Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Porovnání jazyků Java a Kotlin

Bakalářská práce

Autor: Daniel Schmid
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Hradec Králové duben 2021

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 26.4.2021

Daniel Schmid

Poděkování:

Děkuji doc. Mgr. Tomášovi Kozlovi, Ph.D. za odborné vedení práce, za rady a věcné připomínky při vypracovávání bakalářské práce.

Anotace

Tato práce slouží k přiblížení programovacího jazyka Kotlin a k jeho porovnání s programovacím jazykem Java, konkrétně při vývoji mobilních aplikací pro systém Android. Snaží se zachytit těžkopádnost Javy v situacích, které jsou v Kotlinu řešeny zcela automaticky v pozadí, s výhodou přehlednějšího kódu a méně náročnějšího vývoje aplikací pro vývojáře. Praktickou částí jsou dvě funkcionalitou totožné Android aplikace, kdy jedna je naprogramována v Javě a druhá v Kotlinu. Výstupem této praktické části budou metriky ve formě počtu znaků a počtu řádků, na kterých bude vidět možná úspora při používání Kotlinu, a porovnání velikostí některých klíčových souborů.

Annotation

Title: Comparison of Java and Kotlin languages

This bachelor's thesis serves to approach the programming language Kotlin and to compare it with programming language Java specifically in the development of a mobile applications for Android. It tries to show the cumbersomeness of Java in situations that are solved completely automatically in Kotlin which make the clearer code and less demanding application development for developers. The practical part of bachelor's thesis are two Android applications with same functionality. One application is programmed in Java and the other in Kotlin. The output of this practical part will be metrics in the form of the number of characters and the number of lines which will show the potential savings in the use of Kotlin and comparison of the sizes of some crucial files.

Obsah

1 Úvod ... 1

2 Motivace a cíl práce ... 2

3 Základní charakteristika ... 3

4 Syntaxe ... 4

4.1 Proměnné a datové typy ... 4

4.1.1 Null-safety ... 5

4.2 Porovnávání výrazů ... 10

4.2.1 Porovnávání textových řetězců a objektů ... 10

4.2.2 Vícenásobné větvení ... 11

4.3 Pole a kolekce ... 13

4.3.1 Kolekce ... 13

4.3.2 Pole ... 15

4.4 Cykly ... 17

4.5 Metody ... 20

4.5.1 Přístupové metody ... 21

4.6 Třídy ... 23

4.6.1 Abstraktní třídy a rozhraní ... 26

4.6.2 Enum třídy ... 29

4.6.3 Sealed třídy ... 30

4.6.4 Datové třídy ... 33

4.7 Rozšiřující funkce ... 37

4.8 Chytré přetypování ... 37

4.9 High-order funkce ... 38

5 Praktická část ... 39

5.1 Android ... 39

5.1.1 Popis prostředí Android Studio 43
5.1.2 Metriky 44
5.1.3 Velikosti souborů 49
5.2 Spring Boot 52
6 Shrnutí výsledků 53
7 Závěry a doporučení 55
8 Seznam použité literatury 57
9 Seznam použitých zdrojů 58
10 Přílohy 59

Seznam obrázků, tabulek a grafů

Obrázek 1 Diagram kolekcí v Kotlinu. 14
Obrázek 2 Návrhový vzor Model-View-Controller 41
Obrázek 3 Návrhový vzor Model-View-ViewModel 41
Obrázek 4 Struktura aplikace 43
Obrázek 5 Velikost obsahu složek uvnitř src složky v Javě 49
Obrázek 6 Velikost obsahu složek uvnitř src složky v Kotlinu 49
Obrázek 7 Výpis balíčků a tříd získaných z APK (Java) 51
Obrázek 8 Výpis balíčků a tříd získaných z APK (Kotlin) 51

Tabulka 1 Počet znaků u jednotlivých tříd v Kotlinu a Javě 45
Tabulka 2 Procentuální úspora v počtu znaků podle typu třídy 46
Tabulka 3 Počet řádků u jednotlivých tříd v Kotlinu a v Javě 47
Tabulka 4 Procentuální úspora v počtu řádků podle typu třídy 48
Tabulka 5 Velikost balíčků v závislosti na programovacím jazyku v kilobajtech 50
Tabulka 6 Počet znaků bytecodu a procent. zvětšení v Kotlinu oproti Javě 52

Graf 1 Počet znaků podle typu třídy v Kotlinu a v Javě 46
Graf 2 Počet řádků podle typu třídy v Kotlinu a v Javě 48

Seznam ukázek

Ukázka 4.1-a Deklarace proměnných v Javě 5
Ukázka 4.1-b Deklarace proměnných v Kotlinu 5
Ukázka 4.1-c Funkce ošetřující null hodnoty v Javě 8
Ukázka 4.1-d Použití anotace a Objects API 8
Ukázka 4.1-e Použití Optional API 9
Ukázka 4.1-f Ukázka práce s nenulovými a nulovými datovými typy v Kotlinu 9
Ukázka 4.1-g Stejná funkce jako v ukázce 4.1-c s použitím SafeCall operátoru 9
Ukázka 4.1-h Použití ?.let 9
Ukázka 4.2-a Porovnání Stringů v Javě 10
Ukázka 4.2-b Porovnávání Stringů v Kotlinu 10
Ukázka 4.2-c Switch v Javě s rozmezím hodnot 12
Ukázka 4.2-d When v Kotlinu s rozmezím hodnot 12
Ukázka 4.3-a Práce s listem v Javě 14
Ukázka 4.3-b Práce s Listem a MutableListem v Kotlinu 15
Ukázka 4.3-c Práce s polem v Javě 16
Ukázka 4.3-d Práce s polem v Kotlinu 16
Ukázka 4.4-a Práce s cykly v Javě 19
Ukázka 4.4-b Práce s cykly v Kotlinu 19
Ukázka 4.5-a Jednoduchá funkce v Javě 20
Ukázka 4.5-b Jednoduchá funkce a funkce bez návratové hodnoty v Kotlinu 21
Ukázka 4.5-c Přístupové metody v Javě 22
Ukázka 4.5-d Přístupové metody v Kotlinu 22
Ukázka 4.6-a Třída rozšiřující třídu a implementující rozhraní v Javě 25
Ukázka 4.6-b Třída rozšiřující třídu a implementující rozhraní v Kotlinu 26
Ukázka 4.6-c Vytvoření abstraktní třídy a rozhraní v Javě 28
Ukázka 4.6-d Vytvoření abstraktní třídy a rozhraní v Kotlinu 28
Ukázka 4.6-e Enum třída v Javě 29
Ukázka 4.6-f Enum třída v Kotlinu 30
Ukázka 4.6-g Sealed třída v Javě 32

Ukázka 4.6-h Sealed třída v Kotlinu.....33
Ukázka 4.6-i Třída v Javě se stejnou funkcionalitou jako v ukázce 4.6-k.....35
Ukázka 4.6-j Record třídy v Javě (od verze 14).....36
Ukázka 4.6-k Datové třídy v Kotlinu se dvěma konstruktory a přístupovými metodami.....36
Ukázka 4.7-a Využití rozšiřující funkce v Kotlinu.....37
Ukázka 4.8-a Chytré přetypování v Kotlinu.....37
Ukázka 4.9-a Jednoduchá high-order funkce v Kotlinu.....38

1 Úvod

Tato práce slouží ke srovnání dvou podobných, i když syntaxí rozdílných, programovacích jazyků Java a Kotlin. Práce bere v potaz základní znalosti programování v jakémkoli programovacím jazyku pro lepší pochopení problematiky. Primárně je snahou ukázat efektivnost Kotlinu při programování v porovnání s Javou, kdy Kotlin má některé procesy, které je nutné v Javě vykonávat vždy manuálně, zautomatizované. V jednotlivých kapitolách práce bude popis s ukázkou kódu v Javě, následovaný ukázkou kódu se stejnou funkcionalitou v Kotlinu, s vysvětlením a poukázáním na rozdíly mezi těmito ukázkami.

2 Motivace a cíl práce

Proč porovnávat zrovna Javu s Kotlinem? Kotlin je totiž velmi často komunitou vývojářů považován za nástupce Javy. A jelikož se Java drží na předních příčkách popularity programovacích jazyků a Kotlin je za předchozí dva roky, co se popularity týče, nejrychleji rostoucím jazykem, není snad lepších adeptů k porovnání [1].

Konkrétně pak dle IEEE SPECTRUM získala Java v roce 2020 skóre 95.3. Díky tomu se v kategorii pro vývoj webových aplikací umístila na druhém místě a v kategorii pro vývoj mobilních aplikací na místě prvním. Ve stejných kategoriích patří Kotlin vždy do TOP 10 programovacích jazyků. Dle IEEE SPECTRUM dosáhl skóre 57.8, což ho za kategorii pro vývoj webových aplikací řadí na desáté místo a za kategorii pro vývoj mobilních aplikací na místo sedmé [2].

Proto spousta vývojářů řeší problém, který z těchto jazyků pro vývoj, ať už mobilních či webových aplikací používat. Mají-li zvolit klasiku v podobě Javy, nebo poměrně nový jazyk Kotlin, který s sebou přináší řadu výhod. K rozhodování, který z těchto dvou programovacích jazyků použít, může přispět i tato práce.

3 Základní charakteristika

Ačkoliv se může zdát, že se jedná o konkurenční jazyky, tak tomu tak rozhodně není. Kotlin stejně jako Java využívají Java Virtual Machine (JVM) pro převod kódu do Java bytecodu [3]. Stejně tak jsou oba jazyky zcela interoperabilní, což znamená, že pokud je aplikace vyvíjena v Kotlinu, je možné využívat knihoven Javy, to samé platí i opačně [3]. Dokonce lze v rámci jednoho projektu kombinovat oba jazyky. Mnoho IDE například IntelliJ IDEA či Android Studio, které je rádoby odnoží IntelliJ IDEA zaměřenou na vývoj pro mobilní telefon, má vestavěnou funkci pro převod Java kódu do Kotlinu [4].

Oba jazyky jsou staticky typované, tudíž je nutné definovat datový typ proměnných či návratových hodnot funkcí [5]. Datové typy v Kotlinu nejsou zcela totožné jako v Javě, kde lze primární typy deklarovat pomocí klíčového slova nebo pomocí instance objektu, jelikož pro primární typy se v Kotlinu používají pouze objekty [5] [6].

Za příjemné, avšak ne zcela nutné zjednodušení, lze u Kotlinu považovat i absenci středníků za jednotlivými operacemi. Při tvorbě instancí v Kotlinu oproti Javě není nutné používat klíčové slovo *new*.

Pro oba jazyky lze díky JVM najít obrovské uplatnění. Je možné v nich psát desktopové aplikace a webové aplikace s pomocí podpůrných knihoven, které oba jazyky podporují, například Spring nebo Vert.x [4]. Při vývoji webových aplikací bude jistě užitečná i funkcionalita Kotlinu, která dokáže zkompilovat zdrojové kódy do JavaScriptu [4]. Velké využití má při vývoji aplikací pro mobilní telefony se systémem Android, kde byla Java dlouho oficiálním jazykem, nicméně byla společností Google nahrazena právě Kotlinem [6].

4 Syntaxe

Syntaxe je jedním z faktorů, ve kterém se tyto jazyky odlišují. Pokud se však Java vývojář snaží porozumět kódu v Kotlinu, nebude to pro něho příliš velký problém. Kotlin má sice jiné zvyklosti při deklaraci proměnných a vytváření funkcí, občas je i možné narazit na nějaké neznámé klíčové slovo či zjistit, že některá klíčová slova, která v Javě mají nějakou funkci, v Kotlinu vůbec neexistují. Rozdíly však nejsou natolik velké, aby nebylo možné pro Java programátora pochopit kód. Bude-li se snažit programovat v Kotlinu, chce to nějaký čas, než si zvykne na jiný zápis a přestane psát kód, který není v Kotlinu nutný, jelikož se o to postará sám kompilátor. Jak již bylo zmíněno, v Kotlinu se nepoužívají některá klíčová slova, konkrétně *new, static, final, void, implements, extends.* Proč tomu tak je, čím se klíčová slova nahradila a jak se kód při jejich absenci chová, je probráno detailněji v následujících kapitolách.

4.1 Proměnné a datové typy

Java

V Javě je zvykem při deklaraci proměnných uvést datový typ, následně název proměnné a v případě inicializace za rovnítko hodnotu proměnné. Datový typ lze u primárních datových typů určit klíčovým slovem (*int, double, boolean, …*) či třídou. Pro víceslovné názvy proměnných se dodržuje notace camelCase.

Kotlin

V Kotlinu slouží pro deklaraci proměnné klíčová slova *val* a *var*, za nimi následuje název proměnné (též notace camelCase), dvojtečka a datový typ. V případě inicializace se opět použije rovnítko a za ním hodnota. Jak již bylo zmíněno, Kotlin má definované datové typy pouze formou tříd, *tedy Integer, Double, Boolean, String* atd.

Val udává, že se bude jednat o proměnnou s neměnnou hodnotou, tedy o konstantu. Tato proměnná musí být inicializována ihned po deklaraci.

Var naopak slouží pro proměnné, kde se hodnota uvnitř může měnit. Tato proměnná musí být též inicializována, nicméně není nutné, aby to bylo ihned po deklaraci. V případě atributů třídy, by však měla být hodnota inicializovaná ihned, případně v konstruktoru třídy. Je však možné pomocí klíčového slova *lateinit* zaručit kompilátoru, že dojde k inicializaci proměnné později.

