



Twoja przepustka do świata
Spring Framework!

J Sharma, Ashish Sarin

SPRING FRAMEWORK

Wprowadzenie do tworzenia aplikacji

Wydanie II

Helion 

Tytuł oryginału: Getting started with Spring Framework

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-0929-6

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

© Jyotsna Sharma, Ashish Sarin 2015.

Polish edition copyright © 2015 by Helion S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/spfrwp.zip>

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Wprowadzenie	15
Rozdział 1. Wprowadzenie do frameworku Spring	17
1.1. Wprowadzenie	17
1.2. Moduły frameworku Spring	17
1.3. Kontener IoC	18
1.4. Zalety stosowania frameworku Spring	21
Spójne podejście w zakresie zarządzania transakcjami lokalnymi i globalnymi	21
Deklaracyjne zarządzanie transakcjami	23
Zapewnienie bezpieczeństwa	24
JMX (Java Management Extensions)	24
JMS (Java Message Service)	26
Buforowanie	27
1.5. Prosta aplikacja Springa	28
Identyfikacja obiektów aplikacji i ich zależności	28
Utworzenie klas POJO odpowiadających zidentyfikowanym obiektom aplikacji	29
Utworzenie metadanych konfiguracyjnych	30
Utworzenie egzemplarza kontenera Springa	35
Uzyskanie dostępu do obiektów aplikacji z poziomu egzemplarza kontenera Springa	36
1.6. Frameworki zbudowane na bazie Springa	37
1.7. Podsumowanie	38
Rozdział 2. Podstawy frameworku Spring	39
2.1. Wprowadzenie	39
2.2. Reguła projektowa o nazwie „podejście oparte na programowaniu interfejsów”	39
Scenariusz: klasa zależna zawiera odwołanie do konkretnej klasy zależności	39
Scenariusz: klasa zależna zawiera odwołanie do interfejsu implementowanego przez zależność	40
Oferowana przez Spring obsługa podejścia opartego na programowaniu interfejsu	42
2.3. Różne podejścia podczas tworzenia komponentów Springa	44
Utworzenie komponentu za pomocą statycznej metody fabryki	44
Tworzenie komponentu za pomocą metody fabryki egzemplarza	46

2.4. Techniki wstrzykiwania zależności	47
Wstrzykiwanie zależności za pomocą metody typu setter	47
Wstrzykiwanie zależności za pomocą argumentów konstruktora	51
2.5. Zasięg komponentu	54
Singleton	55
Prototyp	63
Wybór właściwego zasięgu dla komponentu	64
2.6. Podsumowanie	64

Rozdział 3. Konfiguracja komponentów 65

3.1. Wprowadzenie	65
3.2. Dziedziczenie definicji komponentu	65
Przykład dziedziczenia definicji komponentu w aplikacji MyBank	65
Co jest dziedziczone?	68
3.3. Dopasowanie argumentu konstruktora	73
Przekazywanie wartości i odwołań do komponentów za pomocą elementu <constructor-arg>	73
Dopasowanie argumentu konstruktora na podstawie typu	74
Dopasowanie argumentu konstruktora na podstawie nazwy argumentu	79
3.4. Konfiguracja różnego typu właściwości komponentu i argumentów konstruktora	82
Wbudowane edytory właściwości	83
Określanie wartości dla różnych typów kolekcji	86
Definiowanie wartości dla tablic	92
Domyślne implementacje kolekcji dla elementów <list>, <set> i <map>	92
3.5. Wbudowane edytory właściwości	93
Edytor CustomCollectionEditor	93
Edytor CustomMapEditor	96
Edytor CustomDateEditor	96
3.6. Rejestracja edytorów właściwości w kontenerze Springa	97
Utworzenie implementacji PropertyEditorRegistrar	97
Konfiguracja klasy CustomEditorConfigurer	98
3.7. Zwięzłe definicje komponentów dzięki użyciu przestrzeni nazw p oraz c	99
Przestrzeń nazw p	99
Przestrzeń nazw c	100

3.8. Schemat util	103
Element <list>	104
Element <map>	105
Element <set>	107
Element <properties>	109
Element <constant>	110
Element <property-path>	111
3.9. Interfejs FactoryBean	113
Aplikacja MyBank — przechowywanie zdarzeń w bazie danych	113
Aplikacja MyBank — przykład użycia interfejsu FactoryBean	115
Uzyskanie dostępu do egzemplarza FactoryBean	118
3.10. Podsumowanie	120

Rozdział 4. Wstrzykiwanie zależności 121

4.1. Wprowadzenie	121
4.2. Komponenty wewnętrzne	121
4.3. Wyrażna kontrola kolejności inicjalizacji procesu za pomocą atrybutu depends-on	123
Aplikacja MyBank — zasugerowane zależności między komponentami	123
Problem niejawnej zależności	124
4.4. Zależności w postaci komponentów o zasięgu singleton i prototypu	129
Zależności komponentu o zasięgu singleton	129
Zależności komponentu o zasięgu prototypu	132
4.5. Pobieranie w komponencie singleton nowego egzemplarza komponentu o zasięgu prototypu	135
Interfejs ApplicationContextAware	136
Element <lookup-method>	138
Element <replaced-method>	142
4.6. Automatyczne wiązanie zależności	145
Wartość byType	146
Wartość constructor	147
Wartość byName	148
Wartości default i no	150
Uniedostępnienie komponentu dla funkcji automatycznego wiązania	150
Ograniczenia automatycznego wiązania	152
4.7. Podsumowanie	153

Rozdział 5. Dostosowanie komponentów i ich definicji	155
5.1. Wprowadzenie	155
5.2. Dostosowanie logiki inicjalizacji i usuwania komponentu	155
Wywołanie przez Spring metody czyszczącej wskazanej	
przez atrybut destroy-method	158
Metoda czyszcząca i komponent o zasięgu prototypu	160
Określenie domyślnych metod inicjalizacji i usuwania obiektu	
dla wszystkich komponentów	160
Interfejsy cyklu życiowego InitializingBean i DisposableBean	161
Adnotacje @PostConstruct i @PreDestroy wprowadzone w specyfikacji JSR 250	161
5.3. Użycie interfejsu BeanPostProcessor do pracy z nowo utworzonym	
egzemplarzem komponentu	163
Przykład użycia interfejsu BeanPostProcessor	
— weryfikacja egzemplarza komponentu	165
Przykład BeanPostProcessor — rozwiązywanie zależności komponentu	168
Zachowanie BeanPostProcessor w przypadku	
komponentów implementujących FactoryBeans	172
RequiredAnnotationBeanPostProcessor	174
DestructionAwareBeanPostProcessor	175
5.4. Modyfikacja definicji komponentu za pomocą BeanFactoryPostProcessor	176
Przykład użycia BeanFactoryPostProcessor	177
Komponent PropertySourcesPlaceholderConfigurer	182
Komponent PropertyOverrideConfigurer	187
5.5. Podsumowanie	190
Rozdział 6. Programowanie oparte na adnotacjach	191
6.1. Wprowadzenie	191
6.2. Identyfikacja komponentów Springa za pomocą adnotacji @Component	191
6.3. Adnotacja @Autowired, czyli automatyczne wiązanie zależności według ich typu	194
6.4. Adnotacja @Qualifier pozwalająca na automatyczne	
wiązanie zależności według ich nazwy	198
6.5. Adnotacje @Inject i @Named zdefiniowane przez specyfikację JSR 330	199
6.6. Zdefiniowana przez JSR 250 adnotacja @Resource	201
6.7. Adnotacje @Scope, @Lazy, @DependsOn i @Primary	202

6.8. Ułatwienie konfiguracji komponentu za pomocą adnotacji @Value	203
6.9. Weryfikacja obiektów za pomocą interfejsu Validator	207
6.10. Określanie ograniczeń za pomocą adnotacji JSR 303	211
Obsługa specyfikacji JSR 303 w Springu	212
6.11. Programowa konfiguracja komponentów Springa za pomocą adnotacji @Configuration i @Bean	216
6.12. Podsumowanie	219

Rozdział 7. Współpraca z bazą danych 221

7.1. Wprowadzenie	221
7.2. Wymagania aplikacji MyBank	221
7.3. Utworzenie aplikacji MyBank opartej na module JDBC	222
Konfiguracja źródła danych	223
Tworzenie obiektów DAO używających klas modułu JDBC	224
7.4. Utworzenie aplikacji MyBank z użyciem Hibernate	231
Konfiguracja SessionFactory	231
Utworzenie klas DAO używających do pracy z bazą danych API Hibernate	232
7.5. Zarządzanie transakcjami za pomocą Springa	233
Wymagania aplikacji MyBank w zakresie zarządzania transakcjami	233
Programowe zarządzanie transakcjami	234
Deklaracyjne zarządzanie transakcjami	238
Oferowana przez Spring obsługa JTA	241
7.6. Podsumowanie	243

Rozdział 8. Komunikaty, wiadomości e-mail, asynchroniczne wykonywanie metod i buforowanie w Springu 245

8.1. Wprowadzenie	245
8.2. Wymagania aplikacji MyBank	245
8.3. Wysyłanie komunikatów JMS	246
Konfiguracja brokera ActiveMQ do działania w trybie osadzonym	247
Konfiguracja JMS ConnectionFactory	247
Wysyłanie komunikatów JMS za pomocą klasy JmsTemplate	248
Wysyłanie komunikatów JMS w transakcji	249

Dynamiczni odbiorcy komunikatów JMS i konfiguracja klasy JmsTemplate	253
Klasa JmsTemplate i konwersja komunikatu	254
8.4. Odbieranie komunikatów JMS	255
Synchroniczne odbieranie komunikatów JMS za pomocą klasy JmsTemplate	255
Asynchroniczne odbieranie komunikatów JMS za pomocą kontenerów odbiorców komunikatów	255
8.5. Wysyłanie wiadomości e-mail	258
8.6. Harmonogram zadań i wykonywanie asynchroniczne	263
Interfejs TaskExecutor	263
Interfejs TaskScheduler	265
Adnotacje @Async i @Scheduled	267
8.7. Buforowanie	269
Konfiguracja interfejsu CacheManager	271
Adnotacje buforowania @Cacheable, @CacheEvict i @CachePut	271
8.8. Aplikacja MyBank	275
8.9. Podsumowanie	277
Rozdział 9. Programowanie zorientowane aspektowo	279
9.1. Wprowadzenie	279
9.2. Prosty przykład AOP	279
9.3. Oferowany przez Spring framework AOP	282
Tworzenie proxy	283
Atrybut expose-proxy	284
9.4. Wyrażenia punktu przecięcia	286
Adnotacja @Pointcut	286
Desygatory execution i args	287
Desygator bean	291
Desygatory oparte na adnotacjach	292
9.5. Typy rad	293
Rada before	293
Rada after returning	294
Rada after throwing	295
Rada after	296
Rada around	297

9.6. Spring AOP, czyli styl schematu XML	298
Konfiguracja aspektu AOP	299
Konfiguracja rady	300
Powiązanie wyrażenia punktu przecięcia z radą	301
9.7. Podsumowanie	302
Rozdział 10. Podstawy modułu Spring Web MVC	303
10.1. Wprowadzenie	303
10.2. Struktura katalogów przykładowych projektów	303
10.3. Poznajemy przykładową aplikację	304
Plik HelloWorldController.java, czyli klasa kontrolera aplikacji sieciowej	
Hello World	305
Plik helloworld.js, czyli strona JSP wyświetlająca komunikat	306
Plik myapp-config.xml, czyli plik XML kontekstu aplikacji sieciowej	307
Plik web.xml, czyli deskryptor wdrożenia aplikacji sieciowej	308
10.4. Servlet DispatcherServlet	311
Uzyskanie dostępu do obiektów ServletContext i ServletConfig	313
10.5. Opracowanie kontrolerów za pomocą adnotacji @Controller i @RequestMapping	314
Opracowanie aplikacji sieciowej za pomocą kontrolera oznaczonego adnotacją	314
10.6. Wymagania aplikacji sieciowej MyBank	317
10.7. Adnotacje Spring Web MVC: @RequestMapping i @RequestParam	318
Mapowanie żądań na kontrolery lub metody kontrolerów	
za pomocą adnotacji @RequestMapping	318
Argumenty metod oznaczonych adnotacją @RequestMapping	324
Typy wartości zwrotnych metod oznaczonych adnotacją @RequestMapping	325
Przekazanie parametrów żądania do metod kontrolera z użyciem adnotacji	
@RequestParam	326
10.8. Weryfikacja	330
10.9. Obsługa wyjątków za pomocą adnotacji @ExceptionHandler	333
10.10. Wczytanie głównego pliku XML kontekstu aplikacji sieciowej	335
10.11. Podsumowanie	335

Rozdział 11. Weryfikacja i dołączanie danych w Spring Web MVC	337
11.1. Wprowadzenie	337
11.2. Dodawanie i pobieranie atrybutów modelu za pomocą adnotacji @ModelAttribute	337
Dodanie atrybutów modelu za pomocą adnotacji @ModelAttribute stosowane na poziomie metody	338
Pobieranie atrybutów modelu za pomocą adnotacji @ModelAttribute	342
Przetwarzanie żądania i metody oznaczone adnotacją @ModelAttribute	343
Zachowanie argumentów metody oznaczonych adnotacją @ModelAttribute	345
Obiekt RequestToViewNameTranslator	346
11.3. Buforowanie atrybutów modelu za pomocą adnotacji @SessionAttributes	347
11.4. Dołączanie danych	350
WebDataBinder, czyli dołączanie danych dla parametrów żądania sieciowego	352
Konfiguracja egzemplarza WebDataBinder	353
Włączenie lub wyłączenie poszczególnych właściwości w procesie dołączania danych	358
Analiza błędów dołączania danych i weryfikacji przechowywanych przez obiekt BindingResult	361
11.5. Weryfikacja w Springu	362
Weryfikacja atrybutów modelu za pomocą oferowanego przez Spring interfejsu Validator	363
Zdefiniowanie ograniczeń za pomocą adnotacji JSR 303	366
Weryfikacja obiektu używającego adnotacji JSR 303	367
11.6. Biblioteka znaczników form	371
Obsługa HTML5 przez bibliotekę form	372
11.7. Podsumowanie	374
Rozdział 12. Tworzenie usług sieciowych typu RESTful za pomocą Spring Web MVC	375
12.1. Wprowadzenie	375
12.2. Usługa sieciowa do obsługi lokat	376
12.3. Implementacja usługi sieciowej w stylu RESTful za pomocą Spring Web MVC	376
Format JSON	377
Implementacja usługi sieciowej FixedDepositWS	379

