VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SKRIPTOVACÍ JAZYKY NA PLATFORMĚ JAVA

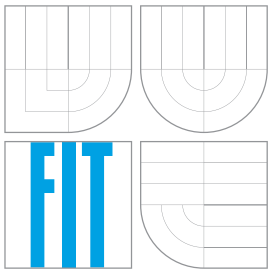
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL GENSEREK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SKRIPTOVACÍ JAZYKY NA PLATFORMĚ JAVA

SCRIPTING LANGUAGE ON THE JAVA PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL GENSEREK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2013

Abstrakt

Práce se zabývá skriptovacími jazyky odpovídajícími normě JSR223 na platformě Java, součástí práce je i popis aplikačního rozhraní vyžadovaného touto normou. Dále je v práci zahrnuto zhodnocení problémů vyplývajících z použití skriptovacích jazyků na platformě Java, včetně jejich možných řešení. Jsou zde rovněž popsány jazyky Python, Ruby, JavaScript, Groovy a Clojure včetně případných rozdílů mezi různými implementacemi jejich běhového prostředí. Práce obsahuje také sadu benchmarků pro srovnání výkonnosti jednotlivých popsaných jazyků a jejich implementací. Součástí jsou rovněž výsledky tohoto srovnání nad různými virtuálními stroji jazyka Java. V rámci práce byla vyvinuta aplikace usnadňující měření výkonnosti skriptovacích jazyků spouštěných v rámci aplikačního rozhraní JSR223.

Abstract

This thesis aims at JSR223 compliant scripting languages on the Java platform including a description of the application interface that is part of the specification. The thesis also discusses possible problems resulting from using the scripting languages on the Java platform including possible solutions of this problems. Description of the languages Python, Ruby, JavaScript, Groovy and Clojure including eventual differences between this languages' different implementations is also included in the thesis. The thesis also contains a set of tests for comparing described languages' performance. The results of this comparison including results for different Java virtual machines are also included in the thesis. Tool for benchmarking of the scripting languages under the JSR223 application interface was developed as part of the thesis.

Klíčová slova

JSR223, Java, Python, Jython, Ruby, JRuby, Clojure, JavaScript, Rhino, Groovy, benchmark, skriptovací jazyky

Keywords

JSR223, Java, Python, Jython, Ruby, JRuby, Clojure, JavaScript, Rhino, Groovy, benchmark, scripting languages

Citace

Michal Genserek: Skriptovací jazyky na platformě Java, bakalářská práce, Brno, FIT VUT v Brně, 2013

Skriptovací jazyky na platformě Java

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Kočího, Ph.D. Další informace mi poskytl Ing. Pavel Tišnovský, Ph.D., zástupce společnosti Red Hat. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Genserek
15. května 2013

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Radkovi Kočímu, Ph.D. za vedení práce a poskytnuté rady. Dále bych chtěl poděkovat Ing. Pavlovi Tišnovskému, Ph.D. z firmy Red Hat.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Vývoj aplikací je bez ohledu na použité prostředky náročným procesem, ať již z hlediska času, financí, anebo úsilí potřebného k vytvoření kvalitního produktu. Je proto pochopitelné, že s pokračující invencí v oboru programování vznikaly různé nástroje a techniky, které měly zvýšit produktivitu a usnadnit vývojářům jejich práci. Vznik jazyka a platformy Java v polovině devadesátých letech přinesl možnost jednoduše psát multiplatformní programy pro širokou škálu systémů a zařízení a postupem času se stal hojně využívaným nástrojem. S postupným nárustem výkonu počítačů si získaly svou oblibu pro vývoj aplikací také různé skriptovací jazyky, jež nebyly nadále diskriminovány svým výkonem a mohly jejich uživatelům nabídnout vlastnosti a možnosti, které kompilované (také nazývané systémové) jazyky nenabízejí. A právě skloubení vlastností skriptovacích jazyků s jazykem Java se ukázalo jako potencionálně výhodné – skriptovací jazyky umožnily ušetřit čas a nabídly zajímavé možnosti s poměrně menším úsilím, než by tomu bylo při použití kompilovaných jazyků. Příkladů takovéhoto propojení jazyků existuje celá řada, ať již pro rychlé prototypování částí aplikace při raném vývoji, vytváření zásuvných modulů třetími stranami v libovolném skriptovacím jazyce, nebo použití takového jazyka k záznamu a zpětnému přehrávání uživatelských akcí (tzv. maker).

Tato práce se zabývá popisem rozhraní, které poskytuje platforma Java pro využití skriptovacích jazyků, také známé jako *Java*^{TM} *Scripting API*, určené normou JSR 223. Kapitola 2 obsahuje popis aplikačního rozhraní JSR 223 a přehled jeho tříd. Následující kapitola číslo 3 se zabývá popisem vybraných jazyků kompatibilních s JSR 223, včetně popisu jejich implementací. Tato kapitola také diskutuje možné problémy při použití těchto jazyků na platformě Java. Kapitola 4 obsahuje popis navržených měřících testů a jejich vyhodnocení. Kapitola 5 obsahuje shrnutí práce a diskutuje možný vývoj skriptovacích jazyků na platformě Java.

Kapitola 2

Aplikační rozhraní pro komunikaci mezi skripty a aplikacemi

Jedním z případů užití skriptovacích jazyků na platformě Java, je jejich využití v aplikacích, které pro svou činnost používají skripty napsané v některém skriptovacím jazyce. Tyto skripty jsou často poskytovány koncovými uživateli dané aplikace, ale nemusí tomu tak být vždy. Platforma Java nabízí framework *Java™ Scripting API*, nezávislý na konkrétním skriptovacím jazyce, sloužící pro vývoj takto rozšiřitelných a přizpůsobitelných aplikací v jazyce Java. Pokud programátor provádějící vývoj aplikace využije a dodrží aplikační rozhraní dané normou JSR 223, pak jeho řešení umožňuje ponechat výběr konkrétního skriptovacího jazyka na koncovém uživateli dané aplikace. Podmínkou však je použití jazyka implementujícího specifikaci JSR 223 [1, 2].

2.1 Aplikační rozhraní

Funkcionalita skriptovacích jazyků v Javě je zajištěna pomocí balíku `javax.script`. Tento balík obsahuje rozhraní a třídy nutné k použití nebo vytvoření skriptovacího jazyka a nachází se v distribuci jazyku Java od verze 1.6. Hlavní oblasti funkcionality, které tento balík zajišťuje, jsou [3]:

1. **Vykonávání skriptů:** skripty jsou řetězce znaků tvořící program, jež je následně zpracován interpretem. Výsledkem spuštění programu ve skriptovacím jazyce je vždy objekt, v důsledku tak není rozlišeno mezi skripty nevracejícími hodnotu a těmi jež tak činí. Skripty jsou také spouštěny synchronně a specifikace neurčuje žádný mechanismus pro jejich asynchronní spouštění, případně přerušení jejich činnosti z jiného vlákna – v případě potřeby musí být tyto mechanismy implementovány přímo v konkrétní aplikaci. Samotné spuštění skriptu je uskutečněno pomocí metody `eval` a rozhraní *Compilable* [1].

2. **Navazování proměnných (*binding*):** umožňuje navázání a viditelnost objektů hostitelské aplikace do skriptu a také opačně viditelnost objektů nebo proměnných skriptu v hostitelské aplikaci. Tato funkcionalita je implementována pomocí rozhraní *Bindings* a *ScriptContext* [1].

3. **Kompilace skriptů:** umožňuje uchovat vnitřní kód pro interpret skriptovacího jazyka a při dalším vykonání skriptu nemusí být již znovu překládán. Tato vlastnost

je volitelná pro různé implementace skriptovacích jazyků a hostitelská aplikace musí kontrolovat, zda objekt interpretu implementuje rozhraní *Compilable* [1].

4. **Invokace metod nebo funkcí:** umožnuje spuštění pouze části již zkompilovaného vnitřního kódu reprezentujícího funkci nebo proceduru ve skriptovacím jazyce. Stejně jako kompilace skriptů, je tato vlastnost volitelná pro různé implementace a pro její použití je třeba zkontrolovat, zda objekt interpretu implementuje rozhraní *Invocable* [1].

5. **Hledání dostupných skriptovacích jazyků:** aplikace využívající aplikační rozhraní mohou mít různé požadavky na interprety skriptovacích jazyků, např. závislost na konkrétní verzi, případně jazyku. Samotné interprety, které jsou zabaleny dle normy JSR 223, jsou objeveny pomocí mechanismu pro vyhledávání balíků (*Script Engine Discovery Mechanism*). Díky tomuto systému nemusí být aplikace přímo sestavovány nebo konfigurovány se seznamem dostupných nebo použitých interpretů [1].

2.1.1 Instance interpretu

Instance interpretů jsou reprezentovány objekty implementujícími rozhraní *ScriptEngine* – ke každému objektu této třídy pak náleží příslušná instance implementující *ScriptEngineFactory*, které vznikly při procesu hledání balíků. Samotnou instanci interpretu lze získat pomocí jeho konstruktoru, případně pomocí metod instance `ScriptEngineManager` jež využije příslušného objektu s rozhraním *ScriptEngineFactory*. Pokud je instance interpretu získána bez pomoci `ScriptEngineManager`, nemá tato instance přístup do globální kolekce dvojic klíč/hodnota společných pro všechny interprety vytvořené jedním objektem třídy `ScriptEngineManager` [1].

2.1.2 Bindings

Rozhraní *Bindings* představuje kolekci dvojic klíč/hodnota, kde klíč nesmí představovat prázdnou hodnotu, nebo hodnotu typu `null`. Třídy implementující toto rozhraní se používají k reprezentaci různých rozsahů platnosti proměnných nebo jiných objektů [1].

2.2 Přehled tříd a rozhraní v JSR 223[3]

Kompletní přehled tříd, rozhraní a výjimek balíku `javax.script` je uveden ve formě diagramu tříd v příloze A.

2.2.1 Rozhraní

- Compilable
- Invocable
- Bindings
- ScriptContext
- ScriptEngine
- ScriptEngineFactory

2.2.2 Třídy

- CompiledScript
- SimpleScriptContext
- AbstractScriptEngine
- ScriptEngineManager

2.2.3 Výjimky

- ScriptException

Kapitola 3

Prostředí a jazyky na platformě Java

Skriptovací jazyky jsou v dnešní době stále častěji implementovány tak, aby pro svůj běh využívaly virtuální stroj. Takto navržené jazyky je možno zkompilovat do *bytecode* pro daný virtuální stroj a tím odpadá nutnost provádět lexikální a syntaktickou analýzu, jako u čistě interpretovaných jazyků při každém jejich spuštění. Toto ale klade nároky na tvůrce implementací těchto jazyků – místo aby se zabývali tvorbou jiných aspektů jejich jazyka, jsou nuceni vytvářet a testovat virtuální stroj, obzvláště pokud jej vyvíjejí zároveň pro více platforem. Platforma Java poskytuje tvůrcům implementací skriptovacích jazyků virtuální stroj nabízející mnoho vlastností, zahrnující například konkurenčnost, garbage collector, reflektivní přístup k objektům a třídám za běhu, nebo například rozhraní *JVM Tools Interface*, umožňující nativním aplikacím prozkoumat a řídit tok běžícího procesu nad JVM. V rámci projektu *Da Vinci Machine Project* přichází platforma Java SE 7 s podporou pro dynamické programovací jazyky. Tato podpora je dána JSR 292, který přidává do bytecode virtuálního stroje Javy instrukci `invokedynamic`, umožňující virtuálnímu stroji vytvářet vazby mezi voláním a konkrétní implementací metody až za běhu aplikace, což vyřešilo některé problémy dynamických jazyků pro JVM (viz kapitola 3.7).

3.1 Běhová prostředí jazyku Java

Programy a aplikace na platformě Java vyžadují ke svému běhu nainstalované specifické běhové prostředí, běžně označované jako JRE (*Java Runtime Environment*). Toto běhové prostředí je zodpovědné za spuštění a běh aplikací určených pro platformu Java a musí nutně sestávat minimálně z virtuálního stroje Javy, dále pak balíku základních knihoven a případně nativních knihoven pro správný běh virtuálního stroje. Tato běhová prostředí jsou typicky vyráběna a kompilována pro konkrétní kombinace hardware a operačních systémů, ale díky jejich vzájemné kompatibilitě, zapříčiněné dodržovaním specifikací (konkrétně *Java Virtual Machine Specification*), dovolují na nich běžícím aplikacím odstínit různorodost platforem [14].

