Vysoká škola ekonomická v Praze
Fakulta informatiky a statistiky
Katedra informačních technologií

Studijní program: Aplikovaná informatika
Obor: Informatika

Analýza rozdílů mezi jazyky Java a C#

BAKALÁŘSKÁ PRÁCE

Student : Ondřej Hartman
Vedoucí : Ing. Rudolf Pecinovský, CSc.
Oponent : Ing. Vladimír Suchan

2016

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze které jsem čerpal.

V Praze dne 21. 2 2016 . . . . . .

Ondřej Hartman

Poděkování

Rád bych na tomto místě poděkoval panu Ing. Rudolfu Pecinovskému, CSc., za odborné vedení mé bakalářské práce, cenné rady a připomínky, za odborné konzultace v dané oblasti a čas, který mi při psaní této práce věnoval.

Abstrakt

Cílem bakalářské práce je popsat rozdíly mezi dvěma programovacími jazyky. Konkrétně se jedná o jazyky C# od společnosti Microsoft Corporation a Java, dříve od společnosti Sun Microsystems, nyní vyvíjený společností Oracle Corporation. Bakalářská práce analyzuje jejich koncepce, podrobně rozebírá přístupy obou jazyků k definici generických datových typů, rozdíly v definování datových typů a datových členů, porovnává operátory a srovnává možnosti sady funkcí LINQ jazyka C# a Streams API jazyka Java na stejných příkladech. Všechny rozdíly jsou demonstrovány na příkladech, jejichž většina je součástí programu, přiloženému k této práci.

Klíčová slova

Java, C#, genericita

Abstract

Main topic of this thesis is to describe differences between two programming languages. It is specifically about programming language C# developed by Microsoft Corporation and programming language Java, created by Sun Microsystems, developed in these times by Oracle Corporation. Bachelors thesis analyses their conceptions, their approach to definition of generic date types, differences in date types and date members, operators and compares possibilities of C# LINQ set of features with Java Streams API with the same examples. All differences are demonstrated by samples, most of them are part of the project, attached to this thesis.

Keywords

Java, C#, genericity

Obsah

1 Úvod 1
1.1 Vymezení tématu 1
1.2 Cíle práce 1
1.3 Způsob dosažení cíle 2
1.4 Výstupy práce a očekávané přínosy 2
2 Rešerše 3
3 Historie 5
3.1.1 Historie Javy 5
3.1.2 Historie C# 7
4 Srovnání platforem Java a .NET 10
4.1.1 Platforma Java 10
4.1.2 Platforma .NET 11
5 Běžné konvence 13
5.1 Jmenné konvence 13
5.1.1 Balíčky a jmenné prostory 13
5.1.2 Datové typy 13
5.1.3 Metody 14
5.1.4 Datové členy 14
5.2 Konvence při psaní kódu 15
5.2.1 Deklarace příkazů a proměnných 15
5.2.2 Datové členy 15
5.2.3 Bloky kódu 15
6 Jmenné prostory a balíčky 16
7 Třídy 19
7.1 Jedna třída ve více souborech 19
7.2 Finální třídy 20
7.3 Statické třídy 20
7.3.1 Statické konstruktory 21
7.4 Vnořené datové typy 22
8 Interfejsy 24
8.1 Defaultní metody 24
9 Metody 25
9.1 Klíčová slova ref, out 25

9.1.1 Ref 25
9.1.2 Out 26
9.2 Parametr s proměnným počtem argumentů 26
9.3 Metoda pro spuštění programu 27
9.4 Přebíjení metod 28
9.4.1 Virtuální metody 28
9.4.2 Zakrývání metod 29
9.5 Přetěžování metod 30
9.6 Přístupové metody 31
9.7 Rozšiřující metody 32

10 Generické datové typy 34
10.1 Rozdíly v jazycích 34
10.1.1 Syrové typy 34
10.1.2 Statické metody s generickými parametry 35
10.1.3 Generické metody 36
10.1.4 Omezení 36
10.1.5 Zástupné znaky / Žolíky 37
10.2 Rozdíly na platformách 39
10.2.1 Defaultní hodnoty 39
10.2.2 Implementace 39
10.2.3 Generické datové typy při běhu programu 42
10.2.4 Kovariance a kontravariance 42

11 Výčtové typy 44

12 Dynamické a implicitní typování 46
12.1 Klíčové slovo var 46
12.2 Co na to Java 47
12.2.1 Diamantové operátory 47

13 Funkční objekty 48
13.1 C# 48
13.1.1 Delegáty 48
13.1.2 Anonymní metody 48
13.1.3 Lambda výrazy 49
13.2 Java 49
13.2.1 Anonymní třídy 49
13.2.2 Lambda výrazy a funkční interfejsy 49

14 C# LINQ vs. Java 8 Streams API 51
14.1 Filtrace 51
14.2 Mapování 51

14.3 Řazení 52
14.4 Seskupování hodnot 52
14.5 Hodnotící operátory 53
14.6 Paralelní výpočty 53

15 Operátory 54
15.1 Přetěžování operátorů 55
15.2 Operátory vyskytující se pouze v jednom z jazyků 56

16 Regulární výrazy 58
16.1 Java 58
16.2 C# 59

17 Anotace a atributy 60
17.1 Anotace 60
17.2 Atributy 61

18 Závěr 62
18.1 Dosažené cíle 62

Terminologický slovník 63

Seznam literatury 65

Seznam obrázků 68

Rejstřík 69

Příloha A: CD 70

1 Úvod

Jak programovací jazyk Java, tak i programovací jazyk C# jsou jedny z nejznámějších a nejpoužívanějších jazyků současné doby. A přesto že oba vychází z programovacího jazyka C++ a díky tomu obsahují spoustu stejných prvků, pořád obsahují velké množství prvků, ve kterých se liší a které by měli programátoři znát, pokud již jeden z těchto jazyků umí a rozhodují se, že by se naučili ten druhý. Tento dokument by měl připravit tyto lidi na přechod z jednoho programovacího jazyka do druhého a vysvětlit hlavní rozdíly, které mezi nimi existují.

1.1 Vymezení tématu

Tématem této práce je seznámit čtenáře s rozdíly mezi programovacími jazyky Java, konkrétně ve verzi *Standard Edition (JavaSE) 8*, vydaným společností *Oracle Corporation* a C# verze 6.0 vydaným společnostní *Microsoft Corporation*. Porovnat je mezi sebou a rozebrat rozdíly, které vznikají z mnoha různých důvodu, které začínají tím, že každý z jazyků funguje na jiné platformě a pokračují přes nejrůznější konvence až například k různým funkčnostem generických datových typů a spoustě dalších rozdílů.

V příloze této práce na CD jsou přiloženy zdrojové kódy textové adventury, vytvořené v prvním semestru bakalářského studia v předmětu *4IT101 – Programování v Javě*. Tato aplikace byla následně upravena pro potřeby této práce a převedena včetně potřebných součástí Frameworku adventury do jazyku C#. Tyto kódy jsou také použity jako většina příkladů v této práci.

1.2 Cíle práce

Hlavními cíli práce bude:

- Rozebrat rozdíly
    - V běžných konvencích
    - Mezi datovými typy
    - Mezi datovými členy
    - Mezi generickými datovými typy
    - Ve funkcionálním programování
    - Mezi operátory
- Předvést na příkladech, jejichž většina je součástí programu přiloženého k této práci

1.3 Způsob dosažení cíle

Pro dosažení cíle použiji zejména dokumentace společnosti *Oracle Java 8 Language Specification* a společnosti Microsoft *C# 5.0 Language Specification* a s jejich pomocí postupně porovnám oba programovací jazyky, srovnám funkčnost každého z nich a rozdíly v jejich použití následně zobrazím na připojených příkladech. Z těchto příkladů zároveň vytvořím jednu a tu samou aplikaci, kterou naprogramuji v obou jazycích a jejich zdrojové kódy přiložím k tomuto dokumentu.

1.4 Výstupy práce a očekávané přínosy

Přínos práce spočívá zejména v popisu rozdílů, díky kterým se mohou následně ostatní programátoři, kteří už jeden z uvedených jazyků znají, jednoduše učit používat i druhý zmíněný programovací jazyk.

2 Rešerše

Před napsáním své práce jsem provedl rešerši literatury. Do této rešerše jsem zařadil knihy a dokumenty týkající se programovacích jazyků Java a C# a dále také spoustu internetových článků, popisujících rozdíly a vlastnosti těchto dvou jazyků.

V oblasti týkající se programovacího jazyka Java jsem využil hlavně dokument od společnosti Oracle *Java 8 Language specification* [2]. Tento dokument přímo od tvůrců Javy velmi detailně popisuje všechny specifikace až do její, v současné době nejnovější verze Java 8 Standard Edition a rozebírá všechny vlastnosti, které může zmíněný programovací jazyk v současné době nabídnout.

Tento dokument jsem doplnil o knihy Ing. Rudolfa Pecinovského, CSc. *Java 7 – učebnice objektové architektury pro začátečníky* [30] a *Java 8 – Úvod do objektové architektury pro mírně pokročilé*[31], zabývajícími se výukou objektově orientovaného programování ve výše zmíněném jazyce. Obě knihy výborně popisují základy programování s pomocí objektově orientovaného programování, které se následně z Javy dá, s malými úpravami, použít i v jiných jazycích, jako je například C#.

Dále pro informace o Java *Streams API* a lambda výrazech jsem využil informací z diplomové práce Maxima Rytycha *Možnosti deklarativního programování v jazyku Java 8* [47], zabývající se možnostmi deklarativního programování v jazyce Java 8 s pomocí již zmíněných lambda výrazů a *Streams API*. A dále také informace z bakalářské práce Jana Sýkory *Knihovna umožňující práci s libovolnými zdroji dat prostřednictvím SQL dotazů* [48], zabývající se vytvořením knihovny umožňující práci s libovolnými zdroji pomocí SQL příkazů.

V oblasti týkající se programovacího jazyka C# jsem využil především dokument *C# Language Specification* [1] od společnosti Microsoft Corporation. Tento dokument, podobně jako dokument jazykové specifikace Javy, obsahující většinu informací o předchozí verzi programovacího jazyka C# 5.0 a všechny jeho důležité vlastnosti. Bohužel, tento dokument je nejnovější jazykovou specifikací jazyka C#.

K tomuto jazyku jsem dále využil internetovou příručku pro programování na platformě .NET a konkrétně v jazyce C# od společnosti Microsoft Corporation. Která na svých stránkách *Microsoft Developer Network* uvádí spoustu velmi zajímavých a snadno pochopitelných příkladů ve dříve zmíněném programovacím jazyce. Na tomto webu se nachází informace o poslední vydané verzi jazyka C# 6.0 a proto jsou zde rozebrány i vlastnosti poslední verze tohoto jazyka.

Při porovnávání jazyků v oblasti generických datových typů, jsem využil článek Miroslava Viriuse *Generické programování v C++, Javě a C#* z knihy *Objekty 2005* [3], ve kterém popisuje rozdíly mezi implementací generických datových typů tří výše zmíněných programovacích jazyků a také výborný článek Jonatana Pryora *Porovnání generických dato-*

vých typů v Javě a C# [24], který porovnává implementaci generických datových typů mezi programovacími jazyky probíranými v této práci.

Následně při tvorbě praktického projektu jsem využil knihu ing. Pecinovského *OOP a Java 8: Návrh a vývoj složitějšího projektu vyhovujícího zadanému rámci* [52]. Tato kniha popisuje tvorbu tohoto projektu na vytvořeném rámci a učí začínající programátory stylům objektově orientovaného programování pomocí mnoha zajímavých informací a příkladů.

3 Historie

3.1.1 Historie Javy

Historie[1] Javy začíná na začátku 90. let minulého století, kdy malá skupina inženýrů firmy *Sun Microsystems* nazývaná *Green Team*, vedená Jamesem Goslingem vytvořila nový programovací jazyk, který měl provést revoluci ve světě. Tímto jazykem byla Java.

Prvním výrobkem, který *Green Team* vytvořil, bylo interaktivní kapesní zařízení používané pro domácí zábavu, na kterém prezentovali svůj procesorově nezávislý programovací jazyk. Avšak společnosti o toto zařízení nejevily zájem. V té chvíli se skupina se svým projektem začleňuje zpět do *Sun Microsystems* a mění svoje zaměření ze set-top boxů na online služby, CD-ROMy a desktopové platformy.

V roce 1997[2] se Java s téměř 400 000 vývojáři stává druhým nejpoužívanějším programovacím jazykem na světě. O dva roky později předefinovala firma *Sun Microsystems* architekturu Java platformy a představila Java 2 platformu ve třech různých variantách. *Java 2 Standard Edition (J2SE)* pro desktopy a pracovní stanice, *Java 2 Enterprise Edition (J2EE)* pro složité serverové systémy a *Java 2 Micro Edition (J2ME)* pro uživatelská zařízení. Tímto velmi zjednodušila programátorům, poskytovatelům služeb a výrobcům zařízení možnost, zaměřit se na vybrané segmenty trhu.

Za pět let od té doby co jazyk Java vznikl, se stihl vyvinout z nástroje pro animování webových stránek do plnohodnotného programovacího jazyka, který se dá použít prakticky všude od čipových karet, přes bankomaty, mobilní telefony a počítačové hry až ke složitému desktopovému a serverovému softwaru. V roce 2000 se na vývojářské konferenci *JavaOne* připojil na jevišti ke generálnímu řediteli společnosti *Sun Microsystems* spoluzakladatel a ředitel společnosti *Apple* Steve Jobs a prohlásil, že *Apple* připojí Javu ke každé verzi jejich operačního systému *Mac OS X*. Následující rok na té samé konferenci pronesla společnost *Sony Computer Entertainment* svou záminku integrovat platformu Java do své konzole *PlayStation 2* a dovolit tak uživatelům stahování nových aplikací a služeb dynamicky a bezpečně z prostředí internetu.

---

[1] History of Java Technology. *Oracle.com* [online]. [cit. 2016-03-06]. Dostupné z: http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html

[2] Java Timeline. *Oracle.com* [online]. [cit. 2016-03-11]. Dostupné z: http://oracle.com.edgesuite.net/timeline/java/

V roce 2004 vznikla na konferenci *JavaOne* velká debata o tom, zda by se měla Java začít vydávat jako open source[3] řešení. V současné době totiž Sun vyžadoval, aby byly všechny projekty, které byly stavěny na platformě Java, certifikovány jako kompatibilní se specifikací Javy. A všechny změny v jazyce Java musely projít procedurami *Java Community Process (JCP)*[4] což je mechanismus pro vývoj standardních technických specifikací pro technologii Javy. Během tohoto jednání odsouhlasili open source model pro Javu zástupci společností *IBM* a *Apache Software Foundation*, zatímco člen společnosti Sun a tvůrce Javy James Gosling, spolu s viceprezidentem společnosti Sun Robem Gingellem a analytikem společnosti *Red Monk* se stavěli proti. Gosslink svůj postoj obhajoval tím, že povolení více open source implementací Javy by vedlo k nekompatibilitě, podobně jako například u Unixu a později také u různých distribucí Linuxu.

Navzdory tomu však v roce 2006 společnost Sun uvolňuje Javu jako open-source projekt. Java je od tohoto roku dostupná pod licencí *GNU General Public Licence*[5], stejné, která je použita například k vývoji *OS Linux*. Sun tímto nabízí zdarma všechny tři platformy Javy pod GPL licencí.

V roce 2007 byla na konferenci *JavaOne* ohlášena *JavaFX* - nové technologické produkty designované ke zjednodušení stavby webových sítí a Java aplikací podporujících rozsáhlou škálu zařízení. A rok na to se Java dostává na další média, konkrétně Blu-ray disky.

V následujícím roce, tedy v roce 2009 je oznámeno odkoupení firmy Sun společností *Oracle*. Je to oznámeno na další konferenci *JavaOne*, kde společně vystoupili předseda firmy Sun Scot McNealy a CEO společnosti Oracle Larry Ellison, který oznámil, že Java je pro ně velice atraktivní platforma a přímo řekl, že „Middleware společnosti Oracle je postaven 100% na Javě."

Další vývoj standardů Javy je následně schvalován pomocí výkonné komise JCP a implementace open-source je přesunuta na projekt OpenJDK.[6] Následující plán, zahrnující zpětnou vazbu od komunity a schvalování pomocí výkonné komise JCP vede ke standardizaci technologií v Javě 7 pro platformu *JavaSE*, vydané v roce 2011. Zároveň s touto verzí Javy vydává Oracle vývojové prostřední *NetBeans* verze 7.2, které má plnou podporu *JavaFX 2.2*. Toto prostředí také dále podporuje jiné programovací jazyky jako například JavaScript, Groovy, C++ a PHP.

