INVOKEDYNAMIC = bardziej dynamiczna JVM

JDD 2012

Waldek Kot
waldek.kot@gmail.com

Agenda

1. Motywacja – dlaczego taki temat ?
2. Trochę teorii o InvokeDynamic
3. Praktyka
   – zaczynając od „Hello World” ☺

Motywacja

Motywacja pozytywna

InvokeDynamic to <u>największa</u> nowość w Java 7

Motywacja negatywna

- bardzo mało informacji o InvokeDynamic
- mała liczba i słabe tutoriale
- fatalny marketing Oracle, wpychający InvokeDynamic w niszę: *„tylko dla ludzi implementujących języki dynamiczne na JVM”*

Dlaczego warto poznać InDy ?

- must-have dla każdego java-gika
- nowy, fundamentalny składnik JVM
  - nowy bytecode
  - nowe API w *java.lang*
- Java 8: łatwiej będzie zrozumieć implementację lambda expressions (czyli upraszczając: „Java closures")
- wpływ na inne-niż-Java języki programowania dla JVM
  - OK, trudno, ale muszę wspomnieć o wykorzystaniu invokedynamic przez języki dynamiczne na JVM (np. Groovie, Scala, JRuby, Jython, JavaScript)

Po co InvokeDynamic ?

Żeby łatwiej tworzyć dynamiczny, generyczny kod, w miarę wydajnie wykonywany przez JVM.

- z tego samego powodu mamy w Java: refleksję, adnotacje, dynamiczne ładowanie kodu czy generyki

Po co InvokeDynamic ?

- … ale: wszystko co daje InDy można dzisiaj zrobić innymi metodami („zasymulować”)
- prawda, tyle tylko, że dzięki InDy:
  - będzie **prościej**, bo… nie musimy tego robić
    - „najlepszy kod to taki którego nie trzeba pisać”
  - będzie **wydajniej**, bo:
    - kod z InDy jest lepiej „rozumiany” przez kompilator JIT JVM

    => JIT będzie w stanie wykonać o wiele więcej optymalizacji, niż przy zastosowaniu „symulatorów” (dodatkowych warstw)

Krótka teoria InvokeDynamic

InvokeDynamic

Pozwala programiście mieć kontrolę nad czymś tak elementarnym, jak **wywoływanie metod**

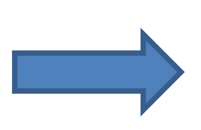
MethodHandle

wskaźnik / uchwyt / referencja do metody

- „metoda” oznacza tutaj także operację dostępu do pól (obiektów i klas), elementów tablic, konstruktorów, *super*, itd.
  - a nawet dostęp do… metod, które nie istnieją ☺.

MethodHandle – Hello World

```java
package pl.confitura.invokedynamic;

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;

public class HelloIndyWorld {
        public static void main(String[] args) throws Throwable {
                MethodHandle mh = MethodHandles.constant(String.class, "Hello World !");
                System.out.println( mh.invoke() );
        }
}
```

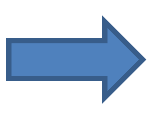


Problems @ Javadoc Declaration Console

```
<terminated> HelloIndyWorld [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (02-07-2012 15:37:56)
Hello World !
```

MethodHandle – coś ciekawszego: uchwyt do własnej metody

```
public class HelloWorld {
        public static class MyClass {
                static public String myConcat(String s1, String s2) {
                        return s1 + s2;
                }
        }

        public static void main(String[] args) throws Throwable {
                MethodType mt = MethodType.methodType(String.class, String.class, String.class);
                MethodHandle mh = MethodHandles.lookup().findStatic(MyClass.class, "myConcat", mt);
                System.out.println( (String) mh.invokeExact("Ala ", "ma kota") );
        }
}
```



```
Console
<terminated> HelloIndyWorld [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (02-07-2012 23:36:13)
Ala ma kota
```

MethodHandle

- jest bardzo lekką konstrukcją (value/wartość)
  - zrozumiałą dla JIT (a także GC) w JVM
- niemal natychmiastowe wykorzystanie MethodHandle to zastąpienie nimi refleksji
  - refleksja (*java.lang.reflect*), mimo iż od Java 1.4 poważnie udoskonalona, to jest znacznie wolniejsza od MethodHandle.