Virtuální stroj Javy (dále jen JVM) slouží k provádění Java bytecode a chová se jako propojující článek mezi aplikací pro platformu Java a hostitelským operačním systémem. Jedná se o abstraktní procesor, který stejně jako reálné procesory zpracovává instrukce dané jeho instrukční sadou a manipuluje s výpočetní pamětí. Samotné JVM není přímo závislé na jazyce Java a pracuje pouze s binárním formátem, který je nezávislý na kon-

krétním hardware či operačním systému a jeho specifikace přesně definuje (včetně např. endianity), jak mají být uloženy třídy, rozhraní a ostatní typy zpracovatelné pomocí JVM. Tento formát se nazývá souborovým formátem `class`. Nezávislost binárního formátu pro JVM a samotného JVM na konkrétním jazyce umožnila vývoj překladačů i pro jiné jazyky než právě Java. Programy a aplikace zapsané v těchto jazycích pak mohou využívat výhod prostředí poskytovaného JVM. Specifikace pro vývoj JVM vyžaduje po vývojářích virtuálního stroje pouze přesné chování a načtení instrukcí reprezentovaných v souborovém formátu `class`. Implementační detaily ostatních částí JVM, jako například způsob reprezentace paměti pro běžící procesy nad JVM, algoritmus podle kterého bude pracovat *garbage collector*, nebo případná optimalizace bytecode, jsou ponechány na tvůrcích virtuálního stroje. Tato volnost pak dává prostor pro vznik alternativních implementací JVM, lišících se vnitřní implementací, ale stále zachovávající instrukční sadu a schopnost spouštět programy reprezentované pomocí souborového formátu `class`. Jako referenční implementace JVM slouží virtuální stroj nesoucí název *Java HotSpot*^{TM} *Performance Engine*, často uváděn pouze pod názvem *Java HotSpot*, který je vyvíjen jako součást projektu OpenJDK. Virtuální stroj *Java HotSpot* obsahují distribuce běhových prostředí *Oracle Java SE Runtime Environment* a *OpenJDK Runtime Environment*. V současnosti obsahuje stroj *Java HotSpot* dvě rozdílné verze – tyto dvě verze jsou nazvány *Server* a *Client*. Rozdíly v těchto verzích se vztahují k vyhodnocení nutnosti JIT (*Just In Time*) kompilace, alokaci paměti a dalším technikám pro efektivnější běh aplikace. Verze nazvaná *Client* je optimální pro běh aplikací, které vyžadují krátkou dobu startu, případně malou paměťovou stopu. Tyto požadavky splňují obecně uživatelské aplikace s GUI. Naproti tomu, verze *Server* je optimálnější pro aplikace s dlouhou dobou běhu [12].

Oracle Java SE Runtime Environment

Tato distribuce běhového prostředí je vydávána firmou Oracle pod licencí *Oracle Binary Code License* a v současné verzi obsahuje obě verze *Java HotSpot* JVM. Virtuální stroj, stejně jako jako většina ostatních nástrojů obsažených v této distribuci, pochází od verze 7 z projektu OpenJDK, na jehož vývoji se firma Oracle také podílí. Předchozí verze byly vydávány firmami Sun Microsystems a Oracle a byly založeny na nesvobodných zdrojových kódech. V 64-bitové verzi je dostupná pouze verze *Server* virtuálního stroje *Java HotSpot*, tento nedostatek by však měl být napraven v dalších verzích [11]. Jako skriptovací jazyk pro JSR 223 je v této distribuci přibalen skriptovací stroj Rhino pro jazyk JavaScript, popsaný v kapitole 3.2.1.

OpenJDK Runtime Environment

Distribuce OpenJDK vznikla ve snaze firmy *Sun Microsystems* uvolnit zdrojové kódy platformy Java pod otevřenou licencí. Pod svobodnou licencí GPL tak byly uvolněny zdrojové kódy připravovaného JDK 7. Jelikož však neexistovala otevřená implementace pro verzi 6, byly z vývojové verze 7 odstraněny vlastnosti porušující kompatibilitu se specifikací pro verzi 6, a takto upravená verze pak byla vydána jako OpenJDK 6. Ačkoliv je distribuce OpenJDK od verze 7 v mnoha směrech naprosto totožná s distribucí firmy Oracle, oficiální repozitáře s kódem OpenJDK nezačleňují JavaScriptový stroj Rhino. Důvodem k tomuto kroku je nekompatibilní licence zdrojových kódů Rhina a OpenJDK. Rhino pro OpenJDK je obsaženo v projektu IcedTea.

IBM Java Runtime Environment

Distribuce běhového prostředí využívající JVM s názvem J9, vyvinutého firmou IBM. Od verze implemetující Java SE 7 využívá tato distribuce balíky `java.util.*` identické s těmi, jež jsou zahrnuty v distribuci firmy Oracle. Cílem tohoto kroku je dle dokumentace zajištění ustáleného výkonu a funkčnosti napříč různými implementacemi platformy Java [8]. Samotný JVM J9, obsažený v této distribuci, se od výše zmíněného *Java HotSpot* odlišuje implementací vnitřních algoritmů pro jednotlivé činnosti vykonávané těmito JVM.

3.2 JavaScript

JavaScript je interpretovaný, vysokoúrovňový, netypovaný dynamický skriptovací jazyk. Syntaxe je převážně odvozena z jazyka Java, syntaxe funkcionálních prvků jazyka pak vychází z jazyků Self a Scheme. Obsahuje rysy objektově orientovaného a funkcionálního programování. Jeho jméno „JavaScript“ bylo zvoleno z marketingových důvodů a s platformou a jazykem Java nemá jazyk JavaScript nic jiného společné. Byl vyvinut společností Netscape a jeho autorem je Brendan Eich. Samotný jazyk byl standardizován ECMA[1] pod názvem *ECMAScript* v roce 1997. Jazyk byl již od svého počátku silně provázán s webem a internetovými prohlížeči (v současnosti lze jeho implementaci nalézt v téměř všech majoritních internetových prohlížečích) – bývá proto také nazýván jazykem Webu [15].

Rozšířenost JavaScriptu postupně vedla k jeho mnoha implementacím, mezi kterými lze nalézt několik implementací poskytujících rozhraní JSR 223.

3.2.1 Rhino

Implementace JavaScriptu nazvaná Rhino byla vyvíjena společností Netscape od roku 1997 výhradně v jazyce Java a v současné době je spravována společností Mozilla Foundation. V současné době se jedná o open-source projekt, jež podporuje všechny prvky jazyka JavaScript 1.7, který je kompatibilní se specifikací jazyka ECMA-262. Interpret je obsažen v oficiálních distribucích JDK verze 6 a 7 společnosti Oracle.

Tabulka 3.1: Údaje poskytované distribucí Rhino

Název	Mozilla Rhino
Verze	1.7 release 3 PRERELEASE
Jazyk	ECMAScript
Verze jazyka	1.8
Podporované přípony	js
Podporovaná jména	js, rhino, JavaScript, javascript, ECMAScript, ecmascript

3.3 Python

Python je dynamický silně typovaný skriptovací jazyk, umožňující programování dle objektově orientovaného paradigma. Syntaxe byla navržena s ohledem na lepší čitelnost programů

[1] European Computer Manufacturer's Association

zapsaných v tomto jazyce. Autorem jazyka je Guido van Rossum a historie se datuje až do osmdesátých let. V současnosti existuje více použitelných implementací jazyka Python, přičemž jako referenční slouží implementace s názvem CPython (název také vyjadřuje skutečnost, že je tato implementace naprogramována v jazyce C). Mezi další známé fungující implementace patří IronPython (vytvořena v jazyce C# pro virtuální stroj platformy .NET), PyPy (implementace v jazyce Python), PyMite (interpret určený pro běh na mikrokontrolérech s limitovanými zdroji), pyjamas a Brython (překladače jazyka Python do JavaScriptu) a Jython, který je blíže popsán v následující podkapitole.

3.3.1 Jython

Jython je implementací jazyka Python v jazyce Java využívající ke svému běhu JVM. Od verze Jython 2.5.1 je součástí distribuovaného balíku i implementace rozhraní JSR 223 pro snadnější integraci interpretu Jythonu do aplikací v Javě. V současnosti Jython nabízí pouze implementace verzí 2.5 a 2.7 jazyka Python. Dle autorů implementace Jython je rychlost modulů naprogramovaných v Javě srovnatelná nebo rychlejší oproti implementaci CPython. Vzhledem k odlišnostem JVM od virtuálního stroje CPythonu a neúplné specifikaci jazyka Python, má mezi sebou Jython a referenční implementace CPython rozdílné chování v následujících bodech [10] (uvažovány verze jazyka Python 2.5 a 2.7):

- Odlišné chování při práci nad čísly s plovoucí desetinnou čárkou.
- Každý objekt v Jythonu je instancí třídy, naproti CPythonu zde neexistují primitivní datové typy.
- Jython nepodporuje *restricted execution mode* – nástroj jazyka Python pro oddělování důvěryhodného a nedůvěryhodného kódu.
- Objekty typu funkce v Jython nemají oproti CPython zapisovatelné členské proměnné `func_code` (obsahující zkompilované tělo funkce) a `func_defaults` (obsahuje implicitní parametry funkce, pokud byly specifikovány).

Tabulka 3.2: Údaje poskytované distribucí Jython

Název	jython
Verze	2.7.0
Jazyk	python
Verze jazyka	2.7
Podporované přípony	py
Podporovaná jména	python, jython

3.4 Groovy

Dynamický jazyk od počátku určený pro platformu Java, který kompiluje zdrojový kód přímo do Java bytecode. Byl navržen jako alternativa k jazyku Java, snažící se o zjednodušení zápisu zdrojového kódu a jeho syntaxe vychází ze skriptovacích jazyků Python, Ruby a Smalltalk. Distribuce nástrojů pro programování v Groovy obsahují rozhraní pro komunikaci dle standardu JSR 223 od verze Groovy 1.6.

Tabulka 3.3: Údaje poskytované distribucí Groovy

Název	Groovy Scripting Engine
Verze	2.0
Jazyk	Groovy
Verze jazyka	2.1.2
Podporované přípony	groovy
Podporovaná jména	groovy, Groovy

3.5 Clojure

Dynamický funkcionální jazyk určený pro platformu Java, stejně jako Groovy je kompilován do bytecode pro JVM. Jedná se o dialekt jazyka Lisp, zároveň přebírající některé prvky z jiných funkcionálních jazyků jako Haskell, ML a ostatních.

Tabulka 3.4: Údaje poskytované distribucí Clojure

Název	Clojure Scripting Engine
Verze	1.2
Jazyk	Clojure
Verze jazyka	1.2
Podporované přípony	clj
Podporovaná jména	Clojure

3.6 Ruby

Ruby je dynamicky typovaný skriptovací jazyk, podporující funkcionální, objektově orientované a imperativní paradigmata, jež vznikl v devadesátých letech a vychází z jazyků Perl, Smalltalk, Eiffel, Ada a Lisp. Jeho autorem je Yukihiro Matsumoto. Stejně jako je tomu u jazyka Python i pro Ruby existuje v současné době několik implementací navzájem se lišících svým zaměřením: Ruby MRI, také nazývána CRuby (referenční implementace jazyka Ruby v C), IronRuby (implementace jazyka Ruby pro virtualní stroj platformy .NET), MacRuby (implementace Ruby využívající technologií dostupných v operačním systému Mac OS X), Rubinius (interpret Ruby napsaný v Ruby) a JRuby, které je blíže popsáno v následující podkapitole.

3.6.1 JRuby

Implementace jazyka Ruby v jazyce Java. Od verze JRuby 1.4 se nachází podpora pro JSR 223 přímo v distribuci JRuby. Samotná implementace obsahuje několik odlišností od referenční implementace Ruby MRI – mezi tyto odlišnosti se řadí [5]:

- Nejsou podporovány nativní rozšiřující knihovny napsané v jazyce C.

- Nejsou podporovány metody `callcc` objektů třídy `Continuation`. Tyto objekty udržují návratovou adresu a běhový kontext a umožnují v jazyce Ruby nelokální skoky. Nabízejí tedy podobnou funkčnost jako metody `setjmp` a `longjmp` standardní knihovny jazyka C [17].
- Metoda `Time.now.usec` nevrací v JRuby čas s přesností na mikrosekundy. Je to dáno nepodporou měření času s přesností na mikrosekundy v JVM – samotné JVM umí měřit čas s přesností na nanosekundy, ale takto získané hodnoty nemusí odpovídat skutečnému času v nanosekundách.
- Rozdílné chování regulárních výrazů.
- Rozdílné chování při nastavování priorit vláknům.

Tabulka 3.5: Údaje poskytované distribucí JRuby

Název	JSR 223 JRuby Engine
Verze	1.7.3
Jazyk	ruby
Verze jazyka	1.7.3
Podporované přípony	rb
Podporovaná jména	ruby, jruby

3.7 Problémy dynamických jazyků na platformě Java

3.7.1 Přetížené metody

Jazyk Java dovoluje použití přetížených metod, konkrétně metod v jedné třídě, majících stejné jméno, ale odlišnou signaturu. Protože je Java jazyk silně typovaný, překladač vybere příslušnou metodu volání přetížené funkce již v době překladu. Naproti tomu, u jazyků s dynamickým typováním, ztrácejí přetížené metody svůj původní smysl, a proto nejsou často v dynamických jazycích podporovány. Tím ovšem nastává problém, pokud chce programátor v dynamickém skriptovacím jazyce nad JVM, pokud to tento umožňuje, využít objekty mající přetížené metody či standardní knihovnu jazyka Java, která taktéž obsahuje v mnohých třídách přetížené metody. V takovém případě je pak na interpretu jazyka, jakou metodu vybere, případně jazyky nad JVM poskytují další mechanismy pro zvolení správné implementace metody pro konkrétní volání.

