Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Využití programovacího jazyka Kotlin pro vývoj mobilních aplikací na platformě Android

Bc. Petr Kuška

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Petr Kuška

Informatika

Název práce

Využití programovacího jazyka Kotlin pro vývoj mobilních aplikací na platformě Android

Název anglicky

Usage of Kotlin programming language for Android development

Cíle práce

Hlavním cílem práce je zhodnocení programovacího jazyka Kotlin společnosti JetBrains z hlediska využití pro vývoj mobilních aplikací na platformě Android. Součástí práce je porovnání jazyka Kotlin s programovacím jazykem Java, jak z hlediska funkčnosti a možností jazyků, tak z hlediska výkonu.
Mezi dílčí cíle patří:
- pohled na aktuální možnosti vývoje mobilních aplikací na platformě Android z hlediska programovacích jazyků, jejich historie a vývoje do budoucnosti
- představit programovací jazyky Kotlin a Java, jejich historii a vývoj, možnosti a metody použití
- vysvětlení základních principů vývoje aplikací na platformě Android
- porovnat implementaci těchto principů, čitelnost kódu a náročnost vypracování implementace na příkladu v obou zmíněných jazycích
- pomocí připravených scénářů a nástrojů otestovat výkon obou programovacích jazyků
- zhodnocení výsledku testování a doporučení
- závěr a predikce budoucího vývoje

Metodika

V teoretické části budou vysvětleny aktuální možnosti vývoje mobilních aplikací pro platformu Android z hlediska programovacích jazyků, jejich historie a vývoje. V dalších kapitolách budou představeny programovací jazyky Kotlin a Java, jejich vývoj, možnosti a metody použití. V praktické části budou na příkladu porovnány implementace obou programovacích jazyků a přístupy k základním principům vývoje mobilních aplikací platformy Android. Dále budou provedeny testy výkonnosti. Zhodnocení výsledků a doporučení bude věnována samostatná kapitola, která bude tvořit podklad pro závěrečnou část práce. V závěrečné části bude diskutována vhodnost použití programovacího jazyka Kotlin pro platformu Android a možný vývoj do budoucna.

Doporučený rozsah práce

50 – 60 stran

Klíčová slova

Kotlin, Java, Android, JetBrains, Oracle

Doporučené zdroje informací

BLUNDELL, Paul a MILANO, Diego Torres. Learning Android Application Testing. Birmingham : Packt Publishing, 2015. str. 314. ISBN 9781784397999

CHELL, Dominic, a další. The Mobile Application Hacker's Handbook. 1. Hoboken : John Wiley & Sons, Incorporated, 2015. str. 813. ISBN 9781118958513.

JEMEROV, Dmitry a Isakova, Svetlana. Kotlin in Action. 1. Greenwich : Manning Publications, 2017. str. 360. ISBN 978-1617293290.

KURNIAWAN, Budi. Java for Android. Quebec : Brainy Software, 2014. str. 567. ISBN 9780992133030.

KURNIAWAN, Budi. Java 7. Quebec : Brainy Software, 2014. str. 846. ISBN 9780980839661.

LEIVA, Antonio. Kotlin for Android Developers. Charleston : CreateSpace Independent Publishing Platform, 2016. str. 200. ISBN 978-1530075614.

SAMUEL, Stephen a BOCUTIO, Stefan. Programming Kotlin. Birmingham : Packt Publishing, 2017. str. 420. ISBN 978-1787126367.

Předběžný termín obhajoby

2017/18 LS – PEF

Vedoucí práce

Ing. Jiří Vaněk, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 22. 5. 2017

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 24. 5. 2017

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 27. 03. 2018

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Využití programovacího jazyka Kotlin pro vývoj mobilních aplikací na platformě Android" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 27. března 2018 _____

Poděkování

Rád bych touto cestou poděkoval vedoucímu práce Ing. Jiřímu Vaňkovi, Ph.D. za odborné konzultace a ochotu vést tuto práci. Také chci poděkovat svým rodičům a přátelům, kteří mě vytrvale podporují po dobu mých studií. Děkuji.

Využití programovacího jazyka Kotlin pro vývoj mobilních aplikací na platformě Android

---

Usage of Kotlin programming language for Android development

Souhrn

Tato práce se zabývá zhodnocením programovacího jazyka Kotlin společnosti JetBrains z hlediska využití pro vývoj mobilních aplikací na platformě Android. Součástí práce je porovnání jazyka Kotlin s programovacím jazykem Java, jak z hlediska funkčnosti a možností jazyků, tak z hlediska výkonu. Práce se zaměřuje zejména na problematiku testování výkonu. Definuje testovací scénáře a navrhuje nástroje vhodné pro jejich měření. Implementuje navržené nástroje a provádí měření, jehož výsledky následně analyzuje a interpretuje. Dále se práce věnuje aktuálním možnostem vývoje mobilních Android aplikací a detailně popisuje funkcionality jazyků Java a Kotlin.

Summary

The aim of this thesis is to evaluate JetBrain's programming language Kotlin from the perspective of developing mobile applications on Android. Part of the thesis is to compare the Kotlin language with the Java programming language in terms of functionality and performance. The paper focuses mainly on the issue of performance testing. It defines test scenarios and design tools appropriate for their measurement. It implements designed tools and performs measurements. The results are then analyzed and interpreted. In addition, the thesis focuses on the current options for development of mobile Android applications and describes in detail functionality of Java and Kotlin.

Klíčová slova:

Kotlin, Java, Android, testování výkonu, měření výkonu, benchmark, funkcionality

Keywords:

Kotlin, Java, Android, performance testing, measurement, benchmark, features

Obsah

1 Úvod

Systém Android má největší zastoupení na světě mezi všemi operačními systémy. V počtu aktivně používaných zařízení je dominantní platformou na trhu a ani součet užívaných zařízení všech ostatních platforem se mu zdaleka nevyrovná. Kromě telefonů, se Android nachází například v tabletech, televizích, ale dokonce i v hodinkách či automobilech.

Vývoj mobilních aplikací je typický svojí cílovou skupinou mladší generace. Ta klade nároky na svižnost a trendovost aplikací. Konkurence na trhu aplikací je vysoká, k červnu 2017 bylo v Obchodu Play přes 3 miliardy aplikací [1]. Dalším specifikem mobilního průmyslu jsou časté změny. Ať už je příčinou neustále přinášet uživatelům něco nového, nebo držet krok s konkurencí a moderními technologiemi, mobilní aplikace mají často kratší cyklus vydávání nových verzí – měsíce, někdy až týdny. To má za následek tlak na vývojová oddělení.

Autor práce je vývojářem Android aplikací a v oblasti vývoje mobilních aplikací má několikaleté zkušenosti. Motivací pro výběr tématu je rozšiřující se zájem o programovací jazyk Kotlin mezi komunitou vývojářů. O Kotlinu se začalo mluvit na konferencích a diskuzních fórech. Přednášky o funkcionalitách Kotlinu slibují nové přístupy k návrhu i implementaci a usnadnění údržby zdrojových kódů. Kotlin je Android komunitou dobře přijímán, avšak o jeho budoucnosti a postoji velkých společností jako je Google panují obavy.

Komplexní posouzení jazyka je náročným procesem. Existuje mnoho úhlů pohledu, které vyžadují výzkumy z různých oblastí věd. Práce si klade za cíl zhodnotit výkon programovacího jazyka Kotlin a porovnat jej s aktuálně nejrozšířenějším jazykem Java.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem práce je zhodnocení programovacího jazyka Kotlin společnosti JetBrains z hlediska využití pro vývoj mobilních aplikací na platformě Android. Součástí práce je porovnání jazyka Kotlin s programovacím jazykem Java, jak z hlediska funkčnosti a možností jazyků, tak z hlediska výkonu.

Mezi dílčí cíle patří:

- pohled na aktuální možnosti vývoje mobilních aplikací na platformě Android z hlediska programovacích jazyků, jejich historie a vývoje do budoucnosti
- popsat programovací jazyky Kotlin a Java, jejich historii a vývoj, možnosti a metody použití
- pomocí připravených scénářů a nástrojů otestovat výkon obou programovacích jazyků
- porovnat jazyky dle čitelnosti kódu a náročnosti vypracování implementace
- zhodnocení výsledku testování a doporučení
- závěr a predikce budoucího vývoje

2.2 Metodika

Metodika řešené práce je založena na analyticko-syntetickém přístupu. Práce rovněž čerpá ze zkušeností autora z oblasti vývoje mobilních aplikací v komerční sféře. V první fázi bude provedeno studium a analýza odborných informačních zdrojů. Na základě získaných poznatků budou v teoretické části popsány aktuální možnosti vývoje mobilních aplikací pro platformu Android z hlediska programovacích jazyků, jejich historie a vývoje. V dalších kapitolách budou představeny programovací jazyky Kotlin a Java, jejich vývoj, možnosti a metody použití. Poslední kapitola teoretické části bude věnována možnostem testování výkonu a nástrojů s ním spojeným.

V praktické části budou definovány testované vlastnosti jazyků a scénáře, jakými budou vlastnosti testovány. Pro testování výkonu bude využita autorem zvolená metodika, implementace bude provedena s využitím některé z popsaných technologií. Budou vypracovány testovací scénáře a proběhne příprava testovacích prostředí. Dále budou provedeny testy výkonnosti.

Zhodnocení výsledků a doporučení bude věnována samostatná kapitola, která bude tvořit podklad pro závěrečnou část práce. V závěrečné části bude diskutována vhodnost použití programovacího jazyka Kotlin pro platformu Android a možný vývoj do budoucna.

3 Přehled řešené problematiky

Kapitola o platformě Android je pohledem na aktuální možnosti vývoje mobilních Android aplikací z hlediska programovacích jazyků, jejich historie a vývoje do budoucnosti. Okrajově uvádí architekturu platformy a základní charakteristiky. Historie Androidu autora natolik zaujala, že se ji rozhodl vysvětlit detailněji.

Představení programovacích jazyků Kotlin a Java byly věnovány samostatné kapitoly. Charakterizují historii, vývoj, možnosti jazyků a jejich metody použití. Detailněji se zabývají nabízenými funkcionalitami a ukazují jejich použití na okomentovaných částech zdrojových kódů.

V poslední kapitole je probírána problematika testování výkonu. Vysvětluje pojem virtualizace a popisuje vhodné nástroje k testování, jakými je např. benchmarking.

3.1 Platforma Android

Android je rozsáhlá open source platforma. Většina jeho součástí – nativní kód, virtuální stroj Dalvik, aplikační framework a standardní aplikace, je naprosto otevřená. Krom samotného linuxového jádra, je operační systém Android dostupný pod licencí Apache, která umožňuje volné použití a úpravy k libovolným účelům. Není problém nahlédnout do zdrojových kódů a podívat se, jak Android funguje. Výrobci zařízení tak mohou Android upravovat a přizpůsobit jej svému specifickému hardwaru. [2]

Android je určen pro mobilní zařízení. Při vývoji se tým zaměřil na jejich typické vlastnosti jako např. omezený zdroj energie a limity spojené s výpočetním výkonem a pamětí. Android však není určen pouze pro telefony. Nepočítá s určitou velikostí displeje, rozlišením, čipovou sadou či dalšími vlastnostmi. Jeho jádro bylo navrženo tak, aby bylo přenositelné. [2]

Od listopadu 2007 se o vývoj nových verzí stará konsorcium Open Handset Alliance. Jedná se o sdružení 84 společností, mezi které patří např. Google, HTC, Intel, LG, Motorola, nVidia a Samsung. Dává si za cíl progresivní rozvoj mobilních technologií, které umožní nižší náklady na vývoj a distribuci mobilního softwaru. Celý tento systém je

koncipován tak, aby respektoval omezené možnosti, které plynou ze samotných mobilních zařízení. [3]

Android použil jazyk Java jako klíčový pilíř při tvorbě stejnojmenného operačního systému. I přesto, že je Android postaven na Linuxovém jádře a byl napsán převážně v programovacím jazyce C, tak SDK Androidu používá jazyk Java jako základ pro svoje aplikace. Z jazyka Java je však používána pouze syntaxe, nikoliv knihovna tříd. Namísto přepoužití standardní knihovny Java Class Library Android implementoval svoji vlastní knihovní třídu. Aplikace pro Android jsou zkompilovány nejprve do JVM „bajtkódu“ a poté do formátu Dalvik Executables (přípona *dex*), který běží na virtuálním stroji Dalvik Virtual Machine. [2]

3.1.1 Historie

Android je poměrně mladou platformou a jeho historie obsahuje mnoho událostí a náhod, bez kterých by možná vůbec nevznikl. Je zejména o suverénním postavení mobilních operátorů, souboji dvou vzkvétajících technologických gigantů a jednom bláznivém inženýrovi, který nebral „ne“ jako odpověď. [4]

V roce 2004 obdržel Steve Perlman naléhavý hovor od svého kamaráda Andyho Rubina. Rubinův startup, Android, měl potíže se splácením a pronajímatel kanceláří hrozil vystěhováním. Rubin nerad žádal znovu o peníze, ale situace byla vážná. Perlman ještě tentýž den zašel do banky vybrat peníze a předal Rubinovi deset tisíc dolarů. Příští den převedl nespecifikovanou částku jako odrazový můstek pro Android. Perlman věřil v Rubinův startup a chtěl mu pomoci. S novými financemi se dal Android opět do chodu. Rubin zajistil více financí a přesunul se s Androidem do Palo Alto, technologického hubu západního pobřeží. V roce 2014 Android běžel na 85 % všech smartphonů světa, zatímco iPhone pouze na 11 %. Aby se tak stalo, musel Rubin porazit dvě největší společnosti dané éry: Microsoft a Apple. Musel bojovat s operátory hluboce zakořeněnými do mobilního průmyslu a musel přesvědčit výrobce telefonů o své bláznivé myšlence – otevřeném operačním systému pro telefony nadcházejícího tisíciletí. [4]

V prvních letech nového tisíciletí však ovládali trh s telefony mobilní operátoři. Rozhodovali o všem, od způsobu, jakým byly telefony propagovány, až do výše ceny, za

kterou se prodávaly. Takový stav se operátorům líbil a nehodlali na něm nic měnit. Nepřáli si, aby jakákoliv společnost, ať už malá nebo velká, zasahovala do jejich zisků. Což byl také důvod, proč si většina v technologickém průmyslu myslela o podobných myšlenkách, jaké měl Rubin, že jsou neuskutečnitelné. [4]

Andy Rubin se za svoji téměř 30letou kariéru v Silicon Valley proslavil jako technický génius, zkušený obchodník a schopný vůdce. Po škole v roce 1986 pracoval jako inženýr ve firmě Carl Zeiss Microscopy. Po roce se přesunul do Švýcarska, kde se zabýval robotikou. V roce 1989 během dovolené na Kajmanských ostrovech potkal inženýra z Applu, Billa Caswella. Rubin Caswella neznal, ale udělal mu laskavost – nabídl mu místo na přespání, když byl Caswell po hádce s přítelkyní vykázán z plážové chatky. Caswell později nabídl Rubinovi práci, a tak se Rubin dostal do Apple, kde pracoval jako softwarový inženýr mezi lety 1989 a 1992. Rubinova vášeň pro robotiku byla v Apple zjevná – dokonce si během svého působení vysloužil přezdívku „Android“. Byl to však i znamenitý šprýmař. Jednou se dostal do potíží, když přeprogramoval interní telefonní systém v Apple tak, aby to vypadalo, že tehdejší CEO John Sculley volá Rubinovým spolupracovníkům s nabídkou podílu ve firmě. [4]

Rubin a Perlman nakonec opustili Apple a šli pracovat do General Magic – společnosti, která se od Apple oddělila na počátku 90tých let. General Magic je přisuzováno vytvoření osobního příručního počítače, které někteří považují za předchůdce moderních smartphonů. Rubin pracoval pro společnost od roku 1995 a v roce 1997 přešel do WebTV, která byla později koupena Microsoftem a přejmenována na MSN TV. Perlman byl zakladatelem WebTV a odešel do Microsoftu s Rubinem. Po odchodu z Microsoftu v roce 1999, Rubin založil vlastní startup Danger, který později vyvinul smartphone T-Mobile Sidekick. V tu dobu to Rubin ještě nevěděl, ale byl to pro něho velký úspěch, který nakonec vedl k odkoupení jeho startupu Googlem. [4]

Většina lidí považovala Rubina za blázna v jeho vytrvalé snaze o revoluci. Když se Perlman na konferenci Whole Foods v roce 2003 zeptal investora, co si myslí o Rubinovu open-source projektu, dostal odpověď: „Ale no tak, Steve. To by musel prodat alespoň milion těch zařízení, aby se to vyplatilo. Vždyť se snaží o nemožné.“ Tou dobou nemohl investor vědět, že se v roce 2014 bude odhadovat počet prodaných Android zařízení na jednu miliardu. [4]

Přestože Rubinovu myšlenku Androidu považovali mnozí za bláznovství, našel ještě jednoho časného podporovatele: Larryho Page. Spoluzakladatel Googlu, byl tehdejším ředitelem produktu, když se dozvěděl o Rubinovu Android projektu. Požádal, aby Rubina oslovili, a tehdejší telefonát byl možná nejdůležitějším v Rubinově životě. Rubin pozvání přijal a hned první týden v lednu 2005 se vydal spolu se Searsem, spoluzakladatelem Androidu, do sídla Googlu v Mountain View. Setkání se zúčastnili oba zakladatelé Googlu, Larry Page a Sergey Brin. Page Rubinovu práci velmi chválil, ale setkání nebylo jen o vychvalování Rubina a jeho produktech. Brin se Rubina neustále vyptával, co by na Sidekicku změnil, aby byl ještě lepší, a proč ho navrhl způsobem, jakým to udělal. Zkoušel ho. [4]

