**Česká zemědělská univerzita v Praze**

**Provozně ekonomická fakulta**

**Katedra informačních technologií**


**Bakalářská práce**

# Analýza výhod reaktivního programování

**Šimon Smrček**

# ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Šimon Smrček

Informatika

**Název práce**

**Analýza výhod reaktivního programování**

**Název anglicky**

**Analysis of reactive programming advatages**

---

**Cíle práce**

Práce se zabývá tématikou vývoje mobilních aplikací. Hlavním cílem je zhodnotit výhody reaktivního programování v porovnání se standardní knihovnou. Vedlejšími cíli bude charakteristika reaktivního programování, standardní knihovny, obecného vývoje aplikací pro mobilní zařízení a operačních systémů.

**Metodika**

Metodika řešené problematiky bakalářské práce je založena na studiu a analýze odborných informačních zdrojů. Analýza výhod bude provedena porovnáním zdrojových kódů na základě ukázkové aplikace zabývající se tématikou správy sportovního klubu. Na základě syntézy teoretických poznatků a výsledků praktické části budou formulovány závěry bakalářské práce.

**Doporučený rozsah práce**

45 stran

**Klíčová slova**

asynchronní programování, reaktivní programování, mobilní aplikace, standardní knihovna

---

**Doporučené zdroje informací**

NURKIEWICZ, Tomasz a Ben CHRISTENSEN. Reactive programming with RxJava: creating asynchronous, event-based applications. Sebastopol, CA: O'Reilly Media, 2016. ISBN 978-1491931653.

Reactive Java programming. New York, NY: Springer Science+Business Media, 2016. ISBN 978-1484214299.

SAUMONT, Pierre-Yves. Functional programming in Java: how to improve your Java programs using functional techniques. Shelter Island: Manning, 2017. ISBN 978-1617292736.

URMA, Raoul-Gabriel, Mario FUSCO a Alan MYCROFT. Java 8 in action: lambdas, streams, and functional-style programming. Shelter Island: Manning, 2015. ISBN 978-1617291999.

---

**Předběžný termín obhajoby**

2018/19 LS – PEF

**Vedoucí práce**

Ing. Jan Masner, Ph.D.

**Garantující pracoviště**

Katedra informačních technologií

Elektronicky schváleno dne 11. 9. 2018

**Ing. Jiří Vaněk, Ph.D.**

Vedoucí katedry

Elektronicky schváleno dne 19. 10. 2018

**Ing. Martin Pelikán, Ph.D.**

Děkan

V Praze dne 22. 01. 2019

---

**Čestné prohlášení**

Prohlašuji, že svou bakalářskou práci "Analýza výhod reaktivního programování" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 5.3.2019 _____

**Poděkování**

Rád bych touto cestou poděkoval Ing. Janu Masnerovi, Ph.D. za vstřícný přístup a věcné připomínky. Dále bych také chtěl poděkovat Vojtovi Smrčkovi za pomoc s objasněním problematiky a všem blízkým, kteří mě podporovali v průběhu práce.

# Analýza výhod reaktivního programování

**Abstrakt**

Bakalářská práce se zabývá problematikou spojenou s vývojem mobilních aplikací za pomoci reaktivního programování. Cílem je porovnat reaktivní programování se standardní knihovnou a následně ukázat jednotlivé výhody a nevýhody.

Teoretická část je rozdělena do tří částí. První z nich je zaměřena na mobilní aplikace a jejich vývoj. Druhá část je zaměřena na asynchronní programování. Třetí část je zaměřena na reaktivní programování, knihovny s ním spojené a některé výhody.

Praktická část je rozdělena do dvou částí. První část se věnuje procesu měření a tomu, jak bude praktická část zpracovávána. Druhá část se věnuje analýze výhod jednotlivých problémů, mezi které patří zobrazení seznamu položek, načtení dat z API, práce s chybami, paralelní vs. sekvenční běh a pravidelné opakování procesu.

Na závěr je provedeno porovnání celkových výsledků a jejich zhodnocení. Dále je také doporučeno, který z daných problémů je lepší řešit pomocí reaktivního programování.

**Klíčová slova:** asynchronní programování, reaktivní programování, mobilní aplikace, standardní knihovna

# Analysis of reactive programming advantages

**Abstract**

This bachelor thesis is focused on the issue of mobile applications development with focus on development with reactive programming. The objective is to compare reactive programming with the standard library and show its advantages and disadvantages.

The theoretical part of the thesis is divided into three sections. The first section focuses on mobile applications and their development. The Second part is focused on asynchronous programming. The third part is focused on reactive programming, libraries connected with it and theoretical advantages.

The practical part of this thesis is divided into two sections. The first is focused on the process of measuring and the approach to practical part. The second part is focused on analysis of the advantages of the individual processes like showing of a list, loading data from an API, error handling, parallel vs. sequence processing and periodical repeating of a process.

The last part of this thesis is the comparison of all the results and their evaluation. As the result of the comparison there is a recommendation of which of the analyzed issues should be solved with reactive programming.

**Keywords**: asynchronous programming, reactive programming, mobile apps, standard library

# Obsah

# Seznam obrázků



# Seznam tabulek



# Zdrojové kódy

# 1 Úvod

Žijeme v době, kdy jsou chytrá mobilní zařízení nedílnou součástí našeho života. Pro jejich snazší užívání se vyvíjejí aplikace, které mohou sloužit k účelům sahajícím od řízení vlastního života až po volnočasové aktivity. Plynulý běh aplikací je základním požadavkem většiny uživatelů. S rostoucí komplexností aplikací je třeba na tento požadavek klást stále větší a větší důraz. A i proto se postupem doby aplikace posouvají právě v tomto kritériu nejvíce.

Většina z nás má na svém mobilním zařízení aplikaci, která byla pomocí reaktivního programování vyvíjena, aniž bychom si to uvědomovali. Díky reaktivnímu programování je snazší a efektivnější naimplementovat chování, kdy zařízení nezamrzne vždy, když provedeme nějakou aktivitu, nebo kdy můžeme snadno otáčet obrazovku bez omezení funkčnosti aplikace.

Díky mému podrobnému porovnání vybraných procesů také uvidíme, kolik výhod v konkrétních číslech získáme tím, když budeme reaktivní programování používat místo standardní knihovny. Touto prací bych tak rád seznámil uživatele s tím, jak jsou v dnešní době některé aplikace programovány a přesně zobrazil objektivní výhody oproti starým způsobům programování.

# 2 Cíl práce a metodika

## 2.1 Cíl práce

Práce se zabývá tématikou vývoje mobilních aplikací. Hlavním cílem je zhodnotit výhody reaktivního programování v porovnání se standardní knihovnou v programování na mobilní operační systém Android a následně doporučit, kdy reaktivní programování používat a naopak, kdy se vývoji pomocí něj vyhnout.

Vedlejšími cíli je charakteristika reaktivního programování, standardní knihovny, synchronního a asynchronního programování, obecného vývoje aplikací pro mobilní zařízení, mobilních operačních systémů a několika teoretických výhod reaktivního programování

## 2.2 Metodika

Metodika řešené problematiky bakalářské práce je založena na studiu a analýze odborných informačních zdrojů. Na základě syntézy zjištěných poznatků bude popsána problematika týkající se aplikací na mobilní zařízení a vývoje na operační systém Android, synchronního a asynchronního programování a reaktivního programování.

Analýza výhod bude provedena porovnáním zdrojových kódů na základě ukázkových aplikací na operační systém Android zabývající se tématikou správy sportovního klubu.
Z obou aplikací budou vybrány ukázky kódu, který provádí totožnou funkci přímo se týkající běhu aplikace, na kterých budou následně měřeny objektivní kritéria sloužící ke snadnému zobrazení a pochopení výhod i nevýhod reaktivního programování. Objektivní měřená kritéria budou počet řádků, rychlost výpočtu procesu, rychlost výpočtu aplikace a velikost APK.

Na základě syntézy teoretických poznatků a výsledků praktické části budou formulovány závěry bakalářské práce.

# 3 Teoretická východiska

## 3.1 Mobilní aplikace

Mobilní aplikace je program, který je zaměřena na smartphony, tablety, emulátory a ostatní mobilní zařízení. Snaha vývojářů mobilních aplikací spočívá ve vytvoření co nejvíce intuitivního uživatelského rozhraní, snadno použitelného i dotykově. Mobilní aplikace jsou dostupné především přes mobilní platformy, které provozují převážně výrobci jednotlivých operačních systémů. Mezi největší distribuční platformy patří Google Play, App Store od Apple, Amazon Mobile-apps a, nyní už téměř nepoužívaný, Windows Phone Store. Při vývoji je nutno brát v potaz i návrh mobilního uživatelského rozhraní, který je, společně s návrhem architektury aplikace, základem tvoření mobilních aplikací.[1]

### 3.1.1 Mobilní operační systémy

Mezi hlavní používané mobilní systémy (platformy) se dnes řadí především Android společnosti Google a iOS společnosti Apple. Každá platforma prosazuje vlastní přístup k tvorbě aplikací pro svá mobilní zařízení a vývojářům je tento přístup a s ním spojené směrnice, doporučován, někdy i nucen. Tyto přístupy jsou i navzdory podobnosti jednotlivých mobilních zařízení velmi odlišné. Poskytují pouze jistou pravděpodobnost, že vývojáři, kteří se jich drží, budou dělat platformu ovladatelnější. Tyto rozdíly mají pozitivní dopad, neboť se jedná o značku pozitivně vnímanou uživatelem, a proto je snazší přizpůsobení na novější verze operačních systémů, které uživatel používal již předtím. Na druhou stranu toto chování brání rozvoji dalších mobilních uživatelských rozhraní, protože se stávají rozhraními specifickými především pro určitou platformu a přechod na rozdílnou platformu vyžaduje určité přeučení.[2]

### 3.1.2 Android

Android je mobilní operační systém založený na jádře Linuxu, který je dostupný jako open source. Jeho vývoj je veden firmou Google, ale výrobce zařízení může za dodržování

[1] Mobile Application (Mobile App). Dostupné z: https://www.techopedia.com/definition/2953/mobile-application-mobile-app.

[2] NEDBÁLEK, Stanislav. Uživatelské rozhraní mobilních aplikací.

určitých podmínek tento operační systém upravovat. Název může být doplněn o název prostředí, které vyvíjí sám vývojář, například MIUI od firmy Xiaomi. Platforma je otevřena všem programátorům, kteří pro ni mohou vytvářet aplikace a následně je za úhradu nebo bezplatně distribuovat přes Google Play. Samotná platforma dává k dispozici nejen operační systém Android s uživatelským rozhraním pro zákazníky, ale i kompletní návod k nasazení operačního systému (například specifikace ovladačů) pro mobilní operátory a jednotlivé výrobce zařízení. V neposlední řadě také poskytuje efektivní nástroje pro vývoj aplikací – Software Development Kit (SDK).[3]

### 3.1.3 Vývoj mobilních aplikací na platformě Android

Oficiálně podporované vývojové prostředí pro aplikace na Android je nyní Android Studio.[4] Vývojáři nejsou nuceni v tomto prostředí programovat a mohou si tedy zvolit jiné IDE (vývojové prostředí; Integrated Development Environment) nebo textový editor s kompilací aplikací pomocí příkazové řádky.[5]

#### 3.1.3.1 Android Software Development Kit (SDK)

Samotné nástroje určené pro vývoj aplikací pro Android jsou obsaženy v SDK, který je dostupný pro všechny přední platformy Windows, macOS i Linux.[6]
Sada Software Development Kit je rozdělena na 3 části:

1. Základní konfigurace SDK:
    - SDK Tools – Jedná se o nástroje pro debugování a testování aplikace, dále také nástroje pro správu Android Virtual Devices, Android emulátor a analýzu grafického layoutu a další programy.
    - SDK Platform-tools – Obsahuje další klíčové nástroje pro vytváření aplikací, které jsou závislé na verzi platformy a musí být aktualizovány při zveřejnění další verze

[3] Android. Dostupné z: https://www.aktualne.cz/wiki/veda-a-technika/android-google/r~i:wiki:1424/?redirected=1540216787.

[4] Android Studio release notes. Dostupné z: https://developer.android.com/studio/releases/.

[5] SERDARU, Silviu. 15 ANDROID APP DEVELOPMENT TOOLS ESSENTIAL FOR EVERY DEVELOPER'S TOOLBOX.

[6] Get the Android SDK. Dostupné z: https://stuff.mit.edu/afs/sipb/project/android/docs/sdk/index.html.

SDK. Za zmínku stojí například nástroj Android Debug Bridge, který slouží k nahrání souborů do zařízení.

- Android SDK platforms – Platforma SDK se skládá z knihovny, systémového obrazu, příkladových kódů, vzhledů emulátoru a dalších zdrojů. Pro správnou kompilaci aplikace a AVD musí být obsažena minimálně jedna Android SDK platforma.

2. Doporučená konfigurace SDK:
   - USB Driver – Komponenta nutná při optimalizaci a testování aplikace, která byla na zařízení nainstalována.
   - Příklady kódů – Obsahuje ukázky kódů dostupné a aktuální pro každou platformu.
   - Dokumentace – Lokální kopie dokumentace pro Android Framework API.
3. Plné konfigurace SDK:
   - Google API – API (Application Programming Interface) umožňující rozhraní Google Maps možné využitelné v mobilních aplikacích.
   - Další SDK platformy – Například Market Licensing package obsahující knihovnu, která ověřuje legálnost kopie.

V Android SDK je dále obsažen i emulátor operačního systému, který umožňuje testování vytvořené aplikace bez fyzického zařízení. Android SDK a AVD Manager poskytuje možnsot konfigurovat volbu síťového připojení, karty a zapnutí jednotlivých virtuálních zařízení. Jednotlivé aplikace by se měli v emulátoru chovat stejně jako ve fyzickém zařízení, existují ale výjimečné situace, jejichž virtualizace je složitější, jako například přijímání hovorů, úroveň nabité baterie nebo video/audio vstup. I tyto situace se pomalu do novějších verzí emulátorů zavádějí.[7]

V dnešní době se jako hlavní prostředí prosazuje především Android Studio.

### 3.1.4 Android Studio

Android studio je vývojové prostředí založené na IntelliJ IDEA a navržené zvláštně pro vývoj aplikací na Android. Oficiálně představeno bylo 16. května 2013 na konferenci Google I/O. Od následujícího měsíce bylo zdarma dostupné pro uživatele na platformách Windows, Mac OS X a Linux. Android studio bylo představeno jako hlavní náhrada za

[7] Nástroje na vývoji aplikací pro Android. Dostupné z: http://www.elitecsoftware.cz/nastroje-na-vyvoji-aplikaci-pro-android/.