Přetížené metody v jazyce JavaScript

Spolupráci mezi jazykem Java a skripty v jazyce JavaScript nabízely webové prohlížeče již před specifikací JSR 223 v podobě nástroje *LiveConnect*. Rezoluce přetížených metod až do verze LiveConnect 3 však nebyla tímto nástrojem snadno řešitelná – byla vybrána první nalezená metoda (prohledávání bylo závislé na použitém JVM a nešlo ze skriptu ovlivnit) s odpovídající signaturou. Od verze LiveConnect 3 byla v tomto nástroji implementována heuristická rezoluce přetížené metody, pracující obdobně jako u skriptovacích strojů Jython a JRuby. Do jazyka byly taktéž přidány prvky umožňující zvolit metodu explicitně na

základě její přesné signatury [16]. Skriptovací stroj Rhino využívá stejných praktik jako LiveConnect verze 3 pro řešení rezoluce přetížených metod [6].

Přetížené metody v Jython

Jython se tento problém snaží řešit automaticky – pokud se signatury přetížených metod liší různým počtem parametrů, je automaticky vybrána metoda s odpovídajícím počtem parametrů. Tyto parametry jsou pak převedeny na odpovídající datové typy. Pokud se však signatury liší pouze datovými typy a nikoliv počtem parametrů, Jython implicitně převede předané parametry (reprezentované datovými typy jazyka Python) na jejich ekvivalentní varianty v jazyce Java. Tento převod však nemusí být vždy přesný – jazyk Python obsahuje odlišné primitivní typy s rozdílnou reprezentací hodnot než jazyk Java (např. pouze jeden typ pro čísla s plovoucí desetinnou čárkou, jeden typ pro celá čísla, neexistuje typ pro znak). Pokud pak dochází po tomto implicitním převodu ke zvolení nežádoucí přetížené metody, musí vývojář daného skriptu uvést explicitní převod na odpovídající obalující třídu pro daný datový typ v jazyce Java [13].

Pro ukázku tohoto problému mějme následující třídu v jazyce Java, definující tři verze přetížené metody s názvem `javaMethod`. Z důvodu stručnosti není kód metod uveden, pro samotné pochopení problému není důležitý.

Příklad 3.1: Demonstrační třída v jazyce Java

```
1 class JavaClass {
2   void javaMethod(int a) { /* ... */ }
3
4   void javaMethod(float a) { /* ... */ }
5
6   void javaMethod(long a) { /* ... */  }
7 }
```

Předpokládejme pak dále, že pomocí JSR 223 umožníme přístup k instanci této třídy ve skriptu zpracovávaném pomocí Jython (v příkladu 3.1 pod názvem `javaObject`) a voláme přetíženou metodu. V takovém případě je vždy pro celočíselný parametr zvolena metoda `void javaMethod(long a)` (příklad 3.1, řádek 6). Pro čísla s plovoucí desetinnou čárkou je vždy zvolena správná varianta, v důsledku neexistence jiné alternativy. Následující skript (příklad 3.2) v jazyce Python ukazuje chování interpretu Jython pro volání přetížené metody s různými parametry. V komentáři na každém řádku je uvedena metoda z příkladu 3.1 která bude pro každé volání zvolena a vykonána.

Příklad 3.2: Řešení přetížených metod v Jython

```
1 from java.lang import Integer, Float
2
3 javaObject.javaMethod(10)           # javaMethod(long)
4 javaObject.javaMethod(Integer(10)) # javaMethod(int)
5 javaObject.javaMethod(3.14)         # javaMethod(float)
6 javaObject.javaMethod(Float(10))    # javaMethod(float)
7 javaObject.javaMethod(True)         # javaMethod(long)
8 javaObject.javaMethod("string")     # TypeError
```

Přetížené metody v JRuby

JRuby řeší problém přetížených metod heuristikou, spočívající ve výběru dle vhodné signatury a implicitní konverzi na odpovídající datové typy v jazyce Java. Tento převod se dá navíc explicitně ovlivnit pomocí metody společné všem objektům v JRuby `to_java`. Výběr správné přetížené metody je pak možno ovlivnit pomocí dalších metod specifických pro JRuby. Metoda `java_send` umožňuje programátorovi specifikovat konkrétní metodu na základě jím poskytnutého jména metody a explicitně určené signatury požadované metody. Tento přístup ale může přinést při nevhodném použití signifikantní výkonnostní nedostatky, jelikož ke své činnosti používá reflexi (umožňující za běhu získávat informace o třídě a jejích metodách) jazyku Java. Další možností je využití metody `java_method`, jež vrátí pro zadanou signaturu referenci na příslušnou metodu, jež pak může být spuštěna pomocí volání `call`, což odstraní problémy s reflexí při použití `java_send` – konkrétní metoda je vyhledávána pouze jednou. Metodám lze na základě signatury přiřadit v JRuby další jméno, pomocí metody `java_alias` [7].

Předpokládejme stejnou třídu z jazyku Java, jež je uvedena v příkladě 3.1, avšak tentokrát k ní bude umožněn přístup pro skript v jazyce Ruby. Příklad 3.3 demonstruje chování JRuby při volání přetížených metod. V komentářích nad jednotlivými bloky kódu je uvedena metoda, jež bude pro každé volání v tomto bloku zvolena. Je také možné si povšimnout odlišného chování od skriptu v Jython (příklad 3.2, řádek 7) při převodu hodnoty typu `boolean`, kde byl tento parametr implicitně převeden na hodnotu typu `long`, avšak JRuby odmítne takovýto převod provést (příklad 3.3, řádek 21).

Příklad 3.3: Řešení přetížených metod v JRuby

```
1  # javaMethod(long)
2  javaObject.javaMethod 10
3
4  # javaMethod(int)
5  javaObject.javaMethod 10.to_java(Java::int)
6  javaObject.java_send :javaMethod, [Java::int], 10
7  myMethod = javaObject.java_method :javaMethod, [Java::int]
8  myMethod.call 100
9
10 # javaMethod(float)
11 javaObject.javaMethod 3.14
12
13 # alias metody javaMethod(float)
14 class Java::JavaClass
15   java_alias :javaMethodFloat, :javaMethod, [Java::float]
16 end
17
18 javaObject.javaMethodFloat 10
19
20 # NameError: no method
21 javaObject.javaMethod true
22 javaObject.javaMethod "string"
```

3.7.2 Klíčová slova

Klíčová (někdy také označována jako rezervovaná) slova, jsou v programovacích jazycích identifikátory se speciálním významem v daném jazyce. Používání klíčových slov mimo jejich patřičný kontext, například jako identifikátory pro pojmenování proměnných, vede ve většině jazyků k syntaktické chybě. Specifikace JSR 223 umožňuje skriptům přístup k objektům jazyku Java – tyto objekty mohou obsahovat identifikátory jež nejsou klíčovým slovem v jazyku Java, ale zároveň spadají do množiny klíčových slov v některém skriptovacím jazyce použitelném na platformě Java.

Konflikty klíčových slov v Jython

Ve skriptech zpracovávaných pomocí Jython, jsou konflikty při přístupu k členským položkám objektu řešeny samotným překladačem Jython, který se dle kontextu snaží rozpoznat, zda se jedná o smysluplný výraz. Pokud však konflikt nelze vyřešit, lze dle dokumentace, odlišit problémový identifikátor pomocí znaku podtržítko [13].

Konflikty klíčových slov v jazyce JavaScript

JavaScriptové stroje implementující specifikaci ECMA 5, umožňují přístup pomocí tečkové notace k členským položkám objektu, jež mají jako identifikátor klíčové slovo jazyka JavaScript. Rhino vyhovuje specifikaci ECMA 5 od verze 1.7R3 [4]. Ve starších verzích JavaScriptových implementací, včetně stroje Rhino, je možno využít alternativní syntaxe, jež jazyk nabízí pro přístup k členským prvkům objektů [15].

Konflikty klíčových slov v jazyce JRuby

Při přístupu k členským položkám, které by svým názvem způsobily konflikt s klíčovým slovem jazyka Ruby, je nutno v JRuby použít již výše zmíněné metody `java_send`, případně `java_method` či použít vlastní alternativní pojmenování pomocí `java_alias`.

Kapitola 4

Měření výkonu jazyků na platformě Java

4.1 Způsob měření

4.1.1 Modul timeit jazyka Python

Modul `timeit` jazyka Python je určený k měření doby běhu malých úseků kódu. Toto měření probíhá vícenásobným spuštěním měřeného kódu a jako výsledek je vrácen čas reprezentující součet délek trvání všech provedených běhů. Z tohoto výsledného času je pak možno určit přibližnou dobu běhu pomocí aritmetického průměru.

4.1.2 Modul Benchmark jazyka Ruby

Modul `Benchmark` poskytuje metody pro měření a výpis doby běhu kódu v jazyce Ruby. Výstupem tohoto modulu je výpis na standardní výstup. V tomto výpisu je zahrnut reálný čas běhu, tj. čas který probíhalo vykonání zkoumaného kódu (v unixových systémech uváděn jako *real time*), systémový čas, tj. čas strávený systémovými voláními (v unixových systémech uváděn jako *sys time*) a čas samotného vykonávání kódu aplikace (*user time* v unixových systémech). Tento modul je podporován také v JRuby a byl použit v testech zabývajících se porovnáním rychlosti mezi implementacemi Ruby MRI a JRuby.

4.1.3 Princip testování skriptů s využitím JSR 223 v jazyce Java

Testování skriptů nad JVM bylo provedeno pomocí vlastní aplikace umožňující měřit dobu běhu skriptu a zároveň poskytující skriptům data a struktury potřebné k jejich požadovanému provedení. Tato aplikace byla implementována v jazyce Java s využitím balíku `javax.script`, který poskytuje API dané normou JSR 223 (blíže popsáno v kapitole 2). Aplikace se skládá z abstraktní třídy `JSR223Benchmark`, která je spustitelná (obsahuje metodu `main`) a dále obsahuje abstraktní metody definující chování jednotlivých měřících testů. Třída pak také obsahuje metody zajišťující inicializaci, samotné provedení měření a vypsání výsledků. Samotné šablony pro měřící testy jsou obsaženy v balíku `benchmarks` a jedná se o třídy v jazyce Java dědící od třídy `JSR223Benchmark`. Tyto třídy implementující abstraktní metody dané jejich rodičovskou třídou, obsahují pouze kód popisující chování měřícího testu, ale nikoli již kód provádějící samotné měření – je tak umožněno relativně snadné přidávání nových druhů měřících testů v budoucnu. Měření doby provedení testu je implementováno pomocí metody `System.nanoTime()`, jež vrací hodnotu nejpřesnějších

dostupných hodin na dané platformě s přesností až na nanosekundy. Je možno provést uživatelem specifikovaný počet iterací, vedoucích k zajištění stability naměřených časů – jednotlivá měření mohou být negativně ovlivněna probíhající činností JVM, jako je například úklid paměti *garbage collectorem*. Algoritmus použitý k provádění měření je odvozen z dokumentace jazyku Java a vypadá v pseudokódu následovně [9]:

Příklad 4.1: Pseudokód měřícího algoritmu

```
1 opakuj (počet_iterací) {
2   nastav prosředí úlohy
3
4   čas_začátku = System.nanoTime()
5   proveď úlohu
6   trvání_úlohy = System.nanoTime() - čas_začátku
7
8   vypiš trvání_úlohy
9 }
```

4.1.4 Analýza a interpretace testů

Pro měření byly využity aktuálně dostupné distribuce skriptovacích strojů. Verze použitých distribucí lze dohledat v tabulkách u popisů jednotlivých jazyků, obsažených v kapitole 3. Tedy pro jazyk JavaScript Rhino v tabulce 3.1, pro jazyk Jython v tabulce 3.2, Groovy tabulka 3.3, Clojure v tabulce 3.4 a pro jazyk JRuby v tabulce 3.5.

Měření proběhlo nad distribucemi JRE firem Oracle, IBM a distribucí OpenJDK – konkrétní verze použitých virtuálních strojů jsou uvedeny v tabulce 4.2. Samotné výsledky měření byly získány na počítači s konfigurací uvedenou v tabulce 4.1.