MethodHandle - kombinatory

- API do „manipulacji” uchwytami
  - dostęp do istniejących metod (statycznych, wirtualnych, interfejsów, konstruktorów, …)
  - manipulowanie ich typami, parametrami, zwracanymi wynikami, …
  - jest nawet konstrukcja typu „if-then-else”
- zwykle wynikiem tych manipulacji są nowe uchwyty do metod
- możliwe jest „składanie” manipulacji (jak funkcji: f (g (h (x)))
  - ciąg (łańcuch) manipulacji na uchwytach tworzy graf
  - ten graf (=intencja programisty) jest zrozumiała dla kompilatora JIT maszyny wirtualnej Java

MethodType – typ metody

- określa dokładny typ metod i uchwytów do metod
- mimo dynamizmu InvokeDynamic, JVM jest wciąż silnie typizowalna (strong typing)

- wskazówka: mimo, iż to mało atrakcyjne zagadnienie, to warto je dobrze poznać, bo wiele błędów w zabawach z InDy bierze się z niezgodności typów, a:
  1. w zdecydowanej większości dają one o sobie znać dopiero w czasie wykonania
  2. JVM jest absolutnie bezwzględny jeśli chodzi o zgodność typów !
     - bezpieczeństwo i szybkość

Nowy bytecode: INVOKEDYNAMIC

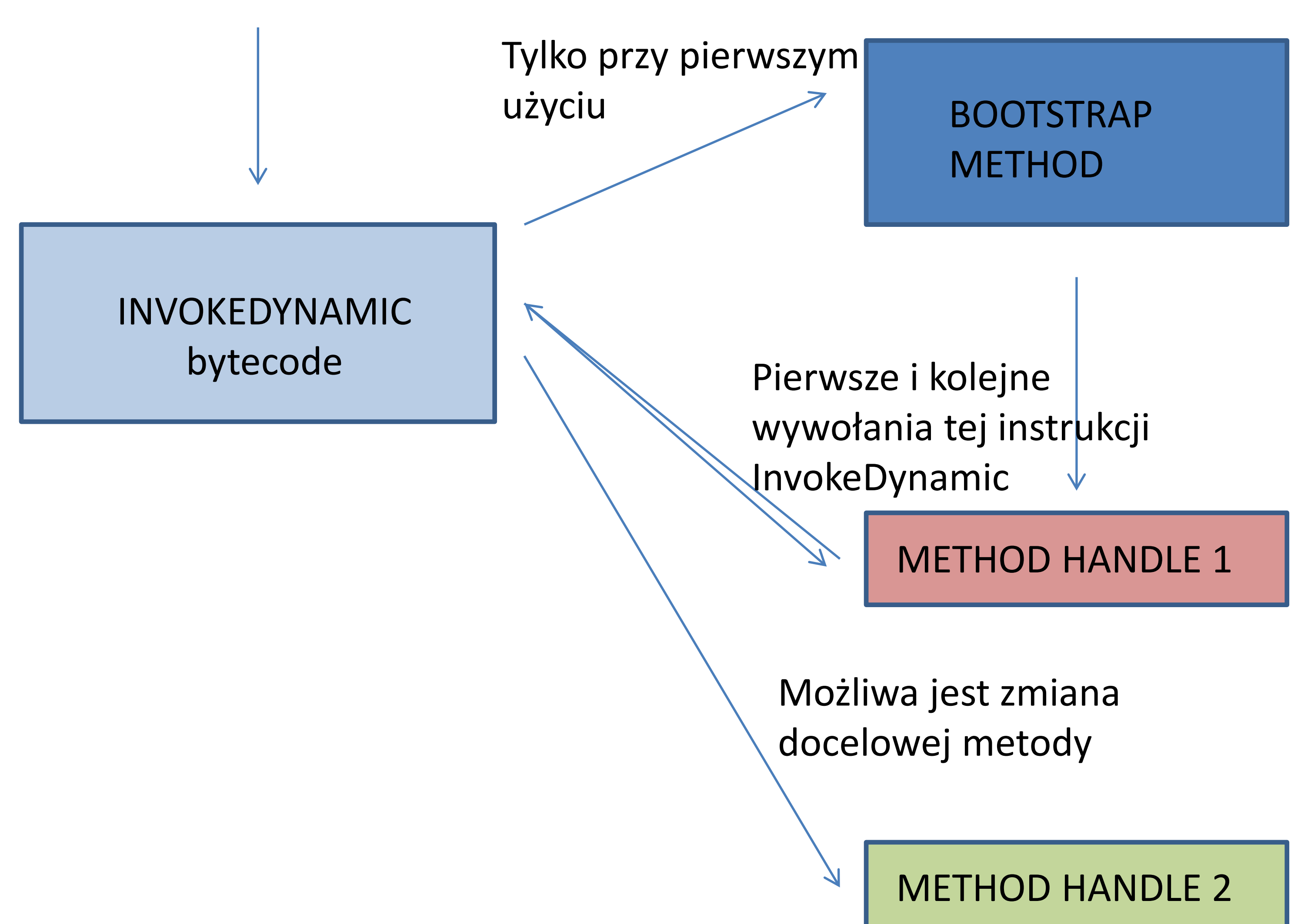
Pierwsze w historii rozszerzenie listy instrukcji JVM !

