O'REILLY®

Wydanie VIII

Java w pigułce

Benjamin Evans
Jason Clark
David Flanagan

Helion

Tytuł oryginału: Java in a Nutshell: A Desktop Quick Reference, 8th Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-289-0161-2

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/javpi8
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/javpi8.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 12.

Moduły platformy Javy

W tym rozdziale przedstawiamy wprowadzenie do **systemu modułów platformy Javy** (ang. *Java Platform Modules System* — JPMS). Ponieważ jednak jest to szeroki temat, zainteresowani czytelnicy mogą potrzebować bardziej wyczerpującego źródła, takiego jak książka *Java 9 Modularity* Sandera Maka i Paula Bakkera (wyd. O'Reilly).

Moduły to dość zaawansowana funkcja, która w głównej mierze dotyczy pakowania i wdrażania całych aplikacji i ich zależności. Zostały one dodane do platformy jakieś 20 lat po pojawieniu się pierwszej wersji Javy i można je traktować jako ortogonalną funkcjonalność względem pozostałej składni języka.

Silny nacisk w Javie na zgodność wsteczną jest tu także bez znaczenia, ponieważ niemodułowe aplikacje również nadal muszą działać. To zmusiło architektów i opiekunów platformy Java do przyjęcia pragmatycznego podejścia do wdrażania modułów przez zespoły.

> Nie ma potrzeby przechodzić na moduły.
>
> Nigdy nie było potrzeby przechodzenia na moduły.
>
> Java 9 i późniejsze wersje obsługują tradycyjne pliki JAR na tradycyjnej ścieżce plików, przez koncepcję modułu bez nazwy, i zapewne będą to robić do samego końca świata.
>
> To, czy zacząć korzystać z modułów, zależy wyłącznie od Ciebie.
>
> — Mark Reinhold
> *https://oreil.ly/4RjDH*

Ze względu na zaawansowany charakter modułów w tym rozdziale przyjmujemy, że znacz nowoczesne narzędzia kompilacji Javy, takie jak Gradle lub Maven.

Jeśli jesteś początkującym programistą Javy, możesz spokojnie zignorować odniesienia do tych narzędzi i przeczytać ten rozdział tylko po to, aby wstępnie zapoznać się z JPMS. Początkujący programista tego języka nie musi w pełni rozumieć tej techniki, kiedy dopiero uczy się w ogóle pisać programy.

Dlaczego moduły

Moduły dodano do Javy z kilku ważnych powodów. Między innymi dodano je po to, aby uzyskać:

- silną hermetyzację,
- dobrze zdefiniowane interfejsy,
- bezpośrednie zależności.

Wszystko to jest na poziomie języka (i projektu aplikacji) oraz zostało połączone z obietnicą dodania nowych elementów funkcjonalności na poziomie platformy, jakimi są:

- skalowalne programowanie,
- poprawiona wydajność (w szczególności rozruchu) i mniejsze zużycie pamięci,
- mniejsza powierzchnia ataku i wyższy poziom bezpieczeństwa,
- możliwość rozwoju wewnętrznych mechanizmów.

Kwestia hermetyzacji wiąże się z tym, że oryginalna specyfikacja języka obejmuje tylko te poziomy widoczności: prywatny, publiczny, chroniony i prywatny pakietowy. Nie ma sposobu na precyzyjniejsze kontrolowanie dostępu, aby wyrazić takie koncepcje jak:

- Tylko określone pakiety są dostępne jako API — pozostałe są wewnętrzne i nie ma do nich dostępu.
- Do niektórych pakietów można uzyskać dostęp przez pakiety z tej listy, ale nie do wszystkich.
- Definicja ścisłego mechanizmu eksportu.

Brak tych i pokrewnych możliwości jest istotnie odczuwalny podczas opracowywania architektury większych systemów w Javie. Poza tym, ale bez odpowiedniego mechanizmu ochrony, byłoby bardzo trudno rozwijać wewnętrzne mechanizmy JDK — ponieważ nic nie stoi na przeszkodzie, aby aplikacje użytkowników nie korzystały bezpośrednio z klas implementacyjnych.

System modułów próbuje rozwiązać wszystkie te problemy jednocześnie oraz dostarczyć rozwiązanie, które będzie odpowiednie zarówno dla JDK, jak i aplikacji użytkownika.

Modularyzacja JDK

Monolityczny JDK, który towarzyszył Javie 8, był pierwszym celem dla systemu modułowego — na moduły podzielono znany plik *rt.jar*.

W Javie 8 rozpoczęto prace nad modularyzacją przez dodanie funkcjonalności o nazwie *Compact Profiles* (kompaktowe profile), które umożliwiły uporządkowanie kodu i zmniejszenie rozmiaru środowiska wykonawczego.

Moduł `java.base` reprezentuje minimum, jakie jest potrzebne aplikacji Java do rozpoczęcia działania. Zawiera on podstawowe pakiety, takie jak:

```
java.io
java.lang
```

```
java.math
java.net
java.nio
java.security
java.text
java.time
java.util
javax.crypto
javax.net
javax.security
```

Oprócz tego w module tym znajdują się podpakiety i nieeksportowane pakiety implementacyjne, takie jak `sun.text.resources`. Niektóre z różnic w kompilacji między Javą 8 i modułową Javą można zaobserwować w poniższym prostym programie, który rozszerza wewnętrzną klasę publiczną znajdującą się w pakiecie `java.base`:

```
import java.util.Arrays;
import sun.text.resources.FormatData;

public final class FormatStealer extends FormatData {
    public static void main(String[] args) {
        FormatStealer fs = new FormatStealer();
        fs.run();
    }

    private void run() {
        String[] s = (String[]) handleGetObject("japanese.Eras");
        System.out.println(Arrays.toString(s));

        Object[][] contents = getContents();
        Object[] eraData = contents[14];
        Object[] eras = (Object[])eraData[1];
        System.out.println(Arrays.toString(eras));
    }
}
```