Když Rubin opouštěl jednání, nevěděl, jaké má Google záměry. Zdali mu chtějí pomoci, nebo vyvíjejí vlastní mobilní software a vyzvídají od konkurence. O 45 dní později přišel telefonát a pozvánka k další schůzce. Záměry Googlu se začaly vyjasňovat. Tentokrát se zúčastnili všichni čtyři zakladatelé Androidu a vzali s sebou prototyp na ukázku. George Harik, poradce pro rozvoj v Google, šel přímo k věci: Google chtěl koupit Android. Zakladatelé byli rozpolcení. Rubin, spoluzakladatel Chris White a Sears byli pro, ale Rich Mines, čtvrtý spoluzakladatel, chtěl, aby společnost zůstala malou firmou. Nakonec však Android nabídku Googlu přijal za 50 milionů dolarů a v červenci 2005 se stěhoval do Googleplexu. [4]

Sídlo Androidu v budově Building 44 nebylo jako zbytek Googlu. Vstup do izolovaných kanceláří bránila postava cylona z televizního seriálu Battlestar Galactica a po celém prostoru se povalovaly různé technické vymoženosti, hračky a roboti. Rubinovo nadšení pro robotiku se zde také projevilo. Ve svém volném čase přeprogramoval obrovské robotické rameno, které pak na vyžádání textovou zprávou uvařilo kávu – nacházelo se ve druhém patře budovy Building 44 a bylo dostatečně silné na zvednutí automobilu. Další Rubinův projekt zahrnoval létání s obrovskou dálkově-řízenou helikoptérou po kampusu Googlu. Když se však snažil vzlétnout, helikoptéra se přetočila vzhůru nohama a stroj za 5000 dolarů se rozletěl na kusy po trávníku před budovou Building 44. Android byl jiný a nechtěl se stát součástí velkého Googlu. Byl vlastně takovým startupem uvnitř Googlu. [4]

Google se snažil prorazit na mobilní trh skrze aplikace dodávané na mobilní platformy jako byly Nokia a Blackberry. Představa Androidu byla však jiná – vytvořit

vlastní operační systém pro distribuci Google služeb. Ostatním zaměstnancům byla taková představa cizí a Android se setkával s nepochopením i mezi svými kolegy z Googlu. Pokud se však Android měl dočkat distribuce, bylo potřeba vyvinout telefon, na kterém by software běžel, a najít operátora, který by takový telefon prodal. A to přesně Apple dělal. Pro Android to však znamenalo vytvořit infrastrukturu, aliance a spojenectví. Spojenectví s výrobci čipů, výrobci telefonů a mobilními operátory. To vše pro vytvoření telefonu, který byl vnímán jako škodlivý prvek průmyslu. A to Andy Rubin uměl, v tom byl dobrý. Nejen, že byl vynikající inženýr, ale také věděl, jak jednat s řediteli a netechnicky zaměřenými lidmi. [4]

Google a Android vytvořili první telefon G1 jako důkaz funkčnosti operačního systému Android ve snaze ukázat potenciálním partnerům, co Android dokáže. Žádný z operátorů však neměl zájem. Verizon odmítl, Sprint nabídka nezaujala a AT&T se nevyjádřil. Dokonce i T-Mobile zpočátku odmítl. Operátoři chtěli prodávat služby skrze telefony a nechat si tak všechen zisk pro sebe. Spolupráce s jinou společností pro ně byla nepřijatelná. V podstatě byli prostředníky mezi výrobci telefonů a koncovými zákazníky a nechtěli na tomto modelu nic měnit. Nakonec se však podařilo T-Mobile přesvědčit. Nick Sears, spoluzakladatel Androidu, dříve pracoval v reklamním oddělení T-Mobile a podařilo se mu přesvědčit tehdejšího technického ředitele T-Mobile, Roberta Dodsona. [4]

Google překonal jednu z největších překážek, když našel operátora ochotného vydat první Android telefon. Ale zrovna když pracoval na posledních úpravách před vydáním, Apple odhalil svůj nový produkt – iPhone. Rubin byl tak překvapený, že při cestě na jednání nechal řidiče zastavit, aby mohl dokoukat on-line přenos. [4]

Vydání nového smartphonu s dotykovým displejem změnilo všechno. Android byl nucen se vrátit zpět na začátek a celý návrh přehodnotit. O budoucnosti dotykového displeje nebylo pochyb. Google s technologií dotykového displeje a gesty experimentoval údajně už dávno předtím, po vydání iPhonu se však vývoj o poznání zrychlil. Nakonec Google smartphone G1 vydal, ačkoliv od původního návrhu se velmi lišil – měl dotykový displej, a naopak upustil od vysunovací klávesnice. Apple v tomto směru vývoj Androidu urychlil a pomohl mu vydat se správným směrem. [4]

Představení iPhonu však přispělo k úspěchu Androidu daleko podivnější cestou. Apple totiž vydal iPhone exkluzivně pro operátora AT&T a očekávání byla natolik vysoká,

že jeho představení přesvědčilo celý svět o tom, že to bude velký hit. V roce 2009 byl úspěch iPhonu již tak značný, že představoval pro Verizon potíže. Verizon totiž neměl žádný smartphone, který by mohl iPhonu konkurovat. Apple tak donutil ostatní operátory přejít na stranu Androidu. [4]

Operátoři vnímali iPhone jako největší hrozbu jejich obchodním modelům. S iPhonem si Apple vytvářel vztah se zákazníkem – nikoliv s AT&T. Uživatelé začali hromadně přecházet k AT&T, aby si mohli iPhone pořídit. Najednou bylo pro Android vyjednávání s operátory jednodušší. Rubin navíc prezentoval Android operátorům jako platformu pro vývojáře, nikoliv spotřebitele, díky čemuž se operátoři a výrobci telefonů cítili pohodlněji. Otevřenost kódu byla také důležitá. Dodávala operátorům jistotu, že Google nebude mít nad Androidem absolutní moc. Strategie byla nabídnout operátorům něco pro jejich křížovou výpravu proti iPhonu. Mohli tak modifikovat telefony a přidat svoji značku, což jim umožňovalo jakýsi stupeň kontroly. Verizon však neměl na iPhone odpověď ve formě smartphonu. Tu poskytla až Motorola se svým prvním Android telefonem. Oproti iPhonu byl poněkud neskladný a měl vysunovací klávesnici, ale byla to nejlepší dostupná alternativa k iPhonu. Verizon utratil 100 milionů dolarů na marketing pod názvem Droid (který byl mimochodem licencován Georgem Lucasem) a tak v říjnu 2009 vydal první telefon jako přímou alternativu k iPhonu. Úspěch nebyl tak velký jako měl iPhone, ale bylo to dost na to, aby si svět začal Androidu všímat. Časem se Android stal mainstreamovým produktem a marginalizoval podíl iPhonu na trhu. [4]

Těžko říci, co přesně bylo důvodem toho, že se z Androidu stala platforma, jakou je dnes. Je to kombinace mnoha věcí. Jednak Rubin věděl, jak oslovit operátory v počátcích nového tisíciletí a věděl, že se nebudou chtít vzdát své moci. Spolu se zbytkem Android teamu v Googlu je přesvědčil, že jeho software by je k tomu nenutil. Současně s tím, operátoři nebyli jediní, kdo měli rozhodující slovo. První Droid byla společná zásluha ze strany Motoroly, Googlu a Verizonu. To se však ukázalo až v konečném důsledku. Rubin zkrátka věděl, jak přesvědčit Google a zbytek mobilního průmyslu že dokáže to, co každý považoval za nemožné. [4]

3.1.2 Dalvik virtual machine (DVM)

Dalvik virtual machine, dále jen Dalvik, je název virtuálního stroje, který v systému Android vytváří běhové prostředí pro aplikace a některé systémové služby. Dalvik byl vytvořen speciálně pro účely Android projektu, neboť JVM nebyla pro mobilní zařízení vhodná. Minimální požadavky byly obecné do takové míry, že bylo zřejmé, že jim bude vyhovovat značné množství zařízení. Z toho důvodu bylo kriticky důležité, aby aplikační platforma byla abstrahována mimo operační systém a hardware, na kterém běží. [5]

Každá Android aplikace běží ve svém vlastním procesu s vlastní instancí virtuálního stroje Dalvik. Ten byl napsán tak, že zařízení může efektivně spouštět více instancí virtuálního stroje zároveň. Dalvik spouští soubory formátu Dalvik Executable (*.dex*), které jsou optimalizovány pro minimální využití paměti zařízení. Na rozdíl od JVM pracuje virtuální stroj Dalvik s registry a spouští třídy zkompilované nejprve Java překladačem a následně transformované do *.dex* formátu (tento překlad zajišťuje utilita *dx* přímo v systému Android). Jeden z minimálních požadavků je operační systém založený na UNIXu. Implementace Dalvik spolého na linuxové jádro k zajištění základních funkcionalit, jakými jsou např. threading nebo low-level memory management. [5]

Přestože byl Dalvik vyvíjen Googlem zcela od počátku a bez dědění jakýchkoliv licencí, některé třídy v Dalvikově knihovně se podobají jejím protějškům v jazyce Java. Tyto podobnosti se staly hlavním bodem sporu mezi Sun/Oracle a Google/Android. Jeho podobnost s JVM byla hlavním bodem žaloby v soudním sporu mezi Oracle a Google. Po dlouhotrvajících sporech, dne 7. května 2012, soud v San Francisku shledal, že pokud by API bylo chráněno copyrightem, pak by společnost Google porušila autorská práva společnosti Oracle použitím Javy v zařízeních s Androidem. Oracle vznesl ve svém postoji otázky týkající se legálnosti používání Javy v Androidu. Nicméně americký okresní soudce William Haskell Alsup rozhodl dne 31. května 2012, že API nemůže být chráněno autorskými právy. Spor dále pokračuje. [6]

3.1.3 Rozšířené jazyky

Android aplikace je možné vyvíjet mnoha způsoby. Operační systém Android podporuje JVM „bajtkód“, tudíž jakýkoliv jazyk, ke kterému existuje kompilátor do JVM

„bajtkódu“, může být použit pro vývoj Android aplikací. Nejpoužívanějším je Java, která je součástí platformy zahrnující JVM. Jedním z jazyků kompilovatelných do JVM „bajtkódu“ je i Kotlin. Oběma jazykům je věnována samostatná kapitola. Android poskytuje taktéž NDK (Native Development Kit) umožňující vývoj částí aplikací v nativním kódu za použití jazyků jakými jsou např. C nebo C++. Existuje i interpret v podobě skriptovací nadstavby nad Javou (SL4A), který umožňuje vývoj v mnoha jazycích, např. Ruby, Python, Perl, Lua a mnoho dalších. Dále existují rozsáhlé frameworky, z nejznáměnších např. Corona, Phonegap a Xamarin. Výhodou těchto frameworků je umožnění tzv. cross-platform vývoje za použití již zmíněných skriptů v Lua nebo webových technologií HTML 5, JavaScript a CSS. Pro jeden kód je tak možné vyvíjet aplikace pro větší množství platforem, např. zároveň pro Android, desktopové aplikace a iOS. [2]

3.1.4 Typy aplikací

Členění Android aplikací by se dalo vymyslet nesčetné množství. Pro představu nejčastějších použití Android aplikací bylo vybráno členění podle účelu a funkce aplikace a podle typu vlákna, na kterém aplikace běží. [7]

3.1.4.1 Členění podle typu vlákna

Většina aplikací, které lze vytvořit pro OS Android, spadá do jedné z následujících kategorií:

Aplikace na popředí

Jedná se o aplikace, které jsou užitečné pouze v případě, když běží v popředí a jejich činnost je efektivně pozastavena, pokud přejdou na pozadí. Typickým příkladem jsou hry nebo aplikace pracující s mapami. [7]

Aplikace na pozadí

Hlavní činnost těchto aplikací probíhá mimo aktuální běh zařízení. Pracují tzv. na pozadí. Příkladem mohou být aplikace sledující telefonní hovory nebo SMS. [7]

Aplikace s přerušovanou činností

Tyto aplikace také většinou pracují na pozadí zařízení, avšak očekává se od nich komunikace s uživatelem formou upozorňování na určité události. Typickém příkladem může být přehrávač médií. [7]

3.1.4.2 Členění podle účelu a funkce

Komplexní aplikaci je obtížné zařadit do jedné z níže uvedených kategorií, většinou obsahuje prvky z více níže uvedených typů. [7]

Zobrazení cloudových dat

Nejčastějším účelem mobilních aplikací je stažení dat z internetu a jejich vhodná prezentace uživateli. Ať už se jedná o zprávy, počasí, či jakákoliv jiná veřejně dostupná data. Ačkoliv jsou data dostupná i s použitím webového prohlížeče, aplikace umožňují zobrazení dat s ohledem na dané zařízení a jeho parametry (zejm. velikost displeje). Aplikace jsou zpravidla rychlejší než prohlížeč a z hlediska ovládání i použitelnější. [7]

Zobrazení webové stránky

Aplikace, které používají v určité své části vykreslování webové stránky pomocí webového enginu. Od prohlížeče se liší například ovládacími prvky, které mohou být s webovým prohlížečem propojeny (např. pro navigaci na webové stránce). [7]

Zábavní a interaktivní aplikace

Jedná se o výpočetně náročné aplikace, využívající výpočetní a grafický čip telefonu. Zahrnuje práci s bitmapami a zvukovými soubory, interakci s uživatelem a síťové prvky pro hru více hráčů. [7]

Práce s daty (hudba, editace fotek)

Jedná se o aplikace, které lokálně čtou či manipulují se soubory. Těmito aplikacemi mohou být např. textové editory, přehrávače hudby či editory fotografií. [7]

Práce se senzory

Aplikace pracující se senzory zařízení. Např. aplikace pro nahrávání zvuku pomocí mikrofonu, čtečky QR kódů či čísel platebních karet, které využívají fotoaparát, či navigace využívající GPS a akcelerometr. [7]

Sběr dat

Aplikace používající mobilní telefon pro automatizovaný sběr dat. Většinou je implementovaný pomocí služby běžící na pozadí, která periodicky odesílá data na server. Tyto aplikace jsou často součástí sociálních sítí. [7]

Utility

Aplikace usnadňující práci na daném zařízení. Může se jednak například o kalkulačku, převodník mezi fyzikálními jednotkami nebo průzkumník souborů. [7]

3.1.5 Vývoj aplikací

Vývoj mobilních aplikací je typický svojí cílovou skupinou. Zejména se jedná o mladší generaci. Oproti desktopovým aplikacím, mají uživatelé mobilních zařízení průměrně vyšší počet, ačkoliv krátkých, interakcí v průběhu dne. To klade na aplikace zcela nové nároky. Zejména pak na délku odezvy a přehlednost aplikace. Uživatel, který stáhne a nainstaluje aplikaci z Obchodu Play, očekává, že po spuštění bude rozhraní natolik jednoduché a intuitivní, že se v něm zorientuje bez nutnosti čtení uživatelských příruček a návodů. Konkurence na trhu aplikací je vysoká, k červnu 2017 bylo v Obchodu Play přes 3 miliardy aplikací. Pokud aplikace nenaplní uživatelova očekávání v prvních několika minutách, uživatel aplikaci odinstaluje a jde hledat jinam. [1]

Dalším specifikem mobilního průmyslu jsou časté změny. Ať už je příčinou neustále přinášet uživatelům něco nového, nebo držet krok s konkurencí a moderními technologiemi, mobilní aplikace mají často kratší cyklus vydávání nových verzí – měsíce, někdy až týdny. To má za následek tlak na vývojová oddělení. Vývoj pro iPhone má výhodu nízkého počtu zařízení, a tedy i nízké potřeby portability a optimalizací, dále má striktní pravidla, která jsou pro vývojáře sice svazující, ale pomáhají jim lépe se orientovat v cizím kódu. Android je typický častým použitím open-source knihoven třetích stran. [1]

3.1.6 Verze

Na rozdíl od ostatních mobilních operačních systémů, které jsou spravovány jedním výrobcem, se platforma Android skládá z mnoha verzí operačních systémů přizpůsobených jednotlivými výrobci mobilních zařízení. To má za příčinu, že s příchodem nových verzí

musí kompatibilitu se starými zařízeními řešit jejich samotní výrobci. Tento proces však zabere mnoho času a někdy výrobci od podpory upouští. Koncoví uživatelé těchto zařízení jsou tak ponecháni na starších verzích, což jim znemožňuje využívat nejnovější funkce. Tomuto jevu se říká fragmentace. Vývojáři jsou nuceni s touto skutečností počítat a svoje aplikace jí přizpůsobit. Fragmentaci se dá do jisté míry vyhnout zvolením vhodného API. Google vychází v tomto ohledu vývojářům vstříc a pravidelně zveřejňuje zastoupení jednotlivých verzí systémů přistupujících na službu Google Play (viz Tabulka 2). Pro srovnání vývoje verzí je uvedena tabulka z bakalářské práce autora z roku 2015 (viz Tabulka 1). [8]

Tabulka 1 - Distribuce verzí Androidu ke dni 2. 2. 2015 [9]

Version	Codename	API	Distribution
2.2	Froyo	8	0.4%
2.3.3 - 2.3.7	Gingerbread	10	7.4%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	6.4%
4.1.x	Jelly Bean	16	18.4%
4.2.x		17	19.8%
4.3		18	6.3%
4.4	KitKat	19	39.7%
5.0	Lollipop	21	1.6%

Tabulka 2 - Distribuce verzí Androidu ke dni 5. 2. 2018 [8]

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.4%
4.1.x	Jelly Bean	16	1.7%
4.2.x		17	2.6%
4.3		18	0.7%
4.4	KitKat	19	12.0%
5.0	Lollipop	21	5.4%
5.1		22	19.2%
6.0	Marshmallow	23	28.1%
7.0	Nougat	24	22.3%
7.1		25	6.2%
8.0	Oreo	26	0.8%
8.1		27	0.3%

Hardwarové fragmentaci se však vyhnout nedá. Jak už bylo zmíněno v úvodu, existuje téměř 19 tisíc různých zařízení používající Android. Nezbývá tedy než napsat oddělený kód pro starší zařízení, nebo tato zařízení vůbec nepodporovat. [8]

