JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 15 (2020) 189-195
Received: 1 April 2020
Accepted: 9 April 2020

Performance analysis of languages working on Java Virtual Machine based on Java, Scala and Kotlin

Analiza wydajności języków działających na Java Virtual Machine na przykładzie języków Java, Scala i Kotlin

Katarzyna Buszewicz*

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
This article presents the results of a literature study related to the construction and operation of Java Virtual Machine, as well as performance tests of selected languages using the aforementioned runtime environment on the example of Java, Scala and Kotlin. Performance testing was carried out using two applications built using the Apache Maven archetype with the built-in Java Microbenchmark Harness library.

Keywords: Java; Scala; Kotlin; Performance

Streszczenie
Niniejszy artykuł przedstawia wyniki badania literaturowego związanego z budową oraz działaniem Java Virtual Machine, jak również badania wydajności wybranych języków, wykorzystujących wspomniane środowisko uruchomieniowe na przykładzie Javy, Scali oraz Kotlina. Badanie wydajności przeprowadzono wykorzystując dwie aplikacje, zbudowane przy użyciu archetypu Apache Maven z wbudowaną biblioteką Java Microbenchmark Harness.

Słowa kluczowe: Java; Scala; Kotlin; wydajność

*Corresponding author
Email address: **buszewicz.katarzyna@gmail.com** (K. Buszewicz)

1. Wstęp

W 2020 roku minie 25 lat od powstania języka Java. Przez wiele lat była najpopularniejszym językiem programowania [1], a podczas jej instalacji wyświetlany jest slogan „3 billion devices run Java". Na tak wielki sukces wpłynęło wiele czynników, między innymi jego środowisko uruchomieniowe, jakim jest Java Virtual Machine. Umożliwia ono translację na kod maszynowy, zgodny z architekturą platformy, na której aplikacja jest uruchamiana, a więc pozwala na działanie raz napisanego kodu na dowolnym urządzeniu z dostępem do JVM.

Jednak nie tylko Java może działać na swojej maszynie wirtualnej. Na przestrzeni lat powstało wiele nowych języków, uruchamianych za pośrednictwem wspomnianego środowiska uruchomieniowego. Część z nich opartych jest na Javie i rozwija ją o nowe elementy tudzież eliminuje konieczność pisania powtarzającego się kodu, a także najczęściej występujące wyjątki. Przykładami takich języków są Scala, która swe zastosowanie znalazła głównie w Big Data oraz Kotlin, ogłoszony 17 maja 2017 jako oficjalnie wspierany język do programowania na systemy Android [2].

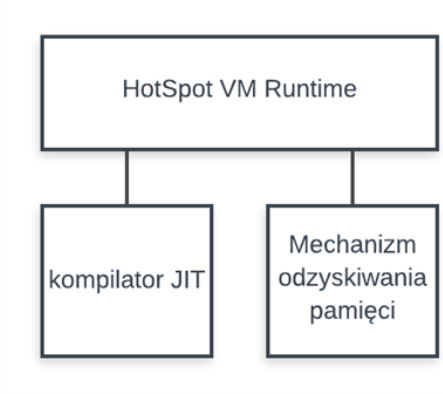
2. Cel badań

Celem badań zaprezentowanych w niniejszym artykule było porównanie wydajności Javy oraz wybranych języków, będących jej udoskonalonymi wersjami na przykładzie Scali oraz Kotlina. Badania zrealizowano zgodnie z wiedzą na temat działania środowiska uruchomieniowego, wspólnego dla wspomnianych języków, jakim jest Java Virtual Machine, co było możliwe dzięki zastosowaniu biblioteki Java Microbenchmark Harness.

3. Java Virtual Machine

Najważniejszym założeniem twórców Javy było stworzenie języka zapewniającego przenaszalność, co promowano sloganem „napisz raz, uruchom gdziekolwiek" (ang. WORA - write once, run anywhere) [3]. Koniecznym do spełnienia tego wymogu był moduł pośredniczący pomiędzy wysokopoziomowym kodem Javy oraz kodem maszynowym, wykonywanym przez procesor. Moduł ten, z punktu widzenia każdej aplikacji Java, powinien stanowić abstrakcję procesora [4]. W tym celu opracowano specyfikację Java Virtual Machine, stanowiącą podstawę licznych implementacji, z których najpopularniejszą obecnie jest Oracle HotSpot VM [5].