- choć można spekulować, że o czymś takim myślano już w momencie powstawania technologii Java, bo:
  - kod tej instrukcji był od początku zarezerwowany (BA)
  - nieprzypadkowo (?) jest ulokowany razem z pozostałymi instrukcjami wywołania metod (INVOKESTATIC, INVOKEVIRTUAL, INVOKESPECIAL, INVOKEINTERFACE)
  - tak naprawdę, to historycznie HotSpot VM powstał nie dla języka Java, a dla języków… Smalltalk oraz Scheme ☺, w których jest znacznie większa niż w języku Java możliwość wpływu programisty na sposób wywoływania metod

Bytecode: INVOKEDYNAMIC

- przy <u>pierwszym</u> napotkaniu tej instrukcji, JVM wywołuje metodę (tzw. BSM – bootstrap method)
  - BSM ma określić <u>docelową</u> metodę
    - uchwyt (graf)

- Argumenty INVOKEDYNAMIC:
  - BSM (nazwa, klasa, typ, opcjonalnie argumenty)
  - (opcjonalnie) nazwa wywoływanej metody, jej typ i argumenty
    - te argumenty będą przekazane do docelowej metody


Tylko przy pierwszym użyciu
BOOTSTRAP METHOD
INVOKEDYNAMIC bytecode
Pierwsze i kolejne wywołania tej instrukcji InvokeDynamic
METHOD HANDLE 1
Możliwa jest zmiana docelowej metody
METHOD HANDLE 2

CallSite

- wynikiem wykonania się BSM jest utworzenie obiektu klasy *java.lang.invoke.CallSite*
  – wewnątrz tego obiektu jest *docelowa metoda (uchwyt),* którą wykona instrukcja INVOKEDYNAMIC, po powrocie z BSM
- „CallSite” czyli: „miejsce wywołania metody”
  – dostajemy referencję do „miejsca” w którym umieszczona jest instrukcja INVOKEDYNAMIC
    - czyli potencjalnie możemy zmodyfikować to „miejsce”, tj. podstawić tam inny uchwyt
      – dynamizm InDy w akcji !

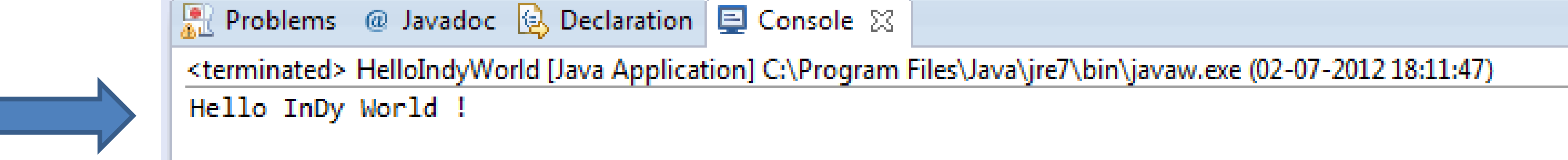
CallSite

- API udostępnia 3 specjalizowane klasy CallSite:
  - *ConstantCallSite* – czyli informujemy JVM, że po wykonaniu BSM, docelowa metoda (target), nie ulegnie już zmianie
  - *MutableCallSite* (oraz jej wariant *VolatileCallSite*), które informują JIT, że docelowa metoda (target) może ulec zmianie
- można tworzyć swoje własne specjalizacje klas *CallSite* !
  - z własną logiką, stanem, itp.