3.2 Java

Java je objektově orientovaný programovací jazyk s relativně jednoduchou syntaxí. Je koncipován pro vývoj robustních, distribuovaných a snadno udržovatelných programů. Základní stavební jednotkou je třída, která obsahuje veškerý kód (výjimkou jsou pouze komentáře a dokumentace, informace o balíčku a importy). Je architektonicky neutrální, což znamená, že nepožaduje určité struktury a ponechává architekturu zcela na programátorovi. Podporuje vícevláknovost, konkurenční vyhodnocování a umožňuje silnou míru zabezpečení. Při návrhu byl kladen důraz na minimální počet implementačních

závislostí a přenositelnost mezi všemi platformami podporujícími Javu bez jakýchkoliv úprav kódu. Zkompilovaný zdrojový kód není přeložen do strojového kódu, ale do „bajtkódu“, který je určen pro interpret nazývaný Java virtual machine, zkráceně JVM. [10]

Ačkoliv jeho popularita pomalu klesá, podle TIOBE[1] indexu je stále nejpoužívanějším jazykem na světě, a to zejména na poli klient-server webových aplikací. V současné době je vyvíjen společností Oracle Corporation a, jak Oracle tvrdí na svých webových stránkách, aktuálně v Javě vyvíjí 10 milionů programátorů a počet zařízení, na kterých Java běží, přesahuje 15 miliard. [11] [12]

3.2.1 Historie

S přihlédnutím k historii světa počítačů a výpočetní techniky, je Java jazykem poměrně mladým. Její historie začíná v roce 1990, ve firmě Sun Microsystems, jedním nespokojeným programátorem. Patrick Naughton byl frustrován stavem aplikačního rozhraní a nástrojů jazyka Sun C++ a C a uvažoval o odchodu do NeXT, nadějné firmy založené společností Apple Computer. Dostal však nabídku pracovat na vývoji nové technologie a tím se zrodil Stealth Project. Krátce poté změnil projekt název na Green Project a k Naughtonovi se přidali James Gosling a Mike Sheridan. Společně s dalšími programátory začali pracovat na technologii pro programování nové generace chytrých spotřebičů, čímž si Sun sliboval nové příležitosti v budoucnosti. [13] [14]

Tým z počátku uvažoval o použití jazyka C++, ale zavrhl jej hned z několika důvodů. Protože vyvíjeli vestavěný systém (embedded system) s omezenými prostředky, rozhodli, že se C++ nehodí z důvodu spotřeby velkého množství paměti a složitosti vedoucí k častějším chybám při vývoji. Jazyk bez automatické správy paměti (garbage collection) by totiž pro programátory znamenal manuální správu paměti. Další obavou byla přenositelnost zabezpečení, distribuce a „vláknování“, kterou jazyky rodiny C++

[1] TIOBE index – vyjadřuje popularitu programovacích jazyků na základě analýzy výstupů dotazů internetových prohlížečů (nezahrnuje např. jazyky SQL či HTML)

postrádaly. Stejně tak chtěli platformu, která by se snadno přenášela na nejrůznější typy zařízení. [13]

Po krátkém experimentování s kombinováním nejrůznějších jazyků jako Mesa a C, se tým rozhodl vytvořit nový programovací jazyk. Pracovní název dostal po dubu, stojícím před kanceláří, tedy Oak. V létě 1992 byli schopni představit první části nové platformy, včetně Green OS, jazyka Oak, knihoven a hardwaru. Při demonstraci 3. září 1992, představili osobního digitálního asistenta (PDA) s grafickým rozhraním a uživatelským pomocníkem nazývaným „Duke“. [13]

Ještě téhož roku v listopadu byl projekt Green ukončen a jeho tým přidělen k Firstperson, dceřiné společnosti Sun Microsystems zabývající se vývojem vysoce interaktivních zařízení. Firstperson měla poptávku po set-top box zařízení od mediální společnosti Time Warner a poté i od videoherní společnosti 3DO. Žádnou ze společností však nabídka nezaujala a Firstperson byl sloučen zpět pod Sun. [14]

V červenci 1994 se tým rozhodl přizpůsobit platformu pro World Wide Web, zkráceně *web*. Cítili, že s nástupem webových prohlížečů jako Mosaic nebo NetScape by se Internet mohl stát stejně interaktivním jako kabelové televize. Naughton proto vytvořil prototyp prohlížeče WebRunner, přejmenovaný v roce 1995 na HotJava. Při registraci ochranné známky se zjistilo, že Oak je již používaný, a tak dostal programovací jazyk současný název Java. Následně téhož roku oznámila společnost Netscape podporu Javy ve svém prohlížeči. V lednu 1996, Sun Microsystem vytvořil skupinu JavaSoft odpovědnou za vývoj technologie. [14]

3.2.2 Java virtual machine (JVM)

Java virtual machine (dále jen JVM) je základním kamenem platformy Java. Je to část technologie, komponenta, zodpovědná za nezávislost platformy na hardwaru a operačním systému, malou velikost kompilovaného kódu a za ochranu uživatelů před škodlivými programy. Je to abstraktní výpočetní stroj. Stejně jako reálný stroj, má sadu instrukcí a manipuluje s oblastmi paměti za běhu (v reálném čase). [15]

První prototyp implementace JVM, vytvořený v Sun Microsystems, emuloval JVM sadu instrukcí v softwaru hostovaném na již zmiňovaném příručním zařízení

připomínajícím personálního digitálního asistenta (PDA). Aktuální implementace emulují JVM na mobilních zařízeních, stolních počítačích a serverech. Avšak JVM nepředpokládá žádné konkrétní implementační technologie, hostitelský hardware ani hostitelský operační systém. Není inherentně interpretován, ale může být stejně rovněž implementován kompilací jeho sady instrukcí do sady konkrétního CPU. Může být dokonce implementován v mikrokódu nebo přímo v křemíku. [15]

O jazyku Java neví JVM absolutně nic. Rozumí pouze binární formě uložené do *class* souborů, do kterých je Java kód zkompilován. Soubory *class* obsahují JVM instrukce neboli „bajtkód“, tabulku symbolů a další pomocné informace. [15]

Z důvodu bezpečnosti JVM ukládá silná syntaktická a strukturální omezení na kód v *class* souborech. Nicméně každý programovací jazyk, který je schopen vyhovět těmto podmínkám a lze jej vyjádřit jako validní *class* soubor, může běžet na JVM. Pokud tedy existuje překladač daného jazyka do „bajtkódu“ JVM, je možné v daném jazyce programovat a využít JVM jako nezávislou platformu. [16]

3.2.3 Syntaxe

Syntaxe jazyka Java je do značné míra odvozena z jazyků C a C++. Na rozdíl od C++, v Javě nejsou globální funkce nebo proměnné, ale data members, které vystupují jako globální proměnné. Veškerý kód patří do tříd a veškeré hodnoty jsou objekty. Jedinou výjimku tvoří primitivní datové typy, které z výkonnostních důvodů nejsou reprezentovány instancí třídy. Některé vlastnosti, jako např. přetěžování operátorů nebo neznaménkový integer, nebyly do jazyka přidány pro zjednodušení a omezení programátorských chyb. [16]

Syntaxe Javy byla postupně rozšiřována. Aktuálně má téměř deset verzí a od verze 8 nabízí i funkcionality jako generické programování, či lambda výrazy. Nutno však podotknout, že Android využívá Javu verze 7 a tyto funkcionality v něm tudíž nejsou dostupné. [16]

3.2.4 Vybraná pravidla syntaxe

Vysvětlení všech pravidel tak robustního jazyka, jakým je Java, by bylo obdobně kategoricky složité jako vysvětlit všechna pravidla gramatiky anglického jazyka. Z důvodu rozsahu práce byla vybrána pouze ta pravidla, která nejsou v Kotlinu dostupná, či jejich implementace je v Kotlinu složitá a dá se chápat jako jeho nedostatek. Veškerá pravidla je možné nalézt v dokumentaci na oficiálním webu Javy. [16] [17]

3.2.4.1 Checked exceptions

Jedná se o příznak v deklaraci metod (viz následující úryvek kódu), který stanovuje, že daná metoda může produkovat výjimku. [16]

```
Appendable append(CharSequence csq) throws IOException;
```

Jakékoliv volání této metody pak musí být ošetřeno blokem *try-catch.*

```
try {
    log.append(message);
} catch (IOException e) {
    // Ošetření výjimky
}
```

3.2.4.2 Primitivní datové typy

Java umožňuje přímé použití primitivních datových typů. V Kotlinu je vše objektem, a přestože po překladu jsou např. čísla, znaky a logické hodnoty převedeny na primitivní datové typy, uživateli se jeví jako instance tříd s odpovídajícími metodami. [16]

3.2.4.3 Statické metody

Statické metody jsou součástí třídy, ale pro jejich volání není třeba vytvářet instanci dané třídy. V Kotlinu se doporučuje použít funkce na stejné úrovni balíčku. [16]

3.2.4.4 Neprivátní atributy

Zatímco Java umožňuje veřejné atributy s přímým přístupem, v Kotlinu se k těmto účelům používají automaticky generované „gettery“ a „settery“. [16]

3.2.4.5 *Wildcard typy*

Wildcard datový typ, někdy překládaný jako „žolíkový“ nebo „zástupný“, je v Javě speciálním datovým typem zaručujícím typovou bezpečnost při deklaraci generických a parametrizovaných struktur. Následující příklad zobrazuje použití *wildcard* datového typu pro deklaraci seznamu obsahujícího objekty podtříd třídy *Animal*. [17]

```
public void printAnimalNames(List<? extends Animal> list) {
    for (Animal object : list) {
        // vypiš jméno zvířete
    }
}
```

3.2.4.6 *Ternární operátor*

Ternární operátor je operátor přijímající tři vstupy a generující jeden výstup. Nejčastěji používaným ternárním operátorem je operátor pro podmíněný výraz umožňující zkrácený zápis na jeden řádek. Na následujících příkladech je v daném pořadí postupně zachycen zápis podmínky pomocí klauzule *if* a pomocí ternárního operátoru *?*. [17]

```
if (podmínka) {
    výraz1;
} else {
    výraz2;
}
```

```
podmínka ? výraz1 : výraz2;
```

3.3 Kotlin

Kotlin je programovací jazyk pro JVM vyvíjený společností JetBrains. Společností, známou pro sadu produktů IntelliJ IDEA, profesionálních vývojových prostředí. Android Studio, oficiální vývojové prostředí platformy Android, je založeno právě na IntelliJ. Hlavní vývojový team sídlí v Ruském Petrohradu, odkud je také odvozen název jazyka – Kotlin je název ruského ostrova nedaleko Petrohradu. [18]

Jazyk Kotlin je staticky typovaný a kompilovatelný do JVM „bajtkódu“ s možností překladu do JavaScriptu. Je intuitivní a díky podobnosti s Javou je pro vývojáře Java platformy jednoduchý na pochopení. S Javou není syntakticky kompatibilní, ale je navržen pro vzájemnou interoperabilitu a na některých knihovnách Javy dokonce závisí (např.

kolekce). Oproti Javě je expresivnější (na stejnou funkcionalitu stačí menší množství kódu), bezpečnější v přístupech k objektům (rozlišuje mezi „nullovatelnými“ a „nenullovatelnými“ datovými typy a kontroluje neošetřené přístupy), vykazuje prvky funkcionálního programování (lambda výrazy, způsob použití kolekcí) a používá tzv. *extension functions* umožňující rozšíření jakékoliv třídy o nové funkce i v případě, že k dané třídě nejsou zdrojové kódy. [18]

Nespornou výhodou je i integrace do bezplatné verze IntelliJ IDE. Android vývojáři využívající Android Studio tak mají zaručenou podporu přímo od společnosti, která IDE vyvíjí. JetBrains používá Kotlin na řadu svých projektů a dle zkušeností jejích vývojářů [18] se Kotlin vyznačuje:

- nižším množstvím kódu
- vyšší čitelností
- vyšší typovou bezpečností
- vyšší expresivitou
- bezproblémovými zkušenostmi s nástroji a interoperabilitou[2]

Zdrojový kód je veřejně dostupný na repozitáři GitHub a vyvíjený pod licencí Apache 2.0. Podle vedoucího vývoje Andreye Breslava je Kotlin navržen jako průmyslově spolehlivý, objektově orientovaný jazyk, který je lepší než Java, ale je s ním naprosto interoperabilní. [18]

3.3.1 Historie

Na rozdíl od Javy, historie Kotlinu nesahá nikterak daleko. Vývoj začal v roce 2010 jako interní projekt společnosti JetBrains a představen byl rok poté jako nový programovací jazyk pro JVM. Jak uvádí JetBrains na svém blogu, desetileté zkušenosti s vývojem v Javě je přesvědčily, že jejich produktivitu by mohlo znatelně zvýšit použití

[2] interoperabilita – schopnost různých systémů vzájemně spolupracovat, poskytovat si služby, dosáhnout vzájemné součinnosti

moderního JVM jazyka vedle Javy. Začali se tedy poohlížet po alternativách. Vedoucí týmu JetBrains, Dimitry Jemerov, prohlásil, že většina jazyků, kromě Scaly, nenabízí možnosti, které jeho tým vyžadoval. Problémem Scaly byl podle Jemerova příliš dlouhý čas kompilace. Jedním ze základních cílů projektu byla totiž kompilace alespoň tak rychlá jako u čisté Javy. Tým měl však i jiné požadavky. Jazyk měl být jednoduchý na naučení a pochopení, měl umožnit postupnou a snadnou migraci z Javy s co možná nejnižším dopadem na zbytek kódu, zpětnou kompatibilitu a mnoho dalších. [19]

Po vyhodnocení ostatních dostupných možností došel tým k závěru, že pro uspokojení požadavků je třeba nový jazyk. A JetBrains měl v té době zkušenosti a zdroje na vytvoření takového jazyka. Jejich hlavní téma podnikání byla tvorba nástrojů pro vývoj a jejich přesvědčení a princip takových nástrojů spočíval v tom, že dané nástroje tvořili i pro svoji vlastní potřebu. Měli za sebou řadu úspěšných produktů, jako např. již zmíněné IntelliJ IDEA. Ale i ReSharper a mnoho dalších IDE, TeamCity a jiné serverové produkty. Rozhodli se proto aplikovat stejné principy i na tvorbu dalšího vývojového nástroje – programovacího jazyka. [19]

Je zřejmé, že takový projekt zahrnuje mnohá důležitá rozhodnutí a v JetBrains věděli, že je obtížné, ne-li nemožné udělat vše správně na první pokus. Značnou dobu proto věnovali experimentování a validaci počátečního návrhu. V únoru 2012 uvolnil JetBrains projekt jako open source a první uživatelé nejen z JetBrains, ale i ze zbytku světa, se připojili k tvorbě prostřednictvím zpětné vazby a komentářů. Vznikla tak nemalá komunita, což dalo JetBrains důležité poznatky a nejrůznější případy užití v cestě za uspokojením dalšího požadavku – zpětné kompatibility. [19]

V únoru 2016 byl představen Kotlin 1.0, považovaný za první oficiálně stabilní vydání a JetBrains slíbil, že se od této verze bude snažit o zachování zpětné kompatibility. Podle JetBrains se již Kotlin v řadě jejich projektů, včetně IntelliJ IDEA i YouTrack, používá a k únoru 2015 obsahovaly produkční zdrojové kódy přes 250 000 řádků kódu. [18] [20]

Kotlin byl Android komunitou dobře přijat, avšak o jeho budoucnosti a postoji velkých společností jako je Google panovaly obavy. Ty se však rozplynuly rok poté, když Android tým na konferenci Google I/O v květnu 2017 oficiálně oznámil podporu jazyka Kotlin pro vývoj Android aplikací. Google prohlásil, že nové Android funkcionality,

frameworky, IDE a knihovny budou Kotlin plně a bezproblémově podporovat. Zatímco je tedy stále možné vyvíjet aplikace v Javě, od verze Android Studia 3.0 bude zaručena podpora Kotlinu bez nutnosti instalace dodatečných rozšíření či obav o kompatibilitě. Nyní jsou tedy na tahu vývojáři a firmy, aby se vyjádřili k možnostem, které Kotlin nabízí. [21]

3.3.2 Syntaxe

Podobně jako Pascal, TypeScript, Haxe, PL/SQL, F#, Go a Scala a na rozdíl od jazyků odvozených od C, jako například C++, Java, C# a D mají proměnné a seznamy parametrů v Kotlinu datový typ deklarovaný až po názvu (oddělený dvojtečkou). Stejně jako v jazycích Scala a Groovy jsou středníky volitelné a většinou se používají jen v případě, kdy je vzhledem k čitelnosti kódu potřeba mít více příkazů na jednom řádku. [22]

Vedle tříd a metod (které se v Kotlinu nazývají *member functions*) vlastních objektově orientovaným jazykům podporuje Kotlin i procedurální programování za použití funkcí. Stejně jako v Javě je vstupním bodem do aplikace funkce s názvem *main*, která přijímá jako jediný argument pole argumentů z příkazové řádky. Kotlin podporuje interpolaci řetězců ve stylu Perlu a Unix/Linux Shellu. Také je podporováno odvozování typů proměnných. [22]

3.3.3 Vybraná pravidla syntaxe

Obdobně jako u jazyka Java, veškerá syntaktická pravidla Kotlinu jsou rozsáhlá a nebylo je možné do práce zahrnout z důvodu omezeného rozsahu. Vybrána byla ta pravidla, která nejsou pro oba jazyky společná, a tedy která vytváří prostor pro diskusi nad výhodami a nevýhodami jazyků. Veškerá pravidla je možné nalézt v dokumentaci na oficiálním webu Kotlinu. [22] [17]