Díky typové inferenci v Kotlinu je v případě inicializace při deklaraci dokonce možné neuvádět datový typ, jelikož je automaticky rozpoznán na základě zadané hodnoty.

```
int number1 = 1;
Integer number2; //deklarace skrze objekt

String camelCase; //String lze deklarovat pouze skrz objekt
```

Ukázka 4.1-a Deklarace proměnných v Javě
Zdroj: Autor

```
var number1: Int = 1; //datové typy jsou definovány jako objekty
var number2 = 2; //typová inference (není nutné uvádět datový typ)
number2 = 3; //var lze měnit

val constant = 4;
constant = 5; //val nelze měnit jedná se o konstantu
```

Ukázka 4.1-b Deklarace proměnných v Kotlinu
Zdroj: autor

4.1.1 Null-safety

NullPointerException je záležitost, která dokáže potrápit leckterého programátora. Velmi často se nejde vyhnout velkému množství podmínek na ošetření objektů ohledně nedefinované hodnoty. Null-safety tedy udává, jak si jazyk dokáže poradit s nenadefinovanými objekty, respektive s prevencí proti vzniku *runtime výjimky*. Dá se říci, že princip spočívá v donucení programátora ošetřit možné vzniky *NullPointerException* a v možnostech, který mu k tomu jazyk poskytne.

Java

Java bohužel null-safety vlastností sama o sobě nedisponuje. Je proto nutné dávat si větší pozor na vznik těchto výjimek. Ošetření je tím pádem pouze v rukou programátora. Velmi často se objekty ošetřují obyčejnou podmínkou, díky které se může kód stát méně čitelný. Nicméně jsou k dispozici i další možnosti.

Jednou z nich je použití *Optional API*, které je dostupné od Javy 8. Princip spočívá v tom, že se hodnota obalí do *Optional* objektu, který si hlídá, zda je hodnota nadefinovaná. V praxi se pro zjištění, zda je výsledek dostupný, použije metoda *isPresent()* a následně lze pomocí *get()* tento výsledek získat. Výhoda je v tom, že je programátor upozorněn, pokud se před použitím *get()* nedotázal na dostupnost výsledku [7].

Další možností je *Objects API*, které definuje několik užitečných metod. Nejpodstatnější je *requireNonNull()*. Tato metoda sice neošetří vznik výjimky, ale dokáže ji při použití *requireNonNull()* ihned vyvolat. Je vhodná, když se provádí více operací a je potřeba, aby byly provedeny všechny. Tudíž se nestane, aby se provedla pouze polovina operací a následně se vyvolala výjimka, a tím došlo k nekonzistentnímu stavu. Dalšími funkcemi jsou pak *isNull()* či *nonNull()*. Jedná se však pouze o predikáty, ale zápis pomocí těchto metod vypadá mnohem lépe než *if(object != null)* [7].

Poslední, zde zmíněnou, možností je použití anotací, které fungují obdobně jako *Objects API* a umožňují nejkratší zápis ze všech zmíněných metod. Anotace *@NotNull* či *@Nullable* se umístí před parametr nějaké metody. Pokud se stane, že při volání této metody se při použití anotace *@NotNull* předá v parametru *null*, dojde opět k okamžitému vyvolání výjimky [7].

Kotlin

Kotlin má možnosti, jak ohlídat proměnné na výskyt *null*, a díky tomu je velmi zajímavý. Je totiž možné lépe predikovat, které proměnné mohou tohoto stavu nabýt a je nutné je ošetřit, a které nikoliv. Standardně, pokud se deklaruje proměnná, například typu *String*, považuje se za nenulovou proměnnou. Není proto možné, aby se v takto vytvořené proměnné vyskytla nenadefinovaná hodnota, a není třeba nic ošetřovat.

Co když je ale potřeba proměnná, do které lze nahrát *null*? Třeba kvůli funkci, která může vracet *null*, pokud se jí nepodaří dosáhnout požadovaného výsledku? V tomto případě stačí přidat otazník (*?*) rovnou za datový typ při deklaraci. Díky tomu bude možné, aby proměnná přijímala i hodnotu *null*. Kde je však ta výhoda, když je možné proměnné přetypovat na *nullable*, a díky tomu může opět vzniknout *NullPointerException*? Jelikož je každá vytvořená proměnná standardně nenulová, dokáže si kompilátor pohlídat, které proměnné mohou obsahovat *null*, a je tedy nutné je ošetřit před tím, než se s nimi bude pracovat.

Jenže hromada *if (variable != null)* podmínek není zrovna něco, co vypadá pěkně. Navíc při větším množství podmínek se díky *if blokům* a složeným závorkám může zhoršovat i čitelnost kódu. I na toto vývojáři Kotlinu mysleli a vytvořili tzv. Safe call operátor, Elvis operátor a let funkci.

Safe call operátor (*?.* – otazník tečka) pokračuje při volání funkce ve zpracování jen tehdy, když je výraz (objekt, proměnná) před operátorem nenulový. Pokud se má zavolat metoda nějakého objektu a objekt je nulový, pak se tato metoda nevykoná, a tudíž nehrozí žádná výjimka.

Elvis operátor (*?:* - otazník dvojtečka) vykoná výraz za operátorem pouze, pokud je výraz před operátorem nulový. To znamená, že pokud je potřeba dostat hodnotu nějakého atributu objektu, který může být nulový, pak je možné za tento operátor nastavit defaultní hodnotu. Ta se použije vždy, když je atribut nulový. V opačném případě se použije hodnota atributu.

Let funkce sama o sobě slouží pouze pro vykonání funkcí, které se umístí do *let bloku*. Avšak s kombinací se safe call operátorem se z ní stává velmi praktická věc. Jak již bylo vysvětleno výše, safe call operátor vykoná výraz jen tehdy, když je objekt nenulový. Stejné je to i u *?.let*. V praxi tedy *?.let* nahradí podmínku *if (object != null)* či velké množství safe call operátorů při častém dotazování se na vlastnosti nebo metody objektu.

Obdobně jako v Javě lze využít i *assertNotNull*, a to pomocí dvou vykřičníků (*!!*) za výrazem, který nesmí být nulový.

```
public String getPhoneNumber(Person person){
    if(person != null){
        Contact contact = person.getContact();
        if(contact != null){
            return contact.getPhoneNumber();
        }
    }
    return null;
}
```

Ukázka 4.1-c Funkce ošetřující null hodnoty v Javě
Zdroj: autor

```
public String getPhoneNumber(@NotNull Person person){ //použití anotace
    Contact contact = person.getContact();
    Objects.requireNonNull(contact); //použití Objects API
    return contact.getPhoneNumber();
}
```

Ukázka 4.1-d Použití anotace a Objects API
Zdroj: autor

```java
public Optional<String> getPhoneNumber(@NotNull Person person){ //anotace
    Contact contact = person.getContact();
    if(contact != null) {
        return Optional.of(contact.getPhoneNumber());
    }
    return Optional.empty();
}

//volání metody
Optional<String> phoneNumber = getPhoneNumber(person);
if(phoneNumber.isPresent()) //nejdříve se dotázat, zda je hodnota
nastavena
    System.out.println(phoneNumber.get());
```

Ukázka 4.1-e Použití Optional API
Zdroj: autor

```kotlin
var ex: String? = null
val exLenght1: Int = ex.length //nelze, jelikož ex může být null

val exLenght2: Int = ex?.length //Safe call operátor
// nelze, jelikož výraz vrací hodnotu dat. typu Int?

val exLenght3: Int = ex?.length ?: -1 //Elvis operátor
// vrátí výraz za operátorem, pokud je výraz před operátorem null
//výraz tedy vrátí -1

var exLenght4: Int? = ex?.length; //lze jelikož proměnná je datového typu
Int?
//jelikož ex je null, tak i exLenght4 bude null
```

Ukázka 4.1-f Ukázka práce s nenulovými a nulovými datovými typy v Kotlinu
Zdroj: autor

```kotlin
fun getPhoneNumber(person: Person?) = person?.contact?.phoneNumber
```

Ukázka 4.1-g Stejná funkce jako v ukázce 4.1-c s použitím SafeCall operátoru
Zdroj: autor

```kotlin
var p: Person? = Person("Karel", "Kryl")

p?.let {
    println("Osoba: ${it.firstName} ${it.lastName}. Narozen ${it.birthDay}.
(${it.age} let)")
    } //lepší způsob než podmínka na != null či opakované používání safe
call operátoru
```

Ukázka 4.1-h Použití ?.let
Zdroj: autor

4.2 Porovnávání výrazů

4.2.1 Porovnávání textových řetězců a objektů

Java

V Javě je nutné pro porovnání objektů vytvořit metodu *equals(),* nebo si ji nechat vygenerovat vývojovým prostředím. To úzce souvisí i s textovými řetězci, tedy s datovým typem *String*. Java programátoři dobře vědí, že když chtějí v Javě porovnávat dva textové řetězce, nemůžou využít dvou rovnítek (==), jelikož se neporovnávají hodnoty řetězců, ale jejich instance. Musí proto použít zmíněnou metodu *equals(),* což může být pro začínající programátory poněkud matoucí.

Kotlin

Pro porovnání objektů v Kotlinu se u datových tříd *equals()* vygeneruje sám (více o datových třídách v kapitole 4.6.4). V Kotlinu je upraveno i porovnávání *Stringu*, tak aby mohl vývojář využít dvou rovnítek. Pokud by však někdo potřeboval porovnat instance textových řetězců, není to vůbec problém. Stačí použít tři rovnítka (===), které mají v Kotlinu stejný význam jako v Javě rovnítka dvě (==).

```
String text1 = "text";
String text2 = "text";

if(text1.equals(text2)) //při porovnávání Stringů v Javě by se měla
používat metoda equals()
```

Ukázka 4.2-a Porovnání Stringů v Javě
Zdroj: autor

```
var text1 = "text";
var text2 = "text";

if(text1 == text2) //bez problému se může použít binární operátor
if(text1.equals(text2)) //nicméně lze použít i tento způsob
```

Ukázka 4.2-b Porovnávání Stringů v Kotlinu
Zdroj: autor

4.2.2 Vícenásobné větvení

Java

Pro vícenásobné větvení se v Javě může využít *switch* neboli přepínač. Porovnávaná proměnná se zapíše do závorek za *switch* a následně se v těle *switche* pomocí *case* vytvářejí jednotlivé případy (možnosti), kterých může proměnná nabýt. Za *case* s konkrétní hodnotou následuje dvojtečka (*:*) a poté potřebný kód ukončený klíčovým slovem *break*. Pokud je třeba, může se nadefinovat, co se má stát, když žádná z uvedených hodnot neodpovídá, a to pomocí *default*.

A co když bude nutné vykonat stejný kód pro nějaké rozmezí hodnot? Je to sice možné, ale poněkud zdlouhavé. V tomto případě by se totožný kód pro hodnoty 1, 2, 3 napsal v Javě následovně: „*case1: case2: case3: …*“. Je tedy patrné, že je zápis zdlouhavý, a pokud by bylo potřeba provádět stejný kód pro větší rozmezí, vyplatí se spíše využít standardní podmínku, nebo si vytvořit pomocnou metodu, která bude vyhodnocovat, zda je hodnota v daném rozmezí.

Kotlin

Co se týče větvení podmínek, hledal by Java vývojář v Kotlinu *switch* marně. K tomu tu totiž slouží klausule *when*, kdy je použití obdobné jako u *switche* v Javě. Porovnávaná proměnná též patří do závorek za *when*. Různí se hlavně v tom, že se nepoužívá klíčové slovo *case*. V Kotlinu se totiž rovnou uvede hodnota nebo nějaký výraz a místo dvojtečky se od kódu oddělí ukazatelem neboli šipkou (->). Stejně tak odpadá nutnost použití *break*. Kód se v případě jednořádkové operace píše rovnou za ukazatel. Pokud by bylo operací více, pouze se všechny umístí do složených závorek, čímž je zřejmé, kde každá větev končí. Pokrytí možnosti, že hodnota nebude odpovídat žádnému z případů, lze učinit pomocí *else*.

Kotlin si na rozdíl od Javy umí poradit i s rozmezím hodnot. Poslouží k tomu operátor *in* a jazykový konstrukt *range*, který se zapisuje pomocí dvou teček (..) či pomocí *rangeTo().* Obdobný příklad, který byl zmíněn u Javy, by se v Kotlinu napsal následovně: „*in 1..3 -> …“.*

Kód je na první pohled mnohem kratší a přehlednější než v Javě. Klíčové však je, že lze stejným způsobem pokrýt i mnohem větší rozmezí bez použití pomocné metody.

```
int value = 0;
switch(value){
    case 1:
    case 2:
    case 3://CODE
        break;
    case 4: //CODE
        break;
    default: //CODE
        break;
}
```

Ukázka 4.2-c Switch v Javě s rozmezím hodnot
Zdroj: autor

```
var value = 0;
when(value){
    in 1..3 -> //CODE
    in 4.rangeTo(7) -> //CODE, lze použít i rangeTo()
    8-> //CODE
    else -> //CODE
}
```

Ukázka 4.2-d When v Kotlinu s rozmezím hodnot
Zdroj: autor

4.3 Pole a kolekce

4.3.1 Kolekce

Java

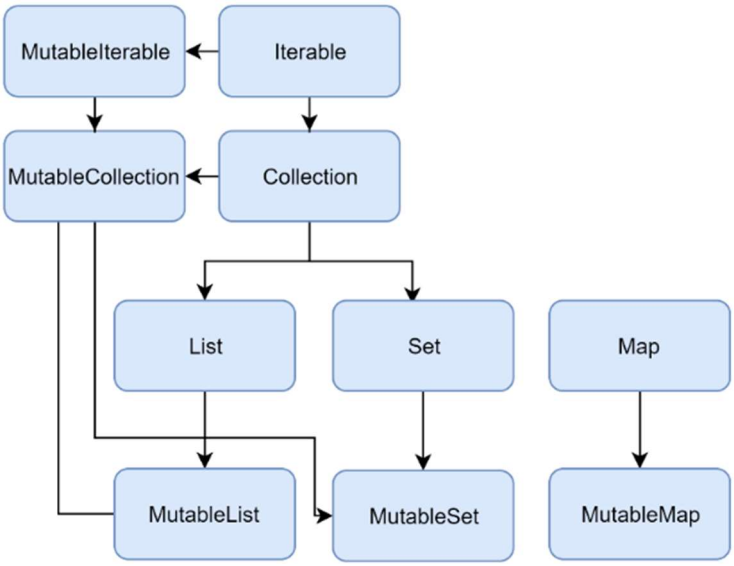
Kolekce do verze Javy 9 byly pouze měnitelné neboli *mutable*, což znamená, že šlo přidávat či odebírat hodnoty. Všechny kolekce pocházely z rodičovského rozhraní *Collection*, jmenovitě *List, Set a Queue*. *SortedMap pak* pochází z rozhraní *Map*. Nicméně, zmíněné kolekce jsou stále jen rozhraním, je tedy nutné používat některá implementovaná řešení. V případě *Listu* například *ArrayList*, u *Setu HashSet* a u *Map* třeba *HashMap*. Implementovaných možností je samozřejmě více a případně je možné si implementovat vlastní.

