

Edward Lavieri, Peter Verhas

Tajniki Java 9

Pisanie reaktywnego, modularnego,
współbieżnego i bezpiecznego kodu



Packt >

Tajniki Java 9

Pisanie reaktywnego, modularnego, współbieżnego
i bezpiecznego kodu

Dr Edward Lavieri

Peter Verhas

Przekład: Jakub Niedźwiedź

APN Promise
Warszawa 2018

Tajniki Java 9

Original English language edition © 2017 Packt Publishing

All rights reserved. Authorised translation from the English language edition book

Mastering Java 9

ISBN 978-1-78646-873-4, published by Packt Publishing.

© Polish edition by APN PROMISE SA, Warszawa 2018

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa

tel. +48 22 35 51 600, fax +48 22 35 51 699

e-mail: mSPress@promise.pl

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiegokolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

Książka ta przedstawia poglądy i opinie autorów. Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń, chyba że zostanie jednoznacznie stwierdzone, że jest inaczej. Ewentualne podobieństwo do jakiegokolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

Wszystkie nazwy handlowe i towarowe występujące w niniejszej publikacji mogą być znakami towarowymi zastrzeżonymi lub nazwami zastrzeżonymi odpowiednich firm odnośnych właścicieli.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji.

APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiegokolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-340-3

Przekład: Jakub Niedźwiedź

Redakcja: Marek Włodarz

Korekta: Ewa Swędrowska

Skład i łamanie: MAWART Marek Włodarz

Zespół

Autorzy

Dr Edward Lavieri

Peter Verhas

Recenzent

Mandar Jog

Opracowanie indeksu

Francy Puthiry

Grafika

Jason Monteiro

O autorach

Dr Edward Lavieri jest długoletnim programistą z dużym doświadczeniem akademickim. Obronił doktorat z informatyki na Colorado Technical University oraz posiada tytuły magistra z dziedziny zarządzania systemami informatycznymi (Bowie State University), edukacji (Capella University) oraz zarządzania operacyjnego (University of Arkansas).

Zajmuje się tworzeniem i prowadzeniem kursów informatycznych od 2002 roku. Edward wycofał się ze służby w amerykańskiej marynarce wojennej po 25 latach. Jako założyciel i dyrektor kreatywny firmy informatycznej three19, Edward stale zajmuje się projektowaniem i tworzeniem oprogramowania. Wykorzystuje różne narzędzia programistyczne i silniki sterujące grami. Jego pasją jest opracowywanie systemów i gier edukacyjnych oraz aplikacji mobilnych.

Edward jest autorem książek: *Adaptive Learning for Educational Game Design* (CreateSpace), *Getting Started with Unity 5* (Packt), *Learning AWS Lumberyard Game Development* (Packt), *LiveCode Mobile Development HOTSHOT* (Packt), *LiveCode Mobile Development Cookbook* (Packt) oraz *Software Consulting: A Revolutionary Approach* (CreateSpace). Był też redaktorem technicznym książki *Excel Formulas and Functions for Dummies* (Wiley Publishing). Opracował również wiele uniwersyteckich kursów dotyczących informatyki, systemów informatycznych i programowania gier.

Peter Verhas jest starszym programistą i architektem systemów informatycznych z ponad 30-letnim doświadczeniem programistycznym. Obecnie pracuje dla firmy EPAM jako starszy programista, gdzie zajmuje się wieloma projektami i aktywnie uczestniczy w działaniach szkoleniowych firmy. Peter jest autorem bloga i jest oddany programowaniu z otwartym kodem źródłowym. Korzysta z języka Java od 2005 roku i jest też współautorem portalu Java Code Geeks.

O recenzencie

Mandar Jog jest ekspertem w zakresie szkolenia informatycznego z ponad 15-letnim doświadczeniem szkoleniowym. Jest ekspertem od technologii, takich jak Java, J2EE i Android. Posiada też certyfikaty SCJP i SCWCD. Od czasu do czasu pisze też artykuły na blogu, gdzie ułatwia czytelnikom zrozumienie skomplikowanych pojęć dotyczących języka Java i J2EE. Regularnie wykłada na wielu uczelniach technicznych oraz prowadzi wiele seminariów i warsztatów.

Był też technicznym recenzentem innej książki wydanej przez Packt: *Modular Programming in Java 9*.

Podziękowania dla Tejaswini, mojej inspiracji w tej życiowej podróży. Jestem też wdzięczny swojemu synowi (Ojas) – jego uśmiech zawsze dodaje mi otuchy.

Spis treści

O autorach	iii
Spis treści	v
Wprowadzenie	1
Rozdział 1: Krajobraz języka Java 9.....	7
Java 9 z lotu ptaka	8
Burzenie monolitu	9
Wykorzystanie powłoki Java Shell	10
Sterowanie procesami zewnętrznymi.....	11
Podnoszenie wydajności dzięki G1	11
Mierzenie wydajności przy pomocy JMH	11
Wprowadzenie do HTTP 2.0	12
Zastosowanie programowania reaktywnego	12
Poszerzanie listy życzeń.....	13
Podsumowanie	14
Rozdział 2: Odkrywanie Java 9	15
Poprawione sporne blokowanie [JEP 143]	16
Cele poprawy.....	16
Dzielona pamięć podręczna kodu [JEP 197].....	17
Alokacja pamięci.....	18
Kompilacja Smart Java, faza druga [JEP 199].....	18
Obsługa ostrzeżeń z narzędzi Lint i Doclint [JEP 212].....	19
Warstwowe przypisywanie typów w javac [JEP 215]	20
Obsługa adnotacji 2.0 [JEP 217]	21
Nowy schemat łańcucha wersji [JEP 223]	22
Automatyczne generowanie testów kompilatora w czasie wykonywania programu [JEP 233]	23
Testowanie atrybutów plików klas generowanych przez Javac [JEP 235].....	24
Przechowywanie łańcuchów tekstowych w archiwach CDS [JEP 250].....	25
Problem.....	25

Rozwiązanie	26
Przygotowywanie kontrolki interfejsu użytkownika JavaFX oraz interfejsów CSS API do modularyzacji [JEP 253]	26
Przegląd JavaFX.....	27
Implikacje dla wersji Java 9	28
Zwarte łańcuchy tekstowe [JEP 254]	29
Stan sprzed wersji Java 9	30
Nowość w wersji Java 9	30
Włączanie wybranych aktualizacji Xerces 2.11.0 do JAXP [JEP 255]	30
Zaktualizowanie JavaFX/Media do nowszej wersji GStreamer [JEP 257]	31
Silnik HarfBuzz do obsługi czcionek [JEP 258]	32
Grafika HiDPI w systemach Windows i Linux [JEP 263]	33
Renderer graficzny Marlin [JEP 265]	34
Unicode 8.0.0 [JEP 267]	34
Nowości w Unicode 8.0.0	34
Zaktualizowane klasy w wersji Java 9	35
Zarezerwowane obszary stosu dla sekcji krytycznych [JEP 270]	35
Sytuacja sprzed wersji Java 9	35
Nowości w wersji Java 9	36
Dynamiczne łączenie modeli obiektowych zdefiniowanych przez język [JEP 276] ..	37
Dowód poprawności	37
Dodatkowe testy dla olbrzymich obiektów w G1 [JEP 278]	38
Poprawienie rozwiązywania problemów z nieudanymi testami [JEP 279]	39
Informacje środowiskowe	40
Informacje o procesach Java	40
Optymalizowanie łączenia łańcuchów tekstowych [JEP 280]	41
Platforma HotSpot do testów jednostkowych w C++ [JEP 281]	41
Umożliwienie korzystania z GTK 3 w systemach Linux [JEP 283]	42
Nowy system budowania HotSpot [JEP 284]	43
Podsumowanie	44
Rozdział 3: Usprawnienia języka Java 9	45
Praca z dojściami do zmiennych [JEP 193]	46
Praca z narzędziami AtoMiC Toolkit	47
Użycie klasy sun.misc.Unsafe	49
Pomijanie ostrzeżeń o przestarzałych funkcjach w instrukcjach importu [JEP 211] 50	

Rozszerzanie funkcji Project Coin [JEP 213]	51
Korzystanie z adnotacji @SafeVarargs	51
Instrukcja try-with-resource	52
Korzystanie z operatora rombowego	53
Zaprzestanie użycia znaku podkreślenia	54
Użycie prywatnych metod interfejsowych	55
Poprawne przetwarzanie instrukcji import [JEP 216]	58
Podsumowanie	59
Rozdział 4: Budowanie modularnych aplikacji w Java 9	61
Wprowadzenie do modularności	62
Przegląd systemu modułów platformy Java [JEP-200]	64
Profil kompaktowy 1	64
Profil kompaktowy 2	65
Profil kompaktowy 3	65
Modularyzacja kodu źródłowego JDK [JEP-201]	68
Organizacja kodu źródłowego JDK sprzed wersji Java 9	69
Narzędzia programistyczne	70
Wdrażanie	70
Dostosowanie do warunków międzynarodowych	70
Monitorowanie	70
RMI	71
Bezpieczeństwo	71
Rozwiązywanie problemów	71
Usługi WWW	72
Narzędzia JavaFX	72
Środowisko uruchomieniowe Java	72
Kod źródłowy	72
Biblioteki	73
Pliki nagłówkowe C	74
Baza danych	75
Zreorganizowany kod źródłowy JDK	75
Zrozumienie modularnych obrazów uruchomieniowych [JEP-220]	75
Przyjęcie formatu plików uruchomieniowych	76
Zmiana struktury obrazu uruchomieniowego	76