O dva roky později vychází Java 7 také pro platformu *JavaEE* a s sebou přináší mnohé vymoženosti, jako jsou například snížení času odpovědi díky používání obousměrné ko-

[3] What is open source. *Opensource.com* [online]. [cit. 2016-03-11]. Dostupné z: https://opensource.com/resources/what-open-source
[4] The Java Community Process(SM) Program. *The Java Community Process* [online]. [cit. 2016-03-11]. Dostupné z: https://www.jcp.org/en/home/index
[5] Licences - GNU Project. *The GNU Operating System and the Free Software Management* [online]. [cit. 2016-03-11]. Dostupné z: http://www.gnu.org/licenses/licenses.html#GPL
[6] OpenJDK. *Openjdk.java.com* [online]. [cit. 2016-03-11]. Dostupné z: http://openjdk.java.net/

munikace pomocí WebSocketů[7], schopných navázat interaktivní komunikaci mezi serverem a webovým prohlížečem, zjednodušení parsování dat s využitím JSONu a podpora asynchronních RESTových webových služeb.

Poslední velká změna v jazyce Java přišla v roce 2014, kdy se světu představila nová verze Javy s označením Java 8. Tato verze přinesla nejen nové programové funkce jako například zakomponování lambda výrazů, nebo nové možnosti paralelního programování pomocí *Streams API,* ale také s sebou přivedla novou verzi JavyFX, pod podobným označením *JavaFX 8*. K této nové verzi společnost Oracle vydala novou verzi jejich vývojového prostředí *NetBeans*, také ve verzi 8.0.

3.1.2 Historie C#

Během vývoje[8] .NET Frameworku byly knihovny s třídami psány pomocí kompilátoru spravovaného kódu (*Managed code compiler system*) nazývaného *Simple Managed C (SMC)*. V lednu 1999 pak Anders Hejlsberg sestavil tým, pro vytvoření nového programovacího jazyka, v té době nazývaného *Cool* (což byla zkratka pro *C-like Object Oriented Language*). *Cool* byl také název, který Microsoft pro svůj nový programovací jazyk zvažoval, jenže kvůli ochranným známkám následně svoji volbu přehodnotil a tak vznikl název C#. Do tohoto jazyku byly následně po veřejném oznámení projektu .NET v červenci roku 2000 portovány knihovny a rozhraní ASP.NET.

Přední návrhář a hlavní architekt C# Anders Hejlsberg, který byl v minulosti zapojen do návrhu jazyků *Turbo Pascal, Embarcadero Delphi* (dříve *CodeGear Delphi* a *Borland Delphi*) a *Visual J++*, uvedl ve spoustě interview a technických dokumentech, že právě chyby v mnoha jiných programovacích jazycích (např. C++, Java, Delphi nebo Smalltalk) pomohly vytvořit základy k CLR, který na oplátku vytvořil základy jazyka C#.

Našli se však i tací, jako byl například tvůrce Javy James Gosling, který spolu se spoluzakladatelem *Sun Microsystems* Billem Joyem tvrdili, že C# není nic jiného než jen imitací Javy. Gosling dokonce prohlásil že „C# je pouze Java postrádající spolehlivost, produktivitu a bezpečnost.“ Dále také Klaus Kreft a Angelika Lagner (tvůrci knihy o datovodech (angl. streams) v jazyce C++), vydali v příspěvek, ve kterém zmiňují že „Java a C# jsou skoro identické programovací jazyky. Pouze se nudně opakují a ztrácí inovaci.“ Anders Hejlsberk však na tyto narážky tvrdí že C# v žádném případě není klonem Javy a je ve svém designu dokonce více podobný jazyku C++ než Javě.

---

[7] WebSockets - Web APIs. *Mozilla Developer Network* [online]. [cit. 2016-03-11]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[8] History of C# Programming. *All About C# Programming* [online]. [cit. 2016-03-13]. Dostupné z: http://aboutcsharpprogramming.blogspot.cz/2012/09/history-of-c-programming.html

Od vydání verze C# 2.0 v roce 2005 své tvrzení také začíná dokazovat. Jak Java tak C# se v té době začaly hodně vyvíjet, každý z nich si však zvolil jinou cestu a začali se odlišovat. Jedna z velkých odlišností přišla při přidání generických datových typů do obou jazyků, kde každý z nich zvolil jiný způsob jejich implementace. Dále C# pokračoval v přidávání dalších funkcí podporujících funkcionální programování, které nakonec vyvrcholilo ve vydání sady funkcí s názvem *LINQ*, která byla vydána spolu s verzi C# 3.0. Spolu s touto knihovnou přijal C# spoustu nových funkcí jako například lambda výrazy,[9] které podporují psaní anonymních funkcí, delegátů nebo stromů výrazů.

Následně vydal Microsoft v roce 2010 C# ve verzi 4.0[10], kde představil datový typ `dynamic`, pojmenování parametrů, dobrovolné parametry a kovarianci a kontravarianci pro generické a delegátové datové typy. Následující verze C# 5.0[11] vyšla v roce 2012 a přinesla obrovskou změnu pro asynchronní programování. Konkrétně se jednalo o možnost psaní asynchronního kódu mnohem jednodušeji a přehledně, skoro stejným způsobem jako synchronní programování. K novému způsobu asynchronního programování přidal Microsoft také dvě nová klíčová slova, `async` a `await`.

Zatím poslední verzí jazyka C# byla verze 6.0[12], která byla vydána spolu s poslední verzí vývojového prostředí *Visual Studio 2015*. Zároveň s touto verzí byl vydán také nový .NET kompilátor, který je nyní vydáván jako open-source a dostal název *Roslyn*. Novinkami v této verzi jazyka jsou například Getter-Only Auto Properties, neboli vlastnosti, ve kterých není nutné definovat privátní členy pro jednoduché situace jako `get`, a `set` a dále možnost tyto vlastnosti rovnou také inicializovat.

```
public Bag Instance {get;} = new Bag();
```

V příkladu je ukázána veřejná instance jedináčka (singleton) `Bag`, která má veřejný pouze getter a zároveň rovnou přiřazenou novou instanci třídy `Bag`. Dále se velmi zjednodušila kontrola proměnných, zda nejsou `null`.

```
public int OnChanged(string name)
{
    ChangeListener?.Invoke(name);
}
```

Operátor ? dokáže zjistit, zda je možné `ChangeListener` zavolat a následně spustí požadovanou funkci. Další změnou jsou statické členy, kdy máme možnost zbavit se vypisování

[9] Výrazy Lambda. *Microsoft Developer Network* [online]. [cit. 2016-03-13]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb397687.aspx
[10] C# 4.0 new features. *ZetCode, tutorials for programmers* [online]. [cit. 2016-03-30]. Dostupné z: http://zetcode.com/lang/csharp/csharp4/
[11] New features added to C# 5.0. *DotNet Tricks* [online]. [cit. 2016-03-30]. Dostupné z: http://www.dotnet-tricks.com/Tutorial/csharp/TaH9171113-New-features-added-to-C
[12] Co je nového v C# 6.0. *Czech MSDN Blog* [online]. [cit. 2016-03-30]. Dostupné z: https://blogs.msdn.microsoft.com/vyvojari/2015/05/11/co-je-novho-v-c-6-0/

třídy před statickými členy použitím `using static ClassName`. Následně můžeme místo `ClassName.Value` volat pouze `Value`. Další novinkou v C# verze 6.0 je tzv. Interpolace stringových typů (*String Interpolation*), která nám velice zjednodušuje formátování stringů. Není nutné používat funkci `String.Format` a kryptické parametry {0},{1}…, ale k vrácení textu se jménem a příjmením nám stačí pouze znak dolaru a názvy proměnných, jak je ukázáno na příkladu pod tímto odstavcem.

```
public override string ToString()
{
    return $"Name: {Name}, Surname: {Surname}";
}
```

A díky další novince v této verzi C# nazvané *Expression-Bodied Method* je možné tuto funkci napsat pomocí lambda výrazu pouze na jeden řádek.

```
public override string ToString() => $"Name: {Name}, Surname: {Surname}";
```

Dalšími změnami v této verzi jsou například operátor `nameof`, který vrátí název daného členu a proto není nutné jej psát ručně, podmíněné výjimky, kdy je možné zadat podmínku, za které bude výjimka odchycena, použití `await` i v `catch` blocích a možnost zjednodušit inicializaci indexovaných prvků, kdy je již není nutné inicializovat jeden po druhém, ale může být použit kód podobný jako v následující ukázce.

```
var result = new Dictionary<string, int>()
{
    ["A"] => 1,
    ["B"] => 2
}
```

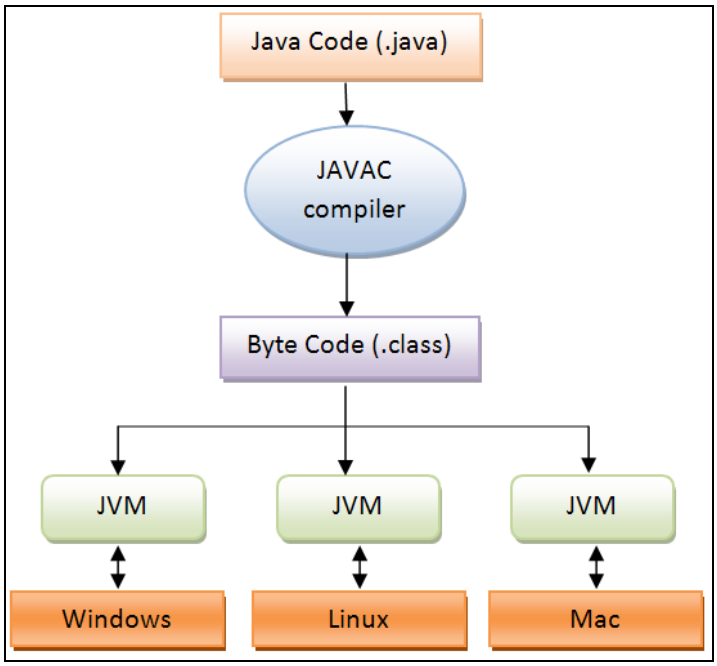
Tímto způsobem může programátor nastavit vícero klíčů a hodnot přímo při inicializaci instance třídy `Dictionary`.

4 Srovnání platforem Java a .NET

Každý ze zde probíraných programovacích jazyků funguje nativně na jiné platformě (Java na Platformě Java a C# na Platformě .NET a z toho se také odvíjí řada jejich rozdílných vlastností.

Platformy Java a .NET, na rozdíl od hardwarově založených platforem, jako jsou například již zmíněný Microsoft Windows, Linux nebo Mac OS jsou pouze softwarové platformy, které mohou být spouštěny nad výše zmíněnými platformami.

4.1.1 Platforma Java



Obrázek 1 – Schéma kompilace zdrojového kódu Javy

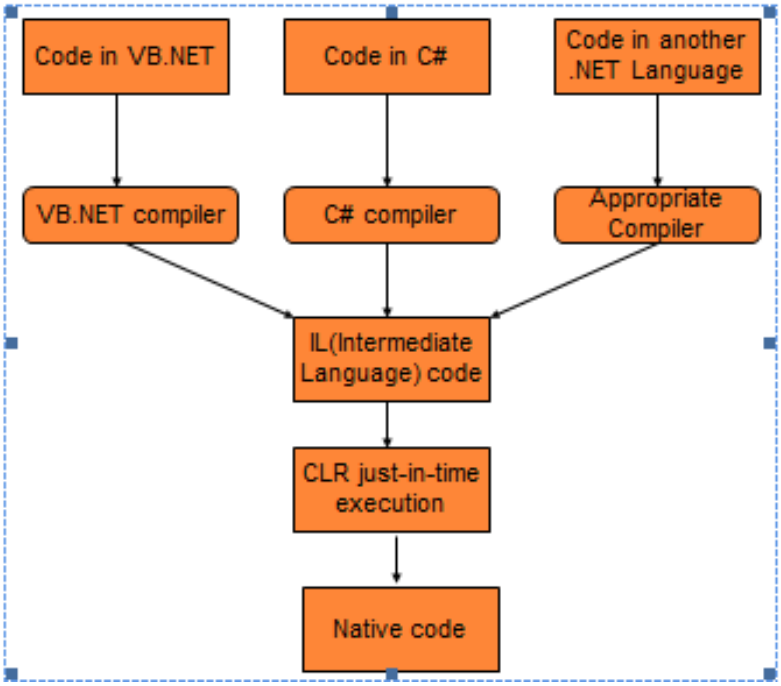
Platforma Java se skládá ze dvou součástí. První je virtuální stroj Javy (angl. *Java Virtual Machine (JVM)*), na jehož instanci se aplikace napsané v Javě nebo jiných jazycích pro Platformu Java spouští, a tou druhou je rozhraní pro programování aplikací v jazyce Java, neboli *Java API* (*Application Programming Interface*). Toto API je kolekce již předpřipravených komponent, které jsou následně sdruženy do jednotlivých knihoven a poskytují spoustu užitečných funkcí při programování v jazyce Java.

V programovacím jazyce Java je všechen zdrojový kód psán do textových souborů s koncovkou java. Tyto soubory jsou následně zkompilovány pomocí javac kompilátoru do souborů s koncovkou class. Tyto soubory však ještě neobsahují procesorem čitelný nativní

kód, namísto toho obsahují tzv. bytekód, jazyk pro virtuální stroj Javy, který tento kód dokáže přečíst a spustit.

Vzhledem k tomu, že Java je platformě závislé prostředí, může být pomalejší než programovací jazyky kompilované přímo do nativního kódu. Ovšem dnešní technologie a hardwarové možnosti současných počítačů dělají tyto rozdíly, pokud se nejedná o software s vysokými hardwarovými nároky, téměř nepostřehnutelné.

4.1.2 Platforma .NET



Obrázek 2 – Schéma kompilace zdrojového kódu pro platformu .NET

Každý z programů napsaných v C# funguje na .NET Frameworku[13]. Komponentě Microsoft Windows, která obsahuje virtuální spouštěcí systém nazývaný *Common Language Runtime (CLR)* a jednotnou sadu knihoven. CLR je komerční implementace *Common Language Interface* stvořená Microsoftem. *Common Language Interface (CLI)* je mezinárodní standard pro vytváření spouštěcích a vývojových prostředí.

Zdrojový kód napsaný v jazyce C# je tedy zkompilován do tzv. *Intermediate Language* (IL), který odpovídá specifikacím CLI. Tento kód, společně s ostatními zdroji jako obrázky a texty, je následně na PC ukládán v archivu s příponami například .exe nebo .dll. Pokud se jedná aplikace pro jiná zařízení, jako jsou například mobilní telefony, přípony mohou být

[13] https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx#Anchor_1

různé. Například .appx pro Windows Store aplikace. Archiv v sobě také ještě obsahuje soubor s informacemi (tzv. manifest), který obsahuje informace o typu sestavení, verzi a bezpečnostní požadavky.

Následně při spuštění je tento soubor načten do CLR a to se poté, díky informacím v něm uloženým, správně nastaví. Po správném nastavení, pokud je to možné, provede CLR *Just-In-Time* kompilaci[14], která následně převede potřebné části kódu v IL do počítačových instrukcí. Díky *Just-In-Time* kompilaci, nemusí CLR převádět celý program, jako to dělají AOT, neboli *Ahead-Of-Time* kompilátory, které převedou všechen kód ještě předtím, než je program spuštěn, ale převádí pouze kód, který je potřeba k současnému používání programu. Podobný způsob *Just-In-Time* kompilace[15] využívá i *Java Virtual Machine* pro kompilaci kódu uloženého v class souborech.

[14] Understanding .NET Just-In-Time Compilation. *Telerik* [online]. [cit. 2016-03-13]. Dostupné z: http://www.telerik.com/blogs/understanding-net-just-in-time-compilation

[15] Understanding Just-In-Time Compilation and Optimalization. *Oracle.com* [online]. [cit. 2016-03-13]. Dostupné z: http://docs.oracle.com/cd/E15289_01/doc.40/e15058/underst_jit.htm

5 Běžné konvence

Oba jazyky mají své standardy pro psaní kódu a jmenné konvence, která se sice dodržovat nemusí, ale programátoři jsou na ně zvyklí. Tyto konvence budou následně použitý v příkladech pro tuto práci, a proto zde budou nejprve probrány.

5.1 Jmenné konvence

Oba jazyky mají různé konvence při pojmenovávání tříd, metod i proměnných, tyto rozdíly zde budou následně probrány.

5.1.1 Balíčky a jmenné prostory

Balíčky v Javě[16] by měly být pojmenovávány pouze malými písmeny a v případě, že by mělo být víceslovné, měla by tato slova následovat přímo za sebou bez jakéhokoliv oddělovacího znaménka. Balíčku, obsahujícímu základní třídy by měl předcházet určitý prefix. Mělo by se jednat například o kód země, následovaný názvem společnosti. V případě společnosti „Company“, sídlící v České republice, by prefix mohl být například `cz.company`.