3.3.3.1 High-order funkce a lambda výrazy

Jedná se o funkce, které přijímají jako parametr funkci, nebo funkci vracejí. Hlavní příklad použití takových funkcí je u návratových funkcí neboli *callback functions*. Dobrým

příkladem je provádění síťových operací, které mohou dopadnout úspěšně, nebo skončit chybou. V takovém případě lze vytvořit funkci, která přijímá dvě návratové funkce, jednu pro úspěšné zpracování a druhou pro případ nastane-li chyba. [22]

```
fun networkCall(onSuccess: (ResultType) -> Unit,
                onError: (Throwable) -> Unit) {
    try {
        // … vytvoří a provede síťový požadavek
        onSuccess(myResult)
    } catch(e: Throwable) {
        onError(e)
    }
}
```

```
networkCall(result -> {
    // použití úspěšného výsledku
}, error -> {
    // zpracování chyby
})
```

Rozšířené funkce mohou být uloženy do proměnné, předávány, nebo vytvořeny v jiné funkci. Pokud funkce není deklarována a je předána přímo jako výraz, nazývá se tzv. lambda funkcí. V Javě 8 je již tato funkcionalita implementována a nazývá se *function literal* neboli „funkční literál“. Jak už však bylo zmíněno, Android používá Javu 7 a tak tato funkcionalita není dostupná. [23]

3.3.3.2 Inline funkce

Již zmíněné *high-order* funkce, které umožňují předávat jiné funkce jako parametry, mají však i své nevýhody. Použití lambda výrazů ve výsledku vytváří anonymní funkce, které jsou paměťové náročné. Vytváření anonymních funkcí se dá vyhnout pomocí deklarace *inline*. [22]

```
fun notInlined(getString: () -> String?) = println(getString())

inline fun inlined(getString: () -> String?) = println(getString())
```

Tyto dvě funkce vykonávají stejnou funkci – vypisují výsledek funkce *getString*. Jedna je deklarována jako *inline* a druhá nikoliv. Použití obou funkcí je v Kotlinu stejné. [22]

```
fun test() {
    var testVar = "Test"

    notInlined { testVar }

    inlined { testVar }
}
```

Rozdíl nastává až při samotném volání funkcí, jak je patrné na ukázce dekompilovaného kódu do Javy. [22]

```
public static final void test() {
    final ObjectRef testVar = new ObjectRef();
    testVar.element = "Test Variable";

    notInlined((Function0)(new Function0(0) {
        public Object invoke() {
            return this.invoke();
        }

        @NotNull
        public final String invoke() {
            return (String)testVar.element;
        }
    }));

    //inlined:
    String var3 = (String)testVar.element;
    System.out.println(var3);
}
```

3.3.3.3 Rozšířující funkce

Extension function, neboli rozšiřující funkce, je funkcionalita převzatá z C#. Umožňuje upravovat chování tříd bez nutnosti dědění a rozšiřování. Pomocí rozšiřujících funkcí lze volat metodu na objektu, přestože v jeho třídě není definovaná. [23]

```
fun String.provedZmeny(): String {
    // ... provede změny a vrátí String
}
```

Reprezentací daného kódu v Javě je statická funkce s parametrem daného objektu a návratovou hodnotou datového typu daného objektu.

```
public class StringExtensions {
    public static String provedZmeny(String objekt) {
        // ... provede změny a vrátí String
        return objekt;
    }
}
```

Volání v jednotlivých jazycích pak vypadá následovně:

```
StringExtensions.provedZmeny(objekt); // Java
objekt.provedZmeny() // Kotlin
```

3.3.3.4 Nullová bezpečnost

Jak již bylo zmíněno v úvodu, Kotlin rozlišuje mezi „nenullovými“ a „nullovými“ datovými typy. Pokud však není výslovně řečeno, že daný typ bude „nullový“, ve výchozím stavu jsou všechny typy „nenullové“. Tímto Kotlin eliminuje většinu chyb „nullových“ referencí již v době překladu – kompilátor nedovolí použití neinicializované „nenullové“ proměnné. Pokud programátor potřebuje z nějakého důvodu ukládat „nullovou“ hodnotu do proměnné, musí tak předem danou proměnnou deklarovat přidáním znaku *?* za název datového typu. [17]

```
var nonNullable: String // musí být inicializovaná
var nullable: String?
```

Před přístupem k „nullové“ proměnné, kompilátor požaduje kontrolu, zdali není hodnota dané reference *null*. To lze udělat dvěma způsoby. Tradičním podmíněným příkazem *if statement* známým z Javy, nebo použitím tzv. *safe call* operátoru. Operátor může být i řetězen za sebe. [17]

```
if (nullable != null) nullable.necoUdelej()
```

```
nullable?.necoUdelej()
```

Kontrola kompilátoru se dá obejít použitím operátoru *!!*, což v případě, že je hodnota proměnné „nullová“, vede k vyhození výjimky a případně i pádu aplikace. [23]

3.3.3.5 Smart casts

Smart casts, volně přeloženo jako „chytrá přetypování“, je v Kotlinu funkce kompilátoru umožňující vynechání explicitních přetypování. Kompilátor sleduje *is* syntaxe (ekvivalent *instanceof* v Javě), které testují, zdali je objekt potomkem třídy, a vkládá přetypování automaticky, když je potřeba. [17]

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x je automaticky přetypováno na String
    }
}
```

3.3.3.6 String templates

„Řetězcové šablony“ mohou obsahovat předem připravené výrazy pro vkládání částí kódu, které jsou vyhodnocovány a připojeny k řetězci. Výrazy uvnitř šablon začínají znakem dolaru *$* a skládají se z jednoduchého názvu proměnné nebo z libovolného výrazu uzavřeného do závorek (viz ukázky kódu). [17]

```
val i = 10
val s = "i = $i" // přeloží se na "i = 10"
```

```
val s = "abc"
val str = "$s.length is ${s.length}" // přeloží se na "abc.length is 3"
```

3.3.3.7 Properties

Třídy v Kotlinu mohou mít *properties* (česky „atributy“). Mohou být deklarovány jako proměnlivé použitím klíčového slova *var* nebo pouze pro čtení klíčovým slovem *val*. [17]

```
class Address {
    var name: String = ...
    var street: String = ...
    var city: String = ...
    var state: String? = ...
    var zip: String = ...
}
```

Plná syntaxe deklarace atributu je zachycena na obrázku (viz ukázka kódu). Inicializátor, *getter* a *setter* jsou nepovinné. Typ je nepovinný pouze pokud může být odvozen z inicializátoru nebo z návratového typu v „getteru“. [23]

```
var <název>[: <typ>] [= <inicializér>]
[<getter>]
[<setter>]
```

3.3.3.8 Primární konstruktory

Třídy v Kotlinu mohou obsahovat jeden primární konstruktor a jeden nebo více sekundárních. Primární konstruktor je součástí hlavičky v deklaraci třídy a uvádí se za název třídy a případně i nepovinné parametry. [23]

```
class Person constructor(firstName: String) {
}
```

Nemá-li primární konstruktor žádné anotace či viditelné modifikátory, klíčové slovo *constructor* může být v takovém případě vynecháno. [23]

```
class Person(firstName: String) {
}
```

Primární konstruktor nemůže obsahovat žádný kód. Inicializační kód může být vložen do inicializačních bloků, které jsou uvozeny prefixem s klíčovým názvem *init*. Během vytváření instance, inicializační bloky jsou prováděny ve stejném pořadí, v jakém byly definovány v těle třídy, prokládány inicializací atributů. [23]

```
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
        println("First initializer block that prints ${name}")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
```

Výstup výše uvedeného programu by pak vypadal následovně:

```
First property: hello
First initializer block that prints hello
Second property: 5
Second initializer block that prints 5
```

3.3.3.9 First-class delegation

Návrhový vzor *delegation* nebo „delegát“ slouží v objektovém programování jako alternativní implementace dědičnosti. Kotlin tento návrhový vzor nativně podporuje čímž

eliminuje potřebu psaní dalšího kódu. V uvedeném příkladu třída *Derived* dědí z rozhraní *Base* a deleguje veškeré veřejné metody na specifikovaný objekt. [17]

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main(args: Array<String>) {
    val b = BaseImpl(10)
    Derived(b).print() // vytiskne 10
}
```

Klauzule *by* v typu nadřazené třídy indikuje, že *b* je uložené lokálně v objektech typu *Derived* a kompilátor automaticky vygeneruje veškeré metody třídy *Base*, které přesměruje na *b*. [17]

3.3.3.10 Typová odvození pro proměnné a atributy

Absence implicitní konverze typů není v Kotlinu výjimkou. Rovněž není nutné uvádět datový typ při deklarace proměnné. Kompilátor zvládá odvodit datový typ z kontextu a aritmetické operace jsou přetíženy pro odpovídající konverzi. [17]

```
val l = 1L + 3 // Long + Int => Long
```

3.3.3.11 Singletons

Návrhový vzor singleton (česky *jedináček*) umožňuje, aby v celém programu existovala pouze jediná instance třídy, je v Kotlinu rovněž podporován. Lze jej implementovat deklarací objektu, která v podstatě uvozuje anonymní třídy. Deklarace objektu probíhá obdobně jako deklarace třídy, namísto *class* se však použije klíčové slovo *object*. [23]

```
object Singleton {
    fun funkce(provider: DataProvider) {
        // ...
    }

    val atribut ...
}
```

3.3.3.12 Declaration-site variance & Type projections

Declaration-site variance je v Kotlinu způsob, jak sdělit kompilátoru, že daný generický datový typ *T* bude vracen pouze členy třídy *Source<T>* a nikdy nebude konzumován. Deklarace tohoto typu se uvozuje modifikátorem *out* a nazývá se *variance annotation*. [17]

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    // Toto je v pořádku, protože T je parametrem out
    val objects: Source<Any> = strs
    // ...
}
```

Protějškem modifikátoru *out* je modifikátor *in*, který naopak uvozuje typ parametru a říká, že daný parametr může být pouze konzumován a nikdy produkován. [17]

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    // 1.0 je typu Double, což je podtyp třídy Number
    x.compareTo(1.0)
    // proto lze přiřadit x do proměnné typu Comparable<Double>
    val y: Comparable<Double> = x // Toto je v pořádku!
}
```

3.3.3.13 Intervalové výrazy

Intervalové výrazy (z angl. *range expressions*) umožňují zjednodušený zápis iterace na uzavřeném intervalu hodnot (viz ukázka kódu). Třída deklarující hodnoty musí implementovat rozhraní *Comparable* a intervaly pro dané typy implementují rozhraní *ClosedRange<T>*, které reprezentuje matematicky uzavřený interval. To obsahuje např.

metodu *rangeTo* dostupnou přes operátor „..“ a *step* vracející hodnotu, o kterou se inkrementuje index v každém dalším kroku. [23]

```
if (i in 1..10) { // ekvivalentní k 1 <= i && i <= 10
    println(i)
}

for (i in 1..4) print(i) // vytiskne "1234"

for (i in 4 downTo 1 step 2) print(i) // vytiskne "42"
```

3.3.3.14 Přetěžování operátorů

Pro předdefinovanou skupinu operátorů umožňuje Kotlin jejich přetížení. Tyto operátory mají fixní symbolickou reprezentaci (např. „+“ nebo „*“) a fixní priority vyhodnocování. K implementaci operátoru je nutné definovat metodu uvozenou klíčovým slovem *operator*, pojmenovanou daným názvem pro odpovídající operátor. Následující tabulka zachycuje výrazy aritmetických operací a překlady na odpovídající funkce. Vlastní infixové operátory lze nasimulovat použitím infixové notace při deklarování metody. [23]

Výraz	Překlad
a + b	a.plus(b)
a – b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem (b)
a..b	a.rangeTo(b)

3.3.3.15 Infixové notace metod

Infixová notace se uvozuje klíčovým slovem „infix“ před deklaraci funkce a umožňuje volání funkce v infixové notaci (vynechání tečky a závorek). Infixové funkce musí splňovat následující pravidla [23]:

- Musí být metodami třídy nebo metodami rozšířenými (*extension function*)
- Musí mít právě jeden parametr

- Parametr nesmí akceptovat proměnlivý počet proměnných a nesmí mít výchozí hodnotu

Příklad deklarace infixové funkce a následné volání je zachyceno na příkladu.

```
infix fun Int.shl(x: Int): Int {
    // ...
}

// volání funkce použitím infixové notace
1 shl 2

// adekvátní příkaz
1.shl(2)
```

3.3.3.16 Companion objekty

V Kotlinu, na rozdíl od Javy nebo C#, třídy nemají statické metody. Ve většině případů je doporučené použít namísto statických metod funkce na stejné úrovni balíčku. Případně, pokud je potřeba funkce, kterou lze volat s absencí instanční třídy, ale která potřebuje přístup k vnitřku třídy (např. tovární metoda), lze takovou funkci zapsat jako metodu objektové deklarace uvnitř deklarace třídy. *Companion* objekt je speciálním případem takového objektu, který umožňuje volání jeho metod stejnou syntaxí, jakou se volají statické metody v Javě. [17]

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}

val instance = MyClass.create()
```

3.3.3.17 Datové třídy

Datové třídy, známé z Javy jako POJO[3] třídy, nebo někdy také jako Java Bean, jsou třídy neobsahující vnitřní logiku, udržující pouze stav určitého objektu a umožňující přístup k daným parametrům. Nejčastějším použitích těchto tříd je reprezentace objektů

[3] POJO – Plain Old Java Object

přijatých v odpovědi od serveru či z databáze. Zatímco v Javě je nutné veškeré deklarování provádět manuálně (viz následující ukázka kódu). [23]

```
public class User {
    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

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

    public int getAge() {
        return this.age;
    }

    // metody equals, hashCode, copy nejsou pro úsporu místa zobrazeny,
v kódu by se však nacházely
}
```

Kotlin umožňuje deklaraci takových tříd pomocí uvedení klíčového slova „data“ před deklaraci třídy. Následně stačí pouze vypsat seznam parametrů a kompilátor vygeneruje zbytek kódu jako gettery, settery, metody *hashCode*, *copy* a *equals* automaticky. [23]

```
data class User(var name: String, var age: Int)
```

3.3.3.18 Odlišná rozhraní pro práci s kolekcemi

Kotlin rozlišuje mezi proměnlivými a neměnnými kolekcemi (seznamy, množinami, mapami atd.). Na úrovni kompilace umožňuje kontrolu nad tím, zdali kolekce může být upravena či nikoliv. Původní třídy pro práce s kolekcemi (např. *List*<*T*>, *Map*<*T*>) Kotlin používá pro neměnné kolekce. Metody pro změnu kolekcí jsou obsaženy ve stejnojmenných třídách s prefixem *Mutable*. Použití kolekcí je demonstrováno na příkladu. [17]

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers) // vypíše "[1, 2, 3]"
numbers.add(4)
println(readOnlyView) // vypíše "[1, 2, 3, 4]"
readOnlyView.clear() // -> neprojde kompilací
val strings = hashSetOf("a", "b", "c", "c") assert(strings.size == 3
```

3.3.3.19 Koprogramy

Ve verzi Kotlinu 1.1 byly přidány koprogramy (z angl. *coroutines*). Jedná se o novou funkcionalitu umožňující asynchronní operace bez nutnosti blokování vlákna. Koprogramy v Kotlinu jsou výpočty, které mohou být v případě potřeby pozastaveny bez nutnosti blokování vlákna. Pozastavení může nastat pouze na místech předem definovaných v kódu a je oproti blokování vlákna nenáročné. Funkcionalita je ve fázi testování, a proto je zatím dostupná pouze v experimentálních verzích jazyka. [23]

3.4 Testování výkonu

Testování výkonnosti softwaru je testování, které se provádí s cílem určit, jak se systém chová pod určitou zátěží.

Testování výkonnosti slouží k několika účelům. Může prokázat, zda software splňuje výkonnostní požadavky. Porovnat dva systémy a určit, který pracuje lépe nebo prokázat, které části systému nebo pracovní zátěže způsobují špatné chování systému. V diagnostických případech je možné použít nástroje, *profiler*, na změření, které části zařízení nebo softwaru se nejvíce podílejí na špatné výkonnosti nebo určit úrovně propustnosti systému a prahovou úroveň akceptovaného času odezvy. V případě opakovaného měření výkonu, či porovnávání dvou systémů, lze použít jako nástroj benchmark.

Při testování výkonnosti je často rozhodující a současně těžko dosažitelné, aby byly testovací podmínky podobné jako předpokládané skutečné podmínky nasazení.