Obsługa typowych operacji.....	78
Usunięcie przywilejów z klas JDK	78
Zachowywanie istniejących zachowań	78
Poznawanie systemu modułów [JEP-261]	78
Ścieżki dla modułów	79
Naruszenia granic kontroli dostępu	79
Środowisko uruchomieniowe	80
Modularne pakowanie aplikacji Java [JEP-275]	82
Zaawansowane możliwości narzędzia Java Linker	82
Opcje narzędzia Java Packager.....	82
JLink – konsolidator Java Linker [JEP-282]	85
Hermetyzacja większości wewnętrznych interfejsów API [JEP-260]	87
Podsumowanie	88
Rozdział 5: Migrowanie aplikacji do Java 9	89
Szybki przegląd Project Jigsaw	90
Classpath.....	90
Monolityczna natura JDK	91
Jak moduły wpasowują się do ogólnych założeń platformy Java	92
Moduł bazowy	93
Niezawodna konfiguracja.....	95
Silna hermetyzacja	96
Planowanie migracji	97
Testowanie prostej aplikacji Java.....	97
Potencjalne problemy z migracją aplikacji	100
JRE	100
Dostęp do wewnętrznych interfejsów API.....	100
Dostęp do wewnętrznych pakietów JAR	101
Dezaktualizacja adresów URL dla pakietów JAR	102
Mechanizm rozszerzeń.....	103
Modularyzacja pakietu JDK	104
Porady od firmy Oracle	106
Kroki przygotowawcze	106
Pozyskanie zbudowanego pakietu JDK 9.....	106
Uruchomienie programu przed rekompilacją.....	106
Aktualizacja bibliotek i narzędzi firm trzecich.....	107

Kompilacja aplikacji	107
Opcje -source i -target przed wersją Java 9	109
Opcje -source i -target w wersji Java 9	110
Uruchomienie narzędzia jdeps dla danego kodu	110
Przerywanie hermetyzacji	113
Opcja --add-opens	114
Opcja --add-exports	114
Opcja --permit-illegal-access	114
Zmiany w obrazie uruchomieniowym	115
Schemat wersji Java	115
Układ JDK i JRE	116
Co zostało usunięte	118
Zaktualizowane odświeżanie pamięci	119
Wdrażanie	120
Wybór wersji JRE	120
Serializowane aplety	120
Aktualizacja JNLP	120
Zagieżdżone zasoby	121
Rozszerzenie FX XML	121
Składnia pliku JNLP	123
Liczbowe porównywanie wersji	123
Przydatne narzędzia	124
Środowisko Java – jEnv	124
Maven	126
Pozyskiwanie M2Eclipse IDE	127
Podsumowanie	129
Rozdział 6: Eksperymentowanie z powłoką Java Shell	131
Czym jest JShell?	132
Początki pracy z JShell	133
Praktyczne zastosowania JShell	138
Tryby sprzężenia zwrotnego	138
Tworzenie niestandardowego trybu sprzężenia zwrotnego	143
Wypisywanie listy aktywów	145
Edytowanie w JShell	146
Modyfikowanie tekstu	146

Podstawowa nawigacja	147
Nawigacja historyczna	147
Zaawansowane polecenia edycyjne	148
Praca ze skryptami	148
Skrypty startowe	148
Ładowanie skryptów	148
Zapisywanie skryptów	149
Zaawansowane skrypty w JShell	149
Podsumowanie	151
Rozdział 7: Wykorzystanie nowego, domyślnego odśmiecania G1	153
Przegląd odśmiecania pamięci	154
Cykl życia obiektu	154
Tworzenie obiektu	154
Dalsze życie obiektu	155
Niszczenie obiektu	155
Algorytmy odśmiecania pamięci	156
Mark and sweep (oznacz i zamieć)	156
Odśmiecanie pamięci CMS	156
Szeregowe odśmiecanie pamięci	157
Równoległe odśmiecanie pamięci	157
Odśmiecanie pamięci G1	157
Opcje odśmiecania pamięci	158
Metody języka Java związane z odśmiecaniem pamięci	162
Metoda System.gc()	162
Metoda finalize()	164
Odśmiecanie pamięci przed wersją Java 9	166
Wizualizowanie odśmiecania pamięci	166
Aktualizacje odśmiecania pamięci w wersji Java 8	167
Studium przypadku – gry pisane w języku Java	168
Odśmiecanie pamięci na nowej platformie Java	169
Domyślne odśmiecanie pamięci	169
Przestarzałe kombinacje odśmiecania pamięci	171
Zunifikowane rejestrowanie działań odśmiecania pamięci	172
Zunifikowane rejestrowanie JVM (JEP-158)	173
Znaczniki	173

Poziomy	174
Dekoracje	174
Informacje wyjściowe	175
Opcje wiersza poleceń	175
Zunifikowane rejestrowanie odśmieciania pamięci (JEP-271)	175
Opcje rejestrowania odśmieciania pamięci	176
Znacznik gc	178
Makra	178
Dodatkowe uwarunkowania	179
Utrzymujące się problemy	179
Sprawianie, aby obiekty podlegały odśmiecaniu pamięci	180
Podsumowanie	182
Rozdział 8: Mikroanalizowanie aplikacji przy pomocy JMH	183
Przegląd mikroanalizowania	184
Podejście do korzystania z JMH	185
Instalowanie Java 9 i Eclipse ze wsparciem dla Java 9	185
Praktyczny eksperyment	187
Mikroanalizowanie przy pomocy Maven	189
Opcje analizowania	194
Tryby	194
Jednostki czasu	195
Techniki pomagające uniknąć pułapek mikroanalizowania	195
Zarządzanie energią	196
Systemowe programy planujące	196
Współdzielenie czasu	196
Eliminowanie ślepych uliczek w kodzie i zawijania stałych	196
Rozbieżność pomiędzy uruchomieniami	197
Pojemność pamięci podręcznej	198
Podsumowanie	198
Rozdział 9: Wykorzystanie interfejsu API ProcessHandle	199
Czym są procesy?	200
Nowy interfejs ProcessHandle	201
Uzyskiwanie identyfikatora PID bieżącego procesu	202
Uzyskiwanie informacji na temat procesu	202

Wypisywanie listy procesów	204
Wypisywanie listy procesów podrzędnych	204
Wypisywanie listy procesów potomnych	205
Wypisywanie listy wszystkich procesów	206
Oczekiwanie na procesy	207
Kończenie procesów	208
Niewielka aplikacja sterująca procesami	211
Klasa Main	211
Klasa Parameters	213
Klasa ParamsAndHandle	214
Klasa ControlDaemon	214
Podsumowanie	218
Rozdział 10: Dokładne śledzenie stosu	219
Przegląd stosu Java	219
Znaczenie informacji o stosie	220
Przykład – ograniczanie elementów wywołujących	222
Przykład – uzyskiwanie obiektu rejestrującego dla elementu wywołującego	225
Praca z klasą StackWalker	226
Pozyskiwanie wystąpienia klasy StackWalker	226
RETAIN_CLASS_REFERENCE	227
SHOW_REFLECT_FRAMES	227
SHOW_HIDDEN_FRAMES	227
Końcowe uwagi na temat stałych wyliczenia	230
Dostęp do klas	230
Metody do przechodzenia przez stos	231
StackFrame	233
Wydajność	234
Podsumowanie	234
Rozdział 11: Nowe narzędzia i usprawnienia narzędzi	235
Nowy klient HTTP [JEP-110]	236
Klient HTTP przed wersją Java 9	236
Nowy klient HTTP w wersji Java 9	239
Ograniczenia nowego interfejsu API	240

Uproszczony interfejs API Doclet [JEP-221]	242
Interfejs API Doclet przed wersją Java 9	242
Wyliczenia interfejsu API	244
Klasy interfejsu API	244
Interfejsy API	245
Problemy z istniejącym wcześniej interfejsem API Doclet	246
Interfejs API Doclet w wersji Java 9	246
Interfejs API drzewa kompilatora	246
Interfejs API modelu języka	248
Interfejs AnnotatedConstruct	248
Wyliczenie SourceVersion	253
Wyjątek UnknownEntityException	254
Javadoc z HTML5 [JEP-224]	255
Wyszukiwanie w Javadoc [JEP-225]	260
Wprowadzenie wyszukiwania z użyciem pierwszych liter słów	261
Usunięcie wyboru wersji JRE [JEP-231]	261
Interfejs API Parser dla silnika Nashorn [JEP-236]	262
Nashorn	262
Korzystanie z Nashorn jako narzędzia wiersza poleceń	263
Korzystanie z Nashorn jako osadzonego interpretera	266
ECMAScript	267
Interfejs API Parser	267
Pliki JAR dla wielu wersji [JEP-238]	268
Identyfikowanie plików JAR dla wielu wersji	270
Powiązane zmiany w pakiecie JDK	271
Interfejs kompilatora JVM na poziomie Java [JEP-243]	272
Adnotacje BeanInfo [JEP-256]	273
JavaBean	273
BeanProperty	274
SwingContainer	274
Klasy BeanInfo	275
Wejście/wyjście dla obrazów TIFF [JEP-262]	275
Interfejs API i usługa rejestrowania platformowego [JEP-264]	278
Pakiet java.util.logging	278
Rejestrowanie w wersji Java 9	280