12.4. Uzyskanie dostępu do usługi sieciowej w stylu RESTful za pomocą klasy RestTemplate	386
12.5. Użycie HttpMessageConverter do konwersji obiektów Javy na żądania lub odpowiedzi HTTP i na odwrót	394
12.6. Adnotacje @PathVariable i @MatrixVariable	395
12.7. Podsumowanie	399

Rozdział 13. *Więcej o Spring Web MVC: internacjonalizacja, przesyłanie plików i przetwarzanie żądań asynchronicznych 401*

13.1. Wprowadzenie	401
13.2. Przetwarzanie żądań za pomocą procedur obsługi interceptorów	401
Implementacja i konfiguracja procedury obsługi interceptorów	401
13.3. Internacjonalizacja za pomocą paczek zasobów	404
Wymagania aplikacji sieciowej MyBank	404
Internacjonalizacja i lokalizacja aplikacji sieciowej MyBank	404
13.4. Asynchroniczne przetwarzanie żądań	407
Konfiguracja asynchronicznego przetwarzania żądań	407
Zwrot obiektu Callable przez metodę oznaczoną adnotacją @RequestMapping	408
Zwrot obiektu DeferredResult przez metodę oznaczoną adnotacją @RequestMapping	409
Ustawienie domyślnego limitu czasu	416
Przechwytywanie żądań asynchronicznych	417
13.5. Obsługa konwersji typu i formatowania	417
Utworzenie własnej implementacji interfejsu Converter	418
Konfiguracja i użycie własnej implementacji interfejsu Converter	418
Tworzenie własnej implementacji Formatter	420
Konfiguracja własnej implementacji Formatter	422
Utworzenie implementacji AnnotationFormatterFactory do formatowania jedynie właściwości oznaczonych adnotacją @AmountFormat	422
Konfiguracja implementacji AnnotationFormatterFactory	423
13.6. Obsługa przekazywania plików w Spring Web MVC	425
Przekazywanie plików za pomocą implementacji CommonsMultipartResolver	425
Przekazywanie plików za pomocą StandardServletMultipartResolver	427
13.7. Podsumowanie	428

Rozdział 14. Zabezpieczanie aplikacji za pomocą Spring Security	429
14.1. Wprowadzenie	429
14.2. Wymagania aplikacji sieciowej MyBank w zakresie bezpieczeństwa	429
14.3. Zabezpieczanie aplikacji MyBank za pomocą frameworku Spring Security	431
Konfiguracja zabezpieczeń dla żądań sieciowych	431
Konfiguracja uwierzytelniania	434
Zabezpieczenie zawartości strony JSP przez użycie oferowanej przez Spring Security biblioteki znaczników	435
Zabezpieczanie metod	436
14.4. Zabezpieczanie egzemplarzy FixedDepositDetails za pomocą modułu ACL	440
Wdrożenie i użycie projektu ch14-bankapp-db-security	440
Tabele bazy danych przeznaczone do przechowywania informacji o listach ACL i użytkownikach	442
Uwierzytelnianie użytkownika	446
Zabezpieczenie żądań sieciowych	447
Konfiguracja JdbcMutableAclService	449
Konfiguracja zabezpieczeń na poziomie metody	452
Bezpieczeństwo egzemplarza obiektu domeny	453
Programowe zarządzanie informacjami ACL	455
Egzemplarz MutableAcl i bezpieczeństwo	458
14.5. Podsumowanie	459
 Dodatek A Import i wdrożenie przykładowych projektów w Eclipse IDE (lub IntelliJ IDEA)	 461
A.1. Przygotowanie środowiska pracy	461
A.2. Import przykładowego projektu do Eclipse IDE (lub IntelliJ IDEA)	461
Import przykładowego projektu	462
Konfiguracja zmiennej M2_REPO w Eclipse IDE	463
A.3. Konfiguracja Eclipse IDE do pracy z serwerem Tomcat 7	463
A.4. Wdrożenie projektu sieciowego w serwerze Tomcat 7	465
Uruchomienie serwera Tomcat 7 w trybie osadzonym	465
 Skorowidz	 467

Rozdział 9. Programowanie zorientowane aspektowo

9.1. Wprowadzenie

Programowanie zorientowane aspektowo (ang. *aspect-oriented programming*, AOP) to rodzaj podejścia programistycznego, w którym obowiązki rozproszone między wieloma klasami są hermetyzowane w oddzielnej klasie określonej mianem *aspektu*. Natomiast obowiązki rozproszone między wieloma klasami są nazywane *zagadnieniami przekrojowymi* (ang. *cross-cutting concern*). Rejestrowanie danych, zarządzanie transakcjami, buforowanie, zapewnienie bezpieczeństwa itd. to przykłady zadań przekrojowych.

Spring oferuje framework AOP używany wewnętrznie przez Spring do implementacji usług deklarycyjnych, na przykład zarządzania transakcjami (patrz: rozdział 7.) i buforowania (patrz: rozdział 8.). Zamiast stosować oferowany przez Spring framework AOP rozważ użycie AspectJ (<http://www.eclipse.org/aspectj/>) jako frameworku AOP dla tworzonej aplikacji. Ponieważ możliwości frameworku Spring AOP są wystarczające w większości sytuacji, a ponadto zapewnia on integrację z kontenerem Springa, w tym rozdziale skoncentrujemy się właśnie na użyciu frameworku AOP dostarczanego przez Spring.

Rozpoczynamy od przykładu użycia AOP.

9.2. Prosty przykład AOP

Przyjmujemy założenie, że na potrzeby audytu konieczne jest przechwytywanie argumentów przekazywanych w klasach, które zostały zdefiniowane w warstwie usługi w aplikacji MyBank. Proste podejście w zakresie rejestracji danych o argumentach metody polega na umieszczeniu w każdej metodzie logiki odpowiedzialnej za rejestrację danych. Takie rozwiązanie oznacza jednak, że każda metoda otrzymuje *dodatkowe* zadanie, jakim jest rejestracja szczegółów dotyczących argumentów metody. Ponieważ zadanie rejestracji danych o argumentach metody pojawia się w wielu klasach i metodach, przedstawia *zagadnienie przekrojowe*.

Przygotowanie obsługi zagadnienia przekrojowego za pomocą AOP wymaga wykonania następujących kroków:

- utworzenie klasy Javy (określonej mianem aspektu);
- dodanie do nowo utworzonej klasy implementacji zagadnienia przekrojowego;
- użycie wyrażenia regularnego w celu określenia metod, do których ma zastosowanie przygotowane zagadnienie przekrojowe.

W terminologii programowania zorientowanego aspektowego metody aspektu implementującego zagadnienia przekrojowe są określane mianem *rady* (ang. *advice*). Każda rada jest powiązana z *punktem przecięcia* (ang. *pointcut*) identyfikującym metody, do których ma zastosowanie rada. Metody, do których stosowana będzie rada, są nazywane *punktami złączenia* (ang. *join points*).

W oferowanym przez Spring frameworku AOP masz możliwość opracowania aspektu za pomocą *stylu adnotacji* (AspectJ) lub *stylu schematu* (XML). W przypadku stylu adnotacji udostępniane przez AspectJ adnotacje takie jak `@Aspect`, `@Pointcut` i `@Before` są używane do przygotowania aspektu. Natomiast w stylu schematu elementy schematu aop służą do konfiguracji komponentu Springa jako aspektu.

IMPORT **Rozdział09/ch09-simple-aop.** (Ten projekt pokazuje aplikację MyBank, w której framework Spring AOP został wykorzystany do rejestracji informacji o argumentach metod przekazanych metodom zdefiniowanym przez klasy w warstwie usługi aplikacji MyBank. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

W listingu 9.1 przedstawiamy przykład aspektu rejestrowania danych, który zapisuje informacje o argumentach przekazanych metodom usług w aplikacji MyBank.

Listing 9.1. Klasa `LoggingAspect`

Projekt: `ch09-simple-aop`

Położenie pliku: `src/main/java/sample/spring/chapter09/bankapp/aspects`

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {
    private Logger logger = Logger.getLogger(LoggingAspect.class);

    @Before(value = "execution(*
        sample.spring.chapter09.bankapp.service.*Service.*(..))")
    public void log(JoinPoint joinPoint) {
        logger.info("Wejście do metody "
            + joinPoint.getSignature().getName() + " klasy "
            + joinPoint.getTarget().getClass().getSimpleName());

        Object[] args = joinPoint.getArgs();
        for (int i = 0; i < args.length; i++) {
            logger.info("args[" + i + "] --> " + args[i]);
        }
    }
}
```

W listingu 9.1:

- Adnotacja `@Aspect` na poziomie typu oznacza, że klasa `LoggingAspect` jest *aspektem* AOP.
- Adnotacja `@Before` na poziomie metody oznacza, że metoda `log()` przedstawia *radę* stosowaną, *zanim* nastąpi wykonanie metod dopasowanych przez atrybut `value`. W podrozdziale 9.5 znajdziesz więcej informacji o różnych typach rad, które możesz zdefiniować.
- Atrybut `value` adnotacji `@Before` określa *wyrażenie punktu przecięcia*, które jest przez framework Spring AOP używane w celu identyfikacji metod (nazywanych *metodami docelowymi*) stosujących zdefiniowane rady. W podrozdziale 9.4 znajdziesz dokładne omówienie wyrażeń

punktu przecięcia. W tym miejscu możesz przyjąć następujące założenie: wyrażenie punktu przecięcia `execution(* sample.spring.chapter09.bankapp.service.*Service.*(..))` określa, że metoda `log()` klasy `LoggingAspect` ma zastosowanie dla *wszystkich* metod publicznych zdefiniowanych przez klasy (lub interfejsy) w pakiecie `sample.spring.chapter09.bankapp`.
→ `service` i których nazwa kończy się na `Service`.

- Argument `JoinPoint` metody `log()` przedstawia metodę docelową, w której zostanie zastosowana rada. Metoda `log()` używa egzemplarza `JoinPoint` w celu pobrania informacji o argumentach przekazanych metodzie docelowej. W listingu 9.1 metoda `getArgs()` egzemplarza `JoinPoint` jest wywoływana i pobiera argumenty przekazane metodzie docelowej.

Aspekt trzeba zarejestrować w kontenerze Springa, aby udostępniany przez Spring framework AOP wiedział o istnieniu aspektu. W listingu 9.1 klasa `LoggingAspect` została oznaczona adnotacją `@Component`, a więc jest automatycznie rejestrowana w kontenerze Springa.

W listingu 9.2 została przedstawiona klasa `BankApp` wywołująca metody klas `BankAccountService` → `ServiceImpl` (implementuje interfejs `BankAccountService`) i `FixedDepositServiceImpl` (implementuje interfejs `FixedDepositService`) aplikacji `MyBank`.

Listing 9.2. Klasa `BankApp`

Projekt: `ch09-simple-aop`

Położenie pliku: `src/main/java/sample/spring/chapter09/bankapp`

```
package sample.spring.chapter09.bankapp;
.....
public class BankApp {
    public static void main(String args[]) throws Exception {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");

        BankAccountService bankAccountService =
            context.getBean(BankAccountService.class);
        BankAccountDetails bankAccountDetails = new BankAccountDetails();
        bankAccountDetails.setBalanceAmount(1000);
        bankAccountDetails.setLastTransactionTimestamp(new Date());
        bankAccountService.createBankAccount(bankAccountDetails);

        FixedDepositService FixedDepositService =
            context.getBean(FixedDepositService.class);
        FixedDepositService.createFixedDeposit(new FixedDepositDetails(1, 1000, 12,
            "nazwa-uzytkownika@nazwa-domeny.pl"));
    }
}
```

W powyższym listingu przez metodę `main()` klasy `BankApp` wywoływane są metoda `createBankAccount()` klasy `BankAccountService` i metoda `createFixedDeposit()` klasy `FixedDepositService`. Jeżeli więc wywołasz metodę `main()` klasy `BankApp`, w konsoli zostaną wyświetlone następujące dane wyjściowe:

```
INFO LoggingAspect - Wejście do metody createBankAccount klasy BankAccountServiceImpl
INFO LoggingAspect - args[0] -->BankAccountDetails [accountId=0, balanceAmount=1000,
↳lastTransactionTimestamp=Sat Oct 27 16:48:11 IST 2012]
INFO BankAccountServiceImpl - Wywołanie metody createBankAccount
```

