stav aktivity (*onSaveInstanceState*) a následně obnovovat (*onCreate*). Ukládání je nutné, aby navenek vznikala iluze stále existence aktivity. Nyní již můžeme přikročit k vytvoření (vývojářského) projektu z hlavního menu průvodů (předchozí projekt předtím uzavřete).

3.3 Vytvoření projektu a jeho počáteční struktura

Aplikaci pojmenujeme familiárně „Mandelbrotka“. V konfiguraci hlavní aktivity zvolíme opět „Empty Activity“ a nic nezměníme ani na na posledním konfiguračním listě.

Podívejme se nejdříve jaké soubory pro nás IDE vytvořilo (lze je procházet pomocí prohlížeče projektových souborů na levé straně). Většina vygenerovaných souborů jsou XML soubory, které deklarativně popisují konfiguraci aplikace a jejího GUI rozhraní.

Klíčovým souborem je *AndroidManifest.xml* (ve složce *app/manifests*). Pro jeho prohlížení lze využít soubor vestavěných konfiguračních editorů (viz záložky dole). Soubor však lze prohlížet (a editovat) i přímo pomocí XML editoru (záložka označená *AndroidManifest.xml* v editační oblasti).

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="cz.ujep.ki.mandelbrotka" >

   <application
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >
      <activity
         android:name=".MainActivity"
         android:label="@string/app_name" >
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>

</manifest>
```

Manifest aplikace obsahuje některé informace, které jsme zadali při jejím vytváření a některá další nastavení.

U některých nastavení (jako je popisek aplikace *android:label*) je namísto fixní hodnoty atributu použit odkaz do souborů zdrojů. Soubory zdrojů (*resources*) mohou být vytvářeny v několika kopiích např. pro vícero jazyků nebo vícero typu displejů (rozlišení či velikost). Ve všech těchto případech může mít atribut jinou (specifickou hodnotu). Například název aplikace se může lišit podle jazyka, použitá ikona podle rozlišení (při velkém rozlišení by mohla být tak malá, že by ji nešlo snadno identifikovat dotykem).

Klíčovou částí manifestu je specifikace aktivity. Je specifikováno celé jméno její třídy i se jménem balíku, to jest *cz.ujep.ki.android.fiser.MainActivity*, ale především je určeno na jaké podněty zvnějšku bude reagovat. Aktivity jsou v Androidu jsou relativně samostatné programové jednotky, které jsou aktivovány podněty z vnějšku tj. z jiných aktivit. Naše aktivita reaguje na podnět od aktivit, jež fungují jako *launcher*, tj. spouštěč aplikací (to nemusí být jen jediná aktivita). To znamená, že se zařadí do seznamu spustitelných aplikací (jež v běžném rozhraní dostupná přes ikonu mřížky čtverečků).

Dalším typem souborů jsou zdroje, které naleznete v adresáři *res* projektu. Ty jsou členěny podle typů (rozvržení, jednoduché hodnoty, styly, apod.), přičemž některé se vyskytují ve více verzích podle konfigurace cílového systému (API, displej, jazyk, apod.). Tj. například kreslitelné objekty (drawable), se člení podle rozlišení displeje (low dpi, medium dpi, high dpi a extra vysoké rozlišení [kdy se dočkáme čtyřikrát x-dpi?]), styly podle čísla verzí apod. Toto rozdělení je však jen počátkem, v plnohodnotné aplikace mohou být i desítky různých variant.

Pro začátečníka jsou klíčové soubory ve složce *values*. Například v *res/values/strings.xml* jsou řetězce, které vidí uživatel aplikace, včetně např. popisku aplikace:

```
<string name="app_name">Mandelbrotka</string>
```

Nyní však pozornost přesuneme na zdrojové soubory ve složce *java*. Zde jsou organizovány podle javovských balíčků. My máme prozatím jen jeden balíček, v němž leží jen jeden soubor s jedinou třídou (v Javě může být jen jediná veřejná třída v souboru). Instancí této třídy bude jediná aktivita naší aplikace.

```
package cz.ujep.ki.android.fiser;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class MainActivity extends Activity {

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

}
```

Třída *MainActivity* je odvozena ze třídy *Activity* (přesněji *android.app.Activity*) a předefinovává dvě její metody: *onCreate*, která je volána při každém vzniku aktivity (včetně znovuzrození viz životní cyklus aktivity výše) a metodu, která připravuje na zobrazení menu (*onCreateOptionsMenu*). Obě se metody mají něco společného — využívají prostředky z *resource* adresáře. V *onCreate* je vyplněno okno aktivity pomocí rozvržení (soubor *res/layout/activity_main.xml*), v *nCreateOptionsMenu* pak menu položkami ze souboru *res/menu/main.xml*). Soubory prostředků se neuznačují přímo, ale pomocí symbolických konstant, které jsou pro každý soubor v adresáři prostředků automaticky vytvořeny. To má dvě hlavní výhody:

1. symbolická konstanta vždy odkazuje aktuální variantu podle konfigurace systému (rozlišení, jazyk apod.)
2. funguje doplňování syntaxe. Stačí použít specifický prefix balíku R, zvolit ze seznamu typ prostředku a pak přímo jeho jméno.

Aplikaci lze již v tomto okamžiku spustit (*Run | Run 'app'*, Shift-F10). Start emulátoru je relativně pomalý a na pomalých zařízeních může trvat i celé minuty. Naštěstí emulátor je nutno spouštět jen jednou (při dalším spuštění lze použít stejnou instanci, proto ji pokud možno nezavírejte).

Po naběhnutí úvodní obrazovky je nutno tažením odemknout obrazovku (stejně jako u fyzického zařízení i když zde je to zcela zbytečné). Po chvíli by se měla objevit aplikace s titulem *Mandelbrotka*, která je však zcela prázdná.

3.4 Vytvoření třídy pohledu (view)

pohled

Dalším krokem je vytvoření tzv. **pohledu** — *view*. Pohled je aktivní část okna aktivity, odpovídá tudíž

widgetům resp. řídícím prvkům (*controls*), jak je znáte z ostatních GUI knihoven. Bázová třída (*android.view.View*) je pouze pravoúhlá oblast bez viditelných grafických prvků a bez možnosti interakce. Z této třídy jsou přímo i nepřímo odvozeny všechny aktivní či pasivní prvky prvky, počínaje textovými popisky, přes různá tlačítka až po složité seznamy.

Náš pohled bude zobrazovat Mandelbrotovu množinu přímým vykreslování a nebude tudíž potřebovat žádnou dodatečnou funkčnost nabízenou odvozenými třídami pohledů. Odvodíme ji tedy přímo ze třídy *android.view.View*.

V *IntelliJ* se nové třídy vytvářejí pomocí průvodce. Protože i tato nová třída by měla ležet v javovském balíčku *cz.ujep.ki.....*, tak je vhodné průvodce vyvolat z kontextového menu, které získáme stiskem pravého tlačítka myši nad jménem balíčku.

V něm zvolte volbu *New* | *UIComponent* | *CustomView* a zadejte jméno *MandelbrotView*.

```
public class MandelbrotView extends View {

      public MandelbrotView(Context context) {
            super(context);
      }

      public MandelbrotView(Context context, AttributeSet attrs) {
            super(context, attrs);
      }

      public MandelbrotView(Context context, AttributeSet attrs, int defStyle) {
            super(context, attrs, defStyle);
      }
}
```

To je však bohužel poslední část kódu, jíž lze generovat plně automaticky. Nyní již musíme začít myslet.

Nejdříve se zamyslíme nad atributy daného pohledu. Pohled je určen dvěma soustavami souřadnic. První je v zásadě pevná a je určena zobrazovacím zařízení či přesněji obdélníkem v sítě pixelů, na němž je pohled zobrazen (ve skutečnosti není tento obdélník zcela fixní, mění se například při otočení zařízení). Tato soustava souřadnic má v souladu s tradicí počítačové grafiky počátek (0,0) v levém horním rohu, pravý dolní roh má souřadnice (šířka - 1, výška - 1).

Druhá soustava souřadnic je dána výřezem komplexní roviny, na níž je Mandelbrotova množina definována. Může být průběžně měněna, čímž je možno dosáhnout zdánlivého přiblížení či vzdálení (zoom). Na počátku je zobrazován výřez komplexní roviny v rozsahu -2 až 1 na ose x a -1 až 1 na ose y (do tohoto výřezu se vejde celá Mandelbrotova množina). Převody mezi těmito dvěma soustavami souřadnic tvoří důležitou část vykreslovací části kódu.

Než přistoupíme k implementaci je ještě nutné vzít v potaz rychlost výpočtu Mandelbrotovy množiny a jejího vykreslování. Výpočet je totiž relativně náročný a na pomalejších strojích může trvat i několik desítek sekund (především v případě, že nemají hardwarovou podporu výpočtů v pohyblivé řádové čárce tj. FPU). Během této doby by aktivita nereagovala na akce uživatele (včetně například pokusu o její zdánlivé ukončení přepnutím na domovskou obrazovku). Protože je toto chování nežádoucí, snaží se Android tyto aplikace detekovat a pokud nereagují déle než zvolený interval (v řádu vyšších jednotek vteřin), pak je nemilosrdně ukončí. Jinak řečeno uživatel by se nemusel vykreslení ani dočkat.

Jediným řešením je přenesení výpočtu Mandelbrotovy množiny do zvláštního vlákna běžícího na pozadí (Android vlákna nejen, že podporuje, ale v mnoha případech i doporučuje). Tím však vzniká další problém — vlákno na pozadí nemůže kreslit do pohledu (resp. obecně nijak manipulovat s GUI). Proto je nutné kreslení rozdělit do dvou fází. Nejdříve je ve výpočetním vlákně využito kreslení do bitmapy uložené v paměti (to může trvat i desítky sekund) a až poté jeho skončení je v hlavním (GUI) vlákně bitmapy zkopírována do pohledu (to už trvá jen milisekundy). Během výpočtu sice omezíme interakci s uživatelem (nemůže například používat dotyky pro zoom), ale základní ovladatelnost zůstává zachována.

Tento rozbor nám již umožňuje navrhnout datovou representaci stavů pohledu (tj. neveřejné členy instancí třídy).

```
private Rect dc;
private RectF mc;
private Bitmap actualBitmap = null;
private Paint paint = new Paint();
private boolean backgroundThread = false;
public float progress = 0.0f;
```

Datový člen dc representuje výřez v souřadnicovém systému displeje (zkratka za ***d**isplay **c**oordination*). Pro representaci je využívána instance třídy *android.graphics.Rect*, která representuje obdélník v celočíselných souřadnicích. Člen *dc* naproti tomu representuje odpovídající obdélník v komplexní rovině (zkratka za ***m**athematics **c**oordination*). Je to instance třídy *android.graphics.RectF*, která representuje obdélník v reálných souřadnicích (tj. v souřadnicích typu *float*).

Poznámka: Třídy jsou uvedeny bez prefixu balíčku (= jmenného prostoru), neboť všechny použité balíčky jsou importovány. Importování je výrazně zjednodušeno tím, že příslušný příkaz je automaticky vložen při doplňování syntaxe. Stačí jen napsat první tři–čtyři znaky jména třídy a pak stisknout Ctrl+Space. Z nabídnutého seznamu vyberte požadovanou třídu, která je pak nejen doplněna, ale je vložen i příkaz pro importování příslušné třídy z balíčku (pokud již není samozřejmě obsažen).

Další datový člen representuje bitmapu, která má být aktuálně vykreslována (instance třídy *android.graphics.Bitmap*). Zbývající členy jsou využívány buď při vykreslování (*paint*) nebo souvisí vláknem generujícím bitmapu (*progress* a *backgroundThread*), těm se budeme věnovat až o něco později.

Všechny datové členy jsou v souladu s principem zapouzdření privátní a nelze je tudíž používat mimo instance třídy *MandelbrotView*. V případě matematických souřadnic (*mc*) by však bylo záhodno umožnit získání a dokonce i změnu zvnějšku (například, pokud bychom chtěli v budoucnu podporovat ruční nastavení výřezu nebo galerii oblíbených výřezů). Proto dodáme dvojici metod pro získání (*getter*) a nastavení (*setter)* tohoto atributu (vlastnosti).

```
public RectF getMc() {
      return mc;
}

public void setMc(RectF mc) {
      this.mc = mc;
}
```

Getter i *setter* je prozatím triviální (nic se nekontroluje, nemění se ani representace). Triviální *gettery* a *settery* je možno automaticky generovat pomocí nástroje *Source | Generate Getters and Setters.*

Vytvořený *setter* ihned použijeme pro inicializaci matematické soustavy souřadnic v konstruktorech (nastavíme výřez, který zajistí vykreslení celé množiny). Protože to musíme učinit ve všech třech konstruktorech, vytvoříme pomocnou metodu pro inicializaci (v Javě nelze vzájemně volat konstruktory).

```
public MandelbrotView(Context context) {
   super(context);
   init();
 }

public MandelbrotView(Context context, AttributeSet attrs) {
    super(context, attrs);
    init();
}

public MandelbrotView(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
```

```
        init();
    }


    public void init() {
        setMc(new RectF(-2.0f, 1.0f, 1.0f, -1.0f)); //implicitní výřez
    }
```

Stejně jako je tomu v případě ostatních GUI knihoven, je jádrem implementace uživatelského pohledu ošetření událostí vznikajících při interakci naší aplikace s uživatelem a okolím. V první verzi budeme ošetřovat jen dvě události: změnu velikosti pohledu (musíme změnit souřadnice zařízení a nastartovat generování nové bitmapy) a požadavek na překreslení (musíme nakreslit aktuální bitmapu). Návrhový vzor pozorovatel (*observer*), který se pro ošetření událostí používá (objekt pohledu se zaregistruje u manažera událostí a pokud daná situace nastane pak je mu automaticky předáno řízení) je v Javě implementován pomocí dynamického polymorfismu založeného na rozhraních. Každý objekt, který chce být informován o změnách musí implementovat rozhraní, jehož metody ošetřují jednotlivé události (jména těchto rozhraní jsou standardně zakončena slovem *Listener*, neboť objekt jakoby naslouchá na konci telefonní linky a je probuzen při vzniku události).

V případě událostí změny velikosti pohledu a požadavku překreslení však není nutné žádné rozhraní explicitně implementovat, neboť příslušné (naslouchací) rozhraní implementuje již bázová třída *View*. Proto stačí dané metody jen předefinovat.