Próba skompilowania tego kodu w Javie 11 spowoduje pojawienie się następującego komunikatu o błędzie:

```
$ javac javanut8/ch12/FormatStealer.java
javanut8/ch12/FormatStealer.java:4:
        error: package sun.text.resources is not visible
import sun.text.resources.FormatData;
         ^
  (package sun.text.resources is declared in module
        java.base, which does not export it to the unnamed module)
javanut8/ch12/FormatStealer.java:14: error: cannot find symbol
        String[] s = (String[]) handleGetObject("japanese.Eras");
                                ^
  symbol: method handleGetObject(String)
  location: class FormatStealer
javanut8/ch12/FormatStealer.java:17: error: cannot find symbol
        Object[][] contents = getContents();
                              ^
  symbol: method getContents()
  location: class FormatStealer
3 errors
```

W modułowej Javie nawet klasy publiczne są niedostępne, dopóki nie zostaną wprost wyeksportowane przez moduł, w którym są zdefiniowane. Za pomocą przełącznika `--add-exports` możemy tymczasowo zmusić kompilator do używania pakietu wewnętrznego (zasadniczo ponownie narzucając stare zasady dostępu):

```
$ javac --add-exports java.base/sun.text.resources=ALL-UNNAMED \
        javanut8/ch12/FormatStealer.java
javanut8/ch12/FormatStealer.java:5:
        warning: FormatData is internal proprietary API and may be
        removed in a future release
import sun.text.resources.FormatData;
                          ^
javanut8/ch12/FormatStealer.java:7:
        warning: FormatData is internal proprietary API and may be
        removed in a future release
public final class FormatStealer extends FormatData {
                                         ^
2 warnings
```

Musimy określić, że eksport jest przydzielany *modułowi bez nazwy*, ponieważ naszą klasę kompilujemy samodzielnie, a nie jako część modułu. Kompilator ostrzega nas, że używamy wewnętrznego API oraz że w przyszłej wersji Javy ten kod może nie zadziałać. Kiedy go skompilujemy i uruchomimy na platformie Java 11, zwróci listę epok w historii Japonii:

```
[, Meiji, Taisho, Showa, Heisei, Reiwa]
[, Meiji, Taisho, Showa, Heisei, Reiwa]
```

Natomiast w Javie 17 otrzymamy inny wynik:

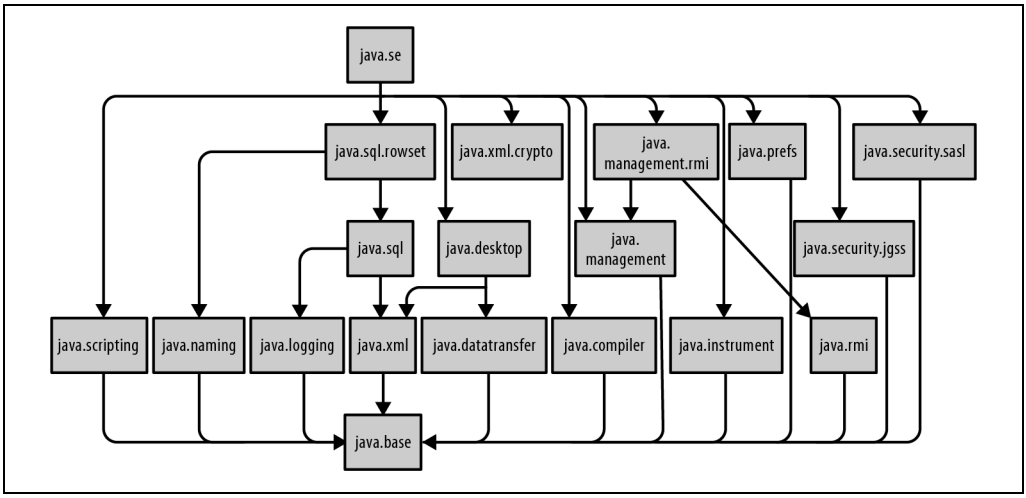
```
$ java javanut8.ch12.FormatStealer

Error: LinkageError occurred while loading main class
         javanut8.ch12.FormatStealer
java.lang.IllegalAccessError: superclass access check failed:
class javanut8.ch12.FormatStealer (in unnamed module @0x647c3190)
         cannot access class sun.text.resources.FormatData (in module
         java.base) because module java.base does not export
         sun.text.resources to unnamed module @0x647c3190
```

To wynika z faktu, że Java 17 wymusza dodatkowe kontrole w ramach zacieśniania hermetyzacji wewnętrznych mechanizmów. Aby program zadziałał, należy dodać flagę środowiska wykonawczego `--add-exports`:

```
$ java --add-exports java.base/sun.text.resources=ALL-UNNAMED \
        javanut8.ch12.FormatStealer
[, Meiji, Taisho, Showa, Heisei, Reiwa]
[, Meiji, Taisho, Showa, Heisei, Reiwa]
```

Choć `java.base` jest absolutnym minimum środowiska wykonawczego, jakiego aplikacja potrzebuje do działania, w czasie kompilacji staramy się, aby widoczna platforma była jak najbliższa starej (Java 8).

To znaczy, że używamy znacznie większego zestawu modułów zebranych we wspólnym module *zbiorczym* o nazwie `java.se`. Na rysunku 12.1 jest przedstawiony jego graf zależności.



Rysunek 12.1. Graf zależności modułu java.se

Te zależności obejmują prawie wszystkie klasy i pakiety, których potrzebuje większość programistów Javy.

Jednym ważnym wyjątkiem jest to, że pakiety Java 8 definiujące API CORBA i Java EE (obecnie zwane Jakarta EE) zostały usunięte i nie wchodzą w skład `java.se`. To znaczy, że żaden projekt zależny od tych API domyślnie nie przejdzie kompilacji od Javy 11 i konieczne będzie skonfigurowanie kompilacji w taki sposób, aby włączyć wprost zależność od zewnętrznych bibliotek dostarczających te API.

Oprócz tych zmian w zakresie widoczności kompilacji, związanej z modularyzacją JDK, system modułów ma na celu także umożliwienie programistom modularyzacji własnego kodu.

Pisanie własnych modułów

W tym podrozdziale opisujemy podstawowe kwestie, które trzeba znać, aby zacząć pisać modułowe aplikacje w Javie.