Od Javy 9 přibyly i *Immutable Collections* a s nimi mnohem snazší způsob inicializace. Neměnné kolekce podporuje *List, Set i Map*. Inicializovat jdou velmi snadno, a to pomocí metody *of()*, která je definovaná ve výše zmíněných rozhraních. Parametrem funkce jsou tedy hodnoty, které mají být do kolekce vloženy.

Kotlin

V Kotlinu se dá rozlišovat mezi dvěma typy kolekcí. Těmi jsou *Collection* a *MutableCollection* (potomek *Collection*). Přímí potomci třídy *Collection* jsou neměnné, tudíž se nedají za běhu přidávat či odebírat prvky. Patří mezi ně *List, Set* nebo *Map*, které jsou známé z Javy jako rozhraní. Pokud je potřeba kolekci za běhu programu měnit, tzn. přidávat a odebírat prvky, pak lze použít potomky třídy *MutableCollection*, tedy *MutableList, MutableSet* a *MutableMap*. Z toho logicky vyplývá, že u kolekcí s rodičem *Collection* neexistují žádné metody pro manipulaci s prvky.

Vytvoření kolekce v Kotlinu je u obou typů velmi jednoduché. Pro vytvoření *Listu* poslouží funkce *listOf()* a pro *MutableListu mutableListOf().* Obdobné to je i u kolekce typu *Set* či *Map*. Metoda *toList()* a range konstrukt (*..* - dvě tečky) je další možností, jak snadno inicializovat hodnoty v Listu. Stejně tak to je možné pomocí inline funkcí *List()* a *MutableList()*, které umožňují inicializovat hodnoty pomocí výrazů.



Obrázek 1 Diagram kolekcí v Kotlinu.
Zdroj: [1]

```
List<Integer> list1 = new ArrayList<>();
list1.add(1); //do listu lze libovolně hodnoty přidávat
list1.add(2);
list1.add(3);
list1.remove(0); //z listu lze hodnoty i mazat

//nebo tento zápis
 List<Integer> list2 = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6));

//od Javy 8, lze volat na list metodu forEach, jinak je nutné použít
standardní cyklus
 list2.forEach(System.out::println); //pracovat s listem lze jednoduše
pomocí foreach a reference na metodu
 list2.forEach(it -> System.out.println(it)); //nebo s foreachem a
lambdou

//od Javy 9
List<Integer> list = List.of(1, 2, 3); //Obdobné je to u setu
Map<Integer, String> map = Map.of(1, "a", 2, "b", 3, "c");
```

Ukázka 4.3-a Práce s listem v Javě
Zdroj: autor

```
val list1 = listOf(1, 2, 3, 4, 5);
list1.add(6); //nelze, jelikož List nelze upravovat

//obdobný list lze zapsat i takto
val list2 = (1..5).toList()
//nebo takto
val list3 = List(5){i -> i + 1} //lze inicializovat i pomocí inline
funkce a výrazu, obdobné i s MutableList
//1, 2, 3, 4, 5

//pracovat s listem lze jednoduše pomocí foreache a lambdy
list3.forEach{ it -> println(it)}

val mList = mutableListOf(1, 2, 3)
mList.add(4) //lze, jelikož se použil MutableList
```

Ukázka 4.3-b Práce s Listem a MutableListem v Kotlinu
Zdroj: autor

4.3.2 Pole

Java

Pole v Javě má pevně danou velikost a prvky v něm lze měnit. Deklaruje se pomocí datového typu a hranatými závorkami za tímto datovým typem. Počet hranatých závorek udává počet dimenzí (většinou jedna nebo dvě). Při inicializaci je třeba použít klíčové slovo *new*, opět datový typ s hranatými závorkami, kdy je tentokrát v hranatých závorkách zadaná požadovaná velikosti pole. Též jej lze inicializovat hodnotami ve složených závorkách, kdy se velikost určí automaticky na základě počtu zadaných hodnot. K hodnotám se přistupuje pomocí indexu v hranatých závorkách. Pro procházení pole lze využít *for cyklus* či *for each cyklus*. Od Javy 8 je dokonce možné využít *Streaming API* nebo odkazu (reference) na metodu.

Kotlin

Pole v Kotlinu má též pevně danou velikost a prvky v něm lze měnit. Opět má od Javy odlišnou deklaraci. Deklaruje se totiž podobně jako kolekce v Kotlinu, a to funkcí *arrayOf()*. Zde však žádný dvojník typu *Mutable* neexistuje. K hodnotám se stejně jako v Javě přistupuje pomocí indexu v hranatých závorkách. A jak je pole reprezentováno, co se týče datových typů? Každý datový typ má svoji „*array verzi*“, například *IntArray, DoubleArray* atd. Takže v případě, kdy je potřeba pole pouze

deklarovat a hodnoty inicializovat později, vytvoří se instance pole pomocí těchto tříd. Pokud se pole rovnou inicializuje, lze se těmto třídám díky typové inferenci a funkci *arrayOf()* zcela vyhnout. A jelikož jsou pole vyjádřená jako objekty, je možné volat spoustu užitečných funkcí rovnou na instanci pole, konkrétně *forEach()*, *indexOf()*, *last()*, *stream()* atd.

```
int[] array = {1, 2, 3, 4, 5};
array[2] = 4; //změna hodnoty skrz index

Object[] oArray = {1, 2, 3, "text", 1.3f }; //pomocí Object lze
mixovat datové typy

int[] array2 = new int[5]; //lze inicializovat index po index,
např. pomocí for cyklu

//od Javy 8 lze využívat streaming API a reference (odkazu) na
metodu
Arrays.stream(array).forEach(System.out::println);
```

Ukázka 4.3-c Práce s polem v Javě
Zdroj: autor

```
val array = arrayOf(1, 2, 3, "text", 1.3f) //pole může mít
namixované datové typy
array[3] = 4 //změna hodnoty skrz index
array.set(3, 4) //ekvivalentní zápis, akorát skrz funkci

val intArray1 = intArrayOf(0, 2, 6, 12, 20) //zde nelze mixovat
typy, jedná se totiž o IntArray
val intArray2 = IntArray(5){ i -> (i + 1) * i } //lze inicializovat
i pomocí nové instance a výrazu - 0, 2, 6, 12, 20
intArray2.forEach { it -> println(it) } //lze vypsat obdobně jako
list
```

Ukázka 4.3-d Práce s polem v Kotlinu
Zdroj: autor

4.4 Cykly

Java

Java poskytuje hned několik možností, jak procházet kolekce a pole či vykonávat nějakou operaci opakovaně. Nejzákladnějšími cykly jsou *for cyklus* – většinou se známým počtem iterací a *while cyklus* – s neznámým počtem iterací.

For cyklus se vytvoří pomocí klíčového slova *for* a za něj do jednoduchých závorek patří středníkem oddělené parametry. V prvním parametru se inicializuje proměnná, která slouží pro běh cyklu (velmi často pojmenovaná *i* od slova *iterate*). Dalším parametrem je podmínka, která se musí splnit, aby byl kód uvnitř cyklu vykonán (při procházení kolekcí: *i < velikostKolekce*). Posledním parametrem je krok, o kolik se má hodnota, inicializovaná v prvním parametru, změnit (velmi často *i++*, resp. *i+=1*, resp. *i = i +1*). Při spuštění cyklu se tedy postupuje tak, že se inicializuje iterační proměnná, zkontroluje se s podmínkou a vykonají se operace v těle cyklu. V dalším kroku se již první parametr ignoruje a začíná se parametrem třetím, tudíž se iterační proměnná nastaví na jinou hodnotu, zkontroluje se platnost podmínky a opět se vykonají operace v cyklu. Takto se pokračuje dále, dokud je podmínka platná. Tento cyklus se využívá převážně pro průchod hodnot v polích či kolekcích.

Dalším cyklem, který je dokonce ještě vhodnější pro procházení polí, je cyklus *foreach*. Je na to totiž přímo přizpůsoben. V Javě se vytvoří opět pomocí klíčového slova *for*, kde do jednoduchých závorek přijdou jen dva parametry, které jsou tentokrát oddělené dvojtečkou. Prvním parametrem je deklarace proměnné, přes kterou budou přístupné hodnoty z procházeného pole. Je tedy nutné, aby byla stejného datového typu jako hodnoty v poli (kolekci). Dalším parametrem za dvojtečkou je pak pole či kolekce, která je určená k procházení. Takto získané hodnoty však nelze měnit. Slouží tudíž jen pro čtení a pro změnu hodnot by se musel použít standardní *for* cyklus. *For* a *foreach* jsou tedy cykly, které mají předem známý počet průchodů.

While cyklus má své kouzlo v jednoduchosti. Stačí mu pouze jeden údaj za klíčovým slovem *while*, a to je podmínka. Dokud je tato podmínka platná, vykonávají se operace v těle cyklu.

Posledním cyklem je *do-while* cyklus. Je velmi podobný cyklu *while*, až na to, že operace, které se mají provést, se vkládají do bloku *do*. Kód, který je uvnitř *do* bloku, se provede alespoň jednou, nehledě na podmínku ve *while* cyklu. Klíčové slovo *while* s podmínkou se píše za konec *do* bloku, tentokrát ale se středníkem za závorkou. Může se například hodit při načtení hodnoty v konzolových aplikacích, dokud uživatel nezadá požadovaný vstup.

Kotlin

Kotlin, ostatně jako většina programovacích jazyků, má stejné cykly, které byly popsány u Javy. Nicméně se opět nepatrně liší zápisem, jelikož se snaží dát do pozadí relační a aritmetické operátory a místo nich používat klíčová slova.

Ve *for* cyklu bude opět potřeba nějaká iterační proměnná, u které není ani vyžadováno, aby měla specifikován datový typ, jelikož se ve většině případů používá *Integer*. Následuje operátor *in* a po něm interval procházení. Možností, jak zadat interval je více. Pokud je například potřeba projít hodnoty od 0 do 10 včetně, zapíše se interval pomocí *range* (*..* – dvě tečky), tedy jako *0..10*. Je-li třeba poslední hodnotu vynechat, stačí použít konstrukt *until*. Též je možné místo vzestupného procházení hodnot procházet hodnoty sestupně. K tomu lze použít klíčové slovo *downTo*. Odpadá také nutnost zadávat krok iterování proměnné, jelikož se velmi často iterovalo právě o jedničku. V případě potřeby lze krok iterace změnit klíčovým slovem *step*. Cyklus je možné jednoduše zapsat i tak, aby naplnil kolekci hodnotami. Do jednoduchých závorek se definuje interval a pomocí tečkové notace se zavolá funkce *toList()*, která vrátí List s hodnotami v zadaném intervalu.

Foreach se v Kotlinu dá zavolat pomocí tečkové notace u všech tříd implementujících rozhraní *Iterable*. Konkrétně pak u veškerých kolekcí či polí.

Cykly *while* a *do-while* mají totožný zápis jako v Javě.

```java
int[] array = {1, 2, 3, 4, 5};


for (int i = 0; i < array.length; i++) { //lze využít obecně při známém
počtu opakování, tj. i průchod pole a kolekcí
    array[i] *= (i + 1);
}

for (int i : array) { //lze využít pro čtení hodnot v poli / kolekce
   System.out.println(i);
}

int i = 0;
while(i < 5){ //dokud je podmínka platná navyšuje se i o jedna
    i++;
}
//i na konci cyklu je 5

do{ //provede se minimálně jednou nehledě na podmínku ve while
    i *= 2;
}while(i < 5);
//i na konci je 10
```

Ukázka 4.4-a Práce s cykly v Javě
Zdroj: autor

```kotlin
for (i in 1..10 ){} //od 1 do 10 včetně

for(i in 1 until 10 step 2){} //od 1 do 10 s krokem 2 (1, 3,
5, 7, 9)

for(i in 10 downTo 1 step 2){} // od 10 do 1 s krokem dva
(10, 8, 6, 4, 2)

val list = (0..10).toList() //lze použít i pro inicializaci
listu

//while a do-while pracují totožně jako v Javě
```

Ukázka 4.4-b Práce s cykly v Kotlinu
Zdroj: autor

4.5 Metody

Java

Vytváření funkcí v Javě je obdobné jako deklarace proměnných. Nejdříve se udá viditelnost funkce (defaultní viditelnost v Javě je *package-private*) a za ni následuje návratový datový typ funkce (datový typ výstupu). V případě, že funkce nic nevrací, se použije klíčové slovo *void*. Za návratový datový typ se zapíše název funkce. Do jednoduchých závorek lze umístit vstupní parametry funkce oddělené čárkou a do složených závorek se vkládají jednotlivé operace.