3.4.1 Základní principy

Stejně jako při každém procesu, i při testování mohou nastat chyby. Často se tak stává z důvodu použití nesprávné metodiky, nebo chybně nastaveného nástroje. Tato kapitola se věnuje základním principům testování výkonu, jejich dodržením se lze vyhnout nejčastějším chybám. [24]

3.4.1.1 Počet proměnných

Testování může být zcela znehodnoceno, pokud se počet příslušných proměnných nesníží na absolutní minimální počet. Pro experimenty je ideální mít pouze jednu proměnnou. Například napětí, teplota okolí a průtok vzduchu jsou řízeny, zatímco teplota se mění. Mít pouze jednu proměnnou není v reálném světě vždy možné, ale je žádoucí se pokusit o snížení účinků všeho, co by mohlo ovlivnit test. Toho lze dosáhnout tím, že se při testování např. grafické karty odpojí nebo vypne jakékoli jiné přídavné zařízení do počítače nebo se odstraní nepotřebné součásti (např. zvuková karta nebo raidové pole). Pokud nejsou proměnné kontrolovány, může být skutečná přesnost snížena, ačkoliv vnímaná přesnost může být vysoká. [24]

3.4.1.2 Opakovatelnost výsledků

Opakovatelnost testu je nejpoužívanějším ukazatelem toho, jak dobře byl experiment proveden. Pokud jsou při různých testováních dodrženy základní principy a testování dojde ke stejným výsledkům, jsou opakovatelné. To platí společně s přesností a precizností. Pokud více lidí opakuje tentýž test a výsledky těchto testů se blíží výsledkům ostatních testů, vypovídá to o přesnosti experimentu. [24]

3.4.1.3 Velikost vzorku

Čím více testů se provádí za správných podmínek, tím větší je pravděpodobnost, že chybné výsledky (malé chyby v každém jednotlivém testu) ovlivní konečné průměry (průměr, medián, režim atd.). Každá instance produktu či položky by měla být testována

vícekrát a je lepší mít více než jednu položku k provádění testů, protože některé příklady mohou být lepší nebo horší než jiné. [24]

3.4.1.4 Směrodatná odchylka / Standardní chyba

Směrodatná odchylka populace je měřítkem rozložení hodnot dat. Standardní chyba je směrodatná odchylka od kořenů počtu vzorků a výsledkem je procentní podíl. Standardní chyba tedy bere v úvahu počet vzorků v experimentu (čím více, tím lépe). [24]

Nízká směrodatná odchylka nebo standardní chyba znamená, že většina hodnot zaznamenaných během testování je v nízkém rozsahu aritmetického průměru dat. V podstatě, čím menší je směrodatná odchylka nebo standardní chyba, tím více je třeba důvěřovat číselným hodnotám a pokud se namísto směrodatné odchylky použije standardní chyba, hodnotitel bude neuspokojen velikostí vzorku testování. [24]

3.4.2 Virtualizace

Virtualizace je v informatice označení postupů, technik a prostředků, které umožňují v počítači přistupovat k dostupným zdrojům jiným způsobem, než jakým fyzicky existují, jsou propojeny atd. Virtualizované prostředí lze mnohem snadněji přizpůsobit potřebám uživatelů, může být snazší pro použití a umožňuje případně skrýt před uživateli pro ně nepodstatné detaily (jako např. rozmístění hardwarových prostředků). Virtualizovat lze na různých úrovních, od celého počítače (tzv. virtuální stroj), po jeho jednotlivé hardwarové komponenty (např. virtuální procesory, virtuální paměť atd.), případně pouze softwarové prostředí (virtualizace operačního systému). [25]

3.4.2.1 Výhody

Mezi výhody virtualizace operačního systému patří zejména možnost spuštění různých nezávisle běžících operačních systémů nad sdíleným hardwarem. K tomu se váže i vyšší bezpečnost. V případě napadení hostujícího systému je nutné provést nejprve útok na hostovaný operační systém a až poté na samotný virtuální systém. Virtualizovat lze jakýkoli operační systém, neboť podpora ze jeho strany není požadovaná. [25]

V komerčním využití umožňují virtualizační nástroje spravovat virtualizované stroje na mnoha fyzických počítačích a provádět jejich úpravy a migrace včetně jejich připojených souborových systémů. Pomocí grafického uživatelského rozhraní je možné je obsluhovat bez nutnosti hlubokých znalostí virtualizace a souvisejících služeb a aplikací. [25]

3.4.2.2 Nevýhody

Nevýhodou virtualizovaných systémů může být jejich náročnost na výkon a místo. U nedostatečně výkonných počítačů, či jejich virtualizovaných ekvivalentů, se často nevyplatí virtualizaci provozovat. Důvodem je fakt, že virtualizační nástroje požadují alespoň jeden systém pro vlastní běh. Běh tohoto systému může znatelně omezit zdroje požadované pro hostované operační systémy. [25]

Další nevýhodou mohou být vysoké nároky na zprovoznění, instalaci a bezpečné nastavení virtualizace. Vyžadovány jsou zkušenosti a pokročilé administrátorské znalosti nutné ke správnému nastavení firewallů, směrování atp. [25]

3.4.2.3 Virtuální stroj

Virtuální stroj, nebo také virtuální počítač, je emulací počítačového systému. Virtuální počítače jsou založeny na počítačových architekturách a poskytují funkčnost fyzického počítače. Jejich implementace mohou zahrnovat specializovaný hardware, software nebo kombinaci. Existují různé druhy virtuálních strojů, z nichž každá má různé funkce [26]:

- **Systémové virtuální stroje** – někdy také nazývané virtuální virtualizační stroje, poskytují náhradu za skutečný stroj. Poskytují funkce potřebné pro spuštění celého operačního systému. Hypervisor používá nativní spouštění pro sdílení a správu hardwaru, což umožňuje několik navzájem izolovaných prostředí, které však existují na jednom fyzickém počítači. Moderní hypervisory používají hardwarově podporovanou virtualizaci, hardware specifickou pro virtualizaci, primárně z hostitelských CPU. [26]

- **Procesové virtuální stroje** – jsou navrženy tak, aby prováděly počítačové programy v prostředí nezávislém na platformě. [26]

Některé virtuální stroje jsou navrženy tak, aby emulovaly různé architektury a umožňovaly provádění softwarových aplikací a operačních systémů napsaných pro další CPU nebo architekturu. Virtualizace na úrovni operačního systému umožňuje rozdělit zdroje počítače na podporu jádra pro několik izolovaných instancí uživatelského prostoru, které se obvykle nazývají kontejnery a mohou se tvářit jako reálné stroje pro koncové uživatele. [26]

3.4.3 Benchmarking

Benchmarking je proces řízeného spouštění počítačového programu, souboru programů nebo jiných operací, který má za účel vyhodnocení relativní výkonnost daného objektu. Toho je obvykle docíleno pomocí spuštění řady standardních testů a modelových situací proti danému programu. Pojem „benchmark“ je také často používán pro účely vypracování komparativních srovnávacích programů. [27]

Benchmarking je obvykle spojen s hodnocením výkonových charakteristik počítačového hardwaru, např. s výkonem CPU při operacích s plovoucí desetinnou čárkou. Existují však okolnosti, kdy je tato technika použitelná také pro software. Softwarové benchmarky se používají např. pro hodnocení překladačů nebo systémů správy databází. [27]

3.4.3.1 Historie

Jak se počítačová architektura vyvíjela, začalo být čím dál obtížnější porovnávat výkon různých počítačových systémů jednoduše podle jejich specifikace. Proto byly vyvinuty testy, které umožnily srovnání různých architektur. Například procesory Pentium 4 obecně pracují s vyšší frekvencí než procesory Athlon XP, což nemusí nutně znamenat vyšší výpočetní výkon. Pomalejší procesor, pokud jde o frekvenci hodin, může fungovat stejně jako procesor pracující s vyšší frekvencí (viz BogoMips a megahertzský mýtus). [27]

Před rokem 2000 architekti počítačů a mikroprocesorů používali SPEC, přestože benchmarky založené na Unixu byly poměrně zdlouhavé, a tudíž nebyly použitelné. [27]

Je známo, že výrobci počítačů konfigurují své systémy tak, aby poskytovaly nerealisticky vysoký výkon při benchmarkových zkouškách, které nejsou v reálném použití replikovány. Například během 80. let minulého století někteří překladatelé mohli odhalit určitou matematickou operaci používanou ve známém měřítku s plovoucí desetinnou čárkou a operaci vyměnit za rychlejší matematicky ekvivalentní operaci. Nicméně, taková transformace byla zřídka užitečná mimo referenční úroveň až do poloviny 90. let, kdy architektury RISC a VLIW zdůraznily význam kompilátorové technologie, protože se týkala výkonu. Benchmarky jsou nyní pravidelně využívány firmami vyvíjejícími kompilátory, aby zlepšily nejen vlastní skóre benchmarku, ale i skutečnou výkonnost aplikací. [27] [28]

3.4.3.2 Účel

Benchmarky jsou navrženy tak, aby napodobovaly určitý typ pracovní zátěže na nějaké součásti nebo na systému. Syntetické benchmarky toho dosahují pomocí speciálně vytvořených programů, které generují pracovní zátěž na dané komponentě. Aplikační benchmarky pracují s reálnými programy v systému. Zatímco aplikační benchmarky obvykle poskytují mnohem lepší měřítko skutečného výkonu v daném systému, syntetická měřítka jsou užitečná pro testování jednotlivých komponent, jako je pevný disk nebo síťové zařízení. [27] [28]

Hodnoty benchmarku jsou obzvláště důležité v návrhu CPU, což architektům procesorů umožňuje měřit a provádět kompromisy v rozhodnutích mikroarchitektury. Například pokud benchmark získává klíčové algoritmy aplikace, bude obsahovat aspekty citlivé na výkon této aplikace. Spuštění tohoto mnohem menšího úryvku kódu na simulátoru přesného cyklu může poskytnout podklady pro zlepšení výkonu. [27] [28]

3.4.4 Profilování

Profilování je vyhledávání míst v programu, které jsou vhodné pro optimalizaci. Vyhledávání probíhá s pomocí speciálních nástrojů, nazývaných profilery, za běhu

programu (tj. forma dynamické analýzy). Na základě konkrétního požadavku na optimalizaci se sleduje například využití paměti, doba a frekvence provádění různých částí programu, systémová volání a podobně. [29]

Možnosti konkrétního profileru závisí na programovacím jazyku, možnostech kompilátoru a platformě na které sledovaný program běží. Pro efektivní využívaní výstupních dat profileru k optimalizaci programu je důležité brát v potaz možné zkreslení výsledků vzniklé samotným připojením profileru k programu a metodou sběru dat. [29]

Optimalizaci práce s pamětí na základě výstupů z profilování řeší programátor. V programovacích jazycích typu C bez pokročilé správy paměti vyžaduje sledování paměti zásah programátora. Na platformách Java a .NET se při profilování paměti sleduje využívaná paměť, její alokace a uvolnění pomocí garbage collectoru. [29]

Profilery používají širokou škálu technik pro shromažďování dat, včetně hardwarového přerušení, kódové instrumentace, simulace sady instrukcí, záchytných bodů operačního systému a čítačů výkonu. [29]

3.4.5 Gradle

Gradle je software sloužící k automatizovanému sestavování programů. Je vyvíjen jako open-source a zavádí doménově specifický jazyk Groovy. Využívá orientované acyklické grafy (podobně jako např. Git) k určení v jakém pořadí se mají úkony provádět. [30]

Gradle byl navržen pro sestavování vícemodulových aplikací, které tak mohou nabývat větších rozměrů. Podporuje inkrementální sestavení pomocí zjišťování, které části sestavovacího stromu zůstaly nezměněny a které tak není třeba znovu provádět. [30]

V Androidu je tento software hojně využíván pro sestavování aplikací. Konfigurační soubory pro sestavení jsou psány v jazyce Groovy, který se syntakticky velmi podobá Javě. Sestavení tak lze optimalizovat pro daný projekt a konfigurační soubory použít např. pro automatizované sestavování a průběžnou integraci do již zveřejněných verzí programu. [30]

4 Praktická část

Na základě teoretických východisek je v následující kapitole provedeno porovnání jazyků. Kapitola se zaměřuje zejména na analýzu měřitelných vlastností. Definuje testované vlastnosti a postup, jakým byly vlastnosti měřeny. Další část je věnována zvoleným technologiím a důvodu jejich výběru. Následně jsou rozebírány postupy měření jednotlivých vlastností a analýza naměřených dat. Závěrem kapitoly je shrnutí výsledků měření a přehled funkcionalit, které jazyky nabízejí.

4.1 Testované vlastnosti

Testované vlastnosti představují charakteristiky jazyků, případně faktory, které jsou s nimi spojené. Jejich zkoumání umožňuje daným jazykům lépe porozumět a v případě, že se podaří zkoumanou vlastnost kvantifikovat, i oba jazyky na základě takové vlastnosti porovnat.

Výběr vlastností byl proveden na základě několika kritérií. Prvním kritériem bylo, aby vybrané vlastnosti korespondovaly s vlastnostmi, které jsou ceněny v praxi. Není žádoucí testovat vlastnost, která v konečném důsledku není rozhodující pro praktické využití a není zahrnuta v rozhodovacím procesu při výběru vhodného jazyka. Dalším kritériem byla možnost objektivnímu přístupu k testování a relativně snadné možnosti měřitelnosti. V praxi mohou být rozhodujícími vlastnostmi a jinými faktory např. rozšířenost jazyka na trhu práce, či oblíbenost jazyka mezi týmem programátorů, kteří byli na projekt přiděleni (často právě ti bývají rozhodujícím faktorem). Tyto vlastnosti a faktory je obtížné kvantifikovat a jejich měření často podléhá humanitním oborům. Z toho důvodu se práce na tyto faktory nezaměřuje a přenechává je dalším zkoumáním.

Z předešlého textu vyplývá, že metody testování vlastností jsou závislé na dané testované vlastnosti a mohou se tedy velmi lišit. Některé vlastnosti lze měřit snáze než jiné a u některých zůstává pouze subjektivní pohled, či metody, které nejsou pro tuto práci vhodné. Na základě zkušeností autora, byly vybrány následující vlastnosti:

- doba běhu
- doba kompilace (čisté)
- doba kompilace (inkrementální)
- délka zdrojového kódu

4.2 Testovací scénáře

V kontextu této práce byl testovací scénář definován jako metoda pro otestování a měření vybrané testované vlastnosti. Každý testovací scénář se skládá z posloupnosti kroků, v jakých je testování prováděno, a z testovacích případů, na kterých jsou měřeny testované vlastnosti. Testovací případy mohou sestávat z krátkých spustitelných částí kódu, ze zdrojových kódů celé aplikace, nebo z projektového repozitáře.

Pro každý testovací scénář je vytvořen benchmark. Benchmark slouží jako nástroj pro opakované provádění měření. Implementuje jednotlivé navržené kroky a umožnuje automatizované měření. Možnost opakovaného automatizovaného měření je důležitá zejména pro možnost ověření výsledků a rovněž pro eliminaci chyb vzniklých během měření v nedokonalém testovacím prostředí. Proměnlivé prostředí je při měření častým problémem a jeho odstranění je jen velmi obtížné. Při každém měření existují působící vnější vlivy, které nelze ovlivnit a je proto nutné počítat s odchylkou při měření. Z tohoto důvodu bylo měření provedeno opakovaně pomocí benchmarku a výsledky měření byly následně statisticky vyhodnoceny.

Testovací scénáře byly navrženy a vytvořeny individuálně pro měření dané vlastnosti. Benchmarky a testovací případy scénářů se mohou značně lišit, a proto je každému scénáři věnována samostatná kapitola.

4.3 Zvolené technologie

Důležitým faktorem při měření je prostředí, ve kterém bude měření prováděno. Prostředí operačního systému je značně proměnlivé a výkon procesoru kolísá. To může být způsobeno vnějšími vlivy jakými jsou např. služby běžící na pozadí operačního systému. Takové prostředí není pro měření příliš vhodné. Během měření by mohlo dojít k nečekané zátěži a dočasnému snížení výkonu procesoru, které by mohlo ovlivnit proces měření a tím zapříčinit znehodnocení výsledků měření.

Z tohoto důvodu nebylo vhodné použít pro měření osobní počítač autora a bylo nutné uvažovat o jiném stroji. Každé měření je zatíženo prostředím, ve kterém je prováděno, a neexistuje ideální prostředí, ve kterém by se dalo provádět měření bez účinků vnějších vlivů. I speciálně navržený operační systém pouze pro účely měření by byl zatížen teplotními vlivy, které způsobují kolísání výkonu procesoru. Pro přiblížení k ideálním podmínkám měření v neměnném prostředí bylo zvoleno prostředí virtuálního serveru.

Technologie profilování nebyla v práci využita. Důvodem je proměnlivé prostředí osobního počítače, na kterém je software pro profilování nainstalován. Software Android Studio je určen pro osobní počítače a na serverové virtuální stroje jej nelze nainstalovat. Dalším důvodem je jeho zaměření na optimalizaci, na kterou není tato práce zaměřena.

Veškerá práce před a po měření byla provedena na osobním počítači autora. Zdrojové kódy byly vypracovány za použití Android Studia, volně rozšířeném softwaru pro vývoj mobilních aplikací pro Android, a poté přesunuty na virtuální server.

4.3.1 Virtuální server

Veškerá kompilace a benchmarking jsou prováděny na plně virtualizovaném operačním systému Linux. Za distribuci bylo zvoleno Ubuntu ve verzi 16.04 LTS z důvodu jeho rozšířenosti, dostupnosti a zkušeností autora.

Virtualizovaný operační systém byl zvolen pro minimalizaci dopadu vnějších vlivů na proces testování. Operační systém je virtualizovaný službou KVM, hostovaný firmou WEDOS Internet, a.s. a má dostatečný výkon, který eliminuje případný vliv náročnosti

kompilace a benchmarku na systémové prostředky. Jedná se o 15 GB SSD, 2 GB DDR3 dedikované RAM a 1 vlákno procesoru Xeon 1.7 GHz.

Na základě zkušeností autora byly na virtuální server postupně nainstalovány součásti potřebné pro sestavení a běh Java a Kotlin aplikací – Java JDK, Android SDK a software pro sestavení aplikací. Problematika virtualizace, instalace a konfigurace virtuálního serveru jsou rozsáhlá témata a z důvodu omezeného rozsahu práce nejsou dále rozebírána.

4.3.2 Gradle

Pro sestavení aplikací a testovacích případů byl zvolen software Gradle. Vhodná konfigurace softwaru umožnila testování jednotlivých testovacích případů. Software v základní konfiguraci neumožňuje měření doby jednotlivých úkolů. Na základě dokumentace softwaru bylo vypracováno rozšíření v podobě implementace třídy, která tato měření umožňuje. Deklarace třídy a část kódu vyhodnocujícího dobu běhu úkolu je zachycena na obrázku (viz Obrázek 1).

```
class TimingsListener implements TaskExecutionListener, BuildListener {

    ...

    @Override
    void afterExecute(Task task, TaskState taskState) {
        def ms = clock.timeInMs
        timings.add([ms, task.path])
        task.project.logger.warn "${task.path} took ${ms}ms"
    }

    ...
}

gradle.addListener new TimingsListener()
```

Obrázek 1 - Ukázka implementace třídy vyhodnocujícího dobu běhu úkolu

4.3.3 Bash

Z důvodu nadměrného množství testů bylo přistoupeno k automatizaci celého procesu. Dalším důvodem automatizace byla možnost opakovaného spouštění testů, které

umožňuje statistickým přístupem eliminovat chyby měření vzniklé nedokonalým testovacím prostředím.