Podstawowa składnia modułów

Kluczem do modularyzacji jest nowy plik o nazwie *module-info.java*, który zawiera opis modułu. Nazywa się on **deskryptorem modułu** (ang. *module descriptor*).

Prawidłowa struktura modułu do kompilacji w systemie plików jest następująca:

- Pod głównym katalogiem źródłowym projektu (*src*) musi się znajdować katalog o takiej samej nazwie, jak nazwa modułu (*moduledir*).
- W katalogu *moduledir* znajduje się plik *module-info.java*, który jest na tym samym poziomie, od którego zaczynają pakiety.

Informacje na temat modułu są kompilowane do postaci binarnej, *module-info.class*, zawierającej metadane, które zostaną użyte, kiedy modułowe środowisko wykonawcze spróbuje połączyć i uruchomić aplikację. Spójrz na przykładową zawartość prostego pliku *module-info.java*:

```
module httpchecker {
    requires java.net.http;

    exports httpchecker.main;
}
```

Ten kod zawiera nowe elementy składni: `module`, `exports` i `requires` — ale to nie są w pełni słowa kluczowe w przyjętym znaczeniu. W specyfikacji Java Language Specification SE 11 napisano:

> Kolejnych dziesięć sekwencji znaków to ograniczone słowa kluczowe: `open`, `module`, `requires`, `transitive`, `exports`, `opens`, `to`, `uses`, `provides` oraz `with`. Sekwencje te są tokenizowane jako słowa kluczowe wyłącznie wtedy, gdy występują jako symbole terminalne w produkcjach `ModuleDeclaration` i `ModuleDirective`.

To znaczy, że te ograniczone słowa kluczowe mogą występować tylko w metadanych modułów i są kompilowane do postaci binarnej przez `javac`. Znaczenie najważniejszych ograniczonych słów kluczowych jest następujące:

`module`
: Rozpoczyna deklarację metadanych modułu.

`requires`
: Określa moduł, od którego zależny jest ten moduł.

`exports`
: Deklaruje, które pakiety są wyeksportowane jako API.

Pozostałe ograniczone słowa kluczowe (odnoszące się do modułów) zostaną opisane w dalszej części tego rozdziału.

Koncepcja *ograniczonego słowa kluczowego* została znacznie rozszerzona w Javie 17, w wyniku czego opis jest znacznie dłuższy i bardziej niejasny. Tutaj używamy starszej specyfikacji, ponieważ odnosi się konkretnie do systemu modułów i lepiej spełnia nasze wymagania.

W naszym przykładzie oznacza to, że deklarujemy moduł `httpchecker` potrzebujący modułu `java.net.http`, który został ustandaryzowany w Javie 11 (jak również ma niejawną zależność od `java.base`). Nasz moduł eksportuje jeden pakiet, `httpchecker.main`, który jest w nim jedynym pakietem i będzie dostępny w innych modułach w czasie kompilacji.

Budowa prostej aplikacji modułowej

W ramach przykładu zbudujemy proste narzędzie sprawdzające, czy witryny internetowe używają protokołu HTTP/2. Do tego celu wykorzystamy API poznany w rozdziale 10.:

```
import static java.net.http.HttpResponse.BodyHandlers.ofString;

public final class HTTP2Checker {
    public static void main(String[] args) throws Exception {
        if (args.length == 0) {
            System.err.println("Podaj adresy URL do sprawdzenia.");
        }
        for (final var location : args) {
            var client = HttpClient.newBuilder().build();
            var uri = new URI(location);
            var req = HttpRequest.newBuilder(uri).build();

            var response = client.send(req,
                    ofString(Charset.defaultCharset()));
            System.out.println(location +": "+ response.version());
        }
    }
}
```

Ten program wykorzystuje dwa moduły — `java.net.http` i wszechobecny `java.base`. Plik modułu dla tej aplikacji jest bardzo prosty:

```
module http2checker {
    requires java.net.http;
    exports httpchecker.main;
}
```

Przy założeniu standardowego układu modułu, ten kod można skompilować następująco:

```
$ javac -d out/httpchecker \
        httpchecker/httpchecker/main/HTTP2Checker.java \
        httpchecker/module-info.java
```

Spowoduje to utworzenie skompilowanego modułu w katalogu *out/*. Do użytku należy go spakować w plik JAR:

```
$ jar --create --file httpchecker.jar \
       --main-class httpchecker.main.HTTP2Checker \
       -C out/httpchecker .
```

Przełącznik `--create` nakazuje narzędziu `jar` utworzenie nowego pliku JAR zawierającego klasy znajdujące się w katalogu. Kropka na samym końcu polecenia jest obowiązkowa i oznacza, że wszystkie pliki klas (względne wobec ścieżki podanej przez `-C`) mają zostać zapakowane do pliku JAR.

Za pomocą przełącznika `--main-class` ustawiliśmy **punkt wejścia** modułu — czyli określiliśmy klasę, która ma zostać wykonana, kiedy użyjemy modułu jako aplikacji. Zobaczmy to w akcji:

```
$ java -jar httpchecker.jar http://www.google.com
http://www.google.com: HTTP_1_1
$ java -jar httpchecker.jar https://www.google.com
https://www.google.com: HTTP_2
```

To pokazuje, że w czasie pisania tej książki witryna Google używała protokołu HTTP/2 do serwowania swojej strony głównej przez HTTPS, ale nadal używa HTTP/1.1 dla starej nieszyfrowanej usługi HTTP.

Wiesz już, jak skompilować i uruchomić prostą aplikację modułową, więc teraz poznasz kilka innych podstawowych funkcji modułowości, które są potrzebne do budowy i uruchamiania pełnoskalowych aplikacji.

Ścieżka modułów

Programiści Javy znają pojęcie **ścieżki klas**. Jednak podczas pracy z modułowymi aplikacjami Javy zamiast niej używa się **ścieżki modułów**. Jest to nowa koncepcja, która zastępuje ścieżkę klas wszędzie, gdzie to możliwe.

Moduły zawierają metadane na temat ich eksportów i zależności — nie są tylko długimi listami typów. To znaczy, że graf zależności modułu można łatwo zbudować, a wyszukiwanie modułów można skutecznie przeprowadzać.

