Jihočeská univerzita
v Českých Budějovicích
University of South Bohemia
in České Budějovice

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

Funkcionální programování v Javě

Functional Programming in Java

Bakalářská práce

Vypracoval: Milan Vysocký

Vedoucí práce: RNDr. Hana Havelková

České Budějovice 2019

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

Fakulta pedagogická

Akademický rok 2018/2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a Příjmení:	Milan Vysocký
Osobní číslo:	P16953
Studijní program:	B7507 Specializace v pedagogice
Studijní obor:	Informační technologie a e-learning
Název tématu:	Funkcionální programování v Javě
Zadávající katedra:	Katedra informatiky

Zásady pro vypracování

Cílem práce je ukázat možnosti funkcionálního programování v Javě. Autor objasní problematiku funkcionálního programování, jeho základní principy, ukáže možnosti jeho realizace v Javě. Představí poměrně nové jazykové rysy Javy, jako jsou např. datovody, lambda výrazy, implicitní metody, představí stream API.

Prostřednictvím těchto prvků autor představí možnosti deklarativního programování, na praktických řešených příkladech předvede možnosti jejich použití a analyzuje jejich implementaci.

Na platformě Javy autor provede srovnání možností objektově orientovaného (imperativního) programování a funkcionálního (deklarativního) programování, zváží výhody a nevýhody jednotlivých přístupů.

Rozsah: 40 stran

Literatura:

[1] *Java Documentation* [online]. Oracle, 2018. Dostupné z: https://docs.oracle.com/javase/8/docs/

[2] PECINOVSKÝ, Rudolf. *Java 8: úvod do objektové architektury pro mírně pokročilé*. Praha: Grada Publishing, 2014. ISBN 978-80-247-4638-8.

[3] PECINOVSKÝ, R. Kontejnery a datovody. 2014. [Online] Dostupné z: http://www.common.cz/attachments/article/723/Rudolf_Pecinovsky_Kontejnery_a_datovody.pd

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne 14. dubna 2019.

Milan Vysocký

Abstrakt

Cílem této bakalářské práce je v teoretické i praktické části představit funkcionální (deklarativní) programování se všemi jeho hlavními rysy, analyzovat jeho implementaci a porovnat ho s objektově orientovaným (imperativním) programování.

Teoretická část je zaměřena na funkcionální programování, kde budou objasněny jeho základní principy, realizace v Javě a důvody, proč tento druh programování vůbec vznikl. Pomocí poměrně nových rysů jazyka tzn. lambda výrazy, datovody, implicitní metody, funkční rozhraní, jejichž implementace bude mimo jiné také dopodrobna analyzovaná, budou představeny možnosti funkcionálního programování v Javě. Pomocí srovnání mezi funkcionálním programováním a programováním objektově orientovaným se zváží výhody a nevýhody jednotlivých přístupů.

V praktické části bakalářské práce bude naprogramováno několik vlastních praktických řešených příkladů, na kterých je vidět použití všech výše uvedených rysů jazyka.

Klíčová slova

Funkcionální programování, Deklarativní programování, Lambda výrazy, Implicitní metody, Datovody, Funkční rozhraní, Java 8

Abstract

The aim of this bachelor thesis is to introduce functional (declarative) programming with all its main features in the theoretical and practical part, analyze its implementation and compare it with object oriented (imperative) programming.

The theoretical part is focused on functional programming, where its basic principles will be clarified, implementation in Java and why this programming paradigm was developed. Using relatively new language features, i.e. lambda expressions, streams, default methods, functional interfaces, whose implementation among other things, will be analyzed, will be introduced the possibilities of functional programming in Java. By comparing functional programming and object-oriented programming, the advantages and disadvantages of each approach are considered.

In the practical part of the bachelor thesis there are programmed several practical examples documenting the possibility of using all the language features mentioned above.

Keywords

Functional programming, Declarative programming, Lambda expressions, Default methods, Streams, Functional interface, Java 8

Poděkování

Děkuji paní RNDr. Haně Havelkové za veškerou pomoc a trpělivost při psaní této bakalářské práce. Velmi děkuji za všechny rady, nápady, připomínky, které mně pomohly při realizaci této práce. Velmi si vážím podpory a laskavosti, kterou mně po celou dobu poskytovala.

Poděkování také patří mé sestře, která mně byla vždy na blízku, když jsem potřeboval povzbudit. Poděkování patří i mým rodičům, kteří mě podporují při mém studiu.

Obsah

1 Úvod **10**
1.1 Cíle práce 10
1.2 Struktura práce 10
2 Funkcionální programování **12**
2.1 Charakteristika funkcionálního programování 14
2.1.1 Funkce první třídy, funkce vyššího řádu 14
2.1.2 Neměnná data 15
2.1.3 Čisté funkce 15
2.1.4 Rekurze 15
2.1.5 Manipulace se seznamy 16
2.1.6 Odložené vyhodnocování 16
2.2 Jak vzniklo funkcionální programování 16
2.2.1 Lambda kalkul (syntaxe) 17
2.3 Výhody a nevýhody funkcionálního programování 18
2.3.1 Výhody 18
2.3.2 Nevýhody 20
3 Prvky Javy jako prostředek funkcionálního programování **22**
3.1 Funkční rozhraní 22
3.2 Implicitní metody 25
3.3 Lambda výrazy 27
3.3.1 Syntaxe 27
3.4 Datovody 30
3.4.1 Druhy operací 31
3.4.2 Průběžné operace 32
3.4.2.2 Map 33
3.4.2.3 Limit 34
3.4.2.4 Sorted 34
3.4.2.5 Ostatní důležité průběžné operace 36
3.4.3 Koncové operace 36
3.4.3.1 ForEach 36
2.4.3.2 Count 37
2.4.3.3 Collect 37
2.4.3.4 Ostatní důležité koncové operace 38
4 Funkcionální vs. objektově orientované programování **39**

4.1 Funkce jako hodnoty proměnných a parametry funkcí 39
4.2 Odložené vyhodnocování 40
4.3 Manipulace se seznamy 41
4.4 Rekurze vs iterace 43
4.5 Neměnná data 44
4.6 Hledání chyb 45
4.7 Shrnutí 45
5 Aplikace 47
5.1 Videopůjčovna 47
5.2 Převodník soustav 51
5.3 Operace s datovody 52
6 Závěr 55

1 Úvod

Tato bakalářská práce se zabývá problematikou deklarativního programování, jež ke své implementaci využívá poměrně nové jazykové rysy z osmé verze Javy tj. lambda výrazy, funkční rozhraní, implicitní metody a datovody. Mimo informací o funkcionálním programování, jeho základních charakteristikách, výhodách a nevýhodách a popisu implementace nových jazykových prvků, je srovnáno v teoretické části funkcionální programování s objektově orientovaným programováním.

V praktické části jsou ukázány tyto prvky na praktických řešených příkladech, jakožto prostředníci funkcionálního programování. Jedná se o rozsáhlejší příklad (aplikaci pracující s databází), na němž je ukázána práce s datovody, převodník soustav, jako ukázka práce s funkčními rozhraními a implicitními metodami a nakonec příklad s implementací datovodů, funkčních rozhraní a implicitních metod. Důležité je také sdělit, že veškeré aplikace jsou naprogramovány i bez těchto prvků.

1.1 Cíle práce

Bakalářská práce Funkcionální programování v Javě má za cíl představit tento druh programování, označovaného též jako deklarativní, i s jeho typickými rysy a porovnat ho s objektově orientovaným programováním. Práce je rozdělena na část teoretickou a na část praktickou. Čistá teoretická část bakalářské práce je rozdělena do třech kapitol, a sice funkcionální programování, jeho hlavní jazykové rysy jako jsou lambda výrazy, funkční rozhraní, datovody, implicitní metody a srovnání mezi funkcionálním programováním a objektově orientovaném programováním. V praktické části je na toto téma představeno několik příkladů, které by měly čtenáři objasnit a pomoci pochopit problematiku funkcionálního programování i se všemi jeho jednotlivými prvky.

1.2 Struktura práce

Ze začátku je objasněna problematika funkcionálního (deklarativního) programování. V následující kapitole jsou popsány poměrně nové jazykové rysy Javy, které se nejčastěji spojují právě s funkcionálním programováním. To jsou zejména lambda výrazy, funkční interfejsy (rozhraní), které byly zavedeny hlavně kvůli lambda výrazům, datovody, nebo implicitní metody. Všechny tyto rysy funkcionálního programování jsou podrobně rozebrány, je popsána jejich implementace a jsou zvážené

jednotlivé výhody a nevýhody. Dále následuje kapitola, kde je porovnáno objektově orientované (imperativní) programování s funkcionálním (deklarativním) programováním. K tomuto úkonu jsou právě k dispozici aplikace a příklady, které jsou záměrně naprogramovány oběma způsoby. Poté následuje praktická část, která je složena z několika delších i kratších příkladů. Je zde například databáze, na které je ukázána práce s datovody, jež obsahují již nadefinované metody, které umožňují s daty provádět různé operace skoro jako jazyk SQL. Je zde vidět i práce s lambda výrazy. Lambda výrazy s implicitními metodami a funkčními rozhraními jsou vysvětleny na kratších praktických příkladech. To ovšem neznamená, že jsou všechny aplikace naprogramovány pouze funkcionálním způsobem. Jednotlivé příklady jsou naprogramovány i bez těchto typických prvků, které se používají při funkcionálním programování, tedy jsou naprogramovány jak funkcionálním způsobem, tak způsobem objektově orientovaným. Všechny praktické příklady jsou dokumentovány a nahrány na přiložené CD, které je použité jako příloha k této bakalářské práci.

2 Funkcionální programování

Funkcionální paradigma je založeno na zápisu programu ve tvaru výrazu namísto příkazů. Nejdůležitějšími složkami těchto výrazů jsou funkce, které lze předat jako argument jiné funkce. Výraz se používá, aby vytvořil hodnotu, zatímco příkaz se používá pro přiřazení hodnoty. Funkcionální (deklarativní) programování se vyznačuje hlavně tím, že se nezabývá postupným vypracováním programu krok po kroku, to znamená, že nemusí psát dlouhé kódy, kde je popsán celý postup, jak se má zadaný program řešit – to ho v podstatě vůbec nezajímá [1]. To je ponecháno různým nástrojům, které prostředí jazyka poskytují. Díky nim pro nás řada pohledů zůstává skryta. Zatímco deklarativní zápis programu představuje již samotné řešení problému zapsané v jistém tvaru [1].

Výpočet funkcionálního programu spočívá ve zjednodušování výrazu až do doby, kdy výraz dále zjednodušit nelze. Tento dále nezjednodušitelný tvar je výsledkem výpočtu [1]. Jinými slovy běh funkcionálního programu je založen na principu skládání funkcí. Skládání znamená, že můžeme spojit více funkcí dohromady, tedy argumentem funkce může být nějaká jiná funkce.

V tomto druhu programování se hodně vyskytuje tzv. rekurze. Jde vlastně o funkci (metodu), která volá sama sebe. Díky rekurzi se ve funkcionálním programování nemusí tak využívat cykly (for, do-while, while). Pokud se jedná o proměnné, tak ani ty se zde nevyužívají tak často. Bude-li se chtít v jisté části programu použít nějaká hodnota, tak namísto použití proměnné, se použije naimplementovaná metoda se správným návratovým typem. Čím se hlavně vyznačuje funkcionální programování, jsou tzv. lambda výrazy. Ty umožňují předávat nějakou část zdrojového kódu podobně jako proměnné. Část kódu, která by zabrala i několik řádků, se tak dokonce může vejít pouze na jednu.

Výsledný kód je pro laického programátora často velmi nečitelný. Ale výsledný kód bývá často mnohem kratší a spolehlivější, než odpovídající kód zapsaný příkazovým stylem. A právě posledně zmiňované kvality přitáhly ke studiu funkcionálního programování mnoho programátorů, kteří používali procedurální nebo objektově orientovaný způsob programování [2].

O funkcionálním (deklarativním) programování v jazyce Java se dalo pořádně mluvit, až když vyšla jeho osmá verze. Ta do tohoto jazyka totiž přinesla jednu z mnoha

novinek, bez které by se funkcionální programování jen tak neobešlo. V tomto případě se mluví o lambda výrazech. Dalšími novinkami, jež přinesla osmá verze jazyka, a podporují funkcionální paradigma v Javě, jsou funkční rozhraní, implicitní metody a datovody.

Příkladem funkcionálního kódu je následující příklad, který umožňuje, zjistit celkový počet sudých čísel v Arraylistu. Velmi nápomocné jsou zejména datovody, které jsou popsány ve třetí kapitole.

```
List<Integer> cisla = Arrays.asList(1, 5, 9, 8, 6, 2, 15, 65);
        long pocet = cisla.stream()
                          .filter(cislo -> (cislo % 2 ==0))
                          .count();
```

Příklad 1: Ukázka funkcionálního kódu

Naopak bez použití datovodů by se musel vytvářet forEach cyklus, podmínka a iterativně zvyšovat proměnnou *pocet*.

```
List<Integer> cisla = Arrays.asList(1, 5, 9, 8, 6, 2, 15, 65);
        int pocet = 0;
        for(int cislo : cisla){
            if(cislo % 2 == 0){
                pocet++;
            }
        }
```

Příklad 2: Ukázka imperativního kódu

Bude-li se chtít programovat deklarativním způsobem, existují dvě možnosti. Tou první je použití jazyka ze skupin programovacích jazyků, které byly navrženy přímo pro deklarativní programování [3]. Mezi čistě funkcionální jazyky lze určitě zařadit takový Haskell, FP, Miranda nebo Hope. V programech nebývá důležité přesné pořadí jednotlivých pravidel, protože kód nebývá zpracováván lineárně tak, jak tomu bývá u imperativních programovacích jazyků [3].