Kotlin

Kotlin má opět mírně odlišný zápis pro vytvoření funkce. Je potřeba počítat s tím, že defaultní viditelnost je *public*. Při vytváření funkce v Kotlinu se nejprve použije klíčové slovo *fun*, následuje název funkce, za který se do kulatých závorek uvedou čárkou oddělené parametry. Neuvádí se, zda se jedná o konstantu (*val*) či o proměnnou (*var*), stačí pouze název proměnné a za dvojtečku uvést datový typ parametru. Je dokonce možné pomocí rovnítka zadat parametru defaultní hodnotu. Není tedy nutné přetěžovat funkce jako u Javy. Nakonec se za dvojtečku zapíše návratový typ funkce. Pokud funkce nic nevrací, nemusí se žádný datový typ uvést. Následně se do složených závorek vkládají operace stejně jako u Javy. Pokud se však jedná o funkci s jednořádkovým příkazem, lze složené závorky vynechat a rovnou za datový typ umístit rovnítko a vykonat potřebnou operaci.

```
public int add(int number, int toAdd){
    return number + toAdd;
}

public int add(int number){ //přidá automaticky jedničku, pokud se
nemá uvádět další parametr
    return number + 1;
}
```

Ukázka 4.5-a Jednoduchá funkce v Javě
Zdroj: autor

```kotlin
fun add(number: Int, toAdd: Int = 1): Int = number + toAdd
//stejná funkce jako v Javě, nemusí se však díky defaultnímu parametru
přetěžovat a lze zapsat na jeden řádek

fun sayHi() = println("Hi") //u funkce, která nic nevrací se nemusí psát
void
```

Ukázka 4.5-b Jednoduchá funkce a funkce bez návratové hodnoty v Kotlinu
Zdroj: autor

4.5.1 Přístupové metody

Přístupové metody pomáhají dodržovat princip zapouzdření. Rozlišuje se mezi dvěma přístupovými metodami, a to *getry* a *setry*. *Getry* slouží pro vracení hodnot atributů a pomocí *setrů* lze hodnotu atributu měnit. Atribut sám o sobě mívá *private* modifikátor a přístupové metody zpravidla *public* modifikátor, záleží však na konkrétní situaci.

Java

V Javě je nutné pro každý atribut vytvořit *getr* a *setr*, případně je lze vygenerovat vývojovým prostředím. Častokrát však mají tyto metody výchozí implementaci a nevykonávají žádné operace navíc. Kvůli tomu se *datové třídy* často mění na výčet *getrů* a *setrů*, které nemají žádnou speciální funkci.

Kotlin

Jak již bylo zmíněno výše, Kotlin má výchozí viditelnost *public* a není tomu náhodou. Pokud se vytváří atributy v nějaké třídě, není třeba uvádět modifikátor, jelikož *public* modifikátor se využívá nejčastěji. Může se zdát, že dochází k porušení principu zapouzdření, ale rozhodně tomu tak není. Jelikož přístupové metody mají častokrát výchozí implementaci, jsou v Kotlinu u každého atributu automaticky vygenerovány kompilátorem. Pokud se u atributu uvede *public* modifikátor (nebo žádný), udá se tím viditelnost obou přístupových metod - ne však atributu samotného. Kotlin následně vytvoří ke všem atributům *getry* a v případě, že se jedná o *var*, tak i *setry*.

Dojde tím tedy ke zpřehlednění kódu, jelikož již není nutné pro všechny atributy vytvářet zvlášť přístupové metody, ale pouze pro ty, u kterých je třeba provést další operace. V případě potřeby změny *setru*, stačí pod atribut zapsat *set(value){}*, kde se do bloku složených závorek může psát vlastní logika. Má-li být *setr* privátní, stačí uvést private modifikátor před *set*. Může se jevit, že se přistupuje k atributům napřímo, ale ve skutečnosti se na pozadí volá konkrétní přístupová metoda v závislosti na tom, zda se hodnota atribut mění či pouze čte.

```
private int age;

private void setAge(int age) {
    this.age = age;
}

public int getAge() {
    return age;
}
```

Ukázka 4.5-c Přístupové metody v Javě
Zdroj: autor

```
var favColor: String //pokud se modifikátory nijak nemění, nemusí se ani
zmínit

var age: Int
private set //pro privátní seter stačí tento zápis

var birthDay: Date
set(value) {
    field = value
    //vypočet věku }
```

Ukázka 4.5-d Přístupové metody v Kotlinu
Zdroj: autor

4.6 Třídy

Třídy jsou velmi důležité pro objektově orientované programování, které vnese do kódu pořádek a přehlednost. I zde se najde pár nepatrných rozdílů v zápisu.

Java

V Javě se vytvoří třída pomocí klíčového slova *class*, za kterým následuje pojmenování třídy. Poté lze definovat, zda vytvářená třída bude implementovat nějaká rozhraní pomocí klíčového slova *implements*, nebo zda bude dědit z nějaké již vytvořené třídy či abstraktní třídy pomocí klíčového slova *extends*. Lze dědit pouze z jedné (abstraktní) třídy, počet implementovaných rozhraní není nijak omezen. Následuje tělo dané třídy, které je uzavřené ve složených závorkách. Zde jde vytvořit libovolné množství atributů a metod patřících k třídě. Pomocí klíčového slova *static* je možné nějaký atribut nebo metodu udělat nezávislou na instanci třídy. Co se týče konstruktorů, ty nesou stejný název jako třída a mohou přijímat libovolný počet vstupních parametrů. Konstruktory se dají i přetěžovat, tudíž vytvořit více konstruktorů pro jednu třídu s rozdílným množstvím parametrů či odlišnými datovými typy u parametrů.

Kotlin

Vytvoření třídy jako takové není v Kotlinu až tak rozdílné. Opět se použije klíčové slovo *class*, za který se uvede název třídy. Nyní už však začínají oproti Javě nějaké změny. Za název třídy se totiž mohou přidat jednoduché závorky, do kterých lze uvádět atributy třídy. Jelikož se jedná o atributy, a ne o parametry nějaké metody, je nutné uvést, zda se jedná o konstantu (*val*) nebo proměnnou (*var*). K čemu to je vlastně dobré? Díky tomu se totiž vytvoří i primární konstruktor přesně s těmito parametry a automaticky se hodnoty předané v parametrech při volání konstruktoru inicializují do atributů třídy.

Co když ale existuje atribut, který má dostat nějakou defaultní hodnotu a není nutné jej mít jako parametr v konstruktoru? Kam se umístí, když chybí tělo konstruktoru, kde by se takovýto atribut bez problému inicializoval? Jednoduše, lze totiž použít blok *init{}*, který je dělaný právě pro tyto účely. Je vykonáván ihned po volání

primárního konstruktoru třídy, nezáleží tedy na tom, kam se v kódu tento blok umístí. Není ani nijak omezen počet *init* bloků. Vykonávají se v tom pořadí, jak byly vytvořeny, tj. v pořadí od shora dolů.

Velmi často se též stane, že nebude stačit pouze primární konstruktor. Vytvořit další není nijak složité. Na rozdíl od Javy se při tvorbě konstruktoru nepoužije název třídy, ale klíčové slovo *constructor()*, kde se do kulatých závorek mohou uvést parametry konstruktoru. V těle konstruktoru je možné inicializovat potřebné atributy, ať už defaultními hodnotami či hodnotami získanými z parametrů a zavolat primární konstruktor. Lze se však obejít i bez těla konstruktoru. Za výčet parametrů, resp. za uzavírací závorku, se za dvojtečkou pomocí *this* zavolá primární konstruktor s hodnotami získanými z parametrů nebo s hodnotami defaultními. Vyplatí se tedy primární konstruktor zapsat se všemi potřebnými parametry a následně vytvářet další konstruktory, kde se zvolené parametry nastaví defaultní konstantní hodnotou.

Samozřejmě může být potřeba implementovat nějaká rozhraní nebo rozšířit nějakou třídu. Zde odpadnou klíčová slova *extends* a *impelements*, jelikož místo nich postačí pouhá dvojtečka. Kotlinu nedělá problém rozeznat, zda implementuje rozhraní či rozšiřuje nějakou třídu. Platí zde stejná omezení jako v Javě. Lze tedy dědit pouze z jedné třídy nebo abstraktní třídy a počet implementovaných rozhraní je bez limitu.

V Kotlinu se vývojář nesetká ani se *static*. Jak je tedy možné vytvořit atribut či metodu nezávislou na instanci třídy? Pro statické atributy a metody slouží blok *companion object{}*, které se vytváří do těla tohoto bloku. Tudíž je všechno statické pohromadě, což dokáže opět zpřehlednit kód. *Companion object* lze i pojmenovat. To se může hodit například pro vytvoření „*Továrny*“ neboli „*Factory*“, což jsou právě statické třídy, které například vytvářejí instance objektů.

```
public class Example extends ExampleAbstract implements ExampleInterface{

    String defaultHi;
    String defaultGoodBay;
    static int goodBayCounter = 0;
    int defaultValue;

    public Example(String defaultHi, String defaultGoodBay){
        this.defaultHi = defaultHi;
        this.defaultGoodBay = defaultGoodBay;
        defaultValue = 123;
    }

    public Example(){
        this("Hi, I'am child of Example Abstract",
                "ExampleInterface told me I should be able to say Good
bay, so.. Good bay");
    }

    @Override
    public void sayGoodBay() {
        goodBayCounter++;
        System.out.println(defaultHi);
    }

    @Override
    public void sayHi(){
        System.out.println(defaultGoodBay);
    }
}
```

Ukázka 4.6-a Třída rozšiřující třídu a implementující rozhraní v Javě
Zdroj: autor

```
class Example(var defaultHi: String, var defaulGoodbay: String):
ExampleAbstract(), ExampleInterface {

    var defaultValue: Int

    companion object { //sem patří statické funkce a atributy
        var goodBayCounter: Int = 0;
    }

    init{ //použije se pro inicializaci proměnných, které nejsou
předány v parametru primárního konstruktoru
        defaultValue = 123;
    }

    constructor(): this("Hi, I'am child of Example Abstract",
        "ExampleInterface told me I should be able to say Good bay,
so.. Good bay")

    override fun sayHi() {
        println(defaultHi)
    }

    override fun sayGoodBay() {
        goodBayCounter++
        println(defaulGoodbay)
    }
}
```

Ukázka 4.6-b Třída rozšiřující třídu a implementující rozhraní v Kotlinu
Zdroj: autor

4.6.1 Abstraktní třídy a rozhraní

Abstraktní třídy i rozhraní jsou velmi užitečné při dědění. Rozhraní se využívá pro definování funkcí zpravidla bez implementace, která se přenechá pro třídu implementující rozhraní. Abstraktní třídy slouží též k definici funkcí, avšak funkce se zde mohou rovnou implementovat. Pokud se má implementace přenechat potomkovi, pak se jedná o funkci abstraktní, kdy se za modifikátor přístupu přidá *abstract*. Abstraktní třídy na rozdíl od rozhraní slouží i k deklaraci atributů. U rozhraní je možné pouze definovat statické konstanty. Abstraktní třídy ani rozhraní nevytváří instance, slouží pouze pro účely dědění. U obou programovacích jazyků platí, že lze rozšiřovat pouze jednu abstraktní třídu (obyčejnou třídu) a implementovat neomezené množství rozhraní.

Java

Vytvoření abstraktní třídy je obdobné jako tvorba obyčejné třídy, kdy se místo *class* použije *abstract class*. Interface se pak vytvoří pomocí klíčového slova *interface*. Pokud je nutné u rozhraní implementovat metodu, lze tak učinit přidáním *default* před datový typ metody. Metoda se pak bere jako implementovaná a třída implementující rozhraní již není nucena tuto metodu implementovat.

Pro implementaci interface slouží *implements*, pro rozšíření abstraktní či obyčejné třídy pak *extends*. Ty se umístí za název třídy, a pokud jich je více, oddělí se čárkou (platí pouze u rozhraní, jelikož lze rozšířit jen jednu třídu).

Před každou funkcí, která se implementuje by měla být přidána anotace *@Override*, která slouží pro kontrolu, že se opravdu implementuje metoda z rozhraní a nevytváří se metoda nová. Pokud by se totiž umístila anotace *@Override* před metodu, která v implementovaných rozhraních nebo rozšiřující abstraktní třídě není, IDE by tuto skutečnost oznámilo a vyvolalo chybu. Použití této anotace však není nutné.

Kotlin

Vytváření rozhraní i abstraktní třídy je v Kotlinu totožné s Javou. U Kotlinu se nemusí nic přidávat za modifikátor přístupu, pokud je třeba implementovat nějakou metodu v rozhraní. Kotlin se o to totiž postará interně, pokud se tedy metoda implementuje, Kotlin ji předeklaruje na *open*. Pokud se neimplementuje, je deklarovaná jako *abstract*.

Třídu lze rozšířit či implementovat rozhraní pomocí dvojtečky (*:*). Kotlin interně rozpozná, zda se jedná o interface nebo o třídu automaticky. Pokud se opět implementuje více rozhraní, oddělí se čárkou.

Funkce, které se implementují, musí obsahovat *override* před *fun*, případně před atributem. Tím je zřejmé, že se implementuje funkce rozhraní nebo abstraktní třídy a nevytváří se funkce nová. Na rozdíl od anotace v Javě, je *override* v Kotlinu nutné použít.