```
    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        dc = new Rect(0, 0, w, h);
        generateNewBitmap();
    };

    @Override
    protected void onDraw(Canvas canvas) {
        if (actualBitmap == null){
          paint.setColor(Color.WHITE);
          canvas.drawColor(Color.BLACK);
          String text = String.format("Wait, please (%.0f%%)", progress * 100);
          canvas.drawText(text, 30, 30, paint);
          return;
        }
        canvas.drawBitmap(actualBitmap, 0, 0, paint);
    }
```

Kód metody *onSizeChanged* (je volána při změně velikosti pohledu) je více než jednoduchý. Pomocí parametrů metody totiž získáme novou šířku *w* a šířku *h* pohledu. Poté stačí jen definovat obdélník definující novou zobrazovací soustavu souřadnic *dc* a nastartovat generování nové bitmapy.

Vykreslovací metoda není o mnoho složitější. Pokud není bitmapa k dispozici (to nastává nejen na začátku, ale i v okamžiku kdy se připravuje nová bitmapa), pak je vykreslen jen bílý text s upozorněním (na černém pozadí). Kreslící plocha je (jak je běžně zvykem) representována instancí třídy *Canvas* (*android.Graphics.Canvas*). Tato instance však nenese stav vykreslovacích nástrojů je tzv. bezestavová (jako je tomu např. v GDI+).

Z tohoto důvodu se podstatně liší metoda pro vyplnění pozadí (*drawColor*) od metody (*drawText*). Zatímco vyplnění nepotřebuje znát žádný stav (je jednoznačně zřejmé, co má udělat — nastavit každý pixel na předanou barvu), je vykreslení textu složitější. Výsledek musí reflektovat nastavení písma, barvy (popředí), transformační matici, apod. Tyto informace však nejsou uloženy jako (globální) stav plátna, ale musí být předány jako atributy instance třídy *android.graphics.Paint*.

Objekt této třídy vytvořen v jedné kopii již při vzniku pohledu (odkazuje na něj datový člen *paint*), neboť v rámci vykreslovací metody by se neměly vytvářet nové objekty (jako varování to označí tzv.

lint tj. program, který na pozadí kontroluje prohřešky proti stylu uplatňovaném při programování pro Android). Před každým použitím v metodě pro vykreslení textu je však nastaven jeho atribut (vlastnost) *Color*, který u textu representuje barvu popředí. Ostatní atributy (např. použitý řez písma) si zachovávají standardní nastavení (u písma je to např. systémový font).

Pro formátování výstupního řetězce je použita statická metoda třídy *String* (pro C# programátory: pozor název třídy musí začínat velkým písmenem). Pro formátování se používají stejné popisovače jako v jazyce C (a dalších mnoha jazycích, bohužel mimo C#). Kromě fixního textu je vypsán i údaj o pokroku při generování nového obrázku. Ten je uložen v datovém členu *progress* a nabývá hodnot [0, 1] (kde 1 representuje přirozeně 100%).

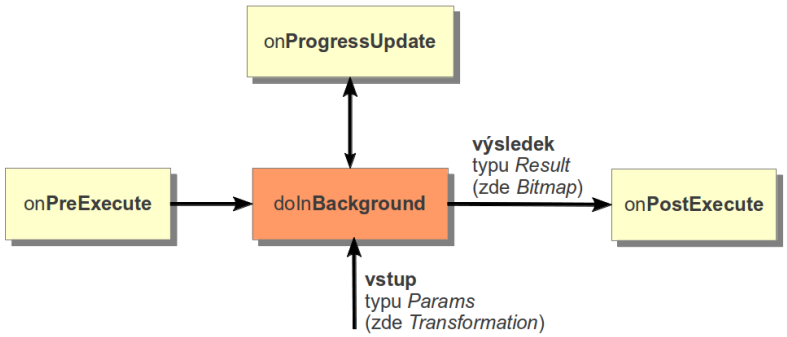
Nyní už se pomalu k blížíme k jádru aplikace, neboť nám zbývá již jen jediný krok — vygenerování bitmapy s Mandelbrotovou množinou. Jak jsem však již předeslal, vše je zkomplikováno tím, že tato činnost musí být provedena ve vlastním vlákně.

API Androidu nabízí hned několik tříd, které udělají z Vašeho programu vícevláknovou aplikaci. Základním řešením je podpora klasického javovského řešení — třídy *Thread*. Její využití je snadné, stačí buď v odvozené třídy předefinovat metodu *run* (ta pak bude vykonána v nově vzniklém vlákně) nebo zavolat konstruktor bázové třídy a předat jí instancí implementující rozhraní *Runnable* (povětšinou se používá anonymní implementace rozhranní). Toto řešení však neřeší komunikaci mezi nově vzniklým vláknem a hlavním (GUI vláknem) a všeobecně je považováno za málo robustní (tj. snadno jej použijete chybným způsobem). I ve standardní Javě je tudíž považováno za překonané.

Android proto nabízí hned několik robustnějších řešení. Pokud Vám stačí chování typu „udělej nějakou úlohu v novém vlákně a po jejím skončení proveď (jednorázovou) obsluhu ve vlákně hlavním“, pak je doporučeným řešením třída *AsyncTask<Params, Progress, Result>* (třída je generická). Objekt této třídy zajistí vykonávání tří postupných akcí (metod) v přesně definovaném pořadí: nejdříve je vykonána akce popsaná metodou *onPreExecute* a to v hlavním vlákně (může tedy přistupovat ke objektům GUI včetně pohledů), poté je vyvolána metoda *doInBackround*, které je předán objekt (či pole objektů) třídy, jež je použita na místě typového parametru *Params* (u nás to bude objekt representující transformaci mezi souřadnicemi pohledu a souřadnicemi komplexní roviny). Jak je zřejmé již z názvu metody, je tato část úlohy vykonána na pozadí ve zvláštním vlákně. Uvnitř této metody tak nelze přistupovat ke GUI prvkům a nelze doporučit ani přístup k ostatním objektům, jejichž kód je vykonáván hlavním vláknem (bezpečný je pouze přístup k objektům odkazovaným pouze instancí třídy *AsyncTask*, jež vykonává úlohu).

Metoda *doInBackround* na základě vstupních parametrů vytvoří na pozadí výsledek, jehož typ je určen třetím typovým parametrem generické třídy (označován jako *Result*). Tento objekt je předán na vstup metody *onPostExecut*e. Ta je opět vykonána v hlavním GUI vlákně. Její funkcí je změnit GUI podle výsledného objektu (v našem případě je výsledkem bitmapa a metoda *onPostExecute* zajistí vykreslení bitmapy).

Celý proces je ilustrován na následujícím obrázku:



Obrázek navíc ukazuje další možnost, kterou třída *AsyncTask* nabízí — úloha na pozadí může občas vyvolávat metodu *onProgressUpdate* na popředí (tj. v GUI vlákně), která zajišťuje aktualizaci informace o pokroku dosaženém při vykonávání úlohy na pozadí (aby uživatel nepodlehl představě, že se nic

neděje). Metodě *onProgressUpdate* lze předat objekt, který pokrok kvantifikuje. To může být instance libovolné třídy (jméno třídy je druhým tj. prostředním parametrem generické třídy *AsyncTask*). V našem případě to bude reálné číslo v rozsahu [0,1] určující jaká část bitmapy je již hotova.

Bohužel v Javě nelze použít typ *float* resp. *double* jako typový parametr generické třídy, neboť to musí být skutečná třída a tou elementární typy v Javě nejsou. Jsou totiž z důvodů efektivity representovány jako přímé hodnoty nikoliv jako plnohodnotné objekty, které jsou z opatřeny dodatečnými informacemi, a z proměnných jsou pouze odkazovány (tj. jsou to na rozdíl od přímých hodnot vždy referenční typy). Stejný problém je nutno řešit i jazyce C#, ale zde se využívá automatický převod přímé hodnoty na objekt a zpět (tzv. boxing a unboxing), který tento rozdíl před uživatelem zcela ukrývá. V Javě je však nutno explicite použít tzv. obalující typ *Float* (všimněte si rozdílné velikosti prvního písmene). Tento typ obaluje hodnotu typu *float* do objektu, který lze použít i v generických konstrukcích, a jenž se ve většině případů implicitně konvertuje na původní hodnotový typ a vice versa (tj. i v Javě je automatický boxing a unboxing, ale není bohužel zcela transparentní).

Než přejdeme k implementaci třídy odvozené z *AsyncTask*, je nutno vyřešit ještě dva problémy spojené s předáváním hodnota a vzájemnou komunikací objektů.

První problém spočívá v parametru metody na pozadí. Aby bylo možno vygenerovat bitmapu je nutno znát obě soustavy souřadnic — jak obdélník popisující šířku a výšku pohledu na displeji tak i odpovídající výřez komplexní roviny. Bohužel parametrem může být jen jediný objekt (to není zcela pravda, ale proměnný počet parametrů nabízený Javou náš problém neřeší). Proto si vytvoříme pomocnou třídu, jejíž instance budou fungovat jako přepravky objekty třídy *Rect* (souřadnice displeje) a *RectF* (souřadnice komplexní roviny).

staticky vnořená třída

Abychom zbytečně neexportovali tento nový typ navenek (je používán jen uvnitř pomocných metod třídy pohledu) budeme ji definovat jako tzv. **staticky vnořenou třídu** (uvnitř třídy *MandelbrotView*).

```
class MabdelbrotView {
            ...
      static class Transformation {
            public Rect dc;
            public RectF mc;

            public Transformation(Rect dc, RectF mc) {
                  this.dc = dc;
                  this.mc = mc;
            }
      }
            ...
}
```

Staticky vnořená třída nemá se svou hostitelskou třídou příliš mnoho společného. Hostitelská třída nabízí pouze samu sebe jako jmenný prostor (tj. jen uvnitř hostitelské třídy je vnořená třída dostupná přímo přes svůj identifikátor, vně hostitelské třídy je nutno použít jméno kvalifikované hostitelskou třídou tj. např. *MandelbrotView.Transformation*). Navíc získají instance hostitelské třídy přístup k privátním členům třídy vnořené a vice versa. To se nám hodí, neboť není nutno vytvářet *gettery* a *settery* pro datové členy vnoření třídy, i když jsou označeny jako privátní.

Druhý problém je obdobný. Uvnitř instancí objektů naší třídy odvozené z *AsyncTask* potřebujeme přístup k objektu aktuálního pohledu. Úloha totiž musí (ve své GUI části) nastavovat bitmapu a také si vynucovat překreslení po její změně (aby se nová bitmapa vůbec objevila na displeji). Naše implementace navíc mění čítač pokroku (= datový člen *progress*) a příznak běhu úlohy na pozadí (datový člen *backgroundThread*). Jinak řečeno instance úlohy na pozadí musí mít přístup k pohledu, který danou úlohu vytvořil.

To lze zajistit vložením odkazu na pohled do instance třídy odvozené z *AsyncTask<Transformation, Float, Bitmap>*. Protože se však jedná o odkaz na objekt jiné třídy, nelze přímo přistupovat k privátním členům pohledu, což jsou bohužel všechny členy které hodláme nastavovat. Jejich zveřejnění či dodání veřejných getterů a setterů není řešením, neboť nechceme aby měl přístup každý (jen a pouze objekty

úloh na pozadí). To lze sice vyřešit vhodnou volbou přístupových specifikátorů je to však poněkud komplikované (v literatuře, kterou si bohužel nepamatuji, je systém přístupových práv v Javě označován jako barokotvarý, což může být po návštěvě drážďanské gämeldegalerie zcela pochopitelné a přiléhavé přirovnání).

vnitřní třída

Naštěstí existuje ještě jedno řešení. Odvozená třída úloh může být definována jako **vnitřní třída**, tj. jako vnořená třída bez specifikace *static* (někteří ji proto označují jako *instančně vnořenou*). Instance instančně vnořené třídy mají s instancemi hostitelské třídy mnohem intimnější vztah. Každá instance vnořené třídy totiž obsahuje (skrytý) odkaz na instanci hostitelské třídy, která ji vytvořila (je to obdoba uzávěrů známých z funkcionálních jazyků). Uvnitř metod instancí vnořené třídy je tak možno přímo přistupovat k datovým členům a metodám tvůrcovské instance hostitelské třídy a to přímo přes *this* (jako by byly zahrnuty do přímo do instance vnořené třídy), přičemž *this* lze samozřejmě ve většině kontextů vynechat.

Definice vnořené třídy odvozené z *AsyncTask* je poněkud delší (obsahuje totiž mimo jiné i vlastní algoritmus vykreslování Mandelbrotovy množiny).

```
class MabdelbrotView {
            ...
 class BitmapAsyncTask extends AsyncTask<Transformation, Float , Bitmap> {
      @Override
      protected void onPreExecute() {
            if(actualBitmap != null)
                  actualBitmap.recycle();
            actualBitmap = null;
            backgroundThread = true;
            progress = 0f;
            invalidate();
      }

      @Override
      protected Bitmap doInBackground(Transformation... params) {
            return getBitmap(params[0]);
      }

      @Override
      protected void onPostExecute(Bitmap result) {
            actualBitmap = result;
            backgroundThread = false;
            invalidate();
      }


      @Override
      protected void onProgressUpdate(Float... values) {
            progress = values[0];
            invalidate();
      }

      private Bitmap getBitmap(Transformation t) {
            Rect dc = t.dc;
            RectF mc = t.mc;
            final int max_iteration = 256;
            final int progressStep = dc.width() / 10;
            int[] palette = new int[max_iteration+1];
            float x0, y0, x, y, xtemp, xx, yy;
```

```
            int iteration;

            for(int i=0; i < max_iteration+1; i++) {
                  palette[i] = Color.rgb((2*i)%256,(3*i)%256, (5*i)%256);
            }

            Bitmap b = Bitmap.createBitmap(dc.width(),
                                           dc.height(),
                                     Bitmap.Config.RGB_565);
            for(int dx = 0; dx < dc.width(); dx++) {
                  if(dx % progressStep == 0)
                        publishProgress((float)dx / dc.width());
                  for(int dy = 0; dy < dc.height(); dy++) {
                        x0 = mc.left + mc.width() * (dx - dc.left) / dc.width();
                        y0 = mc.bottom - mc.height() * (dy - dc.top) / dc.height();
                        xx = yy = x = y = 0.0f;
                        iteration = 0;