Druhým přístupem je použití imperativního jazyka s knihovnou pro podporu deklarativního programování. Jde vlastně jen o skrytí imperativních částí programového kódu a vlastní použití takovéto knihovny je v duchu deklarativního programování [3]. Mezi tyto jazyky patří například Java, C#, Python nebo C++.

2.1 Charakteristika funkcionálního programování

Obrázek 1: Charakteristiky funkcionálního programování [1]

2.1.1 Funkce první třídy, funkce vyššího řádu

Funkcionální programování se vyznačuje několika charakteristikami. Prvním prvkem jsou tzv. funkce první třídy. To jsou takové funkce, které lze bez starostí předávat jako argumenty funkcí, výsledky funkcí nebo je lze ukládat do proměnných. Existují i funkce druhé a třetí třídy. Od funkce první třídy se liší v tom, že funkce druhé třídy může být předána pouze jako argument a funkce třetí třídy nemůže být předána ani jako argument. Dalším prvkem jsou funkce vyššího řádu. Ty souvisí s funkcemi první třídy v tom, že argumentem nebo výsledkem metody nemusí být číselná hodnota nebo řetězec, ale funkce.

[1] *http://www.modernescpp.com* [online]. Dostupný na WWW: http://www.modernescpp.com/index.php/the-definition-of-functional-programming

Funkce vyššího řádu je možno vidět na příkladě níže, kde je vytvořen kolektor pro seřazení jmen.

```
Collections.sort(names, (first, second) -> first.length() - second.length());
```

Příklad 3: Ukázka funkce vyššího řádu [2]

2.1.2 Neměnná data

Velkou roli ve funkcionálním programování hrají i neměnná data. Jde například o objekt, jenž nelze po vytvoření již změnit. Hodnoty se můžou v programu v klidu předávat, aniž by nastal strach, že nějaká jiná funkce, změnila jejich obsah. Proměnnou hodnotu lze ošetřit modifikátorem přístupu typu *private*, zde je alespoň jistota, že hodnotu nezmění nějaká jiná funkce z jiné třídy. O neměnnosti dat se dá přesvědčit na následujícím obrázku, kde se v dané třídě implementuje abstraktní metoda z rozhraní a do proměnné *tvar*, která je již vytvořená uvnitř třídy, se uloží textový řetězec, pojednávající o implementaci k obvodu obdélníku. Kompilátor ovšem začne „nadávat“, že proměnná *tvar* nesmí být měněna nebo musí mít klíčové slovo *final*.

```
local variables referenced from a lambda expression must be final or effectively final
----
(Alt-Enter shows hints)
    tvar = "obdélník";
    System.out.println(2 * (a + b));
};
```

Obrázek 2: Hlídání neměnných dat

2.1.3 Čisté funkce

Čistá funkce se vyznačuje tím, že nemění hodnoty zadávaných parametrů a vrací hodnoty pomocí příkazu return. Výstup funkce nezávisí na jakémkoli stavu nebo na změnách dat během provádění programu. Musí záviset pouze na vstupních argumentech [4].

2.1.4 Rekurze

Jak už bylo psáno dříve, velmi důležitým znakem je i rekurze. O rekurzi lze mluvit tehdy, když nějaká funkce ve svém těle volá buď sama sebe, nebo nějakou jinou metodu, čímž dochází k opakování programu, a tak nahrazuje cykly.

[2] *www.github.com* [online]. Dostupný na WWW: https://github.com/shekhargulati/java8-the-missing-tutorial/blob/master/02-lambdas.md

2.1.5 Manipulace se seznamy

Ve funkcionálním programování se též může hodit jednodušší manipulace se seznamy. Protože zpracování seznamu je ve funkčním programování tak idiomatické, existují speciální názvy prvního prvku a zbytku seznamu: (x, xs), (head, tail) nebo (car, cdr) [5]. Manipulovat se seznamy a provádět nad nimi různé operace, jako je třeba filtrování, řazení nebo mapování, umožňují i tzv. datovody, o kterých je psáno ve třetí kapitole.

2.1.6 Odložené vyhodnocování

Další charakteristikou funkcionálního programování je odložené vyhodnocování. Již z názvu lze odvodit definici tohoto termínu. Ta říká, že vyhodnocení nějakého výrazu je odloženo až do okamžiku, kdy je to opravdu potřebné. Tato strategie může vést ke zmenšení zabrané paměti, protože hodnoty se vytvářejí, až když jsou potřebné [6]. "Předpokládejme, že váš program obsahuje tabulku s 5000 hodnotami. Tuto tabulku generuje ihned po spuštění a pak ji používá, kdykoli je to zapotřebí. Pokud ovšem program používá jen malé procento záznamů v tabulce, mohlo by se vyplatit, kdyby byly vypočteny až v okamžiku, kdy to bude potřeba. Jakmile je záznam vypočten, zůstane v paměti pro budoucí výpočty (v mezipaměti – „kešován“)" [7]. Stručně řečeno, když se ve výrazu objevuje lambda výraz, už se jedná o odložené vyhodnocování. I v případě kdy se ukládá hodnota do proměnné pomocí lambda výrazu. Tato hodnota bude do proměnné přiřazena, až když bude v kódu potřebná.

```
Rozhrani textova promenna = () -> "Text";
```

Příklad 4: Přiřazení hodnoty do proměnné

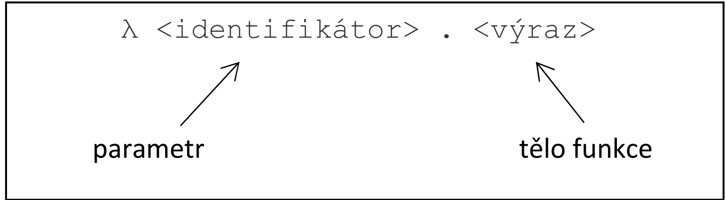
2.2 Jak vzniklo funkcionální programování

Za základ funkcionálního programování je považován tzv. lambda kalkul. V lambda kalkulu má člověk pro komunikaci se strojem jedinou rekvizitu – zápis funkce. Počítač tedy nedělá nic jiného, než vyčísluje (provádí) nařízené funkce, které mohou mít různé parametry. Parametrem jedné funkce může být jiná funkce, čímž může docházet i k rekurzím a cyklům [8].

2.2.1 Lambda kalkul (syntaxe)

Lambda kalkul využívá k práci konstrukce lambda výrazů, na nichž uplatňuje redukční operace. V lambda kalkulu jsou výrazy stavěny podle tří základních prvků: proměnná, aplikace, abstrakce.

V případě proměnné jde o znak nebo řetězec reprezentující blíže nespecifikovanou hodnotu. Běžně se může označovat identifikátorem x, y nebo z.



Obrázek 3: Ukázka syntaxe

Funkce v lambda kalkulu jsou reprezentovány ve formě abstrakce tvořené proměnnou označující parametr a tělem ve tvaru lambda výrazu [9]. Částí mezi λ a tečkou jsou parametry funkce a za tečkou se nachází tělo funkce [10]. Například funkce f(x) = x – 5. V lambda kalkulu to lze zapsat následovně:

```
λ x . x - 5
```

Příklad 5: Zápis funkce s jedním parametrem

Funkce z pátého příklad má jen jeden parametr. Co když bude existovat taková funkce, jež bude mít parametry dva, nebo dokonce i více. Například funkce f(x, y) = x * y. Pak její přepis v lambda kalkulu vypadá následovně:

```
λ x. λ y. x * y
```

Příklad 6: Zápis funkce se dvěma parametry

Aplikace označuje volání funkce s různými argumenty. Například opět funkce f(x) = x – 5. Jako argument funkce se použije například číslo sedm. Když měl lambda kalkul mít pouze matematické využití, očekává se, že po dosazení argumentu do funkce bude zápis vypadat asi následovně: f(7) = x – 5. Ovšem argument funkce se dává až na konec předpisu funkce. Takže zápis vypadá následovně:

```
(λ x . x - 5) 7
```

Příklad 7: Dosazení argument do funkce

2.3 Výhody a nevýhody funkcionálního programování

Než bude srovnáno funkcionální (deklarativní) programování s objektově orientovaným (imperativním) programováním a zváží se, které z těchto dvou druhů programování je výhodnější, tak zatím budou vyjmenovány výhody a nevýhody samotného funkcionálního programování, o kterých se lze všude veřejně dočíst nebo které jsou na první pohled zřejmé.

2.3.1 Výhody

- Asi každého programátora, který se poprvé podívá na nějaký zdrojový kód, který je napsaný ve funkcionálním programování, napadne, že je o hodně kratší a úspornější. Tím pádem je možno vyhnout se spoustě chyb a překlepů, které můžou zabrat spoustu času při jejím hledání. Lépe řečeno orientace ve zdrojovém kódu je mnohem jasnější.

  Příkladem toho může být i následující zdrojový kód metody *vyhledani*. Ta vypíše příjmení prvních pěti nalezených lidí, kteří bydlí v Tachově, z ArrayListu *data*. Tento výpis bude seřazený podle abecedy.

```
public static void vyhledani(){
      data
      .stream()
            .filter(d->d.getMesto().equals("Tachov"))
            .map(Trida::getPrijmeni)
            .limit(5)
            .sorted()
            .forEach(System.out::println);
}
```

Příklad 8: Ukázka deklarativního kódu

  Na tomto příkladě je patrná i jednodušší manipulace s datovou strukturou typu seznam, s kterou lze provědět nemalé operace právě díky datovodům, které jsou probrány ve třetí kapitole.

  Na následujícím příkladě, jde zjistit, že metoda *vyhledani* se zachová stejně jako metoda *vyhledani* z předchozího příkladu.

Tento ovšem funkcionálním způsobem naprogramován není. V tomto případě je potřeba použít cykly, podmínky nebo proměnné hodnoty.

```
public static void vyhledani(){
	int pom = 0;
	ArrayList<String> list = new ArrayList<>();
	for(Trida tri : data){
	    if(pom < 5){
		    if(tri.getMesto().equalsIgnoreCase("Tachov")){
			list.add(tri.getPrijmeni());
			pom++;
		    }
	    }else{
		  break;
	    }
	}
	Collections.sort(list);
	for(String str : list){
	      System.out.println(str);
	}
}
```

Příklad 9: Ukázka imperativního kódu

- S přehledností a kratší délkou funkcionálního kódu souvisí i ladění a testování programů, které je v tomto případě mnohem jednodušší. Například Alvin Alexander, programátor v Javě, Scalu nebo v Lispu, na svém webu uvádí, že čisté funkce závisejí pouze na jejich vstupních parametrech, které produkují jejich výstupy [11]. Samozřejmě je možné udělat chybu při psaní samotné funkce, ale jakmile je známý požadovaný výstup, stačí, sledovat hodnoty jednotlivých funkcí a zjistit, kde se vyskytuje chyba. Testování je tak snadnější, protože není nutné se starat o vedlejší účinky a řada pohledů zůstává skrytá.
- Za další výhodu se dá určitě považovat to, že se zde nemusí tak hojně využívat proměnné. V tomto druhu programování zcela postačí, když se hodnoty předávají pomocí návratových hodnot v metodě.
- Jiří Knesl, vývojář softwaru, na svém webu uvádí, že učit funkcionálnímu programování lidi, kteří jsou zatím neposkvrněni programováním, je mnohem jednodušší, než je vyučovat jiným druhům programování. V tu chvíli odpadává vysvětlování třeba proměnných nebo cyklů, bez kterých se dá obejít, což je

paradoxně jednodušší. Dotyčné osoby byly schopny během pár minut psát jednoduché matematické operace nebo pracovat s řetězci [12].

- Další výhoda se váže k jednomu ze základních charakteristik funkcionálního programování, a sice odložené vyhodnocování.

2.3.2 Nevýhody

- Zdrojový kód napsaný ve funkcionálním paradigmatu nevypadá na první pohled snadně. To může mnoho potencionálních programátorů od tohoto druhu programování odradit. Jedním z takových dalších důvodů, které mohou některé jedince odradit, je přítomnost matematických výrazů, ostatně funkcionální paradigma bylo původně vyvinuto jen kvůli matematice.
- Další nevýhody deklarativního programování jsou spojeny s rekurzí. Ta má sice výhody, ale i řadu nevýhod. Jak už bylo zmíněno, používá se jako náhrada za cykly a funkcionální programování se bez ní jen tak neobejde. Odborně řečeno, rekurze umožňuje nahradit iterativní procházení určitého bloku dat. Proto je rekurze hojně využívaná právě ve funkcionálním programování. Někdy je však výhodnější použití iteraci, jindy rekurzi [13]. Když metoda volá sama sebe nebo jinou metodu, cyklus se bude opakovat do té doby, dokud nedojde k požadovanému cíli. Proto v metodě musí být definovaná podmínka, která ukončí celý chod, jinak by došla paměť na zásobníku a program by mohl vyhodit výjimku. Rekurze bývá také velmi náročná na paměť. Požadavky na paměť se zde zvyšují při každém rekurzivním volání a někdy je také velmi obtížné se v logice rekurze vyznat.
- Uživatelé si často stěžují na neměnnost dat, což je jedna z charakteristik funkcionálního programování. Ta způsobuje to, že po vytvoření nějakého objektu či proměnné hodnoty nelze jejich obsah již změnit. Například na webu Alvina Alexandra se zabývají problémem, jak změnit příjmení dívce, která se právě vdala.