Eclipse Android Development Tools (ADT) na pozici hlavního IDE pro přirozený vývoj aplikací na Android. Android Studio podporuje stejné jazyky jako IntelliJ a CLion, například tedy jazyk Java, C++ a od verze Android Studio 3.0 dále i Kotlin. Nejedná se však o jediný vývojový nástroj, ve kterém jde aplikace pro Android vytvářet.[8]

### 3.1.5 Další vývojové nástroje pro Android

- Visual Studio-Xamarin – Užitečné pro všechny druhy mobilních aplikací, nezáleží zda Android nebo iOS nebo Windows.
- Unreal Engine – Slouží k vývoji real-time technologií.
- GameMaker: Studio – Nástroj sloužící k vytvoření 2D her na mobilní zařízení. Slouží především ke zkrácení a zjednodušení kódu.
- Vysor – Pomáhá s emulováním a následným ovládáním pomocí klávesnice.
- Basic4Android – Slouží k rychlému vývoji mobilních aplikací, ale není zadarmo.
- Dále stojí za zmínku například CppDroid, AIDE, IntelliJ IDEA a Unity 3Dd[9]

## 3.2 Asynchronní programování (AsyncTask)

### 3.2.1 Synchronní a asynchronní programování a jejich porovnání

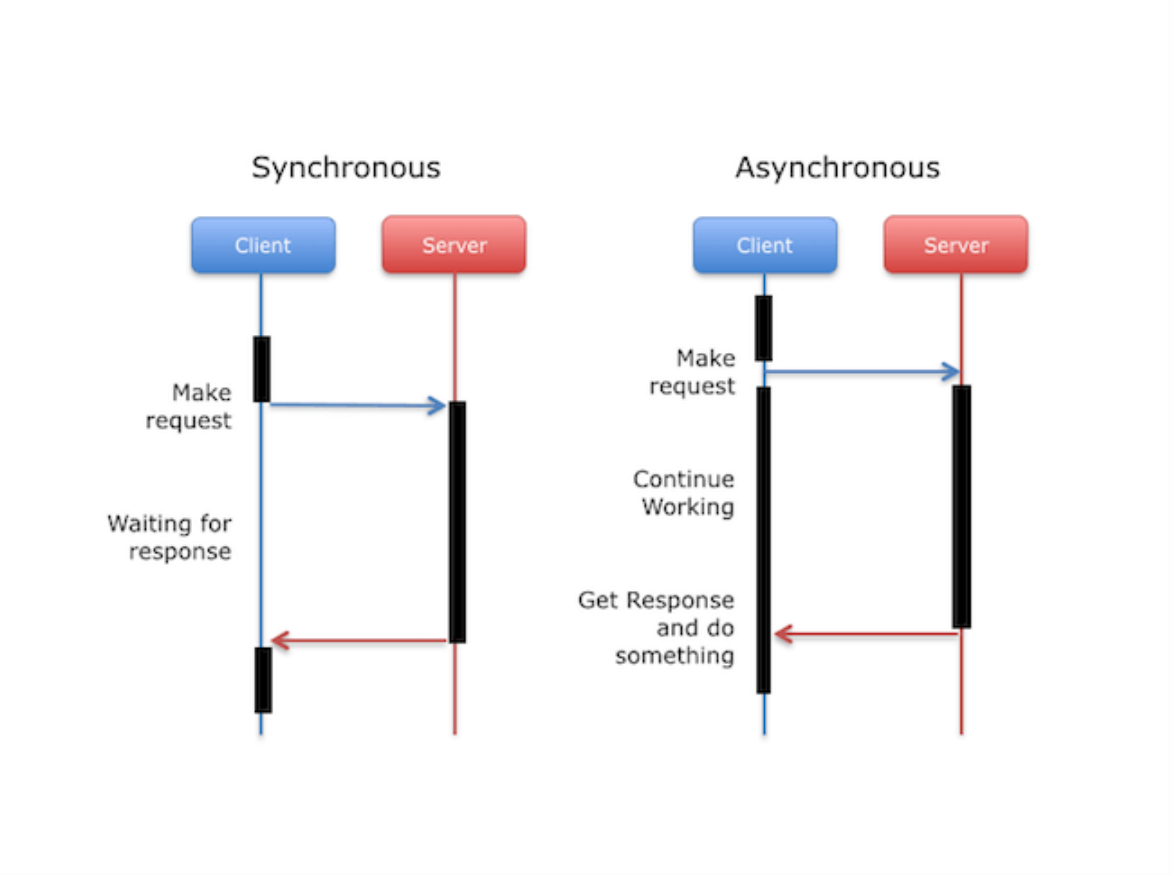
V synchronním programování v daný okamžik probíhá pouze jeden proces. Při volání funkce, která provádí dlouhotrvající činnost, program přestává fungovat po dobu běhu této akce a činnost je obnovena teprve po jejím dokončení a zobrazení výsledku.

Asynchronní programování je způsob paralelního programování, ve kterém jednotlivé pracovní jednotky běží nezávisle na hlavním vlákně aplikace a upozorňují zadávající vlákno na jeho dokončení (zobrazí výsledek), chyby i samotný průběh (viz. obrázek 1).

[8] DUCROHET, Xavier. Android Studio: An IDE built for Android.

[9] STONE, Sydney. Android Studio: An IDE built for Android.

**Obrázek 1/ Synchronní a asynchronní běh[10]**




Porovnání synchronního a asynchronního programování lze ukázat například na jednoduchém příkladu programu, který načítá dva zdroje ze sítě a kombinuje výsledky.

V synchronním programování se žádost vrátí teprve poté, co odvedla svou práci, a proto je nejjednodušší provést jednotlivá načtení jedno po druhém. Nevýhoda těchto načtení spočívá v tom, že druhé načtení bude provedeno teprve poté, co bylo ukončeno načítání zdroje prvního. Celková doba provedení tedy bude nejméně součtem těchto dvou dob odezvy.

Řešení tohoto problém v asynchronním programování spočívá v navýšení počtu řídících vláken. Mezitím, co hlavní vlákno provádí první požadavek, druhé vlákno začne pracovat na druhém a poté oba čekají na to, až přijdou zpět jejich výsledky načež se resynchronizují, aby získaly kombinaci výsledků.

Dá se tedy říct, že čekání na dokončení všech procesů v synchronním modelu je implicitní, zatímco v asynchronním, kde máme nad procesy kontrolu, se jedná o dokončení explicitní.[11]

---

[10] GIRIDHAR, Chetan. How To Simplify Networking In Android: Introducing The Volley HTTP Library.

### 3.2.2 Vlákno

Vlákno je základní jednotkou, které operační systém zařízení přiřazuje čas procesoru. V rámci jednoho procesu může fungovat i více vláken (multithreading). Jednotlivým vláknům přísluší vlastní priorita a systémové struktury, ve kterých je uložen kontext výpočtu po dobu neaktivity vlákna. Tento kontext obsahuje všechny informace nutné pro obnovu výpočtu, uložené obsahy registrů a zásobníku. Samotná vlákna mohou také v rámci jednoho procesu vykonávat různé činnosti – např. čtení požadavků, zpracování dat, vykreslení outputu na obrazovce a komunikaci se sítí.

Operační systém s preemptivním multitaskingem je schopen vytvořit zdání souběžného provádění několika vláken ve více procesech. To je umožněno rozdělením času procesoru mezi jednotlivá vlákna po menších časových intervalech. Jestliže přiřazený interval vyprší, běžící vlákno je pozastaveno, jeho kontext je uložen a obnoví se kontext jiného vlákna ve frontě, které se tak stane řídícím. Délka přiřazeného času je závislá na algoritmu, jenž je v systému implementován. Tyto úseky jsou ale z pohledu uživatele tak krátké, že výsledný dojem i na počítači s jedním procesorem je takový, jako by pracovalo větší množství vláken v jednu chvíli. Jestliže zařízení obsahuje více procesorů, jsou mezi ně vlákna rozdělována ke zpracování a k současnému běhu doopravdy dochází.[12]

Hlavní výhodou vláken je umožnění, aby aplikace měla dobrou odezvu na uživatelský vstup. Kdybychom měli pouze jednou vlákno, tak může nastat situace, kdy hlavní vykonávaná činnost programu zastaví ostatní úlohy, které potřebují dlouhou dobu zpracování, a tudíž může celý program přestat pracovat. Vytvořením pracovního vlákna pro dlouhotrvající úlohy, běžícího zároveň s hlavním vláknem, je možné udržet rychlou odezvu prostředí programu a zároveň provádět další časově náročné operace na pozadí. Dalšími výhodami je rychlejší běh programu, nižší spotřeba výkonu, lepší využití systému, zjednodušené sdílení a paralelní běh úloh.

Nevýhodou je například zvýšená složitost kódu, složitost tvoření vláken, omezení počtu vláken a složitější sledování toku programu.

---

[11] HAVERBEKE, Marijin. Eloquent Javascript: A Modern Introduction to Programming [online].

[12] BENEŠ, Miroslav. Procesy a vlákna.

Vlákna se dělí na vlákna na pozadí (background threads) a na popředí (foreground threads). Obecně se jedná o to stejné s jednou výjimkou: vlákno na pozadí neudržuje běh spravovaného prostředí. Po ukončení všech vláken v popředí ve spravovaném prostředí jsou ukončeny i všechny vlákna na pozadí a systém je ukončen. Background threads jsou ukončena násilně (přerušení výjimkou). Za zmínku také stojí hlavní vlákno (main thread), což je vlákno, na kterém se kreslí uživatelské rozhraní.[13]

### 3.2.3 Uživatelské rozhraní mobilních aplikací

*„Součást zařízení, pomocí které uživatel dané zařízení používá a ovlivňuje jeho chovaní, za účelem dosažení nějakého cíle.“*[14]

Aplikační uživatelské rozhraní by mělo brát v potaz omezení, kontexty, obrazovku, input a pohyblivost obrazovky jako předlohu pro design. Mobilní uživatelská rozhraní většinou bývají zaměřena na uživatele i proto, že jejich vstup umožňuje ovládání systému a výstup reflektuje uživatelův vliv na zařízení. Uživatelské rozhraní; neboli front-endy; spoléhají na back-end, který jim poskytuje přístup k systémům.

Designové požadavky na mobilní uživatelské rozhraní se zásadně liší od požadavků na stolní počítače a notebooky. Kvůli menší a dotykové obrazovce je nutné brát v úvahu všechny okolnosti, aby byla zajištěna použitelnost, čitelnost a konzistentnost. V mobilním zařízení mohou být více užívány symboly a některé ovládací prvky mohou být automaticky skryté, dokud nejsou pro uživatele potřebné. Samotné symboly také musí být menší a na obrazovce často není dostatek místa pro všechny textové popisy, což může být pro nezkušené uživatele matoucí.

Uživatelé musí rozumět jednotlivým ikonám příkazů a jejich významům, ať již prostřednictvím čitelného textu, nebo srozumitelného grafické znázornění. Základní pokyny pro návrh mobilního rozhraní jsou konzistentní napříč všemi moderními mobilními operačními systémy.

Osvědčené postupy pro návrh mobilního uživatelského rozhraní zahrnují například:

---

[13] Processes and threads overview. Dostupné z: https://developer.android.com/guide/components/processes-and-threads.

[14] STONE, Deborah L., Caroline JANETT a Mark WOODROFFE. User interface design and evaluation.

1. Rozvržení informací, příkazů a obsahu v aplikaci by mělo reflektovat toto rozvržení operačního systému v umístění, kompozici i barvách. Zatímco samotné aplikace se mohou do určité míry lišit, konzistence ve většině těchto bodů umožňuje uživatelům intuitivní nebo alespoň snadno pochopitelné ovládání uživatelského rozhraní.
2. Body určené na kliknutí musí být použitelné dotykem prstu. To znamená, že klikací bod nemůže být příliš malý ani úzký v žádném směru, aby si uživatel nevybral nežádoucí blízký bod (občas označováno jako „fat-fingering“).
3. Užitečný obsah by měl zabírat maximální plochu obrazovky. Elementy uživatelského rozhraní by neměly být vytvořeny na úkor samotného obsahu. Je důležité si uvědomit, že cílem uživatelského rozhraní není samotné jeho užívání, ale především usnadnění používání obsahu a aplikací.
4. Počet ovládacích prvků nebo možných zobrazených příkazů by měl být omezený, aby nedošlo ke zmatení uživatele a zabránilo se složité práci s obsahem.

Dosažení rovnováhy mezi naslouchání designerským požadavkům a řešením jednotlivých specifických požadavků různých aplikací není jednoduché. Navíc, uživatelské rozhraní každé aplikace by mělo být přizpůsobeno pro každý mobilní operační systém, protože to je vizuální jazyk, který bude uživatelovi znám a bude s ním nejlépe seznámen. I proto většinou vývojáři mobilních operačních systémů poskytují prostředky pro seznámení designerů mobilních aplikací s tím, jak funguje rozhraní daného operačního systému.[15]

### 3.2.4 AsyncTask

AsyncTask je třída frameworku, které umožňuje krátkým úlohám fungovat asynchronně v pozadí a následně dostat výsledek zpět na hlavní vlákno. Dá se tedy říci, že tato třída dovoluje provádění operací v pozadí a zveřejnění výsledku na hlavním vlákně bez toho, abychom museli manipulovat s vlákny manuálně.