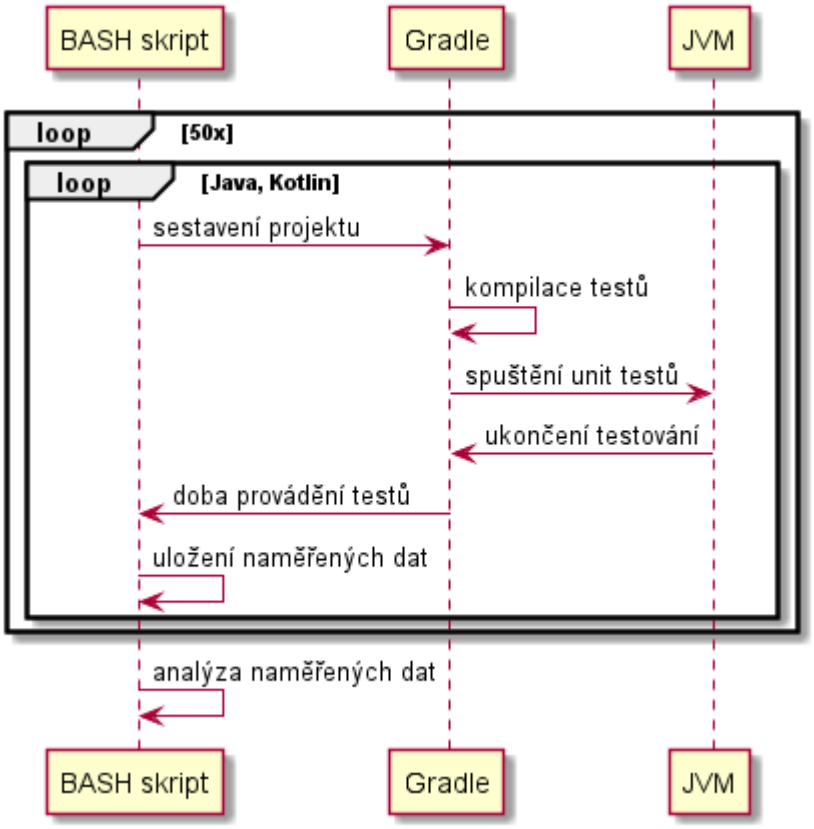
Automatizovaného testování bylo dosaženo pomocí unixového shellu Bash, který umožňuje spouštět předem připravené skripty. Pomocí skriptů byly vyvinuty benchmarky pro každý z testovacích scénářů.

4.4 Testovací scénář – doba běhu programu

Pro testování doby běhu programu byly testovací případy definovány jako nejmenší možné části kódu testující vybranou funkcionalitu daného jazyka. Implementace testovacích případů byla provedena pomocí unit testů. Každý unit test odpovídá jednomu testovacímu případu. Spouštění testů bylo provedeno s využitím sestavovacího softwaru Gradle. Jelikož bylo třeba oddělit dobu běhu programu od ostatních součástí sestavení (např. doby kompilace), bylo nutné přenechat měření doby běhu programu na Gradlu. Software Gradle umožňuje implementovat rozšíření, která mohou běžet během sestavování programu. Pro měření doby běhu jednotlivých úkolů bylo implementováno rozšíření (viz kapitola 4.3.2), které dobu vypočítává z času před spuštěním testu a z času po jeho skončení. Opakované spuštění bylo zajištěno pomocí benchmarku.

4.4.1 Benchmark

Návrh benchmarku spočívá v opakovaném spouštění unit testů. Nejprve se sestaví projekt se zdrojovými kódy Javy. Poté proběhne kompilace unit testů a následné spuštění testů na Java virtuálním stroji. Před samotným spuštěním testů se uloží čas, ve kterém byly testy spuštěny a odečte se od času po skončení testů. Výsledná doba se vrátí Bash skriptu pro uložení. Poté se adekvátními kroky změří doba běhu Kotlinu a celý proces se cyklicky opakuje. Po skončení posledního cyklu proběhne analýza naměřených dat. Zmíněný proces je znázorněn na následujícím diagramu (viz Obrázek 2).



Obrázek 2 – Sekvenční digram benchmarku – doba běhu programu

Implementace benchmarku v jazyce Bash je zobrazena na ukázce (viz Obrázek 3).

```
#!/bin/bash
function runTest()
{
    input=$("${1}"/gradlew testRelease --rerun-tasks)
    input=$(cat input.txt)

    result=$(
        echo "$input" |
        grep "testReleaseUnitTest took" |
        cut -d " " -f3 |
        cut -d "m" -f1
    )
}

...

for (( i=1; i <= REPETITIONS; i++))
do
    echo -ne "| Case\\t${i}\\t"
    runTest "$JAVA_PATH"
    javaTestExecutionTimes[$i]=${result}
    echo -ne "|\\t${result} ms "
    runTest "$KOTLIN_PATH"
    kotlinTestExecutionTimes[$i]=${result}
    echo -ne "|\\t${result} ms "
    echo "|"
done

IFS=$'\n'
...
javaVariance=$(awk -v mean="${javaMean}" '{sum+=$1^2} END {print sum / NR -
mean^2}' <<< "${javaTestExecutionTimes[*]}")
kotlinVariance=$(awk -v mean="${kotlinMean}" '{sum+=$1^2} END {print sum /
NR - mean^2}' <<< "${kotlinTestExecutionTimes[*]}")
javaStdDeviation=$(awk -v x="${javaVariance}" 'BEGIN{print sqrt(x)}')
kotlinStdDeviation=$(awk -v x="${kotlinVariance}" 'BEGIN{print sqrt(x)}')
unset IFS
...
```

Obrázek 3 - Ukázka implementace benchmarku – doba běhu programu

4.4.2 Testovací případy

Na základě testované vlastnosti bylo vytvořeno 20 testovacích případů. Jednotlivé případy se skládají z unit testů implementovaných krátkými spustitelnými částmi kódu, na kterých byla testována doba běhu. Případy testovaly vždy jednu oblast ze základních funkcionalit jazyků či téma spjaté s principy vývoje mobilních aplikací. Pomocí benchmarku byly testovací případy opakovaně spouštěny a dohromady tvořily výsledek běhu programu, který byl následně analyzován.

Oblasti testování spolu s ukázkou implementace v obou jazycích jsou uvedeny v následujícím seznamu vybraných testovacích případů:

- Datové třídy – jeden ze základních principů každé mobilní aplikace, reprezentace dat v objektové struktuře, byla testována deklarací datové třídy a následnou inicializací. Implementace v obou jazycích je zachycena na následujících ukázkách zdrojového kódu.

```java
public class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

@Test
public void test17() {
    Person p = new Person("Name", 10);
    System.out.println(p.getName() + p.getAge());
}
```

```kotlin
data class Person(val name: String, val age: Int)

@Test
fun test17() {
    val p = Person("Name", 10)
    println(p.name + p.age)
}
```

- Kolekce – objekt reprezentující seznam, či jinou posloupnost dat, byl testován nejprve inicializací *ArrayListu*, poté uložením seznamu prvků a následně průchodem všech prvků v seznamu. Obě implementace jsou zachyceny na obrázku.

```java
@Test
public void test15() {
    List<Integer> collection = new ArrayList<>(Arrays.asList(...));
    StringBuilder sb = new StringBuilder();
    sb.append("[");
    Iterator<Integer> iterator = collection.iterator();
    while (iterator.hasNext()) {
        Integer element = iterator.next();
        sb.append(element);
        if (iterator.hasNext()) {
            sb.append(", ");
        }
    }
    sb.append("]");
    System.out.println(sb.toString());
}
```

```kotlin
@Test
fun test15() {
    val collection = ArrayList(Arrays.asList(...))
        val sb = StringBuilder()
        sb.append("[")
        val iterator = collection.iterator()
        while (iterator.hasNext())
        {
            val element = iterator.next()
            sb.append(element)
            if (iterator.hasNext())
            {
                sb.append(", ")
            }
        }
        sb.append("]")
        println(sb.toString())
}
```

- Filtrování seznamů – proces filtrování prvků seznamu, či vyhledávání v seznamu, byl testován podobně jako v předchozím případu – inicializace, naplnění seznamu a poté filtrování dat s následujícím výpisem

```java
@Test
public void test2() {
    List<Integer> list = new ArrayList<>(Arrays.asList(...));
    List<Integer> filtered = new ArrayList<>();
    for (int i : list) {
        if (i > 0) filtered.add(i);
    }
    for (int i : filtered) {
        System.out.println(i);
    }
}
```

```kotlin
@Test
fun test2() {
    val list = ArrayList(Arrays.asList(...))
    val filtered = list.filter { it > 0 }
    filtered.forEach { println("$it") }
}
```

4.4.3 Výsledky měření

Pomocí benchmarku byla otestována doba běhu programu. Opakovaným měřením byl proveden sběr dat pro agregaci a analýzu. Benchmark rovněž provedl výpočet základních statistických charakteristik, které následně zahrnul do tabulky výsledků měření spolu s naměřenými daty. Výstup benchmarku s výsledky měření je zachycen na obrázku.

```
Unit tester run-time benchmark
.-----------------------------------------------.
| Case  #        |     Java [ms] |   Kotlin [ms] |
+----------------+---------------+---------------+
| Case  1        |        119 ms |        117 ms |
| Case  2        |        122 ms |        117 ms |
| Case  3        |        118 ms |        120 ms |
| Case  4        |        119 ms |        117 ms |
| Case  5        |        119 ms |        117 ms |
| Case  6        |        120 ms |        118 ms |
| Case  7        |        120 ms |        117 ms |
| Case  8        |        121 ms |        119 ms |
| Case  9        |        122 ms |        119 ms |
| Case  10       |        122 ms |        119 ms |
| Case  11       |        122 ms |        120 ms |
| Case  12       |        121 ms |        121 ms |
| Case  13       |        118 ms |        122 ms |
| Case  14       |        118 ms |        122 ms |
| Case  15       |        120 ms |        120 ms |
| Case  16       |        122 ms |        121 ms |
| Case  17       |        119 ms |        117 ms |
| Case  18       |        120 ms |        118 ms |
| Case  19       |        120 ms |        122 ms |
| Case  20       |        117 ms |        122 ms |
| Case  21       |        122 ms |        118 ms |
| Case  22       |        120 ms |        122 ms |
| Case  23       |        119 ms |        118 ms |
| Case  24       |        122 ms |        120 ms |
| Case  25       |        121 ms |        120 ms |
+----------------+---------------+---------------+
| Mean           |     120.12 ms |     119.32 ms |
| Median         |     118.00 ms |     122.00 ms |
| Std Deviation |       1.51 ms |       1.83 ms |
.-----------------------------------------------.
```

Obrázek 4 - Výstup benchmarku s výsledky měření (doba běhu programu)

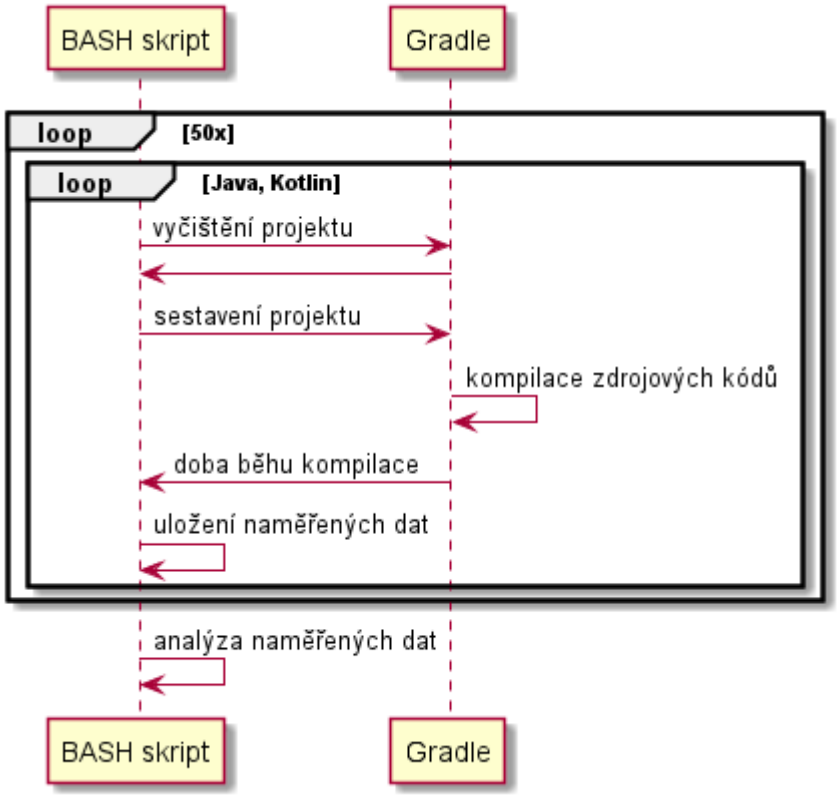
Celkem bylo provedeno 25 měření. Ze základních statistických charakteristik byl vypočten aritmetický průměr, medián a směrodatná odchylka. Směrodatná odchylka se pohybuje v obou případech mezi jedním a dvěma procenty střední hodnoty a prostředí, ve kterém bylo měřeno, lze proto považovat za přílišně nekolísající. Na základě mediánu je doba běhu Kotlinu o 3,38 % vyšší než u Javy.

4.5 Testovací scénář – doba čisté kompilace

K testování doby čisté kompilace byl, obdobně jako v předchozím případě, použit software Gradle. Jako testovací scénáře byly použity celé projekty aplikací. Prvním projektem je soubor unit testů z předchozího scénáře. Další projekt je aplikace Athena, která byla součástí bakalářské práce autora a posledním projektem je osobní aplikace autora sloužící pro trénování slovní zásoby španělského jazyka. Všechny aplikace byly přepsány do Kotlinu s ohledem na zachování stejné funkcionality. Sestavení aplikací bylo provedeno softwarem Gradle. Měření doby kompilace bylo provedeno pomocí navrženého benchmarku.

4.5.1 Benchmark

Navrhovaný benchmark je založen na opakovaném sestavení projektů. Pro každý projekt proběhne nejprve vyčistění, které odstraní zkompilované části projektu a vrátí projekt do stavu před kompilací. Poté proběhne již měřené sestavení projektu. Před spuštění sestavení projektu se uloží čas započetí sestavení, následně proběhne kompilace a po jejím skončení se vypočte doba kompilace odečtením času před započetím kompilace od aktuálního času. Na rozdíl od předchozího scénáře se měření provádí na straně Bash skriptu, a nikoliv na straně Gradlu. Danými kroky se změří nejprve doba kompilace Javy a poté Kotlinu. Celý proces se opakuje, dokud není dosaženo požadovaného množství naměřených dat. Následně proběhne analýza naměřených dat. Sekvenční diagram zmíněného procesu je znázorněn na obrázku (viz Obrázek 5).



Obrázek 5 - Sekvenční digram benchmarku – doba čisté kompilace

4.5.2 Testovací případy

Pro měření čisté doby kompilace nebylo možné použít unit testy s krátkými spustitelnými částmi kódu jako v předchozím scénáři. Doba jejich kompilace by byla příliš krátká a naměřené výsledky by byly neúměrně zatíženy přesností měřících nástrojů. To by se negativně podepsalo na chybovosti naměřených dat.

Z toho důvodu byly jako testovací případy použity celé projekty aplikací. Celkem bylo použito 6 projektů – 3 páry, kde každý pár odpovídá stejné funkcionalitě a implementuje projekty v Javě a v Kotlinu. První dvojice projektů obsahuje veškeré unit testy daného jazyka použité ve scénáři pro testování doby běhu programu. Dalším projektem je aplikace Athena, kterou autor práce vypracoval jako součást bakalářské práce. Původní projekt v Javě byl přepsán do Kotlinu a je součástí druhé dvojice. Jako poslední

byl použit osobní projekt autora s názvem Spanish. Projekt byl rovněž vypracován v Javě a slouží pro trénování slovní zásoby španělského jazyka.