```
INFO LoggingAspect - Wejście do metody createFixedDeposit klasy FixedDepositServiceImpl
INFO LoggingAspect - args[0] -->id :1, wysokość lokaty : 1000.0, czas trwania : 12, e-mail :
↳nazwa-uzytkownika@nazwa-domeny.pl
INFO FixedDepositServiceImpl - Wywołanie metody createFixedDeposit
```

Powyższe dane wyjściowe pokazują, że metoda `log()` klasy `LoggingAspect` jest wykonywana przed metodą `createBankAccount()` klasy `BankAccountService` oraz metodą `createFixedDeposit()` klasy `FixedDepositService`.

W kontekście `LoggingAspect` zobaczmy teraz, jak działa oferowany przez Spring framework AOP.

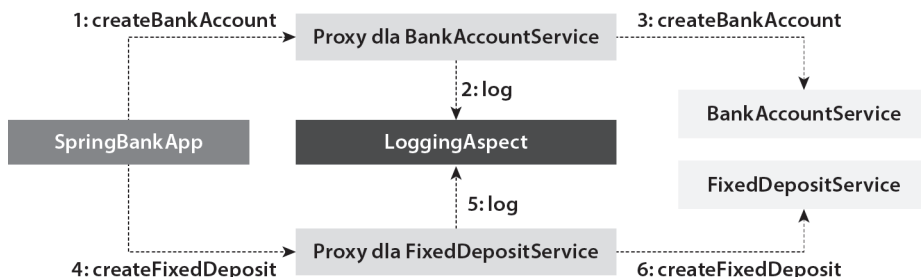


W projekcie *ch09-simple-aop* zdefiniowano w celu użycia stylu adnotacji `AspectJ` zależność od plików JAR o nazwach *spring-aop*, *aopalliance*, *aspectjrt* i *aspectjweaver*. Więcej informacji na ten temat znajdziesz w pliku *pom.xml* znajdującym się w projekcie *ch09-simple-aop*.

9.3. Oferowany przez Spring framework AOP

Framework AOP w Springu jest *oparty na proxy*, a więc *obiekt proxy* jest tworzony dla obiektów będących celem rady. Proxy to obiekt pośredniczący, wprowadzony przez framework AOP i znajdujący się między obiektem wywołującym i obiektem docelowym. W trakcie działania aplikacji wywołania do obiektu docelowego są przechwytywane przez proxy, a rady zdefiniowane dla metod docelowych są wykonywane przez proxy. We frameworku AOP obiekt docelowy jest egzemplarzem komponentu zarejestrowanym w kontenerze Springa.

Diagram na rysunku 9.1 pokazuje metodę `log()` klasy `LoggingAspect` (patrz: listing 9.1) zastosowaną do metod klas `BankAccountService` i `FixedDepositService` (patrz: listing 9.2).



Rysunek 9.1. Obiekt proxy jest odpowiedzialny za przechwytywanie wywołań metod w obiekcie docelowym oraz wykonywanie rad mających zastosowanie dla metody docelowej

Na rysunku 9.1 widać, że proxy jest tworzone dla obiektów `BankAccountService` i `FixedDepositService`. Proxy dla obiektu `BankAccountService` przechwytuje wywołania metody `createBankAccount()`, natomiast proxy dla obiektu `FixedDepositService` przechwytuje wywołania metody `createFixedDeposit()`. Proxy dla obiektu `BankAccountService` najpierw wykonuje metodę `log()` klasy `LoggingAspect`, a następnie metodę `createBankAccount()` klasy `BankAccountService`. Podobnie proxy dla obiektu `FixedDepositService` najpierw wykonuje metodę `log()` klasy `LoggingAspect`, a następnie metodę `createFixedDeposit()` klasy `FixedDepositService`.

Czas wykonania rady (takiej jak metoda `log()` aspektu `LoggingAspect`) zależy od *typu* rady. W stylu adnotacji AspectJ typ rady jest określany za pomocą adnotacji AspectJ. Na przykład adnotacja `@Before` oznacza wykonanie rady *przed* wywołaniem metody docelowej, natomiast adnotacja `@After` oznacza wykonanie rady *po* wywołaniu metody docelowej. Adnotacja `@Around` oznacza wykonanie rady zarówno *przed*, jak i *po* wykonaniu metody docelowej. Ponieważ metoda `log()` klasy `LoggingAspect` została oznaczona adnotacją `@Before`, metoda `log()` będzie wykonywana *przed* wykonaniem metody obiektu docelowego.

Zobaczmy teraz, jak framework AOP w Springu tworzy obiekt proxy.

Tworzenie proxy

Podczas użycia frameworku Spring AOP masz możliwość jawnego tworzenia proxy za pomocą `ProxyFactoryBean` (patrz: pakiet `org.springframework.aop.framework`) lub pozostawienia Springowi zadania automatycznego tworzenia proxy AOP, co jest określane mianem *autoproxying*.

Jeżeli do tworzenia aspektów chcesz zastosować adnotacje w stylu AspectJ, musisz włączyć obsługę tego rodzaju adnotacji, co wymaga użycia elementu `<aspectj-autoproxy>` schematu aop oferowanego przez Spring. Element `<aspectj-autoproxy>` nakazuje także frameworkowi Spring AOP automatyczne tworzenie proxy AOP dla obiektów docelowych. Listing 9.3 przedstawia przykład użycia elementu `<aspectj-autoproxy>` w projekcie *ch09-simple-aop*.

Listing 9.3. Plik `applicationContext.xml` przedstawiający użycie elementu `<aspectj-autoproxy>`

Projekt: *ch09-simple-aop*

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans .....
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation=".....http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">

  <context:component-scan base-package="sample.spring" />
  <aop:aspectj-autoproxy proxy-target-class="false" expose-proxy="true"/>
</beans>
```

Atrybut `proxy-target-class` elementu `<aspectj-autoproxy>` określa rodzaj proxy (oparte na Javie SE lub na CGLIB) tworzonych dla obiektów docelowych. Z kolei atrybut `expose-proxy` określa, czy proxy AOP będzie dostępne dla obiektu docelowego. Jeżeli wartością atrybutu `expose-proxy` jest `true`, wówczas metoda obiektu docelowego może uzyskać dostęp do proxy AOP przez wywołanie metody statycznej `currentProxy()` klasy `AopContext`.

Framework Spring AOP tworzy proxy oparte na CGLIB lub Javie SE. Jeżeli obiekt docelowy nie implementuje żadnego interfejsu, wówczas Spring AOP tworzy proxy oparte na CGLIB. Natomiast w przypadku, gdy obiekt docelowy implementuje jeden lub więcej interfejsów, Spring AOP tworzy proxy oparte na Javie SE. Jeżeli wartością atrybutu `proxy-target-class` elementu `<aspectj-autoproxy>` będzie `false`, wówczas Spring AOP tworzy proxy oparte na Javie SE, gdy

obiekt docelowy implementuje jeden bądź więcej interfejsów. W przypadku przypisania atrybutowi `proxy-target-class` wartości `true` Spring AOP tworzy proxy oparte na CGLIB, nawet jeśli obiekt implementuje jeden lub więcej interfejsów.



Począwszy od wersji Spring 3.2, klasy CGLIB znajdują się w pliku JAR o nazwie *spring-core*. Dlatego nie trzeba w aplikacji wyraźnie wskazywać pliku JAR CGLIB, aby umożliwić frameworkowi Spring AOP tworzenie dla obiektów docelowych proxy opartych na CGLIB.

Przechodzimy teraz do przypadku, w którym preferowane będzie przypisanie wartości `true` atrybutowi `expose-proxy` elementu `<aspectj-autoproxy>`.

IMPORT Rozdział 109/ch09-aop-proxy. (Ten projekt pokazuje aplikację *MyBank*, w której metoda `currentProxy()` klasy `AopProxy` została użyta przez metodę docelową do pobrania obiektu proxy AOP utworzonego przez framework AOP w Springu. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Atrybut `expose-proxy`

W listingu 9.4 została przedstawiona zmodyfikowana wersja klasy `BankAccountServiceImpl`, w której metoda `createBankAccount()` wywołuje metodę `isDuplicateAccount()` w celu sprawdzenia, czy w systemie istnieje już konto bankowe charakteryzujące się podanymi cechami.

Listing 9.4. Klasa `BankAccountServiceImpl`

```
@Service(value = "bankAccountService")
public class BankAccountServiceImpl implements BankAccountService {
    @Autowired
    private BankAccountDao bankAccountDao;

    @Override
    public int createBankAccount(BankAccountDetails bankAccountDetails) {
        if(!isDuplicateAccount(bankAccountDetails)) {
            return bankAccountDao.createBankAccount(bankAccountDetails);
        } else {
            throw new BankAccountAlreadyExistsException("Konto już istnieje.");
        }
    }

    @Override
    public boolean isDuplicateAccount(BankAccountDetails bankAccountDetails) { .... }
}
```

W powyższym listingu widać, że metoda `createBankAccount()` wywołuje metodę `isDuplicateAccount()` w celu sprawdzenia, czy konto bankowe o podanych parametrach istnieje już w systemie.

W tym momencie może pojawić się pytanie, czy metoda `log()` klasy `LoggingAspect` (patrz: listing 9.1) zostanie wykonana, gdy nastąpi wywołanie metody `isDuplicateAccount()` przez metodę `createBankAccount()`. Nawet pomimo dopasowania przez metodę `isDuplicateAccount()` wyrażenia punktu przecięcia określonego za pomocą adnotacji `@Before` metody `log()` klasy `LoggingAspect` (patrz: listing 9.1) metoda `log()` *nie* zostanie wywołana. Wynika to z faktu, że względem metod

wywoływanych przez obiekt docelowy *nie* jest stosowane proxy przez AOP. Ponieważ wywołanie metody nie przechodzi przez obiekt proxy AOP, jakkolwiek rada powiązana z metodą docelową *nie* zostanie wykonana.

Aby zagwarantować, że wywołanie metody `isDuplicateAccount()` będzie do obiektu docelowego przeprowadzone poprzez proxy AOP, w metodzie `createBankAccount()` należy pobrać obiekt proxy AOP, a następnie wywołać metodę `isDuplicateAccount()` w tym obiekcie. Listing 9.5 przedstawia przykład pobrania obiektu proxy AOP w metodzie `createBankAccount()`.

Listing 9.5. Klasa `BankAccountServiceImpl`

Projekt: *ch09-aop-proxy*

Położenie pliku: *src/main/java/sample/spring/chapter09/bankapp/service*

```
package sample.spring.chapter09.bankapp.service;

import org.springframework.aop.framework.AopContext;
.....
@Service(value = "bankAccountService")
public class BankAccountServiceImpl implements BankAccountService {
    .....
    @Override
    public int createBankAccount(BankAccountDetails bankAccountDetails) {
        // Pobranie proxy i wywołanie za jego pomocą metody isDuplicateAccount().
        boolean isDuplicateAccount =
            ((BankAccountService)AopContext.currentProxy()).
                isDuplicateAccount(bankAccountDetails);

        if(!isDuplicateAccount) { ..... }
        .....
    }

    @Override
    public boolean isDuplicateAccount(BankAccountDetails bankAccountDetails) { ..... }
}
```

W powyższym listingu wywołanie metody `currentProxy()` klasy `AopContext` zwraca proxy AOP, które wykonuje wywołanie metody `createBankAccount()`. Jeżeli metoda `createBankAccount()` nie zostanie wykonana za pomocą frameworku Spring AOP lub wartością atrybutu `expose-proxy` elementu `<aspectj-autoproxy>` jest `false`, wówczas wywołanie metody `currentProxy()` zakończy się zgłoszeniem wyjątku `java.lang.IllegalStateException`. Ponieważ proxy AOP implementuje ten sam interfejs co obiekt docelowy, w omawianym listingu zwrócone przez metodę `currentProxy()` proxy AOP jest rzutowane na typ `BankAccountService` i następuje wywołanie metody `isDuplicateAccount()` klasy `BankAccountService`.

Jeżeli przejdiesz do projektu *ch09-aop-proxy* i wykonasz metodę `main()` klasy `BankApp`, zauważysz, że metoda `log()` klasy `LoggingAspect` jest wykonywana, gdy nastąpi wywołanie metody `isDuplicateAccount()` przez metodę `createBankAccount()`.

Teraz przechodzimy do dokładnego omówienia wyrażen punktu przecięcia.

9.4. Wyrażenia punktu przecięcia

Podczas użycia frameworku Spring AOP wyrażenie punktu przecięcia identyfikuje *punkty połączeń*, do których zastosowanie mają rady. W przypadku Springa AOP punkt połączenia *zawsze* jest metodą komponentu. Jeżeli chcesz zastosować rady dla właściwości, konstruktorów, metod niepublicznych oraz obiektów niebędących komponentami Springa, powinieneś użyć AspectJ zamiast frameworku Spring AOP. Jeśli masz zamiar opracować aspekty za pomocą adnotacji stylu AspectJ, wyrażenie punktu przecięcia możesz podać, używając adnotacji AspectJ @Pointcut lub adnotacji AspectJ @Before, @After itd. określających typ rady.

Wyrażenia punktów przecięcia używają tak zwanych *desygnatorów punktu przecięcia*, takich jak execution, args, within, this itd., w celu dopasowania metod, do których ma być zastosowana rada. Na przykład w listingu 9.1 adnotacja @Before zawiera desygnator execution, a tym samym wyszukuje metody, do których zastosowanie będzie miała metoda log() klasy LoggingAspect.

Teraz zobaczysz, jak wyrażenia punktów przecięcia są określone za pomocą adnotacji @Pointcut.

IMPORT **Rozdział 109/ch09-aop-pointcut.** (Ten projekt pokazuje aplikację MyBank, w której użyto adnotacji AspectJ @Pointcut w celu określenia wyrażenia punktu przecięcia. Aby uruchomić tę aplikację, wystarczy wywołać metodę main() w klasie BankApp projektu).

Adnotacja @Pointcut

Atrybut value adnotacji @Pointcut określa wyrażenie punktu przecięcia. Aby użyć adnotacji @Pointcut, utwórz *pustą* metodę i oznacz ją adnotacją @Pointcut. Pusta metoda *musi* być zdefiniowana w taki sposób, aby zwracała void. Rada odwołująca się do metody oznaczonej adnotacją @Pointcut będzie zastosowana w metodach dopasowanych przez wyrażenie punktu przecięcia wskazane przez adnotację @Pointcut.



Użycie adnotacji @Pointcut okazuje się szczególnie użyteczne, gdy wyrażenie punktu przecięcia jest współdzielone przez wiele rad w tym samym aspekcie lub w różnych aspektach.

W listingu 9.6 została przedstawiona zmodyfikowana wersja klasy LoggingAspect (patrz: listing 9.1) używająca adnotacji @Pointcut.