AsyncTask je vytvořený jako pomocná třída okolo vláken a handlerů (třída) a nepředstavuje obecný vláknový framework. AsyncTasky by měly být ideálně použity pro krátké operace (nejdéle pár vteřin). Jestli je nutné, aby vlákna fungovala po delší časový úsek,

[15] Mobile UI (mobile user interface). Dostupné z: https://searchmobilecomputing.techtarget.com/definition/mobile-UI-mobile-user-interface.

lze použít různá API poskytnutá java.util.concurrent balíčkem, jako například Executor, ThreadPoolExecutor a FutureTask.[16]

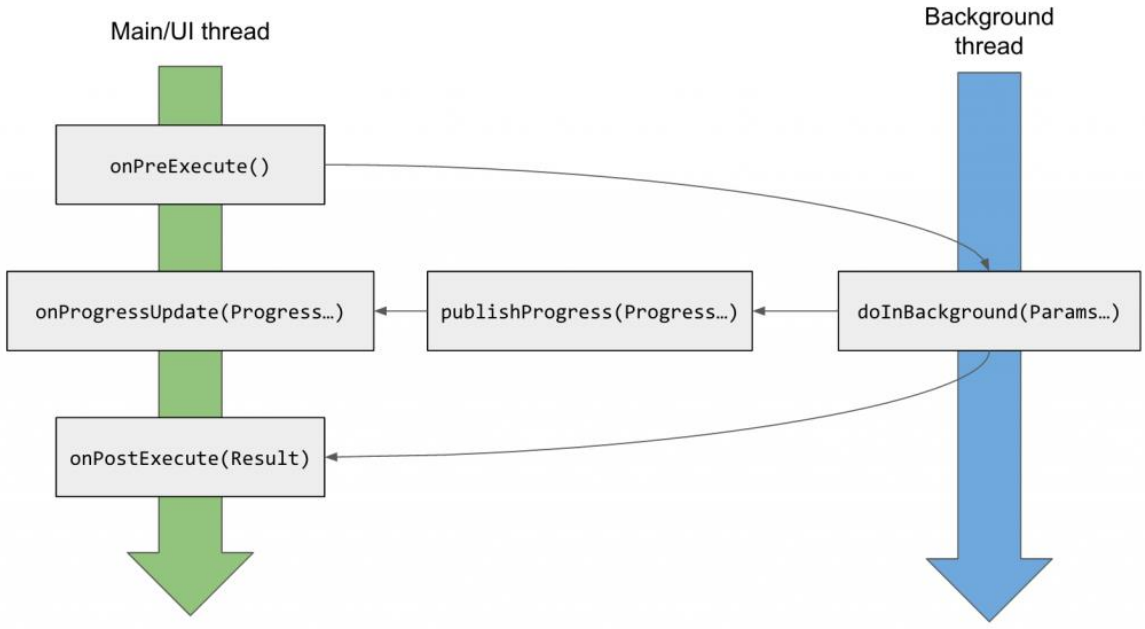
Asynchronní task používá:

1. Params – parametry, které jsou vráceny tasku po provedení
2. Progress – Jednotky průběhu operace, které jsou zveřejněny během výpočtu na pozadí
3. Result – Výsledky výpočty na pozadí

Po provedení asynchronního tasku přichází 4 kroky (viz. obrázek 2):

1. onPreExecute – Krok sloužící k přípravě tasku
2. doInBackground – Krok sloužící k provedení
3. onProgressUpdate – krok sloužící k aktualizaci dosavadní status průběhu tasku, který je prováděný v doInBackground
4. onPostExecute – Když doInBackground ukončí provádění tasku, onPostExecute odešle výsledky zpět na hlavní vlákno uživatelského rozhraní a zastaví AsyncTask process[17]

**Obrázek 2/ Funkce metod AsyncTasku[18]**




[16] SINKAL, Ankhit. Understanding of AsyncTask in Android.

[17] NEUPANE, Abinash. Why You Should Use AsyncTask in Android Development.

[18] AsyncTask overview in Android. Dostupné z: https://bhargavaroyal.wordpress.com/2016/05/03/asynctask/.

Jako příklad jde ukázat například situaci, kdy stahujeme soubor pomocí AsyncTasku: onPreExecute vyhledáme URL a zjistíme, jestli doopravdy existuje. doInBackground začne stahovat tento soubor po jednotlivých blocích a zobrazuje pokrok stahovaní po procentech. onProgressUpdate vykreslí pokrok v procentech na obrazovku. onPostExecute překreslí uživatelské rozhraní, jakmile je soubor stažený a zobrazí soubor.

### 3.2.5 Paralelní programování (pararelismus) a jeho nevýhody

Paralelní programování je v informatice označení konceptu, který zpřístupňuje naprogramování úloh, které jsou schopny současného běhu. Na rozdíl od asynchronního kódu, v paralelismu může běžet více funkcí v jeden moment a slouží především ke zvýšení výkonu v situaci, kdy není možné použít rychlejší zařízení nebo je nutné zjednodušit počítaný algoritmus. Paralelní výpočty mohou být realizovány pomocí více procesorů nebo počítačového clusteru, kde lze jednotlivé zařízení propojit pomocí sítě. Jedná se o rozdělení složitějších úloh na více jednodušších a následném rozdělení na více procesorů.[19]

Problémů paralelního programování je celá řada. Mezi hlavní lze zařadit problém s počtem paralelních výpočtů napojených na procesor. V případě, že jeden procesor bude čekat na druhý, kvůli složitosti jeho výpočtu, společný výpočetní čas se nezlepší. Proto je nutné, aby práce byla vyvážená a aby bylo možné vůbec danou úlohu paralelně provést.

Jestliže je granularita práce příliš velká, složitost při práci a rozdělování může způsobit počítači problémy. Jestliže je na druhou stranu granularita příliš nízká, některá jádra zařízení mohou zůstat bez práce. Proto je nutné, aby rozdělené části byly souměrné a zpracovatelné. Mezi další problémy lze zařadit omezený počet procesorů, složité alokování paměti, vyvažování dat a problém se sdílením cache.[20]

V dnešní době se dá říct, že ve výhodnější pozici je asynchronní programování. Když necháme pracovat něco asynchronně, znamená to, že to není blokující a nemusíme čekat na zpracování úlohy a raději se zaměřit na něco jiného. V paralelním programování běží více

[19] Rozdíl mezi asynchronním a paralelním kódem. Dostupné z: http://www.knesl.com/rozdil-asynchronni-paralelni-kod.

[20] OSTROVSKY, Igor. Parallel Programming with .NET: Most Common Performance Issues in Parallel Programs.

věcí naráz zároveň. Proto má paralelismus užší využití, například jestliže je práce snadno rozdělitelná do souměrných částí.[21]

## 3.3 Reaktivní programování

Jako reaktivní programování je nazýváno programovací paradigma orientované na datové toky a šíření změn. V reaktivním programování je možné jednoduché vyjádření statických nebo dynamických datových toků a samotné základní provedení modelu bude automaticky šířit změny v těchto datových tocích. To znamená, že datové toky reaktivního programování vysílané jednou součástí a základní struktura poskytnutá Rx (reaktivními) knihovnami budou šířit tyto změny ostatním součástem, které jsou spojeny s těmito změnami dat.

Jednoduchým příkladem je řešení rovnice $a := b + c$. V imperativním programování by bylo $a$ přiřazeno $b + c$ v momentu, kdy dochází k vyhodnocení a následně je možné změnit hodny $b$ i $c$ bez ovlivnění $a$. V reaktivním programování je hodnota $a$ automaticky aktualizována při každé změně hodnoty $b$ nebo $c$ a není nutno tedy vykonávat celý výpočet znovu.[22]

Definice reaktivního programování*: „Je vhodné rozlišit zhruba tři druhy počítačových programů. Transformační programy vypočítají výsledky z daných vstupů souboru; typickým příkladem jsou kompilátory, nebo numerické výpočetní programy. Interaktivní programy pracují svou vlastní rychlostí s uživateli nebo s dalšími programy, z uživatelského hlediska je time-sharing systém interaktivní. Reaktivní programy také udržují neustálé interakce se svým prostředím, ale při rychlosti, která je určena prostředím, nikoliv samotný program. Interaktivní programy pracují svým vlastním tempem, a hlavně jednají s komunikačními prostředky, zatímco reaktivní programy fungují pouze v reakci na vnější požadavky a většinou reagují na přesné obsluhy přerušení. Real-time programy jsou obvykle reaktivní. Nicméně zde*

[21] How to articulate the difference between asynchronous and parallel programming?. Dostupné z: https://stackoverflow.com/questions/6133574/how-to-articulate-the-difference-between-asynchronous-and-parallel-programming.

[22] Reaktivní programování. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Reaktivn%C3%AD_programov%C3%A1n%C3%AD&oldid=16047927.

*jsou reaktivní programy, které nejsou obvykle považovány za programy v reálném čase, jako jsou protokoly, systémové ovladače, nebo rozhraní manipulátory člověk–stroj.“*[23]

### 3.3.1 Funkcionální programování

*Funkcionální programování je deklarativní programovací paradigma, ve kterém lze manipulovat s funkcemi stejně snadno, jako například s ostatními hodnotami dat, například s integery a řetězci.*[24] Koncept za funkcionálním programováním je takový, že pokaždé, když voláme funkci se stejnými parametry, dostaneme stejné výsledky. Abychom tohoto dosáhli, nesmí funkcionální programovací jazyky držet žádný stav. Výhoda funkcionálního programování je v jednodušším testování kódu, větší předvídatelnosti a teoreticky tudíž i větší spolehlivosti.

### 3.3.2 Funkcionální reaktivní programování (FRP)

Jsou-li brány v potaz koncepty funkcionálního programování a reaktivního programování, tak lze jako myšlenku funkcionálního reaktivního programování určit modelování věcí jako je například uživatelův vstup více přímým deklarativním způsobem tím, že děláme jeho chování více explicitní. Modely FRP se v čase mění představením dvou nových datových typů: „událostí“ a „chování“. Události reprezentují hodnotu v určitém čase, zatímco chování zobrazuje hodnoty, které se v čase mění. Když jsou tyto koncepty funkcionálně spojeny, celý program se stane kombinací událostí a chování. Funkcionální reaktivní programování zvyšuje úroveň abstrakce kódu, čímž umožňuje programátorovi soustředění na vzájemné logické závislosti jednotlivých událostí, namísto složité implementace samotného kódu.[25]

---

[23] Real time programming: Special purpose or general purpose languages. Dostupné z: http://www-sop.inria.fr/members/Gerard.Berry/Papers/Berry-IFIP-89.pdf.

[24] Funkcionální programování. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Funkcion%C3%A1ln%C3%AD_programov%C3%A1n%C3%AD&oldid=16858691.

[25] HIBBLE, Adam. What is difference between Functional Reactive Programming, Functional Programming, and Reactive Programming?.

### 3.3.3 Reaktivní manifest

Reaktivní manifest se snaží definovat, jak má reaktivní aplikace vypadat a proč. Software vytvořený podle reaktivního manifest by měl lépe reagovat na zátěž, selhání, události, požadavky moderních trendů a uživatelů. Reaktivní manifest vznikl roku 2014 a reaguje na zvýšené požadavky aplikací v posledních letech. V minulosti měly velké aplikace desítky serverů, odezvu několik sekund, hodiny offline údržby a několik gigabytů dat. Aplikace dnešního typu jsou přizpůsobeny tomu, aby mohly být nasazeny na různé zařízení, od mobilních telefonů po cloudové clustery.

Reaktivní systémy by měly být podle Reaktivního manifestu flexibilní, volně vázané a škálovatelné. Jejich základní vlastnosti jsou:

- Reaktivita – Reaktivita není jen základem uživatelské přívětivosti a funkčnosti systému, ale také znamená, že problémy můžou být řešeny rychle a jejich řešení je efektivní.
- Odolnost – Odolný systém zůstává reagující i v případě závady. Odolnost je dosažena replikací, omezením, izolací a delegováním. Chyby jsou drženy v jednotlivých komponentách, čímž se izolují od ostatních komponent a tudíž jednotlivé součásti mohou selhat a se obnovit bez narušení celkového běhu programu.
- Pružnost – Pružný systém reaguje i na proměnlivé vytížení. Reaktivní systémy řeší změny v vstupních datech
- Řízení pomocí zpráv – Reaktivní systémy spoléhají na asynchronní předávání zpráv, čímž zajišťují volné spojení komponent. Toto spojení také poskytuje možnost delegovat chyby v podobě zpráv. Reaktivní komunikace také dovoluje přijímající straně používat zdroje při aktivitě což vede k nižší zátěži systému.[26]

### 3.3.4 Reactive extensions (ReactiveX)

ReactiveX (Rx) je knihovna pro některé programovací jazyky, která zavádí rozdílný pohled na práci s daty, obzvlášť kolekcemi a událostmi. Reactive extensions slouží k tvorbě asynchronních programů a programů založených na událostech. K tomu používají observable sekvence. Rx rozšiřuje vzor observer za účelem podpory sekvence dat nebo události a přidává operátory, které umožňují tvorbu sekvencí dohromady deklarativně přičemž abstraktizují

[26] Reactive Manifesto [online]. Dostupné z: https://www.reactivemanifesto.org/.

starosti s věcmi jako nízkoúrovňová vlákna, synchronizace, bezpečnost vláken, konkurenční datové struktury a neblokující I/O (input/output).

Někdy je nesprávně nazýváno „funkcionální reaktivní programování“, ale jedná se o jinou záležitost. Hlavním rozdílem je, že funkční reaktivní programování pracuje s proměnlivými, zatímco ReacitiveX pracuje s diskrétními hodnotami, které jsou vysílány v průběhu.[27]

Mezi Reactive extensions se řadí RxJava, RxJS, Rx.NET, RxScala, RxClojure, RxSwift, RxKotlin, RxPHP a další.

### 3.3.5 ReactiveX pro programování na Android

#### 3.3.5.1 RxJava

RxJava je implementace Reactive extensions v jazyce Java vytvořená společností Netflix. Jedná o knihovnu, která skládá asynchronní události tím, že následuje Observer pattern (viz. níže). Umožňuje vytvořit asynchronní datový tok na vybraném vlákně, přeměnit jeho data a zpracovat je libovolným Observerem na jiném vlákně. RxJava knihovna obsahuje širokou škálu operátorů aplikovatelnou na data stream jako například map, merge, filter a další. Jedná se o jednu z nejpoužívanějších knihoven pro fungování reaktivního programování pro Android.

RxJava nabízí dvě základní třídy z návrhového vzoru: Observable a Observer. Mimo to jsou také dostupné další části jako Schedulers, Operators a Subscription.

- Observable - Datový tok, který provádí určitou činnost a vysílá data.
- Observer – Protějšek Observable. Přijímá data vyslaná Observable.
- Subscription (v RxJava 2 Disposable) – Vazba mezi Observable a Observer. Může být několik Observerů vázaných na jeden Observable.
- Operator/Transformation – Operator modifikuje data vyslaná Observable před tím, než dorazí na Observer
- Schedulers – Scheduler rozhoduje o vláknu, na které Observable data odešle a který Observer data přijme[28]

[27] ReactiveX. Dostupné z: http://reactivex.io/intro.html.

[28] TAMADA, Ravi. Android Introduction To Reactive Programming – RxJava, RxAndroid.

Poslední vydanou verzí je verze 2.x, která je nyní brána jako stabilní a finální. Verze 1.x bude podporována ještě několik let dohromady s 2.x. Ve verzi 2 se mění některá pojmenování, například místo pojmu Observer se objevuje Consumer.[29]

#### 3.3.5.2 RxAndroid

RxAndroid je specifikace pro Android Platformu s několika novými třídami, které RxJava nemá. Jedná se například o podporu multithreadingového konceptu v androidích aplikacích, jako jsou Schedulers a Loopers. Schedulers v podstatě rozhodují, jestli daný kód poběží na vlákně na pozadí nebo na hlavním vlákně.