"Nejlepší by bylo, kdyby stávající objekt překopírovali do právě nově vytvořeného objektu, ovšem už s novým příjmením" [11].

```
Rozhrani zmena_prijmeni = (Osoby osoba)->{
      Osoby osoba2 = new Osoby(osoba.getJmeno(), "Nováková",
                               osoba.getMesto(), osoba.getVek(),
                               osoba.getPohlavi(), osoba.getVyska(),
                               osoba.getVaha());
```

Příklad 10: Změna příjmení

Zda se všechny tyto výhody a nevýhody potvrdily a jak moc je to výhodné oproti principům z objektově orientovaného programování, tím se bude zabývat čtvrtá kapitola.

3 Prvky Javy jako prostředek funkcionálního programování

V této kapitole jsou podrobně popsány hlavní rysy funkcionálního programování, jejich implementace, jaké nové možnosti tyto prvky přináší, zda jdou některé příklady naprogramovat i bez těchto typických rysů jako jsou lambda výrazy, funkční rozhraní, implicitní metody a datovody.

3.1 Funkční rozhraní

Než bude popsáno, co to vlastně je funkční rozhraní, bude popsáno jen rozhraní samotné. Jde o jednu z konstrukcí, která obsahuje pouze konstanty a metody bez těla. Jedna třída má povoleno implementovat více různých rozhraní [14]. Pokud nějaká třída poté implementuje nějaké rozhraní, znamená to, že implementuje všechny metody specifikované v rozhraní, tedy pokud třída není abstraktní.

```
public interface Vlastnosti{
	public void obvod(int stranaA);
	public void obsah(int stranaA);
}
```

Příklad 11: Rozhraní Vlastnosti

Na příkladě číslo dvanáct je deklarováno rozhraní *Vlastnosti*, které má v sobě metody pro výpočet obvodu a obsahu čtverce s jedním parametrem. Jak je vidět, v rozhraní opravdu stačí mít pouze hlavičky metod. V dané třídě poté stačí všechny metody z rozhraní implementovat.

```
public class Ctverec implements Vlastnosti{
	@Override
	public void obvod(int stranaA){
		System.out.println("Obvod čtverce je " + 4 * stranaA);
	}
	@Override
	public void obsah(int stranaA){
		System.out.println("Obsah čtverce je " + stranaA * stranaA);
	}
}
```

Příklad 12: Implementace rozhraní

V případě doplnění dalších metod do rozhraní, stačí přidat klíčové slovo *extends* do hlavičky rozhraní. To umožní, aby dané rozhraní doplnilo metody a konstanty jiného

rozhraní o další metody resp. konstanty od jiného rozhraní. Tedy úplně stejně jako když jedna třída dědí vlastnosti jiné třídy.

Proč tedy vzniklo v Javě rozhraní? Jedním z důvodů byla skutečnost, že v Javě neexistuje vícenásobná dědičnost. Třída může mít pouze jednoho předka (to znamená, že může zdědit pouze jednu třídu). Toto omezení má své důvody (vícenásobná dědičnost může způsobovat problémy) [15]. Ovšem rozhraní může třída implementovat více.

Nyní bude rozebráno funkční rozhraní. Jako funkční se označuje takové rozhraní, které obsahuje hlavičku jediné abstraktní metody. Definice metody nemusí obsahovat klíčové slovo *abstract*, bude-li v rozhraní jen jedna nedefaultní metoda. Pokud obsahuje další metody, musí nabízet jejich implicitní implementaci. Po implementující třídě tak požaduje implementaci jediné metody [16]. Funkční rozhraní byla zavedena hlavně kvůli lambda výrazům. Každý lambda výraz může vystupovat v roli libovolného funkčního rozhraní, jehož abstraktní metoda typově odpovídá danému lambda výrazu. Pokud tedy má metoda funkčního rozhraní například dva celočíselné parametry a vrací řetězec, lambda výraz musí také přijímat dva celočíselné parametry a vracet řetězec [14]. To je patrné na následujících dvou příkladech.

```
@FunctionalInterface
public interface Rozhrani {
    abstract String vypocet(int cislo1, int cislo2);
}
```

Příklad 13: Ukázka funkčního rozhraní

```
Rozhrani soucet = (cislo1, cislo2) -> {
    return "Součet čísla "+cislo1+" a "+cislo2+" je "+ cislo1+cislo2;
};
```

Příklad 14: Implementace abstraktní metody

Důvodem, proč ve funkčním rozhraní je možno mít pouze jednu abstraktní metodu, je, že kdyby byly v rozhraní třeba dvě abstraktní metody se stejnou návratovou hodnotou i stejným názvem, ale s různými druhy parametrů, tak volající metoda nebude vědět, kterou metodu z těch dvou abstraktních metod volat.

Při vytváření funkčního rozhraní je velmi důležité nezapomenout na klíčové slovo *@FunctionalInterface*, které dané rozhraní správně definuje. Definice funkčního rozhraní

praví, že obsahuje jedinou abstraktní metodu, takže kdyby bylo takových metod víc, překladač by to nepokládal za funkční rozhraní.

```
 2   Unexpected @FunctionalInterface annotation
 3    NewInterface is not a functional interface
 4     multiple non-overriding abstract methods found in interface NewInterface
 5   ----
 6   (Alt-Enter shows hints)
 7
     @FunctionalInterface
     public interface NewInterface {
         void obvod(int stranaA);
11       void obsah(int stranaA);
12   }
```

Obrázek 4: Definice funkčního rozhraní

Je-li definovaná metoda, která je schopna vypočítat jak obvod, tak obsah obdélníku, do funkčního rozhraní je třeba uvést metodu *vypocet* s návratovou hodnotou *void* a se dvěma parametry, která bude ovšem bez těla. Tato metoda nemusí mít opět klíčové slovo *abstract*, bude li metoda v rozhraní pouze jedna.

```
@FunctionalInterface
public interface Obdelnik {
    abstract void vypocet(int a, int b);
}
```

Příklad 15: Ukázka funkčního rozhraní s jedinou metodou

Poté ve třídě, kde bude dané rozhraní implementováno, lze do proměnných typu *Obdelnik* ukládat "implementace" abstraktní metody *vypocet*. Právě tak je možno zaručit, že se abstraktní metoda z funkčního rozhraní dokáže "implementovat" různými způsoby. Abstraktní metoda by se dala "implementovat" ještě vícekrát a ne jen dvakrát, jak je vidět na příkladu.

```
Obdelnik obvod = (a, b) -> System.out.println(2 * a + 2 * b);
Obdelnik obsah = (a, b) -> System.out.println(a * b);

obvod.vypocet(7, 5);
obsah.vypocet(7, 5);
```

Příklad 16: Implementace metody z funkčního rozhraní

Bude-li potřeba mít ve funkčním rozhraní více metod, musela by mít jedna metoda klíčové slovo *abstract* a ostatním dát klíčové slovo *default.*

```
@FunctionalInterface
public interface Rozhrani{
   abstract void vzorec(int stranaA);
   default void obsah(int stranaA){
       System.out.println("Obsah čtverce je " + stranaA * stranaA);
       }
   }
```

Příklad 17: Ukázka funkčního rozhraní

V dané třídě se potom metoda dá implementovat různými způsoby. V následující ukázce kódu jsou proměnné *obvod* a *obsah* typu *Rozhrani*. Zdá se, že obě proměnné jsou deklarovány různými způsoby, ovšem oba zápisy udělají úplně to samé. Jenomže deklarace proměnné obvod obsahuje lambda výraz, který umožňuje celý zápis zkrátit.

```
public static void main (String[] args){
    Rozhrani obvod = (stranaA) -> System.out.println ("Obvod čtverce"
        + ‟ je:” + 4 * stranaA);
    Rozhrani obsah = new NewInterface(){
        public void vzorec(int stranaA){
            System.out.println("Obsah čtverce je: " + stranaA *
stranaA);
       }
    };
    obvod.vzorec(5);
    obsah.vzorec(5);
}
```

Příklad 18: Implementace metod z funkčního rozhraní

3.2 Implicitní metody

Dalším nově přidaným prvkem do osmé verze Javy jsou implicitní (defaultní) metody. Implicitní metody se mohou zdát zbytečností. Dříve, bez výchozích metod, nebylo možné přidat do rozhraní nějakou metodu, aniž by se musely implementovat v dané třídě.

Implicitní metody řeší častou nutnost použít abstraktní třídu pro definici metod, které by se v implementujících třídách opakovaly.

```
@FunctionalInterface
public interface NewInterface{
    void obecnyVzorec(int stranaA);
    default void obsah(int stranaA){
         System.out.println("Obsah čtverce je: " + stranaA * stranaA);
    }
    default void obvod(int stranaA){
         System.out.println("Obvod čtverce je: " + 4 * stranaA);
    }
}
```

Příklad 19: Implicitní metody

Implicitní metody z rozhraní se vyvolají klasicky vytvořením nového objektu a následným zavolání metody.

```
Ctverec ct = new Ctverec();
ct.obsah(7);
```

Příklad 20: Vyvolání implicitní metody

Vojtěch HorděJčuk na svém webu varuje před jmennými konflikty. Jedná se o to, že jedna třída může implementovat více rozhraní než jen jedno a v těchto různých rozhraních může existovat metoda se stejným názvem. V tomto případě je nutno v implementující třídě správnou metodu překrýt [14]. Například na následující ukázce kódu jsou dvě rozhraní, *Obdelník* a *RovnoramenTroj*. Obě tyto rozhraní obsahují metodu *obvod* se dvěma parametry.

Tudíž ve třídě *Tvary*, která implementuje obě rozhraní, je třeba správnou metodu překrýt. Metoda musí mít samozřejmě stejný název, stejnou návratovou hodnotu i stejný počet parametrů.

```
public class Tvary implements Obdelnik, RovnoramTroj{

      @Override
      public void obvod(int stranaA, int stranaB){
            Obdelnik.super.obvod(stranaA, stranaB);
      }
```

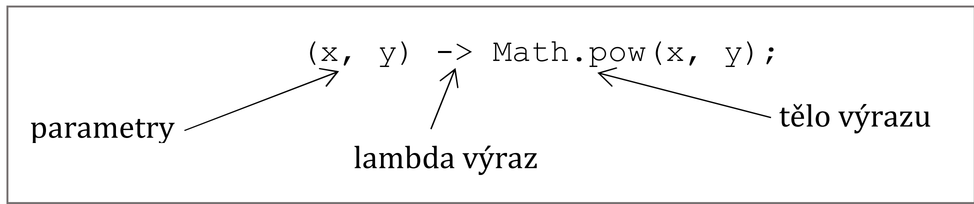
Příklad 21: Definování správné metody

3.3 Lambda výrazy

Jak již bylo zmíněno, lambda výrazy jsou prvky, kterými se funkcionální programování nejvíce vyznačuje. Lambda výrazy se chovají jako instance funkčního rozhraní, jehož metoda má odpovídající parametry a vrací hodnotu odpovídajícího typu [16]. Celá tato část kódu se uloží do proměnné funkčního typu a předává se metodám jako hodnota parametru tohoto typu [17].

3.3.1 Syntaxe

Použití lambda výrazů a jejich syntaxe je vcelku jednoduchá. Celý tento zápis se skládá z parametrů, pokud jsou nějaké k dispozici, lambda výrazu, ten je značen prostou šipkou (->), a těla výrazu, kde je definováno, co se vlastně bude vracet. Tělem může být jednoduchý výraz, nebo celá posloupnost příkazů. Parametry se vždy udávají vlevo a tělo výrazu vpravo.



Obrázek 5: Ukázka syntaxe

Syntaxe celého výrazu dost závisí i na počtu parametrů a příkazů v těle výrazu. Do závorek se totiž parametr dávat nemusí, je-li pouze jeden.