Katalogi XML [JEP-268]	281
Standard OASIS XML Catalog	281
Procesory JAXP	282
Katalogi XML przed wersją Java 9	282
Zmiany na platformie Java 9	282
Wygodne metody fabryczne dla kolekcji [JEP-269]	283
Wykorzystanie kolekcji przed wersją Java 9	284
Wykorzystanie nowych literałów dla kolekcji	286
Funkcje biurkowe specyficzne dla platformy [JEP-272]	287
Ulepszona obsługa metod [JEP-274]	288
Powody dla wprowadzenia poprawek	288
Funkcje wyszukiwawcze	288
Obsługa argumentów	289
Dodatkowe kombinacje	289
Ulepszone wycofywanie przestarzałych elementów [JEP-277]	290
Co tak naprawdę oznacza adnotacja @Deprecated	291
Podsumowanie	291
Rozdział 12: Współbieżność i programowanie reaktywne	293
Programowanie reaktywne	294
Standaryzacja programowania reaktywnego	295
Nowy interfejs API Flow	297
Interfejs Flow.Publisher	297
Interfejs Flow.Subscriber	298
Interfejs Flow.Subscription	298
Interfejs Flow.Processor	298
Przykładowa implementacja	299
Dodatkowe aktualizacje współbieżności	300
Współbieżność na platformie Java	300
Wyjaśnienie pojęcia współbieżności	301
Konfiguracje systemowe	301
Wątki na platformie Java	302
Ulepszenia współbieżności	305
Usprawnienia interfejsu API CompletableFuture	307
Szczegóły klasy	307

Ulepszenia	311
Wskazówki dla cykli spin-wait	312
Podsumowanie	312
Rozdział 13: Usprawnienia zabezpieczeń	313
Protokół Datagram Transport Layer Security	314
Protokół DTLS wersja 1.0	314
Protokół DTLS wersja 1.2	316
Wsparcie dla DTLS w wersji Java 9	319
Tworzenie magazynów kluczy PKCS12	320
Wprowadzenie do magazynów kluczy	320
Java Keystore (JKS)	320
Builder	321
Klasa CallbackHandlerProtection	322
Klasa PasswordProtection	322
Klasa PrivateKeyEntry	323
Klasa SecretKeyEntry	323
Klasa TrustedCertificateEntry	323
Domyślne ustawienie PKCS12 w Java 9	325
Poprawienie wydajności zabezpieczeń aplikacji	325
Wymuszanie zasad zabezpieczeń	325
Obliczanie uprawnień	326
Pakiet java.Security.CodeSource	327
Algorytm sprawdzania pakietów	327
Rozszerzenie negocjacyjne dla protokołu TLS w warstwie aplikacji	328
Rozszerzenie ALPN dla TLS	328
Pakiet javax.net.ssl	329
Rozszerzenie pakietu java.net.ssl	330
Wykorzystanie instrukcji procesora w algorytmach GHASH i RSA	332
Funkcje skrótu	332
Podpinanie OCSP dla TLS	334
Podstawy podpinania OCSP	334
Zmiany dla platformy Java 9	335
Implementacje SecureRandom oparte na DRBG	337
Podsumowanie	337

Rozdział 14: Flagi wiersza poleceń	339
Zunifikowane rejestrowanie JVM [JEP 158]	340
Opcje wiersza poleceń	341
Dekoracje	343
Poziomy	344
Informacje wyjściowe	344
Znaczniki	344
Sterowanie kompilatorem [JEP 165]	345
Tryby kompilacji	345
Tryb kompilacji C1	346
Tryb kompilacji C2	346
Kompilacja warstwowa	346
Sterowanie kompilatorem w wersji Java 9	347
Polecenia diagnostyczne [JEP 228]	349
Agent profilowania sterty [JEP 240]	350
Usunięcie narzędzia JHAT [JEP 241]	351
Sprawdzanie poprawności argumentów flag wiersza poleceń JVM [JEP 245]	352
Kompilacja dla starszych wersji platformy [JEP 247]	353
Podsumowanie	354
Rozdział 15: Najlepsze praktyki w Java 9	357
Wsparcie dla UTF-8	357
Klasa ResourceBundle	358
Zagnieżdżona klasa	359
Pola i konstruktory	363
Metody	363
Zmiany w wersji Java 9	369
Unicode 7.0.0	369
Pakiet java.lang	370
Pakiet java.text	370
Dodatkowe znaczenie	371
Przeniesienie pakietu JDK na platformę Linux/AArch64	372
Obrazy z wieloma rozdzielczościami	373
Repozytorium Common Locale Data Repository (CLDR)	374
Podsumowanie	375

Rozdział 16: Przyszłe kierunki rozwoju	377
Przyszłe zmiany w pakiecie JDK	378
Zmiany w JDK docelowe dla wersji Java 10	378
Konsolidacja repozytoriów	378
Usunięcie narzędzia do generowania rodzimych nagłówków	379
Zgłoszone propozycje związane z JDK	380
Zrównoleglenie fazy pełnego odświeżania pamięci w CMS	380
Interfejsy REST API dla JMX	381
Wsparcie dla alokacji serty	382
Szkieletowe propozycje związane z JDK	382
Przyspieszenie finalizowania obiektów	382
Model pamięci Java	383
Interfejsy obcych funkcji	384
Metody izolowane	384
Ograniczenie marnowania metaprzestrzeni	384
Poprawienie obsługi IPv6	385
Rozpakowane listy argumentów dla dojsć do metod	386
Ulepszone demo MandelbrotSet, wykorzystujące typy wartościowe	387
Wydajne działanie porównywania tablic	388
Przyszłe zmiany w kompilatorze Java	388
Zasady wygaszania opcji -source i -target dla javac	388
Statyczne analizatory z obsługą wtyczek	388
Przyszłe zmiany w Java Virtual Machine	389
Zgłoszone propozycje związane z JVM	389
Platforma Java działająca na kontenerach	389
Umożliwienie wykonywania metod języka Java na procesorach GPU	390
Epsilon GC – Niskokosztowe odświeżanie pamięci	391
Szkieletowe propozycje związane z JVM	391
Zapewnienie stabilnych punktów próbnikowych USDT w metodach skompiłowanych dla JVM	392
Zawężanie współbieżnego monitora	392
Zapewnienie niskokosztowego sposobu próbkowania alokacji na sterce Java	393
Platforma poleceń diagnostycznych	394
Ulepszone redefiniowanie klas	394
Domyślne włączanie trybu NUMA, gdy jest to wskazane	394

Obiekty typów wartościowych	395
Zasady sprawdzania dostępu w JVM	396
Przyszłe zmiany w JavaX	396
Adnotacje specyficzne dla JMX do rejestrowania zarządzanych zasobów	396
Modernizacja implementacji wyglądu i zachowania GTK3	397
Trwające projekty specjalne	397
Annotations pipeline 2.0	399
Audio Synthesis Engine	399
Caciocavallo	399
Common VM Interface	399
Compiler Grammar	399
Da Vinci Machine	400
Device I/O	400
Gaal	400
HarfBuzz Integration	400
Kona	401
OpenJFX	401
Panama	401
Shenandoah	401
Podsumowanie	402
Indeks	403

Wprowadzenie

Wersja Java 9 i jej nowe funkcje rozwijają bogactwo tego języka, który jest jednym z najczęściej używanych języków programowania do budowania sprawnych aplikacji. Java 9 kładzie specjalny nacisk na modularność, implementowaną przez Project Jigsaw. Ta książka stanowi przewodnik po zmianach na platformie Java.

Zapewnia ogólny przegląd oraz szczegółowe wyjaśnienia nowych funkcji wprowadzonych w wersji Java 9 oraz podkreśla znaczenie nowych interfejsów API i innych ulepszeń. Niektóre nowe funkcje Java 9 mają przełomowe znaczenie, a doświadczonemu programiście pomogą usprawnić tworzone aplikacje korporacyjne dzięki zaimplementowaniu tych nowych funkcji. Ten podręcznik dostarczy mnóstwo praktycznych wskazówek, pozwalających zastosować nowo zdobytą wiedzę dotyczącą wersji Java 9, a także sporo dodatkowych informacji związanych z przyszłym rozwojem platformy Java. Dzięki tej książce można poprawić swoją wydajność i przyspieszyć działanie swoich aplikacji. Poznając najlepsze praktyki związane z językiem Java, można stać się ekspertem od wersji Java 9 w swojej firmie.

Przeczytanie tej książki pozwoli nie tylko poznać najważniejsze pojęcia języka Java 9, ale również zrozumieć niuanse związane z ważnymi aspektami programowania w tym świetnym języku.

Zawartość książki

Rozdział 1: *Krajobraz Java 9*, zajmuje się najistotniejszymi funkcjami, wprowadzonymi w wersji Java 9, w tym takimi jak Project Jigsaw, powłoka Java Shell, odśmiecanie G1 oraz programowanie reaktywne. Ten rozdział zapewnia wprowadzenie do tych zagadnień, przygotowując czytelnika do ich dokładniejszego omówienia w kolejnych rozdziałach.

Rozdział 2: *Odkrywanie Java 9*, omawia kilka zmian w platformie Java, które obejmują poprawienie efektywności sterczy, alokowanie pamięci, usprawnienia procesu kompilacji, testowanie typów, adnotacje, zautomatyzowane testy kompilatora oraz poprawione odśmiecanie.

Rozdział 3: *Usprawnienia języka Java 9*, skupia się na zmianach w języku Java. Zmiany te dotyczą obsługi zmiennych, ostrzeżeń o wygaśnięciu, usprawnień dotyczących

funkcjonalności Project Coin wprowadzonych w wersji Java 7 oraz przetwarzania instrukcji import.

Rozdział 4: *Budowanie modularnych aplikacji w Java 9*, bada strukturę modułu Java określoną przez Project Jigsaw oraz implementację Jigsaw jako części platformy Java. Rozdział ten zajmuje się też kluczowymi zmianami wewnętrznymi na platformie Java i ich powiązaniem z nowym systemem modularnym.

Rozdział 5: *Migrowanie aplikacji do Java 9*, bada sposoby migracji aplikacji Java 8 do platformy Java 9. Omawiane są zarówno ręczne, jak i półautomatyczne procesy migracji.