Přestože RxAndroid nabízí velké množství Schedulers, jako nejdůležitější a nejpoužívanější v programování pro android lze snadno označit Schedulers.io() a AndroidSchedulers.mainThread(). Schedulers.io() slouží k funkci operací, které nepředstavují větší zátěž pro procesor, jako například síťové volání, čtení souborů, databázové operace a tak dále. AndroidSchedulers.mainThread() poskytuje přístup k hlavnímu vláknu/vláknu grafického rozhraní Androidu. Na tomto vlákně by neměla být prováděna žádná složitá operace, neboť aplikace může přestat fungovat nebo odpovídat. Mezi další Schedulers se řadí Schedulers.newThread(), Schedulers.computation(), Schedulers.Single(), Schedulers.immediate(), Schedulers.trampoline() a Schedulers.from().[30]

### 3.3.6 Observer pattern

Observer pattern je softwarový návrhový vzor, ve kterém subjekt udržuje seznam svých závislostí, nazývaných Pozorovatelé (Observers) a automaticky je informuje o jakékoliv změně stavu, většinou tak, že zavolá jednu ze svých metod.

Užití Observer pattern spočívá především v implementaci distribuovaných systémů pro zpracování událostí v software řízeném událostmi. Většina moderních jazyků, jako například Java nebo C# mají vestavěné konstrukce, které implementují Observer pattern komponenty tak, aby bylo dosaženo snadného programování a kratšího kódu.

---

[29] What's different in 2.0. Dostupné z: https://github.com/ReactiveX/RxJava/wiki/What's-different-in-2.0.

[30] TAMADA, Ravi. Android Introduction To Reactive Programming – RxJava, RxAndroid.

Observer je také klíčovou součástí model-view-controller vzoru (architektonický vzor, který slouží k vývoji uživatelského rozhraní). Observer pattern je implementovaný ve většině knihoven a systémech, včetně téměř všech GUI (Graphical User Interface) toolkitech. Zásadním problémem návrhového vzoru Observer je fakt, že Subjekt drží pevné reference na Pozorovatele. Proto nemůže „garbage collector“ tyto reference automaticky odstranit i ve chvíli, kdy už není daný Pozorovatel používán. V případě Androidu tento problém nastane například ve chvíli, kdy je Pozorovatelem Aktivita. Při rotaci obrazovky je Aktivita odstraněna a je vytvořena nová instance. Java Virtual Machine ale nemůže původní Aktivitu odstranit, protože na ní existuje reference v Subjektu. Řešením je explicitní odstranění této reference v Subjektu. Problém ale lže řešit i pomocí slabé reference na Pozorovatele.[31]
Druhy Observers v RxJava2 jsou:

- Observer
- SingleObserver
- MaybeObserver
- CompletableObserver[32]

### 3.3.7 Observable

ReactiveX Observable model umožňuje uživateli jednat s datovými toky asynchronních událostí se stejnými operacemi, které se v některých programovacích jazycích používají pro zpracování kolekcí dat. Programátor následně nemusí používat velké množství callbacků a proto je kód snadněji čitelný a méně náchylný na chyby a pády.

V ReactiveX se observer hlásí k Observable a reaguje na jakýkoli objekt nebo sekvenci předmětů, kterou Observable vysílá. Tento návrhový vzor usnadňuje souběžné operace, protože nemusí zastavit funkci během čekání na objekty vyslané Obervable a místo toho využije Observer, který bude připravený čekat na změny, které v budoucnu Observable provede.

Observable se liší od ostatních v tom, jak produkují data a počtem vyslání, které udělají. Různé situace mohou vyžadovat různé Observably.

---

[31] Interface Observer<T>. Dostupné z: http://reactivex.io/RxJava/javadoc/io/reactivex/Observer.html.

[32] RAVI, Tamada. RxJava Understanding Observables.

Druhy Observables s jejich Observers a počtem vyslání:

- Observable – Používá observer „Observer“ a je pravděpodobně nejužívanější Observable. Může vyslat jakékoliv množství předmětů.
- Single – Používá observer „SingleObserver“ a vysílá vždy právě právě jednu hodnotu nebo chybovou hlášku. Stejný výsledek může zajistit i Observable, ale SingleObserver se vždy zaručí za to, že bude vyslání. Ukázkou použití Single je situace, kdy voláme síť a čekáme na odpověď, protože ta bude přinesena najednou.
- Maybe – Užívá observer „MaybeObserver“ a může nebo nemusí vysílat žádnou hodnotu. Využívané v situaci, kdy je možnost, že data budou vyslána.
- Flowable – Užívá Observer „Observer“ a funguje na stejném principu jako Observable, ale používá se pouze v případě, kdy Observable generuje větší množství akcí nebo dat, než může Observer zvládnout. Většinou se používá, když zdroj generuje více než 10 tisíc akcí a odběratel je nemůže pojmout
- Completable – Využívá Observer „CompletableObserver“, nevyšle žádná data a místo toho upozorní stav úkolu o úspěšném nebo neúspěšném splnění. Používá se při provádění úlohy, při které není očekávána žádná hodnota. Příkladem je úprava dat na serveru PUT requestem.[33]

### 3.3.8 Kotlin Coroutines

Kotlin Coroutines je přístup Kotlinu k práci s asynchronním kódem pomocí používání coroutines, což je myšlenka zastavitelných výpočtů, což znamená, že funkce může v určitém okamžiku pozastavit své provedení a pokračovat v libovolném momentu později.

„Coroutines are a new way of writing asynchronous, non-blocking code (and much more)“.

Coroutines jsou lehká vlákna. Lehké vlákno je vlákno, které není drženo procesorem. Znamená to, že není mapováno na přirozeném vlákně, takže nevyžaduje přepínání kontextu na procesor a tudíž jsou rychlejší. Jestliže tedy spustíme jednu z coroutines, vlákno je drženo pouze pokud je potřebné nebo používané a je interně vypuštěno, pokud je neaktivní. Stejně jako vlákna, i coroutines fungují paralelně, čekají na ostatní a komunikují.

[33] Observable. Dostupné z: http://reactivex.io/documentation/observable.html.

Coroutines pracují na sdíleném společenství vláken. Vlákna tedy v programu založeném na coroutines stále existují, ale jedno vlákno je schopné pobrat více coroutines, takže není nutná přítomnost příliš mnoha.[34]

Coroutines a vlákna jsou oboje forma multitaskingu. Rozdíl spočívá v tom, že vlákna jsou spravována operačním systémem, mezitím co coroutines uživatelem.
Koncept coroutines není nový, ani nebyl vynalezen Kotlinem. Používá se již desítky let a je populární i v dalších programovacích jazycích, jako je například Go. Důležitá ale je implementace, které byla zavedena Kotlinem, ve které je většina funkcionality delegována knihovnám.

Výhoda coroutines spočívá v menším zatěžování procesoru. Jelikož normální vlákna běží na úrovni procesoru, je možné jich najednou zvládnout pouze omezené množství a jsou drženy i v době nepoužívání. Coroutines jsou velmi nezatěžující, téměř úplně: Můžeme jich vytvořit tisíce a investovat velmi málo z hlediska výkonu. Pravá vlákna jsou na druhou stranu náročná na spouštění i následné udržení. Tisíce vláken mohou být pro nynější zařízení velký problém. Samotná práce s vlákny je navíc komplikovaná a časově náročná.

Jsou dva základní druhy Coroutines: Stackless a Stackful. Kotlin implementuje stackless coroutines, což znamená, že coroutine nemá vlastní „stack“ neboli zásobník, tudíž se nemapuje na přirozeném vlákně.[35]

### 3.3.9 Výhody Reaktivního programování

#### 3.3.9.1 Spravování vláken

Jedna z výhod reaktivního programování pomocí RxJavy je správa vláken (a obecně multithreading). Ve standardní Javě je práce na více vláknech a přepínaní mezi nim složitá a výrazně zvyšuje chybovost kódu. Je třeba řešit paralelní přístup ke zdrojům a proměnným a další problémy. Standardní knihovna platformy Android přichází s některými řešeními, která práci s vlákny usnadňují. Jedním z běžně používaných je třída AsyncTask.

AsyncTask je vhodný pro jednoduché úkoly, které nemůžeme (nebo nechceme) spouštět na hlavním vlákně. Se stoupající komplexitou problému se ale implementace pomocí

[34] SHEKHAR, Amit. What are Coroutines in Kotlin?.

[35] Asynchronous Programming Techniques. Dostupné z: https://kotlinlang.org/docs/tutorials/coroutines/async-programming.html.

AsyncTasku stává neudržitelnou. AsyncTask na správu vláken používá tři základní metody: onPreExecute, která je spuštěna jako první na vláknu uživatelského rozhraní a slouží k přípravě dat pro operaci na vlákně na pozadí, doInBackground, která slouží k provedení výpočetně náročné operace na pozadí a onPostExecute, která je opět spuštěna na hlavním vlákně a slouží k zobrazení výsledků operace na pozadí. Za zmínku ještě stojí metoda onProgressUpdate, která nám ukáže postup procesu na vláknu na pozadí v uživatelském rozhraní. Všechny AsyncTasky používají stejné vlákno, takže v případě spuštění více instancí na sebe musí jednotlivé instance čekat. Výpočet v tom případě neprobíhá paralelně. Vlákno, které bude použito, si nemůžeme vybrat, je nám automaticky přiřazeno systémem.

RxJava je v základním nastavení pracuje na jednom vlákně. To znamená, že Observable a řetězec operátorů, které jí mohou být přiřazeny, upozorňují Observery na stejném vlákně, na kterém se nachází její subscribe() metoda. Toto vlákno je obvykle hlavní vlákno uživatelského rozhraní. Z označení asynchronní by šlo usoudit, že RxJava je sama o sobě vícevláknová, a ačkoliv více vláken podporuje a nabízí velké množství užitečných nástrojů, které pomáhají asynchronní operace jednoduše řešit, nejedná se o vícevláknové chování.[36] Jestliže tedy chceme manipulovat s vlákny, musíme se k nim dostat přes vestavěné Schedulers. Scheduler si lze představit jako vlákno nebo kolekci vláken, které provádějí rozdílné činnosti. Zjednodušeně tedy lze říci, že jestliže potřebujeme provést úlohu na konkrétním vláknu, musíme pro něj vybrat správný Scheduler, který si následně vybere volné vlákno ze zásoby vláken a úlohu provede. RxJava nabízí několik Schedulers a jestliže nevybereme správný, úloha pravděpodobně nebude provedena optimálně. Mezi základní patří:

- **Schedulers.io()**, který slouží k provedení I/O úloh, které nejsou příliš procesorově náročné. Jedná se například o přístup k systému souborů, přístup do databáze, provádění síťových volání a tak dále. Schedulers.io() je neomezený a počet jeho vláken může růst.
- **Schedulers.computation(),** který slouží k provedení úloh náročnějších na procesor, jako je například zpracování velkých datových setů, práce s obrazem a tak dále. Počet povolených vláken je roven počtu dostupných procesorů, a proto je důležité, aby byl počet funkčních vláken pod tímto Schedulerem omezený (v opačném případě se budou jednotlivá vlákna přetahovat o čas výpočtu na procesoru).

[36] ROY, Aritra. Multi-Threading Like a Boss in Android With RxJava 2.

- **Schedulers.newThread(),** který slouží k vytvoření nového vlákna pro určitou činnost, které se spustí při každém jejím užití. To znamená, že se jedná o Scheduler, který nevyužívá žádnou kolekci vláken. Vytváření stále více vláken může vést ke zpomalení a problémům spojeným s nedostatkem paměti, takže užívání tohoto Scheduleru by mělo být omezené, nejlépe na dlouhotrvající aktivity.
- **Schedulers.single()**, který slouží k provedení úloh sekvenčně. Jedná se o jediné vlákno, takže nejlepší užití je provedení náročných funkcí z rozdílných míst v aplikaci.
- **Schedulers.from(Executor executor)**, který slouží k vytvoření osobního Scheduleru krytého vlastním Executorem. Využití lze najít například v situaci, kdy chceme snížit počet paralelních internetových volání, a tak vytvoříme vlastní Scheduler, který lze přiřadit všem Observables souvisejícím se sítí.
- **AndroidSchedulers.mainThread()**, který slouží k provádění úloh souvisejících s uživatelským rozhraním na hlavním vlákně aplikace. Pro jeho použití musíme implementovat knihovnu RxAndroid a lze využít pouze na platformě Android. Toto vlákno používáme jen pro výpočetně nenáročné operace tak, protože je sdíleno s operačním systémem a v případě jeho blokování může uživatel pozorovat zhoršené chování aplikace.[37]

#### 3.3.9.2 Práce s chybami

Jedním z častých problémů při řešení komplexních asynchronních operací je práce s chybami. Jedna ze základních vlastností RxJavy a reaktivních streamů je usnadnění chyb. AsyncTask nepodporuje žádné integrované řešení pro správu chyb, takže musíme všechny odchytávat manuálně a zpracovávat je. Chybové výpisy následně musíme vracet zpět do aplikace stejně jako data. Obvykle toto implementujeme pomocí nové třídy, která obsahuje jak data, tak chybu - pár <Result, Exception>. Tento přístup velmi prodlužuje kód a může snadno způsobit, že na některou část nebo výjimku zapomeneme. Neodchycená výjimka v AsyncTasku způsobí pád celé aplikace. Manuální řešení v AsyncTasku je možné v případě mála operací, jakmile však chceme řetězit operace, případně volat několik operaci asynchronně, je jejich správa velmi komplikovaná a délka kódu se protahuje.

[37] SHVARTS, James. Understanding RxJava subscribeOn and observeOn.

I když je toto řešení možné, tím, že není systémem vynucené, ho vývojáři velmi často ignorují a následně tím výrazně zvyšují chybovost své aplikace.