Naproti tomu jmenné prostory v C#[17], by měly pro pojmenovávání používat Pascalovu notaci (*Pascal case*). Což je způsob pojmenování, který doporučuje začít pojmenování velkým písmenem, následovaným malými písmeny a pokud je v názvu více slov, nebudou se ničím oddělovat, ale každé další slovo bude začínat velkým písmenem. Pokud se to hodí, měl by být název jmenného prostoru napsán v množném čísle. Výjimku v tomto případě tvoří případné zkratky a názvy značek.

C# na rozdíl od Javy doporučuje místo prefixu začínajícím kódem země začít názvem společnosti, za kterým bude následovat název technologie. V případě že bychom tedy měli stejnou společnost „Company“ jako v příkladu výše, vyvíjející technologii Technology, prefix by mohl být `Company.Technology`.

5.1.2 Datové typy

U obou jazyků se při pojmenovávání datových typů používá Pascalova notace. Oba jazyky také doporučují používat pro názvy tříd podstatné jméno nebo více podstatných jmen a

[16] Java Naming Standards and Programming Conventions. *IWombat.com* [online]. [cit. 2016-04-21]. Dostupné z: http://www.iwombat.com/standards/JavaStyleGuide.html
[17] Namespace Naming Guidelines. *Microsoft Developer Network* [online]. [cit. 2016-04-21]. Dostupné z: https://msdn.microsoft.com/en-us/library/893ke618%28v=vs.71%29.aspx

nedoporučuje používání zkratek, pokud nejsou více rozšířeny, než jejich dlouhé formy (jako například URL nebo HTML).

C# ještě doporučuje, že zkratky by se neměly psát velkým písmenem, pokud nejsou dlouhé dva znaky a méně (např. `UserID` nebo `PageHtml`). A pokud se to hodí, doporučuje používat název rodičovské třídy u názvů třídy potomků (například `ApplicationException` je potomkem třídy `Exception`).

Interfejsy (angl. interface), by stejně jako třídy měly používat Pascalovu notaci, ovšem oba jazyky se shodují na tom, že jejich názvy by měly být přídavná jména, popisující chování třídy. Doporučuje se, aby přídavné jméno v angličtině končilo na able (např. Clonable).

C# k tomu ještě dodává, že každý interfejs[18] by mělo začínat písmenem I, pro okamžité rozeznání, že se jedná o interfejs. Tím se název v Javě Clonable mění na IClonable v C#. Také mu nevadí používat v názvech interfejsů podstatná jména, za předpokladu že popisují chování třídy.

5.1.3 Metody

Java doporučuje pro pojmenovávání metod používat hlavně slovesa. Podstatná jména mohou být použita při popisu, co se bude měnit, například pro gettery a settery. Jméno metody by nemělo být dlouhé a mělo by přesně popisovat, co metoda dělá. To znamená, že pokud je metoda delší a dělá více věcí, než název popisuje, měla by být rozdělena na více metod.

Metoda by měla pro pojmenování používat velbloudí notaci (Camel case), která je stejná jako Pascalova notace, s výjimkou toho, že první písmeno je malé. Pro parametry metody se také používá velbloudí notace.

C# s výjimkou toho, že nepoužívá gettery a settery souhlasí s výkladem Javy s jednou změnou. Pro pojmenovávání metod nepoužívá velbloudí, ale Pascalovu notaci.

5.1.4 Datové členy

V Javě se pro pojmenovávání vlastností třídy používá velbloudí notace a jsou definovány jako privátní. Pro jejich přístup se následně používají přístupové metody. Pro názvy metod i jejich parametrů se v Javě používá velbloudí notace.

V C# se pro pojmenovávání vlastností používá Pascalova notace a jsou definovány jako veřejné. Pokud bychom chtěli někomu zabránit v úpravě atributu, nastaví se mu setter jako privátní. Více v kapitole Přístupové metody. Pro názvy metod se používá Pascalova notace a pro názvy jejich parametrů velbloudí notace.

---

[18] Interface Naming Guidelines. *Microsoft Developer Network* [online]. [cit. 2016-04-21]. Dostupné z: https://msdn.microsoft.com/en-us/library/8bc1fexb%28v=vs.71%29.aspx

5.2 Konvence při psaní kódu

Oba jazyky mají doporučení, podle kterých by se měl jejich kód psát. Některé jsou stejné, některé se liší, ale dalo by se říct, že konvence obou jazyků jsou si hodně podobné. Nejprve zde budou tedy probrány konvence, které jsou pro oba jazyky stejné.

5.2.1 Deklarace příkazů a proměnných

Oba jazyky doporučují, aby se psal pouze jeden příkaz na jednu řádku, stejně tak při deklaraci proměnných, deklarovat jednu na jednom řádku. Dále také doporučují odsazovat kód čtyřmi mezerami, Java dává uživateli na výběr, zda chce odsazovat čtyři mezery, nebo tabulátor o délce čtyř mezer. C# doporučuje použít mezery.

Další doporučení Javy při deklaraci proměnných uvnitř bloků kódu, je deklarovat tyto proměnné přímo za začátkem kódu a nečekat až na první výskyt této proměnné.

5.2.2 Datové členy

Oba jazyky se shodují v tom, že mezi deklaracemi metod by měla být vložena jedna prázdná řádka.

Java také doporučuje vkládat prázdnou řádku mezi deklaraci vlastností třídy a následující metodu.

C#, vzhledem k tomu, že vlastnosti třídy deklaruje jako veřejné a zároveň pro ně deklaruje i getter a setter, doporučuje vkládat prázdné řádky i mezi jednotlivé deklarace vlastností.

5.2.3 Bloky kódu

Java doporučuje, při deklaraci bloků kódu, jako například u metod, ale také u tříd, podmínek, nebo cyklů umísťovat složené závorky, které označují začátek bloku na stejný řádek, kde se nachází jeho deklarace. Složené závorky označující konec bloku doporučuje vložit na nový řádek. Java také doporučuje použít složené závorky pro označení bloku kódu i při definování podmínek nebo cyklů, které obsahují pouze jeden příkaz.

C# doporučuje při deklaraci bloků kódu umísťovat nejen složenou závorku označující konec bloku ale i tu, která označuje začátek bloku na samostatný řádek, pod deklaraci tohoto bloku.

6 Jmenné prostory a balíčky

Pro větší přehled při psaní programů, se pro organizaci používají jmenné prostory (angl. namespace) v jazyce C# a balíčky (angl. package) v jazyce Java. Obě dvě organizační jednotky slouží hlavně ke zlepšení přehledu organizační struktury, pro následné uložení a lepší práci s vytvořenými třídami programu.

Pro obě jednotky platí určitá pravidla. Jako například že všechny datové typy jako třídy, interfejsy, výčtové typy atd. musí mít v každém balíčku nebo jmenném prostoru různé názvy. Je tedy možné mít dva jmenné prostory, například s názvy `Xharo03Hartman` a `GameTxt` a v každém z nich mít vytvořený interfejs `IAuthor`. Ovšem není možné mít interfejs `IAuthor` vytvořený dvakrát v balíčku, nebo jmenném prostoru `GameTxt`.

```
namespace Xharo03Hartman
{
    interface IAuthor
    {
    }
}

namespace GameTxt
{
    interface IAuthor
    {
    }
}
```

Pokud je použita definice, která je vytvořena výše, kompilace proběhne v pořádku, protože každý interfejs s názvem `IAuthor` má svůj v tomto případě jmenný prostor, do kterého patří a interfejsy na sebe vzájemně nevidí. Kdyby však byly interfejsy v jednom jmenném prostoru nebo balíčku, nastala by chyba, protože kompilátor by nebyl schopný zjistit, z jakého interfejsu následující třída dědí.

```
package xharo03_hartman;
interface IAuthor {
    //Kód interfejsu
}
interface IAuthor {
    //Kód interfejsu
}
public class ScenarioManager extends AScenarioManager implements IAuthor {

}
```

Mezi balíčky a jmennými prostory se však vyskytuje také několik rozdílů. Balíčky, na rozdíl od jmenných prostorů slouží pouze jako organizátor souborů a proto na sebe nenavazu-

jí. Mohlo by proto dojít k omylu myslet si, že pokud bychom měli balíčky `eu` a `eu.pedu` je mezi nimi nějaký vztah. To nikoliv, balíček `eu.pedu` je pouze balíčkem s jiným názvem a nemá žádný vztah k balíčku `eu`.

Na rozdíl od toho jmenné soubory mají své místo působení, každý jmenný prostor má své tělo, které je odděleno složenými závorkami a v tomto místě platí působnost onoho balíčku. Po ukončení složených závorek a těla jmenného prostoru, je možné deklarovat nový jmenný prostor s jiným názvem. Díky tomuto je možné také jmenné prostory na sebe navazovat a proto je ve jmenném prostoru možné, kromě typů vypsaných výše, deklarovat také nový jmenný prostor.

```
//První jmenný prostor
namespace TestUtil
{
    //Jmenný prostor v jiném jmenném prostoru
    namespace Common
    {
    }
}
//Další jmenný prostor
namespace Utilities
{
}
```

Tento kód je možné zapsat do jednoho souboru a následná kompilace tohoto souboru projde bez chyby.

Dalším rozdílem mezi Javou a C# je používání jmenných prostorů a balíčků v dalším kódu programu. Jak třídy Javy, tak i třídy C# mohou vidět na třídy, nacházející se ve stejném balíčku a jmenném prostoru, ve kterém byly definovány ony. Při použití tříd, které se ve stejném balíčku, či jmenném prostoru nenachází, je potřeba tuto třídu nejprve nadefinovat. Při definování třídy z jiného balíčku se na ni v obou jazycích můžeme odkázat buď celým názvem včetně jejích balíčků:

```
eu.pedu.adv16s.bp.xharo03_hartman.Main;
```

Nebo ji nejprve naimportovat v Javě do souboru pomocí klíčového slova `import`. Pokud importujeme do souboru v jazyce Java, odkazujeme se přímo na třídu, kterou chceme importovat. Pokud bychom chtěli importovat všechny třídy z balíčku, použijeme k tomu znak hvězdy.

```
//import jedné třídy
import eu.pedu.adv16s.bp.xharo03_hartman.Main;
//import všech tříd z balíčku eu.pedu.adv16s.bp.xharo03_hartman
import eu.pedu.adv16s.bp.xharo03_hartman.*;
```

Na rozdíl od toho v C# importujeme jmenné prostory pomocí klíčového slova `using`. Pomocí tohoto slova se importují všechny třídy v uvedeném jmenném prostoru a ty se dají následně použít. Klíčové slovo `using` se dá použít jak uvnitř jmenného prostoru, tak i vně a následně budou tyto třídy dostupné pro všechny jmenné prostory v současně používaném souboru. C# kromě tohoto nabízí ještě jednu možnost importu pomocí `using` a to s možností definovat si alias pro určitou třídu a naimportovat pouze tu. Proto, pokud by nějaká třída již importovaná třída, nebo třída v současném jmenném prostoru měla stejný název jako ta, kterou hodláme použít, nemusíme použít její celý název, ale můžeme se na ni odvolat pomocí námi zvoleného aliasu. Alias je samozřejmě možné použít i při importu jmenných prostorů.

```
//import jmenného prostoru Commands;
using Pedu.Adv16s.BP.Xharo03Hartman.Commands;
//import jmenného prostoru Commands při použití aliasu C
using C = Pedu.Adv16s.BP.Xharo03Hartman.Commands;
//import pouze třídy CommandHelp ze jmenného prostoru Commands jako alias ComH
using ComH = Pedu.Adv16s.BP.Xharo03Hartman.Commands.CommandHelp;
```

Nyní je možné použít třídu `CommandHelp` ze jmenného prostoru `Commands` jako třídu `ComH`, bez nutnosti se na ni odkazovat celým jejím názvem i pokud bude jmenný prostor obsahovat třídu, se stejným názvem. Pokud by se tato třída měla použít v balíčku Javy, který již tuto třídu obsahuje, bylo by nutné ji volat celým jejím názvem, který by v Javě byl `eu.pecu.adv16s.bp.xharo03_hartman.commands.CommandHelp`.

7 Třídy

Oba programovací jazyky, jak Java, tak C# jsou objektově-orientovanými programovacími jazyky. To znamená, že objekt je základním stavebním prvkem celého programu. Třída je poté základním konstrukčním prvkem v objektově orientovaném programování a slouží jako šablona pro následně objekty, které jsou instancemi oné třídy. V obou případech jak v Javě tak v C# všechny třídy implementují základní třídu `Object`. Od ní následně dědí všechny další třídy.

7.1 Jedna třída ve více souborech

C# na rozdíl od Javy přišel s podporou rozdělení jedné třídy do více souborů pomocí klíčového slova `partial`. Díky tomu je možné vytvořit část třídy, kterou je následně možné doplnit dalšími funkcemi z jiného souboru při vytvoření třídy se stejným jménem včetně jmenného prostoru a klíčového slova `partial`. Díky tomu je tedy možné doplnit třídy o další prvky a ty následně použít jako rozšíření dané třídy.

```
namespace Utilities
{
    partial class InputForm
    {
        public void InitializeComponent()
        {
            ...
        }
    }
}
//Stejná třída v jiném souboru
namespace Utilities
{
    partial class InputForm
    {
        public Field Play()
        {
            button1_Click(null, null);
        }
    }
}
```

Tímto způsobem je možné jednu třídu doplnit o další prvky, jako například ve výše uvedeném příkladu, kde je třída `InputForm` rozšířena v dalším souboru o funkci `Play`. C# tento způsob dělení tříd zavedl například kvůli tomu, aby bylo možné při tvoření grafických formulářů oddělit generovaný kód od toho, který píše programátor. Díky tomu se může

programátor starat o své věci a mít přehled o tom co píše a všechen vygenerovaný kód je uložen v jiném souboru, do kterého může volně přistupovat, když potřebuje něco změnit.

7.2 Finální třídy

Oba jazyky povolují vytvoření třídy, od které následně nemůže žádná další třída dědit a není tak možné ji rozšiřovat o její další potomky. V Javě se tyto třídy označují klíčovým slovem `final`, v C# poté klíčovým slovem `sealed`.

7.3 Statické třídy

Dle definice Microsoftu statické třídy[19] (static classes) v C# jsou třídy, u kterých není možné vytvářet jejich instance pomocí klíčového slova `new`. Není v ní totiž možné vytvářet žádné nestatické členy, všechny vlastnosti a metody, které v ní jsou definovány, jsou definovány jako statické. Tyto třídy mohou být použity jako kolekci pro určitou skupinu metody, které mají něco společného. Příklad této třídy je například třída hry `Constants`, z balíčku `Utils`, obsahující texty hry a metodu pro spojování elementů polí typu `string`.

V C# je implementace statických tříd jednoduchá, při definování třídy se použije klíčové slovo static v přístupových parametrech a kompilátor už se následně postará o to, že ve třídě není možné definovat nestatické členy.

```
public static class Constants
{
    public static string Connect(params string[] strings)
    {
        //Kód metody
    }
}
```

V Javě existuje jiný postup tvorby statických tříd[20]. Java sama nezavádí pro tyto třídy pojem „statická třída“, spíše používá pojmu utility nebo library class. Pro nadefinování knihovní (library) třídy v Javě je tedy nutné dodržet několik následujících pravidel.

- V deklaraci třídy použít klíčové slovo final, pro znemožnění tvorby potomků statické třídy.

[19] Statické třídy a její členové. *Microsoft Developer Network* [online]. [cit. 2016-04-18]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/79b3xss3.aspx
[20] Static Classes In Java. *Stack Overflow* [online]. [cit. 2016-04-18]. Dostupné z: http://stackoverflow.com/questions/7486012/static-classes-in-java

- Konstruktor v této třídě definovat jako privátní, pro zamezení tvorby instancí statické třídy.
- Všechny členy statické třídy definovat jako statické pomocí klíčového slova static. Není možné, aby ve statické třídě byly definovány nestatické členy.

```
public final class Constants {
    private Constants() {
    }
    public static string connect(String... strings) {
    }
}
```

7.3.1 Statické konstruktory

Před rozebíráním statických konstruktorů je nutné si uvědomit jednu věc a to, že pokud je v objektově orientovaném programování by mělo být objektem úplně všechno všechno. To znamená že je objektem i třída a ta potřebuje ke svému stvoření nějaký konstruktor. Tím je výše zmíněný statický konstruktor.

Statický konstruktor[21] je používán pro inicializace statických dat, nebo provádění akcí, které jsou potřeba provést pouze jednou, před inicializací první instance dané třídy, nebo vytvořením prvních statických členů. Je potřeba si však uvědomit, že statický konstruktor není možné přímo zavolat a proto programátor nemá žádnou kontrolu nad tím, kdy se tento konstruktor spustí.