                        while ( xx + yy < 4 && iteration < max_iteration ) {
                              xtemp = xx - yy + x0;
                              y = 2*x*y + y0;
                              x = xtemp;
                              xx = x*x;
                              yy = y*y;
                              iteration++;
                        }
                        b.setPixel(dx, dy, palette[iteration]);
                  }
            }
            return b;
      }
 }
}
```

Podívejme se nejdříve na implementaci, které definují jednotlivé fáze úlohy. V metodě *onPreExecute* jsou vykonávány jen pomocné úlohy. Za prvé je aktuální bitmapa zobrazovaná v pohledu nastavena na *null* (tj. aktuální bitmapa není k dispozici, namísto toho je zobrazeno upozornění, že výpočet probíhá a je nutno čekat ne jeho dokončení). Pokud už byla nějaká aktuální bitmapa definována, pak je uvolněna paměť, kterou využívá v rámci systému (metoda *Bitmap.recycle*). Původní bitmapa totiž stále zůstává v paměti, i když již na ní neodkazuje žádný odkaz a to včetně paměti ležící mimo objekt (ta je spravována operačním systémem). Objekt bude sice nakonec odstraněn a dodatečná paměť uvolněna finalizátorem (destruktorem), ale k tomu dojde až v okamžiku kdy není dostatek paměti na hromadě spravované Javou a je tudíž zavolán garbage collector. K tomu může dojít až o mnoho sekund či minut později, kdy už může být pozdě (dodatečná paměť je spravována operačním systémem nikoliv Javou). Proto je vhodné prostředky OS uvolnit hned jak již nejsou potřeba (v C# se pro stejný účel používá návrhový vzor založený na metodě *Dispose* z rozhraní *IDisposable*).

Metoda *onPreExecute* dále nastavuje příznak úlohy běžící na pozadí, aby tak mohlo být zabráněno běhu více souběžný úloh na pozadí. I když je to v zásadě možné, může to příliš zatížit procesor a komplikovalo by se i zobrazení informace o průběhu. Příznak *backgroundThread* je povětšinou synchronizován s odkazem na aktuální bitmapu (tj. platí, že je nastaven na *true* právě tehdy, když má proměnná *actualBitmap* hodnotu *null*), neplatí to však při vzniku pohledu (bitmapa není k dispozici, ale žádná úloha na pozadí neběží).

Po nastavení čítače pokroku na 0 (zatím není nic hotovo) je volána metoda *MandelbrotView.invalidate* (ta patří stejně jako nastavované datové členy do odkazovaného hostitelského objektu-pohledu). Tato

metoda zneplatní veškerý viditelný obsah pohledu, tím že vloží požadavek na překreslení celé jeho plochy do fronty požadavků. Po určité (povětšinou velmi krátké době) je systémem zavolána metoda *MandelbrotView.onDraw,* která fyzicky zajistí překreslení pohledu (zde nakreslí upozornění na čekání se zobrazením pokroku). Metodu pro zneplatnění pohledu můžeme bezpečně volat, neboť jsme stále v hlavním GUI vlákně. Tím předběžné nastavení končí.

Po určité době po dokončení metody *onPreExecute* (kterou opět nelze určit, je však opět ve většině případů velmi krátká) je v jiném vlákně vyvolána metoda *doInBackground*. Tato metoda běží paralelně s GUI vláknem (v případě, že máte k dispozici více jader pak se může jednat o skutečný paralelismus), tj. GUI vlákno zůstává responzivní (tj. může téměř okamžitě reagovat na požadavky uživatele a systému). Tělo metody je velmi stručné, neboť se pouze volá pomocná metoda pro generování bitmapy (což však může trvat i desítky sekund).

Metoda *onPostExecute* (ta je opět vykonána v GUI vlákně s jistým zpožděním po ukončení předchozí metody) získává nově vytvořenou bitmapu, jíž uloží do datového členu *actualBitmap* (ten samozřejmě leží v hostitelském objektu pohledu), nastaví příznak běžící úlohy na *false* (tj. již je možné spustit další úlohu) a vyžádá si její překreslení zneplatněním současného obsahu pohledu (tj. po jisté době je volána metoda *MandelbrotView.onDraw*).

Velmi jednoduchá je i metoda *onProgressUpdate*, která je volána několikrát během plnění bitmapy. Ta pouze nastaví čítač pokroku na předanou hodnotu (ta je předána z metody *getBitmap* běžící na pozadí) a zneplatní pohled. Při následném vykreslení je již zobrazena nová hodnota.

Teprve nyní se dostaneme ke kódu, který fraktál generuje do bitmapy. Většina kódu metody getBitmap je vytvořena na základě pseudokódu převzatého z *Wikipedie* (kód je pouze přepsán do Javy a mírně upraven):

```
For each pixel (Px, Py) on the screen, do:
{
  x0 = scaled x coordinate of pixel
  y0 = scaled y coordinate of pixel
  x = 0.0
  y = 0.0
  iteration = 0
  max_iteration = 1000
  while ( x*x + y*y < 2*2 AND iteration < max_iteration )
  {
    xtemp = x*x - y*y + x0
    y = 2*x*y + y0
    x = xtemp
    iteration = iteration + 1
  }
  color = palette[iteration]
  plot(Px, Py, color)
}
```

Mandelbrot set. (2013, October 6). In *Wikipedia, The Free Encyclopedia*. Retrieved 18:30, October 6, 2013, from http://en.wikipedia.org/w/index.php?title=Mandelbrot_set&oldid=575943730

Z tohoto důvodu uvádím jen pár poznámek. Bitmapa je vytvořena voláním statické tovární metody *Bitmap.createBitmap*. Tato metoda kromě šířky a výšky bitmapy očekává formát pro uložení jednotlivých pixelů. Z důvodů úspory paměti je použit formát RGB_565, který pro ukládání každého používá 2 byty (toho 5 bitů pro červenou a modrou složku a 6 bitů pro složku zelenou). Velikost bitmapy totiž může být relativně velká (při rozlišení 320x480 je to i v kompaktním formátu 300 KiB).

Podobně se šetří i při representaci reálných čísel. Na místo representace dvojitou přesností (*double*) je použita přesnost jednoduchá (*float*). Cílem tentokrát není úspora paměti (proměnných typu *float* je použito jen pár), ale urychlení výpočtu (to se projeví především v případě, že zařízení nemá vlastní FPU a výpočty jsou emulovány pomocí celočíselné aritmetiky). Navíc je tento typ kompatibilní s representací souřadnicového systému pomocí *RectF* (typ *float* je obecně preferovaným typem ve 2D grafice).

Všimněte si i vyvolání metody *publishProgres*s, na úrovni cyklu přes sloupce (tj. vnějšího hlavního cyklu programu, bitmapa se kreslí po sloupcích). Tato metoda zajistí nepřímou aktivaci metody *onProgressUpdate* v GUI vlákně (je to nepřímá aktivace, metoda jen přidá požadavek do fronty požadavků hlavního vlákna a ihned se vrátí, update se provede až v okamžiku kdy se na požadavek dostane). Aby se GUI vlákno zbytečně nezatěžovalo, není požadavek volán v každém sloupci, ale jen po dokončení každých přibližně 10% sloupců). Metoda přijímá a následně předává nám již známé číslo v rozsahu [0,1].

Poslední zmínku si zaslouží generování palety (tj. mapování počtu iterací na barvy). I když by bylo obecně vhodné využívat předpřipravenou paletu, je z důvodů stručnosti využit jednoduchý cyklus, který vytváří paletu spojením cyklicky se opakujících barevných složek, přičemž perioda se u jednotlivých složek liší.

Po vytvoření vnitřní třídy implementující úlohu na pozadí už zbývá jen doplnit metodu, která vytvoří instanci této třídy a nastartuje proces jejího vykonávání (tato metoda patří přímo třídě *MondelbrotView*):

```
private void generateNewBitmap() {
      if(!backgroundThread) {
            BitmapAsyncTask task = new BitmapAsyncTask();
            task.execute(new Transformation(dc, mc));
      }
}
```

Podmínka brání vícenásobnému vyvolání výpočtu na pozadí, tj. jen v případě, že neběží jiná úloha na pozadí, dojde k vytvoření instance třídy *BitmapAsyncTask* a k volání metody její *execute* (parametrem je přepravka se specifikací obou soustav souřadnic).

V další fázi je nutno pohled umístit do rozvržení, které popisuje vizuální rozhraní aktivita. Proto je nutné ze správce balíčků otevřít soubor *res/layout/activity_main*. Objeví se rozhraní návrháře.

Návrhář rozhraní aktivit vestavěný do *Android Studia* není příliš intuitivní a navíc se často mění. Jednotlivé akce se tak mohou v různých verzích lišit (a nikoliv jen v detailech).

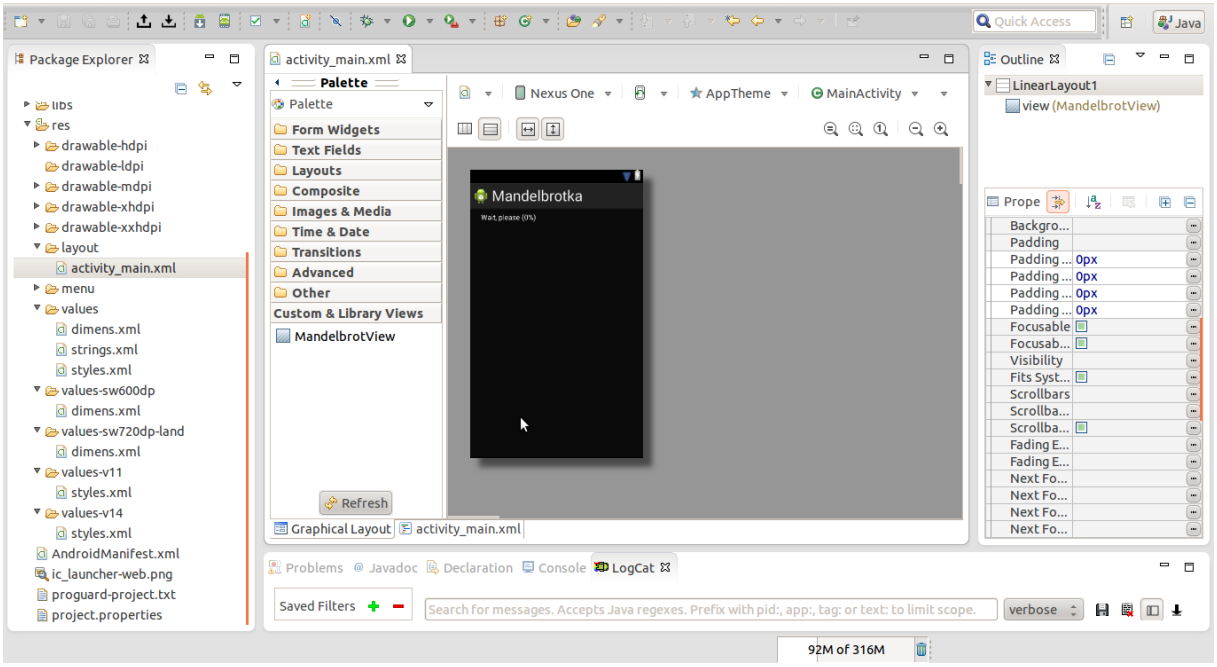
V zásadě je nutno provést pět kroků: za prvé vymazat pohled s textem „Hello, world“ (pokud se tam nachází), změnit hlavní rozvržení (layout) na lineární (ten je pro začátečníka nejjednodušší).

Nyní již můžete přidat nově vytvořený pohled (je dostupný v paletě pohledů v sekci *Custom & Library Views*). Přidání se provede pouhým přetažením do návrhu displeje.

Posledním krokem je odstranění výplně (*paddingu*) kolem pohledu. Protože se jedná o vnitřní okraje (*padding* je součástí pohledu nikoliv okolního rozvržení) stačí zvolit nově přidaný pohled a v editoru vlastností nastavit *padding* ve všech směrech na nula pixelů (*left*, *top*, *right*, *bottom*).

Posledním krokem je pojmenování dané instance pohledu, aby bylo možno tento pohled jednoduše odkazovat z hlavního programu (automaticky zvolený identifikátor je zbytečně dlouhý a u složitějších rozvržení může být matoucí). V property editoru zvolte vlastnost *id* nastavte ji na *@+id/view* (použijte tlačítko […] vpravo, pak stačí zadat jen *view*, prefix *@+i/*, který zajistí vygenerování příslušné javovského symbolu se vloží sám)

Pokud se vše povedlo měli bystě vidět následující obrázek:



Ještě důležitější je pohled do záložky *activity_main.xml* (níže pod paletou a zobrazení displeje). Ten obsahuje návrh v textové (XML) podobě.

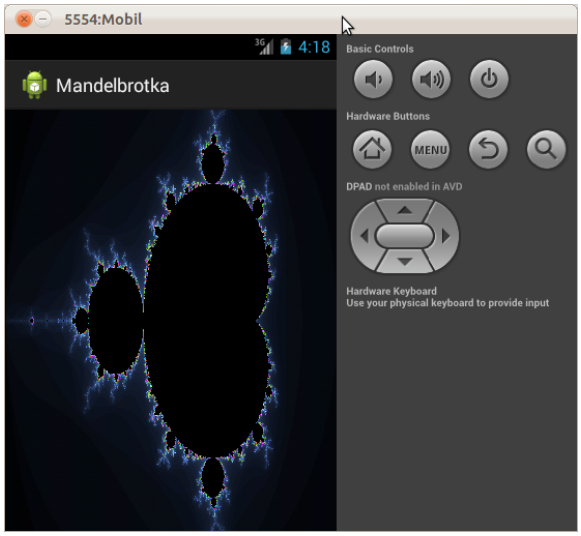
Měl by vypadat takto:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingBottom="0px"
    android:paddingLeft="0px"
    android:paddingRight="0px"
    android:paddingTop="0px"
    tools:context=".MainActivity" >

    <cz.ujep.ki.android.fiser.MandelbrotView
        android:id="@+id/view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</LinearLayout>
```

Pokud se XML v podstatných detailech liší (podstatná jsou jména elementů a hodnoty atributů, nikoliv například pořadí atributů), tak je lze přímo změnit (změna se projeví automaticky i v grafickém návrhu).

Tím máme první návrh aplikace hotový a můžeme ho přeložit a spustit v emulátoru. Pokud je vše OK měli byste vidět nejdříve černou obrazovku s upozorněním a pomalu či rychle rostoucím čítačem pokroku (10, 20, … 90%) a poté i samotnou Mandelbrotku.



3.5 Interakce: dotyky a menu

Navzdory nezpochybnitelné kráse fraktálu, však není aplikace příliš uspokojující, neboť jediné co umí, je zobrazení statického obrázku. Žádná interakce (přepnutí do jiné aplikace např. přes domovské tlačítko však naštěstí funguje), tím spíše animace.

Proto musíme aplikaci trochu rozšířit (ale jen trochu, nebudeme to na začátku přehánět). Protože hlavním požadavkem je přibližovací *zoom* (abychom viděli i krásné detaily), tak vytvoříme jednoduché rozhraní založené na dotycích. Po dotyku se objeví detail centrovaný na jeho středu a zvětšený dvakrát v obou směrech.

Aby konkrétní pohled zachytával dotyky musíme ho registrovat jako příjemce (= listener) příslušných událostí pomocí jeho vlastní metody *setOnTouchListener*. My to provedeme přímo v metodě *onCreate* aktivity (tj. pohled nebude tuto možnost nabízet automaticky bez ohledu na svou domovskou aktivitu). Doplněná metoda bude mít následující tvar (musí být obsažena ve třídě *MainActivity* v souboru *MainActivity.java*, přidán je i nově zavedený datový člen):

```
public class MainActivity extends Activity {
   private MandelbrotView mbw;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      mbw = (MandelbrotView) findViewById(R.id.view);
      if(savedInstanceState != null)
            mbw.setMc((RectF)savedInstanceState.getParcelable("rect"));
      mbw.setOnTouchListener(mbw);
   }
```

V prvním řádku doplněného kódu, získáme odkaz na vložený pohled. Tento odkaz získáme pomocí metody *Activity.findViewById*, jejímž parametrem je symbolický identifikátor pohledu, který je odvozen ze jména zadaného v návrháři (zadáno bylo jméno *@+id/view*, symbolická konstanta má proto tvar *R.id.view*). Vrácený odkaz je typován bázovou třídou *View* a musí být tedy přetypován na správný typ a až pak uložen do připraveného datového členu (prozatím by bylo možno využít i lokální proměnnou, ale odkaz na pohled se nám brzy bude hodit i v dalších metodách).