Rozdział 6: *Eksperymentowanie z powłoką Java Shell*, omawia JShell, nowe narzędzie wiersza poleceń w Java 9. Zakres tego rozdziału obejmuje informacje związane z tym narzędziem, pojęcie pętli read-eval-print oraz polecenia i opcje wiersza poleceń wykorzystywane w JShell.

Rozdział 7: *Wykorzystanie nowego, domyślnego odświeżania G1*, przygląda się dokładnie procesowi odświeżania pamięci i jego obsłudze w Java 9.

Rozdział 8: *Mikroanalizowanie aplikacji przy pomocy JMH*, zajmuje się sposobami pisania testów wydajnościowych przy użyciu biblioteki Java Microbenchmark Harness (JMH) służącej do pisania testów sprawnościowych dla Java Virtual Machine (JVM). Wraz z JMH wykorzystywane jest narzędzie Maven, co pomaga zilustrować moc mikroanalizowania na nowej platformie Java 9.

Rozdział 9: *Wykorzystanie interfejsu API ProcessHandle*, dokonuje przeglądu nowych interfejsów API, które umożliwiają zarządzanie procesami systemu operacyjnego.

Rozdział 10: *Dokładne śledzenie stosu*, obejmuje nowy interfejs API, który pozwala na skuteczne sprawdzanie stosu. Rozdział ten zawiera szczegółowe informacje dotyczące sposobów uzyskiwania informacji o śladzie stosu.

Rozdział 11: *Nowe narzędzia i usprawnienia narzędzi*, obejmuje 16 propozycji (JEP) usprawnień narzędzi Java, które zostały wprowadzone na platformie Java 9. Obejmują one szeroki zakres narzędzi i aktualizacji interfejsów API ułatwiających programowanie w języku Java i dających większe możliwości optymalizacji naszych aplikacji Java.

Rozdział 12: *Współbieżność i programowanie reaktywne*, omawia usprawnienia dotyczące współbieżności, wprowadzone na platformie Java 9. Rozdział ten skupia się głównie na obsłudze programowania reaktywnego, które jest zapewniane przez interfejs API klasy Flow. Omawiane są też dodatkowe usprawnienia współbieżności wprowadzone w Java 9.

Rozdział 13: *Usprawnienia zabezpieczeń*, obejmuje kilka niewielkich zmian w JDK, które dotyczą zabezpieczeń. Usprawnienia zabezpieczeń, wprowadzone na platformie Java 9, zapewniają programistom większe możliwości pisania i utrzymywania aplikacji, które są bardziej bezpieczne, niż to było wcześniej możliwe.

Rozdział 14: *Flagi wiersza poleceń*, zajmuje się zmianami flag wiersza poleceń w wersji Java 9. Wśród pojęć omawianych w tym rozdziale są ujednoczone logowanie JVM, sterowanie kompilatorem, polecenia diagnostyczne, agent profilowania sterty, JHAT, sprawdzanie poprawności argumentów flag wiersza poleceń oraz kompilowanie dla starszych wersji platformy.

Rozdział 15: *Najlepsze praktyki w Java 9*, skupia się na pracy z narzędziami zapewnianymi przez platformę Java 9 do wspierania plików UTF-8, Unicode 7.0.0, Linux/AArch64, obrazów o wielu rozdzielczościach i wspólnych repozytoriów danych.

Rozdział 16: *Przyszłe kierunki rozwoju*, zapewnia przegląd przyszłego rozwoju platformy Java. Obejmuje to przyjrzenie się planom związanym z wersją Java 10 oraz dalszymi zmianami, które prawdopodobnie zobaczymy w przyszłości.

Co jest potrzebne do korzystania z tej książki

Do pracy z tym tekstem potrzebna jest co najmniej podstawowa wiedza na temat języka programowania Java. Potrzebne też będą następujące składniki programowe:

- Java SE Development Kit 9 (JDK)
<http://www.oracle.com/technetwork/java/javase/downloads/>
- Zintegrowane środowisko programistyczne (IDE) do kodowania. Oto propozycje:
 - Eclipse
<https://www.eclipse.org>
 - IntelliJ
<https://www.jetbrains.com/idea/>
 - NetBeans
<https://netbeans.org>

Dla kogo jest ta książka

Ta książka jest dla programistów korporacyjnych i obecnych programistów wykorzystujących język Java. Wymagana jest podstawowa wiedza dotycząca języka Java.

Konwencje

W tej książce znajdziemy wiele stylów tekstu, którymi oznaczane są różne rodzaje informacji. Oto kilka przykładów tych stylów oraz wyjaśnienie ich znaczenia.

Elementy kodu w tekście, nazwy tabel bazodanowych, nazwy folderów, nazwy plików, rozszerzenia plików, nazwy ścieżek dostępu i dane wpisywane przez użytkowników są przedstawiane następująco: „Wewnątrz podkatalogu C:\chapter8-benchmark\src\main\java\com\packt znajduje się plik MyBenchmark.java”.

Adresy URL pisane są kursywą.

Blok kodu jest formatowany następująco:

```
public synchronized void protectedMethod()
{
    ...
}
```

Nowe terminy i ważne słowa są pogrubione.



Ostrzeżenia i ważne uwagi są oznaczone w taki sposób.



Wskazówki są oznaczone w ten sposób.

Informacje od czytelników

Informacje zwrotne od naszych czytelników są zawsze mile widziane. Prosimy o opinie na temat tej książki – co się w niej podoba, a co nie. Informacje od czytelników są dla nas ważne, żebyśmy mogli przygotowywać najbardziej pożądane tytuły.

W celu przekazania ogólnych uwag wystarczy wysłać e-maila na adres feedback@packtpub.com, podając tytuł książki w temacie wiadomości.

Jeśli ktoś jest ekspertem w jakiejś dziedzinie i chciałby napisać lub uczestniczyć w pracach nad książką, może zajrzeć do naszego przewodnika dla autorów pod adresem www.packtpub.com/authors.

Pobieranie przykładowego kodu

Pliki przykładowego kodu dla tej książki można pobrać ze strony <http://www.ksiazki.promise.pl/aspx/produkt.aspx?pid=112120>. Po pobraniu plików należy je rozpakować za pomocą najnowszej wersji jednego z poniższych programów:

- WinRAR lub 7-Zip w systemie Windows
- Zipeg, iZip lub UnRarX w systemie Mac
- 7-Zip lub PeaZip w systemie Linux

Pakiet z kodem dla tej książki jest też dostępny w serwisie GitHub pod adresem <https://github.com/PacktPublishing/Mastering-Java-9>. Również inne pakiety kodu z naszego bogatego katalogu książek i filmów są dostępne pod adresem <https://github.com/PacktPublishing/>. Warto je sprawdzić!

Errata

Chociaż podjęliśmy wszelkie starania, aby zapewnić poprawność treści tej książki, błędy czasem się zdarzają. W razie znalezienia błędu w jednej z naszych książek – na przykład błędu w tekście lub w kodzie – bylibyśmy wdzięczni za zgłoszenie go nam. Dzięki temu możemy oszczędzić innym czytelnikom kłopotów i poprawić kolejne wersje tej książki. Wszelkie poprawki prosimy zgłaszać pod adresem <http://www.packtpub.com/submit-errata>, wybierając daną książkę, klikając łącze **Errata Submission Form** i wprowadzając szczegóły błędu. Po zatwierdzeniu zgłoszenia, zostanie ono przyjęte i opublikowane na naszej stronie WWW lub dodane do listy istniejących poprawek w części Errata dla danego tytułu.

Wcześniej przesłane poprawki można znaleźć pod adresem <https://www.packtpub.com/bookscontent/support>, wpisując tytuł (oryginalny, tzn. angielski) książki w polu wyszukiwania. Żądane informacje pojawiają się w części **Errata**.

Piractwo

Internetowe piractwo materiałów chronionych prawem autorskim jest stałym problemem. W wydawnictwie Packt bardzo poważnie traktujemy ochronę naszych praw autorskich i licencji. W razie natrafienia w Internecie na nielegalne egzemplarze naszych prac w jakiegokolwiek formie prosimy o podanie nam adresu lub nazwy strony WWW, abyśmy mogli podjąć stosowne działania.

Prosimy o kontakt pod adresem copyright@packtpub.com z podaniem łącza do materiału podejrzanego o piractwo.

Dziękujemy za pomoc w ochronie naszych autorów i naszych możliwości dostarczania cennych treści.

Pytania

W przypadku problemów z dowolnym aspektem tej książki prosimy o kontakt pod adresem questions@packtpub.com, a postaramy się pomóc w miarę naszych możliwości.

1

Krajobraz języka Java 9

Po ponad 20 latach swojego istnienia Java jest już w pełni dojrzałym językiem. Dzięki wspaniałej społeczności programistów i szerokiemu przyjęciu w licznych gałęziach przemysłu, platforma ta nadal ewoluuje i dotrzymuje kroku innym technologiom w zakresie wydajności, zabezpieczeń i skalowalności. Zaczniemy naszą podróż od zbadania najistotniejszych funkcji wprowadzonych w wersji Java 9, największych czynników napędzających te zmiany oraz elementów, których możemy oczekiwać w kolejnych wydaniach tej platformy, a które nie zmieściły się w najnowszym wydaniu.

W tym rozdziale omówimy następujące zagadnienia:

- Java 9 z lotu ptaka
- Burzenie monolitu
- Wykorzystanie powłoki Java Shell
- Sterowanie procesami zewnętrznymi
- Podnoszenie wydajności dzięki G1
- Mierzenie wydajności przy pomocy JMH
- Gotowość na HTTP 2.0
- Zastosowanie programowania reaktywnego
- Poszerzanie listy życzeń

Java 9 z lotu ptaka

Można by sobie zadać pytanie, czy Java 9 nie jest po prostu wydaniem serwisowym z dodanymi funkcjami, które nie zmieściły się w wersji Java 8? Wydaje mi się, że Java 9 zawiera mnóstwo nowych elementów, które czynią z niej pełnoprawną, nową wersję.