Statické konstruktory v C#

C# podporuje tvorbu statických konstruktorů, rozdíl mezi klasickým a statickým konstruktorem v C# je pouze ten, že při definování statického konstruktoru se použije klíčové slovo static a žádné přístupové.

```
public class ACommand
{
    static ACommand()
    {
        new CommandDownload();
        ...
    }
}
```

[21] Statické konstruktory (Průvodce programováním v C#). *Microsoft Developer Network* [online]. [cit. 2016-04-18]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/k9x6w0hc.aspx

Statické inicializátory v Javě

Na rozdíl od C#, Java nepoužívá pojmenování statické konstruktory. Místo toho používá statické bloky, nebo také statické inicializátory, kde se do třídy vloží pouze blok kódu ohraničený složenými závorkami, před kterými se deklaruje modifikátor static. Prakticky se ovšem jedná o jedno a to samé, jako statický konstruktor. Pokud by před ním tento modifikátor nebyl, tento kód by byl spuštěn při inicializaci instance.

```
public class ACommand {
    static {
        new CommandDownload();
        ...
    }
}
```

7.4 Vnořené datové typy

Za další rozdíl by se daly považovat třídy, interfejsy nebo výčtové typy, definované uvnitř jiné třídy nebo interfejsu[22]. Zatímco C# má pouze jeden typ vnořených datových typů, v Javě se tyto typy dělí na nestatické, vnitřní třídy (inner class), a statické, vnořené datové typy (nested types). V předešlé větě je uveden výraz vnitřní třídy, protože pouze třídy mohou být vnitřní, vnořené mohou být i interfejsy a výčtové typy.

Statické datové typy v Javě by se daly porovnat s vnitřními datovými typy, které implementuje C#. Tyto datové typy mají přístup pouze ke statickým typům typu, ve kterém je daný typ definován. A nemají přístup k žádným proměnným existujících instancí tohoto typu. Na druhou stranu ale mohou implementovat statické metody.

```
// Statická vnitřní třída v Javě
public class Time {
    private static class Hour {
    }
}

//Její ekvivalent v C#
public class Time
{
    private class Hour
    {
    }
}
```

[22] C# From a Java Developer Perspective. *Dare Obsanjo's Home Page* [online]. [cit. 2016-04-07]. Dostupné z: http://www.25hoursaday.com/CsharpVsJava.html#nested

Na druhou stranu, nestatický datový typ v Javě nemůže implementovat žádné statické metody, ale na rozdíl od toho má přístup k proměnným instance datového typu, ve kterém se nachází.

Java na rozdíl od C# poskytuje i možnost definovat třídu uvnitř metody. Tato třída je pak použitelná pouze v kontextu oné metody a jinde přístupná není.

```
class TestClass {
    public void classInsideMethod() {
        class MethodClass { }
        MethodClass mClass = new MethodClass();
    }
}
```

8 Interfejsy

Interfejs (angl. interface) funguje jako kontejner pro definice funkcionalit, který následně mohou třídy implementovat.

Oba jazyky používají interfejsy pro definici seznamu abstraktních metod, které je následně nutné implementovat uvnitř třídy, a tyto třídy se následně mohou vydávat za potomka onoho interfejsu. V obou jazycích mohou třídy dědit z několika interfejsu, ovšem pouze z jedné rodičovské třídy.

```
public interface INamed {
    string getName();
}
```

8.1 Defaultní metody

Na rozdíl od C#, Java do svých interfejsů dovoluje vkládat také konstanty, statické metody a defaultní metody.

Defaultní metoda je klasická metoda, která v definici obsahuje klíčové slovo default a díky tomu je možné v interfejsu této metodě přiřadit základní funkčnost.

```
public interface IAuthor {
    String getAuthorName();
    String getAuthorID();

    default String getAuthorString() {
        return getAuthorID() + " " + getAuthorName();
    }
}
```

C# na rozdíl od Javy žádné defaultní metody neimplementuje, dle jeho názoru je totiž interfejs pouhým kontejnerem, kde se ukládají abstraktní metody, které musí následně nadefinovat třída, která interfejs implementuje. Ovšem je možné tuto funkcionalitu implementovat pomocí rozšiřujících metod, popsaných v následující kapitole.

9 Metody

Mezi metodami v Javě a C# není moc rozdílů, jedná se spíše o způsob zápisu metod, než velké změny mezi oběma jazyky, ovšem i tyto změny je dobré popsat.

9.1 Klíčová slova ref, out

U obou jazyků je známo, že když předáváme nějaké metodě objekt, je do ní předána jeho reference a tuto referenci je možné uvnitř dané metody měnit a následné změny se projeví i vně této metody. Na druhou stranu, pokud do metody vložíme primitivní datový typ, tak pokud ho uvnitř metody změníme, tato změna se neprojeví vně metody a hodnota parametru tedy zůstává stejná, jako byla před odesláním do metody.

C# na tuto vlastnost reaguje klíčovými slovy `ref` a `out`[23], v Javě, tato funkčnost ještě nebyla implementována a jediná možnost jak přiřadit novou hodnotu do primitivní proměnné je tedy vytvořit funkci která má určitou návratovou hodnotu a tuto hodnotu následně přiřadit do oné proměnné. Metody s použitím klíčových slov `ref` a `out` fungují jiným způsobem.

9.1.1 Ref

Pokud má volaná metoda v parametru klíčové slovo `ref`, místo hodnoty se do ní předá reference na daný primitivní datový typ a změny provedené v této metodě, se následně propíší i do primitivního datového typu, který byl funkci předán.

```
static void WindowInput()
{
    var answer = zUVÍTACÍ_ZPRÁVA;
    var command = "";
    InputBox.Show(null, answer, ref command);
}
public class InputBox
{
    public static void Show(string title, string promptText, ref string value)
    {
        value = textBox.Text;
    }
}
```

[23] Main() and Other Methods (C# vs Java). *Microsoft Developer Network* [online]. [cit. 2016-04-09]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms228506%28v=vs.90%29.aspx

9.1.2 Out

Pokud by volaná metoda obsahovala v parametru místo klíčového slova `ref`, klíčové slovo `out`, byla by ignorována jeho výchozí hodnota, která byla do funkce předána. A do primitivní proměnné přidané pomocí tohoto klíčového slova by bylo nutné přiřadit novou hodnotu, která se následně projeví vně této metody v přiřazené proměnné. Před použitím parametru předaného pomocí klíčového slova `out` je nutné tento parametr v metodě nejdříve inicializovat. Protože nepředává žádnou hodnotu a použití před přiřazením by vyhodilo chybu při kompilaci.

9.2 Parametr s proměnným počtem argumentů

C#

Dalším klíčovým slovem používaným v parametru metody je klíčové slovo `params`. Pokud máme metodu, která jako parametr obsahuje pole, před kterým se vyskytuje toto klíčové slovo. Je možné do metody zadat libovolné množství arguentů stejného typu jako je ono pole. Všechny tyto argumenty v metodě následně najdeme v parametru, který definovaném jako pole onoho typu s klíčovým slovem params. Jedinou podmínkou je, že tento parametr může být v metodě pouze jednou a musí se nacházet na posledním místě v seznamu parametrů. Jinak by nebylo možné poznat, kde tento parametr končí.

Java

Java také obsahuje stejnou možnost zadávání libovolného počtu argumentů stejného typu do parametru metody, který je metodě předán jako pole. Ovšem zde se toto pole neoznačuje pomocí klíčového slova `params`, nýbrž pomocí tří teček místo značení pole, což naznačuje, že pole může mít libovolnou velikost. I zde platí, že parametr může být v metodě pouze jednou a na konci metody.

```
//Metoda s libovolným počtem parametrů v C#
public static string Connect(params string[] strings) { /* Tělo metody */ }

//Ta samá možnost v Javě
public static String connect(String... strings) { /* Tělo metody */ }
```

9.3 Metoda pro spuštění programu

Mnoho spustitelných programů (tím jsou myšleny programy, které se dají spustit, samostatné knihovny a frameworky) napsaných v jazycích Java a C# obsahuje jednu spustitelnou metodu. V Javě se tato metoda označuje jako `main(String… args)`, v C# jako `Main(string[] args)`. Tato metoda musí být v obou programovacích jazycích statická, u jazyka Java také veřejná. C# na rozdíl od Javy doporučuje, že jeho metoda `Main(string[] args)` by veřejná být neměla.

```
//Metoda main v jazyce Java
class Main {
    public static void main(String[] args) {
        if((args.length > 0) && ("-text".equalsIgnoreCase(args[0]))){
            consoleInput();
        }
        else{
            windowInput();
        }
        System.exit(0);
    }
}
```

```
//Metoda main v jazyce C#
static void Main(string[] args)
{
    if (args.Length > 0 && args[0] == "-text")
    {
        ConsoleInput();
        }
        else if (args.Length > 0 && args[0] == "-old")
        {
            WindowInputOld();
        }
        else
        {
            WindowInput();
        }
}
```

Třída obsahující metodu `main`/`Main` nemusí být ani v jednom z jazyků statická a může se dále použít pro vytváření nových instancí. Dalším rozdílem je, že na rozdíl od Javy, může metoda `Main` v C# obsahovat kromě návratového typu `void` také typ `int`.

V obou jazycích může výše zmíněná metoda přijmout nějaké textové parametry, tyto parametry se zadávají jako argumenty programu a oddělují se mezerami. Následně jsou do programu vloženy jako pole textových řetězců.

C# má nastaveno použití tohoto argumentu jako dobrovolné a proto není nutné ho do metody `Main` psát. Na rozdíl od Javy, ta má sice povinný tento parametry do metody `main` vložit, ovšem toto pole může být prázdné, takže se před spuštěním programu žádné argumenty stejně vkládat nemusí.

9.4 Přebíjení metod

Přebíjení[24] (override) je jazyková vlastnost, která dovoluje v potomkovi přebít metodu své rodičovské třídy. To znamená, že dovolí potomkovi definovat metodu stejného jména, se stejnou návratovou hodnotou, která má stejné typy parametrů definované ve stejném pořadí, jako metoda rodičovská. Je však rozdíl v přebíjení metod v jednotlivých jazycích.

9.4.1 Virtuální metody

Java

Java má totiž na rozdíl od C# defaultně metody deklarované jako virtuální a pokud u nich není použité klíčové slovo `final`, které říká, že metoda se nedá dále přebíjet. Je možné je jakkoliv přebíjet v potomcích oné třídy. Při přebíjení metod v Javě se před metodu přidává anotace `@Override`. Není to nutné, ale je to dobrý zvyk. Pokud tam toto slovo totiž je, kompilátor při kompilaci kódu zkontroluje, jestli opravdu v předkovi existuje ona metoda se stejným názvem, návratovou hodnotou a parametry v zadaném pořadí. Pokud tato metoda neexistuje ne, vyhodí chybu. Pokud by tam anotace `@Override` nebyla, kompilátor metodu i tak přepíše, ale pokud jsou například špatně napsané parametry, tak to už kompilátor nezjistí a chybu nevyhodí.

C#

V C# je to právě naopak. Metody jsou defaultně považovány za finální a není možné je přebít. Proto by mohlo docházet ke špatnému použití metod, kdy instance potomka použije metodu předka, i když měl použít svou vlastní přebíjející metodu. Tohle se v C# řeší klíčo-

[24] Overriding and overloading in Java and .NET - differences, changes and gotchas. *CodeProject* [online]. [cit. 2016-04-15]. Dostupné z: http://www.codeproject.com/Articles/692127/Overriding-and-overloading-in-Java-and-NET-differe

vými slovy virtual a override. V rodičovské třídě se nejprve při definování metody použije klíčové slovo virtual, které říká, že metodu je možné v potomcích dané třídy přebíjet a následně se v definici metody v potomkovi použije klíčové slovo override, které říká, že tato metoda přebíjí virtuální metodu své rodičovské třídy.

```
public abstract class AFile {
    public String execute() {
        return Constants.Znespustitelny;

}
public class FilmFile extends AFile {
    public String execute() {
        return Constants.zSHLEDNUTO;
    }
}

public static void Main() {
    AFile file = new FilmFile();
    file.execute();
}
```

Po spuštění předchozího zjednodušeného příkladu v Javě by výstup metody execute byl "Právě jste shlédli film. Doufám že se vám líbil :)". Je to kvůli tomu, že i když se navenek instance c tváří jako instance třídy AFile, stále je instancí třídy FilmFile a proto používá její překrytou metodu print.

Pokud by byl ale podobný kód, pouze trochu upravený pro C# spuštěn v C#, výstup by byl "Chyba: Tento soubor nelze spustit.".

9.4.2 Zakrývání metod

Kvůli tomu, že metoda ve třídě AFile neobsahuje klíčové slovo virtual, program neví, že nějaká další metoda existuje. V tomto případě by tedy nešlo o přebíjení ale o zakrývání metod (method hiding). Metoda ve třídě FilmFile pouze zakryla metodu ve třídě AFile, ovšem protože se poslední instance vydává za instanci třídy AFile, třída neví, že třída FilmFile obsahuje metodu, která překrývá metodu třídy AFile a použije tu ze třídy AFile. Pokud by však bylo zakrývání metody použito úmyslně, mělo by se do deklarace metody přidat klíčové slovo new. Pokud tam není, kompilátor vyhodí varování, použití klíčového slova new při zakrývání metody toto varování skryje.

```
public class FilmFile: AFile
{
    public new string Execute()
    {
        return Constants.zSHLEDNUTO;
    }
}
```

Pokud by měl být výstup v jazyce C# stejný jako v jazyce Java, kód obou tříd by musel vypadat následovně.

```
public class AFile
{
    public virtual string Execute()
    {
        return Constants.zNESPUSTITELNY;
    }
}
public class FilmFile: AFile
{
    public override String Execute()
    {
        return Constants.zSHLEDNUTO;
    }
}
```

V Javě je možné zakrýt pouze statické metody, instanční metody je možné pouze přebít nebo přetížit.

9.5 Přetěžování metod

Přetěžování (overloading), na rozdíl od překrývání je vlastnost, která dovoluje programátorům vytvořit dvě metody se stejným názvem ale jinými parametry a tyto metody umístit do stejné třídy.

```
//C#
public static E GetO<E>(string member, E[] array) where E : struct, INamed
{
    return array.First(e =>
    {
        returne e.GetName()
           .Equals(member,StringComparison.CurrentCultureIgnoreCase);
    });
}
public static E? GetO<E>(string member, List<E> list) where E : struct, INamed
{
    return GetO(member, list.ToArray());
}

//Java
public static <E extends INamed> Optional<E> getO(String member, E[] array) {
    return getO(member, Arrays.stream(array));
}
public static <E extends INamed> Optional<E>
        getO(String member, Stream<E> stream) {
    return stream.filter(iNamed -> iNamed.getName().equalsIgnoreCase(member))
                 .findAny();
}
```

9.6 Přístupové metody

Každý objekt může mít definovaný nekonečný počet vlastností a tyto vlastnosti objekty jistým způsobem nastavují (`set`) a poskytují (`get`) ostatním objektům kolem sebe. Oba dva zde zmiňované jazyky mají svůj vlastní styl poskytování a nastavování vlastností svých objektů.

Java

Java své vlastnosti v objektech dle zavedených konvencí ukrývá (označuje jako `private`) a poskytuje je ostatním objektům pomocí přístupových metod. Ty se běžně označují následovně:

- Metody, kterými se vlastnosti v objektech nastavují, běžně začínají slovem set, a proto se nazývají settery.
- Metody, kterými je možné tyto vlastnosti z objektů získat, běžně začínají slovem `get`, a proto se nazývají gettery.
- Zvláštními typy getterů jsou gettery, které vracejí proměnnou typu `boolean`, ty poté mohou začínat místo slovem `get`, slovem `is`.

Pokud má být poté například zabráněno ostatním objektům tuto vlastnost nastavovat, není nic jednoduššího než nastavit setter vlastnosti jako private, popřípadě ho vůbec neimplementovat.

```
private Folder currentFolder;

public Field getCurrentPlace() {
    return currentFolder;
}

public void setCurrentPlace(Folder folder) {
    currentFolder = folder;
}
```

C#

C# na rozdíl od Javy implementuje gettery a settery svým způsobem. Dle jeho konvencí pokud má být vlastnost třídy přístupná, nastaví se její přístup jako public a poskytne se všem ostatním objektům přímo z instance třídy. Pokud bychom chtěli následně getter a setter vlastnosti nějak změnit, je možné to provést přímo u vlastnosti pomocí klíčových slov `get` a `set` a pokud bychom chtěli, aby měla vlastnost například privátní pouze setter, je možné nastavit přístup setteru vlastnosti jako `private`.

```
private static Bag _instance;
public static Bag Instance
{
    get { return _instance ?? (_instance = new Bag()); }
    private set { _instance = value; }
}
```

9.7 Rozšiřující metody

Rozšiřující metody (Extension methods) je výborná vlastnost C#, umožňující přidávat do třídy nové metody, bez toho, aniž bychom museli tvořit jejich potomky a bez úprav stávající třídy.