Listing 9.6. Klasa LoggingAspect

Projekt: ch09-aop-pointcuts

Położenie pliku: src/main/java/sample/spring/chapter09/bankapp/aspects

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
@Component
public class LoggingAspect {
```

```

@Pointcut(value = "execution(*
    sample.spring.chapter09.bankapp.service.*Service.*(..))")
private void invokeServiceMethods() { }

@Before(value = "invokeServiceMethods()")
public void log(JoinPoint joinPoint) {
    logger.info("Wejście do metody "
        + joinPoint.getSignature().getName() + " klasy "
        + joinPoint.getTarget().getClass().getSimpleName();
        ....
    }
}

```

W powyższym listingu metoda `invokeServiceMethods` została oznaczona adnotacją `@Pointcut`, a atrybut `value` adnotacji `@Before` odwołuje się do metody `invokeServiceMethods()`. Oznacza to, że metoda `log()` będzie zastosowana względem metod dopasowanych przez wyrażenie punktu przecięcia określone za pomocą adnotacji `@Pointcut` w metodzie `invokeServiceMethods()`.

Ponieważ `execution` i `args` są najczęściej używanymi desygntorami punktu przecięcia, teraz dokładnie je przeanalizujemy.

Desygntory `execution` i `args`

Desygntator `execution` punktu przecięcia ma następujący format:

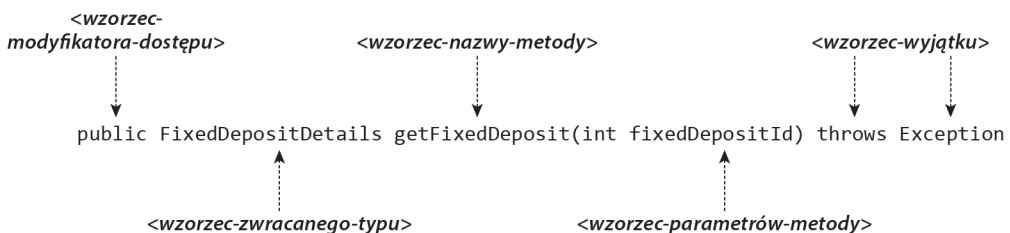
```

execution(<wzorzec-modyfikatora-dostępu> <wzorzec-zwracanego-typu> <wzorzec-deklarowanego-typu>
    <wzorzec-nazwy-metody>(<wzorzec-parametrów-metody>) <wzorzec-wyjątku>)

```

Jeżeli porównasz wyrażenie `execution` z deklaracją metody, zauważysz występujące między nimi podobieństwa.

Na rysunku 9.2 pokazujemy, jak poszczególne fragmenty wyrażenia `execution` można mapować na deklarację metody.



Rysunek 9.2. Mapowanie poszczególnych fragmentów wyrażenia `execution` na różne części deklaracji metody

Framework Spring AOP dopasowuje poszczególne fragmenty wyrażenia `expression` do różnych części deklaracji metody (patrz: rysunek 9.2) w celu znalezienia metod, do których ma zastosowanie dana rada. Na rysunku nie pokazano `<wzorca-deklarowanego-typu>`, ponieważ jest on używany tylko wtedy, gdy zachodzi potrzeba odwołania się do metod, które znajdują się w określonym pakiecie lub są określonego typu.

W tabeli 9.1 wymieniamy poszczególne fragmenty tworzące wyrażenie `execution`.

Tabela 9.1. Elementy tworzące wyrażenie `execution`

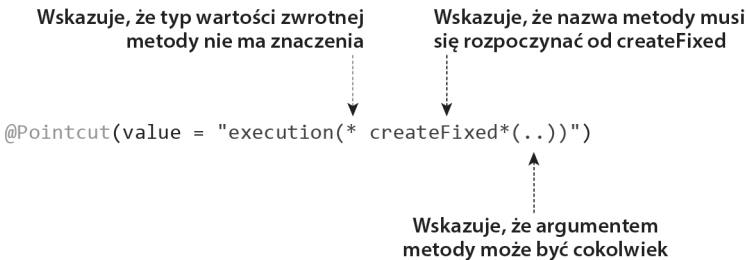
Fragment wyrażenia	Opis
<i>wzorzec-modyfikatora-dostępu</i>	Określa modyfikator dostępu metody docelowej. W Springu AOP jedyną wartością, jaką można określić dla tej części wyrażenia, to <code>public</code> . Ta część wyrażenia <code>execution</code> jest <i>opcjonalna</i> .
<i>wzorzec-zwracanego-typu</i>	Określa w pełni kwalifikowaną nazwę typu zwrótnego metody docelowej. Wartość <code>*</code> oznacza, że typ zwrótny metody docelowej nie ma znaczenia.
<i>wzorzec-deklarowanego-typu</i>	Określa w pełni kwalifikowaną nazwę typu zwrótnego zawierającego metodę docelową. Ta część wyrażenia <code>execution</code> jest <i>opcjonalna</i> . Wartość <code>*</code> oznacza, że wszystkie typy (klasy i interfejsy) w aplikacji są uwzględniane przez wyrażenie punktu przecięcia.
<i>wzorzec-nazwy-metody</i>	Określa wzorzec nazwy metody. Na przykład wartość <code>save*</code> oznacza, że metody o nazwach rozpoczynających się od <code>save</code> będą celem dla danej rady.
<i>wzorzec-parametrów-metody</i>	Określa wzorzec parametru metody. Jeżeli wartością będzie <code>(..)</code> , oznacza to, że metoda docelowa może zawierać dowolną liczbę argumentów lub żaden.
<i>wzorzec-wyjątku</i>	Określa wyjątek (lub wyjątki) zgłaszany przez metodę docelową. Ta część wyrażenia <code>execution</code> jest <i>opcjonalna</i> .

Desygnator `args` określa argumenty, jakie muszą być akceptowane przez metodę docelową *w trakcie działania aplikacji*. Na przykład jeśli wyrażenie punktu przecięcia przeznaczone jest do wyszukiwania metod akceptujących egzemplarz `java.util.List`, wówczas desygnator `args` ma postać `args(java.util.List)`. W dalszej części podrozdziału zobaczysz, jak desygnator `args` może być użyty do udostępniania radzie argumentów przekazywanych metodzie docelowej.

Teraz przeanalizujemy kilka przykładów wyrażań punktu przecięcia używających desygnatorów `execution` i `args`.

Przykład 1.

Na rysunku 9.3 zostało pokazane wyrażenie używające wzorca nazwy metody.

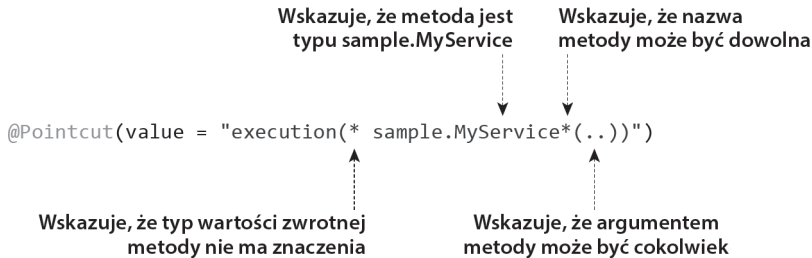


Rysunek 9.3. Wyrażenie punktu przecięcia używające wzorca nazwy metody

Przedstawione wyrażenie punktu przecięcia powoduje dopasowanie metod o nazwach rozpoczynających się od `createFixed`. Jako typ wartości zwrótniej podano `*`, a więc metoda docelowa może zwracać wartość dowolnego typu. Zapis `(..)` oznacza, że metoda może akceptować zero lub więcej argumentów.

Przykład 2.

Na rysunku 9.4 zostało pokazane wyrażenie określające typ metody.

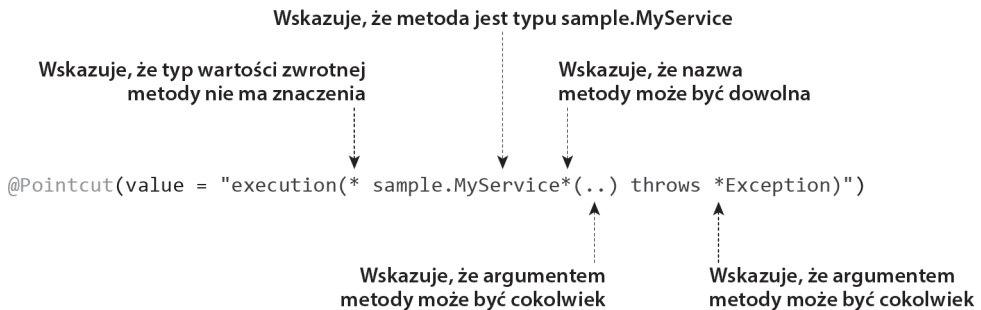


Rysunek 9.4. Wyrażenie punktu przecięcia wskazujące typ (klasa lub interfejs) zawierający metodę docelową

Przedstawione wyrażenie punktu przecięcia powoduje dopasowanie metod typu `MyService` zdefiniowanych w pakiecie `sample`.

Przykład 3.

Na rysunku 9.5 zostało pokazane wyrażenie określające wzorzec metody.

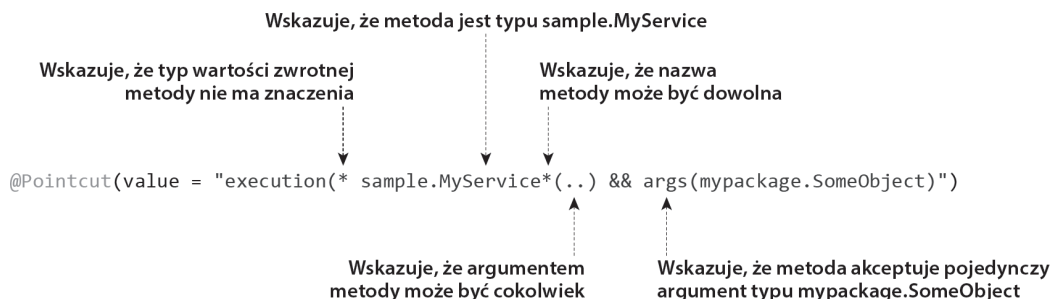


Rysunek 9.5. Wyrażenie punktu przecięcia wskazujące wzorzec metody

Przedstawione wyrażenie punktu przecięcia powoduje dopasowanie metod typu `sample.MyService` zawierających klauzulę `throws`.

Przykład 4.

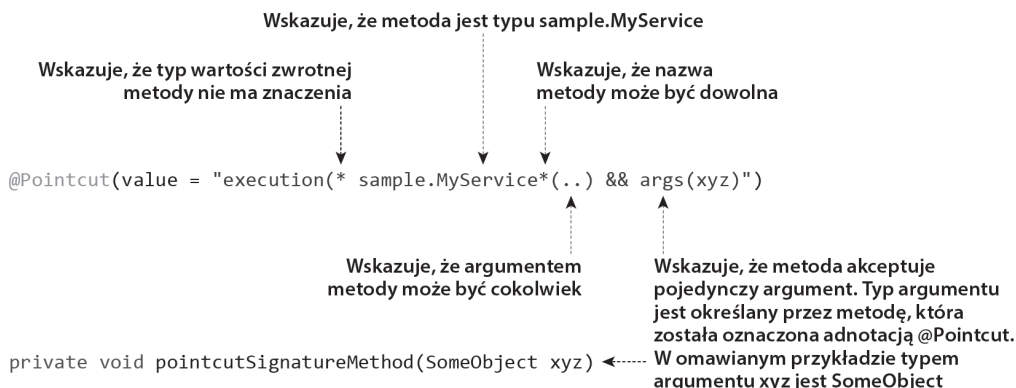
Na rysunku 9.6 zostało pokazane wyrażenie określające egzemplarz obiektu przekazywany metodzie docelowej.



Rysunek 9.6. Wyrażenie punktu przecięcia wskazujące egzemplarz obiektu przekazywany metodzie docelowej

Przedstawione wyrażenie punktu przecięcia pokazuje użycie desygatorów `execution` i `args`. Jeżeli chcesz jednocześnie użyć więcej niż jednego desygatora, możesz je łączyć za pomocą operatorów `&&` i `||`, tworząc w ten sposób skomplikowane wyrażenia punktu przecięcia. Powyższe wyrażenie dopasowuje metody zdefiniowane w pakiecie `sample.MyService` akceptujące egzemplarz `SomeObject`. Operator `&&` w omawianym wyrażeniu punktu przecięcia oznacza, że metoda docelowa *musi* być dopasowana w wyrażeniach określonych przez desygatory `execution` i `args`.

Jeżeli rada ma uzyskać dostęp do jednego lub więcej argumentów przekazanych metodzie docelowej, wówczas nazwy argumentów należy podać w wyrażeniu `args`, jak pokazuje rysunek 9.7.



Rysunek 9.7. Desygator `args` określa argumenty metody docelowej, które muszą być dostępne dla rady

W pokazanym na rysunku 9.7 wyrażeniu punktu przecięcia desygator `args` określa, że metoda docelowa *musi* akceptować argument typu `SomeObject`, a sam argument jest dostępny dla rady za pomocą parametru `xyz`. Przechodzimy teraz do praktycznego przykładu użycia tej funkcji w celu przekazania argumentów metodzie docelowej.

Przekazanie do rady argumentów metody docelowej

W listingu 9.7 została przedstawiona zmodyfikowana wersja klasy `LoggingAspect`, której metoda `log()` jest wykonywana tylko wtedy, gdy argument przekazany metodzie docelowej jest egzemplarzem `FixedDepositDetails`, a egzemplarz `FixedDepositDetails` również jest dostępny dla metody `log()`.

Listing 9.7. Klasa *LoggingAspect* przedstawiająca przekazanie radzie argumentów metody docelowej

```
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
@Component
public class LoggingAspect {
    ....
    @Pointcut(value =
        "execution(* sample.spring.chapter09.bankapp.service.*Service.*(..))
        && args(FixedDepositDetails) ")

    private void invokeServiceMethods(FixedDepositDetails FixedDepositDetails) {
    }

    @Before(value = "invokeServiceMethods(FixedDepositDetails)")
    public void log(JoinPoint joinPoint, FixedDepositDetails FixedDepositDetails) {
        ....
    }
}
```

W powyższym listingu wyrażenie `args` określa, że egzemplarz `FixedDepositDetails` przekazywany metodzie docelowej jest dostępny dla metody `log()` — czyli rady — za pomocą parametru `FixedDepositDetails`. Ponieważ wyrażenie `args` dostarcza metodę `log()` wraz z egzemplarzem obiektu `FixedDepositDetails`, metoda `log()` została zmodyfikowana w taki sposób, aby akceptowała dodatkowy argument typu `FixedDepositDetails`.

Desygatory punktu przecięcia, takie jak `execution`, `args`, `within`, `this`, `target` itd., są zdefiniowane przez AspectJ. Spring AOP definiuje charakterystyczny dla tego frameworku desygator `bean`. Warto więc przeanalizować pokrótce desygator `bean` punktu przecięcia.

Desygator `bean`

Desygator `bean` służy do ograniczenia metod docelowych do określonego identyfikatora (lub nazwy) komponentu. Istnieje możliwość podania dokładnego identyfikatora lub nazwy komponentu, ewentualnie można wskazać wzorzec. Spójrz teraz na kilka przykładów użycia desygatora `bean`.

Przykład 1.

Na rysunku 9.8 został pokazany desygator `bean` wskazujący nazwę komponentu, którego metody są celem dla zdefiniowanej rady.

Nazwa lub identyfikator komponentu, którego
metody są celem dla zdefiniowanej rady

↓

```
@Pointcut(value = "bean(someBean)")
```

Rysunek 9.8. Desygator `bean` określa nazwę komponentu zawierającego metody, w których będzie stosowana rada

Powyższe wyrażenie dopasuje metody zdefiniowane przez komponent o nazwie `someBean`.

Przykład 2.

Na rysunku 9.9 został pokazany desygator bean określający, że rada będzie zastosowana w metodach komponentów, których nazwa rozpoczyna się od `someBean`.

Rada będzie zastosowana w metodach komponentów,
których nazwa rozpoczyna się od `someBean`

↓