Kod, który nie jest jeszcze zmodularyzowany, nadal jest umieszczany na ścieżce klas. Jest on ładowany do specjalnego **modułu bez nazwy**, który może odczytywać wszystkie inne moduły dostępne w `java.se`. Użycie modułu bez nazwy odbywa się automatycznie, kiedy klasy zostają umieszczone na ścieżce klas.

Zapewnia to ścieżkę migracji umożliwiającą zaadaptowanie modułowego środowiska wykonawczego Javy bez konieczności migracji do w pełni modułowej ścieżki aplikacji. Ma to jednak dwie poważne wady: żadna z zalet modułów nie będzie dostępna, dopóki aplikacja nie zostanie w pełni przeniesiona, oraz konieczne jest ręczne utrzymywanie samodzielności ścieżki klas aż do zakończenia modularyzacji.

Moduły automatyczne

Jednym z ograniczeń systemu modułów jest to, że nie możemy odwoływać się do plików JAR na ścieżce klas z modułów bez nazwy. Jest to rodzaj zabezpieczenia — projektanci systemu modułów chcieli, aby graf zależności modułów wykorzystywał pełne metadane oraz mógł polegać na ich kompletności.

Czasami jednak zdarza się, że kod modułowy musi odwoływać się do pakietów, które jeszcze nie zostały zmodularyzowane. Rozwiązaniem w takim przypadku jest umieszczenie niezmodyfikowanego pliku JAR bezpośrednio na ścieżce modułów (i usunięcie go ze ścieżki klas). Plik JAR umieszczony na ścieżce modułów w taki sposób staje się **modułem automatycznym**.

Ma on następujące cechy:

- Nazwa modułu utworzona jest z nazwy pliku JAR (lub odczytana z pliku *MANIFEST.MF*).
- Eksportuje każdy pakiet.
- Wymaga wszystkich pozostałych modułów (włącznie z modułem bez nazwy).

Jest to kolejna cecha, która ma ułatwić migrację, ale mimo to moduły automatyczne stanowią pewne rozluźnienie zasad bezpieczeństwa.

Moduły otwarte

Jak napisaliśmy, proste oznaczenie metody jako publicznej nie stanowi już gwarancji, że dany element będzie wszędzie dostępny. Teraz dostępność zależy także od tego, czy pakiet zawierający ten element jest eksportowany przez moduł, który go zawiera. Inną ważną kwestią związaną z projektem modułów jest dostęp do klas za pomocą technik refleksji.

Refleksja jest tak szerokim i ogólnym mechanizmem, że na pierwszy rzut oka trudno powiedzieć, jak można ją pogodzić z celami silnej hermetyzacji JPMS. Co gorsza, na refleksji bazuje tak wiele najważniejszych bibliotek i frameworków ekosystemu Javy (np. testy jednostkowe, wstrzykiwanie zależności i wiele innych), że brak rozwiązania dla niej uniemożliwiłby zaadaptowanie modułów w jakiejkolwiek prawdziwej aplikacji.

Rozwiązanie tych problemów jest dwojakie. Po pierwsze, moduł może deklarować się jako otwarty w następujący sposób:

```
open module jin8 {
    exports jin8.api;
}
```

Efekt tej deklaracji jest taki, że:

- Wszystkie pakiety w module są dostępne przez refleksję.
- W czasie kompilacji *nie* ma dostępu dla niewyeksportowanych pakietów.

To znaczy, że konfiguracja w czasie kompilacji zachowuje się jak standardowy moduł. Ogólnym celem jest zapewnienie prostej zgodności z istniejącymi frameworkami i kodem oraz ułatwienie migracji. Dzięki otwartemu modułowi zostaje przywrócona możliwość refleksyjnego dostępu do kodu. Ponadto zostaje zachowana sztuczka z metodą `setAccessible()`, która umożliwia dostęp do prywatnych i innych metod.

Bardziej szczegółowa kontrola nad dostępem refleksyjnym jest także możliwa przez ograniczone słowo kluczowe `opens`. Nie tworzy ono modułu otwartego, tylko wybiórczo otwiera określone pakiety do dostępu refleksyjnego przez ich bezpośrednią deklarację:

```
module ojin8 {
    exports ojin8.api;
    opens ojin8.domain;
}
```

Ta technika może być przydatna, kiedy na przykład programista dostarczy model domeny do użytku przez obsługujący moduły system **mapowania obiektowo-relacyjnego** (ang. *object-relational mapping* — ORM).

Można nawet pójść dalej i za pomocą słowa kluczowego `restricted` ograniczyć dostęp refleksyjny do konkretnych pakietów klienta. Tam, gdzie to możliwe, może to być dobra zasada projektowa, ale oczywiście taka technika nie zadziała dobrze z frameworkiem ogólnego przeznaczenia, takim jak ORM.

W taki sam sposób można ograniczyć eksport tylko do wybranych pakietów zewnętrznych. Tę możliwość jednak dodano głównie po to, aby wspomóc modularyzację JDK, więc ma ona ograniczone zastosowanie do modułów użytkownika.

Ponadto pakiet można zarówno wyeksportować, jak i otworzyć, ale nie jest to zalecane, ponieważ w trakcie migracji dostęp do pakietu powinien być możliwy na etapie kompilacji lub refleksyjnie, ale nie na oba te sposoby.

W przypadkach, kiedy potrzebny jest dostęp refleksyjny do pakietu obecnie znajdującego się w module, platforma udostępnia pewne przełączniki, które pełnią funkcję opatrunków na okres przejściowy.

W szczególności za pomocą opcji Javy `--add-opens module/package=ALL-UNNAMED` można otworzyć wybrany pakiet modułu w celu uzyskania dostępu refleksyjnego do całego kodu ze ścieżki klas, przesłaniając zachowanie systemu modułów. W przypadku kodu, który już jest modułowy, można to wykorzystać także w celu udzielenia refleksyjnego dostępu do wybranego modułu.

Podczas migracji do modułowej Javy kod, który refleksyjnie uzyskuje dostęp do kodu wewnętrznego innego modułu, powinien być wstępnie uruchamiany z użyciem tego przełącznika, aż znajdzie się inne rozwiązanie.