Na následujícím příkladě se ukládá do proměnné *sudy,* která je typu funkčního rozhraní *Test*, metoda, která vyhodotí, zda je zadávané číslo sudé, nebo není.

```
Test sudy = n -> (n%2) == 0;
```

Příklad 22: Zápis lambda výrazu s jedním parametrem

V rozhraní je definice metody typu boolean s jedním parametrem. Takže metoda bude vracet buď *true* nebo *false*. V třídě se dá definovat následujícím způsobem. Dokáže-li překladač odvodit typ parametru, nemusí se uvádět [16].

```
Test sudy = n -> (n%2) == 0;
```

Příklad 23: Zápis lambda výrazu s jedním parametrem

Když je ovšem parametrů více než jeden, musí se dát do kulatých závorek. To je k vidění na příkladě metody, která vrací číslo umocněné na n-tou.

```
Test mocnina = (i,n) -> Math.pow(i,n);
```

Příklad 24: Zápis lambda výrazu s více parametry

Dosud zde byly ukázány pouze příklady, kde se vpravo vyhodnocoval obyčejný výraz, který se nemusel dávat do složených závorek. Ty se používají, až když je třeba provést více příkazů a ne jen jeden vyhodocovací výraz. Příkladem může být opět metoda *sudy*, která je nyní ovšem typu String a bude vracet, zda je udávaný parametr sudé, nebo liché číslo.

```
Test sudy = n -> {
       if((n%2) == 0){
              return "sudý";
       }else{
             return "lichý";
       }
};
```

Příklad 25: Lambda výraz s jedním parametrem a více příkazy

Samozřejmě i když je v těle výrazu několik příkazů, logicky i zde může být více parametrů. Důkazem toho je opět ukázka příkladu metody, která do terminálu vypíše číslice, která budou v rozsahu dle zadávaných parametrů.

```
Test mocnina = (i, j) -> {
	for(int k = i; k <=j; k++){
		System.out.println(k);
	}
};
```

Příklad 26: Lambda výraz s více parametry a více příkazy

I kdyby v kódu existovala metoda, která nebude mít žádný parametr, tak vlevo musí zůstat prázdné závorky, i když v nich nic nebude. O tom se dá přesvědčit na příkladu, který vypíše aktuální datum a čas a nepotřebuje k tomu žádné parametry.

```
Test vypis = () -> System.out.println(LocalDateTime.now());
```

Příklad 27: Lambda výraz bez parametru

Lambda výrazy se hojně využívají hlavně v datovodech. O nich ovšem pojednává až následující podkapitola. Metody je možné referencovat operátorem, který se zapisuje jako dvě dvojtečky. Získané reference lze tak použít místo lambda výrazů [14]. Na následující ukázce je vidět tento zápis v operaci m*ap* a *forEach*. Operace *map* by se chovala úplně stejně i se zápisem *t - > t.getPrijmeni*. Stejně tak i třeba zápis *t -> System.out.println(t)* v operaci forEach.

```
public static void vyhledani(){
	data
		.stream()
		.filter(d -> d.getMesto().equals("Tachov"))
		.map(Osoba::getPrijmeni)
		.limit(5)
		.sorted()
		.forEach(System.out::println);
}
```

Příklad 28: Odkaz na metodu

Zde jsou reference, které jsou v tomto případě podporovány.

```
třída::statická metoda
instance::instanční metoda
konstruktor třídy::new
```

Příklad 29: Podporované reference [14]

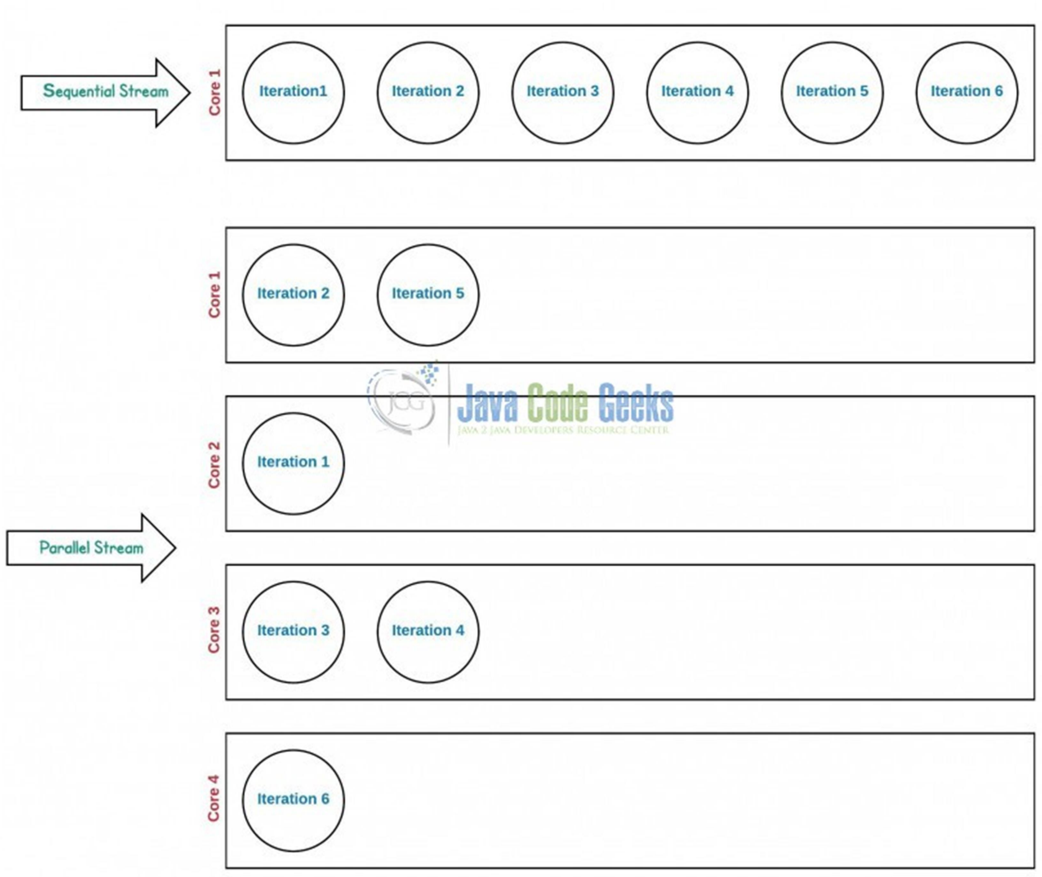
3.4 Datovody

Nově přidanými objekty do osmé verze Javy se staly datovody neboli streamy. Hned na začátku dochází k rozporu s anglickým názvem datovodů, tedy streams. Termín stream by bylo lepší přeložit jako proud. Tento termín se ovšem používá pro zcela jiný druh objektů – objektů sloužících pro realizaci vstupních a výstupních operací. Při současném používání obou typů objektů by pak mohlo docházet k jistému terminologickému zmatení [16]. Datovody zejména umožňují provádět s daty různé operace. Tyto operace jsou velmi podobné příkazům, které se aplikují na data v jazyce SQL.

Datovody bývají označovány jako ekvivalent kolekcí. Od nich se však liší v několika zásadních věcech.

- Datovod přijatá data pouze zpracuje a pošle dál. To znamená, že si je neukládá a tím nezabírá žádnou paměť.
- Datovod nemění vstupní data. Tudíž mohou být zpracovávána několika procesy zároveň.
- Datovod pracuje na základě odložené vyhodnocovací strategie, čímž šetří výpočetní výkon až do poslední chvíle. Začne se provádět, až je ho třeba [18].
- Datovody se nezajímají o to, zda je vstup dat konečný, nebo nekonečný. Prostě převezmou zdrojová data a dopraví je k požadované operaci [16].

Datovod může data zpracovávat dvěma způsoby. Sériově a paralelně. Data v sériovém datovodu jsou zpracovávány postupně, to znamená popořadě jedním vláknem. Zatímco při paralelním zpracování jsou data zpracovány paralelně pomocí více vláken neboli podprocesů. Paralelní zpracování se používá tehdy, nezáleží-li na pořadí, ve kterém budou operace prováděny.



Obrázek 6: Sekvenční vs paralelní proud [3]

Důkazem toho může být příklad, kdy je třeba vypsat čísla od jedničky do pětky. Datovod se zpracovává jak sériově, tak paralelně. Při sériovém zpracování se čísla vypíšou tak, jak je dotyčný uživatel zadal. Ovšem při paralelním zpracování bude výpis prvků přeházený, záleží na tom, které jádro operaci dřív zpracuje.

```
Stream.of("1", "2", "3","4", "5").forEach(System.out::println);

Stream.of("1", "2", "3","4", "5").parallel().forEach(System.out::println);
```

Příklad 30: Paralelní a sériový zpracování

3.4.1 Druhy operací

Operace, které datovody obsahují, se dělí na dva druhy, a sice na průběžné a koncové. Průběžné operace jsou takové operace, jež se po provedení jako výstupní hodnota vrací opět v podobě datovodu. Díky tomu může dojít k takzvanému zřetězení operací, kdy na

[3] *www.examples.javacodegeeks.com* [online]. Dostupný na WWW: https://examples.javacodegeeks.com/core-java/java-8-parallel-streams-example/

výsledný datovod aplikuje některá z dalších operací. Mezi takové průběžné operace se určitě řadí *filter*, *sorted*, *limit*, *map*, *skip* atd.

Koncové operace se aplikují pouze tehdy, pokud se již ukončuje činnost datovodu. Výstupní hodnota v tomto případě může být například kolekce, Integer nebo obyčejné vypsání do konzole. Mezi koncové operace lze zahrnout třeba *count*, *forEach* nebo *collect*. Co tyto operace vůbec znamenají a jak je správně využít si podrobně ukážu později.

Při volání průběžných metod si datovod pouze zapamatuje, ke které metodě (operaci) má dopravit data. Celá sekvence se spustí až v okamžiku zavolání koncové operace. Teprve ta spustí celou posloupnost datovodů, které předávají objekty od jedné operace k druhé, aby byl na konci obdržen požadovaný výstup [16].

3.4.2 Průběžné operace

Nejdůležitější průběžné operace jsou rozebírány na ukázce metody *vyhledani*, která byla představena již na začátku, ale i na jiných praktických příkladech. Metoda *vyhledani* vypíše příjmení prvních pěti nalezených lidí z ArrayListu *data*. Budou vypsáni lidé, kteří bydlí v Tachově, a výpis bude seřazený podle abecedy.

```
public static void vyhledani(){
      data
            .stream()
            .filter(d -> d.getMesto().equals("Tachov"))
            .map(Trida::getPrijmeni)
            .limit(5)
            .sorted()
            .forEach(System.out::println);
}
```

Příklad 31: Ukázka kódu s datovody

3.4.2.1 Filter

Operace filter je určitě jedna z nejdůležitějších operací. Ta má za úkol odfiltrovat data, která mě v tuto chvíli nezajímají. Do příslušného parametru se zadává v podstatě jakýkoli predikát (podmínka).

```
filter(Predicate<? Super T> predicate)
```

Obrázek 7: Syntaxe operace filter

Například na ukázce výše je udaná podmínka, jež hledá takové osoby, které bydlí v Tachově. Namísto hledání osob podle jejich trvalého bydliště mohly být filtrovány podle udávaného věku.

```
.filter(d -> d.getVek() > 40 & d.getVek() < 50)
```

Příklad 32: Hledání dle věku

Jak použít operaci *filter* by se dalo také ukázat na obyčejném vyhledání sudých čísel ze seznamu.

```
.filter(číslo -> číslo % 2 == 0)
```

Příklad 33: Hledání sudých čísel

3.4.2.2 Map

Je-li potřeba přetypovat objekty v datovodu, zavolá se příslušná mapovací metoda, která potřebnou konverzi realizuje a výstupnímu datovodu předá výsledný objekt [16]. Do parametru operace je nutné, aby se zadal typ nového objektu. Poté budou překonvertovány všechny objekty, které operaci projdou.

```
map(Function<? Super T,? extends R> mapper)
```

Obrázek 8: Syntaxe operace map

Mapovací metody mají mnoho variant. Ať už se jedná o obyčejné *map, mapToInt, mapToDouble, mapToLong atd.* Například ve funkci *vyhledani* byla použita operace *map*, která ze všech možných údajů, které jsou o osobách evidovány, vrátila pouze jejich příjmení. Operace map je v metodě vyhledani zapsána zkráceným způsobem, stejně tak by šla zapsat i klasicky s lambda výrazy.

```
map(t -> t.getPrijmeni)
```

Příklad 34: Operace map

Operace *mapToInt*, která byla použita v příkladu, kde se potřebuje zjistit nejvyšší hmotnost z ArrayListu *data*. Atribut hmotnost není typu *int* ale typu *Double*. Na příkladu je také vidět, že celkový výsledek datovodu lze uložit do určité proměnné, se kterou se dá později pracovat.

```
int nejvyssi_hmotnost = data
                        .stream()
                        .mapToInt(n -> Integer.parseInt(n.getHmotnost()))
                        .max()
                        .getAsInt();

System.out.println(nejvyssi_hmotnost);
```

Příklad 35: Ukázka operace mapToInt

Tato operace také umožňuje uložit objekty do předem vytvořené proměnné typu *Stream*. Díky této funkci lze na proměnnou *seznam* aplikovat další rozdílné operace z jiného místa programu.

```
Stream<Osoba> seznam = data .stream()
      .filter(p -> p.getVek() > 50)
      .map(o -> new Osoba (o.getId(), o.getJmeno(), o.getPrijmeni(),
                           o.getVek()));
```

Příklad 36: Uložení datovodu

3.4.2.3 Limit

Tato operace vrátí takový datovod, jež nebude delší než zadávaný parametr typu long, který udává maximální délku datovodu.

```
limit(long maxSize)
```

Obrázek 9: Syntaxe operace limit

V metodě vyhledani je jako parametr nastaveno číslo pět. To znamená, že operace vrátí datovod, kde bude prvních pět nalezených lidí.

3.4.2.4 Sorted

Už z názvu je určitě zcela jasné, že se bude jednat o operaci, která se bude starat o řazení jednotlivých prvků. V tuto chvíli existují dva druhy těchto operací. První z nich je v podstatě velmi jednoduchý.