```
@Pointcut(value = "bean(someBean*)")
```

Rysunek 9.9. *Desygator bean wskazujący, że rada będzie zastosowana w metodach komponentów, których nazwa rozpoczyna się od `someBean`*

Powyższe wyrażenie określa, że rada będzie zastosowana w metodach komponentów, których nazwa rozpoczyna się od `someBean`.

➔ Podobnie jak w przypadku innych desygatorów punktu przecięcia, istnieje możliwość połączenia desygatora bean z innymi za pomocą operatorów `&&` i `||`. Takie połączenie umożliwia tworzenie skomplikowanych wyrażeń.

Teraz przechodzimy do desygatorów, które przeprowadzają dopasowanie na podstawie adnotacji.

Desygatory oparte na adnotacjach

AspectJ dostarcza desygatory punktów przecięcia takie jak `@annotation`, `@target`, `@within` i `@args`, których można użyć w połączeniu ze Springiem AOP do wyszukania metod docelowych. Spójrz na kilka przykładów użycia tego rodzaju desygatorów.

Przykład 1.

Na rysunku 9.10 pokazujemy przykład użycia desygatora `@annotation`.

Rada będzie zastosowana dla metod oznaczonych
oferowaną przez Spring adnotacją `@Cacheable`

↓

```
@Pointcut(value = "@annotation(org.springframework.cache.annotation.Cacheable)")
```

Rysunek 9.10. *Przedstawiony desygator `@annotation` określa, że rada będzie zastosowana dla metod oznaczonych oferowaną przez Spring adnotacją `@Cacheable`*

Powyższe wyrażenie spowoduje dopasowanie metod oznaczonych oferowaną przez Spring adnotacją `@Cacheable`.

Przykład 2.

Na rysunku 9.11 pokazujemy przykład użycia desygatora `@target`.

Rada będzie zastosowana dla metod znajdujących się w obiekcie oznaczonym oferowaną przez Spring adnotacją @Component

`@Pointcut(value = "@target(org.springframework.stereotype.Component)")`

Rysunek 9.11. Przedstawiony desygnator `@target` określa, że rada będzie zastosowana dla metod oznaczonych oferowaną przez Spring adnotacją `@Component`

Powyższe wyrażenie spowoduje dopasowanie metod znajdujących się w obiekcie oznaczonym oferowaną przez Spring adnotacją `@Component`.

W tym podrozdziale przeanalizowaliśmy wybrane desygnatory zdefiniowane przez AspectJ. Trzeba koniecznie zwrócić uwagę na fakt, że *nie* wszystkie desygnatory zdefiniowane przez AspectJ są obsługiwane przez framework Spring AOP. Jeżeli w wyrażeniu punktu przecięcia użyjesz nieobsługiwanego desygnatora, wówczas Spring AOP zgłosi wyjątek `java.lang.IllegalArgumentException`. Na przykład użycie desygnatorów `call`, `set` i `get` w wyrażeniu punktu przecięcia spowoduje, że Spring AOP zgłosi wyjątek `java.lang.IllegalArgumentException`.

Przechodzimy teraz do analizy różnych typów rad oraz sposobów ich tworzenia.

9.5. Typy rad

Dotychczas w tym rozdziale przedstawiliśmy przykłady rad typu *before*. Tego rodzaju rada jest tworzona przez określenie metody aspektu adnotacją `@Before` (patrz: listingi 9.1, 9.6 i 9.7). Inne typy rad, które można utworzyć, to *after*, *after returning*, *after throwing* i *around*.

IMPORT **Rozdział09/ch09-aop-advice**. (Ten projekt pokazuje aplikację MyBank, w której użyto różnego rodzaju rad. Aby uruchomić tę, aplikację wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Przechodzimy teraz do przeanalizowania funkcji i sposobów tworzenia różnego rodzaju rad.

Rada before

Rada *before* jest wykonywana przed wykonaniem metody docelowej. Jeżeli rada *before* nie zgłosi wyjątku, wówczas metoda docelowa *zawsze* będzie wywołana. Istnieje możliwość kontrolowania, czy metoda docelowa ma zostać wykonana — do tego celu używa się rady *around*, którą omówimy w dalszej części rozdziału.

Jak wcześniej wspomnieliśmy, adnotacja AspectJ `@Before` jest używana do wskazania rady jako typu *before*.

Metoda oznaczona adnotacją `@Before` może zdefiniować jej pierwszy argument jako typu `JoinPoint`. Argument typu `JoinPoint` można użyć wewnątrz rady do pobrania informacji o metodzie docelowej. Na przykład w listingu 9.1 pokazano, że egzemplarz `JoinPoint` może być użyty do pobrania nazwy klasy obiektu docelowego oraz argumentów przekazanych metodzie docelowej.

Rada after returning

Rada *after returning* jest wykonywana *po* zakończeniu działania metody docelowej. Warto zwrócić uwagę, że nie zostanie wykonana, jeśli metoda docelowa zgłosi wyjątek. Rada ta jest oznaczona adnotacją `AspectJ @AfterReturning`. Może uzyskać dostęp do wartości zwrotnej metody docelowej, a następnie zmodyfikować ją przed zwrotem tej wartości do obiektu wywołującego radę.

Klasa `SampleAspect` projektu *ch09-aop-advice*s przedstawia aspekt AOP. W listingu 9.8 pokazano, że klasa definiuje radę *after returning* wyświetlającą wartość zwróconą przez metodę `createBankAccount()` klasy `BankAccountService`.

Listing 9.8. Klasa `SampleAspect` przedstawiająca użycie rady *after returning*

Projekt: *ch09-aop-advice*

Położenie pliku: *src/main/java/sample/spring/chapter09/bankapp/aspects*

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.annotation.AfterReturning;
.....
@Aspect
public class SampleAspect {
    private Logger logger = Logger.getLogger(SampleAspect.class);

    @Pointcut(value = "execution(*
        sample.spring..BankAccountService.createBankAccount(..)")
    private void createBankAccountMethod() {}

    @AfterReturning(value = "createBankAccountMethod()", returning = "aValue")
    public void afterReturningAdvice(JoinPoint joinPoint, int aValue) {
        logger.info("Wartość zwrócona przez metodę "
            + joinPoint.getSignature().getName()
            + " wynosi " + aValue);
    }
    .....
}
```

W powyższym listingu metoda `afterReturningAdvice()` przedstawia radę typu *after returning*. Wyrażenie punktu przecięcia określone przed adnotacją `@Pointcut` definiuje punkt złączenia z metodą `createBankAccount()` klasy `BankAccountService`. Dwie kropki (`..`) w wyrażeniu `execution` wskazują, że pakiet `sample.spring` i znajdujące się w nim pakiety będą przeszukiwane w celu znalezienia typu `BankAccountService`.

W listingu 9.8 metoda `afterReturningAdvice()` klasy `SampleAspect` jest wywoływana po metodzie `createBankAccount()` klasy `BankAccountService`. Atrybut `returning` adnotacji `@AfterReturning` określa nazwę, pod którą wartość zwrotna metody docelowej będzie dostępna dla rady. W omawianym listingu wartość zwrotna metody `createBankAccount()` jest dostępna dla metody `afterReturningAdvice()` za pomocą argumentu `aValue`. Typ argumentu `aValue` został ustalony jako `int`, ponieważ metoda `createBankAccount()` zwraca wartość w postaci liczby całkowitej. Trzeba w tym miejscu dodać, że po zdefiniowaniu atrybutu `returning` rada będzie stosowana jedynie do metod

zwracających wartość wskazanego typu. Jeżeli rada *after returning* zostanie zastosowana w metodach zwracających różne typy wartości (w tym także void), wówczas typ argumentu wartości zwrotnej można określić jako `Object`.

Jak widać w listingu 9.8, metoda oznaczona adnotacją `@AfterReturning` może zdefiniować jej pierwszy argument jako typu `JoinPoint`, aby uzyskać dostęp do informacji o metodzie docelowej.

Rada *after throwing*

Rada *after throwing* jest wykonywana, gdy metoda docelowa zgłosi wyjątek. Może uzyskać dostęp do wyjątku zgłaszanego przez metodę docelową. Rada ta jest oznaczana adnotacją `AspectJ @After` ➔ `Throwing`.

W listingu 9.9 przedstawiamy przykład rady *after throwing* wykonywanej po zgłoszeniu wyjątku przez metodę docelową.

Listing 9.9. Klasa `SampleAspect` przedstawiająca użycie rady *after throwing*

Projekt: `ch09-aop-advice`

Położenie pliku: `src/main/java/sample/spring/chapter09/bankapp/aspects`

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.annotation.AfterThrowing;
.....
@Aspect
public class SampleAspect {
    private Logger logger = Logger.getLogger(SampleAspect.class);
    .....
    @Pointcut(value = " execution(* sample.spring..FixedDepositService.*(..)) ")
    private void exceptionMethods() {}
    .....
    @AfterThrowing(value = "exceptionMethods()", throwing = "exception")
    public void afterThrowingAdvice(JoinPoint joinPoint, Throwable exception) {
        logger.info("Wyjątek zgłoszony przez " + joinPoint.getSignature().getName()
            + " Typ zgłoszonego wyjątku : " + exception);
    }
}
```

W powyższym listingu metoda `afterThrowingAdvice()` klasy `SampleAspect` przedstawia radę *after throwing*. Metoda `afterThrowingAdvice()` jest wykonywana, gdy zostanie zgłoszony wyjątek przez dowolną metodę obiektu `FixedDepositService`. W omawianym listingu atrybut `throwing` adnotacji `@AfterThrowing` określa nazwę, pod jaką wyjątek zgłoszony przez metodę docelową będzie dostępny dla metody `afterThrowingAdvice()`. Ponieważ wartością atrybutu `throwing` jest `exception`, wyjątek będzie przekazywany metodzie `afterThrowingAdvice()` za pomocą argumentu o nazwie `exception`. Zwróć uwagę na fakt, że typ argumentu `exception` to `java.lang.Throwable`. Oznacza to, że metoda `afterThrowingAdvice()` będzie wykonywana dla wszystkich wyjątków zgłoszonych przez metodę docelową. Jeżeli chcesz, aby metoda `afterThrowingAdvice()` była wykonywana jedynie po zgłoszeniu przez metodę docelową wyjątku określonego typu, zmień typ argumentu `exception`.

Na przykład jeśli metoda `afterThrowingAdvice()` ma być wykonywana jedynie po zgłoszeniu przez metodę docelową wyjątku `java.lang.IllegalStateException`, wówczas należy podać `java.lang.↵IllegalStateException` jako typ argumentu `exception`.

Jak widać w listingu 9.9, metoda oznaczona adnotacją `@AfterThrowing` może zdefiniować jej pierwszy argument jako typu `JoinPoint`, aby uzyskać dostęp do informacji o metodzie docelowej.

Rada after

Rada *after* jest wykonywana po metodzie docelowej niezależnie od sposobu jej zakończenia — normalnie lub zgłoszeniem wyjątku. Rada ta jest oznaczana adnotacją `AspectJ @After`.

W listingu 9.10 przedstawiamy przykład rady *after* wykonywanej dla metody `createBankAccount()` klasy `BankAccountService` oraz dla metod zdefiniowanych przez interfejs `FixedDepositService`.

Listing 9.10. Klasa `SampleAspect` przedstawiająca użycie rady *after*

Projekt: *ch09-aop-advice*

Położenie pliku: *src/main/java/sample/spring/chapter09/bankapp/aspects*

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.annotation.After;
.....
@Aspect
public class SampleAspect {
    private Logger logger = Logger.getLogger(SampleAspect.class);
    @Pointcut(value = "execution(*
        sample.spring..BankAccountService.createBankAccount(..)")
    private void createBankAccountMethod() {}

    @Pointcut(value = "execution(* sample.spring..FixedDepositService.*(..)")
    private void exceptionMethods() {}
    .....
    @After(value = "exceptionMethods() || createBankAccountMethod()")
    public void afterAdvice(JoinPoint joinPoint) {
        logger.info("Rada after wykonana dla metody "
            + joinPoint.getSignature().getName());
    }
}
```

W powyższym listingu metoda `afterAdvice()` klasy `SampleAspect` przedstawia radę *after*. Metoda `afterAdvice()` jest wykonywana dopiero po zakończeniu działania metody docelowej. Zwróć uwagę na fakt, że atrybut `value` adnotacji `@After` używa operatora `||` w celu połączenia wyrażeń punktu przecięcia przedstawianych przez metody `createBankAccountMethod()` i `exceptionMethods()`. W ten sposób powstaje nowe wyrażenie punktu przecięcia.

Jak widać w listingu 9.10, metoda oznaczona adnotacją `@After` może zdefiniować jej pierwszy argument jako typu `JoinPoint`, aby uzyskać dostęp do informacji o metodzie docelowej.

Rada around

Rada *around* jest wykonywana zarówno *przed*, jak i *po* wykonaniu metody docelowej. W przeciwieństwie do innych rad może kontrolować, czy metoda docelowa w ogóle zostanie wykonana. Rada ta jest oznaczona adnotacją `AspectJ @Around`.

W listingu 9.11 przedstawiamy przykład rady *around* zdefiniowanej przez klasę `SampleAspect` w projekcie *ch09-aop-advice*.