DEMO I
czyli zabawy z MethodHandle

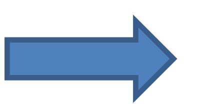
MethodHandle – identity

```java
public class HelloIndyWorld {
	public static void main(String[] args) throws Throwable {
		MethodHandle mh = MethodHandles.identity(String.class);
		System.out.println(mh.invoke("Hello InDy World !"));
	}
}
```



MethodHandle – type

```
public class HelloIndyWorld {
		public static void main(String[] args) throws Throwable {
				MethodHandle mh = MethodHandles.identity(String.class);
				System.out.println(mh.type());
		}
}
```

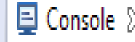


Problems | @ Javadoc | Declaration | Console

```
<terminated> HelloIndyWorld [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (02-07-2012 18:27:18)
(String)String
```

MethodHandle – invoke, invokeExact

```
public class HelloIndyWorld {
        public static void main(String[] args) throws Throwable {
                MethodHandle mh = MethodHandles.identity(String.class);
                //System.out.println(mh.invoke("Hello InDy World !"));
                System.out.println(mh.invokeExact("Hello InDy World !"));
        }
}
```



```
<terminated> HelloIndyWorld [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (02-07-2012 18:31:56)
Exception in thread "main" java.lang.invoke.WrongMethodTypeException: (Ljava/lang/String;)Ljava/lang/String; cannot be called as (Ljava/lang/String;)Ljava/lang/Object;
        at pl.confitura.invokedynamic.HelloIndyWorld.main(HelloIndyWorld.java:19)
```

MethodHandle – uchwyt do metody wirtualnej

```
public class HelloIndyWorld {
        public static class MyClass {
                private String s;

                public MyClass(String s) {
                        this.s = s;
                }

                public int howManyCharacters(String s) {
                        return (s + this.s).length();
                }
        }

        public static void main(String[] args) throws Throwable {
                MethodType mt = MethodType.methodType(int.class, String.class);
                MethodHandle mh = MethodHandles.lookup().findVirtual(MyClass.class, "howManyCharacters", mt);
                MyClass obj = new MyClass("Ala");
                MethodHandle mhBound = mh.bindTo(obj);
                System.out.println( (int) mhBound.invokeExact("ma kota") );
        }
}
```

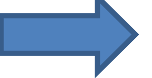
Console

```
10
```

MethodHandle – uchwyt do metody wirtualnej (inaczej)

```
public class HelloIndyWorld {
	public static class MyClass {
		private String s;

		public MyClass(String s) {
			this.s = s;
		}

		public int howManyCharacters(String s) {
			return (s + this.s).length();
		}
	}

	public static void main(String[] args) throws Throwable {
		MethodType mt = MethodType.methodType(int.class, String.class);
		MethodHandle mh = MethodHandles.lookup().findVirtual(MyClass.class, "howManyCharacters", mt);
		System.out.println((int) mh.invokeExact(new MyClass("Ala"), "ma kota"));
	}
}
```

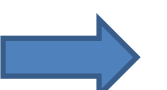
Console



```
10
```

MethodHandle – składanie kombinatorów (funkcji)