Funguje to tím způsobem, že se ve třídě vytvoří nová statická metoda, které se jako parametr předá odkaz na instanci objektu, se kterým budeme pracovat, kterému předchází klíčové slovo `this`. Tím zajistíme, že při zavolání metody z instance třídy se tomuto parametru předá právě ta instance třídy, která metodu volá. Následně můžeme s tímto parametrem provádět různé změny, jako by to byla metoda třídy.

Pro použití metody musí být pomocí using importován jmenný prostor, který obsahuje třídu s onou metodou, pokud se třída, ve kterém se používá, nenachází ve stejném jmenném prostoru.

```
public interface IAuthor
{
    string GetAuthorName();
    string GetAuthorID();
}
public static class AuthorExtensions
{
    public static string GetAuthorString(this IAuthor iAuthor)
    {
        return iAuthor.GetAuthorID() + " – " + iAuthor.GetAuthorName();
    }
}
```

10 Generické datové typy

Tato kapitola bude rozdělená na dvě části. V první části budou probírány ty rozdíly, které programátor přímo vidí při psaní kódu, ve kterém používá generických datových typů. Ve druhé části budou dále probrány rozdíly, které přímo vidět nejsou. To jsou ty, které probíhají až při kompilování a spouštění kódu na platformách jednotlivých jazyků. Nejdříve je však důležité popsat, co přesně generické datové typy jsou a jak se používají.

Generický datový typ je datový typ, který přijímá generický parametr, do kterého je přiřazen typ vybrané třídy, který je následně používán v kontextu onoho generického typu.

```
class ListINamed<E>
{
    public static void Test()
    {
        ListINamed<string> namedList = new ListINamed<string>();
    }
}
```

V příkladu nahoře je vidět generická třída `ListINamed`, která má generický parametr, `E`. Následně je poté ve statické metodě `Test` vytvořena instance této třídy, kde je za generický parametr dosazena třída `string`.

Generické datové typy[25] jsou podporovány v obou zmiňovaných programovacích jazycích. Konkrétně v Javě od verze Java 5.0 a v C# od verze 2.0.

10.1 Rozdíly v jazycích

10.1.1 Syrové typy

Prvním zmíněným rozdílem budou v Javě používané tzv. syrové typy (raw types). V C# je chyba, když vytvoříte generický typ, kterému nepřidáte žádný generický parametr. Java však tuto možnost podporuje a říká se jí syrový typ[26].

[25] Comparing Java and C# Generics. *Jonatan Pryor Research Labs* [online]. [cit. 2016-04-09]. Dostupné z: http://www.jprl.com/Blog/archive/development/2007/Aug-31.html
[26] *Generické programování v jazyce Java*. České Budějovice, 2009. Bakalářská práce. Jihočeská univerzita v Českých Budějovicích.

```
//Kód v Javě, který projde
ArrayList list = new ArrayList();

//Ekvivalent v C# který vyhodí výjimku
List list = new List();
```

Syrové typy neboli typy, které neobsahují žádné generické parametry, jsou základní typy, které v Javě existovaly až do verze 5.0. Ve které, jak již bylo zmíněno, byly zavedeny generické datové typy. U syrových typů je však problém, že nemůžeme okamžitě zjistit, zda do ní přidáváme stejné datové typy. A proto je možné chybně přidat do listu, ve kterém již máme uloženou proměnnou typu `int`, například proměnnou typu `String`. Kvůli tomu nám Java vyhodí výjimku až při běhu programu.

Důvod proč jsou syrové typy v Javě v současné době povoleny, je umožnění spolupráce se starším kódem Javy, který je ještě používán. Ovšem není problém zajistit například práci instance s typem `String` tím, že předáme staré metodě s argumentem typu `List` parametrizovaný typ `List<String>`.

```
public static void setStr(List list) {
    //Kód metody
}
public static void main (String… args) {
    List<String> strList = new ArrayList<String>();
    setStr(strList);
}
```

10.1.2 Statické metody s generickými parametry

Dalším rozdílem, nyní ve prospěch C# je možnost deklarace statických metod s argumenty obsahujícími generické parametry. Pokud bychom tuto metodu deklarovali v Javě, obdržíme chybu, která nám sdělí, že nemůžeme deklarovat statickou metodu s generickými parametry. Je to z toho důvodu, že v Javě je nejprve nutné deklarovat generické parametry pomocí vytvoření instance dané třídy. Teprve poté je možné tyto metody použít a z toho důvodu není nutné, aby tyto metody byly statické.

C# tuto možnost vyřešil po svém. Dovoluje programátorům zadat statické typy přímo za voláním třídy a díky tomu je možné je dostat do statických funkcí.

```
class GenerickaTrida<T>
{
    public static void Metoda(T t)
    {
    //Kód metody
    }
}

class HlavniTrida
{
    static void Main(String[] args)
    {
        GenerickaTrida<int>.Metoda(10);
    }
}
```

10.1.3 Generické metody

Oba programovací jazyky, v současné době podporují generické metody, ve kterých samotná metoda přijímá generické parametry. Tyto metody mohou být deklarovány jak v generických tak i v negenerických třídách a interfejsech. Jediný rozdíl je v zápisu těchto metod. Zatímco v Javě se generické parametry deklarují před návratovým typem metody, v C# se deklarují až za jejím názvem.

```
//Java
public static <T> T metoda(T a, T b) {
    //Kód metody
}

//C#
public static T Metoda<T>(T a, T b)
{
    //Kód metody
}
```

10.1.4 Omezení

V obou zmiňovaných programovacích jazycích je možné omezit typy, které je možné vložit do tříd a metod jako generické parametry. V Javě se omezení vkládá ke generickým parametrům mezi <> znaky. Syntaxe pro přidání parametru s listem omezení vypadá následovně:

```
<generickýParametr extends seznamOmezení>
```

Seznam omezení v tomto případě obsahuje omezení oddělených od sebe znakem `&` a je od generického parametru oddělený klíčovým slovem `extends`. Tím vlastně říkáme, že generický parametr bude dědit od tříd zmíněných v seznamu omezení. Dále je možné použít klíčové slovo `super`, sdělující, že generický parametr musí být předkem oné třídy.

V následujícím příkladu je uvedena třída `ListINamed`, která obsahuje generický parametr `E` s omezením, že musí dědit od třídy `INamed` a celá třída je potomkem třídy `ArrayList`.

```
public class ListINamed<E extends INamed> extends ArrayList<E> {
    //Tělo třídy
}
```

- C# na rozdíl od toho definuje omezení generických parametrů pomocí klíčového slova `where` až na konci deklarace třídy, interfejsu, delegátu nebo metody. Generický parametr je od seznamu omezení oddělení dvojtečkou a argumenty v seznamu omezení jsou od sebe odděleny čárkou.

Pokud v C# definujeme omezení generického parametru, můžeme si vybrat z následujících možností:

- Třídu nebo interfejs, od kterého bude parametr dědit
- Specifikovat, že daný parametr bude hodnotový typ přidáním klíčového slova `struct`
- Specifikovat, že daný parametr bude referenční typ přidáním klíčového slova `class`
- Specifikovat, že typ bude mít veřejný konstruktor přidáním klíčového slova `new()`

V následujícím příkladu tedy definujeme stejný příklad jako výše v jazyce C#:

```
public class ListINamed<E>: List<E> where E: INamed
{
    //Tělo třídy
}
```

10.1.5 Zástupné znaky / Žolíky

Zástupné znaky, popř. žolíky (angl. wildcards), pomáhají v Javě implementovat parametry metod, obsahující generické datové typy, u kterých při psaní kódu ještě přesně nevíme, co budou obsahovat za generické parametry. Například pokud by existovala kolekce instancí třídy `INamed` a metoda by měla vypsat názvy těchto instancí, provedlo by se to například pomocí následujícího kódu:

```
public void colINamedToString(Collection<INamed> col) {
    for (INamed e: col) {
        System.out.print (col.getName());
    }
}
```

Funkce této metody by při předání kolekce s instancemi třídy `INamed` fungovala naprosto správně. Ovšem problém by nastal už ve chvíli, kdyby byla metodě předána kolekce instancí třídy, která je potomkem třídy `INamed`. V tu chvíli by Java vyhodila výjimku, protože jsme metodě předali parametr jiného typu, než očekávala. Tenhle problém umí vyřešit právě výše zmíněné zástupné znaky.

Při tvorbě metod s použitím zástupných znaků jsou na výběr dvě možnosti. Unbounded, neboli nespoutané a bounded, neboli spoutané zástupné znaky.

Při použití nespoutaných zástupných znaků využijeme místo definování generického parametru znak `?`. V tu chvíli je možné do parametru níže zmíněné metody zadat kolekci jakéhokoliv typu a on ji vezme.

```
public static String colINamedToString(Collection<?> collection) {
    //Tělo metody
}
```

Problém by ale nastal, kdyby se tato metoda použila například u metody `getName()`, která byla zmíněna výše. V případě oné metody není možné použít nespoutaný zástupný znak, protože ne všechny kolekce obsahují instance třídy `INamed` a tedy ani její funkce. K tomuto slouží druhá možnost zástupných znaků, neboli ty spoutané.

Díky nim je možné pomocí klíčového slova `extends` definovat, že parametrem je kolekce která obsahuje pouze danou třídu a její potomky a pomocí klíčového slova super, že uvedená třída musí být potomkem dané třídy. Níže uvedený příklad popisuje upravenou metodu `getName()`, která kromě instancí třídy `INamed` akceptuje i instance jejich potomků.

```
public void colINamedToString(Collection<? extends INamed> collection) {
    //Tělo metody zůstává stejné jako výše
}
```

C# bohužel v současné době nemá ekvivalent pro zástupné znaky, ovšem díky generickým metodám se dá napsat ekvivalent těchto funkcí.

Generická metoda, která má jako parametr kolekci s jakoukoliv třídou se dá napsat v C# tímto způsobem:

```
public void ColINamedToString<T>(List<T> list) {}
```

A metoda, která má jako parametr kolekci, která obsahuje instance pouze uvedené třídy, nebo jejich potomků následujícím způsobem:

```
public void ColINamedToString<T>(List<T> list) where T: INamed {}
```

10.2 Rozdíly na platformách

10.2.1 Defaultní hodnoty

Pokud obsahuje generická třída vlastnost generického typu, ke které přiřazuje programátor jeho defaultní hodnotu, v Javě , protože v ní není možné použít jako generický parametr primitivní typ, se jako defaultní hodnota použije hodnota `null`, ne tak v C#. V něm se k přiřazení defaultní hodnoty používá klíčové slovo `default`. Je to kvůli tomu, že primitivní proměnné nepoužívají `null` jako výchozí hodnotu a proto by došlo k chybě, kdyby se někdo snažil do proměnné například typu `int` přiřadit hodnotu `null`. Toto se dá obejít pomocí default, kterému přiřadíme do závorky generický parametr.

```
class GenerickaTrida<T>
{
    T vlastnost = default(T);
}
```

10.2.2 Implementace

Java

Generické datové typy v Javě byly napsány tím způsobem, aby se nemusela měnit struktura class souborů a díky tomu mohl kód, který používal generické datové typy fungovat i na platformách Java 4.0 a starších. Z tohoto důvodu zůstaly generické datové typy v Javě záležitost kompilátoru založená na mazání typů.

Z toho vyplývá několik následujících kompilačních vlastností:

- Generické typy (třídy a interfejsy) si ponechávají svá syrová jména, takže není možné mít v jednom balíčku generickou třídu a negenerickou třídu se stejným názvem jako například `Test<T>` a `Test` protože to je díky syrovým typům stejný typ.
- Všechny instance generických datových typů jsou zkompilovány jako jejích odpovídající syrové typy. Např. z instance třídy `List<String>` se stane instance třídy `List`

- Všechny instance generických datových typů jsou převedeny na instance jejich nejbližších odpovídajících typů:
    - Pokud generický parametr obsahuje klíčové slovo `extends`, převede se parametr na instanci onoho datového typu, od kterého dědí
    - Pokud ne, převede se na instanci typu `Object`, a to včetně parametrů obsahujících klíčové slovo `super`
- Generické metody stejně jako třídy si ponechávají svá syrová jména a proto není možné mít v jedné třídě generickou a negenerickou metodu stejného jména, parametrů a návratového typu
- Přetypování za běhu jsou vkládány již kompilátorem, aby nemohlo dojít k tomu, že přetypovaná metoda nebude tím, co očekáváme. Tímto vlastně kompilátor pomáhá s přetypováním proměnné a generické typy mají výhodu nad negenerickými typy v tom, že přetypovávání je rychlejší

```
//Příklad: Tato generická třída:
class GenericClass<T, U extends Number> {
    T t;
    U u;

    public T tMethod(List<T> list) {
        //Kód metody
    }

    public U uMethod(List<U> list) {
        //Kód metody
    }
}
//Bude po překladu kompilátorem vypadat následovně:
class GenericClass {
    Object t;
    Number u;

    public Object tMethod(List list) {
        //Kód metody
    }

    public Number uMethod(List list) {
        //Kód metody
    }
}
```

C#

Na rozdíl od Javy se toho Microsoft tolik nebojí a proto přidává do intermediate language řadu nových vlastností pro podporu generických datových typů. I tak ale spoustu generických .NETových typů implementuje starší negenerické rozhraní pro podporu starších programů.

Díky rozšíření IL je možné, že kód každé generické třídy nebo generického interfejsu je generován pouze jednou a následně je použit pro všechny referenční typy. Toho je dosaženo díky tomu, že všechny referenční typy jako lokální proměnné a vlastnosti třídy jsou stejné a tomu, že generický kód má rozdílné konvence volání, při kterém jsou do metod přidány implicitní argumenty dovolující provádění operací s generickými typy přímo při běhu programu. Díky tomu budou třídy typu `List<FirstClass>` a `List<SecondClass>` sdílet stejný JIT kód.

Jediný rozdíl nastává pro primitivní typy, u kterých je nutné vytvořit při překladu vždy nový JIT kód. Díky tomu třídy `List<int>` a `List<short>` sdílet stejný kód nebudou.

Díky způsobu implementace generických tříd v IL zde však vzniká ještě jeden zajímavý rozdíl a to u statických proměnných generických tříd. V Javě, vzhledem k tomu, že je kompilátorem kód převeden na syrové typy, jsou následně statické proměnné stejné pro všechny instance třídy s generickým datovým typem. V jazyce C# však toto neplatí.

Pokud vytvoříme instanci generického typu s různými generickými parametry, tyto instance nesdílí stejné statické proměnné, ale mají každý svou. Výstup proměnné `Count` po provedení metody `create` v následujícím příkladu se tedy bude v Javě a C# lišit. Zatímco v Javě její výstup bude 3. V C# bude hodnota vlastnosti `GenericClass<Class1>.Count` roven 2 zatímco hodnota vlastnosti `GenericClass<Class2>.Count` bude roven 1.

```
public void Create()
{
    GenericClass<Class1> tr1 = new GenericClass<Class1>();
    tr1 = new GenericClass<Class1>();
    GenericClass<Class2> tr2 = new GenericClass<Class2>();
}

public class GenericClass<T>
{

    public static int Count = 0;

    public GenericClass()
    {
        Count++;
    }
}
```

Dalším rozdílem je, že při generování IL kódu jsou generické třídy pojmenovány vlastním názvem třídy, následovaným znakem '`' a dále počtem generických parametrů. To například znamená že `List<T>` bude mít v IL název `List`1`. Díky tomu také jazyky na platformě .NET povolují mít v jednom jmenném prostoru více tříd se stejným názvem a různým počtem generických parametrů. Například třídy `Test`, `Test<T>` a `Test<T,U>` by mohly být ve stejném balíčku díky tomu, že do IL budou přeloženy jako `Test`, `Test`1` a `Test`2`. Stejné možnosti nabízí i metody.

10.2.3 Generické datové typy při běhu programu

Podpora generických datových typů při běhu aplikace se mezi Javou a C# výrazně liší. Je to způsobeno hlavně tím, že generické datové typy v Javě jsou kompletně založeny na kompilaci kódu a tedy při běhu programu o nich není možné zjistit žádné informace. Díky tomu není možné s generickými datovými typy v Javě provádět následující věci:

- tvořit instance generických parametrů
- tvořit pole generických parametrů
- získat `class` objekt generického parametru
- užití `instanceof` na generický parametr.