V Kotlinu lze dokonce v rozhraních deklarovat nefinální proměnnou - tedy *var*, nicméně se zde doporučuje vyhnout deklaraci jakýchkoliv atributů.

```
abstract class ExampleAbstract {
    public void sayHi(){
        System.out.println("Hi, I'am abstract example");
    }
}

interface ExampleInterface {
    void sayGoodBay();

    default void countFromOneToThree(){ //interface lze implementovat
po přidání default
        for (int i = 0; i < 3; i++) {
            System.out.println(i + 1);
        }
    }
}
```

Ukázka 4.6-c Vytvoření abstraktní třídy a rozhraní v Javě
Zdroj: autor

```
abstract class ExampleAbstract {
    fun sayHi() {
        println("Hi, I'am abstract example")
    }
}


interface ExampleInterface {
    fun sayGoodBay()

    fun countFromOneToThree() { //Kotlin sám rozpozná, že je funkce
implementována, není třeba nic přidávat
        for (i in 0..2) {
            println(i + 1)
        }
    }
}
```

Ukázka 4.6-d Vytvoření abstraktní třídy a rozhraní v Kotlinu
Zdroj: autor

4.6.2 Enum třídy

Enum třídy jsou velmi užitečným nástrojem pro definování výčtu konstant, které spolu nějak souvisí, konkrétně například dny v týdnu, názvy měsíců, role uživatele (uživatel, moderátor, administrátor) atd.

Java

V Javě je *enum* reprezentován jako třída. Při vytváření takové třídy se místo *class* použije *enum*. Enum třídy, stejně tak jako obyčejné třídy, potřebují i nějaký název. Po názvu už následují jen složené závorky, kam se může uvést výčet konstant. Konstanty jsou zpravidla psané kapitálkami, což může být někdy překážkou. Většinou tehdy, je-li potřeba vypsat název konstanty. Může se tedy vytvořit metoda, která bude vracet názvy podle požadavků. Jelikož je enum třída, neměl by to být problém. Enum v Javě má *getr name()*, který vrací název konstanty. Stačí tedy ve vytvořené metodě patřičnými funkcemi upravit výstup, případně lze přepsat metodu *toString*().

Kotlin

Jinak tomu není ani u Kotlinu, enum je opět typ třídy. Vytváří se jako *enum class,* a pokud je potřeba přidat funkci pro upravený výpis názvu konstanty, jde to rovněž bez problému. Enum v Kotlinu má stejně jako Java atribut *name*, který vrací název konstanty, stačí ho tedy patřičně upravit. Opět by šlo přepsat metodu *toString()*.

```
enum Days{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    public String getDayName(){
        return name().substring(0,1).toUpperCase() +
name().substring(1).toLowerCase();
    }
}
```

Ukázka 4.6-e Enum třída v Javě
Zdroj: autor

```
enum class Days{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY;

    fun getDayName() = name.toLowerCase().capitalize();
}
```

Ukázka 4.6-f Enum třída v Kotlinu
Zdroj: autor

4.6.3 Sealed třídy

Sealed třída udává, kterými třídami může být rozšířena či implementována. Tím dochází ke zvýšení zabezpečení, jelikož není možné, aby třída rozšiřovala sealed třídu bez toho, aniž by jí to bylo touto třídou povoleno, což se může velmi hodit při vývoji knihoven. I přesto, že se jedná v Javě o novinku, čekalo by se, že bude zápis velmi podobný Kotlinu, nicméně se opět v mnoha ohledech liší.

Java

Do nedávna by sealed třídy nebyly v Javě k nalezení, to se však změnilo s verzí 15. Sealed třída se nadefinuje jako *sealed class*. Může se jednat o standardní třídu nebo o třídu abstraktní, stejně tak se může jednat i o interface (*sealed interface*). Sealed třída by měla mít *public* eventuelně *package-private* modifikátor z důvodu viditelnosti pro potomky. Jako každá třída, i tato, musí být nějak pojmenovaná. Za názvem třídy se za klíčové slovo *permits* píšou čárkou oddělené třídy, které z ní mohou dědit, respektive ji rozšířit. Pak už je možné v této třídě vytvářet atributy či metody.

Na třídu, která sealed třídu rozšiřuje, se ovšem musí vztahovat nějaká omezení. Ztratilo by totiž význam, aby bylo pro všechny, až na povolené třídy, zakázané dědění ze sealed třídy, kdyby šlo dědit z nějakého jejího potomka. Proto je nutné, aby byl potomek deklarován jako *final*, což znemožní další dědění. Pokud by však bylo potřeba potomka rozšiřovat, není problém z něj udělat též sealed třídu a umožnit povoleným třídám jej rozšířit. A kdyby dokonce nebylo nutné žádné další omezení, stačí tuto třídu předeklarovat jako *non-sealed*, a tuto třídu bude moci rozšířit libovolný potomek.

Kotlin

Na rozdíl od Javy musí být sealed třída deklarovaná jako *private* (je to výchozí viditelnost sealed třídy). Jak ovšem mohou dědit potomci z rodiče, když není viditelný? Potomci se totiž musí vytvářet ve stejném souboru jako sealed třída. Tím je tedy jasné, jací potomci mohou z této třídy dědit, aniž by bylo nutné tuto informaci někde udávat. Je to ovšem za cenu přehlednějšího kódu, avšak pokud se využije datových tříd, často bude stačit k vytvoření třídy jediný řádek.

I zde platí omezení jako v Javě. Každá třída, která dědí, je automaticky deklarovaná jako *final*, není tudíž možné ji dále rozšiřovat. Lze z ní ovšem udělat opět sealed třídu, a tak umožnit, aby z ní dědit šlo. Není-li však nutné omezovat dědičnost této třídy, stačí ji předeklarovat jako *open* třídu, a díky tomu z ní bude možné bez omezení dědit. Pozor však na datové třídy, ty jsou vždy deklarované jako *final* a nelze to obejít ani s pomocí *open* (více o datových třídách v kapitole 4.6.4).

Sealed rozhraní nebylo donedávna možné v Kotlinu vytvořit. Momentálně jsou však sealed rozhraní označeny jako *Experimental*, což značí, že jsou v prvotní testovací fázi a může se stát, že se jejich funkcionalita může výrazně změnit, nebo se dokonce v dalších verzí Kotlinu nemusí sealed rozhraní ani vyskytovat [8].

```
sealed class ExampleAbstract permits PrimaryExample, SecondaryExample {
//třídy, které mohou dědit jsou vypsané za permits,
    // díky tomu není nutné mít všechny třídy v jednom souboru
    void saySomething(){
        System.out.println("I am ExampleAbstract");
    }
}

sealed class PrimaryExample extends ExampleAbstract permits
TertiaryExample { //musí se udat, že se jedná o sealed třídu a povolit
potomky, kteří mohou dědit
    @Override
    public void saySomething(){
        System.out.println("I am PrimaryExample, child of
ExampleAbstract.");
    }
}


non-sealed class SecondaryExample extends ExampleAbstract{ //non-sealed
třídu lze bezproblému rozšířit
    @Override
    public void saySomething(){
        System.out.println("I am SecondaryExample, child of
ExampleAbstract.");
    }
}

final class TertiaryExample extends PrimaryExample{ //musí se udat, že se
jedná o final třídu, aby nemohla být rozšířena
    @Override
    public void saySomething() {
        System.out.println("I am TertiaryExample, child of PrimaryExample.
I can't have child.");
    }
}
```

Ukázka 4.6-g Sealed třída v Javě
Zdroj: autor

```
sealed class ExampleAbstract {
    open fun saySomething(){ //pokud nebude open, pokládá se za final a
funkce nepůjde přepsat
        print("I am ExampleAbstract")
    }
}

sealed class PrimaryExample: ExampleAbstract(){ //mohou z něho dědit jen
potomci v jednom souboru
    override fun saySomething(){
        print("I am PrimaryExample, child of ExampleAbstract.")
    }
}

open class SecondaryExample: ExampleAbstract(){//open udává, že
SeconderyExample bude možné rozšířit, kdekoliv i mimo soubor
    override fun saySomething(){
        print("I am SecondaryExample, child of ExampleAbstract.")
    }
}

class TertiaryExample:PrimaryExample(){ //pokládá se za final, nelze z
něho již dědit
    override fun saySomething(){
        print("I am TertiaryExample, child of PrimaryExample. I can't have
child.")
    }
}
```

Ukázka 4.6-h Sealed třída v Kotlinu
Zdroj: autor

4.6.4 Datové třídy

Datové třídy jsou velmi vhodným řešením pro třídy, které jsou určené spíše pro ukládání dat. Mají tedy velké množství atributů a neobsahují příliš logiky – mnohdy žádnou. V praxi se tedy jedná o DTO (Data Transfer Object) nebo o POJO (plain old Java object). Jak u Javy, tak u Kotlinu přináší spoustu výhod.

Java

Do verze Javy 14 se Kotlinu podobná datová třída vytvářela klasickým způsobem jako běžná třída, kdy ji byly definovány nějaké atributy, konstruktor(y), včetně metod přístupových.

V Javě 14 pak přibyla nová třída *record*, která je velmi podobná níže zmíněné datové třídě v Kotlinu. Kompilátor automaticky generuje metody *toString()*, *equals()* a *hashCode()*, které se udržují vždy aktuální. Též odpadla nutnost vytvářet přístupové metody. *Record* třída je vždy deklarovaná jako *final*, a proto z ní není možné dědit. Při vytváření konkrétní instance musí být atributy předány ihned v konstruktoru, jelikož vlastnosti tříd jsou *immutable,* a tak nelze volat žádný *setr*. K vlastnostem se pak přistupuje pomocí funkcí, kdy se každá funkce jmenuje stejně jako vlastnost, jejíž hodnotu vrací.

Kotlin

Datové třídy v Kotlinu umožňují též automatickou implementaci metod *equals()*, *hashCode()* a *toString()*, které se též udržují vždy aktuální. Datové třídy v Kotlinu obsahují i metodu *copy()*, jež umožňuje jednoduše zkopírovat instanci a rovnou ji dle potřeby měnit. Díky tomu všemu lze získat opět velmi přehledný kód, který jde velmi často zapsat do jednoho řádku. Nemusí se totiž implementovat zmíněné metody a ani metody přístupové, jelikož jsou interně vygenerovány.

Avšak tyto třídy mají i nějaké nevýhody. Z takto vytvořených tříd není totiž možné dědit, jelikož jsou deklarovány jako *final* třídy. Kvůli tomu nelze z takovéto třídy udělat třídu abstraktní. Dalším omezením je, že primární konstruktor datové třídy musí mít alespoň jeden parametr.

Vytvoření datové třídy není nikterak rozdílné od vytvoření klasické třídy. Místo klíčového slova *class*, se použije *data class* a při zápisu parametrů do konstruktoru je třeba definovat, zda se jedná o proměnnou (*var*) či o konstantu (*val*), aby došlo ke korektnímu vygenerování přístupových metod.

```
public class PersonJ {
    private String firstName; //pro zapouzdření se musí udat private
modifikátor
    private String lastName;
    private Date birtDay;
    private int age;

    public PersonJ(String firstName, String lastName, Date birthDay){
        this.firstName = firstName;
        this.lastName = lastName;
        this.birtDay = birthDay;
    }

    public PersonJ(String firstName, String lastName){
        this(firstName, lastName, null);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PersonJ personJ = (PersonJ) o;
        return age == personJ.age && firstName.equals(personJ.firstName) &&
lastName.equals(personJ.lastName) && Objects.equals(birtDay, personJ.birtDay);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName, birtDay, age);
    }

    @Override
    public String toString() {
        return "PersonJ{" +
                "firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'' +
                '}';
    }

    public int getAge() { return age; }

    public void setBirtDay(Date birtDay) {
        this.birtDay = birtDay;
        //výpočet věku
    }
```

Ukázka 4.6-i Třída v Javě se stejnou funkcionalitou jako v ukázce 4.6-k
Zdroj: autor

```
public record Person (String firstName, String lastName, Date birthDay, int
age){
    public Person(String firstName, String lastName){
        this(firstName, lastName, null, 0);
    }
}

//Vytvoření instance record třídy
Person person = new Person("Jan", "Novák");
person.firstName(); //přístup k atributům skrze funkci
person.setFirstName("Honza"); //NELZE record třída má immutable atributy
```

Ukázka 4.6-j Record třídy v Javě (od verze 14)
Zdroj: autor

```
data class Person(var firstName: String, var lastName: String) { //pozor
data class je final, nelze ji tedy rozšířit
    var age: Int //vytvoří atribut age
        private set //setr nastaví na private

    var birthDay: LocalDateTime? = null
        set(value) { //setr birthDay bude přepočítávat věk
            field = value //nastavení atributu na hodnotu value
            //výpočet věku
        }

    constructor(firstName: String, lastName: String, birthDay:
LocalDateTime?) : this(firstName, lastName) { //sekundární konstruktor
        this.birthDay = birthDay;
    }

    init {
        age = -1; //výpočet věku na základě data narození, vypočte se v
setru dateBirth
    }

    override fun toString(): String {
        return "$firstName $lastName, $birthDay ( $age )";
    }

}
```

Ukázka 4.6-k Datové třídy v Kotlinu se dvěma konstruktory a přístupovými metodami
Zdroj: autor

4.7 Rozšiřující funkce

Rozšiřující funkce umožňují rozšíření již existujících tříd bez nutnosti dědění, a tudíž bez nutnosti vytváření nových potomků.

```
fun String.toTag(tag: String): String { //rozšíření třídy String o
funkci toTag, lze zapsat i na jeden řádek pomocí „="
    return "<$tag>$this</$tag>"
}
```

Ukázka 4.7-a Využití rozšiřující funkce v Kotlinu
Zdroj: autor

4.8 Chytré přetypování

Výhoda chytrého přetypování spočítá v tom, že jakmile se jednou podmínkou zkontroluje, že daná instance je konkrétního datového typu, není již v bloku podmínky potřeba tuto instanci přetypovat. K přetypování totiž dojde zcela automaticky.

```
fun getLength(value: Any) { //Any je datový typ pro libovolný objekt
    if(value is String)
        println(value.length) //díky kontrole, se value automaticky
přetypovalo na String a je možné přistoupit k vlastnosti length
    else
        print("Isn't a String")
}
```

Ukázka 4.8-a Chytré přetypování v Kotlinu
Zdroj: autor

4.9 High-order funkce

High-order funkce dokáží v parametru přijímat funkce a stejně tak mohou i funkci vracet jako svůj výsledek. Jelikož, na rozdíl od Javy, je Kotlin i funkcionálním jazykem, není to pro něho žádný problém. Nicméně, od Javy 8 je možné využívat *lambdy*, což umožňuje výrazně kratší zápis u funkcí, u kterých se velmi často v parametru implementovala vnitřní třída.

```
fun calc(x: Int, y: Int, operation: (Int, Int) -> Int): Int { //vstup
dvě Int hodnoty a funkce, která opět přijímá dvě Int hodnoty
    return operation(x, y) //výsledkem je volání výše definované
metody se získanými parametry x a y
}

//volání metody
calc(5, 3) { i, i2 -> i * i2 }
```

Ukázka 4.9-a Jednoduchá high-order funkce v Kotlinu
Zdroj: autor

5 Praktická část

5.1 Android

Praktickou část tvoří dvě Android aplikace se stejnou funkcionalitou, přičemž jedna z nich je napsaná v Kotlinu (viz příloha 1) a druhá v Javě (viz příloha 2). Aplikace s názvem E-Library má simulovat aplikaci napojenou na libovolnou knihovnu, kde je možné ze seznamu knih zarezervovat knihu k vypůjčení. Stejně tak uživatel vidí aktuální výpůjčky, které lze i prodloužit. Jedná se pouze o ukázku pro práci s Javou a Kotlinem, aby se dalo poukázat na rozdíly při vývoji a vytvořit metriky pro číselné vyjádření úspory práce programátora. Kvůli tomu není aplikace napojena na skutečnou databázi a data jsou zadána napevno v aplikaci.