Vrací totiž datovod sestávajících z prvků řazených podle přirozeného pořadí. Tady se totiž předpokládá, že objekty v datovodu již mají definovaný komparátor [16].

```
sorted()
```

Obrázek 10: Syntaxe operace sorted

Na následujícím příkladu je operace *sorted* použita na streamu, kde se vypíše díky operaci *map* příjmení osob, která budou seřazená podle abecedy. Řadit lze logicky i seznam, který bude naplněn čísly.

I v tomto případě stačí na daný stream použít pouze operaci *sorted* bez parametrů a seznam bude seřazen.

```
data
      .stream()
      .map(Osoba::getPrijmeni)
      .sorted()
      .forEach(System.out::println);
```

Příklad 37: První druh operace sorted

Druhý druh operace *sorted* opět vrátí seřazená data, ovšem v parametru jí musí být poskytnut příslušný komparátor.

```
sorted(Comparator<? Super T> comparator)
```

Obrázek 11: Syntaxe operace sorted s komparátorem

Pomocí operace *sorted* s komparátorem se dá určit, podle čeho se má daný stream seřadit, aniž by se musela použít operace *map* jako v předchozí ukázce. Díky komparátoru lze řadit data dle jakéhokoli atributu. Bude-li atribut typu String, data se seřadí podle abecedy. U atributu typu Integer se data logicky seřadí podle pořadí, jako tomu je na následujícím příkladě, kde se data seřazují podle věku osoby.

```
data .stream()
      .sorted(Comparator.comparing(Osoba::getVek))
      .forEach(d -> System.out.println(d.getId() + " " + d.getJmeno()
                                 + " " + d.getPrijmeni() + " "
                                 + d.getVek()));
```

Příklad 38: Sorted s komparátorem

Bude-li potřeba dosáhnout sestupného pořadí, stačilo by k operaci *sorted* připojit metodu *reversed()*, což platí u atributu typu *String* i *Integer*.

```
.sorted(Comparator.comparing(Osoba::getVek).reversed())
```

Příklad 39: Ukázka obráceného řazení

Samozřejmě může nastat situace, kdy nadefinované komparátory nepomůžou, potom je potřeba vytvořit komparátor vlastní.

3.4.2.5 Ostatní důležité průběžné operace

Existuje dalších mnoho průběžných operací, o kterých by se dalo dopodrobna psát. Jsou zde vyjmenovány jen některé, které se používají nejvíce.

Poměrně často využívanými průběžnými operacemi jsou *max* a *min*. Už z názvu vyplývá, o co se tyto operace starají. Operace *max* byla již vidět na příkladě v podkapitole Map, kde se hledal nejvyšší hmotnost osoby z daného ArrayListu. Stejně tak by se dala použít operaci *min* pro nalezení naopak nejmenší hodnoty.

Užitečná je i operace *distinct*, s jejímž přidáním do řady operací nebude výstup duplicitní.

Velmi často se využívá také operace *findFirst*, která je schopna vyhledat první záznam z datovodu.

Nápomocná může být i operace *peek*. Ta je určena především k ladění. V průběhu operací je schopna ukázat prvky, které právě „protékají“ daným datovodem. Například takový datovod, kde se filtrují prvky podle určeného predikátu. Za operaci *filter* se připojí operace *peek*, která bude v průběhu hlásit, který prvek právě prozkoumává.

```
.peek(p -> System.out.println("Aktuální prvek: " + p))
```

Příklad 40: Operace peek

3.4.3 Koncové operace

3.4.3.1 ForEach

V podstatě se jedná o jednu z nejvíce používaných koncových operací ihned po operaci *Collect.* ForEach provede požadovanou akci pro každý prvek z datovodu. Nejčastěji se používá pro konečný výpis jednotlivých prvků zpracovaných ze streamu.

```
forEach(Consumer<? Super T> action)
```

Obrázek 12: Syntaxe operace forEach

Zápisy této operace byly již výše několikrát vidět.

```
.forEach(System.out::println);
```

Příklad 41: Ukázka operace forEach

```
data.forEach(d -> System.out.println(d.getId() + " "
              + d.getJmeno() + " " + d.getPrijmeni() + " "
              + d.getVek()));
```

Příklad 42: Výpis všech atributů databáze

2.4.3.2 Count

Operace count má za úkol vrátit počet záznamů z právě zpracovaného streamu.

```
count()
```

Obrázek 13: Syntaxe operace count

Na příkladě 44 lze vidět metodu *vyhledani()*, která do proměnné pocet uložila celkový počet osob bydlící v Tachově. Výstup operace *count* se běžně ukládá do proměnné typu long.

```
public static void vyhledani(){
long pocet = data
                   .stream
                   .filter(d -> d.getMesto().equals("Tachov"))
                   .count();

System.out.println(pocet);
}
```

Příklad 43: Ukázka operace count

2.4.3.3 Collect

Bude-li potřeba, aby byla návratová hodnota ze zpracovaného streamu kolekce, použije jako koncová operace metoda collect. Do příslušného parametru se poté zadá, o jakou kolekci by se mělo jednat. Může jít třeba o seznam, nebo množinu.

```
collect(Collector<? super T,A,R> collector)
```

Obrázek 14: Syntaxe operace collect

Například výstup na následujícím příkladě se ukládá do seznamu.

```
List<Osoba> seznam = data
                            .stream()
                            .filter(o -> o.getVek() > 40)
                            .collect(Collectors.toList());
```

Příklad 44: Ukázka operace collect

Stejně tak bylo možné výstup místo seznamu ukládat třeba do množiny, kde by byl zápis místo `collect(Collectors.toList())` logicky `collect(Collectors.toSet())`. Výsledek se mohl ukládat například do kolekce *Map*, kde ovšem vyvstává otázka, podle kterého klíče se bude kolekce mapovat. Zápis této operace je zde trochu jiný. `Collect(Collectors.groupingBy(Třída::kritérium))`.

2.4.3.4 Ostatní důležité koncové operace

Většinou se jako další koncové operace používají například *getAsInt nebo getAsDouble*, které uloží výstup streamu do určité proměnné, která musí být ovšem stejného typu jako výstup dané operace. Jako koncovou operaci lze použít i již nadefinovaný getter. O tom se dá přesvědčit na následující ukázce kódu, kde se do proměnné *pom*, která je typu String, uloží pomocí gettru příjmení hledané osoby.

```
String pom = data
                    .stream()
                    .filter(p -> o.getVek() > 40)
                    .findFirst()
                    .get()
                    .getPrijmeni();
```

Příklad 45: Getter jako výstup

4 Funkcionální vs. objektově orientované programování

V této kapitole jsou popsány rozdíly v implementaci mezi funkcionálním a objektově orientovaným programováním. Zároveň je porovnána, ve kterých případech je implementace jednodušší, lehčí na pochopení nebo méně náročnější na paměť.

4.1 Funkce jako hodnoty proměnných a parametry funkcí

K deklarativnímu programování hodně pomohla funkční rozhraní s lambda výrazy, díky kterým lze funkce z rozhraní implementovat různými způsoby a ukládat je do proměnných typu daného funkčního rozhraní, které poté lze použít jako argumenty funkcí. Důležité je poznamenat, že samotné lambda výrazy nemají tolik možností, pokud se nepoužívají ve spojení právě s funkčními rozhraními. Nebylo nalezeno tolik nepříznivých názorů tak jako příznivých. Například názory Javovských architektů z internetové diskuze na www.quora.com.

"Lambda výrazy umožňují uživatelům předávat "funkce" kódu. Můžeme tedy psát kód snadněji, protože dříve jsme potřebovali několik rozhraní nebo abstraktních tříd. Předpokládejme například, že máme kód, který má nějakou složitou smyčku, podmíněnou logiku nebo složitý pracovní postup. Jeden stejný krok budeme chtít naprogramovat zcela rozdílně. S lambda výrazy, můžeme tak definovat, že jde vlastně o něco jiného" [19].

"Lambda výrazy s funkčními rozhraními činí funkční programování v jazyce Java mnohem stručnějším a rychlejším než v předchozích verzích Javy. Funkční programování je možné ale i v nižších verzích Javy, pomocí anonymních vnitřních tříd, ale to nevypadá tak dobře" [19].

Je možné se ale setkat i s negativním názorem. Například v diskuzi od přispěvatele na webu stackoverflow.com s tématem Lambda výrazy vs Anonymní třídy.

"Lambda výrazy je sice skvělá vlastnost, ale pracuje pouze s typy SAM. To znamená rozhraní pouze s jednou abstraktní metodou. Kompilace se nezdaří, jakmile vaše rozhraní obsahuje více než 1 abstraktní metodu. Zde bývají užitečnější anonymní třídy" [20].

Na ukázce jde vidět, že v anonymní třídě lze deklarovat více abstraktních metod se stejnou návratovou hodnotou i stejným počtem parametrů, což by ve funkčních rozhraních neprošlo.

```
abstract class Metody{
  abstract void vypis(int pom);
  abstract int vypocet1();
  abstract int vypocet2();
  abstract String retezec(String retezec1, String retezec2);
}
public class VelkyPokusny{
    public static void main(String[] args) {

        Metody p=new Metody(){
            void vypis(int pom){System.out.println(pom);}
            int vypocet1(){return 1 + 1;}
            int vypocet2(){return 1 - 1;}
            String retezec(String retezec1, String retezec2){
                              return retezec + " " + retezec2;}
       };
     }
}
```

Příklad 46: Ukázka anonymní třídy

Anonymní třídy zpracovává kompilátor jako nový podtyp pro danou třídu nebo rozhraní, takže pro každou použitou anonymní třídu bude vygenerován nový soubor třídy. Při spuštění aplikace bude načtena a ověřena každá třída, která je vytvořena pro danou třídu. Tento proces je poměrně časově náročný, když existuje velký počet anonymních tříd. Namísto generování přímého bajtkódu pro daný lambda výraz, kompilátor deklaruje předpis a deleguje reálný přístup ke konstrukci. Takže lambda výrazy jsou rychlejší než anonymní třídy, protože jsou prováděny pouze tehdy, když jsou volány [21].

Je pravda, že je škoda, že funkční rozhraní dovolují deklarovat jen jednu abstraktní metodu. Na druhou stranu nikdo nezakazuje mít více funkčních rozhraní a v dané třídě je implementovat.

4.2 Odložené vyhodnocování

Díky lambda výrazům se v Javě zpřístupnil další z rysů deklarativního programování, a sice odložené (lazy) vyhodnocování. Do té doby Java pracovala pouze na způsobu

dychtivého (eager) vyhodnocování. Odložené vyhodnocování je jednoznačnou výhodou funkcionálního programování. Jednoznačně šetří paměť, protože nemusí vyhodnocovat výrazy, které se ani nemusí v programu použít, lze se vyhnout zbytečným výpočtům a také se dá vyhnout chybovým hláškám při vyhodnocování složitých výrazů.

Naproti tomu objektově orientované programování pracuje na způsobu eager vyhodnocování. Jednou konkrétní výhodou eager vyhodnocování je, že je jednodušší zjistit, kde se právě v kódu výpočet nachází nebo který výraz se právě vyhodnocuje. V tomto případě je mnohem jednodušší náš program debuggovat.

V internetové diskuzi na www.quora.com na téma jaké jsou výhody eager vyhodnocování přispěvatel, který pracoval v CodeUnion, napsal:

"Stručně řečeno, eager hodnocení dává programátorovi větší kontrolu (ale také větší zodpovědnost) nad řádem provádění, ovšem klade větší zátěž na kompilátor, oproti lazy hodnocení" [22].

4.3 Manipulace se seznamy

Od okamžiku zavedení datovodů lze lépe manipulovat se seznamy, což je další z charakteristik funkcionálního programování. Ve třetí kapitole bylo uvedeno, které všechny důležité operace se dají s datovody provádět. Tyto operace mimo jiné umožňují vypustit vytváření například cyklů, podmínek, kolektorů nebo zbytečných vytváření proměnných pro získání celkového počtu dat, která splňují zadávaná kritéria. Na příkladě níže lze vidět objektově orientovaný kód. Tento kód by měl vypsat násobky čísla tři až do třiceti. ArrayList *cisla* obsahuje seznam neseřazených neopakujících se čísel do stovky. V objektově orientovaném kódu by se nejprve použila řadící metoda a pomocí forEach by se prošel celý seznam.

Čísla, která projdou podmínkou, se budou vypisovat do terminálu. Daný limit poté ohlídá počet vypsaných čísel.

```
Collections.sort(cisla);
int limit = 0;
for(int cis : cisla){
      if(limit < 10){
            if(cis % 3 == 0){
                  System.out.println(cis);
                  limit++;
            }
      }else{
            break;
      }
}
```

Příklad 47: Objektově orientovaná práce s listem

S použitím datovodů by stačilo data nafiltrovat, seřadit, zadat jim požadovaný limit a vypsat.