4.5.3 Výsledky měření

Navržený benchmark byl použit k otestování doby čisté kompilace. Měření proběhlo v několika opakováních, v každém byly testovány všechny projekty ve stejném pořadí. Po sběru dat a agregaci provedl benchmark analýzu naměřených dat a výpočet základních statistických charakteristik. Ty byly následně zahrnuty do tabulky zachycující výsledky měření a naměřená data (viz Obrázek 6).

```
Build (clean) time benchmark
.-------------------------------------------------------.
| Project # [iteration] |     Java [ms] |   Kotlin [ms] |
+-----------------------+---------------+---------------+
| [1] UnitTests         |      22672 ms |      26523 ms |
| [1] Athena            |      17445 ms |      19534 ms |
| [1] spanish           |      18127 ms |      20392 ms |
| --> Total [∑]         |    * 58244 ms |    * 66449 ms |
| [2] UnitTests         |      22931 ms |      26578 ms |
| [2] Athena            |      16994 ms |      19037 ms |
| [2] spanish           |      18201 ms |      20916 ms |
| --> Total [∑]         |    * 58126 ms |    * 66531 ms |
| [3] UnitTests         |      23728 ms |      26285 ms |
| [3] Athena            |      17860 ms |      19881 ms |
| [3] spanish           |      17715 ms |      20738 ms |
| --> Total [∑]         |    * 59303 ms |    * 66904 ms |
| [4] UnitTests         |      23045 ms |      25574 ms |
| [4] Athena            |      16864 ms |      18859 ms |
| [4] spanish           |      17765 ms |      20408 ms |
| --> Total [∑]         |    * 57674 ms |    * 64841 ms |
| [5] UnitTests         |      22804 ms |      26711 ms |
| [5] Athena            |      17191 ms |      19079 ms |
| [5] spanish           |      18614 ms |      20320 ms |
| --> Total [∑]         |    * 58609 ms |    * 66110 ms |
| [6] UnitTests         |      22884 ms |      25583 ms |
| [6] Athena            |      16765 ms |      19883 ms |
| [6] spanish           |      18655 ms |      20128 ms |
| --> Total [∑]         |    * 58304 ms |    * 65594 ms |
| [7] UnitTests         |      23774 ms |      26647 ms |
| [7] Athena            |      17621 ms |      19448 ms |
| [7] spanish           |      18512 ms |      20872 ms |
| --> Total [∑]         |    * 59907 ms |    * 66967 ms |
| [8] UnitTests         |      23134 ms |      25649 ms |
| [8] Athena            |      17451 ms |      18961 ms |
| [8] spanish           |      17792 ms |      20792 ms |
| --> Total [∑]         |    * 58377 ms |    * 65402 ms |
| [9] UnitTests         |      23796 ms |      26179 ms |
| [9] Athena            |      16790 ms |      19034 ms |
| [9] spanish           |      18131 ms |      20696 ms |
| --> Total [∑]         |    * 58717 ms |    * 65909 ms |
| [10] UnitTests        |      23485 ms |      26630 ms |
| [10] Athena           |      17121 ms |      19469 ms |
| [10] spanish          |      17692 ms |      19971 ms |
| --> Total [∑]         |    * 58298 ms |    * 66070 ms |
+-----------------------+---------------+---------------+
| Mean                  |   58555.90 ms |   66077.70 ms |
| Median                |   58456.50 ms |   65852.00 ms |
| Std Deviation         |     602.84 ms |     637.69 ms |
.-------------------------------------------------------.
```

Obrázek 6 - Výstup benchmarku s výsledky měření (doba čisté kompilace)

Počet opakování byl stanoven na 10 cyklů pro každý projekt. Pro naměřená data byl stanoven aritmetický průměr, medián a směrodatná odchylka. Vzhledem k nízké směrodatné odchylce (v obou případech mezi jedním a dvěma procenty střední hodnoty)

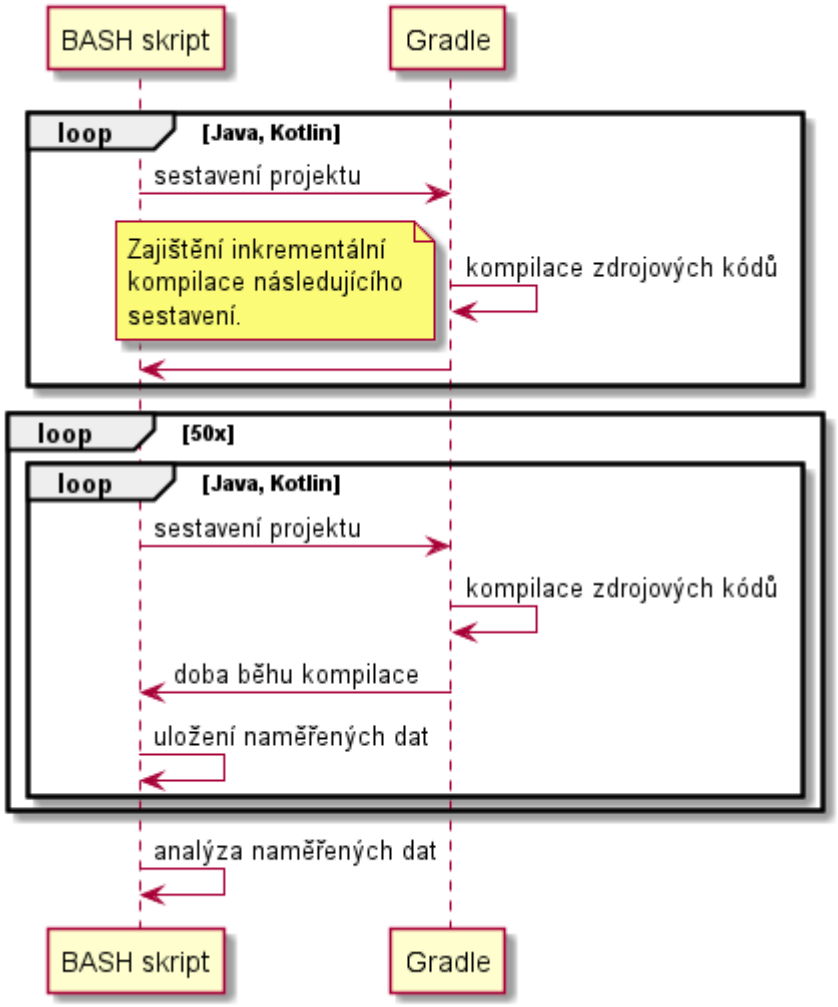
lze prostředí, ve kterém bylo měření prováděno, považovat za výkonnostně stabilní. Nárůst doby čisté kompilace (na základě mediánu) je u Kotlinu přibližně 12,65 %.

4.6 Testovací scénář – doba inkrementální kompilace

Testování doby inkrementální kompilace je obdobou testování doby čisté kompilace. Pro sestavení byl rovněž byl použit software Gradle. Jako testovací případy byly opět použity projekty Athena a osobní aplikace autora. K sestavení aplikací byl použit software Gradle. Na rozdíl od čisté kompilace bylo nutné vyřešit, jak zaručit, aby kompilace byla inkrementální a nenastal případ kompilace čisté, jejíž doba by znehodnotila statistická data. Pro tento případ bylo nutné provést čistou kompilace před započetím měření. K tomuto účelu byl rovněž použit software Gradle.

4.6.1 Benchmark

Benchmark pro testování inkrementální kompilace je obdobou benchmarku pro testování kompilace čisté. Rozdílem je zde prvotní fáze testovacího procesu. Zatímco u čisté kompilace bylo nutné zaručit, že v projektu se nevyskytují již zkompilované zdrojové kódy, u inkrementální kompilace byla tato vlastnost žádoucí. Z toho důvodu bylo před započetím testu nutné projekt zkompilovat a vytvořit tak prvotní zkompilované zdrojové kódy, nad kterými proběhne inkrementální kompilace. Zbytek procesu je shodný s čistou kompilací. Diagram procesu je znázorněn na obrázku (viz Obrázek 7).



Obrázek 7 - Sekvenční digram benchmarku – doba inkrementální kompilace

4.6.2 Testovací případy

Testovací případy pro dobu inkrementální kompilace se shodují s případy pro dobu kompilace čisté (viz Kapitola 4.5.2).

4.6.3 Výsledky měření

Testování doby čisté kompilace bylo provedeno pomocí benchmarku. Měření proběhlo v několika cyklech, v každém byly projekty testovány ve stejném pořadí (shodném s pořadím v předchozím scénáři). Po skončení měření a agregaci dat, byla benchmarkem provedena analýza a výpočet základních statistických charakteristik. Výstup

benchmarku s výsledky měření a vypočtenými charakteristikami jsou zachyceny v tabulce (viz Obrázek 8).

```
Build (incremental) time benchmark
.-------------------------------------------------------.
| Project # [iteration] |     Java [ms] |   Kotlin [ms] |
+-----------------------+---------------+---------------+
| [1] UnitTests         |       8340 ms |       8122 ms |
| [1] Athena            |       5967 ms |       6060 ms |
| [1] spanish           |       7007 ms |       6893 ms |
| --> Total [∑]         |    * 21314 ms |    * 21075 ms |
| [2] UnitTests         |       8465 ms |       7801 ms |
| [2] Athena            |       5940 ms |       5702 ms |
| [2] spanish           |       6792 ms |       6895 ms |
| --> Total [∑]         |    * 21197 ms |    * 20398 ms |
| [3] UnitTests         |       8151 ms |       7738 ms |
| [3] Athena            |       6314 ms |       6057 ms |
| [3] spanish           |       6766 ms |       6772 ms |
| --> Total [∑]         |    * 21231 ms |    * 20567 ms |
| [4] UnitTests         |       8064 ms |       7922 ms |
| [4] Athena            |       6245 ms |       5737 ms |
| [4] spanish           |       7209 ms |       6912 ms |
| --> Total [∑]         |    * 21518 ms |    * 20571 ms |
| [5] UnitTests         |       8484 ms |       7977 ms |
| [5] Athena            |       6346 ms |       5889 ms |
| [5] spanish           |       7156 ms |       6895 ms |
| --> Total [∑]         |    * 21986 ms |    * 20761 ms |
| [6] UnitTests         |       8237 ms |       7807 ms |
| [6] Athena            |       5935 ms |       5870 ms |
| [6] spanish           |       7173 ms |       6632 ms |
| --> Total [∑]         |    * 21345 ms |    * 20309 ms |
| [7] UnitTests         |       8056 ms |       7995 ms |
| [7] Athena            |       6160 ms |       5728 ms |
| [7] spanish           |       7227 ms |       7001 ms |
| --> Total [∑]         |    * 21443 ms |    * 20724 ms |
| [8] UnitTests         |       8083 ms |       7735 ms |
| [8] Athena            |       5986 ms |       5899 ms |
| [8] spanish           |       6749 ms |       6628 ms |
| --> Total [∑]         |    * 20818 ms |    * 20262 ms |
| [9] UnitTests         |       8192 ms |       8194 ms |
| [9] Athena            |       6029 ms |       5781 ms |
| [9] spanish           |       6824 ms |       6640 ms |
| --> Total [∑]         |    * 21045 ms |    * 20615 ms |
| [10] UnitTests        |       8382 ms |       8066 ms |
| [10] Athena           |       6311 ms |       5929 ms |
| [10] spanish          |       6930 ms |       6775 ms |
| --> Total [∑]         |    * 21623 ms |    * 20770 ms |
+-----------------------+---------------+---------------+
| Mean                  |   21352.00 ms |   20605.20 ms |
| Median                |   21665.50 ms |   20535.00 ms |
| Std Deviation         |     305.09 ms |     231.83 ms |
'-------------------------------------------------------'
```

Obrázek 8 - Výstup benchmarku s výsledky měření (doba inkrementální kompilace)

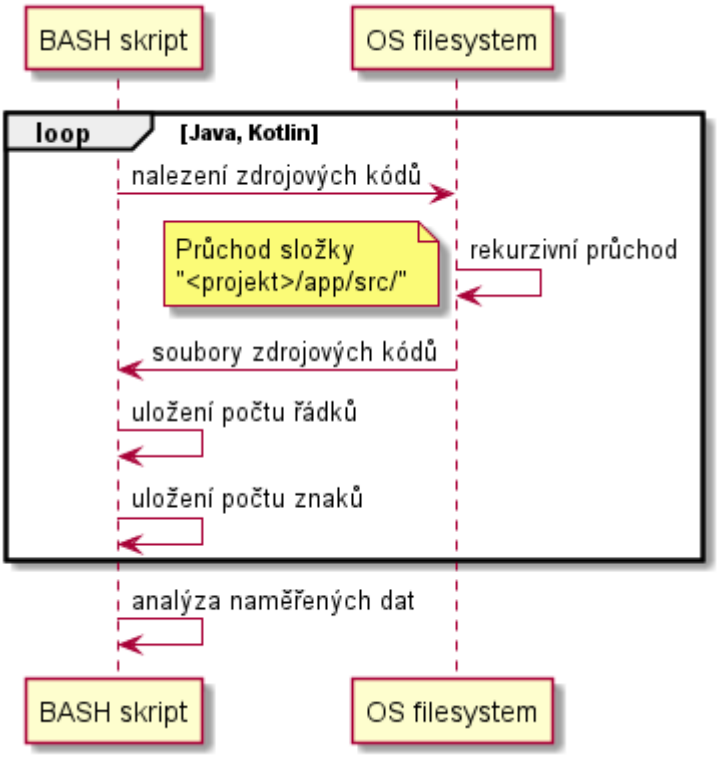
Měření doby kompilace projektů probíhalo v 10 cyklech. Z naměřených dat byl vypočten aritmetický průměr, medián a směrodatná odchylka. Nízká směrodatná odchylka (mezi jedním a dvěma procenty střední hodnoty) signalizuje vhodný výběr testovacího prostředí a nízké poklesy výkonu. Doba inkrementální kompilace je u Kotlinu o 5,21 % nižší než u Javy.

4.7 Testovací scénář – délka zdrojového kódu

Testování délky zdrojového kódu bylo z výše uvedených scénářů nejjednodušší. Sestavení nebylo nutné, a proto nebyl software Gradle použit. Rovněž se nejedná o nedeterministickou vlastnost, tudíž nebylo nutné provádět opakovaná měření. Délka zdrojového kódu je neměnná, a proto ji stačilo změřit pouze jednou. Měření bylo prováděno na všech projektech použitých ve výše zmíněných testovacích scénářích. Bylo tak učiněno z toho důvodu, že se ve všech případech jednalo o kód plnící stejnou funkcionalitu, tudíž bylo možno oba kódy porovnávat na jejich délku.

4.7.1 Benchmark

Benchmarkem je v tomto případě pouze Bash skript, který prochází strukturu projektů a hledá zdrojové kódy. K tomuto účelu byly využity funkce souborového systému v operačním systému Linux. Pro každý projekt byly nejprve nalezeny zdrojové kódy ve složce nalézající se v projektovém adresáři */app/src/*. Pro každý soubor byl zjištěn počet řádků a počet znaků kódu. Hodnoty těchto vlastností byly následně sečteny pro každý projekt a uloženy do pole. Po skončení průchodu všemi projekty byla data analyzována a byl vypočten průměr mezi zjištěnými hodnotami. Sekvenční diagram benchmarku je znázorněn na obrázku (viz Obrázek 9)



Obrázek 9 - Sekvenční digram benchmarku – délka zdrojového kódu

4.7.2 Testovací případy

Testovací případy se skládají z projektů použitých ve výše zmíněných testovacích scénářích – z projektů unit testů a aplikací Athena a Spanish. Na testovacích případech byla testována délka kódu, která byla definována jako počet řádků a počet znaků. Počet řádků je zaběhnutým měřítkem v programátorské komunitě, avšak může být zavádějící v případě, kdy je informace skryta na delším řádku. V případě, kdy počet řádků klesá úměrně s počtem znaků, se dá jazyk pokládat za natolik expresivní, že umožňuje vyjádřit složité struktury pomocí jednoho slova či výrazu.

Podobná situace nastává i u přirozených (lingvistických) jazyků. Složitější, expresivnější jazyky, umožňují jednoslovné vyjádření, které by bylo nutné v jednodušších jazycích vyjádřit pomocí sousloví nebo celé věty. Typickým příkladem mohou být jazyky jako čínština či japonština, u kterých je nutné při překladu určitých znaků či slov do angličtiny, nebo i do jazyka českého, používat dlouhé věty. Podle rodilých mluvčích však často ani ty zcela nevystihují původní význam.

4.7.3 Výsledky měření

Vypracovaný benchmark byl použit k měření délky kódu testovacích případů. Pojem „měření“ může být v daném kontextu poněkud zavádějící, neboť se jedná o deterministické vlastnosti, které lze jednoznačně určit výpočtem. Z toho důvodu nebylo nutné měření opakovat a pro určení hodnoty testovaných případů stačil pouze jeden průchod. Po průchodu všemi projekty a vyčíslení délky kódu, byla data analyzována a výsledky zobrazeny v tabulce (viz Obrázek 10).

```
Code length benchmark
.-------------------------------------------------------------------------.
| Project   | Java [lines] | Kotlin [lines] | Java [chars] | Kotlin [chars] |
+-----------+--------------+----------------+--------------+----------------+
| UnitTests |          436 |            291 |        31785 |          27535 |
| spanish   |          559 |            425 |        15762 |          13474 |
| Athena    |         2539 |           2229 |        91769 |          79459 |
+-----------+--------------+----------------+--------------+----------------+
| Mean      |         1178 |            981 |        46438 |          40156 |
.-------------------------------------------------------------------------.
```

Obrázek 10 - Výstup benchmarku s výsledky měření (délka zdrojového kódu)

Byl vypočten průměr mezi zjištěnými hodnotami a poměr mezi naměřenými hodnotami testovaných vlastností. Na základě průměru bylo vypočteno, že kód v Kotlinu má přibližně o 16,7 % nižší počet řádků a o 13,5 % nižší počet znaků.

4.8 Shrnutí výsledků měření

Shrnutí výsledků měření předchozích kapitol zobrazuje absolutní změny mezi vlastnostmi testovaných jazyků. Výsledky měření testovacích scénářů jsou uvedeny v tabulce (viz Tabulka 3). Testovacímu scénáři, určenému k měření délky zdrojového kódu, byla vyčleněna samostatná tabulka (viz Tabulka 4).

Tabulka 3 - Shrnutí výsledků měření 1

Testovaná vlastnost	Java	Kotlin
Medián průměrné doby běhu	118.00 ms	122.00 ms
Medián průměrné doby kompilace (čistá)	58456.50 ms	65852.00 ms
Medián průměrné doby kompilace (inkrementální)	21665.50 ms	20535.00 ms

Tabulka 4 - Shrnutí výsledků měření 2

Délka zdrojového kódu	Java	Kotlin
Průměrný počet řádků	1178	981
Průměrný počet znaků	46438	40156

4.9 Přehled funkcionalit

Následující kapitola shrnuje poznatky z teoretické části a vytváří přehled nabízených funkcionalit obou jazyků. Zaměřuje se zejména na funkcionality, které jeden z jazyků nabízí a které druhému chybí. Podkladem pro přehled funkcionalit byla pravidla syntaxe z teoretické části. V tabulkách jsou uvedeny názvy funkcionalit s korespondujícími kapitolami.