Wysokopoziomowymi elementami składowymi wirtualnej maszyny Java, przedstawionymi na Rysunku 1, są: kompilator Just-In-Time, środowisko uruchomieniowe VM Runtime oraz mechanizm odzyskiwania pamięci [6]. Występuje ona w wersji 32 lub 64-bitowej, konfigurowalna jest bogatym zestawem flag (mogących różnić się w zależności od implementacji). Dzięki wprowadzeniu instrukcji invokedynamic w wersji siódmej pozwala na interpretację zmiennych dynamicznie typowanych, co umożliwia uruchamianie skryptów tworzonych w językach interpretowanych, jak na przykład JavaScript.

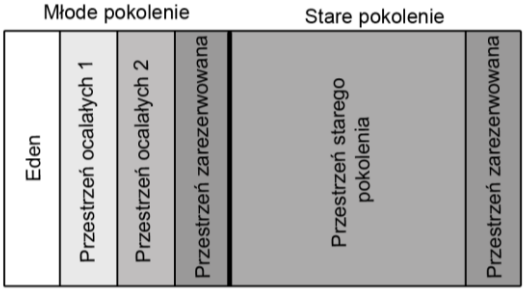


Rysunek 1: Podstawowe elementy budowy Java HotSpot VM

HotSpot VM Runtime, oprócz weryfikacji oraz interpretacji kodu bajtowego zawartego w plikach z rozszerzeniem .class, zapewnia ładowanie klas, analizę argumentów wiersza poleceń, obsługę wyjątków, wielowątkowość, jak również kontrolę cyklu życia maszyny wirtualnej oraz obsługę jej błędów krytycznych [6]. Zapewnia więc pełną kontrolę cyklu życia aplikacji od momentu uruchomienia, aż do momentu usunięcia powiązanych instancji maszyny wirtualnej i wątków oraz zwolnienia zajmowanych zasobów sprzętowych. Dostarczony do uruchomienia **kod bajtowy nie musi pochodzić z kompilacji kodu napisanego w Javie, o ile jest zgodny ze specyfikacją** – to właśnie ta właściwość otworzyła pole twórcom innych języków.

Za dodatkowe optymalizacje kodu oraz kompilację do kodu natywnego, zgodnego z architekturą platformy, na jakiej jest on uruchamiany, odpowiada **kompilator Just-In-Time**. Jego wykorzystanie jest czasochłonne i nie jest opłacalne dla fragmentów kodu, które są rzadko wywoływane. Z tego względu, w początkowej fazie działania aplikacji przeprowadzane jest tak zwane „**rozgrzewanie**" maszyny wirtualnej [7], polegające na zliczaniu wywołań metod oraz określanie, które z nich są „gorącymi punktami" (ang. hot spots). To stąd zaczerpnięto nazwę maszyny wirtualnej Oracle – HotSpot.

Trzecim elementem składowej maszyny wirtualnej Java jest mechanizm zarządzania pamięcią – **Garbage Collector** („odśmiecacz"). Obiektem jego zainteresowania jest sterta, na której przechowywane są obiekty powstałe podczas działania aplikacji. Jej struktura, zaprezentowana na Rysunku 2, budowana jest zgodnie z tak zwaną hipotezą śmiertelności niemowląt, zgodnie z którą obiekty najmłodsze mają znacznie mniejsze szanse na przetrwanie niż obiekty starsze [6], wobec tego sterta dzielona jest na generację młodego pokolenia oraz generację starego pokolenia. Obszary pamięci zarezerwowanej wykorzystywane są do dynamicznych zmian dostępnego rozmiaru sterty. Uruchomienie Garbage Collector'a wiąże się z przerwą w działaniu aplikacji, z tego względu opracowano cztery różne algorytmy, przeznaczone zarówno dla aplikacji wymagających wysokiej przepustowości (algorytm równoległy, wszystkie wątki aplikacji muszą być dostępne),takich, które wymagają jak krótszego czasu odpowiedzi (algorytm szeregowy, jeden wątek zajęty jest odśmiecaniem, a pozostałe obsługują aplikację), a także wymagających korzyści płynących z obu wspomnianych typów (algorytm CMS oraz algorytm G1). Podczas procesu odzyskiwania pamięci usuwane są nieużywane już obiekty, a te, które będą jeszcze potrzebne, przenoszone są do kolejnych sektorów sterty (przestrzeń starego pokolenia) [6].