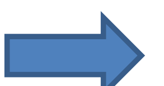
```
public class HelloIndyWorld {
	public static class MyClass {
		private String s;

		public MyClass(String s) {
			this.s = s;
		}

		public String myVirtualConcat(String s) {
			return this.s + s;
		}
	}

	public static void main(String[] args) throws Throwable {
		MethodType mt = MethodType.methodType(String.class, String.class);
		MethodHandle mh = MethodHandles.lookup().findVirtual(MyClass.class, "myVirtualConcat", mt);
		mh = mh.bindTo(new MyClass("Ala "));

		MethodType mtToUpper = MethodType.methodType(String.class);
		MethodHandle mhToUpper = MethodHandles.lookup().findVirtual(String.class, "toUpperCase", mtToUpper);

		MethodHandle mhCombined = MethodHandles.filterArguments(mh, 0, mhToUpper);
		System.out.println((String) mhCombined.invokeExact("ma kota"));
	}
}
```



```
Console
<terminated> HelloIndyWorld [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (03-07-2012 00:00:25)
Ala MA KOTA
```

MethodHandle – filtrowanie wyników metod

```
public class HelloIndyWorld {
		public static class MyClass {
				private String s;

				public MyClass(String s) {
						this.s = s;
				}

				public String myVirtualConcat(String s) {
						return this.s + s;
				}
		}

		public static void main(String[] args) throws Throwable {
				MethodType mt = MethodType.methodType(String.class, String.class);
				MethodHandle mh = MethodHandles.lookup().findVirtual(MyClass.class, "myVirtualConcat", mt);
				mh = mh.bindTo(new MyClass("Ala "));

				MethodType mtToUpper = MethodType.methodType(String.class);
				MethodHandle mhToUpper = MethodHandles.lookup().findVirtual(String.class, "toUpperCase", mtToUpper);

				MethodHandle mhCombined = MethodHandles.filterReturnValue(mh, mhToUpper);
				System.out.println((String) mhCombined.invokeExact("ma kota"));
		}
}
```



<terminated> HelloIndyWorld [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (03-07-2012 0

ALA MA KOTA

MethodHandle – interceptor typu ‚before’

```
public class HelloIndyWorld {
    public static class MyClass {
        private String s;
        public MyClass(String s) {
            this.s = s;
        }

        public String myVirtualConcat(String s) {
            return this.s + s;
        }

        public static String myInterceptor(String s) {
            System.out.println("Intercepted, with arg s: " + s);
            return "^" + s + "^";
```

Console

```
<terminated> HelloIndyWorld [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (03-07-2012 00:18:26)
Exception in thread "main" java.lang.IllegalArgumentException: target and combiner types must match: (String)String != (String)String
	at java.lang.invoke.MethodHandleStatics.newIllegalArgumentException(Unknown Source)
	at java.lang.invoke.MethodHandles.misMatchedTypes(Unknown Source)
	at java.lang.invoke.MethodHandles.foldArguments(Unknown Source)
	at pl.confitura.invokedynamic.HelloIndyWorld.main(HelloIndyWorld.java:33)
```

```
        MethodType mtInterceptor = MethodType.methodType(String.class, String.class);
        MethodHandle mhInterceptor = MethodHandles.lookup().findStatic(MyClass.class, "myInterceptor", mtInterceptor);

        MethodHandle mhCombined = MethodHandles.foldArguments(mh, mhInterceptor);
        System.out.println((String) mhCombined.invokeExact("ma kota"));
    }
}
```

MethodHandle – interceptor typu ‚before’ (poprawniej)

```
public class HelloIndyWorld {
	public static class MyClass {
		private String s;
		public MyClass(String s) {
			this.s = s;
		}

		public String myVirtualConcat(String s) {
			return this.s + s;
		}

		public static String myInterceptor(String s) {
			System.out.println("Intercepted, with arg s: " + s);
			return "^" + s + "^";
		}
	}

	publi
```

Console

```
<terminated> HelloIndyWorld [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (03-07-2012 00:32:22)
Intercepted, with arg s: ma kota
Ala ^ma kota^
```

```
		MethodType mtInterceptor = MethodType.methodType(String.class, String.class);
		MethodHandle mhInterceptor = MethodHandles.lookup().findStatic(MyClass.class, "myInterceptor", mtInterceptor);

		MethodHandle mhCombined = MethodHandles.foldArguments(MethodHandles.dropArguments(mh, 1, String.class),
							mhInterceptor);
		System.out.println((String) mhCombined.invokeExact("ma kota"));
	}
}
```