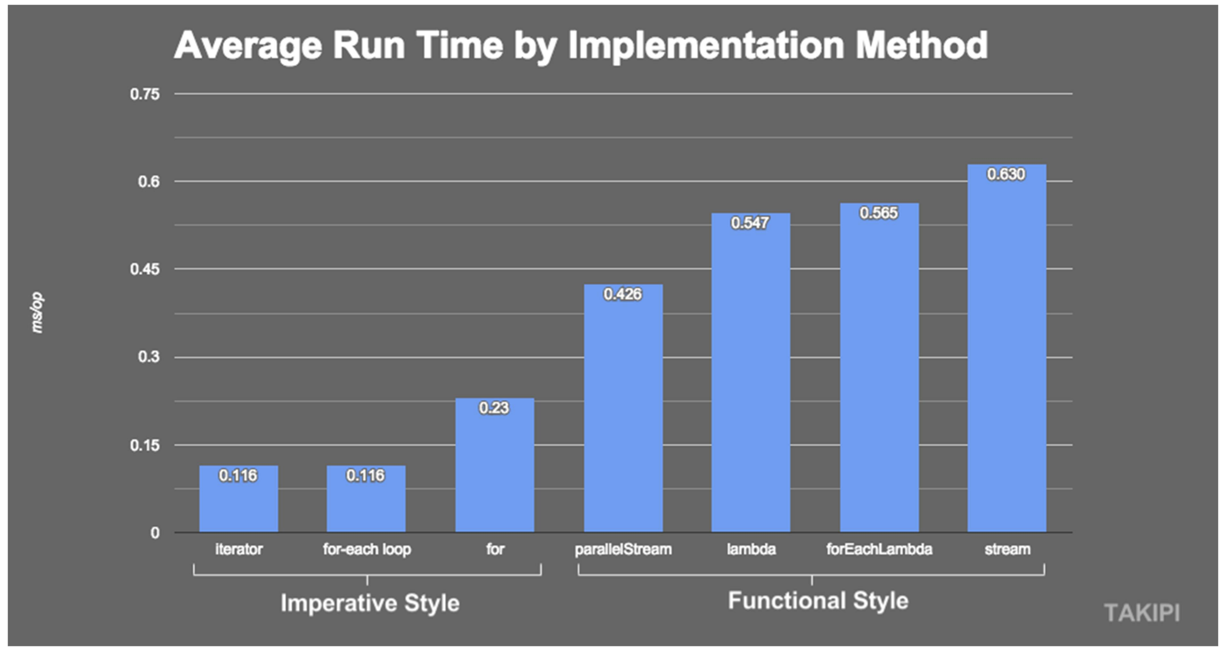
```
cisla.stream()
      .filter(cislo -> (cislo % 3 == 0))
      .sorted()
      .limit(10)
      .forEach(c -> {
            System.out.println(c);
        });
```

Příklad 48: Funkcionální práce s listem

Velkou výhodou datovodů, mimo průběžných nebo koncových operací, je, že pracují na principu výše uvedeného odloženého vyhodnocování, a tak předem neblokují žádnou paměť, jen přijmou data a pošlou je dál další operaci. Data se v datovodech začnou zpracovávat, až když dorazí ke koncové operaci, jako je třeba *forEach()* nebo když se do proměnné ukládá celkový počet filtrovaných dat pomocí operace *count()*. Jinými slovy opět se data začnou vyhodnocovat, až když jsou opravdu potřeba. Výhodou je, že kód pro zpracování datového toku nemusí znát zdroj datového proudu ani jeho konečnou metodu ukončení.

Na webu stackoverflow.com rozebírali jednotlivé výhody a nevýhody při používání datovodů. Výhody datovodů byly uvedeny již výše. Co se nevýhod týče, jeden účastník

diskuze uvedl, že například procházení kódu s datovody pomocí debuggeru je velmi obtížné. Účastník diskuze totiž píše, že procházení přes cyklus je extrémně lehčí, a to i z hlediska využití CPU. Je-li prioritou hrubá rychlost, je použití datovodů horší [23]. Důkazem toho je výsledek testu [24]. Cílem testu bylo najít maximální hodnotu v ArrayListu s rozdílnou implementací a otestovat, jak dlouho jednotlivé úkony trvaly. Daný ArrayList byl naplněn 100 000 náhodnými čísly a program byl implementován sedmi různými způsoby. Implementace byly rozděleny do dvou skupin: Funkční styl s novými jazykovými funkcemi zavedenými v jazyce Java 8 a imperativní styl s klasickými metodami jazyka Java.



Obrázek 15: Výsledky testu [4]

Na grafu lze vidět, že při použití nových metod z Javy 8 byla rychlost vyhodnocení asi 5x delší. Někdy je jednodušší a lepší použít smyčku s iterátorem, i když to znamená napsat o pár řádků více [21].

Výsledky testu sice nedopadly nejlíp, ale dle mého názoru datovody do našeho kódu přinášejí přehlednost a díky operacím je možno se seznamy lépe pracovat.

4.4 Rekurze vs iterace

Dalším charakteristickým rysem deklarativního programování je rekurze. Mnoho uživatelů nemá rekurzi rádo. Zejména asi proto, že je někdy velmi těžké se v logice

[4] www.*blog.overops.com* [online]. Dostupný na WWW: https://blog.overops.com/benchmark-how-java-8-lambdas-and-streams-can-make-your-code-5-times-slower/

rekurze vůbec vyznat. Uživatelé si stěžují, že se jim často zobrazuje výjimka StackOverflowException. Například účastníci diskuze na stackoverflow.com.

"Já osobně dávám přednost použití iterativnímu přidávání před rekurzivní funkcí. Zvláště pokud mám složitou nebo těžkou logiku a počet iterací je velký. To proto, že se při každém rekurzivním volání zvyšuje zásobník, k jehož přetečení by mohlo dojít" [25].

"Rekurze přidává jasnost a snižuje čas potřebný k zápisu a ladění kód. Snižuje časovou náročnost. Provádí se lépe při řešení problémů založených na stromových strukturách. Například problém Hanojské věže je snadněji vyřešen pomocí rekurze na rozdíl od iterace. Ovšem rekurze je pomalejší kvůli udržování režie zásobníku" [25].

Rekurze bývá oproti cyklům často náročnější na paměť. Požadavky na paměť se zde zvyšují při každém rekurzivním volání.

Co se týče mého názoru, i já dávám přednost iterativnímu procházení a cyklům. Zvláště právě kvůli častému přetečení zásobníku a hlavně pro to, že je to pro mě jednodušší naprogramovat.

4.5 Neměnná data

Neměnná data je jedna z dalších charakteristik funkcionálního programování. Uživatelé si často stěžují, že si ve svém kódu s lambda výrazy vůbec nemůžou změnit vlastnosti nějakého objektu, který mají vytvořený mimo funkční metodu.

Ovšem někteří s určitou programátorskou praxí tuto novinku hájí. Například programátor ve Schemu, Scalu a Haskellu.

Představte si funkci, která kromě svého vstupu používá hodnotu, která je v rozsahu, ale je definována mimo tuto funkci (což znamená, že může být viditelná pro jiné funkce, které s ní také mohou pracovat a mohou být na její hodnotě závislí). Pokud je tato hodnota neměnná, není to problém. Pokud je hodnota proměnná, mohou je kdykoliv změnit jiné funkce. Což by změnilo výstup funkce [26].

Budu-li potřeba například při procházení datovodem vyhledat určité objekty a změnit jejich vlastnosti, kompilátor mi tuto změnu nedovolí. Tento problém by se dal vyřešit tak, že by se upravená data uložila do zvlášť vytvořené kolekce nebo by se datovody vůbec používat nemusely a data by se procházela klasicky přes forEach cyklus.

4.6 Hledání chyb

Hledání chyb v deklarativně napsaném programu bývá často velmi jednodušší, už kvůli tomu, že je daný kód kratší než ten objektově orientovaný, ale hlavně proto, že se výsledky funkcí předávají jako argumenty jiných funkcí. V tomto případě stačí, když se znají výstupy jednotlivých funkcí a poté se procházením jen zjistí, ve které funkci je chyba. Jak ve funkčním programu, tak v objektově orientovaném si lze pomoct postupným vypisováním průběžných výpisů do terminálu. Nebo v datovodech umožňuje operace *peek* průběžně vypisovat data, která právě „protékají" datovodem. Zejména v objektově orientovaném programu může být velmi nápomocen již výše zmíněný debugger, který se dá ve funkčním programu velmi těžko využít.

4.7 Shrnutí

Kdybych měl prostřednictvím názorů jednotlivých uživatelů srovnat tyto dvě paradigmata a jednoznačně říci, které z nich je lepší, asi bych se přikláněl k funkčnímu programování. I když jsou tu někdy jisté věci, kde má objektově orientované programování asi navrch.

Objektově orientované programování		Funkcionální programování	
Proměnné hodnoty, abstraktní třídy	−	Výsledky funkcí, abstraktní třídy a funkční rozhraní	+
Eager vyhodnocování	−	Lazy vyhodnocování	+
For-each cyklus	−	Datovody	+
Cykly, rekurze	+	Cykly by měly být nahrazené rekurzí	−
Data lze po vytvoření změnit	+	Neměnná data	−
Delší délka kódu	−	Kratší délka kódu	+

Tabulka 1: Objektově orientované vs funkcionální programování

Co se týče předávání hodnot v programu pouze pomocí argumentů funkce, namísto vytváření proměnných hodnot, zde má podle mě navrch funkcionální programování. Odpadává tady nutnost vytvářet poměrně zbytečné proměnné hodnoty, které zbytečně prodlužují délku kódu. Hodnoty jsou zde pouze výsledky funkcí, které lze ukládat do proměnných typu daného funkčního rozhraní a ty poté předávat pomocí argumentů funkce. Jak jsem říkal, je veliká škoda, že funkční rozhraní dovoluje deklarovat jenom

jedinou abstraktní metodu. V porovnání s abstraktními třídami, kde lze deklarovat metod, kolik chceme, může docházet k dojmu, že tady funkcionální programování ztrácí. Tuto nevýhodu se snaží vyrovnat implicitní metody a možnost implementovat více funkčních rozhraní zároveň. Ovšem nikdo nezakazuje i zde v tomto programovacím paradigmatu využívat abstraktní třídy. Při implementaci funkčních metod se používají lambda výrazy, které mimo jiné zaručují líné vyhodnocování, což je nepochybně další bod pro funkční programování.

Nepochybně největší výhodu vidím v datovodech. I když výsledky testů pro ně nedopadly nejlépe, díky jejich operacím lze data libovolně zpracovávat, aniž by se musely vytvářet zbytečné programové konstrukce.

Co se týče rekurze, ta má sice menší časovou náročnost, ale je třeba si dávat pozor na přetečení zásobníku, je náročnější na paměť a pro začátečníky těžší na pochopení více než cykly. Někdy je těžké se v logice rekurze vyznat. A co se mě týče, já bych se přikláněl radši k cyklům.

Když se zaměřím na neměnnost dat, najdu spíše více negativních názorů než těch kladných. Co se týče kladných názorů, ty mají sice silný argument, ovšem pokud bude naše funkce korektně naprogramovaná, nic takového by se stát nemělo. Takže dle mého názoru jde spíš o nevýhodu. O výhodu by šlo pouze v případě, kdybych neuměl funkci správně ošetřit.

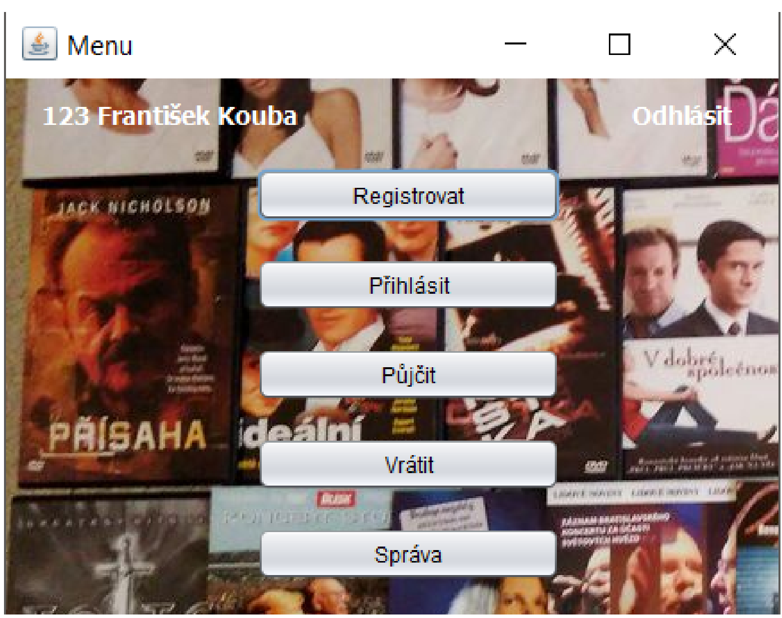
Je pravda, že funkcionální kód je kratší než ten objektově orientovaný a je pravda, že se v něm dá lépe orientovat a hledat chyby sledováním předávaných hodnot z jedné funkce do druhé. Ovšem spuštěný debugger v objektově orientovaném kódu, veškeré procházení kódem a zobrazování aktuálních hodnot určitých objektů udělá za mě. Tedy hledání chyb bych viděl asi tak nastejno.