Bez wątplenia modularyzacja platformy Java (rozwijana jako część projektu Jigsaw) jest największym, nowym elementem wersji Java 9. Początkowo projekt Jigsaw był planowany dla wersji Java 8, ale został przesunięty i był jednym z głównych powodów dalszego opóźnienia ostatecznego wydania wersji Java 9. Jigsaw wprowadza też kilka znaczących zmian do platformy Java i jest jednym z powodów, dla których Java 9 może być uważana za znaczące wydanie. Zajmiemy się szczegółowo tymi funkcjami w kolejnych rozdziałach.

JCP (Java Community Process) zapewnia mechanizmy zmieniające propozycje nowych funkcji (znane także jako **JEPs – Java Enhancement Proposals**) w formalne specyfikacje, stanowiące podstawę do rozszerzania platformy o nowe funkcjonalności. Wersja Java 9 nie różni się w tej kwestii od innych. Oprócz propozycji rozszerzeń języka Java związanych z projektem Jigsaw, istnieje długa lista innych usprawnień, które zostały wprowadzone w wersji Java 9. W tej książce omówimy poszczególne funkcje w postaci logicznych grup opartych na odpowiadających im propozycjach zmian, do których należą:

- Powłoka **Java Shell** (zwana również **JShell**) – interaktywna powłoka dla platformy Java.
- Nowe interfejsy API do pracy z procesami systemu operacyjnego w przenośny sposób.
- Moduł odśmieciania **Garbage-first (G1)**, wprowadzony w wersji Java 7, stał się domyślnym modulem odśmieciania pamięci w Java 9.
- Narzędzie **Java Microbenchmark Harness (JMH)**, które może być używane do przeprowadzania pomiarów wydajnościowych aplikacji Java, jest dołączone jako część dystrybucji platformy Java.
- Obsługa standardów HTTP 2.0 oraz WebSocket poprzez nowe usprawnienia klienckich interfejsów API Concurrency, wśród których znajduje się definicja klasy Flow, opisująca interfejs dla specyfikacji strumieni reaktywnych na platformie Java.

Niektóre z początkowych propozycji, które zostały przyjęte do wdrożenia w wersji 9, nie zmieściły się w niej i zostały przełożone na późniejsze wydanie razem z innymi interesującymi elementami, których programiści mogą oczekiwać w przyszłości.

Dystrybucję JDK 9 dla swojego systemu można pobrać ze strony <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, jeśli ktoś chce zacząć już teraz i poeksperymentować z nowymi pojęciami oraz przykładami, zanim przejdziemy do kolejnych rozdziałów.

Burzenie monolitu

Przez lata użyteczność platformy Java stale się rozwijała i zwiększała, czyniąc z niej jeden, wielki monolit. Aby uczynić tę platformę bardziej przydatną dla urządzeń mobilnych, konieczna była publikacja uproszczonych edycji, takich jak Java CDC i Java ME. Okazały się one jednak zbyt mało elastyczne dla nowoczesnych aplikacji o zróżnicowanych wymaganiach względem funkcjonalności zapewnianych przez JDK. Konieczność wprowadzenia modularnego systemu pojawiła się jako oddolne wymaganie, mające na celu nie tylko modularyzację narzędzi Java (w sumie ponad 5000 klas Java i 1500 plików źródłowych C++ z ponad 250000 wierszami kodu dla środowiska uruchomieniowego Hotspot), ale również zapewnienie programistom mechanizmu tworzenia i zarządzania modularnymi aplikacjami przy użyciu tego samego systemu modułów, co wykorzystywany w JDK. Wersja Java 8 zapewniała pośredni mechanizm umożliwiający aplikacjom wykorzystanie jedynie podzbioru interfejsów API zapewnianych przez cały zestaw JDK, a mechanizm ten nosił nazwę profili kompaktowych. W istocie profile kompaktowe stanowią również podstawę dalszych prac, które należy przeprowadzić, aby zerwać zależności pomiędzy różnymi, oddzielnymi składnikami JDK, wymaganymi w celu umożliwienia implementacji systemu modułów na platformie Java.

Sam system modułów został opracowany pod nazwą projektu Jigsaw, na podstawie którego sformułowano kilka propozycji ulepszeń platformy Java oraz docelową specyfikację JSR (376). Wiele elementów złożyło się na spełnienie wymagań projektu Jigsaw – testowano też implementację dodatkowych funkcji poza tymi, którym udało się wejść do wersji Java 9. Poza tym przeprowadzono zupełną przebudowę bazy kodu JDK wraz z całkowitą reorganizacją dystrybucyjnych obrazów JDK.

W środowisku dyskutowano, czy nie należałoby zaadaptować istniejącego i rozwiniętego systemu modułów Java takiego jak OSGI jako części JDK, zamiast zapewniać całkowicie nowy system modułów. OSGI dotyczy jednak czasu wykonywania aplikacji i zajmuje się rozstrzygnięciem zależności między modułami, instalowaniem, odinstalowaniem, uruchamianiem i zatrzymywaniem modułów (zwanymi również pakietami), niestandardowych programów ładujących moduły, itd. Natomiast projekt Jigsaw dotyczy systemu modułów w czasie kompilacji, gdzie ustalanie zależności odbywa się podczas kompilowania aplikacji. Co więcej, instalowanie i odinstalowywanie modułu

jako części JDK eliminuje potrzebę dołączania go jako zależności bezpośrednio podczas kompilacji. Co więcej, ładowanie klas modułu jest możliwe poprzez istniejącą hierarchię programów ładujących (program inicjujący i rozszerzenia systemowych programów ładujących), choć istniała możliwość wykorzystania niestandardowych programów ładujących moduły podobnie jak w systemie OSGI. Została ona jednak porzucona; ładowanie modułów Java omówimy bardziej szczegółowo, gdy będziemy zajmować się systemem modułów w języku Java.

Dodatkową korzyść z systemu modułów Java stanowi zwiększenie bezpieczeństwa i wydajności. Poprzez dzielenie JDK i aplikacji na moduły Jigsaw jesteśmy w stanie tworzyć dobrze zdefiniowane granice pomiędzy składnikami i ich domenami. Ten podział zadań dopasowuje się do architektury zabezpieczeń platformy i umożliwia lepsze wykorzystanie zasobów. Poświęciliśmy dwa rozdziały na wszystkie powyższe punkty oraz na proces wdrażania Java 9, który również wymaga pewnego stopnia zrozumienia możliwych sposobów migrowania istniejących projektów do platformy Java 9.

Wykorzystanie powłoki Java Shell

Przez długi czas nie było żadnej standardowej powłoki dostarczanej z językiem programowania Java do eksperymentowania z nowymi funkcjami języka lub bibliotekami przy szybkim prototypowaniu kodu. W tym celu można było napisać aplikację testową z metodą `main`, skompilować ją przy pomocy `javac` i uruchomić. Można to było zrobić albo z wiersza polecenia, albo przy użyciu środowiska Java IDE; jednakże w obu przypadkach nie jest to tak wygodne, jak interaktywna powłoka przeznaczona specjalnie do tego celu.

Uruchomienie interaktywnej powłoki w JDK 9 jest tak proste, jak wywołanie następującego polecenia (zakładając, że katalog `bin` instalacji JDK 9 znajduje się w bieżącej ścieżce dostępu):

```
jshell
```

Dla wielu może być dość zaskakujące, że interaktywna powłoka nie została wprowadzona wcześniej na platformie Java, ponieważ wiele języków programowania, takich jak Python, Ruby oraz wiele innych, zawierało już interaktywną powłokę od swoich najwcześniejszych wersji. Jednakże nie dotarło to na szczyt listy priorytetowych funkcji do wprowadzenia we wcześniejszych wersjach Java, aż do obecnej wersji. Powłoka Java wykorzystuje interfejs `JShell API`, który zapewnia możliwości pozwalające między innymi na autouzupełnianie oraz wyliczanie wyrażeń lub fragmentów kodu. Cały

rozdział został poświęcony na omówienie szczegółów powłoki Java, aby programiści mogli maksymalnie z niej skorzystać.

Sterowanie procesami zewnętrznymi

Aż do wersji JDK 9, aby utworzyć proces Java i obsługiwać wejście/wyjście procesu, trzeba było skorzystać albo z metody `Runtime.getRuntime().exec()`, która pozwala nam wykonać polecenie w oddzielnym procesie systemu operacyjnego i uzyskać wystąpienie klasy `java.lang.Process` zapewniające pewne operacje związane z zarządzaniem procesem zewnętrznym, albo skorzystać z nowej klasy `java.lang.ProcessBuilder` z dodatkowymi ulepszeniami związanymi z interakcją z procesem zewnętrznym i również utworzyć wystąpienie klasy `java.lang.Process` reprezentujące proces zewnętrzny. Oba mechanizmy były mało elastyczne i nieprzenośne, ponieważ zestaw poleceń wykonywanych przez procesy wewnętrzne w wysokim stopniu zależał od systemu operacyjnego (trzeba było podejmować dodatkowe wysiłki, aby uczynić pewne operacje na procesach przenośnymi dla różnych systemów operacyjnych). Jeden rozdział jest poświęcony nowemu interfejsowi API dla procesów, dając programistom wiedzę na temat tworzenia i zarządzania procesami zewnętrznymi w znacznie prostszy sposób.