4.9.1 Java

Tabulka 5 - Přehled funkcionalit (Java)

Funkcionalita	Kapitola
Checked exceptions	3.2.4.1
Primitivní datové typy, které nejsou třídou	3.2.4.2
Statické metody	3.2.4.3
Neprivátní atributy	3.2.4.4
Wildcard typy	3.2.4.5
Ternární operátor	3.2.4.6

4.9.2 Kotlin

Tabulka 6 - Přehled funkcionalit (Kotlin)

Funkcionalita	Kapitola
High-order funkce a lambda výrazy	3.3.3.1
Inline funkce	3.3.3.2
Rozšiřující funkce	3.3.3.3
Nullová bezpečnost	3.3.3.4
Smart casts	3.3.3.5
String templates	3.3.3.6
Properties	3.3.3.7
Primární konstruktory	3.3.3.8
First-class delegation	3.3.3.9
Typová odvození pro proměnné a atributy	3.3.3.10
Singletons	3.3.3.11
Declaration-site variance & Type projections	3.3.3.12
Intervalové výrazy	3.3.3.13
Přetěžování operátorů	3.3.3.14
Infixové notace metod	3.3.3.15
Companion objekty	3.3.3.16
Datové třídy	3.3.3.17
Odlišná rozhraní pro práci s kolekcemi	3.3.3.18
Koprogramy	**Chyba! Nenalezen zdroj odkazů.**

5 Zhodnocení výsledků a doporučení

Výsledkem praktické části práce jsou definice testovacích scénářů a testovacích případů, návrh a implementace benchmarků pro testování scénářů a výsledky měření. Testovací scénáře, případy a návrhy benchmarků s výsledky měření jsou k dispozici v jednotlivých kapitolách. Implementace testovacích případů a benchmarků jsou dostupné v příloze (viz Příloha 1 – CD s implementací testovacích případů a benchmarků). Ukázka z implementace benchmarku testovacího scénáře pro dobu běhu programu je dostupná v kapitole 4.4.1.

Zhodnocení výsledků testování výkonu bylo provedeno komparativním způsobem. Jako výchozí hodnoty byly použity výsledky měření testovacích případů Javy a na základě výsledků měření testovacích případů Kotlinu byly vypočteny absolutní a relativní změny.

Výsledky měření testovacích scénářů jsou uvedeny v tabulce (viz Tabulka 7). Bylo zjištěno, že doba běhu programu v Kotlinu je o cca 3 % vyšší. Vzhledem k účelům práce a posouzení jazyka Kotlin pouze z hlediska vývoje mobilních aplikací, lze považovat nárůst za zanedbatelný a konstatovat, že doba běhu Kotlinu je shodná s Javou.

Znatelný rozdíl nastává u doby čisté kompilace. U Kotlinu byl naměřen pokles výkonu o necelých 13 %. Čisté kompilace se však v praxi provádějí jen výjimečně – většinou při sestavování produkčních verzí aplikací, při prvotním sestavení programu, nebo při rozsáhlé změně konfiguračních souborů pro sestavení.

Výsledek měření inkrementální doby kompilace byl poněkud překvapivý. Zatímco u doby čisté kompilace Kotlin zaostával za Javou, u inkrementální byl rychlejší o více než 5 %. Důvodem jsou pravděpodobně optimalizace překladače a linkeru Kotlinu, které převyšují své protějšky v Javě. Autoři jazyka Kotlin si dávali za cíl dobu kompilace alespoň stejně rychlou jako u Javy. Na základě výsledků měření lze potvrdit, že se jim cíl podařilo splnit.

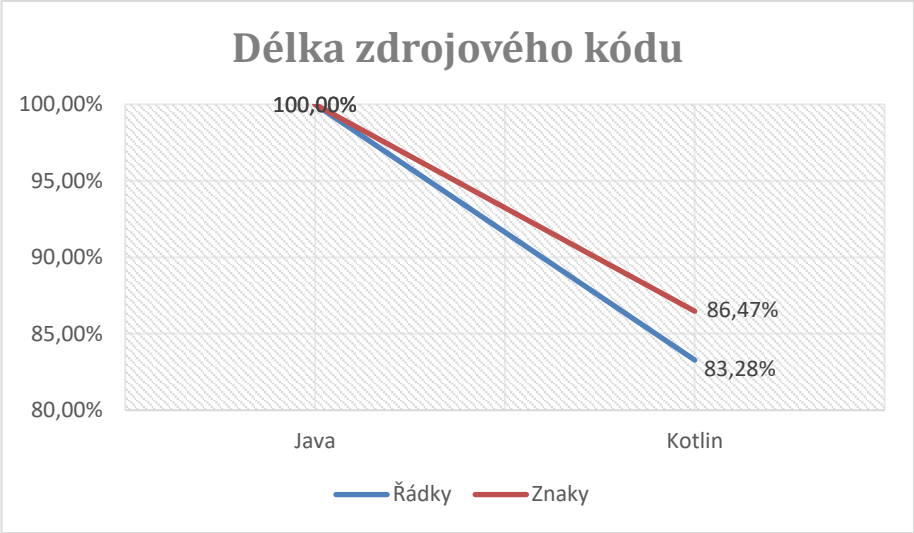
Tabulka 7 - Výsledky měření 1 - absolutní a relativní změny

Testovaná vlastnost	Absolutní	Relativní
Doba běhu	4 ms	+3,38 %
Doba kompilace (čistá)	7395,5 ms	+12,65 %
Doba kompilace (inkrementální)	-1130,5 ms	-5,21 %

Testovacímu scénáři, určenému k měření délky zdrojového kódu, byla vyčleněna samostatná tabulka (viz Tabulka 8). Sledovanými vlastnostmi byl počet řádků a počet znaků kódu. V obou případech byl u Kotlinu patrný pokles o cca 15 %. Zajímavým pozorováním byl však poměr mezi těmito dvěma vlastnostmi. Zatímco počet řádků klesl o necelých 17 %, počet znaků pouze o necelých 14 %. Počet řádků tedy klesá rychleji než počet znaků. Graf poklesu průměrného počtu řádků a průměrného počtu znaků je zobrazen níže (viz Obrázek 11).

Tabulka 8 - Výsledky měření 2 - absolutní a relativní změny

Délka zdrojového kódu	Absolutní	Relativní
Průměrný počet řádků	-197	-16,7 %
Průměrný počet znaků	-6282	-13,5 %



Obrázek 11 - Graf změny průměrného počtu řádků a průměrného počtu znaků

Vypracovaný přehled funkcionalit (viz Kapitola 4.9) zachycuje výhody, které jeden z jazyků nabízí a které naopak druhý postrádá. Jelikož přehled neobsahuje společné charakteristiky jazyků, nebylo vhodné použít vícekriteriální analýzu variant (a následné stanovení vah např. bodovou metodou).

Doporučením pro rozšíření práce může být definice dalších testovacích scénářů a odpovídajících benchmarků. Vzhledem k nabízeným funkcionalitám umožňuje Kotlin vytvářet pokročilé architektonické a implementační struktury. Zajímavými testovanými vlastnostmi by tedy mohly být např. počet souborů zdrojového kódu, průměrná délka zdrojového kódu na soubor, počet úrovní v hierarchii tříd nebo množství tzv. *boilerplate* kódu.

Dalším doporučeným postupem je implementace vyššího množství testovacích případů pro testovací scénář doby běhu programu. Zvýšení doby běhu programu by vedlo ke snížení směrodatné odchylky a ke zvýšení přesnosti výsledků.

Řešením problematiky porovnání čitelnosti a náročnosti vypracování kódu je provedení širší analýzy odborných informačních zdrojů a odpovídajících metodik, případně vypracování případové studie.

6 Závěr

Jednotlivé dílčí cíle práce byly adresovány v samostatných kapitolách. Kapitola o platformě Android je pohledem na aktuální možnosti vývoje mobilních Android aplikací z hlediska programovacích jazyků, jejich historie a vývoje do budoucnosti. Pasáže Kotlin a Java charakterizují historii a vývoj, možnosti jazyků a jejich metody použití a detailněji se zabývají nabízenými funkcionalitami. Kapitola testování výkonu se věnuje problematice virtualizace a nástrojům vhodným k testování.

Praktická část práce se zaměřuje zejména na analýzu měřitelných vlastností. Definuje testovací scénáře, pomocí nichž jsou vlastnosti měřeny a pro každý scénář navrhuje benchmark jako automatizovaný nástroj měření. Implementuje testovací scénáře a navržené benchmarky a provádí opakované měření. Výsledky měření agreguje a provádí analýzu na základě statistických charakteristik naměřených dat.

Výsledky testování výkonu prokázaly, že doba běhu programů v Kotlinu je shodná s Javou. Doba čisté kompilace zdrojového kódu Kotlinu je delší než u Javy. Inkrementální kompilace, která je v praxi používána nejčastěji, se ukázala být nepatrně kratší než u Javy. Dalším výsledkem byla délka zdrojového kódu. Bylo zjištěno, že jak počet řádků, tak počet znaků kódu je u Kotlinu nižší než u Javy. Na základě výsledků testování lze Kotlin považovat za jazyk se stejným, případně lepším, výkonem než Java.

Porovnání jazyků na základě funkcionalit nebylo v práci provedeno, jelikož nebyla nalezena metoda objektivního přístupu. Práce se zabývala především funkcionalitami, které jeden z jazyků nabízí a druhý postrádá. Z toho důvodu nebylo možné použít komparativní metody vícekriteriální analýzy variant. Doporučením autora je provedení širší analýzy odborných informačních zdrojů a odpovídajících metodik, případně vypracování případové studie.

Na základě zkušeností autora s vývojem mobilních aplikací v Javě lze prohlásit, že Kotlin oproti Javě žádné podstatné funkcionality nepostrádá. Po zkušenostech získaných studiem odborných informačních zdrojů a vypracováním implementací autor konstatoval, že Kotlin umožňuje stejné přístupy k vývoji aplikací jako Java a nabízí i některé další.

Problematika čitelnosti kódu a náročnosti vypracování byla autorem vyhodnocena jako velice subjektivní a z důvodu nedostatku materiálů pro objektivní posouzení, nebyla samostatně řešena. Nicméně, na základě zjištění, že počet řádků u Kotlinu klesá vzhledem k Javě rychleji než počet znaků, lze v kombinaci s teoretickými východisky o nabízených funkcionalitách usuzovat, že Kotlin je expresivnějším jazykem, který umožňuje zápis složitějších struktur Javy pomocí jednoho slova či výrazu. Takové jazyky mají obvykle složitější pravidla, čímž kladou vyšší nároky na uživatele jazyka, a tedy i na jeho čitelnost. Náročnost vypracování, je-li chápána jako množství stráveného času, tak závisí na zkušenostech a dovednostech programátora, který daný jazyk používá.

Hlavním cílem práce bylo zhodnocení jazyka Kotlin a porovnání s jazykem Java z hlediska výkonu a funkcionalit, které jazyky nabízí. Na základě výše zmíněných lze konstatovat, že cíle práce byly splněny.

V průběhu tvorby této práce Android tým na konferenci Google I/O oficiálně oznámil podporu jazyka Kotlin pro vývoj Android aplikací. Tím úspěšně eliminoval obavy komunity o budoucnost jazyka Kotlin. Android Studio, oficiální nástroj vydávaný společností Google, již vývoj v Kotlinu podporuje a dá se očekávat, že podpora jazyka Kotlin bude i nadále pokračovat.

7 Seznam použitých zdrojů

[1] *Google Play Store: number of apps 2009-2017*. Statista - The Statistics Portal for Market Data, Market Research and Market Studies [online]. Statista [cit. 27.03.2018]. Dostupné z: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[2] GARGENTA, Marko a NAKAMURA, Masumi. *Learning Android. Second edition*. Sebastopol, CA: O'Reilly Media, 2014. ISBN 978-1-449-31923-6.

[3] MALÁN, Jaroslav. *Tvorba aplikací na platformě .net pro OS Android*. Plzeň, 2013. Bakalářská práce. Západočeská univerzita v Plzni. Fakulta elektrotechnická.

[4] EADICICCO, Lisa. *THE RISE OF ANDROID: How a flailing startup became the world's biggest computing platform* [online]. Business Insider, 2015 [cit. 14.8.2017]. Dostupné z: http://www.businessinsider.com/how-android-was-created-2015-3

[5] EHRINGER, David. *The Dalvik virtual machine architecture* [online]. Mindful Mischief [cit. 14.8.2017]. Dostupné z: http://www.davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf

[6] MULLIN, Joe. *Google wins crucial API ruling, Oracle's case decimated* [online]. Ars Technica [cit. 14.8.2017]. Dostupné z: https://arstechnica.com/tech-policy/2012/05/google-wins-crucial-api-ruling-oracles-case-decimated/

[7] UJBÁNYAI, Miroslav. *Programujeme pro Android*. Praha: Grada, 2012. Průvodce (Grada). ISBN 978-80-247-3995-3.

[8] KUŠKA, Petr. *Mobilní aplikace pro rozpoznávání kulturních plodin*. Praha, 2015. Bakalářská práce. Česká zemědělská univerzita v Praze. Provozně ekonomická fakulta. Vedoucí práce Josef Pavlíček.

[9] *Dashboards* [online] Android Developers, Google LLC [cit. 22.3.2018]. Dostupné z: https://developer.android.com/about/dashboards/index.html

[10] KLAUSEN, Paul. *Java 1: Basic syntax and semantics: Software Development*. 1st edition. Bookboon.com, 2017. ISBN 978-87-403-1689-6

[11] *TIOBE Index*. TIOBE - The Software Quality Company [online]. 5633 AJ Eindhoven, The Netherlands: Victory House II, 2017 [cit. 2017-08-14]. Dostupné z: https://www.tiobe.com/tiobe-index/

[12] *About Java*. [online]. 500 Oracle Parkway Redwood Shores, CA 94065: Oracle Corporation, 2017 [cit. 2017-08-14]. Dostupné z: http://www.java.com/en/about/

[13] SOUTHWICK, Karen. *High noon: the inside story of Scott McNealy and the rise of Sun Microsystems*. New York: John Wiley, 1999. ISBN isbn0471297135.

[14] *Java Technology: The Early Years* [online] Sun Microsystems [cit. 15.8.2017]. Dostupné z: https://web.archive.org/web/20080530073139/http://java.sun.com/features/1998/05/birthday.html

[15] VENNERS, Bill. *Inside the Java Virtual Machine* [online]. Artima [cit. 21.03.2018]. Dostupné z: https://www.artima.com/insidejvm/ed2/index.html

[16] *Java Platform Standard Edition 7 Documentation. Java SE Documentation* [online]. 500 Oracle Parkway Redwood Shores, CA 94065: Oracle Corporation, 2016 [cit. 2017-08-14]. Dostupné z: http://docs.oracle.com/javase/7/docs/

[17] *Comparison to Java - Kotlin Programming Language* [online]. JetBrains, 2018 [cit. 27.3.2018] Dostupné z: https://kotlinlang.org/docs/reference/comparison-to-java.html

[18] LEIVA, Antonio. *Kotlin for Android Developers: Learn Kotlin the easy way while developing an Android App*. N Charleston SC: CreateSpace Independent Publishing Platform, 2016. ISBN 978-1530075614.

[19] *The Kotlin Language: 1.0 Beta is Here!*. Kotlin Blog [online]. Na hřebenech II 1718/10, 14000 Prague 4, Czech Republic: JetBrains, 2015 [cit. 2017-08-14]. Dostupné z: https://blog.jetbrains.com/kotlin/2015/11/the-kotlin-language-1-0-beta-is-here/

[20] *Kotlin 1.0 Released: Pragmatic Language for JVM and Android*. Kotlin Blog [online]. Na hřebenech II 1718/10, 14000 Prague 4, Czech Republic: JetBrains, 2015 [cit. 2017-08-14]. Dostupné z: https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/

[21] *Kotlin on Android*. Kotlin Blog [online]. Na hřebenech II 1718/10, 14000 Prague 4, Czech Republic: JetBrains, 2015 [cit. 2017-08-14]. Dostupné z: https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/

[22] *Kotlin Programming Language* [online]. JetBrains, 2018 [cit. 27.3.2018] Dostupné z: https://kotlinlang.org/docs/reference/

[23] LIMA, Pedro. *Kotlin vs. Java: First Impressions Using Kotlin for a Commercial Android Project* [online]. 303 2nd Street San Francisco, CA 94107: ArcTouch, 2016 [cit. 2017-08-14]. Dostupné z: https://arctouch.com/2017/05/kotlin-vs-java/

[24] *Hardware Testing and Benchmarking*, 2008 [online]. Donutey [cit. 21.3.2018]. Dostupné z: https://web.archive.org/web/20080205031133/http://www.donutey.com/hardwaretesting.php

[25] PORTNOY, Matthew. *Virtualization essentials*. Indianapolis, Indiana: John Wiley & Sons, 2012. Serious skills. ISBN 978-1118176719.

[26] SMITH, James E. a Ravi. NAIR. *Virtual machines: versatile platforms for systems and processes*. Boston: Morgan Kaufmann Publishers, 2005. ISBN isbn1-55860-910-5.

[27] NAMBIAR, Raghunath, POESS Meikel. *Performance evaluation and benchmarking: First TPC Technology Conference*, TPCTC 2009, Lyon, France, August 24-28, 2009 : revised selected papers. New York: Springer, c2009. Lecture notes in computer science, 5895. ISBN 978-3-642-10423-7.

[28] EHLIAR, Andreas a LIU, Dake. *Benchmarking network processors* [online]. Linköping University S-581 83 Linköping, Sweden [cit. 23.3.2018]. Dostupné z: http://www.da.isy.liu.se/pubs/ehliar/ehliar-ssocc2004.pdf

[29] GRAHAM, Susan, KESSLER, Peter, MCKUSICK, Marshall. *gprof: a Call Graph Execution Profiler* [online]. University of California, Berkeley [cit. 27.3.2018]. Dostupné z: https://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf

[30] *Gradle Build Tool* [online]. Gradle [cit. 27.03.2018]. Dostupné z: https://gradle.org/

8 Přílohy

8.1 Seznam obrázků

8.2 Seznam tabulek

8.3 Další přílohy

8.3.1 Příloha 1 – CD s implementací testovacích případů a benchmarků