Teprve nyní můžeme zavolat registrační metodu, která určuje že o veškerých dotycích v rámci našeho jediného pohledu (neboť je adresátem) bude informován tento pohled sám (je totiž i parametrem metody). Obecně může bát příjemcem zpráv libovolný objekt (třeba i naše aktivita).

Po té co vložíme tento objekt, nám *IntelliJ* označí (červeným podtržením a ikonkou v levém pruhu), že v řádku je chyba. Pokud najedeme myší nad podtrženou část (nebo klikneme na ikonku), tak je nám nabídnuto řešení chyby. Bohužel většina navržených řešení nic neřeší (např. přetypování, nebo změna jména metody). Až téměř na konci (alespoň tak je tomu u mne) je však navrženo správné řešení „*Let MandelbrotView implement OnTouchListener*“. Zvolíme toto řešení a podtržení zmizí. Nyní se však chyba pro změnu objeví v souboru *MandelbrotListener.java* (v prohlížeči balíčků se u ikony souboru objeví malá červená ikonka chyby).

Proto si zobrazíme editační okno s tímto souborem, najdeme podtržení v hlavičce třídy *MandelbrotView* a pokusíme se zjistit, jak ji lze automaticky napravit. Tentokrát jsou jen dvě možnosti, z nichž je evidentně správná jen jedna „*Add unimplements method*s“ (řešení „*Make Mandelbrot abstract*“ je nepoužitelné, neboť podpůrný kód musí vytvořit instanci pohledu!).

Po volbě tohoto řešení se do třídy *MandelbrotView* přidá metoda *onTouch* se správnými parametry i typem návratové hodnoty (i když je prozatím téměř prázdná). Tento styl programování (zápis požadovaného dočasně neplatného kódu a až následné automatizované řešení chyb) nejen zrychlí zápis kódu, ale zajistí, že kód je po syntaktické a typové stránce stále správný.

Obsah nově vygenerované metody pro obsluhu dotykových událostí zaměňte za následující kód:

```
public boolean onTouch(View v, MotionEvent event) {
      if (actualBitmap == null)
            return true; // aktuální bitmapa není k dispozici

      int x = (int)event.getX(0);
      int y = (int)event.getY(0);
      float x0 = getMc().left + getMc().width() * (x - dc.left) / dc.width();
      float y0 = getMc().bottom - getMc().height() * (y - dc.top) / dc.height();

       float width = getMc().width() / 2.0f;
       float height = getMc().height() / 2.0f;

       float left = x0 - width / 2.0f;
       float top = y0 - height / 2.0f;

       setMc(new RectF(left, top, left + width, top + height));
       generateNewBitmap();

       return true;
}
```

Kód je relativně dlouhý, ale není složitý. Nejdříve je otestováno, zda právě není k dispozici aktuální bitmapa (a místo fraktálu je tak vidět jen hláška o prodlení). V tomto případě se nic neděje (to brzy napravíme).

Potom je zjištěna pozice dotyku v souřadnicích pohledu (tj. v souřadnicové soustavě representované datovým členem *dc*). To je trochu zkomplikováno tím, že API nativně podporuje vícedotyky (*multitouch*). Proto je nutné dotyky v metodě *getX* indexovat. My vícedotyky prozatím neřešíme a tak zvolíme pozici prvního dotyku (s indexem 0).

Další část kódu přepočítává souřadnice pohledu na souřadnice matematické (komplexní rovinu) a definuje nový výřez (ten má střed v místě dotyku a šířku a výšku poloviční). Po vypočtení nového výřezu je tento nastaven pomocí setteru a je aktivováno generování nové bitmapy (samozřejmě opět na pozadí). Obslužná metoda dotyku pak vrací *true*, čímž potvrzuje že dotyk obsloužila (tak to činí i v případě, že bitmapa není zobrazena, neboť i ignorování je zde obsluhou).

Po novém spuštění by již měla aplikace reagovat na dotyky a zobrazovat i detaily Mandelbrotovy množiny.

Vše se zdá v pořádku, ale při delším používání aplikace zjistíme dva nedostatky (částečně se mohou i prolínat):

1. po opuštění aplikace a opětném návratu do ní, se občas stane, že se místo původního detailu objeví celkový pohled na množinu (tj. aplikace zapomene svůj stav).
2. při otočení zařízení do polohy na šířku (resp. naopak na výšku) během výpočtu bitmapy se po jejím skončení zobrazí původní (neotočená bitmapa)

Nejdříve vyřešíme první problém, který není omezen jen na tuto aplikaci, neboť téměř všechny aplikace v Androidu se musí programově postarat o uchování svého stavu.

Základní princip je jednoduchý. Pokud má být aktivita (dočasně) destruována tj. odstraněna z paměti, pak je volána její metoda *onSaveInstanceState*, která se musí postarat o uložení klíčových informací o svém stavu do jednoduché persistentní databáze, která je dostupná pomocí instance třídy *Bundle*.

serializace

Během tohoto procesu musí být jednotlivé údaje **serializovány** (převedeny do souvislého proudu bytů) tj. ukládané objekty musí být buď jednoduchého typu nebo být tzv serializovatelné (obdoba atributu *Serializable* u .NET). Standardní Java nabízí podporu automatické serializace u většiny tříd, je to však náročný proces a vyžaduje podporu kompilátoru (stejně jako v C#). Android využívá jiný, výrazně odlehčený typ serializace. Většina klíčových tříd implementuje rozhraní *Parcelable* (česky: zabalitelné do přepravního balíku), a tak sami definují metody pro postupnou serializaci a deserializaci (převod zpět do paměti). Toto rozhraní mohou samozřejmě definovat i uživatelské třídy.

```
@Override
protected void onSaveInstanceState(Bundle outState) {
   outState.putParcelable("rect", mbw.getMc());
}
```

Objekt persistentní databáze je do metody předán jako parametr. Databáze je organizovaná jako slovník, tj. jednotlivé hodnoty jsou přístupné přes řetězcový klíč. V našem případě uložíme souřadnice výřezu komplexní roviny pod klíčem "rect"). Třída *RectF*, která výřez representuje naštěstí implementuje rozhraní *Parcelable* (tj. je serializovatelná), takže nám stačí zavolat jedinou metodu pro vložení všech (čtyř) číselných údajů — metodu *putParcelable*.

Obnovení údajů se provádí nejčastěji v metodě *onCreate*. Jen je nutno ošetřit situaci, kdy je tato metoda volána poprvé po instalaci. V tomto případě není databáze s uloženými stavy prozatím k dispozici (a my zobrazíme globální pohled na množinu).

```
if(savedInstanceState != null)
            mbw.setMc((RectF)savedInstanceState.getParcelable("rect"));
```

Tento fragment kódu musí být umístěn na konci metody *onCreate* (resp. přesněji kdekoliv za řádkou, v níž získáme odkaz na pohled pomocí *findByViewId*) a jeho význam je zřejmý. Pokud persistentní databáze (odkazovaná pomocí parametru *savedInstanceState*) již existuje (tj. není *null*), pak je z ní získán (deserializací) objekt representující matematické souřadnice (musí být explicitně přetypován na *RectF*), jenž je následně použit pro nastavení výřezu v pohledu s Mandelbrotovou množinou.

Nyní přistoupíme k problému, jež lze obecně popsat jako ignorování požadavku na překreslení mapy (např. při otočení) v okamžiku, kdy běží výpočet jako reakce na jiný předcházející požadavek např. dotyk. To je zřejmý důsledek skutečnosti, že naše implementace podporuje jen jeden výpočet na pozadí, který je navíc nepřerušitelný. Nezbývá nám tedy prozatím nic jiného, něž požadavek ignorovat (viz následující klíčový kód metody *MandelbrotView.generateNewBitmap*):

```
private void generateNewBitmap() {
      if(!backgroundThread) {
            BitmapAsyncTask task = new BitmapAsyncTask();
            task.execute(new Transformation(dc, mc));
      }
}
```

Protože podpora více souběžných požadavků není v zásadě možná (zatížení systému, problém se zobrazením průběhu), je jediným řešením implementace předčasného přerušení úlohy na pozadí (úloha není dokončena a může být ihned nahrazena jinou).

Nejdříve je nutné předeslat, že i když lze v zásadě vlákno přerušit i nedobrovolně (analogie zabití unixovského procesu) nelze to ve většině případů doporučit. Nelze totiž zaručit, že k přerušení dojede v okamžiku, kdy to nebude mít negativní vliv na další chod programu (tj. když jsou uvolněny všechny prostředky, nebo alespoň správně naalokovány a všechny sdílené datové struktury jsou v konzistentním stavu). Nelze dokonce ani zaručit, že nedobrovolně ukončované vlákno již (nebo naopak ještě) neběží.

Proto je vždy vhodnější volit dobrovolné ukončení, kdy úloha na pozadí cyklicky kontroluje požadovaný stav vlákna, a je-li označeno jako přerušené (přesněji je ve stavu plánovaného přerušení), pak se vlákno dobrovolně ukončí. Tento přístup podporuje i třídy *AsyncTask<>* (a tím samozřejmě i všechny od ní odvozené třídy včetně naší *BitmapAsyncTask*).

Pokud na instanci této třídy zavolána metoda *cancel*, pak mohou nastat dvě eventuality: pokud vlákno ještě běží, pak se nastaví příznak plánovaného ukončení a čeká se na dobrovolné dokončení úlohy. Předpokládá se, že úloha na pozadí průběžně testuje stav objektu *AsyncTask* pomocí metody *isCanceled*. Pokud tato metoda vrátí *true*, pak se úloha (a tím i vlákno) samo ukončí (může však uvolnit prostředky a musí zajistit konzistentní stav dat). Po ukončení úlohy se namísto metody *onPostExecute* zavolá metoda *onCancelled* (v hlavním vlákně!). Až poté se metoda *cancel* ukončí a vrátí hodnotu *true*. Méně pravděpodobná (ale nikoliv nemožná) je i druhá eventualita. Úloha na pozadí již skončila a tak ji již nelze přerušit. Metoda *cancel* vrátí v tomto případě *false*.

V našem případě tak stačí jen mírně modifikovat metodu pro získání bitmapy (*BitmapAsyncTask.getBitmap*) a předefinovat metodu *onCanceled*.

U metody *getBitmap* stačí je dodat test příznaku přerušení. Tento test by se měl průběžně opakovat, tj. měl by být umístěn uvnitř cyklu. V našem případě jsou však využity tři úrovně vnoření cyklů (vnější je přes sloupce bitmapy, prostřední přes jednotlivé pixely a vnitřní přes iteruje přes jednotlivé prvky řady, u nichž je zjišťováno zda jsou omezené či nikoliv).

Vložení na počátek vnějšího cyklu může vést k relativně dlouhé prodlevě (v řádu desetin sekundy), vložení do nejvnitřnějšího může algoritmus zpomalit (i když je testování příznaku velmi rychlé), Zvolil jsem proto zlatou střední cestu a test vložil na začátek cyklu prostředního (tj. per pixel).

```
Bitmap b = Bitmap.createBitmap(dc.width(), dc.height(), Bitmap.Config.RGB_565);
      for(int dx = 0; dx < dc.width(); dx++) { //cyklus přes sloupce
            if(dx % progressStep == 0)
                  publishProgress((float)dx / dc.width());
                  for(int dy = 0; dy < dc.height(); dy++) { //cyklus přes řádky
                        if(isCancelled()) return b;
```

Implementace metody *BitmapAsyncTask.onCancel* je jednoduchá (je analogická metodě *onPostExecute*)

```
@Override
protected void onCancelled(Bitmap result) {
      if(result != null)
            result.recycle();
      actualBitmap = null;
      backgroundThread = false;
}
```

Pokud je již bitmapa alokována, pak je uvolněna její paměť spravovaná systémem. Aktuální bitmapa není samozřejmě nastavena (stará je zahozena a nová není dokončena) a příznak běhu úlohy na pozadí je shozen (= nastaven na *false*).

Poslední změna se pak týká metody *MandelbrotView.generateNewBitmap* (tj. metody startující asynchronní úlohu):

```
private void generateNewBitmap() {
      if(backgroundThread) {
            task.cancel(false);
      }
```

```
        task = new BitmapAsyncTask();
        task.execute(new Transformation(dc, mc));
}
```

Hlavní formální změnou je změna lokální proměnné *task* na datový člen pohledu. Je to nutné, neboť podpora předčasného ukončení vyžaduje přístup k objektu úlohy i po jejím nastartování (objekt je vytvořen v jednom volání metody *generateBitmap* a ukončován v jiném). Parametr false u metody *BitmapAsyncTask.cancel* určuje, že nebude učiněn pokus nedobrovolně ukončit vlákno s úlohou.

Tím dospíváme do dalšího rovnovážného stavu naší aplikace. Bohužel předčasné ukončování se jen obtížně ladí (otočení nemusí mít vždy ten správný požadovaný efekt). Proto dodáme ještě jednu akci, která navíc citelně chybí: možnost opětovného přechodu na globální pohled.

Tuto akci zpřístupníme pomocí hlavního menu, jež je v Androidu dosažitelné pomocí stisku příslušného ovládacího tlačítka (ať již fyzického či softwarového).

Prvním krokem je doplnění (či změna) XML souboru definující obsah hlavního menu *res/menu/main.xml*. Výsledkem by mělo být menu s jednou položkou:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
   <item
       android:id="@+id/globalView"
       android:showAsAction="always"
       android:icon="@drawable/ic_action_refresh"
       android:title="@string/globalView"/>
</menu>
```

Atribut *id* stejně jako v případě deklarativních rozvržení obsahuje identifikátor, jenž bude dostupný i v kódu (pod jménem *R.id.globalView*).

Atribut *showAsAction* určuje, zda bude akce dostupná i pomocí akční lišty (na horní straně dipleje vedle názvu aplikace). To se v tomto případě hodí (je to hlavní akce), a proto tento způsob povolíme (hodnota *always* zajistí, že tam akce bude umístěna stále).

Akce je však v liště representována ikonou, kterou musíme dodat. Navíc by to měla být ikona, jejíž styl odpovídá doporučením uvedeným na stránkách *Android Iconography* (http://developer.android.com/design/style/iconography.html). Pokud navíc nejste zkušený návrhář ikon musíte se spokojit se standardní nabídkou akčně-lištových ikon, která je na těchto stránkách k dispozici. Po delším hledání jsem zvolil ikonku *refresh* (úplné jméno *ic_action_refresh*).

Poslední atribut (*title*) by měl obsahovat název položky menu. Protože se však zobrazuje navenek, měl by být zadán nepřímo. Zde je pouze odkaz na příslušný soubor deklarací (může existovat i více variant, nejen podle jazyka, je i např. podle rozlišení displej). V našem (jednoduchém) případě je soubor uložen jako *res/values/strings*. Změnte jeho obsah tak, aby obsahoval následující XML dokument (nezměněn zůstal název aplikace):

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">Mandelbrotka</string>
   <string name="globalView">Global View</string>
</resources>
```

Teď už zbývá jen jediný krok k dokončení aplikace — je potřeba doplnit metodu pro ošetření aktivace položky menu. Metoda (předefinovaná) se jmenuje *onOptionsItemSelected* a musí být umístěna ve třídě aktivity (*MainActivity* v souboru *MainActivity.java*).