Rysunek 2: Budowa sterty Java

4. Metoda badania wydajności

Charakterystyka pracy maszyny wirtualnej Java, opisana szerzej w Rozdziale 2, narzuca na sposób badania wydajności kodu pewne wymagania. Okazuje się, że różnica czasu pobranego przed wykonaniem operacji oraz po wykonaniu operacji jest miarą niewystarczającą ze względu na rozgrzewanie maszyny. Jednokrotne uruchomienie kodu nie odzwierciedla jego pracy w środowisku produkcyjnym, nie pozwala na osiągnięcie progu wywołań umożliwiającego dokonywanie optymalizacji przez kompilator Just-In-Time. Kolejnym problemem w przypadku badań okazuje się zdolność kompilatora do eliminowania martwego kodu - jeśli wynik działania metody do testów nie jest nigdzie użyty, taka metoda zostanie usunięta.

Aby rozwiązać wyżej opisane problemy, wykorzystano bibliotekę Java Microbenchmark Harness (w skrócie: JHM) autorstwa Alekseya Shipilëva, współtwórcy kompilatora JIT [8]. Biblioteka stworzona jest w języku Java, lecz zapewnia kompatybilność z innymi językami działającymi na JVM. Użycie jej wymaga odpowiedniej konfiguracji, między innymi wykorzystującej odrębną metodę main(), lecz dzięki wykorzystaniu archetypów dla narzędzia Apache Maven [9] możliwe jest utworzenie jednolicie skonfigurowanych projektów zarówno w Javie, w Scali, jak również w Kotlinie. Umożliwia ona między innymi symulację procesu rozgrzewania maszyny wirtualnej, oznaczenie metody jako hot spot, wsparcie do omijania eliminacji martwego kodu dzięki klasie Blackhole, imitującej konsumenta zwracanych przez metody testowe danych, jak również ustawienia wyników pomiaru, konfigurację ilości wątków czy zestawu parametrów. Wykorzystanie wspomnianej biblioteki do testowania wydajności rekomendowane jest także przez twórców Scali [10].

W niniejszej pracy zbadano wydajność aplikacji, rozumianą jako ilość operacji wykonanych w jednostce czasu. Wszystkie pomiary zostały wykonane na laptopie podłączonym do zasilacza, z trybem maksymalnej wydajności, wyposażonym w procesor Intel Core i5-8300H o taktowaniu 2,3 GHz, 16 GB pamięci operacyjnej SO-DIMM DDR4 2666MHz, z systemem Windows 10 64

b. Badane aplikacje umieszczone były na dysku HDD - 5400 obrotów na s. Do badań wykorzystano następujące wersje języków: Javę 1.8 [5], Scalę 2.13.1 [10] oraz Kotlin 1.3.61 [11].

Badanie wydajności przeprowadzono wykorzystując dwie różne aplikacje, każdą z nich zaimplementowano w każdym z badanych języków, każdą uruchomiono dla trzech różnych przypadków. Testy wydajności uruchamiane były jednowątkowo, po dwa cykle, składające się z pięciu iteracji rozgrzewkowych oraz dziesięciu iteracji pomiarowych - konfigurację tę można zaobserwować w użytych adnotacjach z biblioteki Java Microbenchmark Harness na każdym, niżej zamieszczonym listingu.

Pierwsza aplikacja służy do wyliczenia zadanego wyrazu ciągu Fibonacciego metodą rekurencyjną. Jest to rozwiązanie nieoptymalne, o złożoności O(2n). Warto zaznaczyć, że Scala oraz Kotlin posiadają wsparcie rekursji ogonowej, dzięki której algorytm ten mógłby zostać znacznie zoptymalizowany. Aplikacje w Javie i Scali składają się z trzech klas: Fibonacci, posiadającej właściwą implementację algorytmu wyliczającego wartość wyrazu ciągu Fibonacciego, MyState zawierającej przypadki testowe (numery wyrazów ciągu do wyliczenia) oraz FibonacciBenchmark (w przypadku Kotlina singleton ten umieszczono w jednym pliku razem z metodą do pomiaru), zawierającej metodę do uruchamiania pomiaru wraz z odpowiednią konfiguracją. Przykład 1 prezentuje implementację w Javie, Przykład 2 prezentuje implementację w Scali, a Przykład 3 prezentuje implementację w Kotlinie. Pomiary przeprowadzono dla wyliczenia wyrazu piątego, trzydziestego oraz pięćdziesiątego.