Na rozdíl od toho C# díky své podpoře generických datových typů v IL žádné z těchto problémů nemá. Pouze pro vytvoření nové instance generického typu je nutné použít omezení `new()` při jeho deklaraci. Dále pak už není žádný problém ve vytvoření nové instance typu generického parametru nebo i jeho pole.

Pokud je však opravdu nutné v Javě pole s generickými datovými typy za běhu programu vytvořit, je to možné za použití třídy `java.lang.reflect.Array` a `java.lang.Class<T>`. Pak je možné pomocí následujícího příkladu pole s generickými parametry vytvořit.

```
static <T> T[] createArray(Class<T> c, int size) {
    return (T[])java.lang.reflect.Array.newInstance(c, size);
}
```

10.2.4 Kovariance a kontravariance

Kovariance a kontravariance jsou termíny, které znamenají schopnost používat více či méně specifické datové typy, než je ten, který programátor defaultně specifikoval. Tuto schopnost nepodporují pouze generické datové typy ale také například pole v obou výše zmíněných jazycích.

Kovariance znamená možnost přetypování potomka třídy zpět na rodičovskou třídu. Následující příklad v jazyce C# ukazuje, jak přesně kovariance funguje, na předpokladu že máme interface INamed a třídu `AFile`, z nichž `AFile` implementuje `INamed`.

```
public void Main(String[] args)
{
    IEnumerable<AFile> fileList = new List<AFile>();
    IEnumerable<INamed> namedList = fileList;
}
```

Díky to mu že generický parametr interface `IEnumerable` v C# je kovariantní, je možné přetypovat instanci `IEnumerable` s generickým parametrem třídy `AFile` do instance stejné třídy ovšem s generickým parametrem třídy `INamed`, která je jejím předkem. To, že je generický parametr kovariantní poznáme tím způsobem, že u generického parametru je klíčové slovo `out`. Interface `IEnumerable` je tedy v C# nadefinován jako `IEnumerable<out T>`.

Kovariance však není pouze výhodou C#, i Java ji používá. Jedná se například o pole definovaná jak v Javě tak v C#. Díky tomu je možné například přiřadit proměnné, která očekává pole typu `Object` (`object` v C#), instanci pole typu `String` (`string` v C#). Toto pole následně sice vypadá jako pole typu `Object` (`object`) ovšem není možné do něj přiřadit například objekt typu `Integer` (`Int32`), i když také dědí od třídy `Object` (`object`). Pole si totiž pořád drží informace o tom, že v něm jsou uloženy instance typu `String` (`string`) a pokud je do něho vkládána instance jiného typu, vyhodí `ArrayStoreException` (`ArrayTypeMismatchException` v C#)

Kontravariance na druhou stranu znamená přetypování předka dále na svého potomka. V C# je toho příkladem například delegát `Action`. V následujícím příkladu je díky kovarianci možné převést delegát s generickým parametrem `INamed`, na delegát s generickým parametrem `AFile`.

```
public void Main(String args)
{
    Action<INamed> iNamedAction = named => {};
    Action<AFile> aFileAction = iNamedAction;
}
```

Díky kontravarianci generického parametru tohoto delegátu je výše přetypovaný kód naprosto v pořádku. To že má delegát kontravariantní parametr, poznáme tím způsobem, že je před ním uvedené klíčové slovo `in`. Delegát `Action` je tedy v C# definován jako `Action<in T>`.

11 Výčtové typy

Výčtové typy[27] (angl. enum) jsou velmi používanou funkcionalitou v programování a v současné době je implementují Java i C#. V obou programovacích jazycích se výčtové typy označují klíčovým slovem `enum`. Je to datový typ, tvořený množinou pojmenovaných hodnot, složených z identifikátoru a hodnoty. Hodnotou defaultně bývá celé číslo.

Rozdíly mezi nimi jsou však značné. Jednak v tom, že v C# jsou výčtové typy podporovány už od jeho první verze a dokonce již od beta verze od roku 2000. V Javě byla tato funkcionalita přidána až od verze JavaSE 5.0, ale je zde zase vidět určitý pokrok a i to, že se od implementace výčtových typů v C# něco přiučili a vyhnuli se tím spoustě chyb a nedokonalostí.

C#

Výčtové typy v C# jsou pouze hodnotové typy, postavené na určitých číselných typech, standardně na typu `int`, mohou však dědit i od typů byte, `short` a `long`. Každá hodnota ve výčtovém typu v C# má svou číselnou hodnotu která, pokud není určeno jinak, začíná od 0 a pokračuje po 1 dále pro každou další hodnotu.

```
public enum TypeOfScenario
{
    ScHappy,
    ScMistakes,
    ScGeneral,
    ScDemo
}
```

Výše deklarovaný výčtový typ v C# obsahuje typy scénářů adventury. V tomto případě má `TypeOfScenario.ScHappy` hodnotu 0 a `TypeOfScenario.ScDemo` hodnotu 3. Pokud by však došlo na to, že by bylo nutné deklarovat vlastní číselné hodnoty pro jednotlivé typy scénáře, je důležité si zapamatovat, že C# dovoluje deklarovat dvě stejné číselné hodnoty pro dvě různé hodnoty jednoho výčtového typu a tyto hodnoty při následném porovnání vypadají jako dvě stejné hodnoty. A druhá věc je, že číslování stále pokračuje způsobem, že se další položka má o 1 více než předchozí položka. Pokud by tedy v našem příkladu měla hodnota `TypeOfScenario.ScHappy` číselnou hodnotu 1 a `TypeOfScenario.ScGeneral` také číselnou hodnotu 1. Pokud bychom nedeklarovali dále, došlo by k tomu, že `TypeOfScenario.ScMistakes` i `TypeOfScenario.ScDemo` by obě měly číselnou hodnotu 2.

[27] Enum - Comparsion of Java and .NET. *CodeProject* [online]. [cit. 2016-04-12]. Dostupné z: http://www.codeproject.com/Articles/688300/Enum-Comparison-of-Java-and-NET

Java

U Javy před verzí 5.0 bylo nejlepší možností jak vytvořit výčtový typ[28] vytvořit třídu, ve které byly definovány veřejné finální atributy například typu `int`, a tato třída měla privátní konstruktor. Výsledek toho by pak mohl vypadat jako na následujícím příkladu.

```
public class TypeOfScenario {
    public static final int ScHappy = 0;
    public static final int ScMistakes = 1;
    ...
    private Week() {}
}
```

Na tenhle problém reagovala Java ve verzi 5.0 vydáním podpory pro vlastní výčtové typy. Které se však od těch v C# hodně lišily. Hlavní důvod je v tom, že na rozdíl od C#, výčtové typy v Javě jsou referenční typy a jako s takovými se s nimi dá i zacházet.

Je možné vytvořit pro ně vlastní konstruktor, vytvořit ve výčtovém typu novou metodu a celkově s nimi zacházet jako se třídami, až na to, že musí mít privátní konstruktor. Všechny výčtové typy v Javě dědí od třídy `java.lang.Enum`, která následně dědí od třídy `java.lang.Object`, jako všechny třídy v Javě. Díky tomu je také možné přepisovat metody jako `toString` a další, které `java.lang.Enum` dědí od svého předka. Na rozdíl od ostatních tříd, dědících od třídy `Object` mají však výčtové typy další možnosti. V Javě stejně jako v C# je možné používat výčtové typy v příkazu `switch` a dalších, nebo je třeba konvertovat z a do typu `String`.

[28] *Java efektivně: 57 zásad softwarového experta*. 1. vyd. Praha: Grada, 2002, s. 94-95. Moderní programování. ISBN 80-247-0416-1.

12 Dynamické a implicitní typování

Platforma .NET v sobě ve verzi 4.0 přinesla novinku ve formě dynamického typování a tuto vlastnost implementoval i C# s pomocí typu `dynamic`[29]. Pokud bude při inicializaci proměnné použito místo jejího typu nebo klíčového slova `var`, které C# také nabízí, klíčové slovo `dynamic`, bude inicializována klasická statická proměnná, ovšem C# při kompilaci neprovede kontrolu vlastností a metod, které jsou z této instance volány. Ta bude provedena následně až při běhu programu a pokud nebudou vlastnosti nebo metody nalezeny, program vyhodí chybu.

```
class SomeClass
{
    public void ExistingMethod() { /* Tělo metody */ }
}
dynamic someInstance = new SomeClass();
//I když následující metoda neexistuje, kompilátor nevyhodí žádnou chybu.
//Ta bude vyhozena až následně v programu, pokud ji do té doby nikdo nevytvoří.
someInstance.NonExistingMethod();
```

12.1 Klíčové slovo var

V odstavci výše bylo zmíněno klíčové slovo `var`[30]. Toto klíčové slovo bylo uvedeno v jazyce C# ve verzi 3.0 a povoluje implicitní typování proměnných. Což znamená, že při vytváření lokálních proměnných je není nutné explicitně typovat ale místo jejich typu se může dosadit klíčové slovo `var` a kompilátor si při kompilaci následně sám doplní správný typ proměnné místo tohoto klíčového slova.

```
var myNumber = 1;
```

Toto klíčové slovo je ve spoustě případů pouze dobrovolné a programátor ho nemusí používat. Ovšem při inicializaci anonymních typů, je toto klíčové slovo použít nutné. Anonymní typy[31] je způsob v C#, jak zapouzdřit sadu atributů do jednoho objektu, bez nutnosti explicitně definovat jejich typ. Typ každé proměnné je následně definován kompilátorem a není dostupný na úrovni zdrojového kódu, kvůli tomu je nutné použít implicitní datové

[29] Použití typu dynamic. *Microsoft Developer Network* [online]. [cit. 2016-04-25]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/dd264736.aspx
[30] Implicitně typované lokální proměnné. *Microsoft Developer Network* [online]. [cit. 2016-04-25]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb384061.aspx
[31] Anonymní typy. *Microsoft Developer Network* [online]. [cit. 2016-04-25]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb397696.aspx

typy. Tento objekt je definován pomocí klíčového slova new, za kterým následuje inicializace vytvářeného objektu. Tyto objekty jsou určeny pouze pro čtení.

```
var obj = new { Name = "Ondřej Hartman", BirthYear = 1994 };
```

12.2 Co na to Java

Java v jazyce vytváření dynamických datových typů nepovoluje. V platformě tato implementace existuje, ale v jazyce podporována není. Proto, pokud by někdo chtěl pracovat s dynamickým typováním na platformě Java, musel by si vybrat jiný programovací jazyk, který nad Platformou Java funguje, jako například JRuby, Jython nebo Groovy.

12.2.1 Diamantové operátory

I když dynamické typování Java neobsahuje, implicitní typování v sobě implementováno má. Na rozdíl od C# ho však nemá na levé, ale na pravé straně deklarace. Při deklaraci generického datového typu není na pravé straně nutné uvádět generický parametr. Tímto způsobem Java implementuje implicitní typování generických datových typů, kterému se říká diamantový operátor.

```
private static final Collection<AFile> ALL_FILES = new ArrayList<>();
```

13 Funkční objekty

Vzhledem k tomu, že oba zde probírané jazyky jsou objektově-orientované jazyky, znamená to, že cokoliv v nich je objekt. To ale znamená, že pokud chceme napsat nějakou funkci, musíme kolem ní vytvořit třídu, funkce bez třídy zkrátka vytvořit nejde. Pro zjednodušení psaní těchto objektů si každý z jazyků našel svou cestu, oba však zvolily implementaci lambda výrazů.

13.1 C#

13.1.1 Delegáty

Delegáty[32] (angl. delegate) představují odkazy na metody s určitým seznamem parametrů a určitým návratovým typem. Při inicializaci delegátu je možné jej spojit s jakoukoliv metodou, která má stejné parametry a návratovou hodnotu jako onen delegát. Pak je možné skrze tento delegát onu metodu volat.

```
public delegate int PerformCalculation(int x, int y);

PerformCalculation count = (x, y) => x + y;

int result = count(1, 2);
```

13.1.2 Anonymní metody

Do C# verze 2.0, byla jediná možnost jak deklarovat delegát použít pojmenované metody. C# 2.0 přišel s anonymními metodami[33], tedy možností, vytvořit metodu i bez jejího pojmenování.

```
this.FirstOrDefault(delegate(E e)
{
    return e.GetName().Equals(name, StringComparison.CurrentCultureIgnoreCase);
});
```

[32] Delegates (C# Programming Guide). *Microsoft Developer Network* [online]. [cit. 2016-04-29]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms173171.aspx
[33] Anonymous Methods (C# Programming Guide). *Microsoft Developer Network* [online]. [cit. 2016-05-01]. Dostupné z: https://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx

13.1.3 Lambda výrazy

Následně ve verzi C# 3.0 přišel Microsoft s možností zapisovat metody ještě jednodušší cestou a kvůli tomu zavedl lambda výrazy. Díky tomu již nebylo nutné psát celé deklarace anonymních metod, ale pouze jejich parametry a tělo. Tento způsob nakonec C# zavedl jako preferovaný způsob psaní anonymních metod.

```
this.FirstOrDefault(e =>
    e.GetName().Equals(name, StringComparison.CurrentCultureIgnoreCase));
```

13.2 Java

13.2.1 Anonymní třídy

Anonymní třídy[34] (anonymous classes) povolují programátorovi zároveň deklarovat třídu a inicializovat její instanci. Při inicializaci již existující třídy se za klíčové slovo `new` a volání konstruktoru třídy přidají složené závorky, ve kterých se deklarují datové členy třídy, které chceme přebít, nebo vytvořit nové. Ty se následně mohou používat dále v kódu pouze pro jednu danou instanci třídy, pro kterou byly tyto položky nadefinovány.

Java již před verzí 8 používala anonymní třídy ve spoustě případů, například pro akce tlačítek v grafickém rozhraní nebo při vytváření nových vláken. Základem těchto anonymních tříd byl interfejs s jednou abstraktní metodou. Tyto interfejsy byly nazývány SAM Interfaces (Single Abstract Method).

13.2.2 Lambda výrazy a funkční interfejsy

Následně přišla Java 8 s deklaraci lambda výrazů a tyto interfejsy převzala pro jejich používání a dala jim jméno Funkční interfejsy (Functional interfaces). Také k tomu Java přišla s anotací `@FunctionalInterface`, která zajišťuje, že kompilátor zjistí, zda metoda obsahuje opravdu pouze jednu abstraktní metodu, a je tedy možné daný interfejs použít jako funkční interfejs.

[34] Anonymous Classes (The Java Tutorial). *Oracle Java Documentation* [online]. [cit. 2016-05-01]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

```
public Optional<E> getNamed(String name){
    this.stream().filter(new Predicate<E>() {
        @Override
        public boolean test(E e) {
            return e.getName().equalsIgnoreCase(name);
        }
    }).findFirst();
}
```

Díky těmto interfejsům Java tedy jasně ví, že lze implementovat pouze jednu metodu a tuto metodu následně zastoupí lambda výraz. Tímto je možné pomocí lambda výrazu implementovat funkční interfejs.

```
public Optional<E> getNamed(String name){
    this.stream()
    .filter(e -> e.getName().equalsIgnoreCase(name)).findFirst();
}
```

Pro vytvoření vlastního funkčního interfejsu, stačí vytvořit interfejs pouze s jednou abstraktní metodou. Defaultních metod a statických metod může mít interfejs, kolik chce, jde pouze o to, aby měl pouze jednu abstraktní metodu, kterou lambda výraz následně zastoupí.

```
@FunctionalInterface
public interface PerformCalculation {
    public int calculate (int x, int y);
}
```

14 C# LINQ vs. Java 8 Streams API

LINQ[35] je sada funkcí pro platformu .NET, která byla představena v .NET verze 3.0 a rozšiřuje možnosti dotazování jazyků, které fungují na této platformě jako jsou například jazyky C# a Visual Basic. *LINQ* obsahuje spousty možných funkcí, mezi které patří filtrování a práce s kolekcemi a jejich daty a tímto se bude tato kapitola zabývat.

Oracle v prozatím poslední verzi Java 8.0 představil *Streams API*, které poskytlo podobné funkce pro práci s kolekcemi jako má knihovna LINQ. Způsob použití se ovšem mezi oběma jazyky významně liší. V této kapitole bude zobrazeno několik příkladů například filtrace nebo řazení, pomocí kterých budou popsány rozdíly mezi knihovnou *LINQ* v jazyce C# a *Streams API* v jazyce Java.

Příklady pro následující kapitolu budou použity z článku[36] Edwina Dalorza o porovnání knihovny LINQ a *Streams API* Javy 8.0.