Z tym problemem dotyczącym dostępu refleksyjnego (i jego specjalnym przypadkiem) wiąże się także problem szeroko rozpowszechnionego użycia wewnętrznych API platformy przez frameworki. Najczęściej opisuje się to jako „problem braku bezpieczeństwa" — wrócimy jeszcze do tego pod koniec rozdziału.

Dostarczanie usług

System modułów zawiera mechanizm usług, który pozwala ograniczyć inny problem z zaawansowaną formą hermetyzacji. Jego proste objaśnienie reprezentuje poniższy prosty fragment kodu:

```
import com.example.Service;
Service s = new ServiceImpl();
```

Nawet jeśli `Service` znajduje się w eksportowanym pakiecie API, ten wiersz kodu i tak nie przejdzie kompilacji, jeśli jednocześnie nie zostanie wyeksportowany także pakiet zawierający `ServiceImpl`. Potrzebujemy więc mechanizmu umożliwiającego precyzyjny dostęp do klas implementujących klasy usługowe bez konieczności importowania całego pakietu. Na przykład, moglibyśmy napisać coś takiego:

```
module jin8 {
    exports jin8.api;
    requires othermodule.services;

    provides services.Service with jin8.services.ServiceImpl;
}
```

Teraz klasa `ServiceImpl` jest dostępna w czasie kompilacji jako implementacja interfejsu `Service`. Pamiętaj, że aby to zadziałało, pakiet `services` musi znajdować się w innym module, który jest wymagany przez obecny moduł.

Pliki JAR o kilku wersjach

Aby wyjaśnić, jaki problem rozwiązują pliki JAR o wielu wersjach, przeanalizujemy prosty przykład: znajdowanie identyfikatora procesu (PID) aktualnie wykonywanego procesu (tzn. JVM wykonującej nasz kod).

Nie wykorzystujemy przykładu HTTP/2 z wcześniejszego podrozdziału, ponieważ Java 8 nie ma API HTTP/2 API — musielibyśmy wykonać ogromną ilość pracy (w zasadzie musielibyśmy stworzyć pełne przeniesienie!), aby dostarczyć równoważną funkcjonalność w Javie 8.

Może się wydawać, że to proste zadanie, ale w Javie 8 wymaga to zaskakująco dużej ilości szablonowego kodu:

```
public class GetPID {
    public static long getPid() {
        // To dość niezdarne wywołanie używa JMX w celu zwrócenia nazwy, która
        // reprezentuje obecnie działającą JVM. Ta nazwa ma format
        // <pid>@<nazwahosta> tylko w maszynach wirtualnych OpenJDK i Oracle.
        // Nie ma gwarantowanego przenośnego rozwiązania w Javie 8.
        final String jvmName =
            ManagementFactory.getRuntimeMXBean().getName();
        final int index = jvmName.indexOf('@');
        if (index < 1)
            return -1;

        try {
            return Long.parseLong(jvmName.substring(0, index));
        } catch (NumberFormatException nfe) {
            return -1;
        }
    }
}
```

Jak widać, ten kod nie jest nawet blisko prostoty, jaką lubimy. Co gorsza, nie jest nawet obsługiwany w standardowy sposób we wszystkich implementacjach Javy 8. Na szczęście od Javy 11 możemy używać nowego API `ProcessHandle`, np.:

```
public class GetPID {
    public static long getPid() {
        // Użycie nowego API Process Javy 9...
        ProcessHandle processHandle = ProcessHandle.current();
        return processHandle.getPid();
    }
}
```

Teraz wykorzystaliśmy standardowy API, ale prowadzi nas to do zasadniczego problemu: jak programista może napisać kod, aby mieć gwarancję, że będzie on działał na wszystkich nowych wersjach Javy?

Chodzi nam o to, aby poprawnie zbudować i uruchomić projekt na wielu wersjach Javy. Chcemy wykorzystać klasy bibliotek, które są dostępne tylko w nowszych wersjach, ale program ma działać na starszej wersji dzięki użyciu „podkładek" kodu. Wynikiem końcowym ma być jeden plik JAR i nasz projekt nie musi się przełączać na format wielomodułowy — w istocie plik JAR musi działać jako moduł automatyczny.

Przyjrzymy się przykładowemu projektowi, który musi działać poprawnie zarówno w Javie 8, jak i Javie 11 oraz nowszych wersjach. Główna baza kodu jest zbudowana na Javie 8, a część dotycząca Javy 11 musi się opierać na Javie 11. Ta część programu musi być odizolowana od głównej bazy kodu, aby uniknąć błędów kompilacji, ale może być zależna od artefaktów z kompilacji Javy 8.

Aby konfiguracja kompilacji była prosta, do kontroli tej funkcji używa się wpisu w pliku *MANIFEST.MF* w pliku JAR:

```
Multi-Release: True
```

Kod wariantu (tzn. nowszej wersji) jest przechowywany w specjalnym pliku w katalogu *META-INF*. W naszym przypadku jest to *META-INF/versions/11*.

Dla środowiska wykonawczego Javy, które implementuje tę funkcję, wszystkie klasy w katalogu określonej wersji przesłaniają wersje z katalogu głównego treści. Natomiast Java 8 i wcześniejsze wersje ignorują zarówno wpis w manifeście, jak i katalog *versions/* i znajdują tylko klasy znajdujące się w katalogu głównym treści.

Konwersja na plik JAR o wielu wersjach

Aby rozpocząć wdrażanie programu jako pliku o wielu wersjach, należy postępować według poniższego schematu:

1. Izolacja kodu, który jest specyficzny dla wersji JDK.
2. W miarę możliwości umieszczenie tego kodu w pakiecie lub grupie pakietów.
3. Kompilacja projektu w wersji 8.
4. Utworzenie nowego, osobnego projektu dla klas uzupełniających.
5. Konfiguracja jednej zależności dla nowego projektu (artefakt wersji 8).