Listing 1: Ciąg Fibonacciego - Java

```
public class Fibonacci {
//Właściwa metoda wyliczająca wyraz ciągu
//Fibonacciego
 public static long countFibonacciSequence(int
number) {
    if(number <= 1) return number;
    else return countFibonacciSequence(number - 2)
    + countFibonacciSequence(number - 1);
  }
}

import org.openjdk.jmh.annotations.Param;
import org.openjdk.jmh.annotations.Scope;
import org.openjdk.jmh.annotations.State;

//parametry do wywołań - numery wyrazów do wyliczenia
@State(Scope.Thread)
public class MyState {
   @Param({"5", "30","50"})
   public Integer iterations;
}
//Klasa testująca z konsumentem danych - Blackhole
// (choć biblioteka umieści go automatycznie)
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;
import java.util.concurrent.TimeUnit;

public class FibonacciBenchmark extends MyState{
   @Benchmark
   @BenchmarkMode(Mode.Throughput)
   @OutputTimeUnit(TimeUnit.SECONDS)
   @Warmup(iterations = 5)
   @Threads(1)
   @Fork(2)
   @Measurement(iterations = 10)
   public void testFibonacci(Blackhole blackhole) {

blackhole.consume(Fibonacci.countFibonacciSequence(it
erations));
   }
}
```

Listing 2: Ciąg Fibonacciego - Scala

```
object Fibonacci {
//Właściwa metoda wyliczająca wyraz ciągu
Fibonacciego
 def countFibonacciSequence(number: Int) : Long = {
   if(number <= 1) return number
        countFibonacciSequence(number - 2)
        + countFibonacciSequence(number - 1)
 }
}
import org.openjdk.jmh.annotations.{Param, Scope}
@org.openjdk.jmh.annotations.State(Scope.Benchmark)
class MyState {
//parametry do wywołań - numery wyrazów do wyliczenia
 @Param(Array("5", "30", "50"))
 var iterations: Int = 0
}

//Klasa testująca z konsumentem danych - Blackhole
import java.util.concurrent.TimeUnit
import org.openjdk.jmh.annotations._

class FibonacciBenchmark extends MyState {
@Benchmark
@BenchmarkMode(Array(Mode.Throughput))
@OutputTimeUnit(TimeUnit.SECONDS)
@Warmup(iterations = 5)
@Threads(1)
@Fork(2)
@Measurement(iterations = 10)
def testFibonacci(state : MyState): Any =
Fibonacci.countFibonacciSequence(state.iterations)
}
```

Listing 3: Ciąg Fibonacciego - Kotlin

```
object Fibonacci {
//Właściwa metoda wyliczająca wyraz ciągu
Fibonacciego
  fun countFibonacciSequence(
numerWyrazu: Int): Long {
   return if (numerWyrazu <= 1)
     numerWyrazu.toLong()
   else
     countFibonacciSequence(numerWyrazu - 2)
     + countFibonacciSequence(numerWyrazu - 1)
  }
}

/Klasa testująca z konsumentem danych - Blackhole
import com.kbuszewicz.Fibonacci
import org.openjdk.jmh.annotations.*
import java.util.concurrent.TimeUnit
```

```
//parametry do wywołań - numery wyrazów do wyliczenia
@State(Scope.Thread)
open class MyState {
   @Param(  "5", "30", "50") var size: Int = 0
}
open class FibonacciBenchmark : MyState() {
   @Benchmark
   @BenchmarkMode(Mode.Throughput)
   @OutputTimeUnit(TimeUnit.SECONDS)
   @Warmup(iterations = 5)
   @Threads(1)
   @Fork(2)
   @Measurement(iterations = 10)
   fun countFibonacciSequence(): Long {
    return Fibonacci.countFibonacciSequence(size)
   }
}
```

Druga aplikacja ma na celu sprawdzenie wydajności języków w pracy z wyrażeniami regularnymi. Wykorzystuje ona wyrażenie zbudowane zgodnie z normą RFC 5322 [12], służące do testowania poprawności adresów e-mail. Każda implementacja składa się z dwóch plików: TestData, zawierającego przypadki testowe oraz MyBenchmark, zawierającego metodę testującą wraz z konfiguracją. Badanie przeprowadzono na trzech różnych przypadkach:

- Przypadek 1: adres.mail:"/><(){}[].mail@domena;';.ABC.domenowa.pl
- Przypadek 2: poprawny.adres@domena.pl
- Przypadek 3: krotki@adres.pl.