Podnoszenie wydajności dzięki G1

System odśmiecania pamięci G1 został już wprowadzony w JDK 7, a obecnie w wersji JDK 9 jest włączony domyślnie. Jest przeznaczony dla systemów z procesorami wielordzeniowymi i dużą ilością dostępnej pamięci. Jakie są zalety G1 w porównaniu z poprzednimi rodzajami systemów odśmiecania pamięci? Jak uzyskuje się te ulepszenia? Czy jest potrzeba ręcznego dostrajania go i w jakich scenariuszach? Osobny rozdział zajmie się odpowiedziami na te pytania i kilka innych dotyczących G1.

Mierzenie wydajności przy pomocy JMH

W wielu sytuacjach aplikacje Java mogą cierpieć z powodu słabej wydajności.

Problem pogłębia brak testów wydajnościowych, które mogą zapewniać co najmniej minimalny zestaw gwarancji, że wymagania wydajnościowe są spełnione, a przy tym wydajność pewnych funkcji nie będzie zmniejszać się z czasem. Mierzenie wydajności aplikacji Java nie jest trywialne, zwłaszcza ze względu na fakt, że istnieje wiele optymalizacji kompilatora i środowiska uruchomieniowego, które mogą wpływać na statystyki wydajnościowe. Z tego powodu trzeba korzystać z dodatkowych

środków, takich jak fazy rozruchowe i inne sztuczki, aby zapewniać dokładniejsze pomiary wydajności. Java Microbenchmark Harness jest platformą obejmującą wiele technik oraz wygodny interfejs API, które mogą służyć do tego celu. Nie jest to nowe narzędzie, ale zostało dołączone do dystrybucji Java 9. Jeśli ktoś nie dodał jeszcze JMH do swojego zestawu narzędzi, to powinien przeczytać szczegółowy rozdział dotyczący wykorzystania JMH w kontekście tworzenia aplikacji Java 9.

Wprowadzenie do HTTP 2.0

Protokół HTTP 2.0 jest następcą HTTP 1.1, a ta nowa wersja protokołu odpowiada na pewne ograniczenia i wady poprzedniej wersji. HTTP 2.0 poprawia wydajność na kilka sposobów i zapewnia możliwości, takie jak wielokrotne żądania/odpowiedzi w pojedynczym połączeniu TCP, wysyłanie odpowiedzi w trybie wypychania, sterowanie przepływem, priorytetyzację żądań, itd.

Java zapewnia narzędzie `java.net.HttpURLConnection`, które może być używane do ustanawiania niezabezpieczonego połączenia HTTP 1.1. Ten interfejs API był jednak uważany za trudny w utrzymaniu, wprowadzono więc całkowicie nowy kliencki interfejs API do ustanawiania połączenia poprzez protokół HTTP 2.0 lub gniazda WWW. Nowy klient HTTP 2.0 oraz zapewniane przez niego możliwości zostaną omówione w specjalnie mu poświęconym rozdziale.

Zastosowanie programowania reaktywnego

Programowanie reaktywne jest paradygmatem używanym do opisywania pewnego wzorca rozprzestrzeniania się zmian w systemie. Reaktywność nie jest wbudowana w samą platformę Java, ale reaktywne przepływy danych mogą być ustanawiane przy użyciu bibliotek firm trzecich, takich jak RxJava lub projekt Reactor (część platformy Spring Framework). JDK 9 odpowiada również na potrzebę istnienia interfejsu API, który pomaga tworzyć wysoce responsywne aplikacje budowane wokół pojęcia strumieni reaktywnych, zapewniając do tego celu klasę `java.util.concurrent.Flow`. Klasa `Flow` wraz z innymi powiązаныmi zmianami wprowadzonymi w JDK 9 zostanie omówiona w osobnym rozdziale.

Poszerzanie listy życzeń

Poza wszystkimi nowościami w JDK 9, w przyszłych wydaniach platformy można się spodziewać całego zestawu nowych funkcji. Należą do nich między innymi:

- **Typy ogólne dla typów podstawowych:** Jest to jedna z funkcji planowanych dla wersji JDK 10 jako część projektu Valhalla. Inne usprawnienia języka, takie jak dojścia do wartości są już częścią Java 9 i zostaną omówione w dalszej części tej książki.
- **Reifikowane typy ogólne:** Jest to kolejna część projektu Valhalla, której celem jest zapewnienie możliwości zachowywania typów ogólnych w trakcie wykonywania programu. Powiązane cele są określone następująco:
 - Obcy interfejs funkcjonalny ma na celu wprowadzenie nowego interfejsu API do wywoływania i zarządzania funkcjami rodzimymi. Interfejs API odpowiada na niektóre wady JNI, a szczególnie brak prostoty użycia przez programistów aplikacji. Obcy interfejs funkcjonalny jest opracowywany jako część projektu Panama w ekosystemie JDK.
 - Nowy interfejs API dla wartości pieniężnych (opracowywany w ramach specyfikacji JSR 354) był początkowo planowany dla wersji Java 9, ale został przełożony na później.
 - Nowy, lekki interfejs JSON API (opracowywany w ramach specyfikacji JSR 353) również był planowany dla wersji Java 9, ale został odłożony do wersji Java 10.

To tylko kilka nowych elementów, których możemy się spodziewać w kolejnych wydaniach JDK. Celem projektu Penrose jest uzupełnienie luki pomiędzy systemem modułów w języku Java a systemem modułów OSGi oraz zapewnienie różnych sposobów dla interoperacyjności pomiędzy tymi dwoma systemami.

Graal VM jest kolejnym interesującym projektem badawczym, który jest potencjalnym kandydatem dla kolejnych wydań platformy Java. Jego celem jest udostępnienie wydajności środowiska wykonawczego Java dla języków dynamicznych, takich jak JavaScript lub Ruby.

Rozdział poświęcony przyszłości JDK szczegółowo omawia wszystkie te elementy.

Podsumowanie

W tym krótkim rozdziale wprowadzającym ujawniliśmy niewielką część możliwości zapewnianych przez JDK 9. System modułów, wprowadzony w tym wydaniu platformy, jest bez wątpienia kamieniem węgielnym w rozwoju aplikacji Java. Odkryliśmy również, że wiele innych ważnych funkcji i zmian wprowadzonych w JDK 9 zasługuje na specjalną uwagę i zostaną one szczegółowo omówione w kolejnych rozdziałach.

W następnym rozdziale przyjrzymy się 26 wewnętrznym zmianom wprowadzonym na platformie Java.

2

Odkrywanie Java 9

Java 9 jest znaczącym wydaniem środowiska Java i składa się z dużej liczby wewnętrznych zmian tej platformy. Te wewnętrzne zmiany stanowią łącznie ogromny zestaw nowych możliwości dla programistów Java. Niektóre z nich zostały zgłoszone przez programistów, a inne zostały zainspirowane przez Oracle. W tym rozdziale dokonamy przeglądu 26 najważniejszych zmian. Każda zmiana jest powiązana z propozycją **JDK Enhancement Proposal (JEP)**. Dokumenty JEP są indeksowane i przechowywane pod adresem <http://openjdk.java.net/jeps/0>. Na tej witrynie można uzyskać dodatkowe informacje na temat każdego dokumentu JEP.



Program propozycji JEP jest częścią wsparcia Oracle dla otwartego kodu, otwartych innowacji i otwartych standardów. Choć można znaleźć inne projekty Java z otwartym kodem źródłowym, to Oracle wspiera jedynie OpenJDK.

W tym rozdziale omówimy zmiany wprowadzone na platformie Java. Zmiany te mają kilka znaczących konsekwencji, w tym:

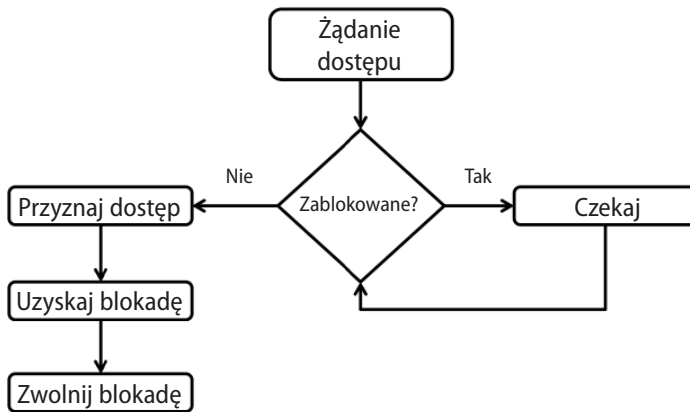
- Efektywność pamięci serty
- Alokacja pamięci
- Usprawnienia procesu kompilacji
- Adnotacje do testowania typów
- Zautomatyzowane testy kompilatora
- Poprawione odśmiecianie pamięci

Poprawione sporne blokowanie [JEP 143]

JVM wykorzystuje pamięć sterty dla klas i obiektów. JVM alokuje pamięć na stercie, zawsze gdy tworzymy jakiś obiekt. Pomaga to wykorzystywać system odśmiecania na platformie Java, który zwalnia pamięć wcześniej używaną do przechowywania obiektów, do której nie ma już żadnych odwołań. Pamięć stosu Java jest nieco inna i zwykle jest znacznie mniejsza niż pamięć sterty.

JVM dobrze sprawuje się przy zarządzaniu obszarami danych wspólnie wykorzystywanych przez wiele wątków. Kojarzy monitor z każdym obiektem i klasą; monitory te mają blokady będące w danym momencie pod kontrolą pojedynczego wątku. Blokady te, sterowane przez JVM, przekazują w istocie monitor obiektu kontrolującemu wątkowi.