W Gradle można także wykorzystać pojęcie **zestawu źródła** (ang. *source set*) i skompilować wersję 11 kodu przy użyciu innego (nowszego) kompilatora. Następnie można ją wbudować w plik JAR za pomocą kodu podobnego do poniższego:

```
jar {
  into('META-INF/versions/11') {
    from sourceSets.java11.output
  }

  manifest.attributes(
    'Multi-Release': 'true'
  )
}
```

W Maven obecnie najprostszym rozwiązaniem jest użycie wtyczki Maven Dependency i dodanie klas modułowych do ogólnego pliku JAR w ramach osobnej fazy `generate-resources`.

Migracja do modułów

Wielu programistów Javy mierzy się z pytaniem, czy i kiedy przerobić swoje aplikacje na modułowe.

Moduły powinny być domyślnie wykorzystywane w nowych aplikacjach, zwłaszcza w tych o strukturze mikrousługowej.

Wiele aplikacji w ogóle nie wymaga migrowania. Jednak modularyzacja istniejących baz kodu może być opłacalna, ponieważ lepsza hermetyzacja i ogólna architektura zwracają się w dłuższej perspektywie — nowym programistom łatwiej jest się wdrożyć, a projekt o przejrzystej strukturze jest prostszy do zrozumienia i utrzymania.

Zastanawiając się nad możliwością migracji istniejącego programu (w szczególności o budowie monolitycznej), można posłużyć się poniższymi wskazówkami:

1. Najpierw zaktualizuj środowisko wykonawcze aplikacji do Javy 17 (początkowo działając ze ścieżki klas).
2. Zidentyfikuj wszystkie zależności aplikacji, które zostały zmodularyzowane, i przeprowadź ich migrację do modułów.
3. Zachowaj wszystkie niezmodularyzowane zależności jako moduły automatyczne.
4. Wprowadź jeden *monolityczny moduł* całego kodu aplikacji.

W tym momencie minimalnie zmodularyzowana aplikacja powinna być już gotowa do użycia. Ten moduł zazwyczaj będzie otwarty na tym etapie procesu. Następnym krokiem jest refaktoryzacja architekturalna. W tej fazie aplikacje można dzielić na indywidualne moduły według potrzeby.

Kiedy już kod aplikacji działa w modułach, dobrym pomysłem może być ograniczenie refleksyjnego dostępu przez `opens`. Proces pozbywania się niechcianych możliwości dostępu można zacząć od jego ograniczenia do wybranych modułów (np. ORM lub modułów wstrzykiwania zależności).

Użytkownicy Maven powinni pamiętać, że to nie jest system modułowy, ale ma zależności, które (w odróżnieniu od zależności JPMS) są wersjonowane. Wciąż trwają prace nad pełną integracją JPMS z narzędziami Maven (obecnie jest jeszcze wiele wtyczek, które dopiero to czeka). Można jednak powiedzieć, że powoli pojawiają się pewne ogólne wytyczne dotyczące modułowych projektów Maven:

- Staraj się tworzyć po jednym module na POM Maven.
- Nie modularyzuj projektu Maven, jeśli nie jesteś jeszcze gotowy (lub nie musisz tego natychmiast robić).
- Pamiętaj, że od Javy 11 nie ma konieczności budowania na bazie łańcucha narzędzi Javy 11+.

Ostatni punkt oznacza, że jedną ze ścieżek migracji projektów Maven jest rozpoczęcie budowy projektu jako Java 8 i sprawdzenie, czy artefakty Maven są wdrażane bez problemu (jako moduły automatyczne) w środowisku Java 11 (lub 17). Dopiero po pomyślnej realizacji pierwszego kroku należy przejść do pełnej modularyzacji.

W procesie modularyzacji można wykorzystać pewne przydatne narzędzia. W Javie 8 i nowszych wersjach dostępny jest program `jdeps` (zob. rozdział 13) — pozwala on sprawdzić, których pakietów i modułów potrzebuje nasz kod. To bardzo ułatwia migrację z Javy 8 i dlatego zaleca się używanie `jdeps` podczas przemodelowywania architektury.

Obrazy środowiska wykonawczego

JPMS stworzono głównie ze względu na to, że aplikacje mogą potrzebować tylko niektórych klas obecnych w tradycyjnym monolitycznym środowisku Javy 8 i wystarcza im do działania skromniejszy zbiór modułów. Takie aplikacje szybciej się uruchamiają i zajmują mniej pamięci. W związku z tym nasuwa się pytanie: skoro nie wszystkie klasy są potrzebne, to czy nie lepiej byłoby dostarczać aplikację ze zredukowanym niestandardowym obrazem środowiska wykonawczego, który zawiera tylko to, co niezbędne?

Aby to zademonstrować, zapakujemy program sprawdzający HTTP/2 jako samodzielne narzędzie z własnym niestandardowym środowiskiem wykonawczym. Do tego celu możemy użyć narzędzia `jlink` (które jest obecne w platformie od Javy 9):

```
$ jlink --module-path httpchecker.jar:$JAVA_HOME/jmods \
      --add-modules httpchecker,jdk.crypto.ec \
      --launcher http2chk=httpchecker \
      --output http2chk-image
```

Przyjęliśmy założenie, że plik JAR *httpchecker.jar* zawiera klasę główną (punkt wejścia). Wynikiem jest katalog wyjściowy, *http2chk-image*, o rozmiarze około 39 MB, a więc znacznie mniejszym niż pełny obraz. Ponadto, jako że narzędzie wykorzystuje nowy moduł HTTP, wymagane są biblioteki zabezpieczeń, kryptografii itd. potrzebne do połączeń przez HTTPS.

W katalogu niestandardowego obrazu możemy uruchomić narzędzie `http2chk`, aby się przekonać, że działa ono nawet wtedy, gdy maszyna nie ma wymaganej wersji programu `java`:

```
$ java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
$ ./bin/http2chk https://www.google.com
https://www.google.com: HTTP_2
```

Wdrażanie niestandardowych obrazów środowiska wykonawczego to wciąż nowość, ale ma wielki potencjał w zakresie zmniejszenia ilości kodu i zachowania konkurencyjności Javy w świecie mikrousług. W przyszłości narzędzie `jlink` może nawet zostać połączone z nowymi metodami kompilacji, w tym z kompilatorem AOT (ang. *ahead-of-time*).