Listing 4: Test wyrażeń regularnych - Java

```
//Klasa zawierająca przypadki testowe
import org.openjdk.jmh.annotations.Param;
import org.openjdk.jmh.annotations.Scope;
import org.openjdk.jmh.annotations.State;
@State(Scope.Thread)
public class TestData {
@Param({
"adres.mail:\"/><(){}[].mail@domena;';.ABC.domenowa.p
l",
           "poprawny.adres@domena.pl",
           "krotki@adres.pl"})
   public  String data;
}

//Test wyrażenia regularnego
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

public class MyBenchmark extends TestData{
   @Benchmark
   @BenchmarkMode(Mode.Throughput)
   @OutputTimeUnit(TimeUnit.SECONDS)
   @Warmup(iterations = 5)
   @Threads(1)
   @Fork(2)
   @Measurement(iterations = 10)
   public Boolean checkRegex() {
       String regex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}~-
]+(?:\\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|\"(?:[\\x01-
\\x08\\x0b\\x0c\\x0e-\\x1f\\x21\\x23-\\x5b\\x5d-
\\x7f]|\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-
\\x7f])*\")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?|\\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?|[a-z0-9-]*[a-z0-9]:(?:[\\x01-
\\x08\\x0b\\x0c\\x0e-\\x1f\\x21-\\x5a\\x53-
\\x7f]|\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-
\\x7f])+)\\])";
       return data.matches(regex);
   }
}
```

Listing 5: Test wyrażeń regularnych – Scala

```
//Klasa zawierająca przypadki testowe
import org.openjdk.jmh.annotations.{Param, Scope,
State}

@State(Scope.Thread)
class TestData {
@Param(Array(
"adres.mail:\"/><(){}[].mail@domena;';.ABC.domenowa.p
l",
   "poprawny.adres@domena.pl",
   "krotki@adres.pl"))
 var data : String = ""
}

//Test wyrażenia regularnego
import java.util.concurrent.TimeUnit
import org.openjdk.jmh.annotations._

class MyBenchmark extends TestData {
 @Benchmark
 @BenchmarkMode(Array(Mode.Throughput))
 @OutputTimeUnit(TimeUnit.SECONDS)
 @Warmup(iterations = 5)
 @Threads(1)
 @Fork(2)
 @Measurement(iterations = 10)
 def checkRegex() : Boolean = {
   val regex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}~-
]+(?:\\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|\"(?:[\\x01-
\\x08\\x0b\\x0c\\x0e-\\x1f\\x21\\x23-\\x5b\\x5d-
\\x7f]|\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-
\\x7f])*\")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?|\\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?|[a-z0-9-]*[a-z0-9]:(?:[\\x01-
\\x08\\x0b\\x0c\\x0e-\\x1f\\x21-\\x5a\\x53-
\\x7f]|\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-
\\x7f])+)\\])".r
   regex.pattern.matcher(data).matches()
 }
}
```