MethodHandle – interceptor typu ‚before’ (najpoprawniej)

```
public class HelloIndyWorld {
	public static class MyClass {
		private String s;
		public MyClass(String s) {
				this.s = s;
		}

		public String myVirtualConcat(String s) {
				return this.s + s;
		}

		public static String myInterceptor(String s) {
				System.out.println("Intercepted, with arg s: " + s);
				return "^" + s + "^";
		}
	}

	public static void main(String[] args) throws Throwable {
		MethodType mt = MethodType.methodType(String.class, String.class);
		MethodHandle mh = MethodHandles.lookup().findVirtual(MyClass.class, "myVirtualConcat", mt);
		mh = mh.bindTo(new MyClass("Ala "));

		MethodType mtInterceptor = MethodType.methodType(String.class, String.class);
		MethodHandle mhInterceptor = MethodHandles.lookup().findStatic(MyClass.class, "myInterceptor", mtInterceptor);

		MethodHandle mhCombined = MethodHandles.foldArguments(MethodHandles.dropArguments(mh, 1,
							mh.type().parameterType(0)), mhInterceptor);

		System.out.println((String) mhCombined.invokeExact("ma kota"));
	}
}
```

MethodHandles / MethodHandle

Dostępne są także metody:

- wstawiające argumenty wybranego typu (*insertArgument*)
- zmieniające kolejność argumentów (*permuteArguments*)
- tworzące uchwyty przechwytujące wyjątki (*catchException*)
- tworzące uchwyty rzucające wyjątki (*throwException*)
- konwertujące argumenty do podanych typów (*MethodHandle.asType*)
- dopasowujące uchwyt, tak aby argumenty były przekazywane w postaci tablicy (*MethodHandle.asCollector*) lub odwrotnie (*MethodHandle.asSpreader*)
- obsługujące uchwyty o zmiennej liczbie parametrów (*MethodHandle.asVarargsCollector*)
- …

MethodHandles

- jest nawet dostępna konstrukcja IF-THEN-ELSE !
  - *MethodHandles.guardWithTest(test, ifTrue, ifFalse)*

MethodHandle – przykład z *.guardWithTest*

```
public class HelloIndyWorld {
		public static class MyClass {
				public static String withDog(String s) {
						return s + " ma psa";
				}

				public static String withCat(String s) {
						return s + " ma kota";
				}
		}
```

Console

<terminated> HelloIndyWorld [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (03-07-2012 01:20:22)

```
Ala ma psa
Ala ma kota
```

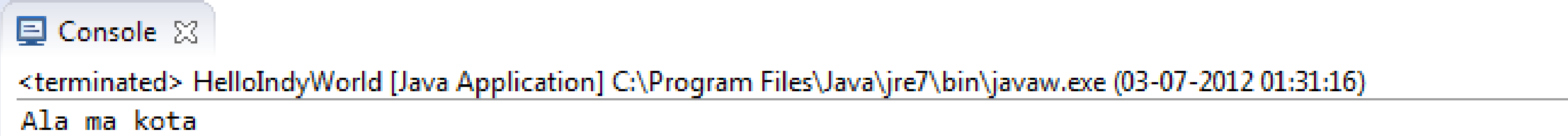
```
				mhTest = MethodHandles.dropArguments(mhTest, 1, mhDog.type().parameterType(0));

				mhDog = MethodHandles.dropArguments(mhDog, 0, boolean.class);
				mhCat = MethodHandles.dropArguments(mhCat, 0, boolean.class);

				MethodHandle mhCombined = MethodHandles.guardWithTest(mhTest, mhDog, mhCat);

				System.out.println((String) mhCombined.invokeExact(true, "Ala"));
				System.out.println((String) mhCombined.invokeExact(false, "Ala"));
		}
}
```

MethodHandleProxies

- możliwe jest tworzenie obiektów, implementujących podany interfejs (typu SAM, czyli z jedną metodą, tzw. interfejs funkcyjny)
- implementacją tej metody będzie metoda wskazywana przez podany uchwyt