Reaktivní programování (v našem případě RxJava) řeší chyby jiným způsobem. Pracuje pomocí streamů a ve streamu se vyskytují tři typy objektů; datový objekt, chybový objekt a objekt, který značí, že se stream uzavírá. Protože jsou všechny části komplexní operace součástí jednoho streamu, stačí jednou na konci nastavit, jaká bude reakce programu, jestliže nastane chyba, a to se poté bude aplikovat na všechny výjimky i chyby. Tím se snadno zbavíme redundantního kódu. Navíc je toto zpracování chyb vždy jako úplně poslední krok, takže obvykle běží na hlavním vlákně, kde můžeme tyto chyby uživatelsky přívětivě zobrazit uživateli.[38] RxJava navíc umožňuje jednoduše a konzistentně zpracovávat chyby v průběhu streamu pomocí operací jako je doOnError, onErrorComplete, onErrorResumeNext, onErrorReturn, retry, retryUntil atd.[39]

#### 3.3.9.3 Paralelní funkčnost RxJava vs. Sekvenční funkčnost AsyncTask

Jako jednu z hlavních výhod reaktivního programování lze jednoznačně označit možnosti paralelismu neboli paralelního běhu dvou operací. To slouží především k citelnému zrychlení aplikace. Navíc lze pomocí RxJavy snadno určovat, jestli aplikace poběží sekvenčně na jednom vlákně, nebo paralelně na více vláknech.[40]

AsyncTask funkcionalitu paralelismu téměř nepodporuje a všechna řešení s ním jsou složitá a dalo by se říci, že až téměř nemožná. I proto je tato funkce nahrazována jednodušším sekvenčním řešením. Sekvenční řešení umožňuje téměř všechny funkce paralelního řešení nahradit, ale platí za to často složitějším zápisem, a především pomalejším výpočtem. Mezi další nevýhody lze řadit nutnost ukládání mezivýsledků. Jestliže totiž po každé operaci data neuložíme, tak na konci nebude možné jejich párování. Toto je důležité především ve chvíli, kdy chceme jednotlivé AsyncTasky na sebe navázat a výsledek zobrazit na uživatelském rozhraní. Android často ukončí životní cyklus Aktivit a tak můžeme přijít o výsledky AsyncTasku.

---

[38] HAMBRICK, Ross. Replace AsyncTask and AsyncTaskLoader with rx.Observable – RxJava Android Patterns.

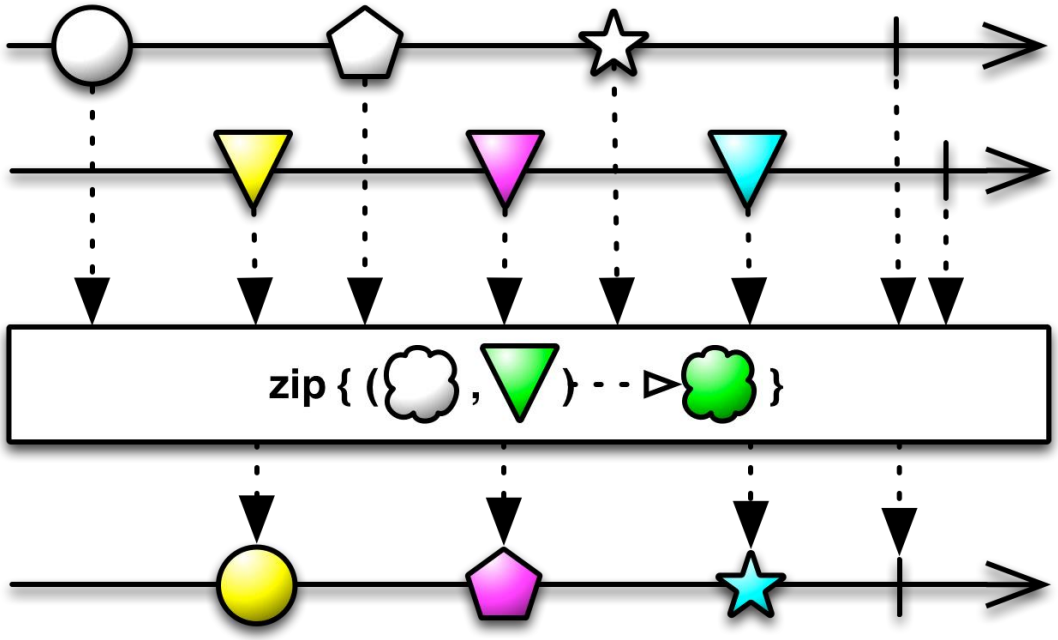
[39] Error Handling Operators. Dostupné z: https://github.com/ReactiveX/RxJava/wiki/Error-Handling-Operators.

[40] Synchronizing Network Calls With RxJava. Dostupné z: https://medium.com/@DoorDash/synchronizing-network-calls-with-rxjava-d30f7db66fb9.

Paralelismus je obzvlášť užitečný ve chvílích, když potřebujeme vzít data z více zdrojů a spojit je dohromady. Jestliže nám nezáleží na pořadí těchto dat, můžeme jednoduše vzít jeden zdroj (řekněme zdroj A), načíst ho a následně udělat to stejné i s druhým (zdroj B). V tom případě by paralelismu nebylo potřeba a sekvenční přístup by byl dostačující. Výsledky je nutné průběžně ukládat, protože jinak jsou při startu další části sekvence ztraceny a nebude možné jejich konečné párování. Další problém nastává, jestliže bude mít jeden ze zdrojů jakoukoliv chybu. V tom případě by byla funkce daného zdroje přerušena a druhý by byl ukončen nezávisle na prvním. V reaktivním programování se při chybě ve funkcionalitě jednoho zdroje celý proces okamžitě zastaví, tudíž neuložíme neužitečná data a výsledek nebude vadný.

Reálný problém nastává za předpokladu, že je potřeba, aby data ve výsledku tvořila jakýsi "zip". To znamená, že je požadováno, aby byla informace ze zdroje A následována informací ze zdroje B. V tomto případě už sekvenční přístup AsyncTasku ani RxJava nestačí. Operátor, který slouží k podobným aktivitám je .zip (viz. obrázek 3). Použití zipu samo o sobě paralelismus znamenat nemusí, ale když je použit subscribeOn(Schedulers.newThread()) na každou observable obsaženou v zip operátoru, tak scheduler.newThread() pro každou z nich vytvoří vlastní vlákno. Ve stejnou chvíli se tedy budou spouštět všechny observable, tudíž by výsledek měl vypadat přibližně A, B, A, B (jestliže se bude jednat o dva zdroje). RxJava tady zpracovává paralelně přicházející data ze dvou zdrojů a tato data spojuje dohromady.

**Obrázek 3/ Funkce operátoru zip**[41]




K propojení většího množství Observables můžeme použít i několik dalších operátorů. Za zmínku stojí combineLatest, join a groupJoin, merge, mergeDelayError, rxjava-joins, startWith a switchOnNext.[42]

### 3.3.9.4 Životní cyklus aktivit a fragmentů

Dalším problémem, který v AsyncTasku není plně dořešený je situace, kdy je nutné odejít z aktivity nebo otočit zařízení v průběhu běhu AsyncTasku. Jestliže je poslána jednorázová zpráva, tak AsyncTask postačuje, jestliže je ale například potřeba změnit uživatelské rozhraní na základě výsledků, které z procesu získáme, dostaneme NullPointerException a chybovou hlášku aplikace, neboť Aktivita nebude dostupná a nebude mít žádnou hodnotu.

Vývojář musí zajistit, že odkaz na úlohu je někde uložen a zrušit ho, když je Aktivita ukončena a nebo zaručit, že Aktivita je ve správném stavu, než se pokusí upravit uživatelské rozhraní v onPostExecute(). Toto zvyšuje šum, který zabraňuje v dokončení práce v jednoduchém a udržitelném stavu.[43]

---

[41] DRILLER, Jens. RxJava Tidbits #2: Parallelism with Observable.zip().

[42] SINGH, Rohit. RxJava Parallelization Concurrency : Zip() Operator.

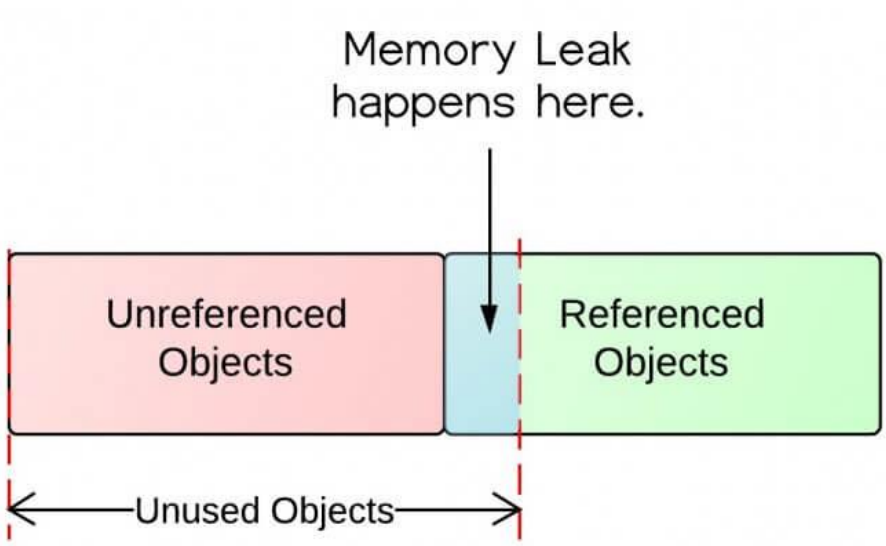
[43] HAMBRICK, Ross. Replace AsyncTask and AsyncTaskLoader with rx.Observable – RxJava Android Patterns.

#### 3.3.9.5 Příprava na otočení

Jestliže uživatel zůstává na aktivitě a v průběhu otočí zařízení, čímž zničí aktivitu nebo fragment, může nastat ukončení této aktivity, její znovunačtení a pokus o dokončení.
Za předpokladu, že je použit AsyncTask, může nastat problém, protože aktivita, která byla přerušena a znovu zapnuta otočením obrazovky nemůže najít odkaz na svůj počátek a tím pádem nebude ukončena. Dále také může nastat situace, při které dojde ke změně stavu, ačkoliv výsledek je potřebný k úpravě uživatelského rozhraní.[44]

S přípravou na otočení je úzce spjat problém, který se nazývá „Memory leak“ (viz. obrázek 4) neboli únik paměti. Jedná se o situaci, kdy program nemůže uvolnit objekty, na které stále odkazuje, takže ani samotná paměť jich nikdy není zbavena. Velké množství úniků paměti nelze na první pohled zahlédnout nebo alokovat, ale při otáčení obrazovky by měl program jasně reagovat na nemožnost postupu akce chybovou hláškou.[45]

**Obrázek 4/ Jak funguje memory leak[46]**




Jako řešení je nutné vytvořit Disposable (rozhraní obsahující 2 metody) sloužící k odhlášení odběru Observable při zrušení fragmentu, čímž přestanou dočasné úniky paměti. Poté je vytvořena dlouhotrvající akce, například volání dat ze sítě. Proto je vytvořena Observable, která toto zajistí a Subscriber, který pomůže zprávu zachytit. Zároveň je také

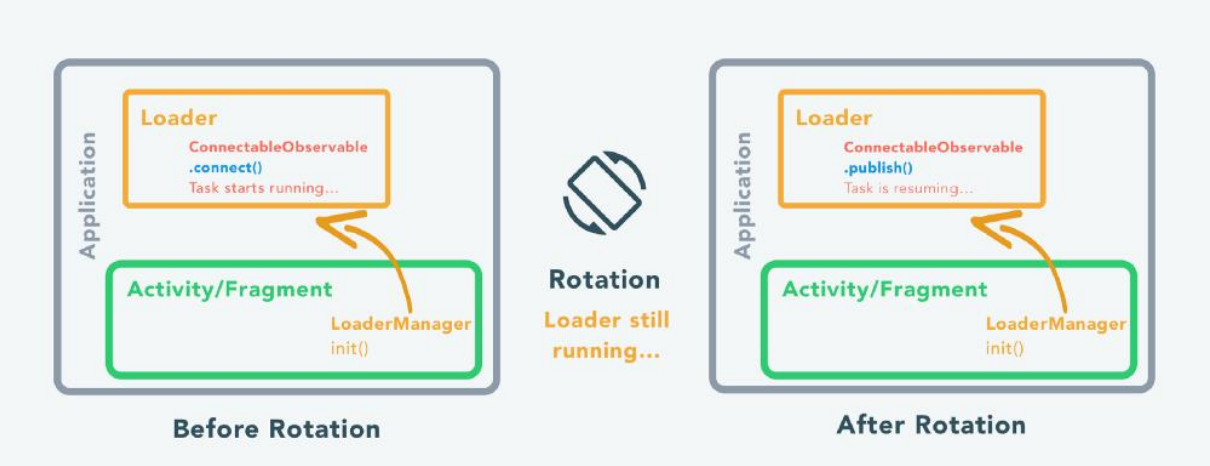
[44] HAMBRICK, Ross. Replace AsyncTask and AsyncTaskLoader with rx.Observable – RxJava Android Patterns.

[45] PARASCHIV, EUGEN. How Memory Leaks Happen in a Java Application.

[46] PARASCHIV, EUGEN. How Memory Leaks Happen in a Java Application.

informován TextView, když bude úloha započata a ukončena. Následně spustíme předchozí stejný dlouhotrvající proces, ale místo obyčejné Observable je použit ConnectableObservable, která dovoluje provést subscribe bez jeho okamžitého spuštění. Observable je vložena do Loaderu (nová třída), který je zodpovědný za provádění dotazů na různých vláknech, sledování datových zdrojů kvůli změnám a doručování nových výsledků registrovaným recipientům a v tomto případě slouží k podpoře Observable ve složitějších aktivitách, jako je třeba onDestroy nebo změny v konfiguraci (otočení obrazovky).

**Obrázek 5/ Otáčení obrazovky RxJava[47]**




Při spuštění úlohy je tedy spuštěn Observable a vložen do Loaderu pomocí compose() metody. V onCreateView() musí být obnovena kterákoliv Observable, která byla předtím spuštěná, tím, že požádáme LoaderManager na odkázání ke správnému Loaderu. Jestli takový Loader existuje, tak je bez spuštění proveden subscribe a úloha pokračuje (viz. obrázek 5).

Pokud jsou provedeny operace, které načítají data určená pouze na čtení, lze pro vyřešení tohoto problému použít AsyncTaskLoader. Ve standardním programování na Android bohužel dochází k tomu, že musíme použít kód, který je obsažen na velkém množství míst bez většího přínosu nebo změny (boilerplate code). Mezi další problémy lze také zařadit nedostatek prostoru pro řešení chyb, žádné cacheování mezi Aktivitami a obecně nepřehlednější kód.[48]

---

[47] BOISNEY, Philippe. How Memory Leaks Happen in a Java ApplicationRxJava 2 | Android : How to properly handle rotations with ConnectableObservable & Loader.

[48] BOISNEY, Philippe. How Memory Leaks Happen in a Java ApplicationRxJava 2 | Android : How to properly handle rotations with ConnectableObservable & Loader.