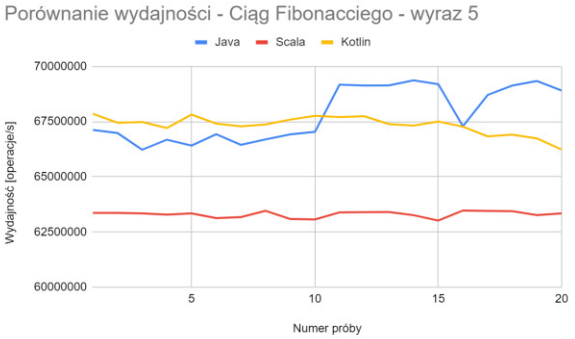
Listing 6: Test wyrażeń regularnych – Kotlin

```
//Klasa zawierająca przypadki testowe
import org.openjdk.jmh.annotations.*

@State(Scope.Thread)
open class TestData {
@Param(
"adres.mail:\"/><(){}[].mail@domena;';.ABC.domenowa.p
l",
           "poprawny.adres@domena.pl",
           "krotki@adres.pl")
   var data : String = ""
}
//Test wyrażenia regularnego
import org.openjdk.jmh.annotations.*
```

```
import java.util.concurrent.TimeUnit

open class MyBenchmark : TestData() {
   @Benchmark
   @BenchmarkMode(Mode.Throughput)
   @OutputTimeUnit(TimeUnit.SECONDS)
   @Warmup(iterations = 5)
   @Threads(1)
   @Fork(2)
   @Measurement(iterations = 10)
   fun checkRegex(): Boolean {
       val regex = "(?:[a-z0-9!#\$%&'*+/=?^_`{|}~-
]+(?:\\.[a-z0-9!#\$%&'*+/=?^_`{|}~-]+)*|\"(?:[\\x01-
\\x08\\x0b\\x0c\\x0e-\\x1f\\x21\\x23-\\x5b\\x5d-
\\x7f]|\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-
\\x7f])*\")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?|\\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?|[a-z0-9-]*[a-z0-9]:(?:[\\x01-
\\x08\\x0b\\x0c\\x0e-\\x1f\\x21-\\x5a\\x53-
\\x7f]|\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-\\x7f])+)\\])"
       return data.matches(regex.toRegex())
   }
}
```

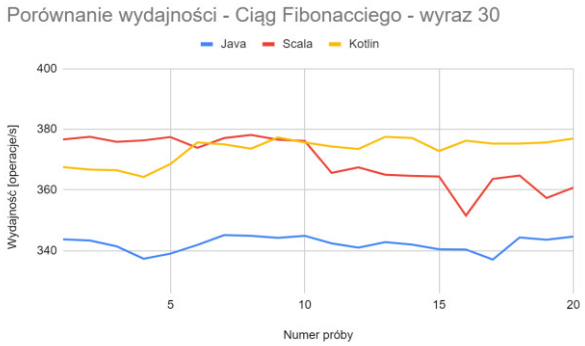
5. Wyniki badań

Zamieszczone poniżej rysunki 3-5 prezentują wyniki pomiarów wydajności aplikacji wyliczającej wartość zadanego wyraz ciągu Fibonacciego. Wynika z nich, że w większości przypadków Java okazała się mniej wydajna od pozostałych języków, które uzyskały zbliżone wartości. Porównanie wszystkich trzech wykresów ukazuje, jak bardzo nieoptymalny jest zastosowany algorytm rekurencyjny. Dla wyliczania wyrazu 50. wydajność wszystkich języków znacząco spadła, co przekłada się także na mniejszą dokładność otrzymanych wyników.

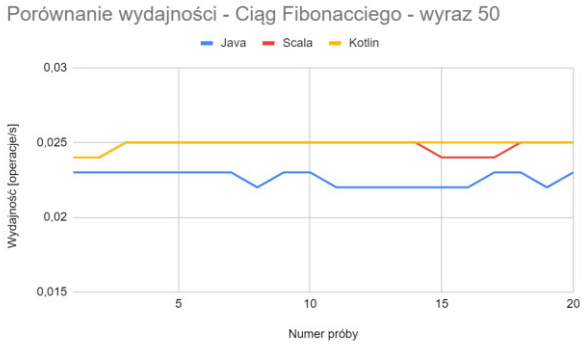


Rysunek 3: Porównanie wydajności – Ciąg Fibonacciego – wyraz 5

Tabela 1 przedstawia miary statystyczne dla uzyskanych pomiarów. Wynika z niej, że Kotlin w dwóch przypadkach okazał się najbardziej wydajnym językiem. W jednym przypadku była to Java. Scala w dwóch przypadkach plasuje się pomiędzy wynikami pozostałych języków, lecz jej rezultaty zbliżone są do wyników Kotlina. Pogrubieniem oznaczono najwyższe średnie wyników.