Struktura metody je jednoduchá a neměnná. Jádrem je konstrukce *switch*, která zpracování větví podle identifikátoru zvolené položky menu (*switch* je zde akceptovatelný, neboť počet položek by měl být omezen na maximálně 5-8). V případě naší položky (*R.id.globalView*) je obsluha jednoduchá. Pohled je

znovu inicializován a je spuštěno generování nové bitmapy. To se však neděje přímým voláním příslušných metod (*init*, *generateBitmap*), které by měly zůstat neveřejné, ale použitím nově vytvořené vysokoúrovňové metody *MandlebrotView.globalView*:

```
public void globalView() {
            init();
            generateNewBitmap();

      }
```

Nyní můžete menu vyzkoušet, stejně jako přerušení generování na pozadí (položku menu zvolíte v okamžiku, kdy je vidět černá obrazovka s upozorněním).

OTÁZKY

1. Z jakého důvodu je v Androidu použit víceúrovňový životní cyklus aktivit?
2. Podle jakých hlavních kategorií se člení soubory zdrojů?
3. Co je návrhový vzor *Observer*? Jakou roli hraje v Javě a Androidu?
4. Jaký je rozdíl mezi (staticky) vnořenou třídou a vnitřní třídou?
5. Proč je přerušení asynchronního vlákna pomocí metody cancel označováno jako dobrovolné?
6. Co je to akční lišta?
7. Jak je aplikaci běžně signalizováno otočení displeje?

OTÁZKY K ZAMYŠLENÍ

1. Jaký je v Androidu vztah mezi aplikací a procesem?
2. Jaké metody musí definovat serializovatelný objekt (Parcelable)?
3. Jaké maximální přiblížení umožňuje použití typu *float*? (vycházejte ze šířky mantisy tohoto typu).

ÚKOLY

1. Přidejte do aplikace i možnost zmenšení. Použijte novou položku menu.
2. Zobrazte v obraze i základní informace o aktuálním výřezu. (důležité je aby byly vždy vidět)

4 Internetové služby a persitentní úložiště dat : Převodník měn

CÍLE KAPITOLY

Ukázková aplikace převodník měn ilustruje následující mechanismy programování v Androidu:

- spolupráce několika interních aktivit (zde omezená na dvě aktivity) pomocí tzv. úmyslů (*intent*)
- návrh aktivity za pomocí **rozvržení** využívajících **hotových pohledů**
- využití **služeb** (service) pro činnosti na pozadí (tj. neinteragující přímo s uživatelem)
- parsování XML dokumentu získaného z Internetu (což je část mechanismu využití tzv. **WWW služeb**)
- využití jednoduché **relační databáze** (o jediné tabulce) pro persistentní ukládání dat

Tyto dovednosti lze využít v širokém spektru aplikací, neboť tvoří skutečné jádro aplikací v Androidu.

4.1 Zadání

Hlavní funkce aplikace je zobrazení tabulky kursů získaných z XML dokumentu České národní banky na následující adrese:

`http://www.cnb.cz/cs/financni_trhy/devizovy_trh/kurzy_devizoveho_trhu/denni_kurz.xml`

Aplikace by navíc měla poskytovat jednoduchý kalkulátor pro převod mezi českou korunou a libovolnou měnou, jejíž kurs je v daném dokumentu uveden.

Speciálním požadavkem je uchování kursů v lokální databázi, tj. podpora off-line zobrazení a přepočtů. Tato funkce může být klíčová, neboť poplatky za roamingová data mohou být téměř astronomické.

4.2 Návrh

Na převodníku měn si ukážeme další typický rys aplikací v Androidu — ***modularitu***. Aplikace v Androidu se skládá z několika modulů, které spolu nesdílejí žádné společné objekty a komunikují spolu pomocí protokolu, který se podobá protokolu webových aplikací, neboť požadavek je zakódován do URL.

Navíc, zašleme-li v tomto protokolu požadavek (v názvosloví Androidu úmysl – *intent*), pak nemusí být předem znám modul, který požadavek splní. Úmysl totiž neadresuje přímo modul, ale definuje obecně službu, která má být provedena (například zavoláno číslo, získány dat o kontaktu, zobrazen soubor, přehrání multimédií, apod.) Je na systému (ve spolupráci s uživatelem), jaký modul danou službu obstará.

Zajímavým rysem této volné vazby mezi příjemcem a službou, je to, že modul nemusí ležet ve stejné aplikaci, ale může být poskytován jinou aplikací. Tím se de facto stírají hranice mezi aplikacemi a musí být zaveden zcela nový pojem – např. řetězec uživatelské interakce. Tento řetězec prochází mezi aplikacemi, využívaje jejich GUI moduly (aktivity), procesy na pozadí (služby) a zdroje dat ať již lokální

tak internetové (poskytovatelé obsahu). Navíc může reagovat pomocí tzv. příjemců veřejného vysílání (*broadcast receiver*) na události systému i ostatních aplikací.

Pokud již máte zkušenosti s praktickým využíváním Androidu, tak můžete být poněkud překvapeni, neboť hranice mezi aplikacemi nejsou ani ve světě Androidu zdaleka tak výrazně setřeny. Ve skutečnosti je výše uvedený model spíše ideálem, který je v některých oblastech relativně široce reflektován (např. propojení kontaktů, zpráv a kalendářů), jinde nepřesahuje možnosti desktopu (přehrávání multimédií) a v některých oblastech se vůbec neprojevuje (neexistují zde žádné jasně definované a standardizované služby, např. u textových editorů nebo skriptování).

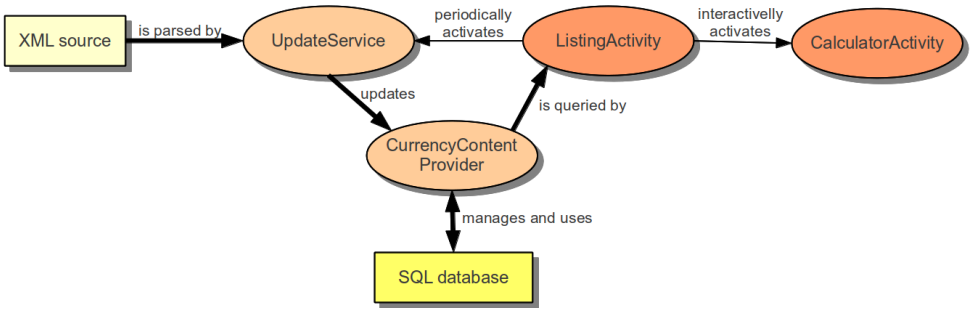
Podobná tendence se projevuje i v na úrovni kódu, kdy jsou úmysly používány i pro účely, které s původní sítí služeb souvisejí jen zcela okrajově.

1. úmysly se používají pro vzájemné volání (fixně určených) modulů v rámci jedné aplikace. Typickým využitím je přepínání mezi aktivitami (obrazovkami). Moduly však zůstávají relativně striktně odděleny, neboť nesdílejí žádné společné objekty.
2. úmysly se používají i pro implementaci distribuovaného objektového systému. Vazba mezi moduly, je již velmi těsná včetně zdánlivého přímého sdílení dat.

Tento modulární pohled na aplikaci zohledníme i v našem návrhu. Aplikace bude obsahovat čtyři moduly:

1. aktivitu zobrazující tabulku kursů (*ListingActivity*). Tato aktivita je hlavní aktivitou aplikace (tj. je zobrazena při prvním spuštění aplikace)
2. pomocná aktivita kalkulátorů kursů (zvolená měna vzhledem ke naší koruně české) — *CalculatorActivity*
3. služba pro aktualizaci kursů (*UpdateService*)
4. poskytovatel kursů (*CurrencyContentProvider*), který ostatním nabízí (abstraktně pojatou) databázi kursů. Pro interní uložení bude používat SQL databázi.

Vztah mezi jednotlivými moduly lze nejlépe ilustrovat obrázkem:



Všimněte si, že centrálním modulem z pohledu toku a zpracování dat je poskytovatel obsahu a *ListingActivity* z hlediska řízení. Jediným modulem, který je navržen tak, aby mohl být používán i z jiných aplikací je poskytovatel kursové databáze. Je to však pouze potenciál, neboť neposkytuje žádné standardizované a veřejně známé rozhraní.

4.3 Vytvoření resp. import projektu

Při vytvoření projektu zadejte název *MConverter* a hlavní aktivitu pojmenujte *ListingActivity*. Během úvodního průvodce není potřeba aktivovat žádná speciální nastavení.

Soubor manifestu by měl mít následující tvar:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="cz.ujep.ki.android.fiser"
   android:versionCode="1"
```

```
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="10"
        android:targetSdkVersion="18" />

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

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="cz.ujep.ki.android.fiser.ListingActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Jediným dodatečně přidaným prvkem (je možno ho přidat přímo v XML editoru) je nastavení požadovaných práv. Android z bezpečnostních důvodů vyžaduje, aby aplikace explicitně specifikovala práva k prostředkům, které mohou být zneužity. Uživatel pak při instalaci musí určit, zda tato práva poskytne či nikoliv (otázkou je, zda to není přecenění běžného uživatele). Naše aplikace potřebuje jen jediné explicitní právo — přístup k internetu (identifikátor *android.permission.INTERNET*).

Na rozdíl od prvního projektu nezůstane soubor manifestu beze změny a budeme ho průběžně rozšiřovat, neboť v něm musí být uvedena informace o každém dodatečném modulu (aktivitě, službě, apod.)

Kód projektu *MConverter* je již o poznání složitější a rozsáhlejší. Proto může být vhodným přístupem i import již hotového projektu a jeho prozkoumání a následné rozšiřování.

4.4 *ContentsProvider* — přístup k databázi

poskytovatel obsahu

Poskytovatel obsahu Poskytovatel obsahu je modul, který poskytuje ostatním modulům (resp. i ostatním aplikacím) data. Navenek se chová jako datová služba standardu REST, uvnitř používá datová úložiště (běžně je to databáze *SQLite*)

V naší aplikaci je poskytovatel obsahu centrálním modulem a proto začneme od něj. Prvním krokem je vytvoření třídy *CurrencyContentProvider* (kontextové menu projektu *New* | *Class*). Klíčovým nastavením je bázová třída, musí jí být *android.content.ContentProvider*. Dále si můžete nechat vygenerovat kostru abstraktních metod.

Druhým krokem je volba tzv. autority. V zásadě je to jedinečný identifikátor identifikujícího poskytovatele a jeho protokol, který nabízí (jaké data lze získat, jak je identifikovat a jaký budou mít formát). Autorita má stejný formát jako javovské balíčky, tj. měla by to být opačně zapsaná DNS doména (kterou bychom měli vlastnit, či být její správci). My použijeme *cz.ujep.ki.android.fiser.providers.currencies*.

Tuto autoritu musíme společně s poskytovatelem zaregistrovat v souboru manifestu:

```
<provider
      android:name="cz.ujep.ki.android.fiser.CurrencyContentProvider"
      android:authorities="cz.ujep.ki.android.fiser.providers.currencies" >
</provider>
```

Nyní již můžeme přistoupit k implementaci. Základním rozhodnutím je volba úložiště. I když Android nabízí několik persistentních úložišť, je tím nejpružnějším a nejpoužívanějším SQL databáze vytvořená a spravovaná pomocí knihovny *SQLite*. Pokud tuto knihovnu znáte, máte drobnou konkurenční výhodu, ale v zásadě postačuje znalost základní a tudíž téměř univerzální SQL syntaxe.

Pro přístup ke databázi se nepoužívá přímý SQL kód, ale pomocné třídy, které jej obalují do rozhraní vyšší úrovně. Bohužel tím získáme vyšší robustnost za cenu dosti nepřehledného a rozvláčného kódu. Navíc SQL není zcela ukryto a musíte jej proto alespoň trochu znát.

Další pomocnou třídou, která se v poskytovatelích obsahu běžně vyskytuje je *UriMatcher*. Jeho funkce je dána vnějším rozhraním provideru, které imituje API webových služby typu REST (*http://en.wikipedia.org/wiki/Representational_state_transfer*). Tyto služby používají URL pro identifikaci zdrojů data (tabulek) i jednotlivých položek a to i v několika úrovních adresace. Typický REST požadavek může vypadat např. takto:

http://geodb.test/staty (HTTP metoda GET)

= vrať celou tabulku států (v dohodnutém formátu)

http://geodb.test/staty/cz (HTTP metoda PUT)

= změň či přepiš položku tabulky států (*cz* je zde primární klíč)

Poskytovatel proto musí interně překládat podobně formovaná URL na symbolické identifikátory databází a případné klíče.

Definice poskytovatele začíná definicemi různých symbolických konstant (používání pojmenovaných konstant namísto literálů je v Androidu velmi rozšířené):

```
public class CurrencyContentProvider extends ContentProvider {
  private final static String DB_NAME = "currencies.db";
  private final static int DB_VERSION = 1;
  private final static String TABLE_NAME = "currencies";
```

Nejdříve jsou definovány konstanty, které určují interně používané identifikátory. Jméno databáze je jméno souboru, který je vytvořen v souborovém systému lokálního úložiště. Databáze jsou viditelné pouze v rámci dané aplikace (tj. jméno nemusí být globálně unikátní). Číslování verzí se týká změn struktury databáze (přidání tabulek, změna sloupců). Pokud dojde k takovéto změně je nutné toto číslo zvýšit (za chvíli uvidíme proč).

```
  public final static String AUTHORITY
           = "cz.ujep.ki.android.fiser.providers.currencies";
  public final static String CONTENT_URI
           = "content://" + AUTHORITY + "/" + TABLE_NAME;
  public final static String CONTENT_TYPE
           = "vnd.android.cursor.dir/vnd.cz.ujep.ki.android.fiser.currencies";
```

Další skupinu tvoří veřejné identifikátory. Je to za prvé identifikace autority (musí být stejná jako v manifestu!). Použití usnadňuje i URI, které bude použito z vnějšku pro přístup k poskytovateli (je to obdoba REST URL). Obsahuje schéma (*content:*), autoritu (vnější identifikátor poskytovatele) a cestu k tabulce (poskytujeme jen jednu tabulku). Poslední řetězcová konstanta, je MIME typ výsledků dotazu. Protože výsledkem není přenositelný obecný formát (XML nebo JSON), ale binární kursor závislý na struktuře naší tabulky, je nutno vytvořit nový unikátní identifikátor, jehož základním typem je *vnd.android.cursor.dir*.

```
  public final static String _ID = "_id";
  public final static String CODE = "code";
  public final static String NAME = "name";
  public final static String AMOUNT = "amount";
  public final static String RATE = "rate";
  public final static String COUNTRY = "country";
```

Další dávka konstant definuje jména sloupců databáze. Jména mohou být libovolná (samozřejmě SQL kompatibilní). Je však vhodné doplnit číselný primární klíč s fixním identifikátorem *_id* a to navzdory tomu, že tabulka již sloupec použitelný jako primární klíč obsahuje (jinak si výrazně zkomplikujete život, neboť mnohé třídy tento klíč implicitně předpokládají).

```
private static final int CURRENCY_MATCH = 1;
```

Poslední (opět soukromá) konstanta je využívána instancí třídy *UriMatcher* (bude representovat globální dotaz na tabulkou *currency*).