#### 3.3.9.6 Callback Hell neboli problém s více volání

Callback neboli zpětné volání je funkce předaná jiné funkci ve formě argumentu, který je následně volaný, aby splnil nějakou akci nebo rutinu.[49]

Callback hell je situace, která vzniká v asynchronním programování v momentu, kdy je na v kódu velké množství vnořených zpětných volání, čímž se kód stává složitý na čtení a opravu. Callback hell většinou vzniká v situacích, kdy kód píšeme v delším časovém úseku, a tudíž si neuvědomujeme, že zanořených zpětných volání je příliš.[50]

## 3.4 Shrnutí

Vývoj mobilních aplikací je odvětví, které se v informatice stále rozvíjí a trendy jsou proměnlivé. Synchronní programování bylo v tomto případě nahrazováno asynchronním už nějakou dobu, ale ani standardní knihovna v asynchronním programování už není pro udržení současného vývoje dostatečná. Proto vzniklo reaktivní programování sloužící ke snadnější obsluze datových toků a pro jeho rozšíření také knihovny jako je RxJava nebo RxAndroid. Ty mají předem odhadnutelné teoreticky shrnutelné výhody, které jednoznačně určují přímé přednosti oproti standardní knihovně, především se jedná o spravování vláken, práci s chybami, paralelní funkčnost a jednodušší možnost otočení obrazovky. Ani reaktivní programování ovšem není finální fáze a v některých firmách bývá jako alternativa používáno Kotlin Coroutines.

---

[49] Callback (computer programming). Dostupné z: https://en.wikipedia.org/w/index.php?title=Callback_(computer_programming)&oldid=874031687.

[50] SAILAPPAN, Vijay. Using RxAndroid to fix callback hell.

# 4 Vlastní práce

## 4.1 Zpracování praktické části

Cílem této práce je porovnat několika základních programovacích technik při vývoji pro Android v tradičním a reaktivním programování. Porovnání je zaměřeno především na počet řádků kódu, rychlost výpočtu procesu, rychlost výpočtu aplikace a velikost APK. Pro řešení je použito vývojové prostředí Android Studio, které slouží přímo k programování na mobilní systém Android, a v dnešní době je pro tento účel nejpoužívanějším nástrojem. Android Studio je přímo podporováno firmou Google, která stojí za platformou Android. Kód je napsán v programovacím jazyce Java, který je oficiálně podporovaný firmou Google pro vývoj aplikací pro platformu Android. Android Studio, ve kterém bude kód k praktické části bakalářské práce zpracován, je šířen formou freeware a všechna užitá rozšíření jsou dostupná ke stažení na internetu.

Problematika je naznačena ukázkou kódu napsaného pomocí standardní knihovny AsyncTask, která je porovnána s řešením pomocí RxJavy se stejnou funkcionalitou. Všechny příklady se týkají aplikace spojenou se správou sportovního celku.

### 4.1.1 Testovaná kritéria

Testovaná kritéria byla vybírána s ohledem na jejich objektivitu a relevantnost v daném měření. Pro uživatele je jedním z rozhodujících faktorů možnost vidět výhody v rychlosti výpočtu, a proto je kritérium bráno jako stěžejní. Jako další důležitá kritéria byly vybrány počet řádků, rychlost výpočtu celé aplikace a velikost APK. U některých ukázek se také můžou objevit další objektivní ukazatele relevantní k dané ukázce. Přesný výčet je následující:

- **Počet řádků** – dané kritérium má za úkol zjistit délku napsaného kódu. Jestliže se jedná o kód skládající se z více částí, zobrazen je buď jejich součet nebo počet řádků relevantních k danému tématu. Pro uživatele je důležité, aby počet řádků byl co nejnižší.
- **Rychlost výpočtu aplikace** – tímto kritériem sledujeme rychlost buildu celé aplikace. Před každým novým výpočtem je nutné předchozí výpočet vyčistit. Měření probíhá třikrát, následně je z těchto měření vypočítán aritmetický průměr a vyhodnocuje se finální hodnota kritéria. Kritérium by mělo mít co nejnižší hodnotu.

- **Rychlost samotné operace** – kritérium sleduje rychlost výpočtu samotné operace pomocí dvou logů měřících začátek a konec zpracování operace.

```
Log.i("Měření", "Start: " + System.currentTimeMillis());
Log.i("Měření", "End: " + System.currentTimeMillis());
```

  Toto měření opět proběhne třikrát a finální hodnota kritéria je vyhodnocena na základně aritmetického průměru. Kritérium by opět mělo mít minimální hodnotu.

- **Velikost APK (Android application package)** – dané kritérium sleduje velikost souboru, který je distribuovaný na mobilní zařízení a z kterého je následně instalována naše aplikace. Kritérium je měřeno po vyčištění předchozího buildu a jeho hodnota by měla být minimální.

## 4.2 Porovnání řešení

### 4.2.1 Synchronní aktivita - Zobrazení seznamu položek v aplikaci

Jestliže chceme zobrazit seznam dat, která už jsou načtená v paměti, nepotřebujeme asynchronní volání. Nejedná se o dlouhotrvající operaci, takže není problém, aby tato operace proběhla na hlavním vlákně aplikace.

V našem řešení (viz. zdrojový kód 1) posíláme funkci loadData parametr callback, který obsahuje funkci, která zobrazí výsledek na uživatelském rozhraní. Následně máme metodu, která nám synchronně vrátí data z paměti. Tato metoda se jmenuje getPlayersFromMemory.

**Zdrojový kód 1/ Standardní knihovna - Zobrazení seznamu položek v aplikaci**

```
public void loadData(Consumer<List<String>> callback) throws Exception {
    callback.accept(getPlayersFromMemory());
}
```

Stejného výsledku jde dosáhnout i za pomoci RxJavy. Kód ale bude odlišný, neboť je nutné použít Observable.just, která slouží k vyslání dat, například listu (viz. zdrojový kód 2). Volaná metoda za pomocí reaktivního programování bude vypadat takto:

**Zdrojový kód 2/ RxJava - Zobrazení seznamu položek v aplikaci**

```
public void loadData(Consumer<List<DataModel>> callback) {
    Observable.just(getPlayersFromMemory()).subscribe(callback);
}
```

Na první pohled lze usoudit, že změna není nijak razantní, objektivní výsledky měření vypadají takto:

**Tabulka 1/ Vlastní výsledky - Výsledky měření synchronní aktivity – Zobrazení seznamu**

| | Standardní knihovna | Reaktivní programování |
|---|---|---|
| **Počet řádků** | 3 řádky | 3 řádky |
| **Rychlost výpočtu aplikace** | 58s 231ms | 59s 251ms |
| **Rychlost samotné operace** | 231 ms | 228 ms |
| **Velikost APK** | 3,2mb | 3,4mb |

Z výsledků lze jednoznačně vidět, že v tomto případě nepřevládají výhody reaktivního programování, naopak, ve většině měřených kategorii zaostává (viz. tabulka 1). To lze přisoudit faktu, že RxJava pro tuto práci není přímo vytvořena a v tomto případě je její užití nadbytečné. Počet řádků v případě obou řešení je stejný, zatímco rychlost výpočtu aplikace a velikost APK hovoří pro standartní knihovnu. Jako jedinou výhodu lze určit rychlost samotné operace, která je o 3 ms rychlejší.

### 4.2.2 Asynchronní aktivita – Načtení dat z API

Načtení dat z API je nutné provádět asynchronně. V případě synchronního přístupu spadne aplikace na výjimce. To, že je výpočet asynchronní znamená, že alespoň jeden proces běží na vlákně na pozadí. Třída AsyncTask je vytvořena k tomuto účelu. Nevýhoda však je, že nemůžeme ovlivnit vlákno, na kterém bude AsyncTask spuštěn. Tudíž se může stát, že při spuštění více časově a výpočetně náročných operací pomocí AsyncTasku budou operace běžet sériově (a nikoli paralelně) a budou se vzájemně blokovat.

Pro konzistenci používáme jednoduché rozhraní Consumer, které je spuštěno automaticky na vlákně na pozadí. Uvnitř metody můžeme následně volat API, aniž by aplikace spadla. Uvnitř metody používáme Retrofit k načtení seznamu hráčů z API. Tento seznam je následně návratová hodnota metody doInBackground.

Druhá metoda onPostExecute očekává výsledek od doInBackground a po jeho ukončení ho považuje za svůj parametr. Za předpokladu správného výpočtu zavolá callback s parametrem seznamu hráčů a ten je následně dostupný na hlavním vlákně.

**Zdrojový kód 3/ AsyncTask - Načtení dat z API**

```
class SimpleDataLoaderAT extends AsyncTask<Void, Void, List<DataModel>> {
    private IMyApi myApi;
    private Consumer<List<DataModel>> callback;

    SimpleDataLoaderAT(IMyApi myApi, Consumer<List<DataModel>> callback) {
        this.myApi = myApi;
        this.callback = callback;
    }

    @Override
    protected List<DataModel> doInBackground(Void... params) {
        Call<List<DataModel>> call = myApi.callPosts();
        List<DataModel> posts;
        try {
            posts = call.execute().body();
        } catch (IOException e) {
            e.printStackTrace();
            posts = new ArrayList<>();
        }
        return posts;
    }
    @Override
    protected void onPostExecute(List<DataModel> dataModels) {
        super.onPostExecute(dataModels);
        try {
            callback.accept(dataModels);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Obdobného výsledku dosáhneme pomocí RxJava (viz. zdrojový kód 4). Stejně jako v řešení pomocí AsyncTasku i zde používáme callback třídy Consumer pro předání výsledných dat na uživatelské rozhraní.

V prvním kroku dostaneme Observable z API. Retrofit přímo vytvoří Observable a předá nám ji jako výsledek volání metody.

Na této Observable nastavíme pomocí metody subscribeOn hodnotu Schedulers.io, která zajistí, že se proces bude odehrávat na vlákně na pozadí s vysokou prioritou. Abychom obdrželi výsledek na hlavním vlákně (a mohli následně pracovat s uživatelským rozhraním), zavoláme metodu observeOn s parametrem mainThread.

Volání následně spustíme pomocí metody subscribe, jejíž výsledek rovnou předáme do callbacku na uživatelské rozhraní.

**Zdrojový kód 4/ RxJava - Načtení dat z API**

```
class SimpleRxDataLoader implements AsyncApp.Loader<List<DataModel>> {
    private IMyApi myApi;
    SimpleRxDataLoader(IMyApi myApi) {
        this.myApi = myApi;
    }
 public void loadData(Consumer<List<DataModel>> callback) {
        myApi.observablePosts()
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(callback);
    }
}
```

Na první pohled lze vidět, že první kód (viz. zdrojový kód 3) je delší než druhý kód (viz. zdrojový kód 4). Objektivní výsledky měření vypadají takto:

**Tabulka 2/ Vlastní výsledky - Výsledky měření načtení dat z API**

| | **Standardní knihovna** | **Reaktivní programování** |
|---|---|---|
| **Počet řádků** | 30 řádků | 10 řádků |
| **Rychlost výpočtu aplikace** | 1m 9s | 57s |
| **Rychlost samotné operace** | 555 ms | 423 ms |
| **Velikost APK** | 4,46 mb | 4,68 mb |

V tomto případě (viz. tabulka 2) lze vidět, že výhody reaktivního programování převažují ve většině bodů. Především poté v počtu řádků, rychlosti výpočtu aplikace a v rychlosti samotné operace. Jako jedinou nevýhodu lze brát velikost APK, která je zvětšená knihovnou RxJava.

### 4.2.3 Asynchronní aktivita – Práce s chybami

Teoretické základy práce s chybami je možné najít v kapitole práce s chybami (viz. kapitola 3.3.9.2). Práce s chybami je naznačena na ukázce podobné té z nahrávání dat z API (viz. zdrojový kód 3 a 4), ale tentokrát nám budou stačit pouze metody. Před jejich samotnou úpravou je vytvořena pomocná třída, která slouží k nastavení základních stavů úspěšného načtení i chybového načtení z API.

Jako první bude zobrazena problematika na standardní knihovně (viz. zdrojový kód 5). Porovnávaný kód se skládá ze dvou metod: doInBackground a onPostExecute. Jako návratová

hodnota třídy doInBackground je použita třída ApiResponse, která umožňuje předávat jak výsledná data, tak i případnou výjimku. Metody probíhají stejně jako bez ošetření chyb, ale v try i catch jsou definovány návratové hodnoty při úspěšném i neúspěšném spuštění. Při úspěšném aplikace nahraje data na hlavní vlákno, při neúspěšném dostaneme chybovou hlášku.

**Zdrojový kód 5/ Standardní knihovna - Práce s chybami**

```
@Override
protected ApiResponse doInBackground(String... params) {
    Call<List<DataModel>> call = myApi.callPosts();
    List<DataModel> posts;
    try {
        posts = call.execute().body();
        return new ApiResponse(posts);
    } catch (IOException e) {
        e.printStackTrace();
        return new ApiResponse(e.getMessage());
    }
}
@Override
protected void onPostExecute(ApiResponse dataModels) {
    super.onPostExecute(dataModels);
    try {
        callback.accept(dataModels);
    } catch (Exception e) {
        e.printStackTrace();
    }

}
```

Jako největší výhodu reaktivního programování v tomto případě lze označit fakt, že na ošetření chyb stačí vytvořit pouze dva Consumery, jeden typu list a druhý throwable, které předáme jako parametry metody subscribe, z nichž první je zavolán v případě, že práce na pozadí i popředí proběhla bez chyb a vrátila úspěšný výsledek, a druhy je zavolán v případě, že kdekoliv v celém Observable streamu nastala chyba. Consumer s listem slouží ke úspěšnému výsledku a throwable k zobrazení chybové hlášky. (viz. zdrojový kód 6)

**Zdrojový kód 6/ RxJava - Práce s chybami**