Jelikož Android SDK oficiálně podporuje pouze Javu ve verzi 8, nelze při vývoji používat novější verzi Javy, a tudíž ani nové funkce, které právě novější verze nabízí [9]. Jmenovitě hlavně record třídy, které jsou dostupné až od Javy 14 a mohly by být při vývoji této aplikace použity.

Aplikace je vytvořena dle návrhového vzoru Model-View-ViewModel (MVVM), který je pro vývoj mobilních aplikací pro Android mnohem vhodnější než Model-View-Controller (MVC). Zejména díky tomu, že většina funkcí v Androidu vyžaduje znát *Context*, což je třída, která například umožňuje přístup ke zdrojům aplikace, spouštět nové *aktivity*, vytvářet *view* či používat systémové služby. *Context* je možné získávat v rámci *Aktivit*, které se starají o celé *View*. Proto má v Android aplikacích *View* mnohem větší zodpovědnost, než tomu bývá zvykem u MVC.

Android SDK obsahuje knihovny pro snadnou implementaci návrhového vzoru MVVM, jmenovitě abstraktní třídu *LiveData<T>* a třídu *MutableLiveData<T>.* Znak *T* zde zastupuje generický datový typ. Tato třída má dvě důležité metody, a to *setValue()* (v Kotlinu přístup napřímo skrze vlastnost value) a *postValue()*, kdy *setValue* slouží pro nastavení hodnoty v rámci hlavního vlákna aplikace a *postValue* slouží pro nastavení hodnoty skrze jiné než hlavní vlákno. Tyto operace se tedy využívají v Modelu, kde se získávají či upravují data.

Abstraktní třída *LiveData* nabízí metody pro práci s pozorovateli. Nejpodstatnějšími jsou *observe()* a *observeForever()*.

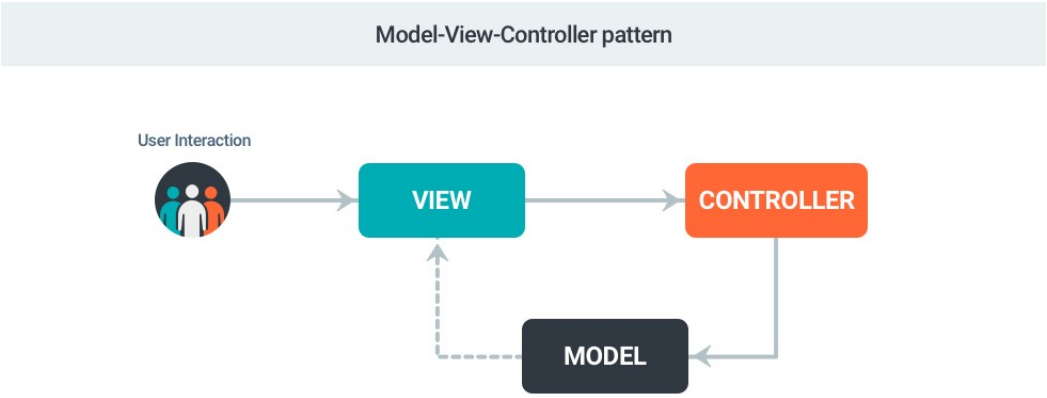
Observe() přijímá dva parametry, a to vlastníka observeru (většinou *Aktivita*, ve které se *Observer* vytváří) a implementaci *Observer* rozhraní. Dalšími metodami pro práci s pozorovateli jsou například *removeObserver(), removeObservers()* nebo *hasObservers().*

Zbývá tedy ještě něco zmínit o rozhraní *Observer<T>*. Toto rozhraní má pouze jedinou metodu a tou je *onChanged(T t)*, která přijímá hodnotu datového typu *T*. V těle této metody se tedy vykonávají veškeré operace, které se mají provést na *View* po získání dat.

Další důležitou abstraktní třídou je *ViewModel*, kterou se rozšiřují programátorem vytvářené *ViewModel* třídy. Není potřeba implementovat žádné metody, tato abstraktní třída slouží pouze k umožnění *Aktivitě* či *Fragmentu* získat instanci na vytvořené *ViewModely*. Instance se však nevytvářejí voláním konstruktoru, ale pomocí třídy *ViewModelProvider*. Tato třída jako parametr v konstruktoru přijímá vlastníka view modelu (což je *Aktivita* či *Fragment*). Následně lze pomocí *get()* metody získat instanci na požadovaný *ViewModel*. Parametrem této metody je název třídy (včetně .class přípony), která musí mít implementovanou zmíněnou abstraktní třídu *ViewModel*.

Model-View-Controller (MVC)

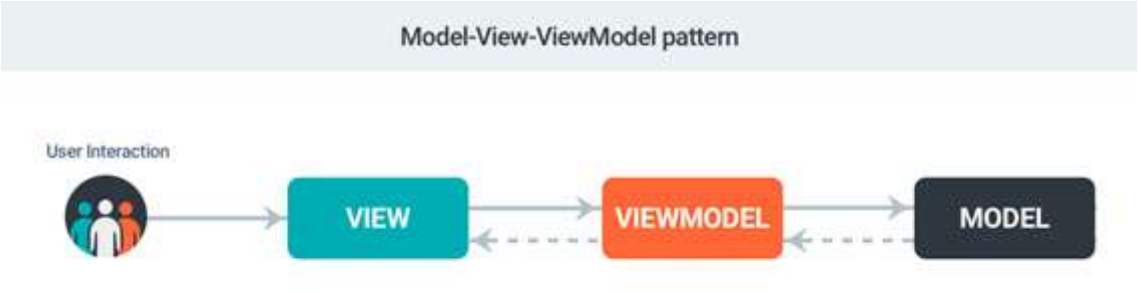
V MVC architektuře je uživatel v interakci s *View*, které komunikuje s *Controllerem*. *Controlleru* zasílá data získaná od uživatele a ten manipuluje s daty skrze *Model*, který má na starosti právě strukturu dat. V praxi tedy model komunikuje například s nějakou databází pomocí *SQL*. Poté, co jsou data připravená, je *Model* zasílá do *View*, které se na základě těchto dat aktualizuje.



Obrázek 2 Návrhový vzor Model-View-Controller
Zdroj: [2]

Model-View-ViewModel (MVVM)

V MVVM architektuře je opět uživatel v interakci s *View*, nicméně zde zmizela vazba mezi *Modelem* a *View*. *View* pro zjednodušení komunikuje pouze s *ViewModelem*. Ten následně manipuluje s daty skrze *Model* stejně jako u MVC architektury. *Model* získaná data zasílá zpět *ViewModelu* a ten upozorní *View* na změnu dat.



Obrázek 3 Návrhový vzor Model-View-ViewModel
Zdroj: [2]

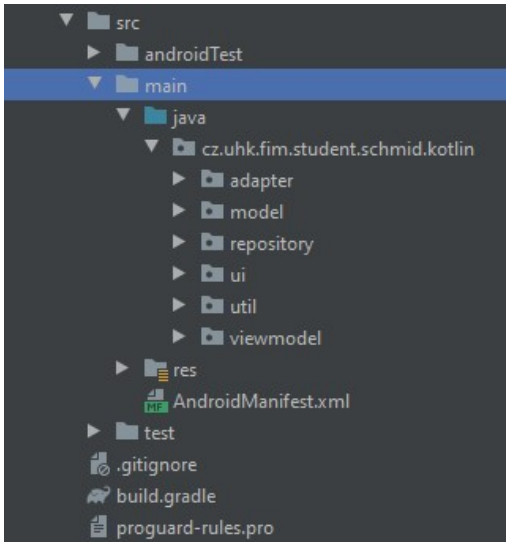
Struktura aplikace

Aplikace je přehledně strukturovaná do několika balíčků (viz obrázek č. 4). V balíčků **ui** jsou veškeré *aktivity* a *fragmenty*. Dalším balíčkem je **viewmodel**, který obsahuje všechny *ViewModel* třídy. V **repository** jsou k nalezení třídy starající se o data, které zastupují *Model* z MVVM vzoru. Datové třídy jsou vytvořeny v balíčku **model**.

Pro zobrazení dat se využívá *RecyclerView*, který je vhodný pro grafické zobrazovaní několika položek. *RecyclerView* potřebuje pro svou práci *LayoutManager* a *Adapter*. *LayoutManager* má již několik implementovaných variant, například *LinearLayoutManager*, který zobrazuje položky pod sebe či *GridLayoutManager* zobrazující položky ve sloupcích. Vlastní *LayoutManager* není tedy nutné vytvářet. Co je však třeba vytvořit jsou *Adapter* třídy, které jsou umístěny v balíčku **adapter**. *Adapter* spolu s *ViewHolder* se stará o grafickou reprezentaci dat, kdy *ViewHolder* implementující třídu *RecyclerView.ViewHolder* deklaruje a inicializuje *view* prvky z *layoutu* (*TextView, ImageView, Button, ...*). *Adapter* rozšiřující abstraktní třídu *RecyclerView.Adapter* si udržuje kolekci dat a stará se o nastavení hodnot z kolekce do prvků z *ViewHolder*. *ViewHolder* třídy jsou v této aplikaci vytvořeny uvnitř *Adapter* tříd.

Poslední **util** balíček slouží pro pomocné třídy, které jsou převážně využívány v repository třídách.

Zdroje jako jsou obrázky, *layouty*, nebo texty jsou umístěny mimo zdrojové kódy v balíčku **res**, jehož struktura je automaticky vygenerována při vytvoření projektu.



Obrázek 4 Struktura aplikace
Zdroj: autor

5.1.1 Popis prostředí Android Studio

Android Studio je velmi oblíbeným vývojovým prostředím od Googlu pro vývoj aplikací pro systém Android. Tím se rozumí nejen mobilní telefony s Androidem, ale i chytré hodinky či chytré televize. Dokonce je možné s pluginem Kotlin Multiplatform Mobile vyvíjet multiplatformní aplikaci jak pro Android, tak i pro iOS [10].

Již po spuštění Android Studia je patrná jistá grafická podobnost s IntelliJ IDEA. Není tomu náhodou. Google totiž spolupracuje s firmou JetBrains, což je právě tvůrce zmíněného IDE [6] [11]. Google se přidáváním různých rozšíření snaží udělat vývoj aplikací co nejednoduší. Při založení projektu se například automaticky vytvoří potřebná struktura projektu, přednastaví se *Gradle* soubory pro práci se základními knihovnami a vygeneruje se *AndroidManifest.xml*, což je velmi důležitý soubor pro správu oprávnění aplikace a jejích aktivit. Velmi jednoduše lze vytvořit i Android emulátor, kde je poměrně velký výběr zařízení i s možností volby verze Android systému. Velmi užitečný je i jednoduchý přístup ke Google službám a až stovky možností pro inspekci kódu (funkce inspekce kódu slouží pro hledání chyb či nedostatků v kódu a je již obsažena v IntelliJ IDEA), které jsou přímo uzpůsobeny pro Android aplikace [11].

5.1.2 Metriky

Níže popsané metriky slouží pro číselnou reprezentaci úspory při používání Kotlinu místo Javy. Jak již bylo zmíněno v kapitole 5.1, při vývoji datových tříd by se v případě Javy mohlo využít *record* tříd. Bohužel však Android SDK podporuje zatím Javu 8, a tudíž nelze *record* třídy použít, jelikož jsou dostupné až od Javy 14.

5.1.2.1 Metrika na základě počtu znaků

Na základě této metriky je možné poukázat na to, o kolik méně znaků (textu) programátor napíše při vývoji v Kotlinu v porovnání s Javou u kódu s totožnou funkcionalitou.