Tabulka 4.1: Konfigurace na které bylo provedeno měření

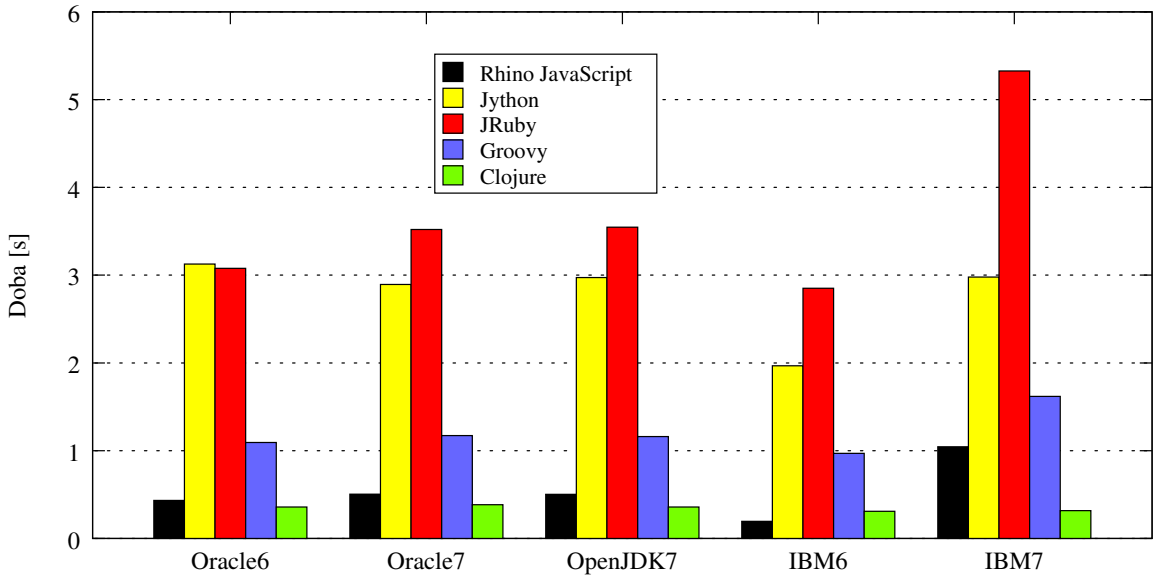
Operační systém	GNU/Linux, openSUSE 12.1
Procesor	Intel Pentium(R) Dual-Core CPU T4200 @ 2.00GHz
RAM	4GiB

Tabulka 4.2: Verze JVM

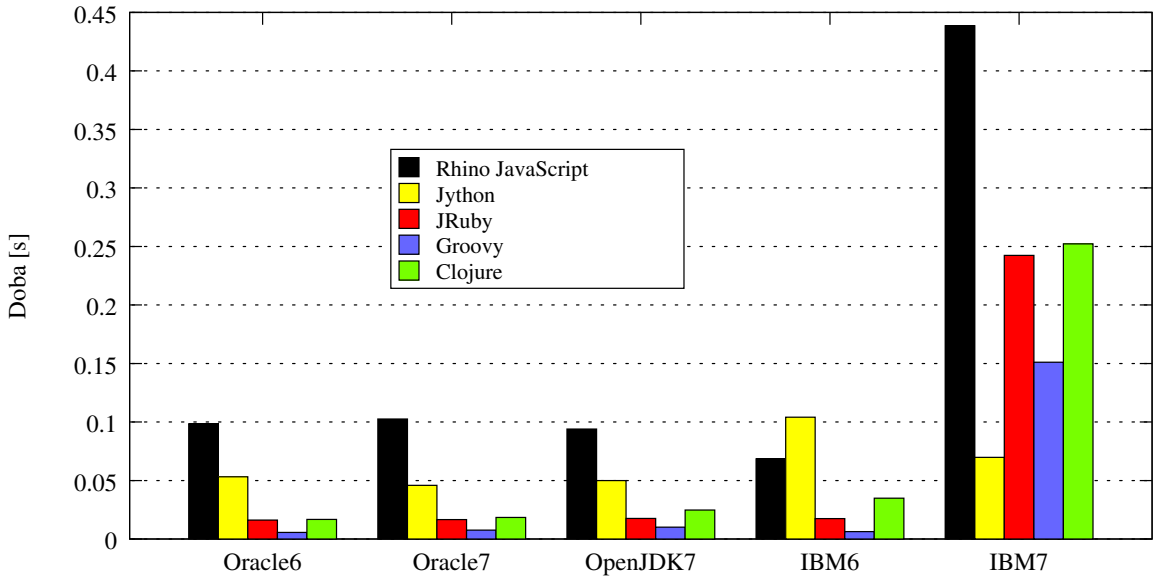
Označení v grafu	**Verze JVM (java -version)**
Oracle6	Java(TM) SE Runtime Environment (build 1.6.0_45-b06) Java HotSpot(TM) 64-Bit Server VM (build 20.45-b01, mixed mode)
Oracle7	Java(TM) SE Runtime Environment (build 1.7.0_07-b10) Java HotSpot(TM) 64-Bit Server VM (build 23.3-b01, mixed mode)
OpenJDK7	OpenJDK Runtime Environment (IcedTea7 2.3.3) (suse-55.1-x86_64) OpenJDK 64-Bit Server VM (build 23.2-b09, mixed mode)
IBM6	Java(TM) SE Runtime Environment (build pxa6460sr13fp2-20130424_01(SR13 FP2)) IBM J9 VM (build 2.4, JRE 1.6.0 IBM J9 2.4 Linux amd64-64 jvmxa6460sr13fp2-20130423_146146 (JIT enabled, AOT enabled)
IBM7	Java(TM) SE Runtime Environment (build pxa6470sr4fp2-20130426_01(SR4 FP2)) IBM J9 VM (build 2.6, JRE 1.7.0 Linux amd64-64 Compressed References 20130422_146026 (JIT enabled, AOT enabled)

Sada testů *maxi*

Popis testu	Do skriptu je předáno jednorozměrné pole hodnot typu *integer*. V tomto poli je následně pomocí cyklu nalezen index na kterém se nachází největší hodnota pole. Sleduje se doba provedení skriptu.
Implementace	JavaScript, Python, Ruby, Groovy, Clojure
Zaměření	Předávání jednorozměrného pole do skriptu pomocí JSR 223 API
Vyhodnocení	Při prvním běhu, jehož výsledky jsou zobrazeny v grafu 4.1, lze pozorovat nezávisle na verzi JVM/JRE vyrovnaný čas všech jazyků běžících nad HotSpot JVM, s výjimkou JRuby, které dosáhlo lepších výsledků nad JVM HotSpot v Oracle JRE 6. Všechny jazyky vykázaly nejlepší čas na IBM J9 pro JRE 6. Nejhorších výsledků bylo dosaženo s použitím IBM J9 v JRE 7 – až na JRuby a Clojure, které měly srovnatelné výsledky s JVM HotSpot, byly ostatní skripty nezanedbatelně pomalejší. V dalších bězích (graf 4.2) opět dosáhly vyrovnaného výsledku JRE využívající HotSpot JVM. Tyto časy také představovaly s výjimkou Rhino nejlepší naměřené výsledky. Rhino bylo rychlejší nad JVM IBM J9 pro JRE 6 přibližně o 25% oproti HotSpot libovolné verze. Jython byl nad IBM J9 JRE 6 nejpomalejší. Nejhorších výsledků dosáhlo opět IBM J9 v JRE 7 – pouze Jython dosáhl času srovnatelného s HotSpot JVM, ostatní jazyky dosáhly řádově vyšších časů než na ostatních JVM.



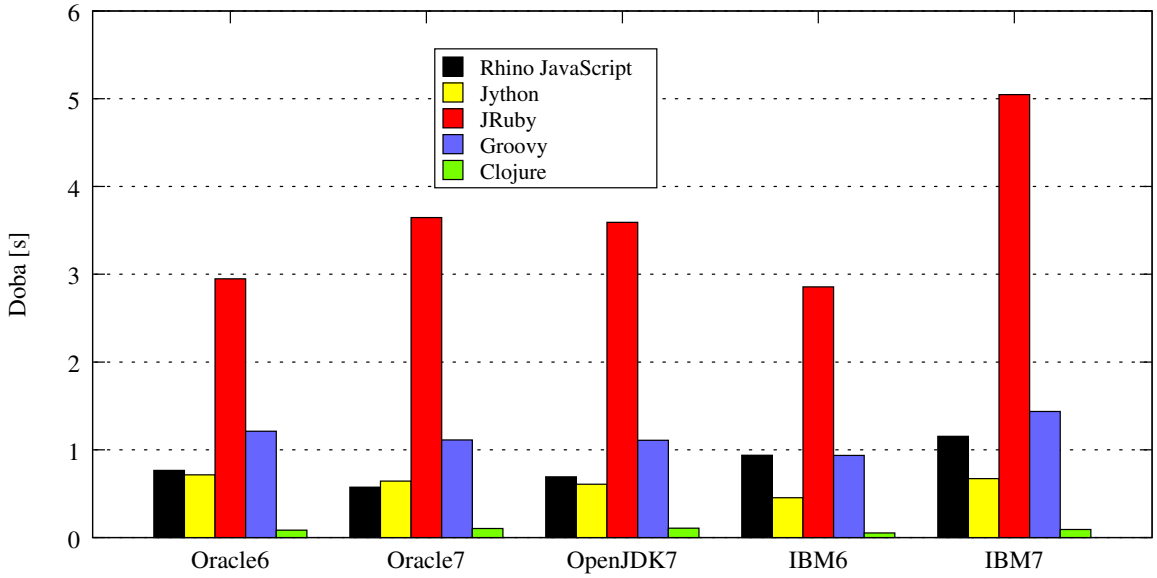
Obrázek 4.1: Sada testů *maxi* – první běh



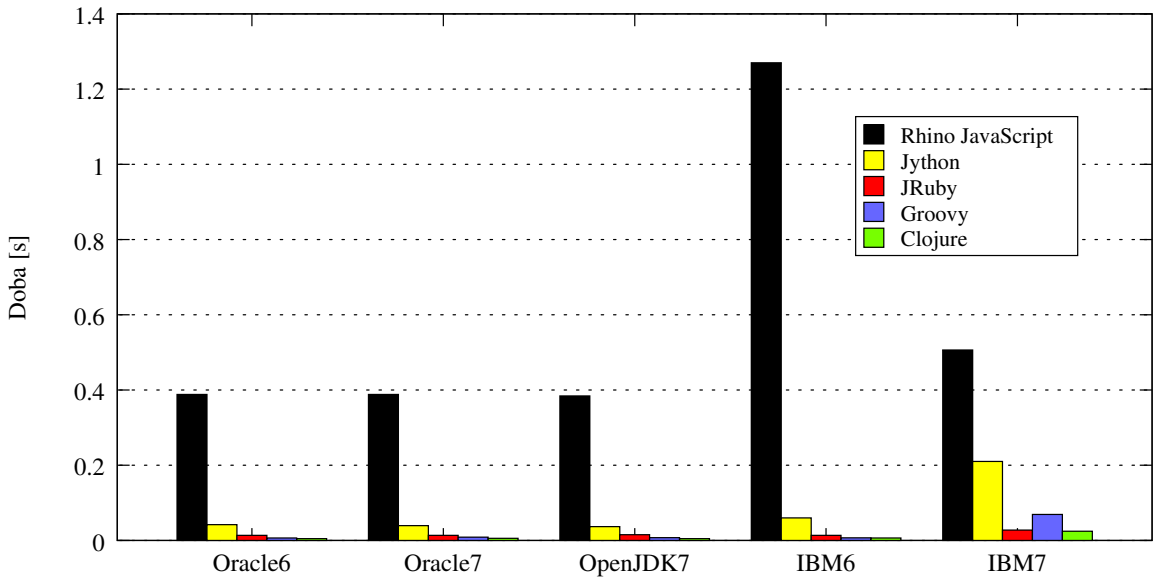
Obrázek 4.2: Sada testů *maxi* – další běhy

Sada testů *sum*

Popis testu	Do skriptu je předáno jednorozměrné pole hodnot typu *float*. Hodnoty v tomto poli jsou následně pomocí cyklu sečteny. Sleduje se doba provedení skriptu.
Implementace	JavaScript, Python, Ruby, Groovy, Clojure
Zaměření	Předávání jednorozměrného pole do skriptu pomocí JSR 223 API
Vyhodnocení	Při prvním běhu (graf 4.3) dosáhly jazyky nad všemi JRE srovnatelných výsledků. Nejmenší rozdíly byly mezi JRE využívajícími HotSpot JVM. Největší rozdíl lze pozorovat u jazyku JRuby nad IBM J9 mezi verzí pro JRE 6 a JRE 7, kde rozdíl činí přibližně 40% v neprospěch JRE 7. Na grafu 4.4 s výsledky dalších běhů je patrná i značná optimalizace za běhu JVM, která je nejvýznačnější u jazyku JRuby a nejméně patrná u jazyku Rhino (oproti grafu 4.3). Nad JVM IBM J9 v JRE 6 dosáhlo Rhino horšího času než při prvním běhu – tento fenomén může souviset s neefektivní optimalizací a agresivním chováním *garbage collector*u.