14.1 Filtrace

Jednoduchá filtrace je jednou z nejjednodušších operací, které se dají pomocí výše zmíněných funkcí provést. V obou případech se jedná pouze o zapsání jedné podmínky a v obou případech se dají použít lambda výrazy, které kód výrazně zjednodušují.

```
//Java
String[] names = {"Sam","Pamela", "Dave", "Pascal", "Erik"};
List<String> filtered = Arrays.stream(names)
    .filter(name -> name.contains("am"))
    .collect(Collectors.toList());

//C#
string[] names = { "Sam", "Pamela", "Dave", "Pascal", "Erik" };
List<string> filteredNames = names.Where(c => c.Contains("am")).ToList();
```

14.2 Mapování

Pomocí funkce map ve *Stream API* Javy můžeme na každý z prvků kolekce aplikovat nějakou funkci a s tímto výsledkem následně dále pracovat. V jazyce C# se dá následující

[35] LINQ (Language-Integrated Query). *Microsoft Developer Network* [online]. [cit. 2016-04-18]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb397926.aspx

[36] Java Streams Preview vs .NET High-Order Programming with LINQ. *Informatech CR Blog* [online]. [cit. 2016-04-18]. Dostupné z: https://blog.informatech.cr/2013/03/24/java-streams-preview-vs-net-linq/

funkce nahradit funkcí `Select` následovanou funkcí `ToList`, která nám vrátí kolekci pozměněných hodnot, se kterou je následně možné dále pracovat. Jak ostatně ukazuje následující příklad, který vypíše do konzole všechna jména v kolekci a ještě před nimi zobrazí slovo „Hello“.

```
Arrays.stream(names).map(name -> "Hello " + name).forEach(System.out::println);

names.Select(name => "Hello " + name).ToList().ForEach(Console.WriteLine);
```

Na příkladu výše je možné vidět, že pokud existuje metoda, která požaduje stejné parametry, jako předává metoda, kterou používáme, je možné použít ji místo lambda výrazu a v obou případech se parametry přicházející z metod dosadí do parametrů metody předané.

14.3 Řazení

Řazení je jednou z funkcí, které je možné najít téměř v každém programu, knihovna *LINQ* a *Stream API* k tomu přinášejí pomůcku, která tuto funkci velmi zjednodušuje a hlavně také zkracuje. Například řazení podle délky slov již není nutné psát pomocí několika cyklů, ale stačí na to pár řádků kódu.

```
Arrays.stream(names).sorted(Comparator.comparingInt(String::length))
    .toArray(String[]::new);

names.OrderBy(name => name.Length).ToArray();
```

V obou jazycích je také možné vytvořit řazení dle více kritérií, v C# k tomu slouží metoda `ThenBy`, která by se napojila mezi `OrderBy` a `ToArray`. V Javě by to bylo trochu složitější, nejdříve by se musela vytvořit nová instance třídy `Comparator`, které by bylo následně pomocí funkcí `comparing` a `thenComparing` zadáno co se porovnává a tato instance by poté byla předána do metody `sorted`, která byla použita v příkladu výše.

14.4 Seskupování hodnot

Každá hodnota v polích má určité vlastnosti, podle nichž je možné ji následně dělit do mnoha různých skupin. Oba jazyky obsahují funkce, které toto seskupování povolují, a proto je možné vytvořit například z pole názvů výše skupiny jmen, které obsahují jména, jež jsou stejně dlouhá.

```
Map<Integer, List<String>> groups = Arrays.stream(names)
    .collect(Collectors.groupingBy(String::length));
```

```
var groups = names.groupBy(name => name.Length);
```

14.5 Hodnotící operátory

Jako další příklad možností zde budou uvedeny metody, které dokáží podle určitého parametru profiltrovat celé pole a řeknou nám, zda všechny nebo některé hodnoty v kolekci vyhovují zadanému parametru.

Nejprve budou uvedeny metody pro zjišťování, zda parametru vyhovují všechny zadané elementy.

```
Arrays.stream(names).allMatch(name -> name.length() == 5);
```

```
names.All(name => name.Length == 5);
```

Je možné si všimnout, že tyto dva kódy se od sebe vzájemně moc neliší, nebo alespoň ne tolik, jako v několika předchozích příkladech. V případě, že bychom chtěli zjistit, zda alespoň nějaké prvky v poli námi zadanému parametru vyhovují, zaměnili bychom v jazyce C# metodu `All` za metodu `Any`, se stejnými parametry a v jazyce Java metodu `allMatch` za metodu `anyMatch`, také se stejnými parametry.

14.6 Paralelní výpočty

Jako poslední příklad zde bude uvedena možnost distribuce výpočtů mezi jednotlivá vlákna procesoru. V Javě se pro to dají použít funkce *Streams API* `parallelStream`, nebo `parallel` které vrátí paralelní datovod, se kterým se následně pracuje. C# k paralelním výpočtům používá paralelní implementaci knihovny LINQ tzv. PLINQ, a její výchozí funkce `AsParallel`, která převede klasický LINQ dotaz na paralelní.

```
Arrays.stream(names).parallel().allMatch(name -> name.length() == 5);
```

```
Names.AsParallel().All(name => name.Length == 5);
```

15 Operátory

Operátor[37] je znak, nebo skupina znaků, která označující operaci, která se má na daném místě provést. Například při sčítání dvou čísel použijeme operátor +. Java i C# mají spoustu operátorů stejných, zejména pokud se jedná o matematické operátory jako sčítání (+), odčítání (-), dělení (/), násobení (*) nebo modulo (%), unární operátory jako například inkrementace (++) nebo dekrementace (--), nebo porovnávací a podmínkové operátory jako například rovná se (==), nerovná se (!=), větší než (>), menší než (<), a zároveň (&&).

Další skupinou by se daly nazvat operátory, které jsou sice použity v obou jazycích, ale zapisují se jinak. Příkladem těchto operátorů by mohl být například operátor porovnání typů instancí, pro které se Javě používá klíčové slovo `instanceof`, kdežto v C# je možné použít například klíčové slovo `typeof`, pokud chceme zjistit, zda se jedná o instanci přesně daného typu, nebo klíčové slovo `is`, pokud chceme zjistit, zda se jedná o instanci onoho typu, nebo jeho potomka.

```
//Porovnání zda se jedná o třídu nebo potomka třídy ve třídě AFile
public boolean equals(Object obj) {
    return obj instanceof AFile;
}
```

```
//Porovnání zda se jedná přímo o instanci třídy AFile
public boolean Equals(object obj)
{
    return obj.getType() == typeof(AFile);
}
//Porovnání zda je o třídu nebo jejího potomka
public boolean Equals(object obj)
{
    return obj is AFile;
}
```

Dalším podobným operátorem je operátor přetypování. Ten je možné v Javě i C# napsat stejně a to pomocí závorek, uvnitř kterých je uvedena třída, na kterou se má daná instance přetypovat.

```
double dNumber = 1.5;
int iNumber = (double) dNumber;
```

Vzhledem k tomu, že `iNumber` je proměnná typu `int`, bude hodnota za tečkou oříznutá a výsledná hodnota této proměnné bude 1.

[37] Operace a operátory. PECINOVSKÝ, Rudolf. *Java 7: učebnice objektové architektury pro začátečníky*. Vyd. 1. Praha: Grada, 2012, s. 312. Knihovna programátora (Grada). ISBN 978-80-247-3665-5.

Další možností přetypování v C# je pomocí klíčového slova `as`. Tato možnost je však možná pouze pro přetypování instancí rodičovských třídy dále na své potomky. Pokud toto přetypování není možné, nastaví proměnné, do které je přetypovaná reference přiřazena hodnotu `null`.

```
ACommand cmd = new CommandMove();
CommandMove hcmd = cmd as CommandMove;
```

Toto přetypování funguje na principu, že se nejprve zjistí, zda je možné hodnotu přetypovat a pokud se zjistí, že ano, nastaví ji příslušný typ, pokud to možné není, nastaví ji hodnotu `null`. V C# by se tento kód dal do podmínky přepsat následovně:

```
ACommand cmd = new CommandMove();
CommandMove hcmd = cmd is CommandMove? (CommandMove) cmd: null;
```

Posledním zde zmíněným operátorem bude ten, který pravděpodobně zná každý programátor, který se někdy setkal s knihovnou LINQ v C# nebo Javou ve verzi 8.0. Jedná se o lambda výrazy.

Lambda výrazy v Javě jsou zastoupeny operátorem `-> a` je možné použít například k inicializaci instancí funkčních interfejsů. V následujícím příkladu je uvedeno vyhledání elementu podle názvu s pomocí *Streams API* a lambda výrazu.

```
this.stream().filter(e -> e.getName().equalsIgnoreCase(name)).findFirst();
```

V C# jsou lambda výrazy zastoupeny operátorem `=>` a používají se například pro vytvoření delegátů, jako je například delegát `Func<T, TResult>` který přijímá jako parametr hodnotu `T` a vrací hodnotu `TResult`, v následujícím příkladu tedy příjme hodnotu `INamed` a vrátí booleovskou hodnotu `true`, pokud se našel element se správným názvem.

```
this.FirstOrDefault(e => e.GetName().Equals(name,
    StringComparison.CurrentCultureIgnoreCase));
```

15.1 Přetěžování operátorů

Přetěžování operátorů[38] je vlastnost, která dovoluje vytvořit si vlastní možnost, jak má operátor na daném místě fungovat. Tuto vlastnost v současné době podporuje ze dvou zmiňovaných jazyků pouze C#. Java tuto vlastnost nepodporuje.

V C# je možné pro třídu přetížit operátor, aby pracoval způsobem, který po něm daný programátor potřebuje. Při přetížení operátoru se ve třídě použije klíčové slovo `operator`, za

[38] Poznáváme C# a Microsoft .NET. *Živě.cz* [online]. [cit. 2016-04-17]. Dostupné z: http://www.zive.cz/clanky/poznavame-c-a-microsoftnet--8-dil/sc-3-a-122182/

kterým následuje operátor, který má být přetížen. Funkce, která se má při přetížení použít musí být označená jako `public static`. V následujícím příkladu je uvedeno přetížení operátoru ekvivalence.

```
public static bool operator == (Field field1, Field field2)
{
    return field1.getX() == field2.getX() && field1.getY() == field2.getY();
}
```

15.2 Operátory vyskytující se pouze v jednom z jazyků

V obou jazycích se vyskytuje několik operátorů, které v tom druhém nejsou implementovány a je nutné je v druhém jazyce napsat jinak. V téhle části budou tyto operátory zmíněny, a pokud to bude nutné, bude zde také ukázán způsob, jakým je možné je napsat ve druhém jazyce.

Java obsahuje oproti C# pouze jeden operátor navíc a tím je operátor bitového posunu bez znaménka zapisovaný jako `>>>`.

V C# je těchto operátorů o něco více. Za zmínku stojí například operátory `checked` a `unchecked`. Kde `unchecked` zařídí, že není kontrolováno přetečení číselných proměnných při jejich konverzích a operacích s nimi, `checked` naopak zařídí, že přetečení kontrolováno bude.

Dalším pro mnohé jistě známým operátorem je operátor `"?."`. Tento operátor nejprve zjistí, zda volaná instance není `null` a pokud ne, zavolá požadovanou funkci.

```
AFile.GetFile("file.txt")?.Execute();
```

V Javě máme pro toto zjištění dvě možnosti. Jedna možnost je nejprve zjistit, zda uvedená instance není `null` a teprve poté by bylo možné požadovanou funkci zavolat.

```
if (file != null) {
    file.Execute();
}
```

Druhou možností by bylo použití nové třídy, kterou Java 8 zavádí a to třídy `Optional<T>`. Tuto možnost implementovala Java 8 a umožňuje lepší práci s metodami, u kterých je možné že vrátí `null`. Tato třída poskytuje spousty funkcí, pomocí kterých je možné pracovat s hodnotami, které mohou vrátit `null`. V tomto případě by se použila metoda `ifPresent`.

```
AFile.GetFile("file.txt").ifPresent(file -> file.execute());
```

Posledním zmíněným operátorem zde bude operátor `??`, který obdrží dvě proměnné a zjistí, zda první proměnná není `null`. Pokud ne, vrátí první proměnnou a pokud ano, vrátí druhou.

```
var file = AFile.GetFile("file.txt") ?? file2;
```

V Javě by bylo nutné znovu použít příkaz `if`. Nebo již výše zmíněnou třídu `Optional`. V tomto případě by byla použita její metoda `orElse`.

```
AFile file = AFile.GetFile("file.txt").orElse(file2);
```

16 Regulární výrazy

Regulární výrazy[39] vznikly kvůli potřebě práce s textovými řetězci. Jedná se o textový řetězec složený z určitých znaků, které zastupují například čísla, písmena nebo jiné textové znaky. Úkolem těchto řetězců je poté vyhledat v textu určité části, které odpovídají právě danému regulárnímu výrazu a provést s ním určité operace.

Příkladem regulárního výrazu může být například následující regulární výraz pro zjednodušené zjištění, zda text odpovídá emailové adrese.

```
[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}
```

Tento regulární výraz nám říká, že daná emailová adresa se musí skládat z jednoho a více písmen, čísel, nebo tečky, podtržítka a pomlčky, následovanými zavináčem a dále znovu jedním nebo více znaky představujícími písmena, čísla tečky nebo pomlčky, následovanými tečkou a dvěma až čtyřmi znaky anglické abecedy. Neboli jménem následováním zavináčem a následovaným webovou adresou.

16.1 Java

Java pro regulární výrazy používá třídy `Pattern` a `Matcher`. Nejdříve se vytvoří instance třídy `Pattern`, které se předá regulární výraz.

```
Pattern pattern =
    Pattern.compile("[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}");
```

Následně se vytvoří instance třídy `Matcher` pomocí instance třídy `Pattern`, kterému se předá text, se kterým má zadaný regulární výraz pracovat.

```
Matcher matcher = pattern.matcher("mail@hofmanix.cz");
```

S instancí třídy `Matcher` můžeme následně provádět operace, jako jsou například vyhledávání v textu nebo nahrazování textu jiným textem.

```
If (matcher.find()) {
    System.out.println("E-mail is correct.");
}
```

[39] Regulární výrazy v Javě. *ITnetwork.cz* [online]. [cit. 2016-04-22]. Dostupné z: http://www.itnetwork.cz/java/pokrocile/tutorial-java-regularni-vyrazy/

16.2 C#

V C# se pro práci s regulárními výrazy používá třída `Regex`, které se nyní v konstruktoru předá regulární výraz, se kterým se bude pracovat.

```
Regex regex = new Regex("[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}");
```

Třída `Regex` následně obsahuje i metody pro práci s těmito regulárními výrazy. Například pro zjištění, zda text odpovídá regulárnímu výrazu, se použije funkce `IsMatch`, které se předá text, který se zjišťuje.

```
if (regex.IsMatch("mail@hofmanix.cz")
{
    Console.WriteLine("E-mail is correct.");
}
```

17 Anotace a atributy

17.1 Anotace

Dle knihy Java 7 od Ing. Pecinovského jsou Anotace speciální datové typy (formálně interfejsy) sloužící k zadávání dodatečných informací o deklarované či definované entitě – atributu, metodě či datovém typu. Zapisují se před anotovanou deklaraci či definici a označují se uvozujícím znakem @, za nímž následuje identifikátor dané anotace.

Příkladem nejznámějších anotací jsou anotace `@Deprecated`, pro označení metody, která by se již neměla používat a `@Override`, pro označení přebíjející metody.

```
@Override
public String getName() {
    return this.getClass().getName();
}
```

Tyto jednoduché anotace pouze označují metody, ovšem nemají žádné parametry. Další možností jsou tedy metody, které kromě svého identifikátoru obsahují i nějaké parametry.

```
@Author(
    name = "Ondřej Hartman",
    date = "4/27/2016"
)
```

Pokud anotace obsahuje pouze jednou možnou hodnotu, je možné při zadávání vynechat její název.

```
@SurpressWarnings(value = "unchecked")
@SurpressWarnings("unchecked")
```

Při deklaraci vlastní anotace se vytvoří interfejs, před který se přidá @. Tím se vytvoří nová anotace. Poté se v interfejsu definují metody, kde každá z nich představuje parametr interfejsu. Tuto anotaci je následně možné použít stejně jako každou jinou zadáním jejího identifikátoru a jejích parametrů.

```
@interface MyAnnotation {
    String stringValue();
    int integerValue();
    String defaultStringValue() default "Default value";
}
```

17.2 Atributy

C# tyto informace o třídách také implementuje, ovšem na rozdíl od Javy je nazývá atributy. Tyto atributy se deklarují nad konkrétním elementem, ke kterému tato informace patří a zapisuje se do hranatých závorek. K jednomu elementu je možné přidat i více atributů, ty se poté oddělují čárkami. Parametry se zadávají do závorek za každý atribut

```
[Obsolete]
public static class InputBox
{
...
}
```

Pro vytvoření vlastního atributu je nutné vytvořit třídu, která bude dědit od třídy `System.Attribute`. Název třídy by se měl dle konvencí skládat z názvu atributu, následovaného slovem `Attribute`, slovo `Attribute` následně není nutné při používání atributu používat. Například Atribut `Obsolete` použitý výše má název třídy `ObsoleteAttribute`.