Listing 9.11. Klasa `SampleAspect` przedstawiająca użycie rady *around*

Projekt: *ch09-aop-advice*

Położenie pliku: *src/main/java/sample/spring/chapter09/bankapp/aspects*

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.springframework.util.StopWatch;
.....
@Aspect
public class SampleAspect {
    .....
    @Around(value = "execution(* sample.spring.*Service.*(..))")
    public Object aroundAdvice(ProceedingJoinPoint pjp) {
        Object obj = null;
        StopWatch watch = new StopWatch();
        watch.start();
        try {
            obj = pjp.proceed();
        } catch (Throwable throwable) {
            // Przeprowadzenie wszelkich niezbędnych działań.
        }
        watch.stop();
        logger.info(watch.prettyPrint());
        return obj;
    }
}
```

W powyższym listingu metoda `aroundAdvice()` przedstawia radę *around*. Argument `ProceedingJoinPoint` metody `aroundAdvice()` jest przeznaczony do kontrolowania, czy metoda docelowa w ogóle zostanie wykonana. Trzeba koniecznie zwrócić uwagę na fakt, że argument `ProceedingJoinPoint` *musi* być pierwszym argumentem przekazanym radzie *around*. Kiedy wykonujesz metodę `proceed()` klasy `ProceedingJoinPoint`, metoda docelowa będzie wywołana. Oznacza to, że jeśli nie wywołasz metody `proceed()` klasy `ProceedingJoinPoint`, wówczas metoda docelowa *nie* zostanie wykonana. Po przekazaniu `Object[]` metodzie `proceed()` wartości znajdujące się w `Object[]` są przekazywane jako argumenty metody docelowej. Gdy rada *nie* wywoła metody docelowej, sama rada może zwrócić wartość.

Ponieważ metoda docelowa jest wywoływana jedynie po wywołaniu metody `proceed()` klasy `ProceedingJoinPoint`, rada *around* pozwala na wykonywanie zadań *przed* wywołaniem metody docelowej i *po* nim oraz na współdzielenie informacji między wspomnianymi zadaniami. W listingu 9.11 metoda `aroundAdvice()` rejestruje ilość czasu niezbędnego do wykonania metody docelowej.

Metoda `aroundAdvice()` uruchamia stoper (przedstawiany przez obiekt `StopWatch` klasy Springa) przed wywołaniem metody `proceed()` klasy `ProceedingJoinPoint`. Zatrzymanie stopera następuje po wywołaniu metody `proceed()` klasy `ProceedingJoinPoint`. Metoda `prettyPrint()` klasy `StopWatch` jest używana do wyświetlenia czasu potrzebnego na wykonanie metody docelowej.

Jeżeli zachodzi potrzeba modyfikacji wartości zwróconej przez metodę docelową, należy wartość zwrótną metody `proceed()` klasy `ProceedingJoinPoint` rzutować na typ zwrrotny metody docelowej, a następnie ją zmodyfikować. Metoda wywołująca będzie widziała wartość zwracaną przez metodę. Dlatego typ wartości zwrótny metody rady *trzeba* określić jako `Object` lub jako typ wartości zwracanej przez metodę docelową. Metoda rady ma możliwość zwrócenia wartości zwracanej przez metodę docelową lub zwrócenia zupełnie innej wartości. Na przykład zamiast wywoływać metodę docelową rada *around* może przeanalizować argument (lub argumenty) przekazywany metodzie docelowej i zwrócić wartość pochodzącą z bufora, jeśli istnieje w nim element odpowiadający przekazanemu zestawowi argumentów.

Dotychczas analizowaliśmy przykłady pokazujące tworzenie aspektów w stylu adnotacji `AspectJ`. Przechodzimy teraz do przykładów użycia zwykłych komponentów Springa jako aspektów AOP.

9.6. Spring AOP, czyli styl schematu XML

W schemacie w stylu XML zwykły komponent Springa działa w charakterze aspektu. Metoda zdefiniowana w aspekcie jest za pomocą oferowanego przez Spring schematu aop powiązana z typem rady i wyrażeniem punktu przecięcia.

IMPORT **Rozdział 09/ch09-aop-xml-schema.** (Ten projekt jest taki sam jak przedstawiony wcześniej *ch09-aop-advice*, z wyjątkiem faktu, że klasa `SampleAspect` w projekcie *ch09-aop-xml-schema* jest prostą klasą Javy, która nie używa adnotacji `AspectJ`).

W listingu 9.12 została przedstawiona klasa `SampleAspect` projektu *ch09-aop-xml-schema* definiującego rady.

Listing 9.12. Klasa `SampleAspect`

Projekt: *ch09-aop-xml-schema*

Położenie pliku: *src/main/java/sample/spring/chapter09/bankapp/aspects*

```
package sample.spring.chapter09.bankapp.aspects;
.....
public class SampleAspect {
    .....
    public void afterReturningAdvice(JoinPoint joinPoint, int aValue) {
        logger.info("Wartość zwrócona przez metodę "
            + joinPoint.getSignature().getName()
            + " ma wartość " + aValue);
    }

    public void afterThrowingAdvice(JoinPoint joinPoint, Throwable exception) {
        logger.info("Wyjątek zgłoszony przez " + joinPoint.getSignature().getName()
            + " Typ wyjątku to : " + exception);
    }
    .....
}
```


W powyższym listingu widać, że klasa `SampleAspect` definiuje metody przedstawiające rady AOP. Zwróć uwagę na fakt, że klasa `SampleAspect` *nie* jest oznaczona adnotacją `@Aspect`, a jej metody nie są oznaczone adnotacjami takimi jak `@After`, `@AfterReturning` itd.

Zobaczmy, jak element `<config>` schematu aop oferowanego przez Spring jest używany do konfiguracji zwykłego komponentu Springa jako aspektu AOP.

Konfiguracja aspektu AOP

W stylu schematu XML konfiguracja dotycząca AOP znajduje się w elemencie `<config>` schematu aop oferowanego przez Spring. Sam aspekt AOP jest konfigurowany za pomocą elementu `<aspect>` znajdującego się w elemencie `<config>`.

W listingu 9.13 pokazujemy sposób konfiguracji klasy `SampleAspect` za pomocą elementów `<config>` i `<aspect>`.

Listing 9.13. Plik `applicationContext.xml` przedstawiający przykład użycia schematu aop

Projekt: `ch09-aop-xml-schema`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans ..... xmlns:aop="http://www.springframework.org/schema/aop" ..... >
  .....
  <bean id="sampleAspect"
        class="sample.spring.chapter09.bankapp.aspects.SampleAspect" />

  <aop:config proxy-target-class="false" expose-proxy="true">
    <aop:aspect id="sampleAspect" ref="sampleAspect">
      .....
    </aop:aspect>
  </aop:config>
</beans>
```

Ponieważ element `<config>` opiera się na automatycznym proxy, element `<config>` definiuje atrybuty `proxy-target-class` i `expose-proxy`. Jak zapewne pamiętasz, te same atrybuty są definiowane przez element `<aspectj-autoproxy>` schematu aop oferowanego przez Spring. W podrozdziale 9.3 znajdziesz więcej informacji na temat atrybutów `proxy-target-class` i `expose-proxy`.

W przedstawionym powyżej listingu 9.13 definicja komponentu `sampleAspect` określa klasę `SampleAspect` jako komponent. Element `<aspect>` konfiguruje komponent `sampleAspect` jako aspekt AOP. Atrybut `id` elementu `<aspect>` określa unikalny identyfikator aspektu, natomiast atrybut `ref` wskazuje komponent Springa, który ma zostać skonfigurowany jako aspekt AOP.

Skoro mamy już skonfigurowany aspekt AOP, zobaczmy teraz, jak można mapować metody zdefiniowane w aspekcie AOP na różne typy rad i wyrażen punktów przecięcia.

Konfiguracja rady

Radę można skonfigurować za pomocą jednego z następujących elementów znajdujących się w elemencie `<aspect>`: `<before>` (do konfiguracji rady typu *before*), `<after-returning>` (do konfiguracji rady typu *after returning*), `<after-throwing>` (do konfiguracji rady typu *after throwing*), `<after>` (do konfiguracji rady typu *after*) i `<around>` (do konfiguracji rady typu *around*).

Zobaczmy więc, jak rady zdefiniowane w klasie `SampleAspect` projektu *ch09-aop-xml-schema* zostały skonfigurowane w pliku XML kontekstu aplikacji.

Konfiguracja rady after returning

Rysunek 9.12 pokazuje, jak metoda `afterReturningAdvice()` klasy `SampleAspect` została skonfigurowana jako rada typu *after returning* za pomocą elementu `<after-returning>`.

```
public void afterReturningAdvice(JoinPoint joinPoint, int aValue) {  
    logger.info("Value returned by " + joinPoint.getSignature().getName()  
        + " method is " + aValue);  
}  
  
↓ ↓  
  
<aop:after-returning method="afterReturningAdvice" returning="aValue"  
pointcut="execution(*sample.spring..BankAccountService.createBankAccount(..))" />
```

Rysunek 9.12. Metoda `afterReturningAdvice()` klasy `SampleAspect` została skonfigurowana jako rada typu *after returning* z użyciem elementu `<after-returning>`

Atrybut `method` elementu `<after-returning>` określa nazwę metody, która ma być skonfigurowana jako rada typu *after returning*. Atrybut `returning` służy do tego samego celu co atrybut `returning` adnotacji `@AfterReturning` — wartość zwrótną metody docelowej udostępnia radzie. Atrybut `pointcut` wskazuje wyrażenie punktu przecięcia używane do wyszukania metod, dla których będzie zastosowana rada.

Konfiguracja rady after throwing

Rysunek 9.13 pokazuje, jak metoda `afterThrowingAdvice()` klasy `SampleAspect` została skonfigurowana jako rada typu *after throwing* za pomocą elementu `<after-throwing>`.

```
public void afterThrowingAdvice(JoinPoint joinPoint, Throwable exception) {  
    logger.info("Exception thrown by " + joinPoint.getSignature().getName()  
        + " Exception type is : " + exception);  
}  
  
↓ ↓  
  
<aop:after-throwing method="afterThrowingAdvice" throwing="exception"  
pointcut="execution(* sample.spring..FixedDepositService.*(..))" />
```

Rysunek 9.13. Metoda `afterThrowingAdvice()` klasy `SampleAspect` została skonfigurowana jako rada typu *after throwing* z użyciem elementu `<after-throwing>`

Atrybut `method` elementu `<after-throwing>` określa nazwę metody, która ma być skonfigurowana jako rada typu *after throwing*. Atrybut `throwing` służy do tego samego celu co atrybut `throwing` adnotacji `@AfterThrowing` — wyjątek zgłoszony przez metodę docelową zostaje udostępniony radzie. Atrybut `pointcut` wskazuje wyrażenie punktu przecięcia używane do wyszukania metod, dla których będzie zastosowana rada.

Inne typy rad (*before*, *after* i *around*) są konfigurowane w dokładnie taki sam sposób jak przedstawiliśmy powyżej dla rad *after returning* i *after throwing*.

Zobaczmy teraz różne sposoby powiązania wyrażenia punktu przecięcia z radą.

Powiązanie wyrażenia punktu przecięcia z radą

Elementy `<after>`, `<after-returning>`, `<after-throwing>`, `<before>` i `<around>` schematu `aop` oferowanego przez Spring definiują atrybut `pointcut`, którego można użyć do określenia wyrażenia punktu przecięcia powiązanego z radą. Jeżeli to samo wyrażenie punktu przecięcia ma być współdzielone przez wiele różnych rad, wówczas do jego zdefiniowania można użyć elementu `<pointcut>` znajdującego się w elemencie `<config>`.

W listingu 9.14 przedstawiamy przykład elementu `<pointcut>` użytego do zdefiniowania wyrażenia punktu przecięcia.

Listing 9.14. Plik `applicationContext.xml` przedstawiający użycie elementu `<pointcut>`

```
<beans ..... xmlns:aop="http://www.springframework.org/schema/aop" ..... >
  .....
  <bean id="sampleAspect"
        class="sample.spring.chapter09.bankapp.aspects.SampleAspect" />

  <aop:config proxy-target-class="false" expose-proxy="true">
    <aop:pointcut expression="execution(* sample.spring.*Service.*(..)"
                  id="services" />

    <aop:aspect id="sampleAspect" ref="sampleAspect">
      <aop:after method="afterAdvice" pointcut-ref="services" />
      <aop:around method="aroundAdvice" pointcut-ref="services"/>
    </aop:aspect>
  </aop:config>
</beans>
```

W powyższym listingu element `<pointcut>` określa wyrażenie punktu przecięcia. Atrybut `expression` wskazuje wyrażenie, natomiast atrybut `id` określa unikalny identyfikator dla tego wyrażenia. Zdefiniowane przez element `<pointcut>` wyrażenie punktu przecięcia jest używane przez atrybut `pointcut-ref` elementów `<after>`, `<after-returning>` itd. Na przykład w omawianym listingu elementy `<after>` i `<around>` używają atrybutu `pointcut-ref` w celu odwołania się do wyrażenia punktu przecięcia nazwanego `services`.

9.7. Podsumowanie

W tym rozdziale przedstawiliśmy koncepcję programowania zorientowanego aspektowo i pokazaliśmy, jak można użyć frameworku AOP w Springu do realizacji zagadnień przekrojowych w aplikacjach Springa. Zobaczyłeś, jak można tworzyć aspekty za pomocą stylu adnotacji AspectJ i schematu XML. Przeanalizowaliśmy temat tworzenia i konfiguracji różnego typu rad. Poruszyliśmy także temat wyrażen punktu przecięcia, które można stosować do dopasowania metod w aplikacji. Więcej wyczerpujących informacji na temat oferowanego przez Spring frameworku AOP znajdziesz w dokumentacji Springa. W kolejnym rozdziale dowiesz się, jak tworzyć aplikacje sieciowe za pomocą modułu Spring Web MVC.