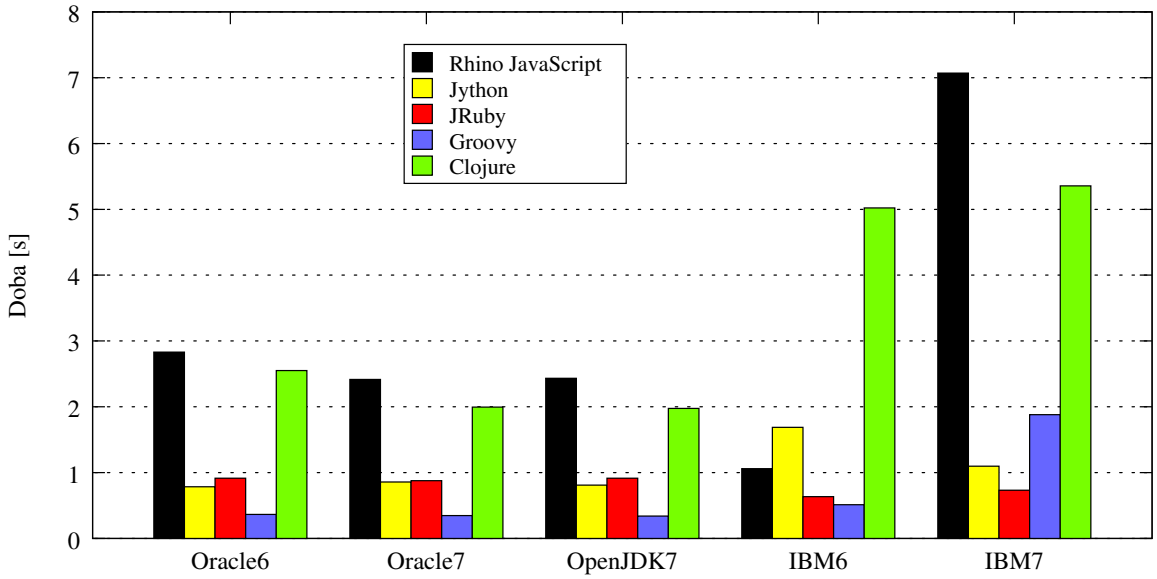
Obrázek 4.3: Sada testů *sum* – první běh



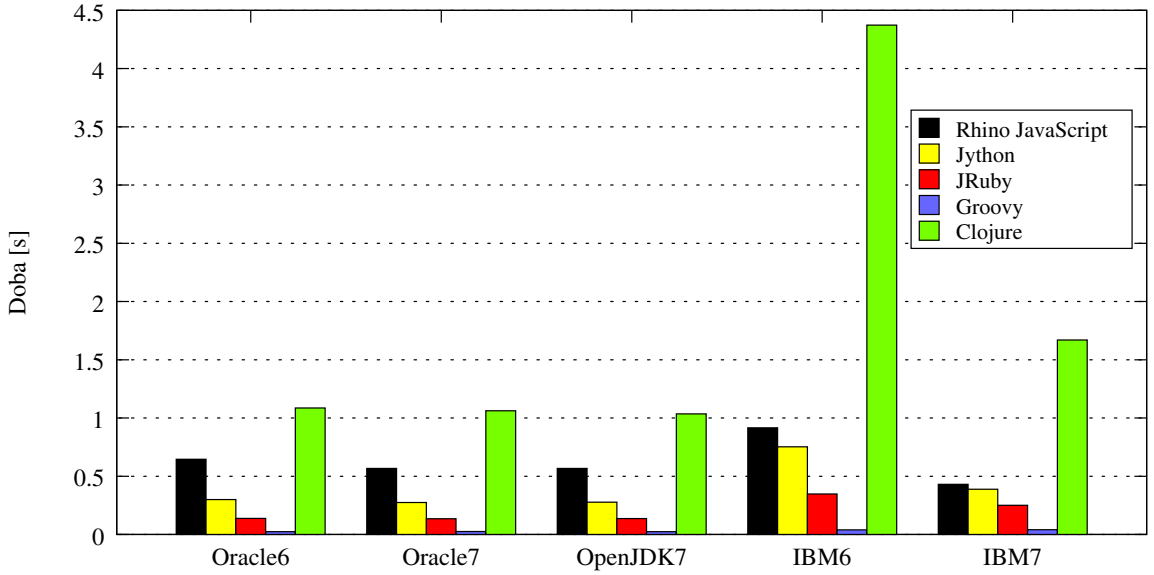
Obrázek 4.4: Sada testů *sum* – další běhy

Sada testů *2dtreshold*

Popis testu	Do skriptu je předáno dvourozměrné pole náhodných hodnot typu *integer*. Položky v tomto poli jsou následně změněny na hodnoty `0` nebo `1` pomocí prahování. Sleduje se doba provedení skriptu.
Implementace	JavaScript, Python, Ruby, Groovy, Clojure
Zaměření	Předávání dvourozměrného pole do skriptu pomocí JSR 223 API
Vyhodnocení	V prvním běhu (graf 4.5) dosáhly HotSpot pro JRE 7 (Oracle i OpenJDK) téměř shodných výsledků. Rhino a Clojure nad HotSpot v JRE 6 byly ve srovnání s JRE 7 HotSpot o 0.5s pomalejší. Rhino dosáhlo absolutně nejlepšího výsledku nad IBM J9 v JRE 6 a absolutně nejhoršího s IBM J9 pro JRE 7. V dalších bězích (graf 4.6) se vlivem optimalizací JVM srovnaly výsledky JVM HotSpot v různých JRE. Nad IBM J9 JRE 6 dosahovaly výsledky zhruba dvojnásobných naměřených hodnot oproti ostatním JVM s výjimkou jazyku Clojure, kde byl rozdíl více než čtyřnásobný. Výsledky IBM J9 JRE 7 byly srovnatelné s výsledky získanými pomocí HotSpot JVM, s výjimkou Clojure, jež dosáhlo horšího času.



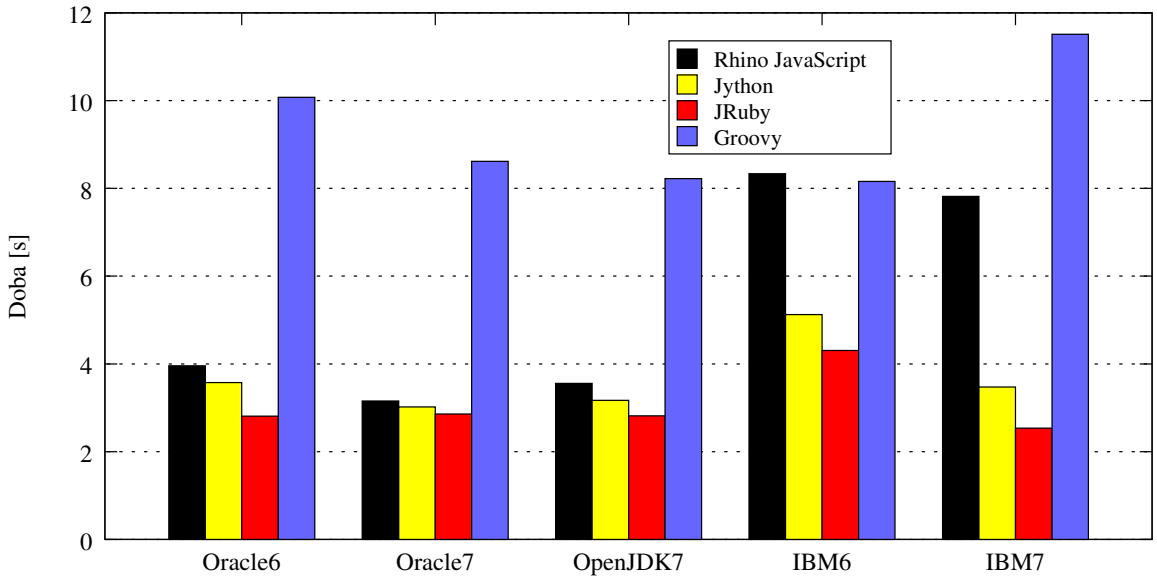
Obrázek 4.5: Sada testů *2dtreshold* – první běh



Obrázek 4.6: Sada testů *2dtreshold* – další běhy

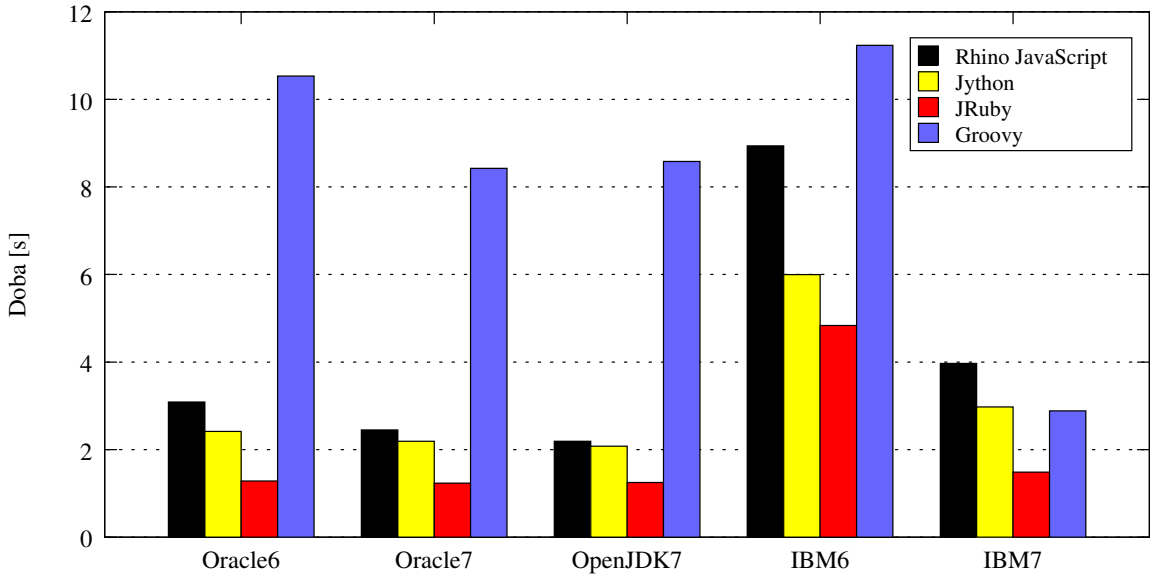
Sada testů *lsystem*

Popis testu	Do skriptu je předán objekt třídy `HashMap` jazyku Java. Hodnoty a klíče uložené v této mapě slouží jako přepisovací pravidla pro *Lindenmayerův systém*[1]. Skript obsahuje funkci `generate`, která pro předaný řetězec, mapu přepisovacích pravidel a počet iterací vygeneruje řetězec odpovídající požadované iteraci L-Systému. Sleduje se doba provedení skriptu.
Implementace	JavaScript, Python, Ruby, Groovy
Zaměření	Předávání objektu jazyku Java do skriptu pomocí JSR 223 API, invokace funkce, spojování řetězců
Vyhodnocení	V prvním běhu (graf 4.7) dosáhly JVM HotSpot srovnatelných výsledků. Nejpomalejší běh byl zaznamenán na IBM J9 JRE 6, pouze jazyk Groovy byl v tomto prostředí srovnatelný s HotSpot v JRE 7. IBM J9 JRE 7 vykázalo horší výsledky pro Rhino a Groovy, ale velmi podobné pro Jython a JRuby ve srovnání s HotSpot JRE 7. V dalších bězích (graf 4.8) došlo u HotSpot k mírné optimalizace u jazyků Rhino, JRuby a Jython, jazyk Groovy zůstal bez výrazných změn. U IBM J9 JRE 6 došlo ke zvýšení času potřebného k dokončení úlohy u všech měřených jazyků. U IBM J9 JRE 7 došlo k výrazné optimalizaci a snížení doby běhu u jazyků Rhino, Groovy a JRuby.



Obrázek 4.7: Sada testů *lsystem* – první běh

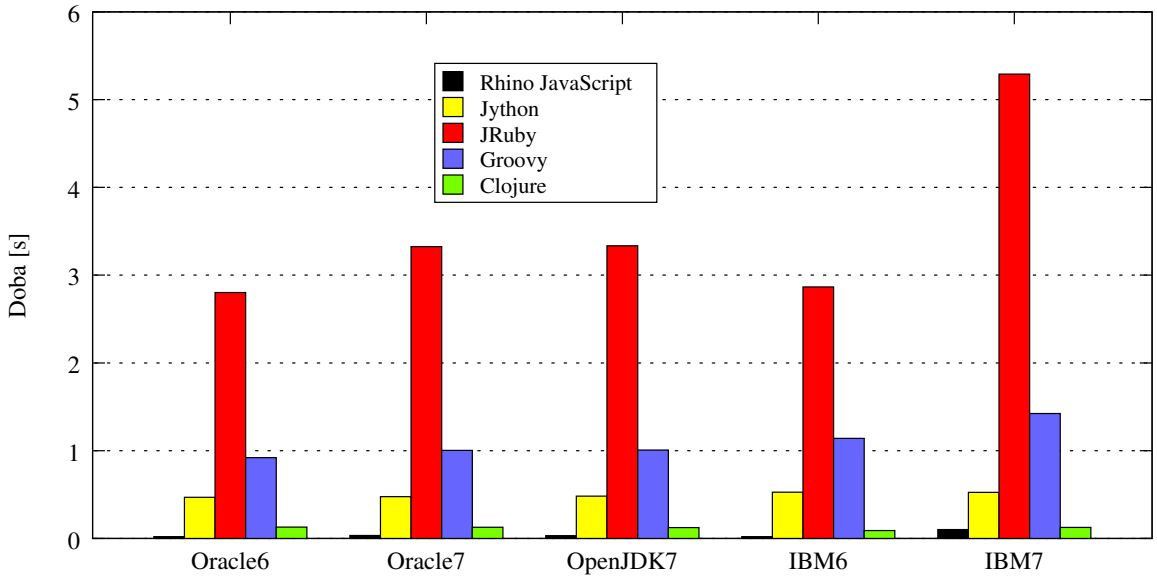
[1]Lindenmayerův systém je variantou formální gramatiky používané ke generování soběpodobných křivek a fraktálních útvarů, využitelný například k simulaci růstu rostlin.