Pomocí atributu se nad touto třídou specifikuje, pro které elementy se může daný atribut použít a uvnitř je nutné vytvořit konstruktor, který je následně volán a do něhož jsou přidány jeho parametry. Pro nastavení vlastností, které nejsou součástí konstruktoru, je možné použít pojmenované parametry.

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct)]
public class TestAttribute: Attribute
{

    public string Name;
    public string Number;
    public string Value;

    public TestAttribute(string name, int number)
    {
        Name = name;
        Number = number;
    }
}

[Test("Ondřej Hartman", 10, Value="value")]
public class AttrTest
{
}
```

18 Závěr

V této práci jsem se zabýval rozebíráním rozdílů mezi programovacími jazyky Java a C#. V práci jsou jasně vypsané rozdíly mezi oběma jazyky a je možné se podle ní naučit základním rozdílům pro naučení se programovat v jednom jazyce, pokud z toho druhého již znám alespoň základy.

V práci jsou uvedeny jednoduché a srozumitelné příklady, které jsou jasně rozděleny, a je možné jednoduše poznat, který příklad náleží kterému jazyku. Tyto příklady jsou součástí přiloženého projektu, kde jsou dva stejné programy napsané v obou jazycích a je možné je spolu průběžně porovnávat.

18.1 Dosažené cíle

- V práci byly rozebrány všechny rozdíly definované v cílech této práce
- Rozdíly byly podrobně rozebrány a následně prezentovány pomocí příkladů
- Většina těchto příkladů je součástí praktického příkladu přiloženého na CD vytvořeného v obou programovacích jazycích

Dle mého názoru jsem cíle práce celkově naplnil. Rozebral jsem všechny rozdíly, které byly v cílech definovány, a následně jsem je znázornil pomocí příkladů přiložených k textu.

Většina těchto příkladů pochází z praktické části mé bakalářské práce. Ty části, které se v praktické části nenacházejí, jsou nahrazeny příklady, které vhodně vysvětlují používání těchto částí kódu.

Terminologický slovník

Termín cz	Termín en	Význam [zdroj]
Třída	Class	Základní konstrukční prvek v OOP (viz kapitola 7 Třídy)
Interfejs	Interface	Kontejner pro skupinu funkcionalit, které může následně implementovat třída (viz kapitola 8 Interfejsy)
Výčtový typ	Enum	Datový typ, tvořený množinou pojemnovaných hodnot (viz kapitola 11 Výčtové typy)
Metoda	Method	Část programu, kterou je možné volat z různých částí kódu, a která provádí nějakou činnost (viz kapitola 9 Metody)
Přetížení	Overloading	Možnost vytvořit v jedné třídě více metod se stejným jménem ale jinými parametry a návratovou hodnotou (viz kapitola 9.5 Přetěžování metod)
Přebíjení	Overriding	Možnost vytvořit v potomkovi třídy metodu se stejným názvem, návratovou hodnotou a parametry jako má předek a tím přepsat stávající metodu (viz kapitola 9.4 Přebíjení metod)
Zakrývání metod	Method hiding	Zakrytí metody předka stejnou metodou v potomkovi (viz kapitola 9.4.2 Zakrývání metod)
Jmenný prostor	Namespace	Organizační jednotka sloužící ke zpřehlednění kódu programu (viz kapitola 6 Jmenné prostory a balíčky)
Balíček	Package	viz Jmenný prostor
Vnořený typ	Nested type	Statický datový typ nacházející se uvnitř jiného datového typu (viz kapitola 7.4 Vnořené datové typy)
Vnitřní třída	Inner class	Nestatická třída, nacházející se uvnitř jiné třídy (viz kapitola 7.4 Vnořené datové typy)
Generický datový typ	Generic date type	Generický datový typ je datový typ, přijímající generický parametr (viz kapitola 10 Generické datové typy)
Žolík	Wildcard	Zástupné znaky pomáhající implementovat parametry metod, u kterých není přesně jasné, jakého typu budou. (viz kapitola 10.1.5 Zástupné znaky / Žolíky)
Kovariance	Covariance	Možnost přetypování potomka třídy zpět na rodičovskou třídu (viz 10.2.4 Kovariance a kontravariance)

Kontravariance	Contravariance	Možnost přetypování rodičovské třídy na svého potomka (viz 10.2.4 Kovariance a kontravariance)
Dynamický typ	Dynamic type	Datový typ, u kterého není do doby než se zavolá jisté, zda opravdu obsahuje datový člen, který je volán. (viz kapitola 12 Dynamické a implicitní typování)
Explicitní typování	Explicit typing	Zapisování typu proměnné při deklaraci na obou stranách výrazu. (viz kapitola 12 Dynamické a implicitní typování)
Implicitní typování	Implicit typing	Možnost při deklaraci vynechat typ proměnné na jedné straně (viz kapitola 12 Dynamické a implicitní typování)
Regulární výraz	Regular expression	Textové řetězce tvořené zástupnými znaky zastupujícími například čísla, písmena nebo jiné znaky (viz kapitola 16 Regulární výrazy)
Operátor	Operator	Znak, nebo skupina znaků, označující operaci, která se má na daném místě provést. (viz kapitola 15 Operátory)

Seznam literatury

[1] *C# Language Specification Version 5.0* – Microsoft Corporation, 2012 [online] Dostupné z: https://www.microsoft.com/en-us/download/details.aspx?id=7029

[2] GOSLING J., JOY B., STEELE G., BRACHA G., BUCKLEY A. *The Java Language Specification, Java SE 8 Edition*, 2015 [online] Dostupné z: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

[3] SNÁŠEL V. (ed.) *OBJEKTY 2005.* 1. vyd. VŠB – Technická univerzita Ostrava, 2005. 324 s. ISBN 80-248-0595-2

[4] About the Java Technology. *Oracle.com* [online]. [cit. 2016-03-06]. Dostupné z: https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

[5] TIOBE Index. *TIOBE* [online]. 2016 [cit. 2016-03-06]. Dostupné z: http://www.tiobe.com/tiobe_index?page=index

[6] History of Java Technology. *Oracle.com* [online]. [cit. 2016-03-06]. Dostupné z: http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html

[7] Java Timeline. *Oracle.com* [online]. [cit. 2016-03-11]. Dostupné z: http://oracle.com.edgesuite.net/timeline/java/

[8] What is open source. *Opensource.com* [online]. [cit. 2016-03-11]. Dostupné z: https://opensource.com/resources/what-open-source

[9] The Java Community Process(SM) Program. *The Java Community Process* [online]. [cit. 2016-03-11]. Dostupné z: https://www.jcp.org/en/home/index

[10] Licences - GNU Project. *The GNU Operating System and the Free Software Management* [online]. [cit. 2016-03-11]. Dostupné z: http://www.gnu.org/licenses/licenses.html#GPL

[11] WebSockets - Web APIs. *Mozilla Developer Network* [online]. [cit. 2016-03-11]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[12] Introduction to the C# Language and .NET Framework. *Microsoft Developer Network* [online]. [cit. 2016-03-12]. Dostupné z: https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx

[13] Zapouzdření. *Gymnázium Matyáše Lercha* [online]. [cit. 2016-03-12]. Dostupné z: http://www.gml.cz/projekty/objektove-programovani/programy/output/ch02s04.html

[14] Dědičnost. *Gymnázium Matyáše Lercha* [online]. [cit. 2016-03-12]. Dostupné z: http://www.gml.cz/projekty/objektove-programovani/programy/output/ch02s02.html

[15] Polymorfismus. *Vysoká škola báňská - Technická univerzita Ostrava* [online]. [cit. 2016-03-12]. Dostupné z: http://www.cs.vsb.cz/benes/vyuka/upr/texty/objekty/ch01s03s03.html

[16] Understanding .NET Just-In-Time Compilation. *Telerik* [online]. [cit. 2016-03-13]. Dostupné z: http://www.telerik.com/blogs/understanding-net-just-in-time-compilation

[17] History of C# Programming. *All About C# Programming* [online]. [cit. 2016-03-13]. Dostupné z: http://aboutcsharpprogramming.blogspot.cz/2012/09/history-of-c-programming.html

[18] Výrazy Lambda. *Microsoft Developer Network* [online]. [cit. 2016-03-13]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb397687.aspx

[19] C# 4.0 new features. *ZetCode, tutorials for programmers* [online]. [cit. 2016-03-30]. Dostupné z: http://zetcode.com/lang/csharp/csharp4/

[20] New features added to C# 5.0. *DotNet Tricks* [online]. [cit. 2016-03-30]. Dostupné z: http://www.dotnet-tricks.com/Tutorial/csharp/TaH9171113-New-features-added-to-C

[21] Co je nového v C# 6.0. *Czech MSDN Blog* [online]. [cit. 2016-03-30]. Dostupné z: https://blogs.msdn.microsoft.com/vyvojari/2015/05/11/co-je-novho-v-c-6-0/

[22] Moduly a knihovny. *Fakulta informačních technologií VUT v Brně* [online]. [cit. 2016-03-31]. Dostupné z: http://www.fit.vutbr.cz/~martinek/clang/modules.html

[23] C# From a Java Developer Perspective. *Dare Obsanjo's Home Page* [online]. [cit. 2016-04-07]. Dostupné z: http://www.25hoursaday.com/CsharpVsJava.html#nested

[24] Comparing Java and C# Generics. *Jonatan Pryor Research Labs* [online]. [cit. 2016-04-09]. Dostupné z: http://www.jprl.com/Blog/archive/development/2007/Aug-31.html

[25] Main() and Other Methods (C# vs Java). *Microsoft Developer Network* [online]. [cit. 2016-04-09]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms228506%28v=vs.90%29.aspx

[26] *Generické programování v jazyce Java*. České Budějovice, 2009. Bakalářská práce. Jihočeská univerzita v Českých Budějovicích.

[27] Změny v JDK 7. *Ing. Vojtěch Hodějčuk* [online]. [cit. 2016-04-10]. Dostupné z: http://voho.cz/wiki/java-zmeny-jdk7/

[28] Enum - Comparsion of Java and .NET. *CodeProject* [online]. [cit. 2016-04-12]. Dostupné z: http://www.codeproject.com/Articles/688300/Enum-Comparison-of-Java-and-NET

[29] Overriding and overloading in Java and .NET - differences, changes and gotchas. *CodeProject* [online]. [cit. 2016-04-15]. Dostupné z: http://www.codeproject.com/Articles/692127/Overriding-and-overloading-in-Java-and-NET-differe

[30] PECINOVSKÝ, Rudolf. *Java 7: učebnice objektové architektury pro začátečníky*. Vyd. 1. Praha: Grada, 2012. Knihovna programátora (Grada). ISBN 978-80-247-3665-5.

[31] PECINOVSKÝ, R. *Java 8 - Úvod do objektové architektury pro mírně pokročilé.* 1. vyd. Praha: Grada Publishing, a.s, 2014. 656 s. ISBN 978-80-247-4638-8. URL: http://www.grada.cz/java-8_8543/kniha/katalog/?dopln=listovani-google

[32] Poznáváme C# a Microsoft .NET. *Živě.cz* [online]. [cit. 2016-04-17]. Dostupné z: http://www.zive.cz/clanky/poznavame-c-a-microsoftnet--8-dil/sc-3-a-122182/

[33] LINQ (Language-Integrated Query). *Microsoft Developer Network* [online]. [cit. 2016-04-18]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb397926.aspx

[34] Java Streams Preview vs .NET High-Order Programming with LINQ. *Informatech CR Blog* [online]. [cit. 2016-04-18]. Dostupné z: https://blog.informatech.cr/2013/03/24/java-streams-preview-vs-net-linq/

[35] Statické třídy a její členové. *Microsoft Developer Network* [online]. [cit. 2016-04-18]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/79b3xss3.aspx

[36] Static Classes In Java. *Stack Overflow* [online]. [cit. 2016-04-18]. Dostupné z: http://stackoverflow.com/questions/7486012/static-classes-in-java

[37] Statické konstruktory (Průvodce programováním v C#). *Microsoft Developer Network* [online]. [cit. 2016-04-18]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/k9x6w0hc.aspx

[38] Java Naming Standards and Programming Conventions. *IWombat.com* [online]. [cit. 2016-04-21]. Dostupné z: http://www.iwombat.com/standards/JavaStyleGuide.html

[39] Namespace Naming Guidelines. *Microsoft Developer Network* [online]. [cit. 2016-04-21].
Dostupné z: https://msdn.microsoft.com/en-us/library/893ke618%28v=vs.71%29.aspx

[40] Interface Naming Guidelines. *Microsoft Developer Network* [online]. [cit. 2016-04-21].
Dostupné z: https://msdn.microsoft.com/en-us/library/8bc1fexb%28v=vs.71%29.aspx

[41] Regulární výrazy v Javě. *ITnetwork.cz* [online]. [cit. 2016-04-22]. Dostupné z: http://www.itnetwork.cz/java/pokrocile/tutorial-java-regularni-vyrazy

[42] Implicitně typované lokální proměnné. *Microsoft Developer Network* [online]. [cit. 2016-04-25]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb384061.aspx

[43] Anonymní typy. *Microsoft Developer Network* [online]. [cit. 2016-04-25]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb397696.aspx

[44] Použití typu dynamic. *Microsoft Developer Network* [online]. [cit. 2016-04-25]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/dd264736.aspx

[45] *Java efektivně: 57 zásad softwarového experta*. 1. vyd. Praha: Grada, 2002, s. 94-95. Moderní programování. ISBN 80-247-0416-1.

[46] Delegates (C# Programming Guide). *Microsoft Developer Network* [online]. [cit. 2016-04-29]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms173171.aspx

[47] RYTYCH, M. *Možnosti deklarativního programování v jazyku Java 8 Maxim Rytych.* Diplomová práce. 2015.

[48] SÝKORA, J. *Knihovna umožňující práci s libovolnými zdroji dat prostřednictvím SQL dotazů [elektronický zdroj].* Bakalářská práce. 2015.

[49] Introduction to Functional Interfaces - A Concept Recreated in Java 8. *DZone Java* [online]. [cit. 2016-04-29]. Dostupné z: https://dzone.com/articles/introduction-functional-1

[50] Anonymous Methods (C# Programming Guide). *Microsoft Developer Network* [online]. [cit. 2016-05-01]. Dostupné z: https://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx

[51] Anonymous Classes (The Java Tutorial). *Oracle Java Documentation* [online]. [cit. 2016-05-01]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

[52] PECINOVSKÝ, Rudolf. *OOP a Java 8: Návrh a vývoj složitějšího projektu vyhovujícího zadanému rámci*. 1. Eva & Tomáš Brucknerovi, 2015. ISBN 978-80-87924-03-7.

Seznam obrázků

Obrázek 1 – Schéma kompilace zdrojového kódu Javy 10
Obrázek 2 – Schéma kompilace zdrojového kódu pro platformu .NET 11

Rejstřík

anonymní metody, 47
anonymní třídy, 48
as, 53
bytekód, 10
camel case, 13
defaultní hodnota, 38
delegáty, 47
diamantový operátor, 46
dynamic, 45
enum, 43
final, 19
generický datový typ, 33, 62
generický parametr, 34
gettery, 30
import, 16
inner class, 21
instanceof, 53
interfejs, 23
is, 53
Java Virtual Machine, 9
knihovní třída, 19
kontravariance, 42
kovariance, 41
LINQ, 50
main, 26
metoda, 24, 34, 35
namespace, 15
namespace alias, 17
nested types, 21
new, 19, 28, 36, 41, 46, 48
objekt, 20
operátor, 53
optional, 55
out, 24
override, 28
package, 15
params, 25
partial, 18
pascal case, 12
platforma
  .NET, 10
  Java, 9
Přebíjení, 27
přetěžování, 54
Přetěžování, 29
přetypování, 53
raw type, 33
ref, 24
regulární výraz, 57
sealed, 19
settery, 30
static, 19
statická třída, 19
statický inicializátor, 21
statický konstruktor, 20
Streams API, 50
syrový typ, 33
třída, 18
typeof, 53
var, 45
virtual, 28
výčtový typ, 43
wildcard, 36
zakrývání metod, 28
žolíky, 36

Příloha A: CD

K bakalářské práci je přiloženo CD na kterém se nachází následující obsah:

- Textová adventura psaná v Javě a upravená pro potřeby této bakalářské práce
- Framework pro tuto textovou adventuru vytvořený pro předmět *4IT101 – Programování v Javě*
- Textová adventura převedena z jazyka Java do jazyka C#
- Potřebná část frameworku pro textovou adventuru převedena do jazyka C# pro zajištění kompatibility obou adventur