Kód dále pokračuje definicemi pomocných objektů a jejich inicializací, která se provádí v metodě *onCreate()*, jejíž funkce se podobá obdobně metodě v aktivitách. Je volána při vytvoření poskytovatele, což se děje v okamžiku, kdy je dotazován a přitom ještě (nebo už) neexistuje.

```
 private OpenHelper dbHelper;
      private UriMatcher matcher;

       @Override
       public boolean onCreate() {
             dbHelper = new DatabaseHelper(getContext());
               matcher = new UriMatcher(UriMatcher.NO_MATCH);
               matcher.addURI(AUTHORITY, TABLE_NAME, CURRENCY_MATCH);
             return true;
       }
```

OpenHelper je třída, která zapouzdřuje vytváření databáze (musí být vytvořena pro každou databázi zvlášť). V konstruktoru očekává tzv. kontext, který (zjednodušeně) určuje společný kontext aplikace nebo její části. Ve většině případů jej získáme voláním metody *getContext* (zde je to metoda zděděná ze třídy *ContentProvider*), nebo je přímo identická s modulem (např. u hlavní aktivity). *URIMatcher* je inicializován mapováním, které mapuje URI s danou autoritou a cestou na symbolickou konstantu (číslo) CURRENCY_MATCH.

Dále je nutno definovat třídu *DataBaseHelper*, což je specializace třídy *SQLiteOpenHelper* pro naši databázi. Pro jednoduchost jí vytvoříme jako (neveřejnou) statickou vnořenou třídu:

```
private static class OpenHelper extends SQLiteOpenHelper {
             OpenHelper(Context context) {
          super(context, DB_NAME, null, DB_VERSION);
       }

             @Override
             public void onCreate(SQLiteDatabase db) {
                   String command =
                   "CREATE TABLE " + TABLE_NAME + " (" +
                   _ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                   CODE + " VARCHAR(3), " +
                   NAME + " VARCHAR(32), " +
                   COUNTRY + " VARCHAR(32), " +
                   AMOUNT + " REAL, " +
                   RATE + " REAL " + ");";
                   db.execSQL(command);
             }

             @Override
             public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
                   db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
                   onCreate(db);
             }
```

```
}
```

Funkce je jasná. Předefinovaná metoda *onCreate* vytváří novou tabulku (obecně všechny tabulky databáze). K tomu využívá SQL příkaz CREATE TABLE, který je zkonstruován za použití symbolických konstant. Při určování domén nemusíte být příliš striktní, neboť SQLite poskytuje jen několik málo typů, které jsou navíc často zcela zaměnitelné (např. VARCHAR(n) je totéž co CHAR(N) a TEXT).

Podobně je konstruována metoda, která se volá v případě, kdy se zvýší číslo verze předané konstruktoru bázové třídy. Použité řešení není sofistikované, ale pro jednoduché databáze akceptovatelné. Stará databáze je smazána a nová je vytvořena metodou *onCreate*.

Následující sekce pokračuje v předefinování metod třídy poskytovatele obsahu (*CurrencyContentProvider*). Tato část využívá opakovaně několik idiomů a v zásadě se nemění (jen se komplikuje při použití databáze s více tabulkami).

```
@Override
public String getType(Uri uri) {
      switch(matcher.match(uri)) {
      case CURRENCY_MATCH:
            return CONTENT_TYPE;
      default:
            throw new IllegalArgumentException("Unknown URI " + uri);
      }
}
```

Metoda mapující dotazovací URI na MIME type získaných položek. Zde vrátíme jednoznačně definovaný podtyp typu *vnd.android.cursor.dir* (máme pro něj již symbolickou konstantu). Hlavně se však všimněte, jak je URI požadavku mapováno na symbolickou konstantu (CURRENCY_MATCH) pomocí instance *UriMatcher*u . Konstanta je následně mapována na obslužný kód pomocí konstrukce *switch*.

```
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
SQLiteDatabase db = dbHelper.getWritableDatabase();
int count;
switch (matcher.match(uri)) {
   case CURRENCY_MATCH:
      count = db.delete(TABLE_NAME, selection, selectionArgs);
      break;
   default:
      throw new IllegalArgumentException("Unknown URI " + uri);
}
getContext().getContentResolver().notifyChange(uri, null);
return count;
}
```

Funkce provádějící požadavek na výmaz. Struktura všech výkonných metod je podobná. Za prvé získáme objekt, který zapouzdřuje vstupy a výstupy databáze. Výmaz databázi mění a proto se pokusíme získat objekt, který umožňuje i zápis do databáze (*getWritableDatabase*). Potom zkontrolujeme dotazovací URI. Odpovídá-li vzoru (tj. matcher URI rozezná a vrátí symbol odpovídající naší jediné tabulce), pak provedení delegujeme na objekt zapouzdřující databázi. Metoda vrací počet pozměněných řádek, které si uložíme a nakonec je vrátíme jako návratovou hodnotu naší funkce. Před tím však musíme do systému notifikovat, co se změnilo, aby mohli být informováni objekty, které se zaregistrovali u pozorovatele (návrhový vzor *observer*, neboť poskytovatel slouží jako model v architektuře MVC tj. *Model-View-Controller*).

```
@Override
public Uri insert(Uri uri, ContentValues values) {
      SQLiteDatabase db = dbHelper.getWritableDatabase();
```

```
long id;
switch (matcher.match(uri)) {
   case CURRENCY_MATCH:
      id = db.insert(TABLE_NAME, CODE, values);
      break;
   default:
      throw new IllegalArgumentException("Unknown URI " + uri);
}

if (id > 0) {
      Uri itemUri = ContentUris.withAppendedId(uri, id);
      getContext().getContentResolver().notifyChange(uri, null);
      return itemUri;
}

throw new SQLException("Failed to insert row into " + uri);
}
```

Implementace příkazu INSERT má stejnou strukturu jako výše. Jsou zde jen dvě změny — metoda notifikuje a vrací URI rozšířené o primární klíč nového záznamu. Nově přidávaný řádek je předán jako objekt třídy *ContentValues*. V zásadě je to slovník mapující názvy sloupců na hodnoty (ještě se s ním setkáme).

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {

SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

      switch (matcher.match(uri)) {
   case CURRENCY_MATCH:
      qb.setTables(TABLE_NAME);
      break;
   default:
      throw new IllegalArgumentException("Unknown URI " + uri);
}

SQLiteDatabase db = dbHelper.getReadableDatabase();
Cursor c = qb.query(db, projection, selection, selectionArgs,
      null, null, sortOrder);

c.setNotificationUri(getContext().getContentResolver(), uri);
return c;
}
```

Metoda realizující příkaz SELECT. Tento příkaz je o něco složitější, takže je nutné nejdříve jej sestavit z jednotlivých částí (jsou předány jako parametry a odpovídají jednotlivým částem příkazu např. *projection* je zobrazení definované bezprostředně za *SELECT*, *selection* a *selectionArgs* popisují filtr uvedený v části *WHERE*). Metoda vrací tzv. kursor, což je de facto iterátor přes virtuální tabulku získanou dotazem (ke kursoru se ještě vrátíme). Všimněte si také, že databáze je jen pro čtení.

```
@Override
public int update(Uri uri, ContentValues values, String selection,
                  String[] selectionArgs) {
      SQLiteDatabase db = dbHelper.getWritableDatabase();
int count;
```

```
        switch (matcher.match(uri)) {
            case CURRENCY_MATCH:
                count = db.update(TABLE_NAME, values, selection, selectionArgs);
                break;

            default:
                throw new IllegalArgumentException("Unknown URI " + uri);
        }

        getContext().getContentResolver().notifyChange(uri, null);
        return count;
        }
}
```

Zde již není nic nového, metoda update je obdobou metody *delete*, kde nová hodnota je předána jako u metody *insert*.

4.5 *UpdateService* — čtení dat na pozadí

Služby jsou moduly, které vykonávají činnosti, které nejsou přímo svázány s grafickým uživatelským prostředím. Běžně se označují jako činnosti na pozadí, což však vede k určitému zmatení mezi vlákny a službami.

služba (service)

- je modul tj. je oddělen od zbytku aplikace (především od aktivit) a může být tudíž aktivován i z jiných aplikací. S trochou zjednodušení je to vlastně aktivita bez možnosti využití displeje a s trochu jiným režimem životního cyklu.
- je standardně vykonávána ve stejném vlákně jako aktivita. Tj. činnost, kterou provádí by měla být buď krátká (maximálně desetiny sekundy), nebo asynchronní (spuštěná skutečně na pozadí jako je např. přehrávání multimédií). Pokud není splněna ani jedna možnost, je možno využít explicitně vytvořeného vlákna (stejně jako u aktivit).
- služba může být buď jednorázová (po zavolání se téměř bezprostředně ukončí), nebo trvalá. Pokud je trvalá musí zajistit svou persistenci (stejně jako jako aktivita), či svůj trvalý nepřerušený běh (v tomto případě se musí zaregistrovat do lišty událostí)

vlákno

- není modulem, ale pouhou součástí určitého modulu, pro něhož vykonává asynchronní činnost (běžící paralelně s hlavním vláknem). Hlavní vlákno tak může neprodleně reagovat na požadavky GUI nebo systému. Vlákno může se zbytkem modulu sdílet data tj. objekty (jen je nutno zajistit synchronizaci přístupu ke sdíleným objektům).

V naší aplikaci služba jen přečte a rozparsuje XML dokument a údaje z něho vloží do výše implementovaného poskytovatele obsahu. Je to tedy jednorázová služba. Musí však být implementována pomocí vláken, neboť přenos a parsování dat z internetu může trvat poněkud déle (i vteřiny). Android navíc nenabízí asynchronní verzi XML parseru (jako je tomu v .NET).

Avšak ani v tomto případě nepoužijeme vlákno přímo, neboť i zde existuje pohodlnější řešení na vyšší úrovni. Pokud nám stačí scénář, kdy po obdržení požadavku (intentu) stačí vytvořit vlákno a předat mu požadavek ke zpracování (a načekat na jeho dokončení a výsledky); pak lze namísto třídy *android.app.Service* použít specializovanější bázovou třídu *android.app.IntentService*).

Hlavní metodou této třídy je *onHandleIntent*. Nejdříve si však připravíme a inicializujeme pomocné datové členy:

```
public class UpdateService extends IntentService {
   private static final String uri =
         "http://www.cnb.cz/cs/financni_trhy/devizovy_trh/kurzy_devizoveho_trhu/denni_kurz.xml";
```

```
Handler guiHandler;

public UpdateService() {
        super("UpdateService");
    }

    @Override
    public void onCreate() {
        guiHandler = new Handler();
        super.onCreate();
    }
```

Řetězcová konstanta obsahuje URL stránky, z níž budeme načítat data (stejně jako všude v Androidu, i zde platí: nepoužívat literály, ale symbolické konstanty). Zajímavější je však druhý datový člen (*guiHandler*), který bude po inicializaci obsahovat tzv. *handler*. Handler uchovává informace o vlákně a jeho událostní smyčce a umožňuje ostatním vláknům tuto smyčku používat (tj. volat kód, který se provede ve smyčce jiného vlákna). V našem případě bude tímto vláknem hlavní (GUI) vlákno. Proto musíme zajistit, aby byl objekt handleru vytvořen v hlavním GUI vlákně (handler se vždy vztahuje ke vláknu, v němž byl vytvořen).

I když by bylo možno handler inicializovat již při definici nebo v konstruktoru objektu služby (konstruktor i inicializátor se vykoná v hlavním vlákně), je tato inicializace přesunuta až do metody *onCreate*, která se volá při každém vytvoření objektu služby (stejně jako u aktivity nebo poskytovatele obsahu). I tato metoda se volá v hlavním vlákně (a zcela v souladu s Androidím přístupem, je vše vykonáno, až při poslední možné příležitosti).

Poté služba se služba zablokuje a čeká do okamžiku, kdy jiný modul vyjádří úmysl ji využít (tj. aktivuje službu a předá ji objekt třídy *Intent*). V tomto okamžiku se vytvoří nové vlákno, které začne vykonávat metodu *onHandleIntent* (aktivující *intent* je předán jako parametr této metody).

```
@Override
    protected void onHandleIntent(Intent intent) {
        URL url;
        InputStream input;

        try {
            url = new URL(uri);
            input = url.openConnection().getInputStream();
        } catch (MalformedURLException e) {
            Log.e("Update service", "Malformed URL");
            return;
        } catch (IOException e) {
            //TODO: toast for users
            Log.e("Update service", "IO Exception");
            return;
        }
```

Na začátku metody se pokusíme otevřít proud, ze kterého budeme číst XML dokument. Využijeme k tomu instanci třídy URL. To se přirozeně nemusí povést a proto musíme ošetřit výjimky (na rozdíl od C# je v Javě nutné výjimky, alespoň formálně ošetřit). Prozatím je jediným ošetřením výpis chybového hlášení do logu (pomocí metody Log.*e*, kde *e* symbolizuje *error*). To je adekvátní v případě výjimky třídy *MalformedURLException*, neboť ta může vzniknout jen programátorovou chybou (URL je uvedeno přímo v programu), je však zcela nedostatečná u chyby třídy *IOException*, kde uživatel nebude nijak informován (prostě se jen nic neprovede a nevypíše se zdravice (*toast*) se statistikou, viz dále).

Jádrem metody je přečtení všech elementů XML dokumentů a uložení získaných dat o měně do poskytovatele obsahu. I když je operace algoritmicky triviální, je její zápis poněkud rozvleklý:

```
XmlPullParser parser = Xml.newPullParser();
ContentResolver resolver = getContentResolver();
int updated = 0;
int inserted = 0;
```

Nejdříve si musíme připravit zdroj, což je tzv. *pull* (vytahovací) XML parser. Tento parser čte jednotlivé XML konstrukce (tagy, text mezi tagy), a to na požádání. Tento přístup je běžný v .NET nebyl však podporován standardní Javou (ta používá SAX server, který sice pracuje se stejnými konstrukcemi, sám však vyvolává jednotlivé metody, tj. typu *push* (podstrkovací)).

Objekt třídy *Resolver* nám umožňuje získat přístup k manažeru poskytovatelů (konkrétní poskytovatel je určen pomocí URI). Číselné proměnné jsou jednoduché čítače (abychom mohli poskytnout na závěr nějakou statistiku).