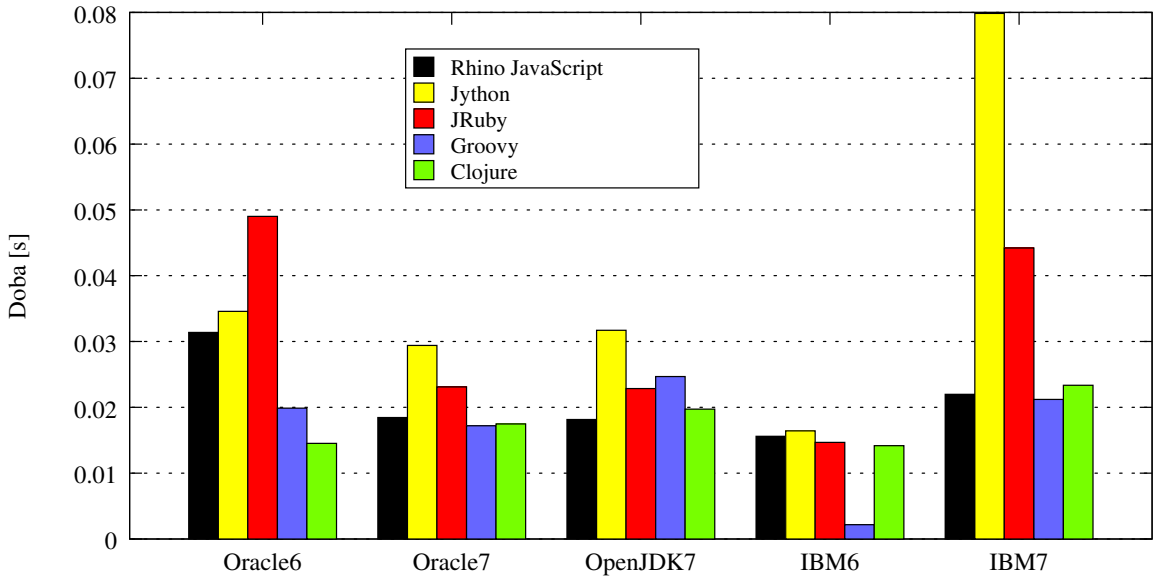
Obrázek 4.8: Sada testů *lsystem* – další běhy

Sada testů *recursion*

Popis testu	Skript obsahuje funkci, která provádí vysoký počet rekurzivních volání sebe sama. Sleduje se doba provedení skriptu.
Implementace	JavaScript, Python, Ruby, Groovy, Clojure
Zaměření	Volání funkcí, rekurze
Vyhodnocení	Při prvním běhu (graf 4.9) dosahovaly veškeré jazyky shodných výsledků. Jedinou výjimku tvořil jazyk JRuby nad IBM J9 JRE 7, jehož běh trval přibližně o 40% času déle. Při následných bězích (graf 4.10) dosahovalo nejlepších výsledků IBM J9 JRE 6. Výsledky JVM HotSpot JRE 7 byly až na malé rozdíly spolu srovnatelné. JVM HotSpot JRE 6 dosahovalo podobných časů pro jazyky JRuby, Clojure a Groovy jako IBM J9 JRE 7. Nejpomalejším výsledkem byl běh Jython na IBM J9 JRE 7.
Poznámky	Pro úspěšné spuštění testů je nutno navýšit velikost zásobníku JVM pomocí volby `-Xss`. Testy byly prováděny s parametrem `-Xss4m`.



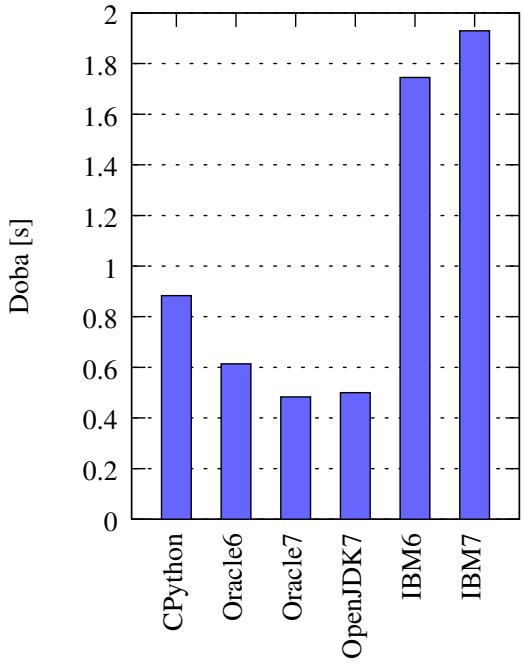
Obrázek 4.9: Sada testů *recursion* – první běh



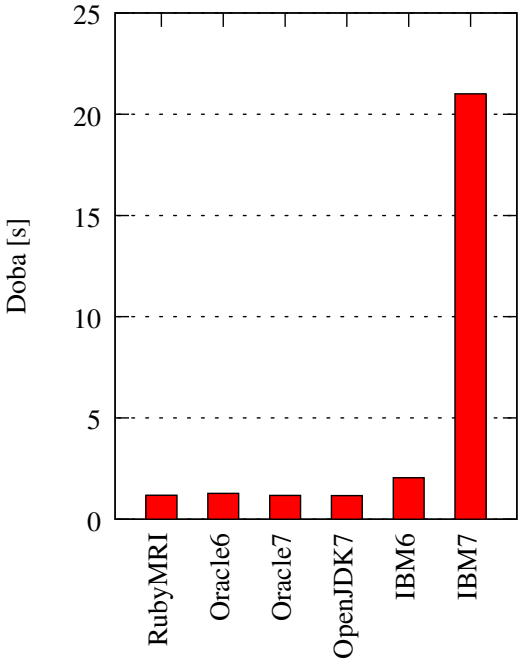
Obrázek 4.10: Sada testů *recursion* – další běhy

Test *exception*

Popis testu	Skript obsahuje proceduru volající funkci s vysokým počtem rekurzivních zanoření, při nejhlubším zanoření je vyvolána výjimka. Tato výjimka je odchycena až ve volající proceduře. Sleduje se doba provedení skriptu nad JVM a pomocí oficiálních virtuálních strojů použitých jazyků.
Implementace	Python, Ruby
Zaměření	Výjimky, rekurze
Vyhodnocení	Ve výsledcích testu pro jazyk Python uvedených v grafu 4.11 lze vidět, že při běhu nad JVM HotSpot bylo dosaženo přibližně o třetinu lepšího času než při běhu skriptu nad virtuálním strojem CPython. Naopak v případě běhu nad IBM J9 bylo pro obě jeho verze dosaženo výrazně horších výsledků – doba běhu byla přibližně dvakrát delší něž v případě běhu nad CPython. V testech jazyku Ruby (graf 4.12) bylo v případě běhu nad JVM HotSpot dosaženo srovnatelných výsledků s Ruby MRI. Při běhu nad IBM J9 pro JRE 6 byla doba běhu vyšší přibližně dvakrát, na IBM J9 pro JRE 7 byl dosažený čas několikanásobně delší.
Poznámky	Pro úspěšné spuštění testů je nutno navýšit velikost zásobníku JVM pomocí volby `-Xss`. Testy byly prováděny s parametrem `-Xss4m`.



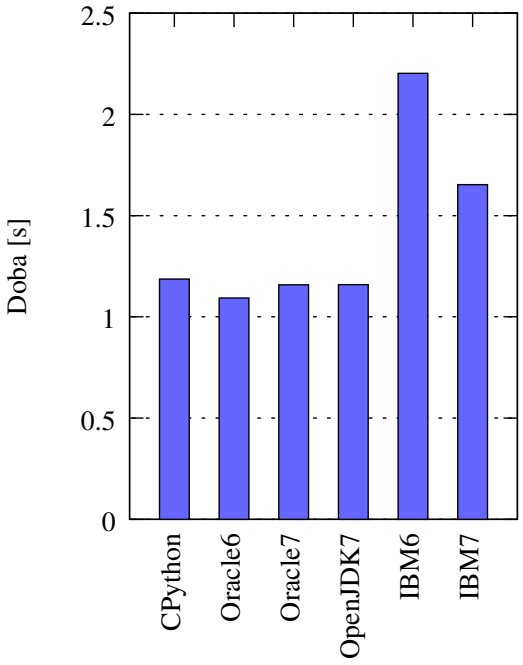
Obrázek 4.11: Test *exception* – Python



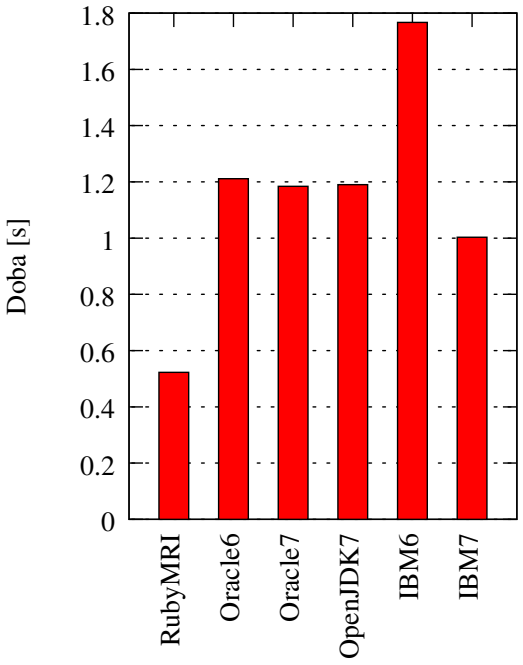
Obrázek 4.12: Test *exception* – Ruby

Test *files*

Popis testu	Skript otevírá postupně velké množství souborů, každý soubor po řádku zpracuje (přečte číslo na daném řádku) a největší nalezené číslo uloží do kolekce. Sleduje se doba provedení skriptu nad JVM a pomocí oficiálních virtuálních strojů použitých jazyků.
Implementace	Python, Ruby
Zaměření	Vstup/Výstup, soubory
Vyhodnocení	V testech jazyku Python, jejichž výsledky jsou znázorněny v grafu 4.13, lze pozorovat srovnatelný čas při běhu nad JVM HotSpot a CPython. Doba běhu pro IBM J9 pro JRE 7 byla o polovinu vyšší, pro IBM J9 pro JRE 7 byl nárůst času ještě výraznější. Ve výsledcích testu pro jazyk Ruby, které jsou uvedeny v grafu 4.14, lze vidět srovnatelné výsledky pro JVM HotSpot, časy běhu nad tímto virtuálním strojem byly ale přibližně dvakrát delší než u běhu nad Ruby MRI. Nejhorších býsledků dosáhl skript při běhu nad IBM J9 pro JRE 6, kde dosáhl trojnásobného času oproti Ruby MRI. Doba běhu nad IBM J9 pro JRE 7 byla nižší než v případě JVM HotSpot, ovšem stále vyšší než na Ruby MRI.
Poznámky	K testu je nutná sada `N` testovacích souborů, pojmenovaných `test{N}.txt`, kde `{N}` značí celé číslo z intervalu $\langle 0, 1000)$. Tyto soubory je možno vygenerovat přiloženým skriptem.



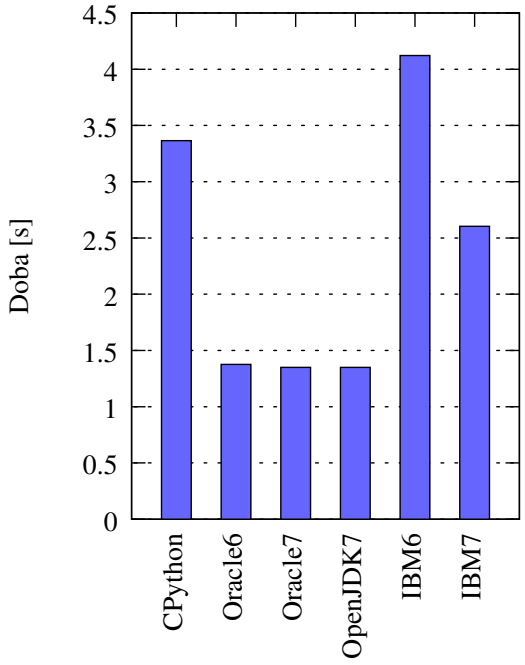
Obrázek 4.13: Test *files* – Python



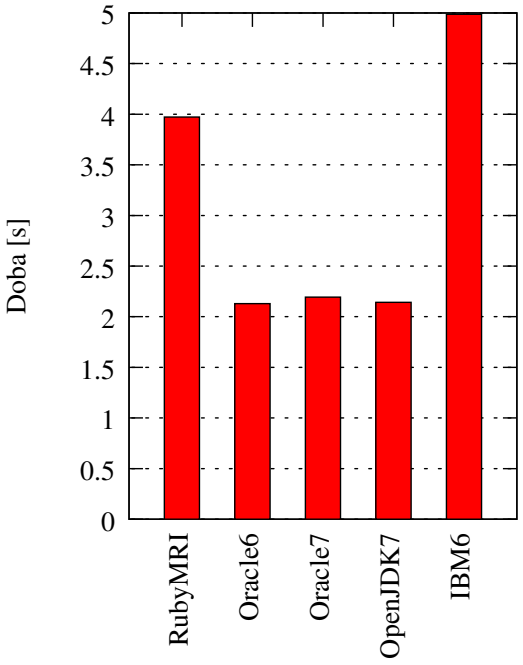
Obrázek 4.14: Test *files* – Ruby

Test *fncall*

Popis testu	Skript provádí iterativní výpočet hodnoty π dle rekurentního vztahu. Jednotlivé dílčí výpočty jsou prováděny samostatnými funkcemi volanými z cyklu provádějící výpočet. Sleduje se doba provedení skriptu nad JVM a pomocí oficiálních virtuálních strojů použitých jazyků.
Implementace	Python, Ruby
Zaměření	Volání funkcí
Vyhodnocení	V testu jazyku Python (graf 4.15) dosáhly nejnižších časů běhy na JVM HotSpot, běh nad CPython byl více než dvakrát pomalejší. Doba běhu nad IBM J9 pro JRE 7 byla dvakrát delší než při běhu nad JVM HotSpot. Nejhorších výsledků dosáhl běh nad IBM J9 pro JRE 6. V případě testu v jazyce Ruby, jehož výsledky lze vidět v grafu 4.16, dosáhly taktéž nejlepších výsledků běhy nad JVM HotSpot, které byly přibližně dvakrát rychlejší než běh nad Ruby MRI. Nejhorší výsledky vykazoval běh nad IBM J9 pro JRE 6.
Poznámky	Běh nad IBM J9 pro JRE 7 pro jazyk Ruby není ve výsledcích zahrnut, protože v něm došlo k chybě, jejíž příčinu se nepodařilo odhalit.