A

ACL, access control list, 429, 440

adnotacja

- @ Scheduled, 268
- @AmountFormat, 422
- @Aspect, 280
- @Async, 267, 268
- @Autowired, 194–198
- @Bean, 216
- @Before, 280
- @Cacheable, 271, 272
- @CacheEvict, 271, 273
- @CachePut, 271, 274
- @Component, 191, 293
- @Configuration, 216
- @ConstructorProperties, 80–82
- @Controller, 314, 317
- @ControllerAdvice, 357
- @DependsOn, 203
- @ExceptionHandler, 333, 385, 414, 416
- @InitBinder, 354–357
- @Inject, 199
- @Lazy, 202
- @MatrixVariable, 395, 398, 399
- @ModelAttribute, 337–345
- @Named, 199
- @NotBlank, 367
- @PathVariable, 395
- @Pointcut, 286
- @PostConstruct, 161, 162
- @PostFilter, 455
- @PreAuthorize, 439, 453, 454
- @PreDestroy, 161, 162
- @Primary, 203
- @Qualifier, 198
- @Qualifier, 198, 199
- @RequestBody, 383, 384
- @RequestMapping, 314–318, 324, 340, 380–382, 408–413
- @RequestParam, 318, 326, 420
- @Required, 174
- @Resource, 201
- @ResponseBody, 382, 383, 394

- @ResponseStatus, 384
- @RolesAllowed, 439
- @Scheduled, 267, 412
- @Scope, 202
- @Secured, 437–439
- @SessionAttributes, 347, 348
- @Transactional, 238–241
- @Valid, 370
- @Value, 203–207
- JSR 303, 211, 213, 368
- RequestMappingHandlerMapping, 318
- Spring Web MVC, 316

adnotacje buforowania, 271

adres URI, 377

AOP, aspect-oriented programming, 279, 282

API

- Hibernate, 232
- Validation, 362

aplikacja MyBank, 22, 28, 221, 275, 304

aplikacje

- sieciowe, 314
- w stylu REST, 375

argument

- JoinPoint, 281
- typu HttpServletRequest, 324

argumenty konstruktora, 82

aspekt AOP, 299

asynchroniczne

- odbieranie komunikatów JMS, 255
- przetwarzanie żądań, 407
- wykonywanie metod, 245

atrybut

- access, 433
- defaultValue, 330
- depends-on, 123, 128
- destroy-method, 157, 158, 223
- error-page, 448
- expression, 301
- init-method, 157
- method, 320
- name, 34, 79
- proxy-target-class, 283
- ref, 121
- required, 329

atrybut
 scope, 55
 secured-annotations, 438
 transaction-manager, 240
 type, 194, 217
 value, 241, 280, 329
 xmlns, 100
 automatyczne wiązanie zależności, 145, 150, 194, 198

B

baza danych, 221
 bezargumentowy konstruktor klasy, 35
 bezpieczeństwo, 24, 429, 458
 bezpieczeństwo egzemplarza obiektu domeny, 453
 biblioteka
 CGLIB, 141, 144
 form, 372
 znaczników, 421
 znaczników form, 371
 broker ActiveMQ, 247
 buforowanie, 27, 245, 269
 ACL, 451
 atomytów modelu, 347

C

CRUD, 375, 449

D

definicja komponentu PersonalBankingService, 52
 definiowanie
 komponentów, 48, 99
 ograniczeń, 366
 wartości, 92
 właściwości lokalnych, 186
 zależności, 33
 deklaracyjne zarządzanie transakcjami, 23, 238
 deskryptor
 wdrożenia aplikacji, 308
 args, 287, 290
 bean, 291
 execution, 287
 desygatory oparte na adnotacjach, 292
 DI, dependency injection, 17
 dodanie
 atomytów modelu, 337, 340
 nazw komponentów, 90

 odwołania do komponentu, 89
 wartości null, 91
 dołączanie danych, 337, 350–352, 358
 dopasowanie argumentu konstruktora, 73, 74, 79
 dostawca DaoAuthenticationProvider, 434
 dostęp do
 egzemplarza FactoryBean, 118
 obiektów, 36, 313
 parametrów żądania, 328
 usługi sieciowej, 386, 387
 zmiennych szablonu, 397
 dostosowanie
 komponentów, 155
 logiki inicjalizacji, 155
 dynamiczni odbiorcy komunikatów JMS, 253
 dziedziczenie definicji komponentu, 65–68, 80, 128

E

Eclipse IDE, 461
 edytor
 CustomCollectionEditor, 95
 CustomMapEditor, 96
 PropertyEditor, 353
 edytory właściwości, 83, 93, 97
 egzemplarz
 FactoryBean, 118
 JMS Session, 249
 element
 <access-denied-handler>, 448
 <annotation-driven>, 272, 316, 356, 398
 <around>, 301
 <aspect>, 299
 <aspectj-autoproxy>, 283, 299
 <authentication>, 436
 <bean>, 31, 122, 128, 403
 <beans>, 62
 <broker>, 247
 <component-scan>, 192
 <config>, 301
 <constant>, 110
 <constructor-arg>, 53, 73, 76
 <entry>, 106
 <exclude-filter>, 193
 <executor>, 264
 <filter-mapping>, 431
 <filter-name>, 431
 <form>, 372
 <form-login>, 433

<global-method-security>, 437, 439
 <http>, 433
 <include-filter>, 193
 <interceptor>, 403
 <intercept-url>, 433
 <jta-transaction-manager>, 242
 <list>, 87, 104, 122
 <listener-container>, 256
 <logout>, 436
 <lookup-method>, 138, 139
 <map>, 88–90, 105
 <pointcut>, 301
 <prop>, 87
 <properties>, 109
 <property>, 32, 33, 100, 121
 <property-path>, 111
 <property-placeholder>, 186
 <queue>, 249, 251
 <ref>, 403
 <remember-me>, 433
 <replaced-method>, 142
 <scheduled-tasks>, 266
 <scheduler>, 266
 <set>, 88, 107
 <xss-protection>, 434
 FieldRetrievingBean, 111
 ListFactoryBean, 104
 MapFactoryBean, 107
 PropertiesFactoryBean, 110
 PropertyPathFactoryBean, 112
 SetFactoryBean, 108
 elementy schematu util, 103

F

fabryka, 35, 46
 filtr DelegatingFilterProxy, 431
 filtrowanie klas komponentów, 192
 format
 JSON, 377
 XML, 66
 formatowanie, 417
 formularz, 359
 framework
 AOP, 279, 282
 Hibernate, 232
 Hibernate Validator, 367
 Spring Security, 431
 Spring Web MVC, 306

frameworki zbudowane, 37
 funkcje automatycznego wiązania, 150

H

harmonogram zadań, 263
 Hibernate, 231

I

identyfikacja
 komponentów, 191
 metody komponentu, 144
 obiektów, 28
 identyfikator lokaty, 437
 implementacja
 AmountFormatter, 422
 AnnotationFormatterFactory, 423
 AutowiredAnnotationBeanPostProcessor, 195
 BeanFactoryPostProcessor, 176, 177
 BeanFactoryPostProcessors, 181
 BeanPostProcessors, 181
 CacheManager, 270
 CommonsMultipartResolver, 425, 427
 Formatter, 420–422
 HttpMessageConverter, 383
 HttpMessageConverters, 394
 interfejsu Converter, 418
 PlatformTransactionManager, 242
 PropertyEditorRegistrar, 97
 RequiredAnnotationBeanPostProcessor, 174
 usługi sieciowej, 376, 379
 import projektu, 461
 informacja
 o listach, 442
 o lokatach, 441
 informacje ACL, 455
 inicjacja
 egzemplarzy, 114
 komponentu, 61
 inicjalizacja
 egzemplarza WebDataBinder, 358
 WebDataBinder, 355
 IntelliJ IDEA, 461
 interceptor, 403
 interfejs
 ApplicationContextAware, 136, 137
 BeanFactoryPostProcessor, 176
 BeanPostProcessor, 163, 165

interfejs

- Cache, 270
- CacheManager, 269–271
- Converter, 417, 418
- DependencyResolver, 168, 169
- DestructionAwareBeanPostProcessor, 175
- DisposableBean, 161
- FactoryBean, 113, 114, 172
- FixedDepositService, 438, 453–455
- InitializingBean, 161
- JavaMailSender, 261
- LocaleContextResolver, 405
- MailSender, 260
- MessageListener, 257
- MessageSource, 406
- MethodReplacer, 143
- MutableAcl, 458
- PlatformTransactionManager, 242
- ServletConfigAware, 313
- ServletContextAware, 313
- TaskExecutor, 263
- TaskScheduler, 265
- TransactionTemplate, 236
- Validator, 207, 363
- WebBindingInitializer, 355

internacjonalizacja, 401, 404

IoC, Inversion of Control, 17

J

JDBC, 23

JDO, Java Data Objects, 231

język wyrażeń SpEL, 206

JMS, Java Message Service, 26, 30

JMX, Java Management Extensions, 24

JNDI, 23

JPA, Java Persistence API, 231

JSON, JavaScript Object Notation, 375

JTA, Java Transaction API, 23, 241

K

katalog, 303

klasa

- AccountStatementServiceImpl, 151, 194
- AclAuthorizationStrategyImpl, 450
- AmountFormatAnnotationFormatterFactory, 423
- AmountFormatter, 421, 424

ApplicationConfigurer, 177

ApplicationContext, 58

AsyncRestTemplate, 392, 393

BankAccountDaoImpl, 228

BankAccountServiceImpl, 284, 285

BankApp, 35, 119, 275

BankAppConfiguration, 216, 218

BankAppWithHook, 159

BankDetails, 83, 84

BankStatement, 101

CGLIB, 284

CollectionTypesExample, 94

Configuration, 205

ControllerFactory, 71

CustomEditorConfigurer, 98

CustomerRegistrationServiceImpl, 146, 195

CustomerRequestService, 140, 141

CustomerRequestServiceContextAwareImpl, 137

CustomerRequestServiceImpl, 136, 139, 142, 147, 196

DatabaseOperations, 65

DataTypesExample, 86

DependencyResolutionBeanPostProcessor, 169, 170, 171

EmailMessageListener, 260

EmailMessageSender, 53

EventSenderFactoryBean, 115, 172

EventSenderSelectorServiceImpl, 123, 125

FileUploadController, 426

FixedDepositController, 19, 32, 324–331, 351, 369, 370, 379–381, 408, 419

FixedDepositDao, 39

FixedDepositDaoFactory, 44, 46, 50

FixedDepositDaoImpl, 156, 161, 225, 232

FixedDepositDetails, 208, 211, 332, 350, 367

FixedDepositDetailsValidator, 363, 365

FixedDepositJdbcDao, 50

FixedDepositMessageListener, 257

FixedDepositProcessorJob, 261, 262

FixedDepositService, 21, 25–27, 43

FixedDepositServiceImpl, 114, 124, 149, 156, 191, 209, 236, 250, 456–458

FixedDepositServiceImplJsr303, 215

FixedDepositValidator, 208

FixedDepositWSCClient, 387, 388, 390

FixedDespositController, 29

HttpEntity, 381

InstanceValidationBeanPostProcessor, 167

IOUtils, 387

- JavaMailSenderImpl, 258
- JdbcTemplate, 224, 225
- JmsTemplate, 248, 249, 253–255
- LocaleChangeInterceptor, 406
- LocaleContextHolder, 406
- LocalSessionFactoryBean, 231
- LocalValidatorFactoryBean, 214, 368
- LoggingAspect, 280, 286, 291
- MailSender, 261
- MyAuthFailureHandler, 448
- MyErrorHandler, 386
- MyMethodReplacer, 142
- MyPropertyEditorRegistrar, 357
- MyRequestHandlerInterceptor, 402
- NamedParameterJdbcTemplate, 226, 227
- PersonalBankingService, 48, 51, 53
- PersonalBankingServiceImpl, 70
- PropertyOverrideConfigurer, 188
- PropertySourcesPlaceholderConfigurer, 185, 188
- ResponseEntity, 380
- RestTemplate, 386, 392
- ResultContext, 410
- SampleAspect, 295, 296
- SampleBean, 76
- ServiceTemplate, 75, 81
- SimpleCacheManager, 271
- SimpleJdbcInsert, 228, 229
- SimpleMailMessage, 259
- SingletonTest, 58
- SingletonTest z JUnit, 56
- TransactionTemplate, 234, 252
- TransferFundsServiceImpl, 77
- UserRequestControllerImpl, 73
- WebDataBinder, 360
- klasy
 - aplikacji MyBank, 114
 - DAO, 66
 - JdbcTemplate, 225
 - POJO, 29
 - zależne, 40
- kolejność
 - atributu modelu, 362
 - inicjalizacji procesu, 123
 - wykonywania metod, 349
 - wywołania, 344
- komponent
 - configuration, 205
 - customerRegistrationService, 134, 135
 - dataSource, 449
 - LocalValidatorFactoryBean, 213
 - lookupStrategy, 449
 - PropertyOverrideConfigurer, 187
 - PropertySourcesPlaceholderConfigurer, 182, 186
 - SimpleUrlHandlerMapping, 307, 308
- komponenty
 - o zasięgu prototypu, 135, 160
 - o zasięgu singleton 58, 61, 62
 - wewnętrzne, 121
 - zarządzane, 25
- komunikat, 134, 245, 345, 350
 - błędu, 427
 - JMS, 246
- komunikaty JMS, 246
 - asynchroniczne odbieranie, 255
 - dynamiczni odbiorcy, 253
 - synchroniczne odbieranie, 255
 - wysyłanie, 246–249
- konfiguracja
 - ACL, 449
 - aspektu AOP, 299
 - asynchronicznego przetwarzania żądań, 407
 - brokera ActiveMQ, 247
 - CallableProcessingInterceptor, 417
 - CustomEditorConfigurer, 98
 - Eclipse IDE, 463
 - egzemplarza
 - ContextLoaderListener, 335
 - ThreadPoolTaskExecutor, 264
 - WebDataBinder, 353
 - filtru DelegatingFilterProxy, 431
 - implementacji
 - AnnotationFormatterFactory, 423
 - DeferredResultProcessingInterceptor, 417
 - Formatter, 422
 - interfejsu CacheManager, 271
 - interfejsu WebBindingInitializer, 355
 - JdbcMutableAclService, 449
 - JdbcTemplate, 225
 - JmsTemplate, 248, 253
 - JMS ConnectionFactory, 247
 - klasy, 231
 - komponentów, 65, 217
 - menedżera transakcji JTA, 242
 - radę after returning, 300
 - radę after throwing, 300
 - RestTemplate, 386

konfiguracja

- SessionFactory, 231
- SimpleCacheManager, 271
- SimpleMailMessage, 259
- ThreadPoolTaskScheduler, 266
- TransactionTemplate, 252, 235
- uwierzytelniania, 434
- WebBindingInitializer, 356
- właściwości, 85
- wykonawcy ThreadPoolTaskExecutor, 264
- zabezpieczenia żądania sieciowego, 432
- zabezpieczeń, 431, 437, 452
- zmiennej M2_REPO, 463
- źródła danych, 223, 224

kontekst

- aplikacji, 307
- aplikacji sieciowej, 335

kontener IoC, 18**kontener podstawowy, 17****kontroler FixedDepositController, 344, 359–361, 365****konwersja**

- komunikatu, 254
- typu, 417, 420

L**liczba wątków, 264****limit czasu dla żądań, 416****lista lokat, 430, 441****logika inicjalizacji, 155, 156****logowanie, 429, 441****lokalizacja aplikacji sieciowej, 404****M****mapowanie**

- metod HTTP, 380
- obiektoowo relacyjne, 221
- wyrażenia execution, 287
- żądań
 - akceptowane typy MIME odpowiedzi, 323
 - metody HTTP, 320
 - parametry żądania, 321
 - ścieżka dostępu żądania, 319
 - wartość nagłówka żądania, 323

MBean, managed bean, 25**menedżery transakcji JTA, 242****metadane konfiguracyjne, 20, 30****metoda**