Czym jest więc sporne blokowanie? Gdy wątek czeka w kolejce na aktualnie zablokowany obiekt, mówimy, że znajduje się w konflikcie o tę blokadę. Poniższy diagram pokazuje ogólny schemat tego konfliktu:



Jak widać na poniższej ilustracji, wątki oczekujące nie mogą skorzystać z zablokowanego obiektu, dopóki nie zostanie on zwolniony.

Cele poprawy

Ogólnym celem JEP 143 było zwiększenie całościowej wydajności zarządzania przez JVM konfliktami związanymi z zablokowanymi monitorami obiektów Java. Usprawnienia spornego blokowania zostały wprowadzone wewnątrz JVM i nie wymagają od programisty żadnych działań w celu osiągnięcia z nich korzyści. Cele usprawnienia były związane z przyspieszeniem operacji. Należą do nich:

- Szybsze wejście do monitora
- Szybsze wyjście z monitora
- Szybsze powiadomienia

Powiadomienia te są operacjami `notify()` i `notifyAll()`, które są wywoływane, gdy status zablokowania obiektu się zmienia. Nie jest łatwo przetestować te usprawnienia. Większa wydajność jest przydatna na każdym poziomie, więc możemy być wdzięczni za to usprawnienie, choć nie jest łatwo zaobserwować je w testach.

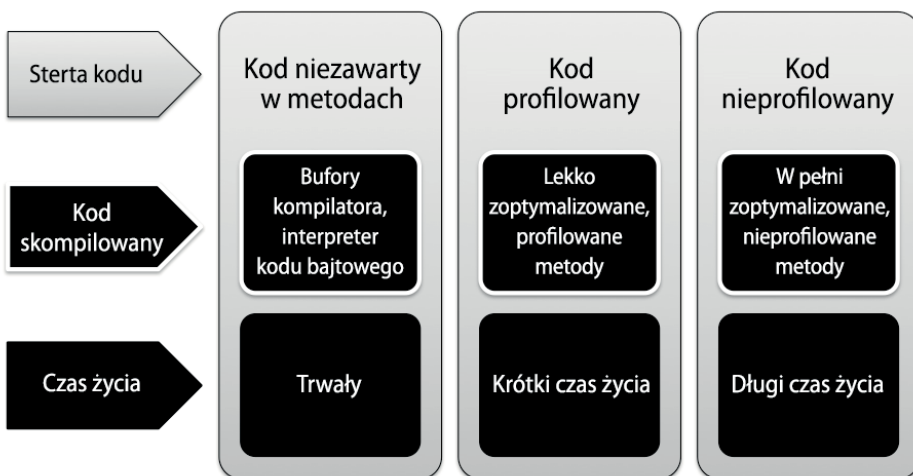
Dzielona pamięć podręczna kodu [JEP 197]

Aktualizacja związana z dzieloną pamięcią podręczną kodu (JEP 197) została ukończona i daje w efekcie szybsze i wydajniejsze czasy wykonywania kodu. Sednem tych zmian było dzielenie pamięci podręcznej kodu na trzy odrębne odcinki – kod niezawarty w metodach, kod profilowany i kod nieprofilowany.



Pamięć podręczna kodu jest obszarem pamięci, w którym Java Virtual Machine przechowuje wygenerowany kod rodzimy.

Każdy z wymienionych wyżej odcinków pamięci podręcznej kodu będzie przechowywał określony typ skompilowanego kodu. Jak widać na poniższym diagramie, obszary sterty kodu są dzielone według typu skompilowanego kodu:



Alokacja pamięci

Sterta kodu przechowująca kod niezawarty w metodach jest przeznaczona dla wewnętrznego kodu JVM i składa się ze stałego bloku pamięci o wielkości 3 MB. Pozostała część pamięci podręcznej kodu jest równo podzielona na segmenty kodu profilowanego i kodu nieprofilowanego. Możemy to zmieniać poprzez polecenia wiersza poleceń.

Poniższe polecenie może być używane do definiowania rozmiaru sterty kodu dla skompilowanego kodu niezawartego w metodach:

-XX:NonMethodCodeCodeHeapSize

Poniższe polecenie może być używane do definiowania rozmiaru sterty kodu dla profilowanych, skompilowanych metod:

-XX:ProfiledCodeHeapSize

Poniższe polecenie może być używane do definiowania rozmiaru sterty kodu dla nieprofilowanych, skompilowanych metod:

-XX:NonProfiledCodeHeapSize

Ta funkcja platformy Java 9 z pewnością poprawia wydajność aplikacji Java. Wpływa ona też na inne procesy wykorzystujące pamięć podręczną kodu.

Kompilacja Smart Java, faza druga [JEP 199]

Dokument JDK Enhancement Proposal 199 ma na celu poprawienie procesu kompilacji kodu. Wszyscy programiści Java znają narzędzie **javac** do kompilowania kodu źródłowego na kod bajtowy, który jest używany przez JVM do wykonywania programów Java. **Kompilacja Smart Java**, nazywana też Smart Javac albo **sjavac**, dodaje inteligencję (*smart*) do procesu javac.

Najważniejszym usprawnieniem wprowadzanym przez sjavac jest chyba to, że ponownie kompilowany jest tylko niezbędny kod. Niezbędny w tym kontekście oznacza kod, który został zmieniony od ostatniego cyklu kompilacji.

To usprawnienie może nie być ekscytujące dla programistów, jeśli pracują tylko nad niewielkimi projektami. Weźmy jednak pod uwagę ogromne zyski wydajnościowe, gdy stale musimy kompilować swój kod dla średnich lub dużych projektów. Czas, jaki programiści mogą w ten sposób zaoszczędzić, jest wystarczającym powodem do wprowadzenia JEP 199.

Jak zmieni to sposób kompilowania przez nas kodu? Na razie pewnie w niewielkim stopniu. Javac pozostanie domyślnym kompilatorem. Chociaż sjavac oferuje lepszą wydajność w przypadku przyrostowych kompilacji, firma Oracle uznała, że nie ma to jeszcze odpowiedniej stabilności, aby stać się częścią standardowego przepływu zadań związanych z kompilacją.



Więcej informacji na temat narzędzia javac można uzyskać pod adresem: <http://cr.openjdk.java.net/~briangoetz/JDK-8030245/webrev/src/share/classes/com/sun/tools/sjavac/Main.java-.html>.

Obsługa ostrzeżeń z narzędzi Lint i Doclint [JEP 212]

Nie trzeba się przejmować, jeśli ktoś nie zna narzędzi Lint i Doclint na platformie Java. Jak można się domyślić z tytułu tego podrozdziału, są one źródłami zgłaszającymi ostrzeżenia dla javac. Przyjrzyjmy się każdemu z tych narzędzi:

- **Lint** analizuje kod bajtowy i kod źródłowy dla javac. Celem narzędzia Lint jest wykrywanie luk bezpieczeństwa w analizowanym kodzie. Lint może też zapewnić rozeznanie w sprawach skalowalności i blokowania wątków. Lint ma też inne zastosowania, a ogólnym celem tego narzędzia jest oszczędność czasu programisty.



Więcej na temat Lint można przeczytać pod adresem: [https://en.wikipedia.org/wiki/Lint_\(software\)](https://en.wikipedia.org/wiki/Lint_(software)).

- **Doclint** jest podobnym narzędziem do Lint, ale jest konkretnie przeznaczony dla javadoc. Zarówno Lint, jak i Doclint zgłaszają błędy i ostrzeżenia podczas procesu kompilacji. JEP 212 skupia się na obsłudze tych ostrzeżeń. Podczas korzystania z bibliotek podstawowych nie powinny występować żadne ostrzeżenia. To nastawienie doprowadziło do opracowania propozycji JEP 212, która została zaimplementowana w wersji Java 9.



Rozbudowaną listę ostrzeżeń Lint i Doclint można przejrzeć w serwisie JDK Bug System pod adresem <https://bugs.openjdk.java.net>.

Warstwowe przypisywanie typów w javac [JEP 215]

JEP 215 stanowi imponujące przedsięwzięcie w zakresie optymalizacji planu sprawdzania typów w javac. Przyjrzymy się najpierw, jak działa sprawdzanie typów w Java 8, a następnie zbadamy zmiany wprowadzone w wersji Java 9.

W Java 8 sprawdzanie typów dla rozbudowanych wyrażeń jest obsługiwane przez **przypisywanie spekulatywne**.



Przypisywanie spekulatywne jest metodą sprawdzania typów będącą częścią procesu kompilacji javac. Wiąże się to ze znaczącymi kosztami przetwarzania.

Użycie tego podejścia do sprawdzania typów jest dokładne, ale brakuje mu wydajności. Sprawdzenia te zawierają pozycje argumentów i są znacznie wolniejsze przy testowaniu wewnątrz rekurencji, polimorfizmu, zagnieżdżonych pętli i wyrażeń lambda. Celem JEP 215 była więc zmiana planu sprawdzania typów w celu uzyskania szybszych wyników. Same wyniki uzyskiwane dzięki przypisywaniu spekulatywnemu nie były niedokładne, ale nie były generowane wystarczająco szybko.

Nowe podejście, wprowadzone w wersji Java 9, wykorzystuje przypisywanie warstwowe. Narzędzie to implementuje warstwowe podejście do sprawdzania typów argumentów w wyrażeniach dla wszystkich wywołań metod. Obsługiwane są też nadpisanie metod. Aby ten nowy plan działał, tworzone są nowe typy strukturalne dla każdego z następujących typów argumentów metod:

- Wyrażenia lambda
- Wyrażenia złożone (poly)
- Zwykłe wywołania metod
- Odwołania do metod
- Wyrażenia tworzące wystąpienia w formie rombowej (<>)

Zmiany javac wynikające z JEP 215 są bardziej złożone, niż zostało to omówione w tym podrozdziale. Nie ma to większego wpływu na programistów, poza efektywniejszym działaniem javac i zaoszczędzonym czasem.