Problemy z modułami

Choć system modułów jest flagowym dodatkiem do Javy 9, któremu poświęcono bardzo dużo pracy, nie jest on pozbawiony wad. To było raczej nieuniknione — moduły fundamentalnie zmieniają architekturę i sposób dostarczania aplikacji Java. Uniknięcie wszystkich problemów podczas przerabiania dużego i dojrzałego ekosystemu, jakim jest Java, byłoby praktycznie niemożliwe.

Unsafe i pokrewne problemy

Klasa `sun.misc.Unsafe` jest powszechnie używana i popularna wśród twórców frameworków i innych implementatorów w świecie Javy. Jest to jednak wewnętrzna klasa implementacyjna, która nie należy do standardowego API platformy Javy (jak wyraźnie informuje nazwa pakietu). Ponadto jej nazwa silnie wskazuje, że nie jest przeznaczona do użytku przez aplikacje Javy.

Klasa `Unsafe` jest niewspieranym wewnętrznym API, które może zostać wycofane lub zmodyfikowane w którejkolwiek kolejnej wersji Javy bez względu na używające jej aplikacje. Każdy korzystający z niej kod jest wprost uzależniony od maszyny wirtualnej HotSpot i jest potencjalnie niestandardowy, przez co może nie działać w innych implementacjach.

Choć klasa `Unsafe` nie należy do oficjalnej specyfikacji Java SE, stała się *de facto* standardem i w ten czy inny sposób kluczowym elementem implementacji praktycznie każdego większego frameworku. W kolejnych wersjach została składem niestandardowych, ale niezbędnych elementów. Jest to prawdziwe wysypisko różności o różnym poziomie bezpieczeństwa. Przykładowe zastosowania klasy `Unsafe`:

- szybka serializacja i deserializacja;
- bezpieczny wątkowo dostęp do pamięci natywnej o rozmiarze 64 bitów (np. poza stertą);
- atomowe operacje na pamięci (np. porównywanie i zamiana);
- szybki dostęp do pól/pamięci;
- zamiana JNI na rozwiązanie dla wielu systemów operacyjnych;
- dostęp do elementów tablicy przy użyciu semantyki ulotnej (zobacz rozdział 6.).

Podstawowym problemem jest to, że wielu frameworków i bibliotek nie dało się przenieść do modułowego JDK bez zamiany na pewne elementy z klasy `Unsafe`. To z kolei ma wpływ na wszystko, co używa jakichkolwiek bibliotek z szerokiego zakresu frameworków — praktycznie na każdą aplikację w ekosystemie Javy.

Aby naprawić ten problem, firma Oracle stworzyła nowe obsługiwane API dla niektórych z potrzebnych elementów funkcjonalności i przeniosła API, których nie dało się zhermetyzować w porę, do modułu `jdk.unsupported`. To wyraźnie pokazuje, że API jest niewspierany i programiści wiedzą, że używają go na własne ryzyko. Daje to klasie `Unsafe` odrobinę czasu (bardzo ograniczonego) przy jednoczesnej zachęcie dla twórców bibliotek i frameworków, aby przechodzili na nowe API.

Przykładem API zamiennego jest VarHandles. Rozszerza on koncepcję **uchwytów do metod** (z rozdziału 11.) i dodaje nowe elementy funkcjonalności, takie jak tryby barier współbieżności dla Javy 11. Wszystko to wraz z pewnymi poprawkami w JMM ma na celu umożliwić stworzenie standardowego API dostępu do nowych niskopoziomowych funkcji przetwarzania bez dawania programistom pełnego dostępu do niebezpiecznych składników, jakie są dostępne w klasie `Unsafe`.

Więcej informacji na temat klasy `Unsafe` i pokrewnych technik specyficznych dla platformy można znaleźć w książce *Optimizing Java* Benjamina J. Evansa, Jamesa Gougha i Chrisa Newlanda (wyd. O'Reilly).

Brak wersjonowania

W Javie 17 standard JPMS nie obejmuje wersjonowania zależności.

Projektanci celowo podjęli tę decyzję, aby zmniejszyć poziom złożoności dostarczanego systemu. Nie wyklucza to pojawienia się modułów z wersjonowanymi zależnościami w przyszłości.

W obecnej sytuacji do wersjonowania zależności modułu potrzebne są narzędzia zewnętrzne. W przypadku Mavena jest to Project Object Model (POM). Zaletą tego podejścia jest to, że pobieranie wersji i zarządzanie nimi odbywa się w lokalnym repozytorium narzędzia budującego.

W każdym razie najważniejsze jest to, że informacja o wersji zależności musi być przechowywana poza modułem i nie stanowi części artefaktu JAR.

Nie da się od tego uciec — to dość paskudne, ale przynajmniej można powiedzieć, że ta sytuacja nie jest gorsza od tej, kiedy zależności były dedukowane ze ścieżki klas.

Powolna adaptacja

Od Javy 9 model wydawania wersji Javy uległ radykalnej zmianie. W Javie 8 i 9 obowiązywał model „kluczowego składnika wersji", w którym jedna fundamentalna cecha — taka jak lambdy w Javie 8 lub moduły w Javie 9 — zasadniczo definiowała całą wersję, przez co data wydania była determinowana przez datę ukończenia prac nad danych kluczowym składnikiem.

Problem z tym modelem polega na tym, że może utrudniać pracę przez brak pewności co do tego, kiedy pojawi się nowa wersja. Drobna modyfikacja, która nie zdąży wejść do jednej wersji, może potem długo czekać, aż pojawi się kolejne wydanie. W efekcie od Javy 10 wprowadzono nowy model publikowania wersji, który ma *ściśle określone ramy czasowe*.

To oznacza, że:

- Wersje Javy są obecnie klasyfikowane jako wydania **elementów funkcjonalności**, które pojawiają się w regularnych sześciomiesięcznych odstępach czasu.
- Składniki zostają wprowadzone do platformy dopiero po całkowitym ukończeniu prac nad nimi.
- Główne repozytorium cały czas jest w stanie gotowym do wydania.