MethodHandleProxies - przykład

```
public class HelloIndyWorld {
        public interface MyInterface {
                String myMethod(String s1, String s2);
        }

        public static class MyClass {
                public static String myConcat(String s1, String s2) {
                        return s1 + s2;
                }
        }

        public static void main(String[] args) throws Throwable {
                MethodType mt = MethodType.methodType(String.class, String.class, String.class);
                MethodHandle mh = MethodHandles.lookup().findStatic(MyClass.class, "myConcat", mt);

                MyInterface myObj = MethodHandleProxies.asInterfaceInstance(MyInterface.class, mh);

                System.out.println( myObj.myMethod("Ala ", "ma kota") );
        }
}
```



DEMO II
czyli wywołanie instrukcji bytecode’u
INVOKEDYNAMIC

DEMO III
czyli krótki benchmark wywołań metod: InvokeDynamic vs. Static vs. Reflection

DEMO IV
czyli przykład wykorzystania jednej z cech BSM:
„Laziness”

Inne przydatne konstrukcje

API InvokeDynamic udostępnia jeszcze kilka innych przydatnych konstrukcji pomocniczych (prostszych i/lub bardziej wydajnych niż gdyby samodzielnie budować ich odpowiedniki):

- *java.lang.ClassValue<T>* – pozwala programiście przypisać do obiektów Class swoje własne wartości. Najczęściej używane jako bufor, podobny do ThreadLocal, z tym, że zamiast z wątkiem, wartości są „związane” z klasą
- *SwitchPoint* – rodzaj semafora, który bezpiecznie wątkowo może poinformować o pewnej zmianie związane z nim uchwyty
- *MethodHandleProxies* – pozwala utworzyć obiekt implementujący określony interfejs z 1 metodą (czyli tzw. SAM – Single Abstract Method); „implementacją” ten metody będzie podany uchwyt

DEMO V
czyli przyspiesz swój kod rekurencyjny

(a tak naprawdę, to: jak czerpać intelektualną przyjemność z czytania kodu zawierającego InvokeDynamic)
(zwłaszcza, gdy tego kodu nie musisz napisać ☺)

Podsumowanie

- InvokeDynamic pozwala nam mieć kontrolę nad wywoływaniem metod
  - „żeby łatwiej i wydajniej pisać dynamiczny kod"
  - a zatem nie tylko nisza: „języki dynamiczne na JVM" !
- API dostarcza sporo ciekawej maszynerii
  - uchwyty do metod (MethodHandle)
  - kombinatory
  - INVOKEDYNAMIC - instrukcję bytecode'u dynamicznego wywołania metody
  - obiekty callsite (constant, mutable, volatile)
  - ClassValue, SwitchPoint, …
- i w dodatku są one (oraz intencje programisty) lepiej rozumiane przez JIT JVM

  => lepsza wydajność
- sprytne połączenie tych elementów pozwala tworzyć ładny, bardziej generyczny i dynamiczny kod
  - szczególnie w połączeniu z technikami takimi jak AOP (AspectJ 1.7 zaczyna już wspierać InvokeDynamic)

Inne zastosowania InvokeDynamic

- Java 8
  - Implementacja Lambda Expressions
- nowy sposób na wprowadzanie optymalizacji ?
  - wykorzystanie GPU
- dynamiczna rekonfiguracja kodu bez potrzeby restartowania JVM
- własna implementacja ciekawych konstrukcji programistycznych (multi-methods, multiple-inhritance, meta-object protocol, …)
  - dla gików
- Da Sky's the da limit…

Potencjalne usprawnienia w Java 8…

Java 8 ma przynieść *JSR292.next* i mówi się np. o takich rzeczach jak:

- możliwość introspekcji MethodHandle
- anonimowe classloader'y
  - łatwiejsze i wydajniejsze generowanie klas/metod i modyfikacji constant pool
- Fixnums, czy zmniejszona presja na autoboxing, poprzez zawarcie wartości liczby w samej referencji
- Interface injection, czyli dynamiczne (w czasie wykonania) dodawanie metod do klas/instancji
- Coroutines, czyli uogólnienie subroutines, czyli metody z więcej niż 1 punktem wejścia
  - m.in. mielibyśmy prostszy kod współbieżny

Dzięki !

Waldek Kot
waldek.kot@gmail.com

☺