```
@Override
public void loadData(final Consumer<ApiResponse> callback) {
    myApi.observablePosts()
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(
             new Consumer<List<DataModel>>() {
                @Override
                public void accept(List<DataModel> dataModels) throws
Exception {
                    callback.accept(new ApiResponse(dataModels));
                }
            },
            new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    callback.accept(new
ApiResponse(throwable.getMessage()));
                }
            });
}
```

Razantní změna na první pohled vidět není, ale v tomto případě je nutné si uvědomit, že v případě RxJavy (viz. zdrojový kód 6) je potřeba chybu zpracovat jen jednou nezávisle na délce řetězce. Vždy stačí jen jeden Consumer, který chybu zpracuje. V případě několika AsyncTasků volaných po sobě (viz. zdrojový kód 5) musíme chybu vždy manuálně zpracovat a předávat dal. Navíc musíme řešit to, jak se zachovat, pokud chyba nastane uprostřed. Toto vše je v RxJave vyřešeno za nás. Objektivní výsledky měření jsou:

**Tabulka 3/ Vlastní výsledky - Výsledky měření práce s chybami**

| | **Standardní knihovna** | **Reaktivní programování** |
|---|---|---|
| **Počet řádků** | 23 řádků | 18 řádků |
| **Počet řádků přímo spojených s chybami** | 14 řádků | 11 řádků |
| **Rychlost výpočtu aplikace** | 1m 10s | 58 s |
| **Rychlost samotné operace** | 827 ms | 398 ms |
| **Velikost APK** | 4,48 mb | 4,69mb |

Z výsledků (viz. tabulka 3) lze vidět, že v tomto případě převažují výhody reaktivního programování. Ty se ukážou především v počtu řádků, rychlosti výpočtu aplikace a rychlosti samotné operace. O největším rozdílu se dá mluvit především u rychlosti samotné operace,

která je při zpracování v RxJava více než dvojnásobně rychlejší. Velikost APK je opět ovlivněna knihovnou RxJava.

### 4.2.4 Výhody paralelního běhu proti sekvenčnímu

Jak už bylo dříve zmíněno (viz. kapitola 3.3.9.3), asyncTask není na rozdíl od reaktivního programování standardně schopen paralelní exekuce. Jestliže spustíme dvě dlouhotrvající operace, druhá bude čekat na dokončení první a tím se doba exekuce aplikace zvýší. Reaktivní programování oproti tomu provede obě operace v jedné chvíli na jiných vláknech a tím se rychlost znatelně zvýší.

Jestliže se aplikace snaží o provedení většího počtu dlouhotrvajících procesů, je nutné je předem definovat. Jelikož jejich samotná exekuce nezáleží na tom, jestli budou provedeny v asyncTasku (viz. zdrojový kód 7) nebo rxJave (viz. zdrojový kód 8), můžeme použít náhodný proces, který bude trvat právě 10 sekund.

Nejdříve je vytvořena metoda, která simuluje dlouhotrvající volání. Pro konzistentní měření tato metoda na 10 vteřin uspí vlákno, na kterém právě běží. a následně o tom vypíše časové údaje. Na vlákně na pozadí (doInBackground) je následně definována proměnná, která má vlastnosti této metody přiřazené. U asyncTasku nemůže chybět ošetření chyb a výjimek, po kterém je na hlavním vlákně vypsán výsledek procesu s časem (onPostExecute), který slouží k uložení mezivýsledku. V asyncTasku bude dlouhý proces vypadat takto:

**Zdrojový kód 7/ AsyncTask – Dlouhotrvající proces**

```
static class LongTermAT extends AsyncTask<String, Void, String> {
    @Override
    protected String doInBackground(String... params) {
        try {
            long result = longMethodAt();
            return params[0] + " - success + "+result;
        } catch (Exception e) {
            return params[0] + " - failure + "+ e.getMessage();
        }
    }

    private long longMethodAt() throws InterruptedException {
        Thread.sleep(10000);
        return System.currentTimeMillis();
    }

    protected void onPostExecute(String posts) {
    }
}
```

V RxJavě na subscribe emitter vytvoří objekt, který následně s deseti sekundovým zpožděním vyšle. Tímto simulujeme dlouhotrvající operaci (viz. zdrojový kód 8).

**Zdrojový kód 8/ RxJava - Dlouhotrvající proces**

```
private Observable<Long> longMethod(String name) {
    return Observable.create(new ObservableOnSubscribe<Long>() {
        @Override
        public void subscribe(ObservableEmitter<Long> emitter) throws
Exception {
            emitter.onNext(System.currentTimeMillis());
        }
    }).delay(10, TimeUnit.SECONDS);
}
```

Následně přichází jejich volání (viz. zdrojový kód 9 a 10). V asyncTasku jsou vytvořeny dva proměnné a do nich je vložena metoda LongTermAt z předchozí ukázky. Ty jsou následně spuštěné a sekvenčně provedené.

**Zdrojový kód 9/ Standardní knihovna - Volání**

```
private void longTerm() {
    AsyncTask<String, Void, String> asyncTask1 = new LongTermAT();
    AsyncTask<String, Void, String> asyncTask2 = new LongTermAT();
    asyncTask1.execute("task 1");
    asyncTask2.execute("task 2");
}
```

V RxJave pro paralelní spuštění dvou operaci využijeme metodu zip (viz. obrázek 3). Metoda zip slouží ke kombinování výsledků více emitterů z různých Observables a následné vyslání položky za každou kombinaci (více na http://reactivex.io/documentation/operators/zip.html ). Následně je kombinace vrácena na hlavní vlákno pomocí .subscribe, jsou ošetřeny chyby a výjimky a výsledek je vypsán.

**Zdrojový kód 10/ RxJava - Volání**

```
private void observableLongTerm() {
    compositeDisposable.add(Observable.zip(longMethod("task 1"),
longMethod("task 2"), new BiFunction<Long, Long, String>() {
        @Override
        public String apply(Long aLong, Long aLong2) throws Exception {
            return "Zipped - " + aLong + ", " + aLong2;
        }
    }).subscribe(new Consumer<String>() {
        @Override
        public void accept(String s) throws Exception {
        }
    }, new Consumer<Throwable>() {
        @Override
        public void accept(Throwable throwable) throws Exception {
        }
    }));
}
```

Objektivní výsledky jsou:

**Tabulka 4/ Vlastní výsledky - Výsledky měření paralelního běhu proti sekvenčnímu**

| | Standardní knihovna | Reaktivní programování |
|---|---|---|

| | | |
|---|---|---|
| **Počet řádků** | 27 řádků | 28 řádků |
| **Rychlost výpočtu aplikace** | 1m 1s | 1m 4s |
| **Rychlost samotné operace** | 20 005 ms | 10 035 ms |
| **Velikost APK** | 4,69 mb | 4,69mb |

V tomto případě jsou výsledky (viz. tabulka 4) relativně vyrovnané, ačkoliv objektivně se dá říci, že standardní knihovna má výhodu ve vice kritériích. Jedná se o počet řádků a rychlost výpočet aplikace, ačkoliv i zde jsou výsledky téměř stejné. Za to obrovský rozdíl se nachází v případě rychlosti samotné operace, kde je operace vytvořená RxJavou o 9,07 vteřiny rychlejší, tedy téměř dvojnásobně. Doopravdy se tedy ukazuje rozdíl mezi synchronním a paralelním během.

### 4.2.5 Pravidelné opakování

Pravidelného opakování určitého procesu je možné dosáhnout jak za pomoci standartní knihovny, tak i AsyncTasku. Stejně tak je možné danou operaci udělat na hlavním vlákně, ačkoliv na pozadí je operace užitečnější, protože neblokuje uživatelské rozhraní. Pravidelné opakování procesu je v aplikaci použito pro obnovení výsledků. Aplikace tedy pravidelně provede obnovu dat a načte nová.

V AsyncTasku (viz. zdrojový kód 11) není zobrazena kompletní třída nahrávající data, ale jsou zobrazena pouze data relevantní pro tuto ukázku, tedy taková, která slouží k obnovení dat. To bude provedeno v metodě onPostExecute, kam jsou nahrávána data poté, co je dokončen výpočet na pozadí. Pro opakování je nutné vytvořit handler, pomocí kterého určíme čas, ve kterém se proces zopakuje a runnable, ve které je spouštěný samotný proces.

**Zdrojový kód 11/ Standardní knihovna - Pravidelné opakování**

```
    protected void onPostExecute(final ApiResponse dataModels) {
         super.onPostExecute(dataModels);
        final Handler handler = new Handler();
        Runnable refresh = new Runnable() {
            @Override
            public void run() {
        try {
            callback.accept(dataModels);
        } catch (Exception e) {
            e.printStackTrace();
        }
                handler.postDelayed(this, 10 * 1000);
            }
        };
        handler.postDelayed(refresh, 10 * 1000);
    }
```

V RxJavě je pro opakování (viz. zdrojový kód 12) relevantní celá metoda loadData, která slouží k nahrání výsledků ze zdroje. V té je vytvořena Observable, na které je pomocí metody interval nastavena doba, po které se bude proces opakovat. Následně jsou vstupní data vyrovnána do nového pole, ve kterém je seznam výsledků spuštěn. Poté probíhá přiřazení na jednotlivá vlákna a ošetření výjimek.

**Zdrojový kód 12/ RxJava - Pravidelné opakování**