```
try {
      parser.setInput(input, null);
      int eventType = parser.getEventType(); //načtení první konstrukce

      while(eventType != XmlPullParser.END_DOCUMENT) { //dokud není konec dokumentu
            if(eventType == XmlPullParser.START_TAG) && //pro každý element "řádek"
                              parser.getName().equals("radek")) {
                  //vytvoříme kontejner pro db. řád
                  ContentValues val = new ContentValues();
                  //a vložíme do něj jednotlivé sloupce
                  putAttrString(parser, "kod", val, CurrencyContentProvider.CODE);
                  putAttrString(parser, "mena", val, CurrencyContentProvider.NAME);
                  putAttrInt(parser, "mnozstvi", val, CurrencyContentProvider.AMOUNT);
                  putAttrDouble(parser, "kurz", val, CurrencyContentProvider.RATE);
                  putAttrString(parser, "zeme", val, CurrencyContentProvider.COUNTRY);
                  //pokusíme se ho updatovat
                  Uri contentUri = Uri.parse(CurrencyContentProvider.CONTENT_URI);
                  int cols = resolver.update(
                        contentUri, val, CurrencyContentProvider.CODE + "=?",
                        new String[]{val.getAsString(CurrencyContentProvider.CODE)});
                  if(cols == 0) { //nepodaří-li se to, tak jej vložíme (insert)
                        resolver.insert(contentUri, val);
                        inserted++;
                  }
                  else {
                        updated++;
                  }
              }
            }
      eventType = parser.next(); //přejdeme na další XML konstrukci
  } //konec cyklu přes prvky XML
} catch (XmlPullParserException e) {
      Log.e("Update service", "Malformed XML");
            return;
} catch (IOException e) {
            Log.e("Update service", "Malformed XML");
            return;
}
```

Základem je cyklus přes všechny elementy s názvem řádek (přesněji přes jejich počáteční tagy). U každého tagu získáme jeho atributy a vložíme je do kontejneru řádku (= instance třídy *ContentValues*). Pro to využijeme pomocných metod *putAttr…* (jejich definici uvidíme za chvíli). Metodám předáváme

identifikaci zdroje (parser, který odkazuje na počáteční tag a jméno atributu) a identifikací cíle (kontejner a jméno sloupce). Metody provedou navíc přetypování a kontrolu typu (proto existují tři odlišné metody).

Po naplnění kontejneru řádku se jej pokusíme nejdříve použít pro aktualizaci, neboť to je běžnější (řádek se povětšinou vkládá jen jednou při prvotním spuštění, poté se jen aktualizuje). Všimněte se jak je konstruován požadavek na databázi (ten volá metodu *update* poskytovatele). Klíčem je konstrukce části WHERE SQL příkazu UPDATE. Jeden parametr určuje levou stranu porovnávání, druhý (v podobě identifikátor slovníku) identifikátor měny získaný z kontejneru řádku (tj. vygeneruje např. SQL ve tvaru *UPDATE … WHERE code = "USD"*). Pokud se aktualizace řádku nepodaří (řádek ještě neexistuje a metoda vrací 0 pozměněných řádků), je použito vložení (to je jednodušší, neboť není nutno inicializovat řádek). V obou případech se inkrementují příslušné čítače.

Po ukončení celkové aktualizace vypíšeme uživateli krátkou zprávu s počtem přidaných a aktualizovaných řádků. Pro to využijeme tzv. zdravici (angl. *toast*), což je malý obdélník s textem zobrazovaný uprostřed displeje. Jinou možnost prakticky prakticky nemáme, neboť služba nemá přístup k displeji a může dokonce nastat situaci, že v okamžiku výpisu upozornění používá displej jiná zcela nesouvisející aplikace (druhou, složitější, možností je výpis do horní lišty událostí). Výpis trochu komplikuje skutečnost, že se nenacházíme v hlavním GUI vlákně. Pokud bychom zdravici vypsali z tohoto vlákna, pak by se sice zobrazila, ale nikdy by nezmizela (nezbude nic jiného než restartování systému). Naštěstí máme handler vytvořený v hlavním okně takže stačí, použít jeho metodu *post*, který kód pro výpis vloží do událostní smyčky hlavního vlákna.

Kód samotný je vložen jako předefinovaná metoda instance anonymní třídy implementující rozhraní *Runnable*. Tento syntaktický prostředek typický pro Javu vypadá jako volání konstruktoru na rozhraní, které je však doplněno tělem, v němž jsou definovány všechny metody rozhraní (lze to použít i s abstraktní třídou). Překladač Javy vytvoří podle definice novou bezejmennou třídu, jejíž objekt poté zkonstruuje. Je to poněkud rozvleklejší obdoba anonymních delegátů a lambda funkcí jazyka C# (naštěstí v *IntelliJ* stačí napsat jen *new* a jméno rozhraní a editor doplní prázdné definici potřebných metod).

```
final String toastText = "Done. Inserted: " + inserted + ". Updated: " + updated;

guiHandler.post(new Runnable() { //toast from non-GUI thread
      @Override
      public void run() {
        Toast toast = Toast.makeText(getBaseContext(), toastText, Toast.LENGTH_SHORT);
        toast.show();
      }}); //konec volání metody post
} //konec definice metody onHandleIntent
```

Z implementace služby už chybí jen definice pomocných metod pro čtení atributů, přetypování a uložení do kontejneru řádků (instance *ContentValues*).

```
private static void putAttrString(XmlPullParser parser, String attrName,
                                  ContentValues cv, String colName) {
      cv.put(colName, parser.getAttributeValue("", attrName));
}

private static void putAttrInt(XmlPullParser parser, String attrName,
                                     ContentValues cv, String colName) {
      cv.put(colName, Integer.parseInt(parser.getAttributeValue("", attrName)));
}

private static void putAttrDouble(XmlPullParser parser, String attrName,
                                     ContentValues cv, String colName) {
      String svalue = parser.getAttributeValue("", attrName);
      if (svalue.contains(",")) { //the source uses "," as decimal point
            svalue = svalue.replace(',', '.');
```

```
        }
        cv.put(colName, Double.parseDouble(svalue));
}
} //konec třídy update service
```

Třídu služby nakonec zaregistrujeme v manifestu.

```
<service android:name="cz.ujep.ki.android.fiser.UpdateService" ></service>
```

4.6 Hlavní aktivita — seznamový pohled

Kód hlavní aktivity není příliš rozsáhlý využívá však velkého množství idiomů, které již částečně známe avšak s mnohými se však ještě musíme seznámit:

```
public class ListingActivity extends Activity implements OnItemClickListener {
        private Cursor mc;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
              super.onCreate(savedInstanceState);
              setContentView(R.layout.activity_main);
              mc = getDataFromProvider();
            SimpleCursorAdapter adapter = new SimpleCursorAdapter(
                this,
                R.layout.item,
                mc,
                new String[]{ CurrencyContentProvider.CODE,
                                CurrencyContentProvider.NAME,
                               CurrencyContentProvider.AMOUNT,
                               CurrencyContentProvider.RATE},
                new int[]{R.id.itemCode,R.id.itemName,R.id.itemAmount, R.id.itemRate});
        ListView v = (ListView)findViewById(R.id.currencyView);
        v.setAdapter(adapter);
        v.setOnItemClickListener(this);
        }
```

Začátek metody *onCreate* je zcela klasický — nastavení rozvržení. Toto rozvržení je specifikováno pomocí souboru *res/layout/activity_main.xml* a je zcela triviální (celý displej pod lištami je vyplněn pohledem třídy *ListView*)

```
<RelativeLayout 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=".ListingActivity" >

   <ListView
      android:id="@+id/currencyView"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_centerHorizontal="true"
      android:layout_centerVertical="true" >
   </ListView>
</RelativeLayout>
```

Druhým krokem je vyplnění seznamu daty od poskytovatele. Nejdříve získáme kursor (použita je pomocná metoda definovaná později, získání kursoru není totiž zcela triviální). Hlavním krokem je však vytvoření adaptéru, který sváže kursor s vizuálním návrhem jednotlivých řádků. Adaptéry jsou dalším klíčovým návrhovým vzorem Androidu a spolu se zobrazovačem (zde je to *ListView*) realizují část ***V**iew* a ***C**ontroller* architektury MVC. I když je použitý adaptér označen jako jednoduchý (instance *SimpleCursorAdapter*), tak jeho konstruktor vyžaduje relativně dost parametrů:

1. kontext (zde je kontextem samotný objekt aktivity)
2. odkaz na deklarativní definici rozvržení řádků tabulky (viz dále)
3. kursor, který bude zobrazen
4. projekce tj. pole jmen těch sloupců kursoru, která budou zobrazeny
5. pole jmen identifikátorů, použitých v návrhu rozvržení pro podpohledy (typicky textová návěští). Podpohled, jehož identifikátor je uveden na n-tém místě, zobrazí text z n-tého sloupce kursoru (v pořadí uvedením v předchozím seznamu).

Zde použité rozvržení je deklarováno v souboru *res/layout/item.xml* a obsahuje čtyři textová návěští v jednom horizontálně uspořádaném řádku (text návěští se nikde nepoužije).

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="horizontal" > <!-- horizontální uspořádání -->

   <TextView
      android:id="@+id/itemName"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="mena"
      android:textStyle="bold" />

   <TextView
      android:id="@+id/itemCode"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="kod"
      android:textColor="#AA0000" /> <!--text kódu měny je červený-->

   <TextView
      android:id="@+id/itemAmount"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="mnozstvi" />

   <TextView
      android:id="@+id/itemRate"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="kurz" />
</LinearLayout>
```

Po vytvoření adaptéru, stačí adaptér zaregistrovat u seznamového pohledu (*setAdapter*). Poslední řádek určuje, že události volby jednotlivé položky dotekem bude obsluhovat naše aktivita (tím, že zavolá

aktivitu kalkulátoru).

Pomocná metoda, získávající kursor je po logické stránce triviální, z jednotlivých částí je zkonstruován příkaz *select* a ten je aplikován na poskytovateli, jež je učen pomocí URI. Je provedena projekce, která vrací všechny sloupce (včetně umělého primárního klíče) a položky jsou setříděny vzestupně podle kódu měny.

```
private Cursor getDataFromProvider() {
      // run query
      Uri uri = Uri.parse(CurrencyContentProvider.CONTENT_URI);
      String[] projection = new String[] {
            CurrencyContentProvider._ID,
            CurrencyContentProvider.CODE,
            CurrencyContentProvider.NAME,
            CurrencyContentProvider.AMOUNT,
            CurrencyContentProvider.RATE,
            CurrencyContentProvider.COUNTRY
      };
      String selection = null;
      String[] selectionArgs = null;
      String sortOrder = CurrencyContentProvider.CODE + " ASC";

      return managedQuery(uri, projection, selection, selectionArgs, sortOrder);
      }
```

Jednotlivé části dotazu SELECT jsou použity v metodě *Activity.managedQuery*, který vrací tzv. řízený dotaz, to znamená, že životní cyklus dotazu je řízen životním cyklem aktivity (jinak řečeno dotaz bude automaticky uvolněn v okamžiku, kdy už nebude potřeba). Tento přístup není zcela košer (nemůžeme přesněji řídit kursor, ani to není obecný přístup), v nových verzích je dokonce označen za zastaralý. Je však velmi jednoduchý a použitelný i ve starších verzích (do API 10 včetně).

Další část kódu hlavní aktivity ošetřuje menu, které obsahuje jen jednu položku — žádost o updatování.

```
      @Override
      public boolean onCreateOptionsMenu(Menu menu) {
            getMenuInflater().inflate(R.menu.activity_main, menu);
            return true;
      }

      @Override
      public boolean onMenuItemSelected(int featureId, MenuItem item) {
            switch(item.getItemId()) {
            case R.id.menu_update:
                  Intent intent = new Intent(this, UpdateService.class);
                  startService(intent);
            }
            return super.onMenuItemSelected(featureId, item);
```

Kód by měl být zřejmý, po vyvolání menu je vytvořen požadavek (*intent*) na službu, která je identifikována jménem třídy, jejíž instance službu vykoná (je to tedy fixní vazba na konkrétní implementaci služby). Zápis *UpdateService.class* vrací objekt, který z hlediska reflexe representuje danou třídu. Intent je následně použit jako parametr metody *startService*.

Poslední metodou třídy hlavní aktivity (*ListingsActivity*) je obsluha volby jedné z položek seznamu. Tato metoda musí být implementována, neboť třída přislíbila implementaci rozhraní *OnItemClickListener* a její instance samu sebe registrovala jako obsluhu (*listener*).

```
      @Override
      public void onItemClick(AdapterView<?> parent, View view, int position, long id)
```

```
        {
        mc.moveToPosition(position);
        Bundle bundle = new Bundle();
        bundle.putString("CODE",
                    mc.getString(mc.getColumnIndex(CurrencyContentProvider.CODE)));

        bundle.putString("NAME",
                    mc.getString(mc.getColumnIndex(CurrencyContentProvider.NAME)));

        bundle.putDouble("AMOUNT",
                    mc.getDouble(mc.getColumnIndex(CurrencyContentProvider.AMOUNT)));

        bundle.putDouble("RATE",
                    mc.getDouble(mc.getColumnIndex(CurrencyContentProvider.RATE)));

        bundle.putString("COUNTRY",
                    mc.getString(mc.getColumnIndex(CurrencyContentProvider.COUNTRY)));

        Intent intent = new Intent(this, Calculator.class);
        intent.putExtras(bundle);
        startActivity(intent);
}
```

Po vyvolání události získá obslužná rutina relativně velké množství údajů. My však používáme jen index zvolené položky (= řádku s údaji o jedné měně). V první části kódu posuneme ukazatel kurzoru na zvolenou položku. Pak se kursoru dotazujeme na jednotlivé sloupce (indexem je číslo sloupce, které získáme z jeho jména voláním metody *Cursor.getColumnIndex*, kód by neměl být samozřejmě závislý na pořadí sloupců v kursoru) a jejich hodnoty vkládáme do objektu třídy *Bundle*. *Bundle* je jak již víme jednoduchý slovník, který mapuje hodnoty na (serializované) objekty.

Po převzetí všech údajů o měně nastartujeme novou aktivitu. Parametrem je opět úmysl (intent), ke kterému je však tentokrát přibalen objekt s dodatečnými parametry — naplněná instance třídy *Bundle*. Tento mechanismus lze obecně používat k předání dodatečných parametrů nově startované aktivitě či službě.