Obsługa adnotacji 2.0 [JEP 217]

Adnotacje Java odnoszą się do specjalnego rodzaju metadanych znajdujących się wewnątrz plików z kodem źródłowym Java. Nie są one usuwane przez javac, więc mogą być nadal dostępne dla JVM podczas działania programu.

Adnotacje wyglądają podobnie do odwołań JavaDocs, ponieważ zaczynają się od symbolu @. Istnieją trzy typy adnotacji. Zbadajmy każdy z nich:

- Najbardziej podstawową formą adnotacji jest **marker**. Są to samodzielne adnotacje, których jedynym składnikiem jest nazwa adnotacji. Oto przykład:

```
@thisIsAMarkerAnnotation
public double computeSomething(double x, double y)
{
    // zrób coś i zwróć wartość double
}
```

- Drugi typ adnotacji zawiera *pojedynczą wartość*, czyli jakieś dane. Jak widać w poniższym kodzie, po adnotacji zaczynającej się od symbolu @ następuje nawias zawierający dane:

```
@thisIsAMarkerAnnotation (data="compute x and y coordinates")
public double computeSomething(double x, double y)
{
    // zrób coś i zwróć wartość double
}
```

Alternatywnym sposobem zakodowania adnotacji z pojedynczą wartością jest pominięcie składnika data=, jak pokazano na poniższym kodzie:

```
@thisIsAMarkerAnnotation ("compute x and y coordinates")
public double computeSomething(double x, double y)
{
    // zrób coś i zwróć wartość double
}
```

- W trzecim typie adnotacji występuje *więcej niż jeden składnik danych*. Przy tym rodzaju adnotacji nie można pominąć składnika data=. Oto przykład:

```
@thisIsAMarkerAnnotation (data="compute x and y
    coordinates", purpose="determine intersecting point")
public double computeSomething(double x, double y)
{
    // zrób coś i zwróć wartość double
}
```


Co się więc zmieniło w wersji Java 9? Aby odpowiedzieć na to pytanie, musimy przypomnieć sobie kilka zmian wprowadzonych w Java 8, które miały wpływ na adnotacje Java:

- Wyrażenia lambda
- Powtarzane adnotacje
- Adnotacje typów Java

Te zmiany związane z Java 8 wpłynęły na adnotacje Java, ale nie zapoczątkowały zmian w sposobie przetwarzania ich przez javac. Istniały pewne rozwiązania pozwalające obsługiwać nowe adnotacje przez javac, ale nie były one efektywne. Co więcej ten typ kodowania (bezpośrednie kodowanie obejść) jest trudny w utrzymaniu.

Dlatego propozycja JEP 217 skupiła się na refaktoringu potoku przetwarzania adnotacji przez javac. Refaktoring ten dotyczył wewnętrznych elementów javac, więc nie powinien być widoczny dla programistów.

Nowy schemat łańcucha wersji [JEP 223]

Przed wydaniem Java 9 numery wersji nie odpowiadały standardowi przemysłowemu dotyczącemu wersjonowania – **wersjonowaniu semantycznemu**. Na przykład ostatnie cztery wydania JDK miały numery wersji:

- JDK 8 update 131
- JDK 8 update 121
- JDK 8 update 112



Wersjonowanie semantyczne wykorzystuje schemat składający się z numeru głównego, numeru pomniejszego i numeru łatki (0.0.0):

- ▶ **Numer główny** oznacza nowe zmiany w interfejsach API, które nie są wstecznie kompatybilne.
- ▶ **Numer pomniejszy** zmienia się, gdy dodawana jest funkcjonalność, która zachowuje kompatybilność wsteczną.
- ▶ **Numer łatki** oznacza poprawki błędów lub drobniejsze zmiany, które są wstecznie kompatybilne.

Firma Oracle przyjęła system semantycznego wersjonowania dla wersji Java 9 i kolejnych. W przypadku platformy Java stosowany będzie schemat **numer główny-numer pomniejszy-numer zabezpieczeń** dla pierwszych trzech elementów numerów wersji Java:

- **Numer główny:** Główne wydanie platformy składające się ze znaczącego zestawu nowych funkcji
- **Numer pomniejszy:** Aktualizacje i poprawki błędów, które są wstecznie kompatybilne
- **Numer zabezpieczeń:** Poprawki uważane za krytyczne dla poprawy bezpieczeństwa

Na podstawie tego opisu propozycji JEP 223, schemat wersjonowania może wydawać się uproszczony. W istocie opracowano bardzo szczegółowy zestaw reguł i praktyk związanych z zarządzaniem przyszłymi numerami wersji. Możemy zademonstrować tę złożoność na następującym przykładzie:

```
1.9.0._32.b19
```

Automatyczne generowanie testów kompilatora w czasie wykonywania programu [JEP 233]

Java jest niewątpliwie najczęściej używanym językiem programowania, dostępnym na coraz większej liczbie różnorodnych platform. Zaostrza to problem związany z uruchamianiem ukierunkowanych testów kompilatora w efektywny sposób. Celem JEP 233 było stworzenie narzędzia, które mogłoby zautomatyzować testy kompilatora.

Utworzone narzędzie rozpoczyna od wygenerowania losowego zestawu kodu źródłowego Java i/lub kodu bajtowego. Wygenerowany kod będzie miał trzy kluczowe cechy:

- Będzie poprawny składniowo.
- Będzie poprawny semantycznie.
- Wykorzystuje losowe ziarno pozwalające na ponowne wykorzystanie tego samego losowo wygenerowanego kodu.

Losowo wygenerowany kod źródłowy będzie zapisywany w następującym katalogu:

```
hotspot/test/testlibrary/jit-tester
```

Te przypadki testowe będą zachowywane do późniejszego ponownego wykorzystania. Mogą być uruchamiane z katalogu `j-treg` albo z pliku `makefile` tego narzędzia. Jedną z zalet ponownego uruchamiania zapisanych testów jest testowanie stabilności systemu.

Testowanie atrybutów plików klas generowanych przez Javac [JEP 235]

Niewystarczające możliwości (bądź ich brak) tworzenia testów dla atrybutów plików klas stanowiły bodziec do powstania propozycji JEP 235. Celem jest zapewnienie całkowitego i poprawnego tworzenia atrybutów plików klas przez javac. Oznacza to, że nawet jeśli jakieś atrybuty nie są wykorzystywane przez plik klasy, to wszystkie pliki klas powinny być generowane z pełnym zestawem atrybutów. Musi też istnieć sposób testowania, czy pliki klas zostały utworzone poprawnie, jeśli chodzi o atrybuty plików.

Przed wersją Java 9 nie było sposobu testowania atrybutów plików klas. Uruchamianie klasy i testowanie kodu w celu uzyskania oczekiwanych wyników było najczęściej stosowaną metodą testowania plików klas generowanych przez javac. Ta technika nie wystarcza do testowania poprawności atrybutów plików.

Istnieją trzy kategorie atrybutów plików klas – atrybuty wykorzystywane przez JVM, atrybuty opcjonalne i atrybuty niewykorzystywane przez JVM.

Do atrybutów wykorzystywanych przez JVM należą:

- BootstrapMethods
- Code
- ConstantValue
- Exceptions
- StackMapTable

Do atrybutów opcjonalnych należą:

- Deprecated
- LineNumberTable
- LocalVariableTable
- LocalVariableTypeTable
- SourceDebugExtension
- SourceFile

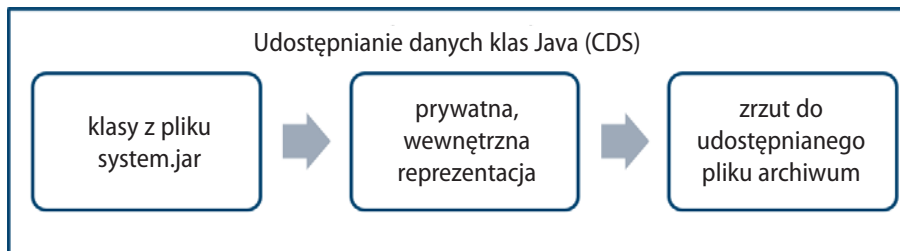
Do atrybutów niewykorzystywanych przez JVM należą:

- AnnotationDefault
- EnclosingMethod
- InnerClasses
- MethodParameters

- RuntimeInvisibleAnnotations
- RuntimeInvisibleParameterAnnotations
- RuntimeInvisibleTypeAnnotations
- RuntimeVisibleAnnotations
- RuntimeVisibleParameterAnnotations
- RuntimeVisibleTypeAnnotations
- Signature
- Synthetic

Przechowywanie łańcuchów tekstowych w archiwach CDS [JEP 250]

Metoda, w której łańcuchy tekstowe są przechowywane w archiwach **CDS (Class Data Sharing)** – udostępnianie danych klas) i z nich pozyskiwane, jest nieefektywna, zbyt czasochłonna i marnuje pamięć. Poniższy diagram ilustruje metodę, w której Java przechowuje łańcuchy tekstowe w archiwum CDS:



Nieefektywność wynika z aktualnego planu przechowywania. Zwłaszcza, gdy narzędzie **Class Data Sharing** zrzuca klasy do udostępnianego pliku archiwum, stałe pule elementów `CONSTANT_String` przedstawiają łańcuchy tekstowe w formie UTF-8.



UTF-8 jest 8-bitowym standardem kodowania znaków o zmiennej długości.

Problem

Przy obecnym użyciu UTF-8, łańcuchy tekstowe muszą być konwertowane na obiekty tekstowe, będące wystąpieniami klasy `java.lang.String`. Konwersja ta odbywa się na żądanie, co może powodować spowolnienie systemów i niepotrzebne zużycie