Rysunek 4: Porównanie wydajności – Ciąg Fibonacciego – wyraz 30



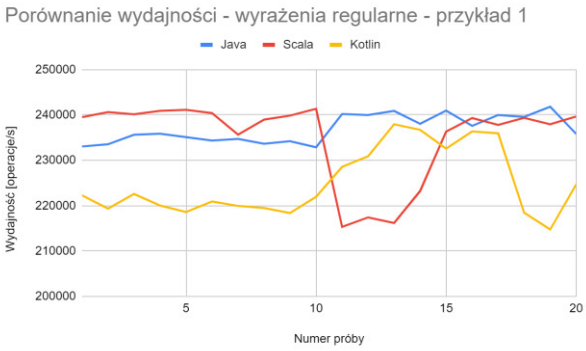
Rysunek 5: Porównanie wydajności – Ciąg Fibonacciego – wyraz 50

Tabela 1: Miary statystyczne – Ciąg Fibonacciego

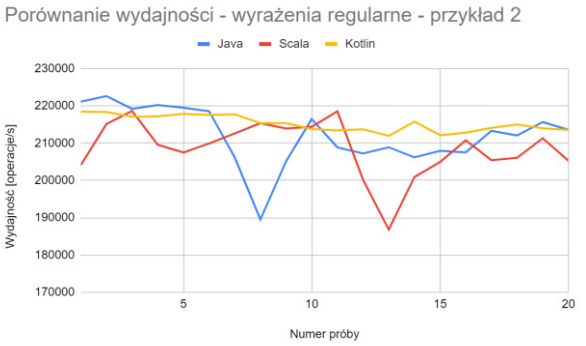
Ciąg Fibonacciego – wyraz 5			
	Java [op/s]	Scala [op/s]	Kotlin [op/s]
Średnia	**67841531**	63302384	67342733
Odchylenie standardowe	1218689	138881	407298
Ciąg Fibonacciego – wyraz 30			
	Java [op/s]	Scala [op/s]	Kotlin [op/s]
Średnia	342,2261	369,5602	**373,3022**
Odchylenie standardowe	2,4362	7,9624	4,1432
Ciąg Fibonacciego – wyraz 50			
	Java [op/s]	Scala [op/s]	Kotlin [op/s]
Średnia	0,0226	0,0248	**0,0249**
Odchylenie standardowe	0,0005	0,0004	0,0003

Kolejne próbki w badaniu wydajności aplikacji testującej wyrażenia regularne, zaprezentowane na rysunkach 6-8 okazały się mieć znaczne wahania wartości, szczególnie w przypadku próbek 11-20. We wszystkich trzech przypadkach Java uzyskała najmniejsze wahania. Najsłabiej pod tym względem prezentuje się Scala.

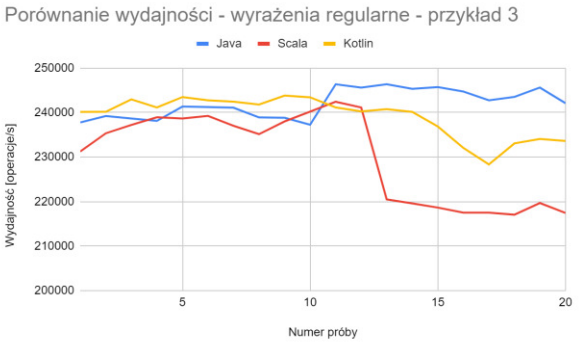
Tabela 2. prezentuje miary statystyczne dla uzyskanych pomiarów, zgodnie z którymi Java uzyskała w dwóch przypadkach największą wydajność przy jednocześnie najmniejszych wahaniach wartości, jednak wartości średnie wydajności wszystkich języków są do siebie zbliżone. W jednym przypadku najbardziej wydajny okazał się Kotlin. Pogrubieniem oznaczono wyniki o najwyższej średniej wartości.