5 Aplikace

Praktická část této bakalářské práce obsahuje několik javovských aplikací, které jsou přiloženy na příslušné CD. Aplikace jsou záměrně vytvořeny funkcionálním a objektově orientovaným způsobem, aby byly vidět rozdíly mezi těmito programovacími způsoby a hlavně všechny charakteristické rysy funkcionálního programování při použití v praxi. Nutno podotknout, že tyto aplikace obsahují zejména lambda výrazy, datovody a funkční rozhraní, které byly do Javy přidány až v osmé verzi. To znamená, že k rozběhnutí těchto aplikací je nutno mít nainstalovanou alespoň osmou verzi Javy. Zejména k rozběhnutí aplikace videopůjčovna, která pracuje s datovody, je třeba mít naimportovány podpůrné třídy (commons-lang-2.6, commons-logging-1.1.1, hsqldb, jackcess-2.1.6,rs2xml, ucanaccess-4.0.1) a ODBC driver typu Microsoft Access Driver (*.mdb, *.accdb). Doporučený program k rozběhnutí a prohlédnutí všech těchto aplikací je NetBeans 8.2.

5.1 Videopůjčovna

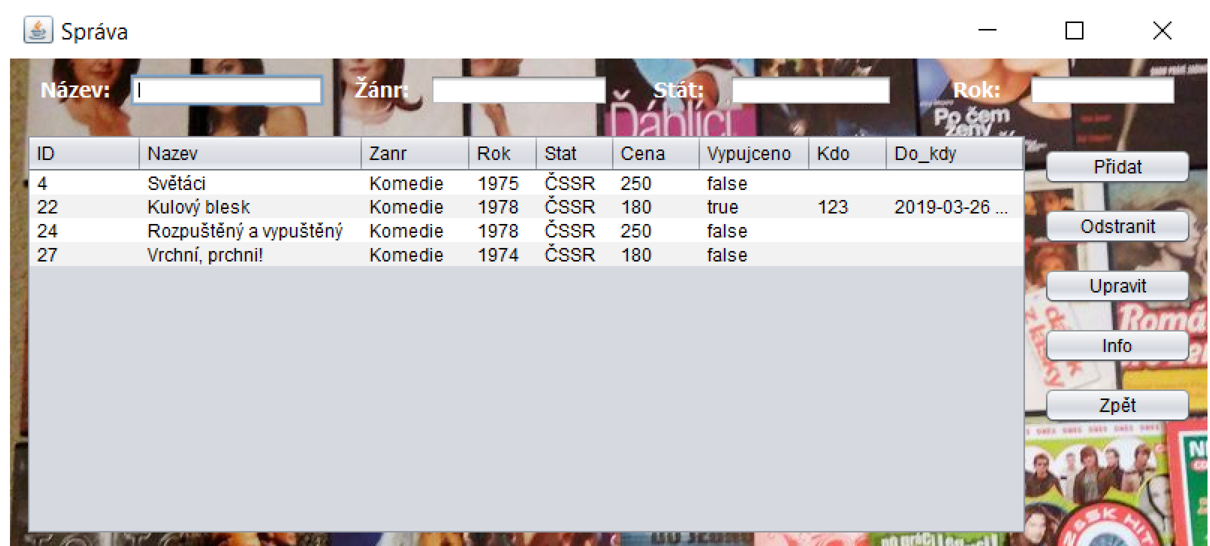
První aplikací je klasická videopůjčovna. Na této aplikaci je ukázána zejména práce s datovody. Jde o klasickou databázi se zákazníky a filmy, kde si zákazníci mohou filmy, půjčovat, vracet a vyhledávat. Zaměstnanec videopůjčovny zase může filmy vyřazovat, přidávat nebo zobrazovat informace o filmech a zákaznících. Odstraňovat zákazníky z databáze videopůjčovny můžou zaměstnanci jen tehdy, nemá-li zákazník nic vypůjčeno. Data do aplikace se nahrávají z přiložené databáze MS Access.



Obrázek 16: Menu videopůjčovny

Tato aplikace je naprogramována jak objektově orientovaným, tak funkcionálním způsobem. Jelikož se data nahrávají z databáze, tak se přímo nabízí použít v Javě SQL dotazy, jako například příkazy INSERT, UPDATE, DELETE nebo SELECT.

Zejména poslední příkaz SELECT se v této aplikaci používá při vyhledávání různých filmů. Tabulka s databází filmů se aktualizuje při každém stisknutí klávesy díky metodě *KeyReleased()*, která reaguje na každé stisknutí tlačítka.



Obrázek 17: Správa videopůjčovny

Filmy jsou poté vyhledávány díky vyhledávacímu dotazu s operátorem LIKE, kde se hodnoty porovnávají s maskou řetězce, který je z obou stran opatřený procenty. Znak procenta představuje nula, jeden nebo více znaků. Například na ukázce níže jsou vyhledávány takové záznamy, které budou odpovídat všem zadávaným kritériím. Výsledky z dotazu jsou poté předány do tabulky.

```
      String naz =  "%" + nazev + "%";
      String zan =  "%" + zanr  + "%";
      String ro  =  "%" + roky  + "%";
      String sta =  "%" + stat  + "%";
String query =
"Select ID, Nazev, Zanr, Rok, Stat, Cena, Vypujceno, Kdo, Do_kdy from
Tabulka1 WHERE Rok LIKE '"+ro+"' AND Stat LIKE '"+sta+"' AND Nazev LIKE
'"+naz+"' AND Zanr LIKE '"+zan+"'";
ResultSet rs = st.executeQuery(query);
table.setModel(DbUtils.resultSetToTableModel(rs));
```

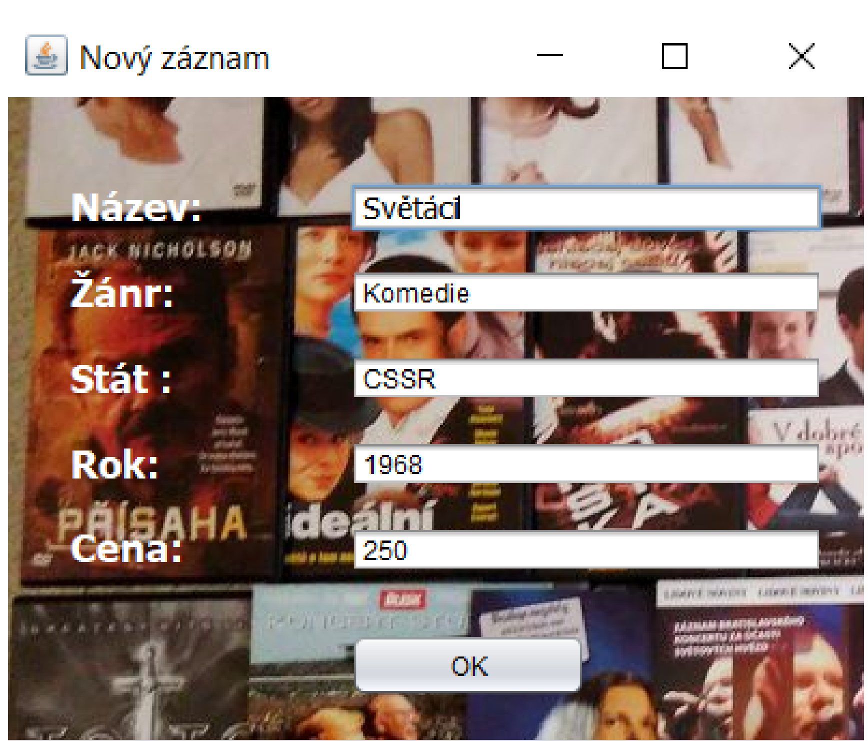
Příklad 49: Vyhledávací dotaz

Naproti tomu při použití datovodů, musí být data nejdříve v nějakém seznamu, aby mohla být převedena do streamu. Pomocí operace *filter* je možno vyhledat takové záznamy, které budou odpovídat požadavkům. Aby byla v programu s datovody docílena stejná funkčnost jako s SQL dotazy, bylo by nutno využít regulérních výrazů. Poté postačí stream konvertovat zpátky na list a zavolat příslušnou metodu, rá se postará o aktualizaci tabulky.

```
List <Film> data = seznam.stream()
                    .filter(r -> Pattern.compile(zanr.toUpperCase())
                                        .matcher(r.getZanr()
                                        .toUpperCase()).find())
                    .collect(Collectors.toList());
vypis_po_hledani(data, table);
```

Příklad 50: Ukázka datovodu

Velkou část práce datovody ulehčí, když bude potřeba v databázi například nějaký záznam aktualizovat. Když bude potřeba nějaký záznam upravit, je nutné, aby se nejdříve údaje požadovaného záznamu nahrály do patřičného formuláře.



Obrázek 18: Formulář k upravení a vložení záznamu

Pokud by bylo třeba použít SQL dotazy, nikoli datovody, musí se nejdříve vytvořit připojení k této databázi, poté pomocí SQL dotazu vyhledat určitý záznam a nahrát ho do formuláře.

```
Connection con = null;
Statement st;
String url="jdbc:ucanaccess://Pujcovna.accdb";
try {
     con=DriverManager.getConnection(url);
     PreparedStatement dotaz = con.prepareStatement("SELECT * FROM Tabulka2
                                                   WHERE ID = '"+
                                                   cislo_zakaznika +"'");
     ResultSet vysledky = dotaz.executeQuery();
     st = con.createStatement();
     while (vysledky.next()) {
        id = vysledky.getInt("ID");
        jmeno_osoby.setText(vysledky.getString("Jmeno"));
        prijmeni_osoby.setText(vysledky.getString("Prijmeni"));
        mesto_osoby.setText(vysledky.getString("Mesto"));
        ulice_osoby.setText(vysledky.getString("Ulice"));
        telefon_osoby.setText(String.valueOf(vysledky.getInt("Telefon")));
     }
} catch (SQLException ex) {
System.err.println(ex);
}
```

Příklad 51: Kód s SQL dotazem

Ovšem s použitím datovodů stačí použít místo SQL dotazu operaci *filter*, kde se „odfiltrují" záznamy, které nejsou potřeba, a nakonec se pomocí operace *forEach* údaje nahrají do formuláře.

```
osoby.stream().filter(o -> o.getId() == cislo_zakaznika)
              .forEach(o -> {
                  jmeno_osoby.setText(o.getJmeno());
                  prijmeni_osoby.setText(o.getPrijmeni());
                  mesto_osoby.setText(o.getMesto());
                  ulice_osoby.setText(o.getUlice());
                  telefon_osoby.setText(String.valueOf(o.getTelefon()));
               });
```

Příklad 52: Kód s použitím datovodů

5.2 Převodník soustav

Aplikace převodník soustav je zaměřená hlavně na představení funkčních rozhraní a implicitních metod. Tato aplikace umožňuje převádět čísla z desítkové soustavy do soustav od dvojkové po devítkovou a naopak nebo umožňují provádět se dvěma čísly, které jsou ve stejné soustavě logický součin a součet.

Obě metody, které se starají o převod jak z desítkové soustavy, tak do desítkové soustavy, mohou být implementovány různými způsoby a uloženy do proměnných typu daného rozhraní. Ve funkčním rozhraní je poté vytvořena abstraktní metoda se dvěma celočíselnými parametry a návratovou hodnotou typu String. Na ukázce kódu je vidět, že při implementaci metody i daný lambda výraz musí mít dva celočíselné parametry a i návratová hodnota musí být typu *String*. Kvůli nutnosti použití metody *reverse* a kvůli neměnnosti hodnoty typu *String* je nutné, aby byl výsledek převodu ukládán do proměnné typu *StringBuilder.*

```
Rozhrani do_desitkove = (prevadene_cislo, cis_soustava) -> {
   double vysledek = 0;
   int mocnina = 0;
   return String.valueOf((int) pr.prevod(vysledek, prevadene_cislo,
                                          cis_soustava, mocnina));
};

Rozhrani z_desitkove = (prevadene_cislo, cis_soustava) -> {
      StringBuilder sb = new StringBuilder();
      while(prevadene_cislo != 0){
            sb.append(prevadene_cislo%cis_soustava);
            prevadene_cislo = prevadene_cislo/cis_soustava;
      }
      return sb.reverse().toString();
};
```

Příklad 53: Implementace abstraktní metody dvěma různými způsoby

Ve funkčním rozhraní je možno vytvořit implicitní metody, které budou vypočítávat logický součin a logický součet dvou zadávaných čísel, které budou ve stejné soustavě. Tyto metody se dají implementovat rovnou na místě.

Implicitní metody přijímají jako argumenty parametry typu String. Jako argumenty mohou být použít naše implementované abstraktní metody.

```
@FunctionalInterface
abstract interface Rozhrani {
    String vypocet (int prevadene_cislo, int cis_soustava);
    default int logicky_soucet (int cislo1, int cislo2){
      return((cislo1) | (cislo2));
    }
    default int logicky_soucin (int cislo1, int cislo2){
      return((cislo1) & (cislo2));
    }
}
```

Příklad 54: Ukázka funkčního rozhraní

Bude-li třeba zobrazit výsledek převodu například z desítkové soustavy do dvojkové, jednoduše pomocí proměnné *z_desitkove* se zavolá abstraktní metoda *vypocet* s požadovanými parametry. Bude-li potřeba převést číslo z desítkové soustavy do dvojkové a zároveň poté vypočítat jejich logický součin, tak opět pomocí proměnné *z_desitkove* se zavolá abstraktní metoda vypocet, jejíž parametry budou implicitní metoda *logicky_soucin* s parametry a číselná hodnota, která udává číselnou soustavu, do které číslo bude převedeno.

```
System.out.println(z_desitkove.vypocet(56, 2));
System.out.println(z_desitkove.vypocet(z_desitkove.logicky_soucin(56, 18),
```

Příklad 55: Vyvolání metod z funkčního rozhraní

Tato aplikace je naprogramována i bez použití funkčních rozhraní a implicitních metod. V tomto případě hlavičky metod *z_desitkove*, *do_desitkove, logicky_soucin* a *logicky_soucet* se musí tedy deklarovat v rozhraní a všechny metody poté implementovat až v dané třídě.