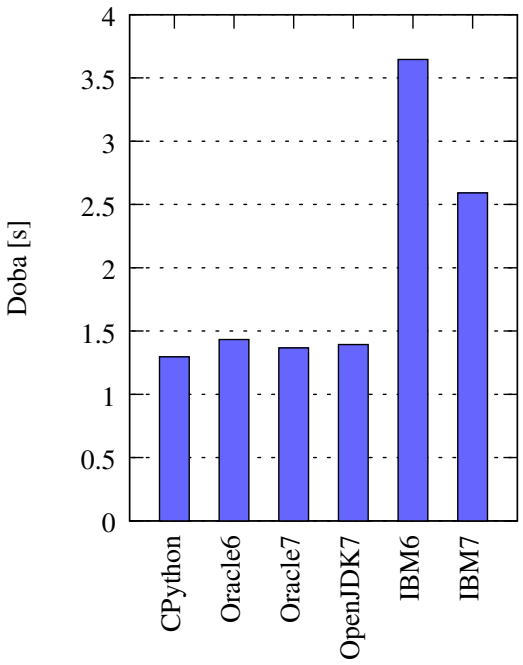
Obrázek 4.15: Test *fncall* – Python



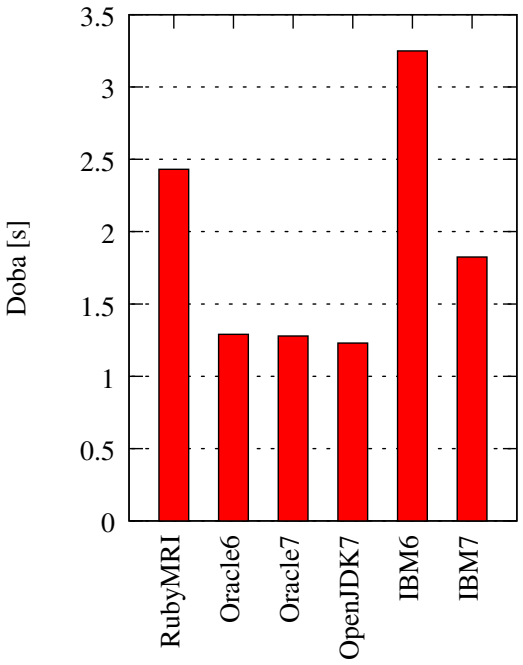
Obrázek 4.16: Test *fncall* – Ruby

Test *garbage*

Popis testu	Skript vytváří v průběhu svého provádění datové struktury, na které poté ztratí referenci. Takto ztracená datová struktura pak může být odstraněna z paměti *garbage collectorem*. Sleduje se doba provedení skriptu nad JVM a pomocí oficiálních virtuálních strojů použitých jazyků.
Implementace	Python, Ruby
Zaměření	Garbage collector
Vyhodnocení	V grafu 4.17 pro jazyk Python lze vidět vyrovnané časy běhu pro CPython a JVM HotSpot. Doba běhu nad IBM J9 pro JRE 6 byla více než dvakrát delší, běh nad IBM J9 pro JRE 7 trval přibližně dvakrát déle než u JVM HotSpot. Pro test v jazyce Ruby, jehož výsledky jsou znázorněny v grafu 4.18, rovněž platí vyrovnané výsledky pro běhy nad JVM HotSpot. Tyto běhy dosáhly poloviního čas v porovnání s během nad Ruby MRI. Nejhorších výsledků dosáhl běh nad IBM J9 pro JRE 6, výsledky pro tento virtuální stroj pro JRE 7 byly dobou svého běhu mezi JVM HotSpot a Ruby MRI.



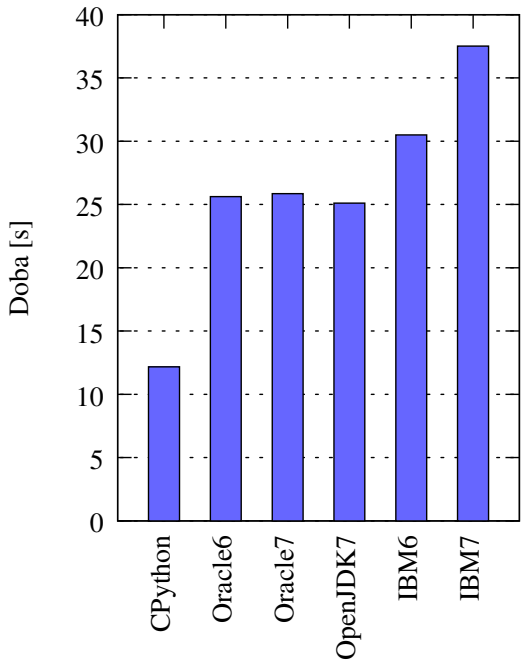
Obrázek 4.17: Test *garbage* – Python



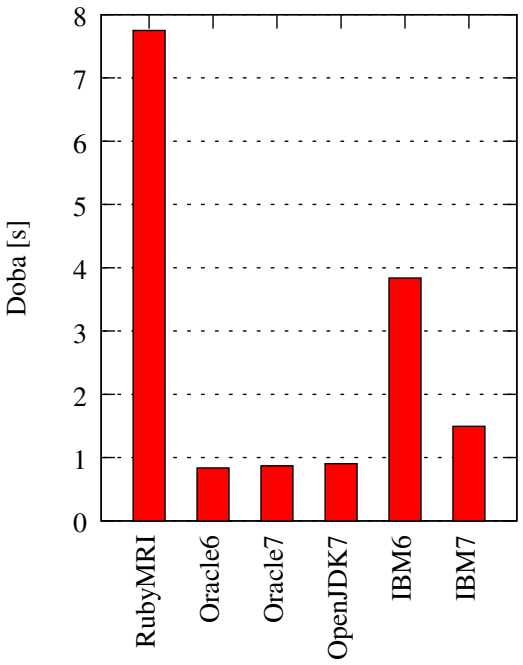
Obrázek 4.18: Test *garbage* – Ruby

Test *primes*

Popis testu	Skript pomocí algoritmu Eratosthenovo síto nalezne všechna prvočísla až do zadaného maxima a nalezené hodnoty uloží do datové struktury typu seznam. Sleduje se doba provedení skriptu nad JVM a pomocí oficiálních virtuálních strojů použitých jazyků.
Implementace	Python, Ruby
Zaměření	Operace nad datovou strukturou typu seznam/list
Vyhodnocení	V testu jazyku Python (graf 4.17) lze vidět, že nejnižšího času dosáhl běh nad CPython, následován běhy nad JVM HotSpot, které dosáhly přibližně dvojnásobných časů. Běh nad IBM J9 pro JRE 6 byl o 5s pomalejší, nejhorších výsledků dosáhl běh nad IBM J9 pro JRE 7. V testu jazyku Ruby, jehož výsledky lze vidět v grafu 4.20, dosáhl jednoznačně nejhorších výsledků běh nad Ruby MRI, který trval dvakrát déle něž druhý nejpomalejší běh nad IBM J9 pro JRE 6. Nejlepších výsledků dosáhly běhy nad JVM HotSpot, které byly přibližně osmkrát rychlejší než Ruby MRI. Běh nad IBM J9 pro JRE 7 trval přibližně o polovinu déle než nad JVM HotSpot.



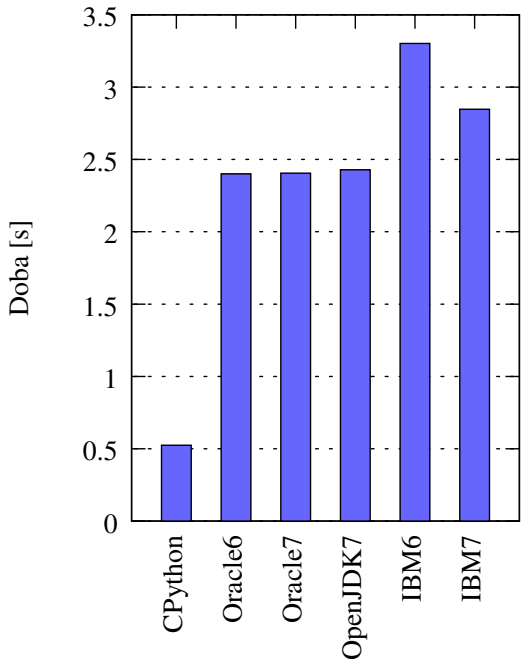
Obrázek 4.19: Test *primes* – Python



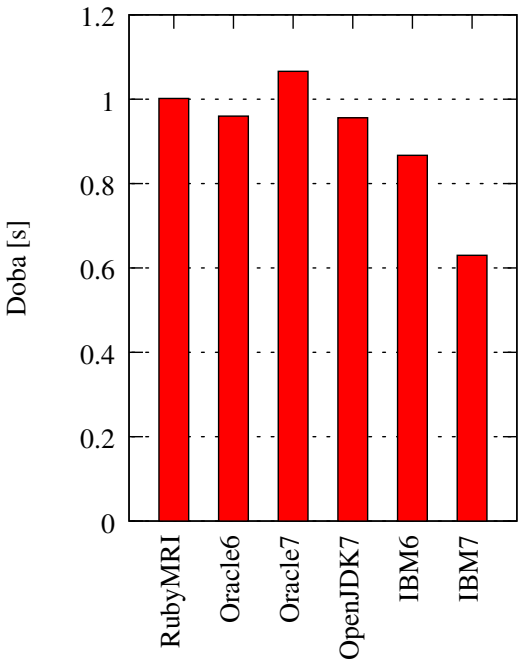
Obrázek 4.20: Test *primes* – Ruby

Test *regexp*

Popis testu	Skript provádí aplikaci regulárního výrazu na rozsáhlý řetězec. Sleduje se doba provedení skriptu nad JVM a pomocí oficiálních virtuálních strojů použitých jazyků.
Implementace	Python, Ruby
Zaměření	Regulární výrazy
Vyhodnocení	V tomto testu pro jazyk Python (graf 4.21) dosáhl výrazně nejlepších výsledků běh nad CPython, druhé nejlepší výsledky poskytovaly běhy nad JVM HotSpot, které ale trvaly skoro čtyřikrát déle. Běh nad J9 pro JRE 6 byl ještě asi o čtvrtinu delší a měl nejhorší výsledky. Výsledky pro běh nad IBM J9 pro JRE 7 byly druhé nejhorší. V testu pro jazyk Ruby (graf 4.22) dosahovaly běhy pro Ruby MRI a JVM HotSpot srovnatelných výsledků, mírně lepších výsledků dosáhnul běh nad J9 pro JRE 6. Nejrychlejšího průběhu dosáhnul test nad IBM J9 pro JRE 7.
Poznámky	Jako testovací textový řetězec je použita kniha uvolněná pod svobodnou licencí v rámci projektu Gutenberg[2]. Soubor obsahující knihu je přiložen ve složce obsahující data pro testy.