```
public void loadData(final Consumer<ApiResponse> callback) {
    Observable.interval(10, TimeUnit.SECONDS).timeInterval()
            .flatMap(new Function<Timed<Long>,
ObservableSource<List<DataModel>>>() {
                @Override
                public ObservableSource<List<DataModel>>
apply(Timed<Long> longTimed) throws Exception {
                    return myApi.observablePosts();
                }
            })
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(
             new Consumer<List<DataModel>>() {
                @Override
                public void accept(List<DataModel> dataModels)
throws Exception {
                    callback.accept(new ApiResponse(dataModels));
                }
            },
            new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws
Exception {
                    callback.accept(new
ApiResponse(throwable.getMessage()));
                }
            });
}
```

Objektivní výsledky jsou:

**Tabulka 5/ Vlastní výsledky - Výsledky měření pravidelného opakování**

| | **Standardní knihovna** | **Reaktivní programování** |
|---|---|---|
| **Počet řádků** | 17 řádků | 29 řádků |
| **Počet řádků přímo spojených s opakováním** | 8 řádků | 9 řádků |
| **Rychlost výpočtu aplikace** | 58s 329ms | 53s 491ms |
| **Rychlost samotné operace** | 10 436 ms | 10 451 ms |
| **Velikost APK** | 4,69 mb | 4,69mb |

Výhody reaktivního programování při opakování procesu se příliš neprojevují (viz. tabulka 5). Naopak se dá říci, že standardní knihovna v tomto případě převyšuje RxJavu. Především poté v počtu řádků (12), ačkoliv počet řádků spojených s opakováním už tak rozdílný není. Rychlost samotné operace se dá označit za téměř stejnou, ačkoliv mírnou výhodu má standardní knihovna. Jedinou výhodou tedy zůstává rychlost výpočtu aplikace, kde je reaktivní programování o necelých 5 sekund rychlejší. Velikost APK je totožná.

# 5 Výsledky a diskuze

## 5.1 Výsledky z hlediska kritérií

Pomocí porovnání jednotlivých procesů byly určeny jednotlivé výhody reaktivního programování. V jednotlivých bodech lze výhody a nevýhody zobrazit takto:

### 5.1.1 Počet řádků

Z hlediska počtu řádků vychází výsledky vyrovnaně. Ačkoliv se nejedná o největší předpokládanou výhodu reaktivního programování, počet řádků je menší ve dvou z pěti zkoumaných procesů a v jednom je stejný jako u standardní knihovny. V celkovém součtu je řádků spojených s reaktivním programováním 81 a řádků spojených s AsyncTaskem 91.

Dá se tedy říci, že na pěti operacích bylo pomocí reaktivního programování ušetřeno 10 řádků oproti standardní knihovně. Největší počet řádků byl ušetřen při načtení dat z API (viz. tabulka 2) a to 20. Největší počet řádků oproti užití standardní knihovny bylo pravidelného opakování (viz. tabulka 5) a to 12. U ostatních operací jsou výsledky vyrovnanější, a to 3 řádky na obou stranách při zobrazení seznamu položek (viz. tabulka 1), 14 standardní knihovna ku 11 reaktivní programování při práci s chybami (viz. tabulka 3) a 27 standardní knihovna ku 28 reaktivní programování při sekvenčním/paralelním běhu (viz. tabulka 4).

### 5.1.2 Rychlost výpočtu aplikace

Z hlediska rychlosti výpočtu aplikace má navrch reaktivní programování, a to především v procesech, na které se zaměřuje. Výhoda reaktivního programování se ukazuje ve třech z pěti zkoumaných procesů. V celkovém součtu časů na procesech je použitím reaktivního programování ušetřeno 25 sekund. Největší rozdíl se ukazuje při načtení dat z API (viz. tabulka 2) a práci s chybami (viz. tabulka 3), kde je oproti standardní knihovně ušetřeno u obou 12 sekund. Nevýhodná čísla se ukazují u synchronní aktivity (viz. tabulka 1) a paralelního běhu (viz. tabulka 4), kde AsyncTask ušetří 1 a 3 vteřiny. Výpočet aplikace při opakujícím se procesu (viz. tabulka 5) je rychlejší o 5 sekund.

### 5.1.3 Rychlost samotné operace

Rychlost samotné operace je jedním z důležitějších ukazatelů pro reaktivní programování, i proto se jeho výhody ukazují hned u čtyř z pěti měřených procesů. V celkovém součtu reaktivní programování přesahuje výsledky standardní knihovny o 9611ms. To je ovlivněno především paralelním během aktivit (viz. tabulka 4), kde je standardní knihovna běží o téměř 10 sekund (9070 ms) déle. Dále je také větší rozdíl při načtení dat z API (viz. tabulka 2) a práci s chybami (viz. tabulka 3) a to 122 ms a 431 ms. U synchronní aktivity (viz. tabulka 1) se jedná pouze o rozdíl 3 ms, ale reaktivní programování je v tomto případě stále výhodnější. Jediný proces ukazující záporné výsledky je pravidelné opakování (viz. tabulka 5), kde AsyncTask předbíhá RxJavu o 15 ms.

### 5.1.4 Velikost APK

Posledním měřeným kritériem je velikost APK. Velikost APK je hodnota, ve které reaktivní programování nikdy neuspělo, neboť pracujeme s externí knihovnou RxJava, která nám velikost APK oproti standardní knihovně zvyšuje. Standardní knihovna v tomto případě převyšuje reaktivní programování ve třech z pěti sledovaných procesech a ve dvou jsou výsledky totožné. Největší nevýhody se ukazují v případě synchronní aktivity (viz. tabulka 1) a načtení dat z API (viz. tabulka 2) a to o 0,2. U práce s chybami (viz. tabulka 3) vede standardní knihovna o 0,1 mb. Výsledky opakování procesu (viz. tabulka 5) i paralelního běhu (viz. tabulka 4) jsou stejné.

## 5.2 Výsledky z hlediska testovaných procesů

Z hlediska procesů lze označit za nejvýhodnější načtení dat z API (viz. tabulka 2), kde reaktivní programování převyšuje standardní knihovnu ve třech ze čtyř kritérii. Jako další zobrazuje výhody reaktivního programování také práce s chybami (viz. tabulka 3), která převyšuje AsyncTask ve čtyřech z pěti hodnocených kritérii. U ostatních aktivit z hlediska počtu vítězných kritérii vyhrává standardní knihovna, ale vítězná kritéria reaktivního programování často zobrazují neporovnatelné rozdíly. Například v případě paralelních aktivit (viz. tabulka 4) sice standardní knihovna předbíhá RxJavu v poměru výhodných kritérii, ale rychlost operace ukazuje obrovské rozdíly. Reaktivní programování jednoznačně neuspěje v případě synchronního běhu (viz. tabulka 1) a pravidelného opakování procesu (viz. tabulka

5), i když zde se nachází kritéria, ve kterých reaktivní programování standardní knihovnu předčí.

# 6 Závěr

V první části bylo představeno reaktivní programování a RxJava jako přístup, který za vhodného použití zjednoduší samotný proces programování. Byla popsána i jeho alternativa AsyncTask a práce s ním. Nakonec byly nastíněny i jednotlivé situace, ve kterých by reaktivní programování mělo excelovat.

Tyto znalosti byly aplikovány při implementaci dvou ukázkových aplikací v praktické části, u kterých byla následně porovnána objektivní kritéria: počet řádků, rychlost výpočtu aplikace, rychlost samotné operace a velikost APK. Kritéria byla poměřena na synchronních i asynchronních aktivitách, konkrétně na zobrazení seznamu položek v aplikaci (viz. kapitola 4.2.1), načtení dat z API (viz. kapitola 4.2.2), práci s chybami (viz. kapitola 4.2.3), paralelním a sekvenčním běhu (viz. kapitola 4.2.4) a pravidelném opakování procesu (viz. kapitola 4.2.5).

Z objektivních výsledků, které byly získány poměřením, bylo doporučeno RxJavu používat především v případě načtení dat z API (viz. kapitola 4.2.2) a práci s chybami (viz. kapitola 4.2.3), kde objektivní výsledky převyšují výsledky standardní knihovny na většině měřených kritériích. V případě paralelního a sekvenčního běhu (viz. kapitola 4.2.4) byla RxJava doporučena, ačkoliv objektivní výsledky ve většině kritérií dopadly pozitivně spíše pro AsyncTask, protože kritérium rychlost samotného procesu je nesrovnatelně vyšší. Reaktivním programováním nebylo nedoporučeno provádět synchronní aktivitu zobrazení seznamu položek v aplikaci (viz. kapitola 4.2.1) a pravidelné opakování procesu (viz. kapitola 4.2.5), kde RxJava zaostává ve většině měřených kritériích (ačkoliv se najdou i kritéria, ve kterých RxJava stále AsyncTask převyšuje).

Z hlediska kritérií se největší výhody se ukazují při rychlosti samotné operace, kde reaktivní programování předčí standardní knihovnu o 9611 ms. Další výhoda se také ukazuje v případě rychlosti výpočtu aplikace, kde je ušetřeno přibližně 25 vteřin. Malou výhodu lze nalézt i v případě počtu řádků, kdy jsme na zobrazených procesech ušetřili 9 řádků. Za největší nevýhodu lze počítat velikost APK, která je nejen díky působení knihovny RxJava větší.

Z porovnání objektivních výsledků vyplývá, že použitím reaktivního programování na doporučených aktivitách lze dosáhnout stejné funkcionality pomocí méně investovaných prostředků než za pomocí standardní knihovny. To může vést k lepší čitelnosti kódu, vyšší rychlosti aplikace a jejímu plynulejšímu běhu.

# 7 Seznam použitých zdrojů


Android. Aktuálně.cz [online]. Economia, 2011 [cit. 2019-02-23]. Dostupné z: https://www.aktualne.cz/wiki/veda-a-technika/android-google/r~i:wiki:1424/?redirected=1540216787 .

Android Studio release notes. Developers.android.com [online]. Android, 2016 [cit. 2019-02-23]. Dostupné z: https://developer.android.com/studio/releases/ .

Asynchronous Programming Techniques. Kotlinlang [online]. Delaware: Kotlin Foundation, 2017 [cit. 2019-02-23]. Dostupné z: https://kotlinlang.org/docs/tutorials/coroutines/async-programming.html .

AsyncTask overview in Android. Bhargavaroyal.wordpress.com/ [online]. Unknown: bhargavaroyal, 2016 [cit. 2019-02-23]. Dostupné z: https://bhargavaroyal.wordpress.com/2016/05/03/asynctask/ .

BENEŠ, Miroslav. Procesy a vlákna. Katedra informatiky FEI VŠB-TU [online]. Ostrava: Katedra informatiky FEI VŠB-TU Ostrava, 2003 [cit. 2019-02-23]. Dostupné z: http://www.cs.vsb.cz/benes/vyuka/pte/texty/vlakna/ch01s01.html .

BOISNEY, Philippe. How Memory Leaks Happen in a Java ApplicationRxJava 2 | Android : How to properly handle rotations with ConnectableObservable & Loader. ProAndroidDev [online]. Unknown: Medium, 2017 [cit. 2019-02-23]. Dostupné z: https://proandroiddev.com/rxjava-2-android-how-to-proper-handle-rotations-with-connectableobservable-loader-9356ff47df61 .

Callback (computer programming). In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2019-02-23]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Callback_(computer_programming)&oldid=874031687 .

DUCROHET, Xavier. Android Studio: An IDE built for Android. Android developers blog [online]. Mountain View: Android Developers, 2013 [cit. 2019-02-23]. Dostupné z: https://android-developers.googleblog.com/2013/05/android-studio-ide-built-for-android.html.

DRILLER, Jens. RxJava Tidbits #2: Parallelism with Observable.zip(). Medium [online]. Unknown: Medium, 2016 [cit. 2019-02-23]. Dostupné z: https://medium.com/rxjava-tidbits/rxjava-tidbits-2-parallelism-with-observable-zip-6ef3c5a61a22 .

Error Handling Operators. GitHub [online]. San Francisco: GitHub, 2018 [cit. 2019-02-23]. Dostupné z: https://github.com/ReactiveX/RxJava/wiki/Error-Handling-Operators .
Funkcionální programování. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2019-02-23]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Funkcion%C3%A1ln%C3%AD_programov%C3%A1n%C3%AD&oldid=16858691 .
Get the Android SDK. Stuff.mit.edu [online]. Unknown: Android Developers, 2018 [cit. 2019-02-23]. Dostupné z: https://stuff.mit.edu/afs/sipb/project/android/docs/sdk/index.html .
GIRIDHAR, Chetan. How To Simplify Networking In Android: Introducing The Volley HTTP Library. Smashing magazine [online]. Freiburg: Smashing Media, 2017 [cit. 2019-02-23]. Dostupné z: https://www.smashingmagazine.com/2017/03/simplify-android-networking-volley-http-library/ .
HAMBRICK, Ross. Replace AsyncTask and AsyncTaskLoader with rx.Observable – RxJava Android Patterns. Stable Kernel [online]. Atlanta: Stable Kernel, 2014 [cit. 2019-02-23]. Dostupné z: https://stablekernel.com/replace-asynctask-and-asynctaskloader-with-rx-observable-rxjava-android-patterns/ .
HAVERBEKE, Marijin. Eloquent Javascript: A Modern Introduction to Programming [online]. 3. vydání. San Francisco: No Starch Press, 2018 [cit. 2019-02-23]. ISBN 13: 9781593279509. Dostupné z: https://eloquentjavascript.net/ .
HIBBLE, Adam. What is difference between Functional Reactive Programming, Functional Programming, and Reactive Programming?. Quora [online]. Brisbane: Quora, 2015 [cit. 2019-02-23]. Dostupné z: https://www.quora.com/What-is-difference-between-Functional-Reactive-Programming-Functional-Programming-and-Reactive-Programming .
How to articulate the difference between asynchronous and parallel programming?. Stack Oveflow [online]. 2011 [cit. 2019-02-23]. Dostupné z: https://stackoverflow.com/questions/6133574/how-to-articulate-the-difference-between-asynchronous-and-parallel-programming .
Interface Observer<T>. ReactiveX [online]. Unknown: ReactiveX, 2013 [cit. 2019-02-23]. Dostupné z: http://reactivex.io/RxJava/javadoc/io/reactivex/Observer.html .
Mobile UI (mobile user interface). Tech Target [online]. Newton: SearchMobileComputing, 2015 [cit. 2019-02-23]. Dostupné z: https://searchmobilecomputing.techtarget.com/definition/mobile-UI-mobile-user-interface .

Nástroje na vývoji aplikací pro Android. Elitec software [online]. Uherský Brod: Android Developers, 2016 [cit. 2019-02-23]. Dostupné z: http://www.elitecsoftware.cz/nastroje-na-vyvoji-aplikaci-pro-android/ .

NEUPANE, Abinash. Why You Should Use AsyncTask in Android Development. Upwork [online]. Mountain View: Upwork, 2017 [cit. 2019-02-23]. Dostupné z: https://www.upwork.com/hiring/mobile/why-you-should-use-asynctask-in-android-development/ .

Observable. ReactiveX [online]. Unknown: ReactiveX, 2013 [cit. 2019-02-23]. Dostupné z: http://reactivex.io/documentation/observable.html .

OSTROVSKY, Igor. Parallel Programming with .NET: Most Common Performance Issues in Parallel Programs. Microsoft blogs [online]. Unknown: Igor Ostrovsky, 2008 [cit. 2019-02-23]. Dostupné z: https://blogs.msdn.microsoft.com/pfxteam/2008/08/12/most-common-performance-issues-in-parallel-programs/ .

PARASCHIV, EUGEN. How Memory Leaks Happen in a Java Application. Stackify [online]. Leawood: Stackify, 2017 [cit. 2019-02-23]. Dostupné z: https://stackify.com/memory-leaks-java/.

Processes and threads overview. Developers.android.com [online]. Android, 2016 [cit. 2019-02-23]. Dostupné z: https://developer.android.com/guide/components/processes-and-threads .

Reactive Manifesto [online]. Reactive Manifesto, 2014 [cit. 2019-02-23]. Dostupné z: https://www.reactivemanifesto.org/ .

ReactiveX. ReactiveX [online]. Unknown: ReactiveX, 2013 [cit. 2019-02-23]. Dostupné z: http://reactivex.io/intro.html .

Reaktivní programování. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2019-02-23]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Reaktivn%C3%AD_programov%C3%A1n%C3%AD&oldid=16047927 .

Real time programming: Special purpose or general purpose languages. Inria [online]. Valbonne: Inria, 1989 [cit. 2019-02-23]. Dostupné z: http://www-sop.inria.fr/members/Gerard.Berry/Papers/Berry-IFIP-89.pdf .

ROY, Aritra. Multi-Threading Like a Boss in Android With RxJava 2. Gojek [online]. Unknown: Medium, 2017 [cit. 2019-02-23]. Dostupné z: https://kotlinlang.org/docs/tutorials/coroutines/async-

programming.htmlhttps://blog.gojekengineering.com/multi-threading-like-a-boss-in-android-with-rxjava-2-b8b7cf6eb5e2 .
Rozdíl mezi asynchronním a paralelním kódem. Jiří Knesl [online]. Telč: Jiří Knesl, 2014 [cit. 2019-02-23]. Dostupné z: http://www.knesl.com/rozdil-asynchronni-paralelni-kod .
SAILAPPAN, Vijay. Using RxAndroid to fix callback hell. Medium [online]. Unknown: Medium, 2016 [cit. 2019-02-23]. Dostupné z: https://medium.com/@Vijay_S/using-rxandroid-to-fix-callback-hell-feed172118f5?fbclid=IwAR3HuGtzlxkcsKGPnGq1-cbyfm2GQseEOPZR-6LY7lksvrMky42bMEO5hds .
SERDARU, Silviu. 15 ANDROID APP DEVELOPMENT TOOLS ESSENTIAL FOR EVERY DEVELOPER'S TOOLBOX. Optasy [online]. New York: Android, 2018 [cit. 2019-02-23]. Dostupné z: https://www.optasy.com/blog/15-android-app-development-tools-essential-every-developers-toolbox .
SHEKHAR, Amit. What are Coroutines in Kotlin?. MindOrks [online]. Unknown: MindOrks, 2017 [cit. 2019-02-23]. Dostupné z: https://blog.mindorks.com/what-are-coroutines-in-kotlin-bf4fecd476e9 .
SHVARTS, James. Understanding RxJava subscribeOn and observeOn. ProAndroidDev [online]. Unknown: Medium, 2017 [cit. 2019-02-23]. Dostupné z: https://proandroiddev.com/understanding-rxjava-subscribeon-and-observeon-744b0c6a41ea .
SINGH, Rohit. RxJava Parallelization Concurrency : Zip() Operator. AndroidPub [online]. Unknown: Medium, 2018 [cit. 2019-02-23]. Dostupné z: https://android.jlelse.eu/rxjava-parallelization-concurrency-zip-operator-fe87a36441ff .
SINKAL, Ankhit. Understanding of AsyncTask in Android. Tech Target [online]. Unknown: Medium, 2017 [cit. 2019-02-23]. Dostupné z: https://medium.com/@ankit.sinhal/understanding-of-asynctask-in-android-8fe61a96a238 .
STONE, Deborah L., Caroline JANETT a Mark WOODROFFE. User interface design and evaluation. Boston, Mass.: Morgan Kaufmann, 2005. ISBN ISBN9780120884360.
STONE, Sydney. Android Studio: An IDE built for Android. Altex Soft [online]. Carlsbad: Altex Soft, 2018 [cit. 2019-02-23]. Dostupné z: https://www.altexsoft.com/blog/engineering/top-20-tools-for-android-development/ .
Synchronizing Network Calls With RxJava. Medium [online]. Unknown: Medium, 2017 [cit. 2019-02-23]. Dostupné z: https://medium.com/@DoorDash/synchronizing-network-calls-with-rxjava-d30f7db66fb9 .

TAMADA, Ravi. Android Introduction To Reactive Programming – RxJava, RxAndroid. Android Hive [online]. AndroidHive, 2018 [cit. 2019-02-23]. Dostupné z: http://reactivex.io/intro.htmlhttps://www.androidhive.info/RxJava/android-getting-started-with-reactive-programming/ .

TAMADA, Ravi. RxJava Understanding Observables. AndroidHive [online]. Unknown: Android Hive, 2018 [cit. 2019-02-23]. Dostupné z: https://www.androidhive.info/RxJava/rxjava-understanding-observables/ .

What's different in 2.0. GitHub [online]. San Francisco: GitHub, 2007 [cit. 2019-02-23]. Dostupné z: https://github.com/ReactiveX/RxJava/wiki/What's-different-in-2.0 .