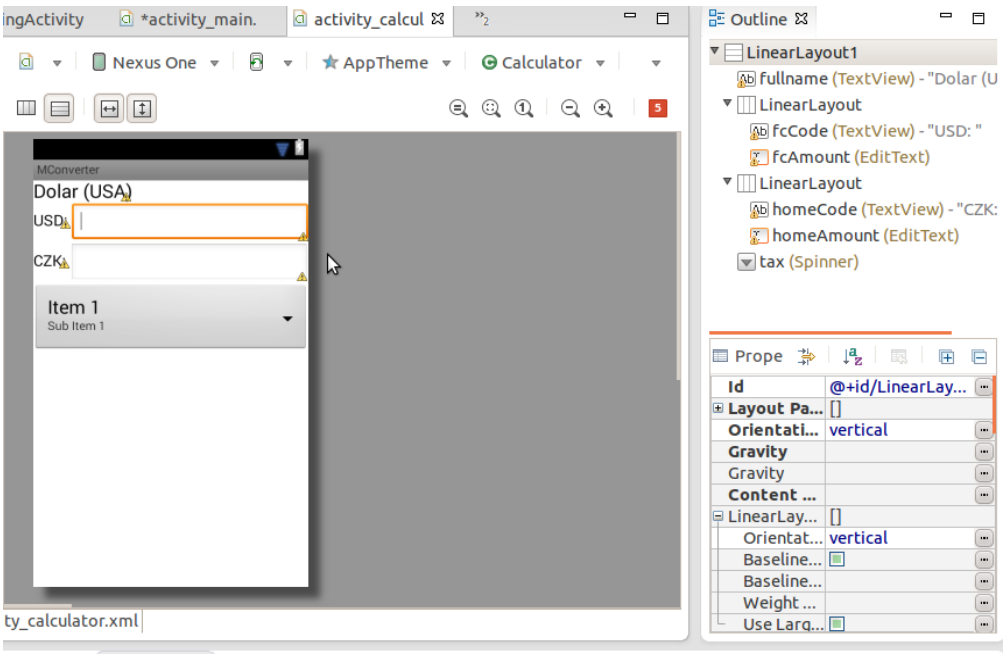
Tím se dostáváme k poslední třídě — aktivitě *CalculatorActivity*.

4.7 *CalculatorActivity* — aktivní formulář

CalculatorActivity je aktivita, která se podobá běžným formulářům či dialogovým oknům známým z desktopových graficky orientovaných rozhraní.

Formulář by měl být aktivní (proto kalkulátor), tj. změna obnosu v jedné měně se hned projeví v obnosu měny druhé (tj. křížem). Navíc budeme podporovat přibližné nastavení transakčních poplaků, které se nejčastěji projevují v nastavení méně výhodného kursu pro prodej či koupi (střední kurs ČNB nikde nedostanete).

Protože tato aktivita obsahuje více pohledů, je její návrh v návrháři poněkud složitější. Můj návrh má vzhled zobrazený na následujícím obrázku (XML zápis je již příliš dlouhý). Jedná se však o vnoření vertikální lineární rozložení obsahující dvě rozložení horizontální (s popiskem a a editačním řádkem) a rozbalovacího seznamu (*spinner*, výběr z několika variant po rozbalení). Všimněte si především identifikátorů u jednotlivých pohledů (vpravo v osnově).



Kód aktivity začíná klíčovou metodou *onCreate*, která je však již o něco delší (přestože používá několik pomocných metod):

```
public class Calculator extends Activity implements OnItemSelectedListener {
	EditText fcAmount;
	EditText homeAmount;
	TextView fcCode;
	Spinner tax;
	TextView fullName;
	private double rate;
	private double taxrate;
```

Datové členy slouží primárně k uchování odkazů na jednotlivé dílčí pohledy (jejich jméno je shodné se identifikátorem pohledu). Proměnná obsahuje *rate* obsahuje střední kurs, proměnná *texrate* kurs odvozený ze středního kursu (poplatek za výměnu vyjádřený jako procento částky).

```
Override
	protected void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);
	setContentView(R.layout.activity_calculator);
	fcAmount = (EditText)findViewById(R.id.fcAmount);
	homeAmount = (EditText)findViewById(R.id.homeAmount);
	fcCode = (TextView)findViewById(R.id.fcCode);
	tax = (Spinner)findViewById(R.id.tax);
```

Zatím je to zcela klasické nastavení rozvržení a získání odkazů na jednotlivé pohledy.

```
	Intent intent = getIntent();
	Bundle b = intent.getExtras();
```

Poté se získá úmysl, který aktivitu spustil a je získán přiložený balík parametrů.

```
	fullName.setText(b.getString("NAME") + " (" + b.getString("COUNTRY") + ")");
	fcCode.setText(b.getString("CODE") + ": ");
	fcAmount.setText("1");
```

Zde jsou nastaveny základní informační pohledy. Jméno měny spolu se státem, kód měny (druhou je vždy koruna) a počet jednotek měny (na začátku je to vždy 1 jednotka)

```
ArrayAdapter<CharSequence> adapter =
                  ArrayAdapter.createFromResource(this,
                  R.array.taxes,
                  android.R.layout.simple_spinner_item);
      adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
tax.setAdapter(adapter);
tax.setOnItemSelectedListener(this);
```

Nastavení rozvinovacího seznamu (*spinneru*). Seznam textových hodnot je získán z XML souboru *res/-value/string.xml*, a který má následující tvar (zkráceno).

```
<string-array name="taxes">
      <item>0%</item>
      <item>2%</item>
      ...
</string-array>
```

Stejně jako u *ListView* , je nutno nejdříve vytvořit adaptér spojující seznam (odkazovaný identifikátorem zdroje) a rozložení jednotlivého řádku. V tomto případě je však řádek jednoduchý, a tak lze použít vestavěný zdroj (layout), který je odkazován identifikátorem *android.R.layout.simple_spinner_-item* (identifikátory vestavěných zdrojů začínají prefixem *android.R* a nikoliv jen *R*). Podobně je použit i standardní rozvinovací tlačítko (zdroj *android.R.layout.simple_spinner_dropdown_item*). Následně je adaptér registrován.

Aktivita se také registruje jako příjemce událostí volby položky při použití rozvinovacího seznamu (aby mohla reagovat na změnu sazby poplatku). Musí proto implementovat rozhraní *OnItemSelectedListener.*

```
rate = b.getDouble("RATE") / b.getDouble("AMOUNT");
taxrate = rate * (1.0 + parseTax((String)(tax.getSelectedItem())));
```

Následuje výpočet obou kursů. Procentuální hodnota poplatku je přečtena z rozvinovacího seznamu, což vyžaduje trochu pomocného kódu, který je umístěn v metodě *parseTax* (viz dále).

```
refresh(fcAmount.getText().toString(), homeAmount, taxrate);
```

Toto je volání klíčové pomocné metody, která zde inicializuje hodnotu editační řádky obnosu domovské měny (CZK) podle obnosu měny zahraniční (první parametr) za použití předaného kursu (s poplatkem).

```
fcAmount.addTextChangedListener(new TextWatcher() {
      @Override
      public void onTextChanged(CharSequence s, int start, int before, int count) {}

      @Override
      public void beforeTextChanged(CharSequence s, int start, int count, int after) {}

      @Override
      public void afterTextChanged(Editable s) {
            if(fcAmount.hasFocus()) {
                  refresh(s.toString(), homeAmount, taxrate);
            }
      }});
```

Následuje definice obsluhy změn editačního pole cizí změny. Ta je tak jednoduchá, že je použita anonymní implementace rozhraní *TextWatcher*. Implementace musí definovat tři metody, ale jen jedna není prázdná. Metoda *afterTextChanged* je voláno po provedení editace a její funkce je zřejmá. Musí znovu vypočíst obnos domácí měny, aby odpovídala změněnému vstupu. Všimněte si testu, zda má hlídaný vstupní řádek zaměření (*focus*). Pokud tomu tak není, tak byl změněn programově (opačnou rutinou při změně obnosu v domácí měně), což by vedlo k nekonečné rekurzi (změna obnosu v CZK mění obnos v zahraniční měně, ta zase mění obnos v domácí, atd.).

```
homeAmount.addTextChangedListener(new TextWatcher() {
      @Override
      public void onTextChanged(CharSequence s, int start, int before, int count) {}

      @Override
      public void beforeTextChanged(CharSequence s, int start, int count, int after) {}

       @Override
     public void afterTextChanged(Editable s) {
            if(homeAmount.hasFocus())
                  refresh(s.toString(), fcAmount, 1.0/taxrate);
      }});
} //konec metody onCreate
```

To je totéž v opačném gardu. Změna obnosu v domácí měně si vynutí změnu obnosu měny zahraniční. Metoda *refresh* má prohozené odkazy na editační pole a využívá převracenou hodnotu kurzu.

Nyní už nám zbývá již jen pár metod (včetně výše použitých pomocných):

```
private static double parseTax(String s) {
      return Double.parseDouble(s.substring(0, s.length() - 1)) / 100.0;
}
```

Pomocná metoda *parseTax* převádí řádek rozvinovacího seznamu na odpovídající číslo. Odstraňuje poslední znak (to je procento, to není příliš robustní, ale programátor si to může ohlídat), převádí na číslo typu *double* a dělí stem.

```
private static void refresh(String amount, EditText target, double rate) {
      double origAmount;
            if (amount.equals(""))
            origAmount = 0.0;
      else {
            NumberFormat format = NumberFormat.getInstance(Locale.getDefault());
            Number number;
            try {
                  number = format.parse(amount);
                  origAmount = number.doubleValue();
            } catch (ParseException e) {origAmount = Double.parseDouble(amount);}
            }
            double targetAmount = origAmount * rate;

            target.setText(String.format("%.2f", targetAmount));
}
```

Nejsložitější pomocná metoda. V zásadě se jedná o pouhý výpočet cílového obnosu podle předaného kursu (předposlední řádek) a jeho nastavení v editačním řádku. Vše však komplikuje konverze zdrojového údaje, která musí zohlednit:

1. možnost prázdného vstupního řádku (ten je pak chápán jako nulový). Chybný formát není potřeba kontrolovat, neboť editační řádky mají definován typ vstupu (*inputType*) na "decimal", tj. žádnou jinou hodnotu než číslo nelze vložit (editační řádky s běžnými vstupními poli jsou k dispozici již v toolboxu).
2. nutnost ošetření národních nastavení (vstupní číslo je vždy v národním nastavení). Navíc to ukazuje další cestu jak přetypovat řetězec (a navíc ukazuje, že v Javě existuje nadtřída všech číselných tříd, na rozdíl od .NET).

```
@Override
public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
```

```
        taxrate = rate * (1.0 + parseTax((String)parent.getItemAtPosition(position)));
        if(fcAmount.hasFocus())
              refresh(fcAmount.getText().toString(), homeAmount, taxrate);
        if(homeAmount.hasFocus())
              refresh(homeAmount.getText().toString(), fcAmount, 1.0 / taxrate);
}
```

Metoda implementující rozhraní *OnItemSelectedListener*, která je volána při volbě jiné hodnoty na rozvinovacím seznamu (tj. změně úrovně kursovního poplatku). Kód je zřejmý, vypočítá se nový opravený kurs a zjistí se jaký editační řádek má zaměření. Tento řádek se nezmění, naopak se stane zdrojem pro změnu obnosu ve druhé měně.

Rozhraní *OnItemSelectedListener* vyžaduje i definici metody, která je volána v případě, že není zvolena žádná položka s rozvinovacího seznamu. To u nás nemůže nastat (doufám), ale v každém případě musí být tato metoda definována, i když jako prázdná.

```
@Override
      public void onNothingSelected(AdapterView<?> arg0) {}
} //konec definice třídy CalculatorActivity
```

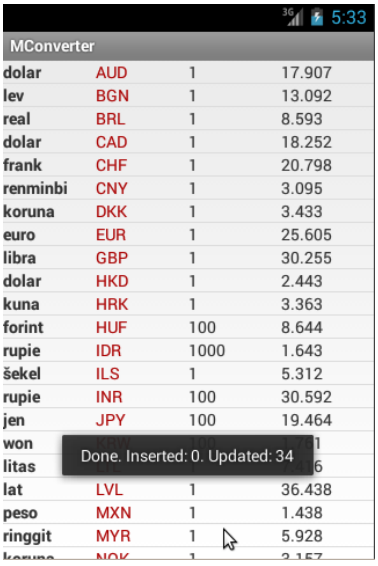
Tím máme hotovu implementaci aktivity kalkulátoru a celého projektu. Zbývá jen dodat aktivitu do souboru manifestu:

```
      <activity
         android:name="cz.ujep.ki.android.fiser.Calculator"
         android:label="@string/title_activity_calculator" > </activity>
```

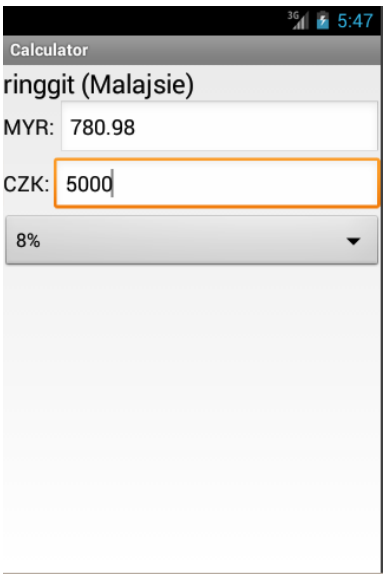
Včetně řetězce ve zdroji řetězců (*res/value/string.xml*):

```
  <string name="title_activity_calculator">Calculator</string>
```

Pro kontrolu si ukažme vzhled hlavní aktivity.



a na grafické rozhraní kalkulátoru.


5:47
Calculator
ringgit (Malajsie)
MYR: 780.98
CZK: 5000
8%

OTÁZKY

1. Jaký je v Androidu vztah mezi službou a vláknem?
2. Jaké rozhraní nabízí navenek poskytovatel obsahu?
3. Jakou roli hraje verzování datového úložiště?
4. Jakou funkci má (datový) adaptér?
5. K čemu slouží handler?

OTÁZKY K ZAMYŠLENÍ

1. Jaké vestavěné poskytovatele obsahu Android obsahuje?
2. Jaké prostředky pro informování uživatele o běhu služby lze využít kromě *zdravice* (toastu)?
3. Jaký je rozdíl mezi PULL a PUSH parserem u XML?
4. Kde je uložena databáze spravovaná poskytovatelem obsahu?

5 Geolokace

ODKAZY NA LITERATURU

ALLEN, Grant. *Android 4: průvodce programováním mobilních aplikací.* 1. vyd. Brno: Computer Press, 2013. Kapitoly 39-40 (527-548).

OTÁZKY

1. Jaký rozsah mají souřadnice longitude, latitude a altitude?
2. Jaký je rozdíl mezi COARSE_LOCATION a FINE_LOCATION?
3. Co je POI?

OTÁZKY K ZAMYŠLENÍ

1. Jaké metody geolokace Android používá? Uveďte jejich výhody a nevýhody?
2. Jaké dílčí služby nabízí Google API?

6 Sensory

ODKAZY NA LITERATURU

Android Developers. Sensors Overview.
Dostupné na http://developer.android.com/guide/topics/sensors/sensors_overview.html
Android Developers. Motion Sensors
Dostupné na http://developer.android.com/guide/topics/sensors/sensors_motion.html
Android Developers. Environment Sensors.
Dostupné na http://developer.android.com/guide/topics/sensors/sensors_motion.html

OTÁZKY

1. Jaké základní typy a druhy senzorů existují?
2. Jaké základní třídy existují?
3. Jak se získávají data se senzorů?
4. Popište prostorovou soustavu využívanou senzory?
5. K čemu je gyroskop?
6. Co je lineární akcelerometr?

OTÁZKY K ZAMYŠLENÍ

1. Co jsou virtuální senzory? Jakou mají funkci?
2. Co je rosný bod? Jak se počítá?