5.3 Operace s datovody

Na poslední aplikaci je ukázáno použití datovodů, funkčních rozhraní, lambda výrazů a implicitních metod dohromady. V této aplikaci je definovaný ArrayList a v něm jsou uložené objekty typu *Osoba.* Tato Osoba má vlastnosti jako je jméno, příjmení, bydliště nebo pohlaví. Ve funkčním rozhraní je opět definovaná jediná abstraktní metoda typu Stream s parametry typu ArrayList a String. V dané třídě lze do proměnné typu daného

rozhraní uložit celý stream i s danými operacemi. Jelikož je výstupem *stream*, tak je možné s ním nadále pracovat, to znamená dále připojovat další operace.

```
NewInterface filtrovani_mesto = (list, kriterium) ->{
             return list.stream()
            .filter(l -> l.getMesto().equalsIgnoreCase(kriterium))
            .map(o -> new Osoby(o.getId(), o.getJmeno(), o.getPrijmeni(),
                                o.getMesto(), o.getVek(), o.getPohlavi(),
                                o.getVyska(), o.getVaha()));
```

Příklad 56: Proměnná filtrovani_mesto

K této dané metodě poté se může připojit například implicitní metoda *razeni_prijmeni*, která pomocí parametru umožňuje navázat na metodu z proměnné filtrovani_mesto. Metoda daný stream seřadí podle příjmení a vypíše tolik osob, kolik udává daný limit.

```
default void razeni_prijmeni (Stream<Osoby> osoby, int limit){

osoby.sorted(Comparator.comparing(Osoby::getPrijmeni))
                          .limit(limit)
                          .forEach(o -> {
                              toString(o);
                            });
}
```

Příklad 57: Implicitní metoda razeni_prijmeni

Bez použití všech výše uvedených prvků by bylo nutné každou metodu zvlášť implementovat v dané třídě a bez datovodů by se musel daný seznam procházet přes forEach cyklus a s pomocí podmínky daná data vyhledat a ukládat je zvlášť do nově vytvořeného listu.

```
@Override
public ArrayList<Osoby> filtrovani_mesto(ArrayList<Osoby> osoby,
                                          String kriterium){
    for(Osoby o : seznam){
       if(o.getMesto().equalsIgnoreCase(kriterium)){
           data.add(new Osoby(o.getId(), o.getJmeno(), o.getPrijmeni(),
                              o.getMesto(), o.getVek(), o.getPohlavi(),
                              o.getVyska(), o.getVaha()));
       }
    }
        return data;
}
```

Příklad 58: Metoda filtrovani_mesto

Pro seřazení dat v metodě *razeni_prijmeni* by se musel využít vytvořený komparátor, který seřazení dat zařídí. Data by se procházela opět pomocí forEach cyklu a pomocí iterativního přidávání by se vypsalo tolik dat, kolik opět udává limit.

```
@Override
public void razeni_prijmeni(ArrayList<Osoby> data, int limit) {
        int i = 0;
        Collections.sort(data, comparePrijmeni);
        for(Osoby o : data){
            if(i<limit){
                toString(o);
              i++;
            }
        }
}
```

Příklad 59: Metoda razeni_prijmeni

6 Závěr

Bakalářská práce se zabývala možnostmi použití funkcionálního programování v Javě. V teoretické práci bylo popsáno, co to je funkcionální programování, jeho základní charakteristiky, výhody a nevýhody. Byly představeny relativně nově jazykové rysy, které umožňují funkcionálně programovat, popsány způsoby jejich implementace nebo ukázány možnosti jejich využití. Ke konci teoretické části proběhlo srovnání mezi objektově orientovaným a funkcionálním paradigmatem. Poté na praktických řešených příkladech byly ukázány možnosti využití výše popisovaných prvků ve spojení s funkcionálním programováním i možnosti implementace těchto příkladu bez probíraných prvků.

Cílem této bakalářské práce bylo ukázat možnosti funkcionálního programování s poměrně novými jazykovými rysy a porovnat ho s objektově orientovaným programování bez těchto rysů. Prostřednictvím mnoha protichůdných názorů se podařilo zjistit, které prvky a charakteristiky uživatelé opravdu vnímají jako výhody a zda jim opravdu ušetřují čas a práci.

Tato bakalářská práce mi dala možnost poznat a vyzkoušet si práci s datovody, funkčními rozhraními a implicitními metodami. Poznal jsem dopodrobna další druh paradigmatu, který jsem chtěl už dlouho poznat.

Všechny probírané prvky urychlily příchod funkcionálního paradigmatu do Javy. Dle mého názoru Java opět postoupila o krok dopředu a s pomocí datovodů, lambda výrazů i funkčních rozhraní si konečně funkcionální programování najde více podporovatelů.

Seznam použité literatury a zdrojů

[1] ŠKVARDA, Libor a Martin ŽÁK. Co je to funkcionální programování [online]. 2006 Dostupné z: http://programujte.com/clanek/2006032503-co-je-to-funkcionalni-programovani

[2] Funkcionální programování. *Jak se naučit programovat* [online]. Dostupné z: http://jaksenaucitprogramovat.py.cz/cztutfctnl.html

[3] Deklarativní programování. https://cs.wikipedia.org [online]. Dostupné z: https://cs.wikipedia.org/wiki/Deklarativn%C3%AD_programov%C3%A1n%C3%AD

[4] https://www.mitrais.com/news-updates/object-oriented-programming-vs-functional-programming-best-of-both-worlds/

[5] Recursion, List Manipulation, and Lazy Evaluation. *Modernescpp* [online]. 2. 2. 2017. Dostupné z: http://www.modernescpp.com/index.php/recursion-list-manipulation-and-lazy-evaluation

[6] Chris Smith. *Programming F#*. [s.l.]: O'Reilly Media, Inc., 2009-10-22. Dostupné online. ISBN 978-0-596-15364-9.

[7] MCCONNELL, Steve. Techniky ladění kódu. *Dokonalý kód: umění programování a techniky tvorby software* [online]. Brno: Computer Press, 2005. ISBN 80-251-0849-x

[8] Programovací paradigmata [online]. Dostupné z: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=342

[9] Funkcionální programování: kapitola 2. Lambda kalkul [online]. Dostupné z: http://www.cs.vsb.cz/navrat/vyuka/flp/texty/fp/ch02.html

[10] ZEMEK, Petr. Λ-kalkul rychle a pochopitelně [online]. 2008. Dostupné z: https://publications.petrzemek.net/articles/PZ_-_IPP-Lambda-Kalkul.pdf

[11] ALEXANDER, Alvin. The Definition of “Pure Function”. *Https://alvinalexander.com* [online]. Dostupné z: https://alvinalexander.com/scala/fp-book/definition-of-pure-function

[12] KNESL, Jiří. Proč funkcionálně?. *Http://www.knesl.com* [online]. Dostupné z: http://www.knesl.com/proc-funkcionalne

[13] Princip rekurze [online]. Dostupné z: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=27486;lang=cz;design=8

[14] HORDĚJČUK, Vojtěch. *Změny v JDK 8* [online]. Dostupné z: http://voho.eu/wiki/java-zmeny-jdk8/

[15] Java interface tajemství zbavený. Dadajax's weblog [online]. 2009. Dostupné z: https://www.dadajax.net/java-interface-tajemstvi-zbaveny/

[16] PECINOVSKÝ, Rudolf. Java 8: úvod do objektové architektury pro mírně pokročilé. Praha: Grada Publishing, 2014. ISBN 978-80-247-4638-8.

[17] Datovod [online]. 2008 Dostupné z: https://cs.wikipedia.org/wiki/Datovod

[18] Part 2 - Lambda vs anonymous class in Java 8. *Only fullstack developer* [online]. 2019. Dostupné z: https://onlyfullstack.blogspot.com/2019/02/lambda-vs-anonymous-class-in-java-8.html

[19] What is your review of Functional Programming In Java: Harnessing The Power Of Java 8 Lambda Expressions?. *Https://www.quora.com* [online]. Dostupné z: https://www.quora.com/Why-are-Java-8-lambda-expressions-regarded-as-a-big-change-in-the-Java-programming-language

[20] Java8 Lambdas vs Anonymous classes. *Https://stackoverflow.com* [online]. Dostupné z: https://stackoverflow.com/questions/22637900/java8-lambdas-vs-anonymous-classes/31982211#31982211

[21] ZHITNITSKY, Alex. How Misusing Streams Can Make Your Code 5 Times Slower. Https://blog.overops.com [online]. 2015 Dostupné z: https://blog.overops.com/benchmark-how-java-8-lambdas-and-streams-can-make-your-code-5-times-slower/

[22] What are the advantages of eager evaluation?. *Https://www.quora.com* [online]. Dostupné z: https://www.quora.com/What-are-the-advantages-of-eager-evaluation

[23] In Java, what are the advantages of streams over loops?. *Https://stackoverflow.com* [online]. Dostupné z: https://stackoverflow.com/questions/44180101/in-java-what-are-the-advantages-of-streams-over-loops

[24] Benchmark: How Misusing Streams Can Make Your Code 5 Times Slower. Https://blog.overops.com [online]. Dostupné z: https://blog.overops.com/benchmark-how-java-8-lambdas-and-streams-can-make-your-code-5-times-slower/

[25] What are the advantages and disadvantages of recursion?. *Https://stackoverflow.com* [online]. Dostupné z: https://stackoverflow.com/questions/5250733/what-are-the-advantages-and-disadvantages-of-recursion

[26] Why is immutability important in functional programming?. *Https://www.quora.com* [online]. Dostupné z: https://www.quora.com/Why-is-immutability-important-in-functional-programming

Seznam obrázků

Charakteristiky funkcionálního programování 14
Hlídání neměnných dat 15
Ukázka syntaxe 17
Definice funkčního rozhraní 24
Ukázka syntaxe 27
Sekvenční vs paralelní proud 31
Syntaxe operace filter 33
Syntaxe operace map 33
Syntaxe operace limit 34
Syntaxe operace sorted 35
Syntaxe operace sorted s komparátorem 35
Syntaxe operace forEach 36
Syntaxe operace count 37
Syntaxe operace collect 37
Výsledky testu 43
Menu videopůjčovny 47
Správa videopůjčovny 48
Formulář k upravení a vložení záznamu 49

Seznam tabulek

Objektově orientované vs funkcionální programování 45

Seznam příkladů

Ukázka funkcionálního kódu ... 13
Ukázka imperativního kódu ... 13
Ukázka funkce vyššího řádu ... 15
Přiřazení hodnoty do proměnné ... 16
Zápis funkce s jedním parametrem ... 17
Zápis funkce se dvěma parametry ... 17
Dosazení argument do funkce ... 17
Ukázka deklarativního kódu ... 18
Ukázka imperativního kódu ... 19
Změna příjmení ... 21
Rozhraní Vlastnosti ... 22
Implementace rozhraní ... 22
Ukázka funkčního rozhraní ... 23
Implementace abstraktní metody ... 23
Ukázka funkčního rozhraní s jedinou metodou ... 24
Implementace metody z funkčního rozhraní ... 24
Ukázka funkčního rozhraní ... 25
Implementace metod z funkčního rozhraní ... 25
Implicitní metody ... 26
Vyvolání implicitní metody ... 26
Definování správné metody ... 27
Zápis lambda výrazu s jedním parametrem ... 28
Zápis lambda výrazu s jedním parametrem ... 28
Zápis lambda výrazu s více parametry ... 28
Lambda výraz s jedním parametrem a více příkazy ... 28
Lambda výraz s více parametry a více příkazy ... 29
Lambda výraz bez parametru ... 29
Odkaz na metodu ... 29
Podporované reference ... 30
Paralelní a sériový zpracování ... 31
Ukázka kódu s datovody ... 32
Hledání dle věku ... 33
Hledání sudých čísel ... 33
Operace map ... 33
Ukázka operace mapToInt ... 34
Uložení datovodu ... 34
První druh operace sorted ... 35
Sorted s komparátorem ... 35
Ukázka obráceného řazení ... 36
Operace peek ... 36
Ukázka operace forEach ... 37
Výpis všech atributů databáze ... 37
Ukázka operace count ... 37
Ukázka operace collect ... 38
Getter jako výstup ... 38
Ukázka anonymní třídy ... 40

Objektově orientovaná práce s listem 42
Funkcionální práce s listem 42
Vyhledávací dotaz 48
Ukázka datovodu 49
Kód s SQL dotazem 50
Kód s použitím datovodů 50
Implementace abstraktní metody dvěma různými způsoby 51
Ukázka funkčního rozhraní 52
Vyvolání metod z funkčního rozhraní 52
Proměnná filtrovani_mesto 53
Implicitní metoda razeni_prijmeni 53
Metoda filtrovani_mesto 54
Metoda razeni_prijmeni 54

Příloha

1. CD s aplikacemi (videopůjčovna, převodník soustav, operace s datovody) naprogramovány objektově orientovaným a funkcionálním způsobem spustitelné v prostředí NetBeans 8.2