Rysunek 6: Porównanie wydajności wyrażenie regularne – przykład 1



Rysunek 7: Porównanie wydajności – wyrażenie regularne, przykład 2



Rysunek 8: Porównanie wydajności – wyrażenie regularne, przykład 3

Tabela 2: Miary statystyczne – wyrażenia regularne

Wyrażenie regularne – przykład 1			
	Java [op/s]	Scala [op/s]	Kotlin [op/s]
Średnia	**236894**	235056	225022
Odchylenie standardowe	3027,2212	8965,2080	7410,1065
Wyrażenie regularne – przykład 2			
	Java [op/s]	Scala [op/s]	Kotlin [op/s]
Średnia	211979	208555	**215243**
Odchylenie standardowe	7779,2440	7423,2723	2123,2199
Wyrażenie regularne – przykład 3			
	Java [op/s]	Scala [op/s]	Kotlin [op/s]
Średnia	**242030**	230130	239129
Odchylenie standardowe	3203,9172	10039,6288	4498,0414

6. Wnioski

Podsumowując, przeprowadzone badania nie wykazały znacznych różnic w wydajności wszystkich badanych języków, lecz Java oraz Kotlin uzyskały po trzy razy najlepsze średnie wyniki na sześć przeprowadzonych testów. Scala trzykrotnie uzyskała najsłabszy średni wynik.

Podobne dane zaprezentowano w artykule na podstawie pracy magisterskiej [13], porównującym wybrane implementacje Java z serwisu [14] z implementacjami przekonwertowanymi z Kotlina oraz własnymi, stworzonymi w sposób zgodny ze stylem oraz możliwościami oferowanymi przez Kotlin. Dla sześciu przebadanych algorytmów, implementacja Javy okazała się szybsza niż obie implementacje w Kotlinie, a w kolejnych dwóch przypadkach implementacja Javy oraz implementacja własna autora wykazały zbliżony czas wykonania algorytmu, jednak dłuższy niż w przypadku implementacji będącej konwersją aplikacji Java do Kotlina. Przytoczone wyżej badania nie zostały przeprowadzone przy użyciu biblioteki Java Benchmark Harness.

Przeprowadzanie badań wydajności aplikacji działających na Java Virtual Machine wiąże się z koniecznością uwzględnienia charakterystyki jej działania. Obarczone jest to także błędami wynikającymi z pracy ekosystemu, w jakim badania są przeprowadzane, a więc innych procesów tworzących przerwania i powodujących zmianę przetwarzanego przez procesor wątku. Wpływ na wyniki mają także dostępne zasoby sprzętowe.

Obecnie na rynku dostępne są narzędzia wspierające przeprowadzanie testów wydajności, takie jak biblioteka Java Benchmark Harness, pozwalające na uzyskanie możliwie najbardziej obiektywnych wyników, uwzględniających zasady działania środowiska uruchomieniowego JVM.

Literatura

[1] Data is beautiful: Most popular programming languages 1965 - 2019 https://www.youtube.com/watch?v=Og847HVwRSI [18.10.2019]

[2] TechCrunch.com: Google makes Kotlin a first class language for writing android apps https://techcrunch.com/2017/05/17/google-makes-kotlin-a-first-class-language-for-writing-android-apps/ [18.10.2019]

[3] B. Evans, Java: The Legend, Wydawnictwo O'Reilly Media, Sebastopol 2015

[4] J. Engel, Programming for the Java™ Virtual Machine, Addison Wesley, Boston 1999

[5] T. Lindholm, The Java Virtual Machine Specification. Java SE 8 Edition, Oracle Parkway, Redwood City 2014

[6] Hunt C., John B.: Wydajność Javy, Wydawnictwo Helion, Gliwice 2013

[7] J. Kubryński, Co każdy programista Java powinien wiedzieć o JVM, Programista, Nr 3/2015, s. 24-27

[8] Tutorials.Jenkov.com: JMH - Java Microbenchmark Harness http://tutorials.jenkov.com/java-performance/jmh.html [09.02.2020]

[9] Dokumentacja Apache Maven https://maven.apache.org/guides [09.02.2020]

[10] Dokumentacja języka Scala https://docs.scala-lang.org/ [12.01.2020]

[11] Dokumentacja języka Kotlin https://kotlinlang.org/docs/reference/ [26.01.2020]

[12] Dokumentacja normy RFC 5322 https://tools.ietf.org/html/rfc5322 [29.02.2020]

[13] Medium.com, J. Anioł - Java vs. Kotlin — Part 1: Performance: https://medium.com/rsq-technologies/comparative-evaluation-of-selected-constructs-in-java-and-kotlin-part-1-dynamic-metrics-2592820ce80 [12.03.2020]

[14] The Computer Language Benchmarks Game: Java versus Kotlin fastests programs https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/java-kotlin.html [12.03.2020]