Tabulka 1 Počet znaků u jednotlivých tříd v Kotlinu a Javě

Třída	**Kotlin**	**Java**
BookItemBase (D)	359	977
BookItemInteface (D)	110	111
LibraryBookItem (D)	1853	2144
BookItem (D)	1730	2151
Genre (D)	293	433
BookDetailActivity (UI)	9298	9537
MainActivity (UI)	1782	1942
LoginActivity (UI)	639	722
MyLibraryFragment (UI)	1677	1753
ReservedFragment (UI)	1745	1854
LibraryFragment (UI)	2413	2641
LibraryViewModel (VM)	872	1155
MyLibraryViewModel (VM)	746	1093
ReservedViewModel (VM)	751	975
BookItemAdapter (A)	2234	2802
MyLibraryBookItemAdapter (A)	4351	5068
BookDetailActivityUtil (U)	899	1049
BookItemUtil (U)	1143	1239
LibraryRepository (R)	3537	4134
ReservedRepository (R)	1635	2173
MyLibraryRepository (R)	2650	3255

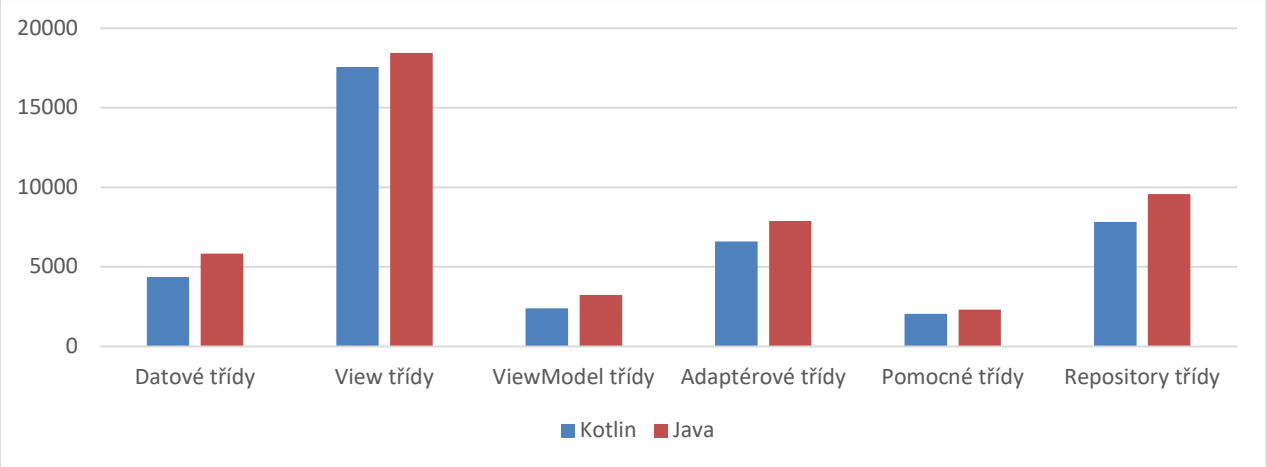
Zdroj: autor

Tabulka 2 Procentuální úspora v počtu znaků podle typu třídy

Typ třídy	Kotlin	Java	Procent. úspora
Datové třídy	4345	5816	25,3 %
View třídy	17554	18449	4,9 %
ViewModel třídy	2369	3223	26,5 %
Adaptérové třídy	6585	7870	16,3 %
Pomocné třídy	2042	2288	10,8 %
Repository třídy	7822	9562	18,2 %
Celkem	**40717**	**47207**	**13,7 %**

Zdroj: autor

Graf 1 Počet znaků podle typu třídy v Kotlinu a v Javě



Zdroj: autor

5.1.2.2 Metrika na základě počtu řádků

Tato metrika slouží pro poukázání na možné zvýšení přehlednosti kódu při používání Kotlinu v porovnání s Javou.

Tabulka 3 Počet řádků u jednotlivých tříd v Kotlinu a v Javě

Třída	**Kotlin**	**Java**
BookItemBase (D)	9	46
BookItemInteface (D)	5	6
LibraryBookItem (D)	60	87
BookItem (D)	54	85
Genre (D)	19	25
BookDetailActivity (UI)	252	246
MainActivity (UI)	46	52
LoginActivity (UI)	24	30
MyLibraryFragment (UI)	50	51
ReservedFragment (UI)	53	58
LibraryFragment (UI)	71	78
LibraryViewModel (VM)	29	43
MyLibraryViewModel (VM)	27	40
ReservedViewModel (VM)	28	39
BookItemAdapter (A)	58	87
MyLibraryBookItemAdapter (A)	103	142
BookDetailActivityUtil (U)	27	38
BookItemUtil (U)	42	41
LibraryRepository (R)	88	103
ReservedRepository (R)	51	69
MyLibraryRepository (R)	66	89

Zdroj: autor

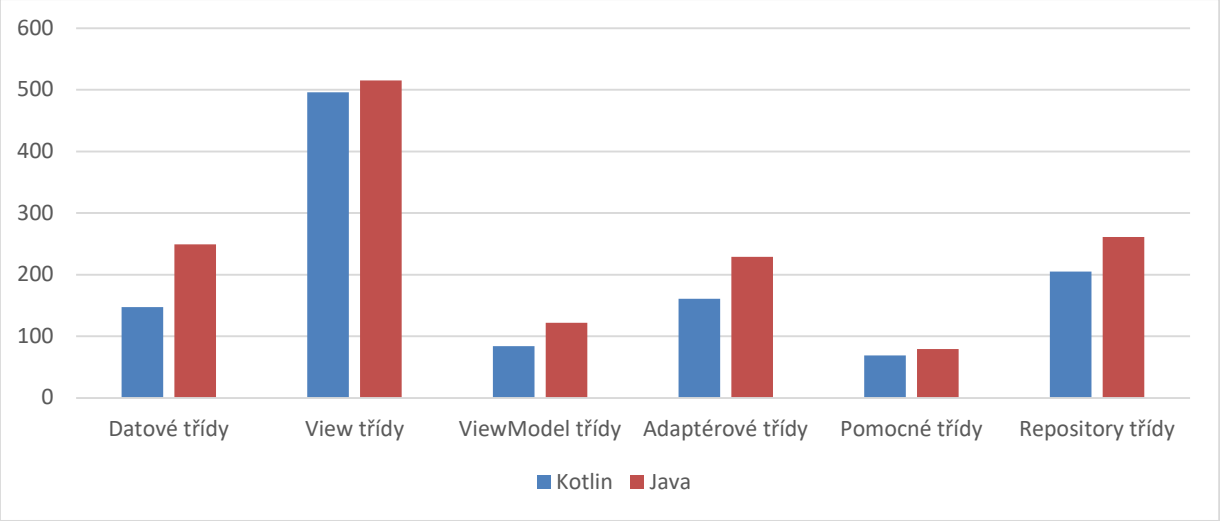
Zkratky v závorkách slouží pro kategorizaci tříd, kdy **D** jsou třídy datové, **UI** jsou třídy grafické (view), **VM** jsou třídy implementující ViewModely, **A** třídy sloužící jako adaptéry, **U** jsou třídy pomocné a **R** jsou repository třídy.

Tabulka 4 Procentuální úspora v počtu řádků podle typu třídy

Typ třídy	Kotlin	Java	Procent. úspora
Datové třídy	147	249	41 %
View třídy	496	515	3,7 %
ViewModel třídy	84	122	31,2 %
Adaptérové třídy	161	229	29,7 %
Pomocné třídy	69	79	12,7 %
Repository třídy	205	261	21,5 %
Celkem	**1162**	**1455**	**20,1 %**

Zdroj: autor

Graf 2 Počet řádků podle typu třídy v Kotlinu a v Javě

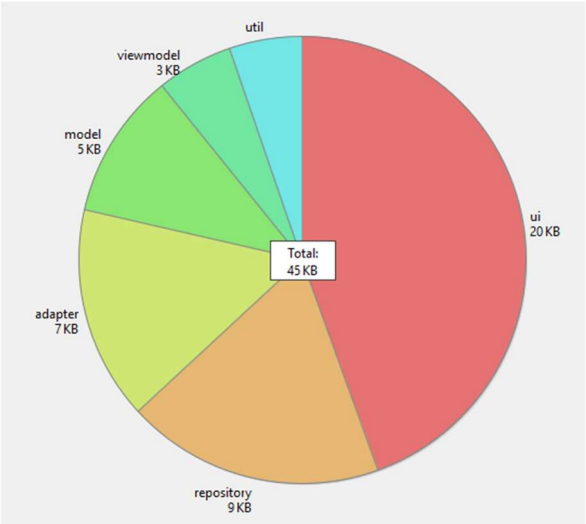


Zdroj: autor

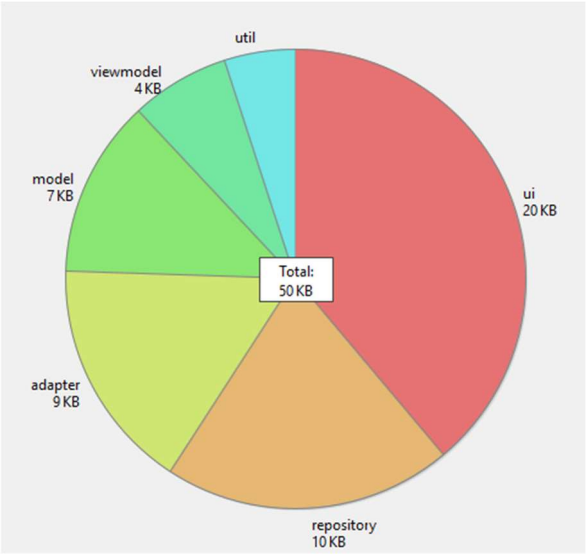
5.1.3 Velikosti souborů

5.1.3.1 Zdrojové kódy

Velikosti zdrojových kódů se u obou aplikací do jisté míry liší, nejedná se však o nic příliš zásadního. U obrázků č. 5 a 6 lze vidět, že velikost souborů vytvořených v Kotlinu je celkově pouze o 5 KB větší. Největší rozdíl 2 KB je u složek s modelovými a adaptérovými třídami.



Obrázek 5 Velikost obsahu složek uvnitř src složky v Javě
Zdroj: autor



Obrázek 6 Velikost obsahu složek uvnitř src složky v Kotlinu
Zdroj: autor

5.1.3.2 Balíčky APK a AAB

APK (Android application Package) je balíčkovací formát, díky kterému je možné instalovat hotové aplikace do koncových zařízení. AAB (App Bundles) je, na rozdíl od APK, publikační formát, u kterého se po nahrání na Google Play vytvoří sada několika optimalizovaných APK balíčků. Například pro konkrétní rozlišení zařízení či pro konkrétní funkcionalitu. Uživatel tedy nemá ve staženém balíčku zbytečné soubory (např. obrázky s příliš velkým či malým rozlišením), a díky tomu zabere aplikace v uložišti mnohdy výrazně méně místa.

Podle tabulky 5.1.3 je zřejmé, že APK a ani AAB balíčky z Kotlinové verze aplikace nemají výrazně větší velikost. Jedná se pouze o rozdíl 95 KB v případě APK balíčku a 96 KB u AAB balíčku.

Tabulka 5 Velikost balíčků v závislosti na programovacím jazyku v kilobajtech

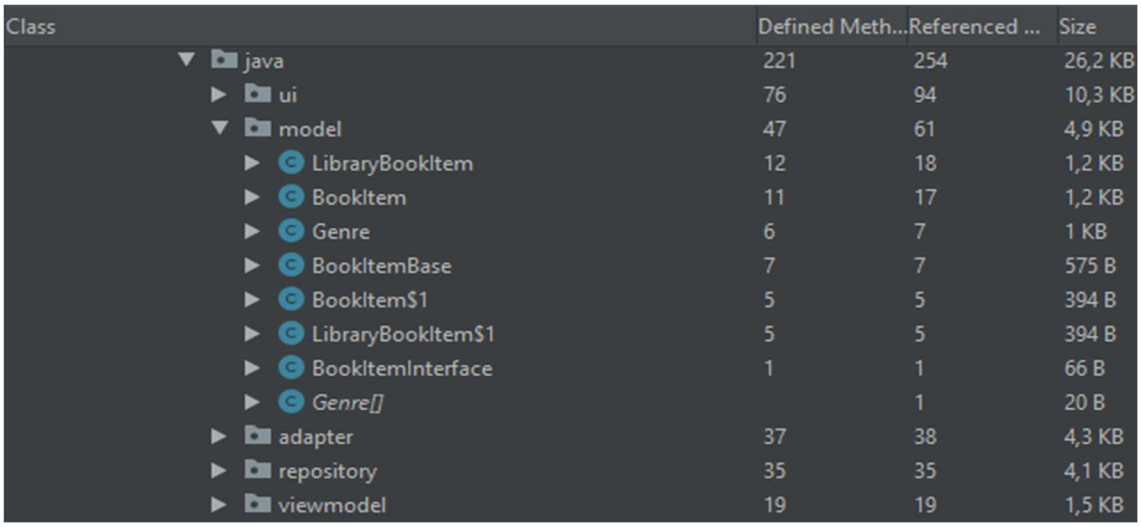
Typ balíčku	Java	Kotlin	Rozdíl
APK	4670	4765	95
AAB	3990	4086	96

Zdroj: autor

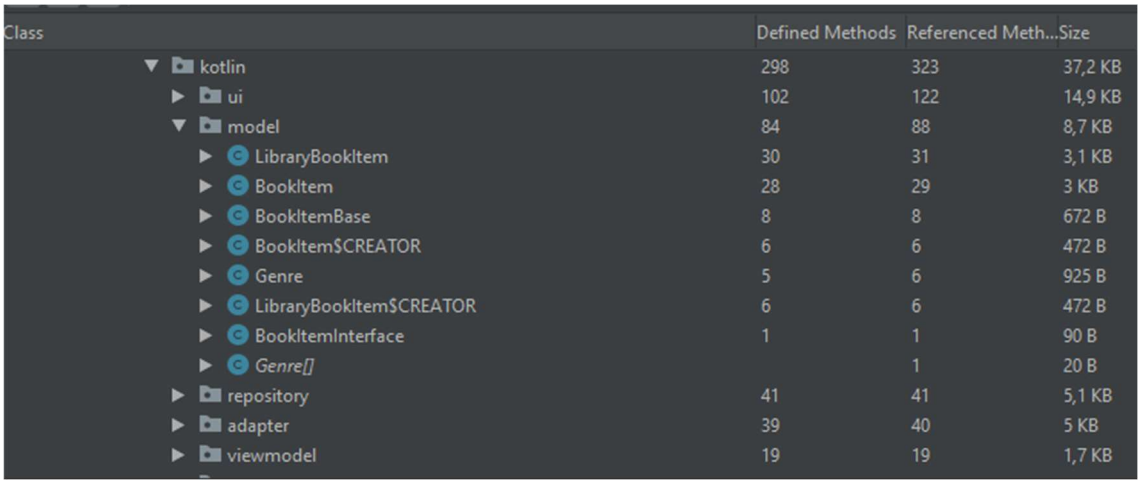
5.1.3.3 Bytecode

Na základě výpisu tříd ze zkompilovaného APK balíčku byly vybrány 3 modelové třídy podle největšího rozdílu velikosti (viz obrázky 7 a 8) pro zjištění, o kolik znaků je bytecode vygenerovaný z Kotlinových tříd větší než bytecode vzniklý z tříd napsaných v Javě. Jedná se o třídy LibraryBookItem, BookItem a BookItemBase. Dle tabulky 5.13 je Kotlinová verze BookItemBase přibližně o 178 % větší. Mnohem výraznější zvětšení je u BookItem, kde činí 383 % a u LibraryBookItem dokonce 413 %.