Te wydania są dobre tylko przez sześć miesięcy, po których tracą wsparcie. Niektóre wydania są oznaczane przez Oracle jako **LTS** (ang. *long-term support* — wsparcie długoterminowe). Obejmuje je wydłużony okres wsparcia w formie płatnej usługi firmy Oracle.

Początkowo wersje LTS miały pojawiać się co trzy lata, ale obecnie planowane jest skrócenie tego okresu do dwóch lat. To znaczy, że obecnie wersje LTS Oracle to 8 (dodano wstecznie), 11 i 17. Następna oczekiwana wersja to Java 21, która ma wyjść w październiku 2023 r.

Należy jednak pamiętać, że kompilacje OpenJDK są tworzone nie tylko przez Oracle, lecz także np. przez Amazon, Azul, Eclipse Adoptium, IBM, Microsoft, Red Hat i SAP. Ci dostawcy oferują różne możliwości aktualizowania JDK (włącznie z aktualizacją zabezpieczeń) bez opłat.

Ponadto niektórzy z wymienionych dostawców udostępniają nowe i wcześniejsze modele wsparcia.

Szczegółowy opis tego tematu znajduje się w przewodniku pt. *Java Is Still Free* („Java wciąż jest darmowa") (*https://oreil.ly/Cz61R*) przygotowanym przez społeczność Java Champions (*https://oreil.ly/NGIpB*), które jest niezależnym towarzystwem liderów ze środowiska Javy w branży programistycznej.

Choć generalnie społeczność skupiona wokół Javy pozytywnie przyjęła przyspieszony cykl wydawania nowych wersji, Java 9 i kolejne wersje są znacznie oporniej adaptowane niż wcześniejsze wydania tego języka. Może to wynikać z tego, że zespoły preferują dłuższe okresy wsparcia i dlatego nie przechodzą na kolejne wersje co sześć miesięcy. W praktyce tylko wersje LTS zyskują szeroką adaptację, która i tak jest powolna w porównaniu z błyskawicznym przyjęciem Javy 8.

Nie bez znaczenia jest też fakt, że przejście z Javy 8 na Javę 11 (lub 17) nie jest tak proste jak podmiana plików (w odróżnieniu od tego, jak było w przypadku wersji 7 i 8, a także w mniejszym zakresie w przypadku wersji 6 i 7). Podsystem modułów fundamentalnie zmienia wiele aspektów platformy Javy, nawet jeśli aplikacje przeznaczone dla użytkowników nie korzystają z modułów.

Cztery lata po pojawieniu się Javy 11 można chyba powiedzieć, że prześcignęła ona w końcu Javę 8, tzn. jest więcej programów działających na platformie Java 11 niż na platformie Java 8. Dopiero się okaże, jakie tempo adaptacji osiągnie Java 17 i jaki wpływ na środowisko wywrze Java 21 (przy założeniu, że wersja 21 rzeczywiście będzie LTS).

Podsumowanie

Moduły wprowadzone w Javie 9 mają rozwiązać kilka problemów na raz. Cele dotyczące skrócenia czasu rozruchu, zmniejszenia zużycia pamięci i obniżenia poziomu złożoności przez wyłączenie dostępu do mechanizmów wewnętrznych zostały osiągnięte. Na bardziej długoterminowe cele poprawy jakości architekturalnej aplikacji i mentalne przestawienie programistów na nowe metody kompilowania i wdrażania trzeba będzie jeszcze trochę poczekać.

Faktem jednak jest, że od Javy 17 niewiele projektów i zespołów w pełni przeszło na modułowość. Tego należy się spodziewać, ponieważ modułowość to długoterminowy projekt, który odpłaca się powoli oraz bazuje na sieciowych efektach w obrębie ekosystemu.

Twórcy nowych aplikacji zdecydowanie powinni rozważyć możliwość zastosowania podejścia modułowego od samego początku, jednak historia modułowości na platformie Java tak naprawdę dopiero się zaczyna.

Czy już korzystasz z najlepszych narzędzi Javy?

Programiści Javy mają do dyspozycji wiele przydatnych narzędzi i z każdą kolejną wersją języka mogą korzystać z coraz to lepszych możliwości. Powinni więc sukcesywnie zapoznawać się z tymi nowościami, jednak nie każdy ma czas na studiowanie dokumentacji. Nowoczesna Java wymaga od programisty nie tylko znajomości składni i interfejsów API, musi on dobrze opanować również zagadnienia współbieżności, obiektowości, a także pamięci i systemu typów.

Oto kolejne wydanie zwięzłego podręcznika dla programistów Javy, który ma ułatwić maksymalne wykorzystanie technologii tego języka w wersji 17. Treść została skrupulatnie przejrzana i uzupełniona o materiał dotyczący nowości w obiektowym modelu Javy. Pierwsza część książki obejmuje wprowadzenie do języka i do pracy na platformie Javy. Druga zawiera opis podstawowych pojęć i interfejsów API, których znajomość jest niezbędna każdemu programiście Javy. Mimo niewielkiej objętości w podręczniku znalazły się liczne przykłady wykorzystania potencjału tego języka programowania, a także zastosowania najlepszych praktyk programistycznych w rzeczywistej pracy.

W książce między innymi:

- podstawy języka i biblioteka Javy 17
- model programowania zorientowanego obiektowo
- typy generyczne, wyliczenia, adnotacje i wyrażenia lambda
- techniki współbieżności i model pamięci
- najnowsze interfejsy API wejścia i wyjścia Javy
- narzędzia programistyczne pakietu OpenJDK

Benjamin Evans jest starszym inżynierem oprogramowania w Red Hat, otrzymał tytuł Java Champion. Jest jednym z założycieli jClarity.

Jason Clark jest głównym inżynierem i architektem w New Relic. Pracował z potokami przetwarzania danych JVM o petabajtowej skali.

David Flanagan jest programistą i autorem książek o JavaScripcie i Javie.

To pozycja obowiązkowa dla osób, które szukają przejrzyście podanej wiedzy, jak działa Java i jak się rozwijała w czasie.

Achyut Madhusudan, programista z Red Hat

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-0161-2

9 788328 901612

Cena: 89,00 zł