- addCallback(), 394
- addPermission(), 456, 457
- afterCompletion(), 402
- afterReturningAdvice(), 300
- closeFixedDeposit(), 326
- convert(), 418
- convertAndSend(), 254
- createBankAccount(), 276, 282, 284
- createFixedDeposit(), 24, 157, 237, 240, 282
- createMessage(), 251
- currentProxy(), 283
- doInTransaction(), 237
- editFixedDeposit(), 348
- exchange(), 388, 389
- findFixedDepositsByBankAccount(), 246, 276
- getBeanDefinition(), 178
- getBeanDefinitionNames(), 178
- getCurrentSession(), 233
- getErrorCount(), 210
- getFieldTypes(), 423
- getFixedDeposit(), 277
- getFixedDepositDao(), 47, 51
- getFixedDepositDetails(), 27
- getFixedDepositFromCache(), 277
- getFixedDepositList(), 379–390, 412
- getForEntity(), 389
- getObject(), 115–117
- getObjectType(), 115
- getValue(), 179
- handleError(), 387
- handleException(), 385, 391, 416
- handleFileUpload(), 427
- handleRequest(), 305
- hasPrototypeDependency(), 178–180
- init(), 56
- isPrototype(), 180
- isSingleton(), 115, 178
- listFixedDeposits(), 340, 343, 408
- log(), 281
- myExceptionHandler(), 334
- obtainCustomerRegistrationDetails(), 196
- onMessage(), 257
- openFixedDeposit(), 328, 348, 390–394, 414
- openInvalidFixedDeposit(), 392
- perform(), 323
- perform(), 321, 341
- postForEntity(), 389

postHandle(), 402
 postProcessBeforeInitialization(), 170
 preHandle(), 401
 prepare(), 263
 processResults(), 411–415
 queryForObject(), 228
 reimplement(), 143
 reject(), 209
 send(), 251, 261, 262
 sendEmail(), 262
 setDisallowedFields(), 360
 submitRequest(), 27, 140, 141
 supports(), 363
 testInstances(), 63
 testReference, 57
 testSingletonScope(), 58
 testSingletonScopePerBeanDef(), 60
 update(), 226
 validate(), 210, 364, 365, 368
 viewFixedDepositDetails(), 419
 withTableName(), 229

metody

czyszczące, 158, 160
 docelowe, 280
 fabryki, 81
 HTTP, 320, 375
 inicjalizacji, 160
 komponentu MyBeanPostProcessor, 164
 przeciążone, 145
 typu setter, 47, 50

moduł, 17

ACL, 440, 442
 JDBC, 222, 224
 Spring Web MVC, 303

moduły frameworku Spring Security, 431

modyfikacja definicji komponentu, 176

MyBatis, 231

N

nagłówek żądania, 323

narzędzie Maven, 304

nazwa

argumentu, 79
 argumentu konstruktora, 79
 komponentu, 90, 152
 parametru żądania, 329

niejawne zależności, 124, 128

O

obiekt

BindingResult, 361
 Callable, 408
 DeferredResult, 409–412
 EhCacheBasedAclCache, 451
 HttpHeaders, 382
 HttpServletResponse, 382
 HttpSession, 349
 JDBC Connection, 22
 proxy, 282
 Queue, 411, 413
 RequestToViewNameTranslator, 346
 SecurityExpressionHandler, 452
 ServletConfig, 313
 ServletContext, 313
 weryfikatora, 303

obiekty DAO, 224

obsługa

błędów w odpowiedzi HTTP, 386
 HTML5, 372
 interceptorów, 401, 403
 JMS, 26
 JMX, 24
 JTA, 241
 konwersji typu, 417
 lokat, 376
 przekazywania plików, 425
 wyjątków, 333
 zadań, 264

odbieranie komunikatów JMS, 255

odpowiedź HTTP, 380

odwołanie do komponentu, 89

ograniczenia, 211, 213

ograniczenia automatycznego wiązania, 152

określanie ograniczeń, 211

opcja debug, 82

operacje CRUD, 375, 449

operacje użytkownika, 345

opracowanie aplikacji sieciowej, 314

ORM, object relational mapping, 221

P

parametry żądania, 321, 326

plik

appConfig.properties, 127

applicationContext.xml, 31, 55, 72, 183–188,
 248, 386, 393

plik

- applicationContext-security.xml, 434, 446, 450, 452
- bankapp-config.xml, 326, 405
- createFixedDepositForm.jsp, 330, 332, 371
- database.properties, 183, 185, 189
- email.properties, 259
- emailtemplate.properties, 260
- fileupload-config.xml, 426
- FixedDepositController.java, 372
- fixedDepositList.jsp, 407, 435
- helloworld.js, 306
- list.jsp, 340
- myapp-config.xml, 307
- pom.xml, 465
- uploadForm.jsp, 425
- web.xml, 308, 309
- webservice.properties, 183–186, 189

pliki

- kodu źródłowego, 305
- konfiguracyjne, 305
- XML, 335

pobieranie, 135

pobieranie atrybutów modelu, 337, 342

POJO, plain old Java objects, 20

programowa konfiguracja komponentów, 216

programowanie

- interfejsów, 39, 42
- klas, 41
- oparte na adnotacjach, 191
- zorientowane aspektowo, AOP, 279

programowe zarządzanie transakcjami, 234, 252

projekt ch01-bankapp-xml, 37

prosta aplikacja, 28

protokół SMTP, 259

prototyp, 63

proxy, 283

przechowywanie zdarzeń, 113

przechwytywanie żądań asynchronicznych, 417

przeciążanie metod, 145

przekazywanie

- odwołań do komponentów, 73
- parametrów żądania, 327
- plików, 425, 427
- wartości, 73

przestrzeń nazw

- c, 100

- p, 99

przesyłanie plików, 401

przetwarzanie żądań, 310, 343

przetwarzanie żądań asynchronicznych, 401, 407

punkt

- przecięcia, pointcut, 279, 286
- przecięcia z radą, 301
- złączenia, join points, 279

R

rada, advice, 279

- after, 296
- after returning, 294, 300
- after throwing, 295, 300
- argumentów, 290
- around, 297
- before, 293

rejestracja edytorów właściwości, 97

REST, Representational State Transfer, 375

rola

- ROLE_ADMIN, 436
- ROLE_CUSTOMER, 436
- ROLE_XYZ, 439

rozwiązywanie zależności komponentu, 168

S

schemat

- mvc, 398
- security, 434
- spring-security-3.2.xsd, 432
- util, 103, 110

sekwencja

- działań, 234
- zdarzeń, 131, 135

serwer Tomcat, 463, 464

serwlet DispatcherServlet, 310, 311

singleton, 55, 58

SMTP, Simple Mail Transfer Protocol, 259

specyfikacja

- JSR 250, 201
- JSR 303, 214

SpEL, Spring Expression Language, 191, 204

Spring Security, 429

stała

- HttpStatus.BAD_REQUEST, 385
- HttpStatus.OK, 385

statyczna metoda fabryki, 44

struktura katalogów, 303, 304

styl

- adnotacji, 280

- schematu XML, 298

szablon adresu URI, 396, 397

Ś

ścieżka dostępu żądania, 319

środowisko

- IDE

 - Eclipse, 15

 - IntelliJ IDEA, 15

- pracy, 461

T

tabela

- ACL_CLASS, 443

- ACL_ENTRY, 444

- ACL_OBJECT_IDENTITY, 443

- ACL_SID, 443

- AUTHORITIES, 446

- USERS, 446

tabele bazy danych, 221

tablica, 92

technologia JMX, 25

testowanie klasy zależnej, 41

transakcje

- globalne, 23

- JTA, 242

tryb osadzony, 465

tworzenie

- bazy danych, 222

- egzemplarza kontenera, 35

- EventSenderFactoryBean, 173

- implementacji PropertyEditorRegistrar, 97

- komponentów, 44

- komponentu, 46

- kontenera, 135

- metadanych konfiguracyjnych, 30

- obiektów DAO, 224

- proxy, 283

- usług sieciowych, 375

typy

- kolekcji, 86

- MIME odpowiedzi, 323

- rad, 293

- wartości zwrotnych, 325

U

uniedostępnienie komponentu, 150

uprawnienie ADMINISTRATION, 459

URI, Uniform Resource Identifier, 375

uruchomienie serwera Tomcat 7, 465

usługa sieciowa

- FixedDepositWS, 376, 379, 386–390

- RESTful, 375

usuwanie

- atributów modelu, 349

- komponentu, 155

- obiektu, 160

uwierzelnianie użytkownika, 429, 434, 446, 459

W

wartości argumentów konstruktora, 102

wartość

- atributu type, 194, 217

- byName, 148

- byType, 146

- constructor, 147

- default, 150

- no, 150

- null, 91

wątek, 264

wbudowane

- edytory właściwości, 93

- implementacje PropertyEditor, 85

wdrażanie projektów, 461

wdrożenie projektu sieciowego, 465

weryfikacja, 330

- atributów modelu, 363–370

- danych, 337

- egzemplarzy komponentu, 165

- obiektów, 207

- obiektu, 215, 367, 369

- w Springu, 362

wiadomości e-mail, 245, 258

wiązania zależności, 145, 194

- według nazwy, 201

widok, 308

widok Servers, 465

właściwości

- komponentu, 82

- lokalne, 186

właściwość

- annotatedClasses, 232
- basenames, 406
- cacheManager, 451
- corePoolSize, 264
- deferredResult, 411
- depositAmount, 332
- eventSenderClass, 127
- FixedDepositDao, 175
- localOverride, 184
- locations, 183
- propagationBehaviorName, 235
- rejectedExecutionHandler, 264
- tenure, 332
- url, 184
- webServiceUrl, 184

włączenie

- adnotacji buforowania, 272
- adnotacji Spring Web MVC, 316
- opcji debug, 82

wstrzykiwanie zależności, 35, 47, 50–53, 121

wstrzyknięcie egzemplarza, 46, 119

wybór

- Javy SDK, 464
- zasięgu komponentu, 64

wyjątek, 333

- java.lang.NullPointerException, 67
- ValidationException, 416

wykonawca

- ThreadPoolExecutor, 263
- ThreadPoolTaskExecutor, 265

wykonywanie

- asynchroniczne, 263
- zadań, 265

wymagania aplikacji

- MyBank, 221, 245
- sieciowej, 317, 404

wymagany parametr żądania, 329

wyrażenia

- punktu przecięcia, 286
- regularne, 397

wyrażenie

- execution, 288
- określające wzorzec metody, 289
- punktu przecięcia, 280, 288, 290
- SpEL, 204

wysyłanie

- komunikatów
 - JMS, 246, 248
 - JMS w transakcji, 249
- wiadomości e-mail, 258, 260

wyświetlanie komunikatu, 306

wywołanie weryfikacji atrybutów, 370

wzorzec nazwy komponentu, 152

Z

zabezpieczanie

- aplikacji, 429, 431
- egzemplarzy FixedDepositDetails, 440
- metod, 436
- strony JSP, 435
- zadań sieciowych, 432, 447

zabezpieczenia

- dla metod komponentu, 438
- domyślne, 439
- na poziomie metody, 24

zagadnienia przekrojowe, cross-cutting concern, 279

zalety frameworka, 21

zależności

- komponentu, 129
 - o zasięgu prototypu, 132
 - o zasięgu singleton, 132
- między komponentami, 123
- między modułami, 18
- niejawne, 127
- o zasięgu prototypu, 131
- obiektów, 28

zapisywanie danych, 225

zapytanie SQL, 447

zarządzanie

- buforem, 451
- informacjami ACL, 455
- transakcjami, 233, 234
 - deklaracyjne, 238
 - programowe, 234
- transakcjami JMS, 252

zasięg komponentu, 54, 64

- globalSession, 312
- request, 312
- session, 312
- singleton, 129

zasób transakcyjny, 22

zmienna

 M2_REPO, 463

 środowiskowa CLASSPATH, 116

znacznik, *Patrz* element

znaczniki biblioteki form, 373

zwrot obiektu DeferredResult, 409, 411

Ż

żądania

 asynchroniczne, 416

 HTTP, 377

żądanie, 310

 Atrybut pathVar, 399

 DELETE, 378

 GET, 378

 POST, 378

 PUT, 378

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA



Helion SA

Pojawienie się na rynku Spring Framework było jednym z punktów zwrotnych w historii języka Java. Dzięki tej bibliotece tworzenie aplikacji w Javie stało się przyjemnością, a zadania, które do tej pory spędzały programistom sen z powiek, okazały się trywialne. Spring wciąż jest rozwijany, a każda kolejna wersja wprowadza powiew świeżości do świata Javy!

Jeżeli chcesz poznać możliwości tego frameworka oraz wykorzystać drzemiący w nim potencjał w Twoim kolejnym projekcie, to trafiłeś na doskonałą książkę. Wprowadzi Cię ona bezboleśnie w świat odwróconej kontroli (Inversion of Control, w skrócie IoC), wstrzykiwania zależności oraz łatwego dostępu do danych i tworzenia aplikacji internetowych. Każda kolejna strona to bezcenna wiedza na temat tworzenia REST-owych interfejsów API, bezpieczeństwa aplikacji (Spring Security) oraz korzystania z usług JMS. Książka ta będzie świetnym wyborem, jeżeli chcesz poznać ten wyjątkowy szkielet!

Dzięki tej książce:

- poznasz możliwości Spring Framework
- zaznajomisz się z dostępnymi adnotacjami i skutkami ich zastosowania
- zbudujesz aplikację internetową w architekturze MVC
- skorzystasz z zaawansowanych mechanizmów bezpieczeństwa dzięki Spring Security
- błyskawicznie wykorzystasz potencjał drzemiący w Spring Framework

Najlepsze wprowadzenie do Spring Framework!

J Sharma — freelancer i doświadczony programista języka Java. Posiada szeroką wiedzę na temat Spring Framework, baz danych oraz narzędzi stosowanych w świecie Javy i nie tylko. Ma również doświadczenie w zakresie projektowania aplikacji.

Ashish Sarin — jest zdobywcą prestiżowego tytułu Sun Certified Enterprise Architect. Od ponad 14 lat tworzy aplikacje. Jest także autorem książek poświęconych Spring Roo oraz portletom.

Helion

34334 numer katalogowy

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Sprawdź najnowsze promocje:

➤ <http://helion.pl/promocje>

Książki najchętniej czytane:

➤ <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

➤ <http://helion.pl/novosci>

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-283-0929-6



9 788328 309296

Informatyka w najlepszym wydaniu

cena: 69,00 zł