Bytecode vygenerovaný Kotlinem má až několikanásobně více znaků. Nicméně, co se týče velikosti vygenerovaných bytecode souborů, jedná se vždy o zanedbatelný rozdíl několika kilobajtů. V praxi to tedy znamená, že by musela být aplikace obsáhlejší, aby se rozdíl vůbec pohyboval v řádech megabajtů.



Obrázek 7 Výpis balíčků a tříd získaných z APK (Java)
Zdroj: autor



Obrázek 8 Výpis balíčků a tříd získaných z APK (Kotlin)
Zdroj: autor

Tabulka 6 Počet znaků bytecodu a procent. zvětšení v Kotlinu oproti Javě

Třída	Kotlin	Java	Procent. zvětšení
BookItemBase	6452	3620	178 %
LibraryBookItem	32625	7892	413 %
BookItem	31262	8154	383 %

Zdroj: autor

5.2 Spring Boot

Záměrem bylo i porovnání obou jazyků při vývoji webových aplikací s využitím Springu. Avšak používání knihovny Spring je v obou jazycích velmi podobné a rozdíly jsou pouze v syntaxi těchto jazyků, které byly popsány výše v kapitole 4.

6 Shrnutí výsledků

Na základě tabulek a grafů z kapitoly 5.1.2 lze říci, že při používání Kotlinu vývojář napíše výrazně méně kódu, než kdyby používal Javu. Co se týče celkového počtu znaků, jedná se o přibližně 13,7% úsporu a u počtu řádků až o 20,1% úsporu, což může být u větších aplikací podstatně znát.

Velmi podstatná úspora byla zaznamenána hlavně u datových tříd, kdy počet znaků byl až o 25 % nižší než u Javy a počet řádků dokonce o 41 % nižší. Těchto hodnot bylo dosaženo hlavně díky absenci přístupových metod a primárního konstruktoru u Kotlinu. Kdyby bylo nutné implementovat i *equals()*, *hashCode()* či *toString()* metody, mohla by být úspora jistě o pár jednotek procent vyšší. Je však potřeba i zmínit, že pokud by se dalo využít v Javě *record* tříd, nebyla by úspora tak vysoká, jelikož lze *record* třídu v Javě implementovat stejně jednoduše jako datovou třídu v Kotlinu.

Výraznou úsporu lze vidět i u viewmodelových tříd či adaptérových tříd. Značný vliv na to má opět absence konstruktoru a výčtu atributů těchto tříd, jelikož lze atributy třídy uvést rovnou za název třídy, čímž se rovnou implicitně vytvoří i primární konstruktor. U viewmodelových tříd měl významný vliv i zápis funkcí do jednoho řádku, vynechávání public modifikátoru u veřejných metod a datového typu návratové hodnoty (díky typové inferenci).

U pomocných tříd je to velmi závislé na možnostech zkrácení zápisu jednotlivých jazyků a primárně na tom, jaké funkce se v pomocných metodách používají. Jelikož se však jedná o třídy se statickými funkcemi, nelze od Kotlinu očekávat výrazný rozdíl. Nejvíce výhod má totiž u tříd, kde je nutné vytvářet instance a definovat nějaké vlastnosti. Pokud by se však jednalo o třídu s velkým množstvím statických funkcí, mohla by být změna výraznější právě díky bloku *companion object*, který shromažďuje veškeré statické funkce, a nebylo by tedy nutné ke všem funkcím přidávat modifikátor *static* jako v případě Javy. Navíc pomocné funkce bývají většinou veřejné, tudíž nebude třeba u každé funkce zmiňovat *public* modifikátor, což bude mít s počtem přibývajících funkcí též značný vliv.

Co se týče velikostí, ať už zdrojových souborů, APK balíčků či bytecodu, nejedná se o žádné výrazné zvětšení, které by mělo vývojáře nějak znepokojovat. Vždy byly rozdíly Kotlinových souborů vyšší většinou o jednotky kilobajtů a v případě kompletních APK či AAB balíčku se blížil rozdíl jen ke 100 kilobajtům (viz kapitola 5.1.3). V praxi by musela být aplikace výrazně rozsáhlejší, aby dosahovala rozdílů v řádech megabajtů. Ani tak by to však nemělo významný vliv pro vývojáře či pro koncové uživatele aplikace.

7 Závěry a doporučení

Kotlin se osvědčil jako velmi praktický a pro vývoj úsporný jazyk, kdy v rámci celého praktického projektu bylo možné napsat téměř o 14 % méně kódu a až o 20 % méně řádků než u Javy, což má podstatný vliv na přehlednost kódu. Nejvíce úspory bylo zaznamenáno u datových tříd, které jsou součástí všech typů aplikací, ať už desktopových, webových či mobilních. U desktopových a webových aplikací by však šlo využívat nové verze Javy, a díky tomu by nebyla úspora v Kotlinu tak rozdílná. Dále byl podstatně úspornější kód i u viewmodelových tříd, které jsou však spíše typické pro Android aplikace.

U porovnání velikostí obou aplikací nebylo v žádné z kategorií dosaženo významného rozdílu. V případě, že se vývojář rozhoduje, který z programovacích jazyků použít, nedají se tyto nepatrné rozdíly považovat za příliš velký ukazatel kvality programovacího jazyka.

Co se týče možností jazyků, velmi užitečná je null-safety vlastnost, kterou Kotlin disponuje a v praxi dokáže předcházet vzniku výjimky *NullPointerException*. Tyto výjimky jsou velmi těžké na odhalení, jelikož se jedná o *runtime* výjimku, kterou je možné objevit jen za běhu aplikace. Za zmínku jistě stojí i rozšiřující funkce, které se dají využít hlavně k přidávání funkcí již vytvořených knihoven či funkcionální konstrukty, které Kotlin jakožto i funkcionální jazyk rovněž nabízí.

Bohužel je syntaxe Kotlinu poněkud nezvyklá a pro Java vývojáře nebo C# vývojáře může být přechod na Kotlin zprvu složitější. Nicméně, velkou výhodou je vzájemná interoperabilita Javy a Kotlinu, která zaručuje obrovskou komunitu vývojářů, která je při objevování nových programovacích jazyků vždy velmi důležitá.

Pro vývojáře, kteří už mají základní zkušenosti s jakýmkoliv programovacím jazykem, je Kotlin rozhodně skvělou volbou. Pro někoho, kdo se programování teprve učí, je však lepší volbou Java, jejíž syntaxe je přirozenější a hlavně lze vidět, jaké principy je třeba dodržovat a čemu se při vývoji nelze vyhnout, jelikož to za programátora neudělá kompilátor.

Je na každém vývojáři, jaký programovací jazyk si zvolí pro vývoj své aplikace, jisté je však to, že Kotlin mu ušetří práci, udrží přehledný kód a zaručí bezpečnost, co se null-safety týče.

8 Seznam použité literatury

[1] Liam Tung. Programming languages: Kotlin rises fastest but JavaScript lures milions more developers. In: zdnet.com [online]. 17. 4. 2020 [cit. 14. 4. 2021]. Dostupné z: https://www.zdnet.com/article/programming-languages-javascript-now-used-by-12-million-developers-but-kotlin-rises-fastest/

[2] IEEE SPECTRUM. Interactive: The Top Programming Languages. In: spectrum.ieee.org [online]. 2020 [cit. 14. 4. 2021]. Dostupné z: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020

[3] SOUHRADA, Václav. Kotlin – jazyk, který je dobré znát. In: Eman.cz [online]. 12. 7. 2017 [cit. 8. 8. 2020]. Dostupné z: https://www.eman.cz/blog/kotlin-jazyk-ktery-dobre-znat/

[4] KODYTEK, Samuel. Úvod do jazyka Kotlin, platformy a IntelliJ. In: itnetwork.cz [online]. 1. 4. 2018 [cit. 8. 8. 2020]. Dostupné z: https://www.itnetwork.cz/kotlin/zaklady/uvod-do-jazyka-kotlin-platformy-a-intellij

[5] SAMUEL, Stephen, BOCUTIU, Stefan. Programming Kotlin. Packt Publishing. 2017. ISBN 9781787126367

[6] LEIVA, Antonio. Kotlin for Android Developers. Createspace Independent Publishing Platform. 2016. ISBN 9781530075614

[7] Baeldung. Avoid Check for Null Statement in Java. In: baeldung.com [online]. 6. 1. 2021 [cit. 14. 4. 2021]. Dostupné z: https://www.baeldung.com/java-avoid-null-check

[8] JetBrains s. r. o. Kotlin Reference. In: kotlinlang.org [online]. 2011-2020 [cit. 8. 8. 2020]. Dostupné z: https://kotlinlang.org/docs/reference/

[9] Google Inc. Use Java 8 language features and APIs. In: developer.android.com [online]. 13. 03. 2021 [cit. 14. 4. 2021]. Dostupné z: https://developer.android.com/studio/write/java8-support

[10] JetBrains s. r. o. Create your firt multiplatform application. In: kotlinlang.org [online]. 1. 3. 2021 [cit. 23. 3. 2021]. Dostupné z: https://kotlinlang.org/docs/mobile/create-first-app.html

[11] Xavier Ducrohet. Android Studio: An IDE built for Android. In: android-developers.googleblog.com [online]. 15. 5. 2013 [cit. 23. 3. 2021]. Dostupné z: https://android-developers.googleblog.com/2013/05/android-studio-ide-built-for-android.html

9 Seznam použitých zdrojů

[1] JetBrains s. r. o. Kotlin collection interfaces hierarchy [obrázek].
In: kotlinlang.org [online]. ©2011-2020 [cit. 09. 01. 2021].
Dostupné z: https://kotlinlang.org/docs/reference/collections-overview.html
[2] WOXAPP The use of the ModelViewViewModel pattern on Android [obrázek].
In: woxapp.com [online]. ©2011-2020 [cit. 24. 03. 2021].
Dostupné z: https://woxapp.com/our-blog/the-use-of-the-modelviewviewmodel-pattern-on-android

10 Přílohy

[1] SCHMID, Daniel. Praktický projekt E-Library vytvořený v Kotlinu.
In: github.com [online]. 22. 4. 2021 [cit. 22. 4. 2021].
Dostupné z: https://github.com/dandys71/E-Library_Kotlin
[2] SCHMID, Daniel. Praktický projekt E-Library vytvořený v Javě.
In: github.com [online]. 22. 4. 2021 [cit. 22. 4. 2021].
Dostupné z: https://github.com/dandys71/E-Library_Java

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor: **Daniel Schmid**

Studium: I1800224

Studijní program: B1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název bakalářské práce: **Porovnání jazyků Java a Kotlin**

Název bakalářské práce AJ: Comparison of Java and Kotlin languages

Cíl, metody, literatura, předpoklady:

Cíl práce: Porovnání syntaxe jazyků a jejich jednoduchosti implementace při vývoji Android aplikací a webových aplikací za pomoci knihovny Spring.

Osnova práce:

1. Úvod
2. Metodika porovnání
3. Charakteristika jazyků Java a Kotlin
4. Popis testovacího prostředí/aplikace
5. Výsledky porovnání a diskuse
6. Závěr

1. SAMUEL, Stephen, BOCUTIU, Stefan. Programming Kotlin. Packt Publishing. 2017. ISBN 9781787126367
2. LEIVA, Antonio. Kotlin for Android Developers. Createspace Independent Publishing Platform. 2016. ISBN 9781530075614
3. SOUHRADA, Václav. Kotlin – jazyk, který je dobré znát. In: Eman.cz [online]. 12. 7. 2017 [cit. 8. 8. 2020]. Dostupné z: https://www.eman.cz/blog/kotlin-jazyk-ktery-dobre-znat/
4. KODYTEK, Samuel. Úvod do jazyka Kotlin, platformy a IntelliJ. In: itnetwork.cz [online]. 1. 4. 2018 [cit. 8. 8. 2020]. Dostupné z: https://www.itnetwork.cz/kotlin/zaklady/uvod-do-jazyka-kotlin-platformy-a-intellij
5. JetBrains s. r. o. Kotlin Reference. In: kotlinlang.org [online]. {\copyright 2011-2020 [cit. 8. 8. 2020]. Dostupné z: https://kotlinlang.org/docs/reference/
6. JetBrains s. r. o. Kotlin collection interfaces hierarchy [obrázek]. In: kotlinlang.org [online]. {\copyright 2011-2020 [cit. 23. 8. 2020]. Dostupné z: https://kotlinlang.org/docs/reference/collections-overview.html

Garantující pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Datum zadání závěrečné práce: 7.8.2020