Obrázek 4.21: Test *regexp* – Python

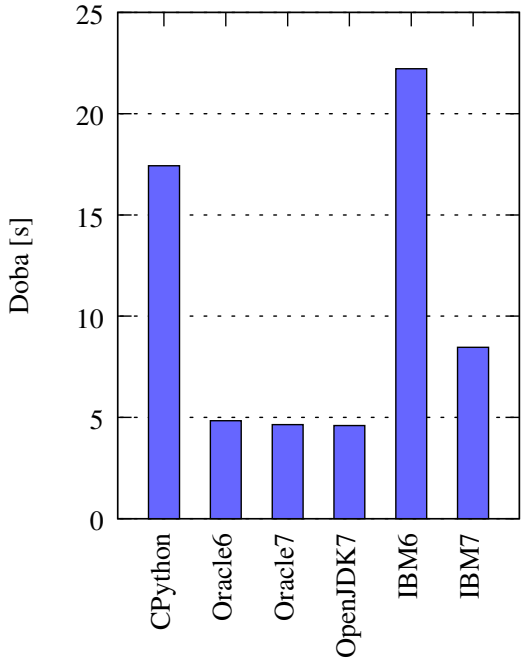


Obrázek 4.22: Test *regexp* – Ruby

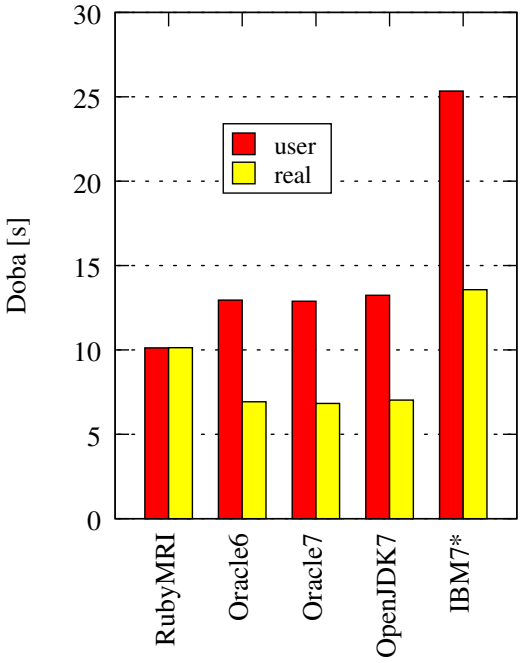
[2] http://www.gutenberg.org/

Test *thread*

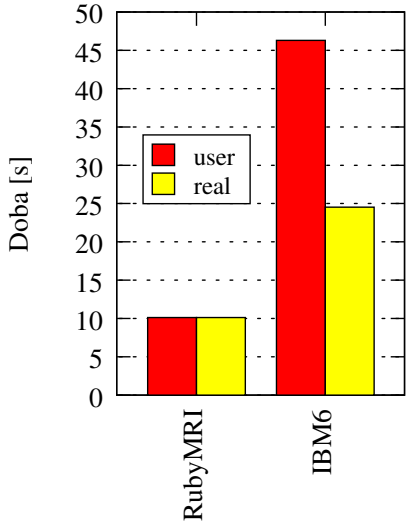
Popis testu	Skript vytvoří osm vláken, které paralelně provádějí v cyklu jednoduchý výpočet. Test je považován za dokončený, jakmile všechna vlákna dokončí svou úlohu. Sleduje se doba provedení skriptu nad JVM a pomocí oficiálních virtuálních strojů použitých jazyků.
Implementace	Python, Ruby
Zaměření	Paralelismus, *Global Interpreter Lock*
Vyhodnocení	Ve výsledcích testu pro jazyk Python, které jsou znázorněny v grafu 4.23, lze vidět, že nejlepších výsledků dosahovaly běhy nad JVM HotSpot, následované během nad IBM J9 pro JRE 7. Běh nad Cpython vykazoval několikanásobně horší výsledky, stejně jako běh nad IBM J9 pro JRE 6, který byl ještě o něco pomalejší. Ve výsledcích testu pro jazyk Ruby (graf 4.24) jsou uvedeny vzhledem k testování na dvoujádrovém procesoru (viz 4.1) vždy dva časy – čas vykonávání kódu a reálný čas běhu. (Pro jazyk Python zjištění času vykonávání kódu s ohledem na možnosti použité knihovny `timeit` není možné.) Při srovnání podle reálného času nejlepších výsledků dosáhly běhy nad JVM HotSpot, při běhu nad IBM J9 pro JRE 7 byl čas přibližně dvojnásobný. Běh nad IBM J9 pro JRE 6 trval několikanásobně delší dobu. Při běhu nad IBM J9 pro JRE 7 došlo vždy po osmé iteraci testu k chybě, jejíž příčinu se nepodařilo zjistit, proto jsou v grafu uvedeny časy pro tuto osmou iteraci.
Poznámky	Interprety CPython a Ruby MRI využívají pro ochranu kritických sekcí virtuálního stroje mechanismus nazvaný *Global Interpreter Lock* – v důsledku jeho použití však může v těchto virtuálních strojích běžet pouze jedno vlákno v jeden čas (preemptivní paralelismus) i v případě, že v systému existují prostředky pro souběžný běh více vláken. JVM společnosti IBM i Oracle neobsahují mechanismus *Global Interpreter Lock* a umožňují reálný paralelismus. V tomto testu jsou tedy skripty spuštěné nad JVM zvýhodněny na systémech s prostředky pro souběžný běh více vláken.



Obrázek 4.23: Test *thread* – Python



Obrázek 4.24: Test *thread* – Ruby



Obrázek 4.25: Test *thread* – Ruby, IBM6

Kapitola 5

Závěr

Cílem práce bylo srovnat možnosti použití skriptovacích jazyků odpovídajících normě JSR 223 na platformě Java. V práci je zahrnut popis aplikačního rozhraní daného touto normou. Součástí práce je diskuze o problematice použití skriptovacích jazyků na platformě Java a návrh řešení některých problémů. V práci jsou také popsána některá běhová prostředí jazyka Java. V rámci práce byla implementována sada testů pro porovnávání výkonnosti jednotlivých jazyků a jednoduché prostředí pro spouštění těchto testů. Dále pak práce obsahuje výsledky měření výkonnosti skriptovacích jazyků běžících nad JVM. Z těchto výsledků lze odvodit, že neexistuje výrazný rozdíl z pohledu výkonnosti skriptovacích jazyků mezi distribucemi jazyku Java využívající HotSpot JVM. Měření také ukázala, že implementace existujících jazyků na platformě Java mohou být rychlejší než nativní interpretery těchto jazyků.

Podpora skriptovacích jazyků na platformě Java je stále vyvíjena a vznikají pro ni nová rozšíření, jako například norma JSR 292 pro podporu dynamicky typovaných jazyků v JRE 7. Skriptovací jazyky pro platformu Java jsou rovněž aktivně vyvíjeny. Měření tvořící součást této práce prokázalo, že použití těchto jazyků je při určitých úkolech výhodnější a nabízejí tak vhodnou alternativu. Využití skriptovacích jazyků běžících nad JVM jako součást Java aplikací může zkrátit dobu vývoje a přinést vývojářům další výhody v podobě volnější syntaxe oproti jayzku Java a vyšší přenositelnosti těchto skriptů.

Literatura

[1] *Scripting for the JavaTM Platform [online]*. Sun Microsystems, Inc., 2006-12-11 [cit. 2013-01-14].
URL http://www.jcp.org/en/jsr/detail?id=223

[2] *Java Scripting Programmer's Guide [online]*. Oracle, 2011 [cit. 2013-01-21].
URL http://docs.oracle.com/javase/6/docs/technotes/guides/scripting/programmer_guide/index.html

[3] *Package javax.script [online]*. Oracle, 2011 [cit. 2013-01-22].
URL http://docs.oracle.com/javase/6/docs/api/javax/script/package-summary.html

[4] *New in Rhino 1.7R3 [online]*. Mozilla, 2012-05-13 [cit. 2013-05-01].
URL https://developer.mozilla.org/en-US/docs/New_in_Rhino_1.7R3

[5] *Differences Between Mri And Jruby [online]*. 2012-06-20 [cit. 2013-04-08].
URL https://github.com/jruby/jruby/wiki/DifferencesBetweenMriAndJruby

[6] *Rhino Scripting Java [online]*. Mozilla, 2012-11-22 [cit. 2013-04-02].
URL https://developer.mozilla.org/en-US/docs/Scripting_Java

[7] *Calling Java From JRuby [online]*. 2013-04-04 [cit. 2013-04-26].
URL https://github.com/jruby/jruby/wiki/CallingJavaFromJRuby

[8] *IBM J9 Virtual Machine [online]*. IBM, Oracle, 2013 [cit. 2013-04-22].
URL http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/index.jsp?topic=%2Fcom.ibm.java.lnx.70.doc%2Fuser%2Fjava_jre.html

[9] *Java Platform Standard Edition 7 Documentation [online]*. Oracle, 2013 [cit. 2013-05-07].
URL http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime()

[10] *Differences between CPython and Jython [online]*. [cit. 2013-04-06].
URL http://www.jython.org/archive/21/docs/differences.html

[11] *Frequently Asked Questions About the Java HotSpot VM [online]*. Oracle, [cit. 2013-04-17].
URL http://www.oracle.com/technetwork/java/hotspotfaq-138619.html

[12] *The Java HotSpot Performance Engine Architecture [online]*. Oracle, [cit. 2013-04-26].
URL http://www.oracle.com/technetwork/java/whitepaper-135217.html

[13] *Accessing Java from Jython [online]*. [cit. 2013-04-29].
URL http://www.jython.org/archive/21/docs/usejava.html

[14] Bosanac, D.: *Scripting in Java: Languages, Frameworks, and Patterns*. Pearson Education, 2007, ISBN 9780132702294.

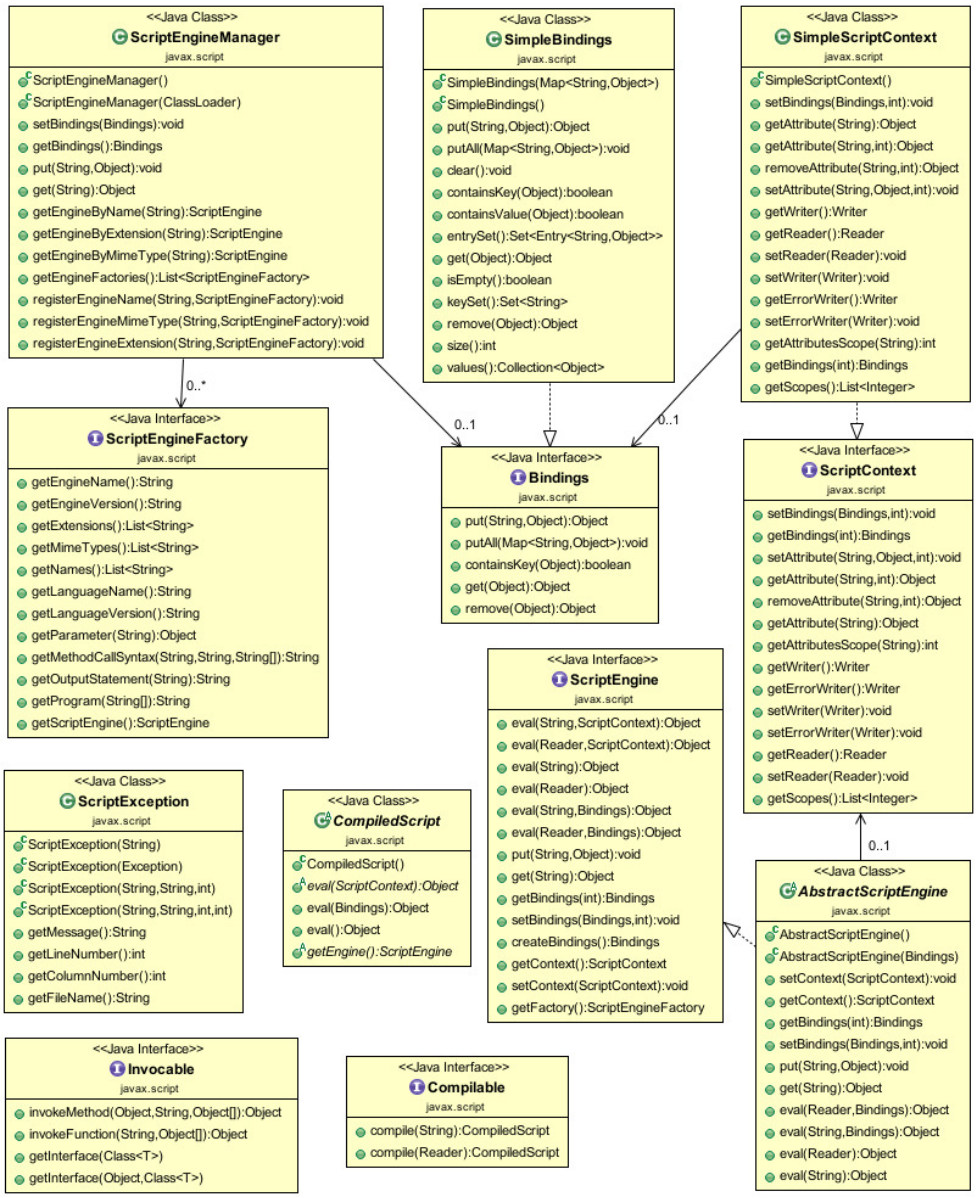
[15] Flanagan, D.: *JavaScript: The Definitive Guide*. O'Reilly Media, 2011, ISBN 9781449308162.

[16] Furman, S.: *Java Method Overloading and LiveConnect 3 [online]*. 2012-08-06 [cit. 2013-05-01].
URL http://www-archive.mozilla.org/js/liveconnect/lc3_method_overloading.html

[17] Thomas, D.; Fowler, C.; Hunt, A.: *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, 2004, ISBN 9780974514055.

Příloha A

Diagram tříd balíku javax.script



Obrázek A.1: Diagram tříd rozhraní JSR 223