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.



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.

# 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 określanej 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 deklaracyjnych, 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 (określanej 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 `BankAccountServiceImpl` (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-użytkownika@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-użytkownika@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`.

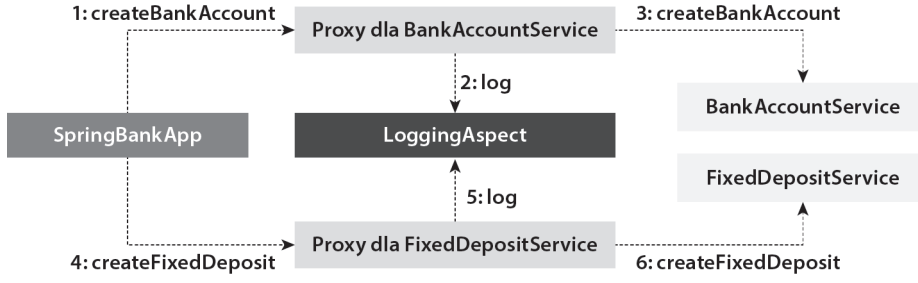
W kontekście `LoggingAspect` zobaczmy teraz, jak działa oferowany przez Spring framework AOP.

➔ 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 `FixedDeposit ↪Service`. Proxy dla obiektu `BankAccountService` przechwytuje wywołania metody `createBank ↪Account()`, 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** **Rozdzial09/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 `isDuplicateAc`
`↪count()` 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 `create`
`↪BankAccount()`. 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, jakakolwiek 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 `isDuplicate`
`↪Account()` klasy `BankAccountService`.

Jeżeli przejdziesz 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 wyrażeń punktu przecięcia.

## 9.4. Wyrażenia punktu przecięcia

Podczas użycia frameworku Spring AOP wyrażenie punktu przecięcia identyfikuje *punkty złączeń*, do których zastosowanie mają rady. W przypadku Springa AOP punkt złączenia *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** **`Rozdzial09/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 desygnatorami punktu przecięcia, teraz dokładnie je przeanalizujemy.

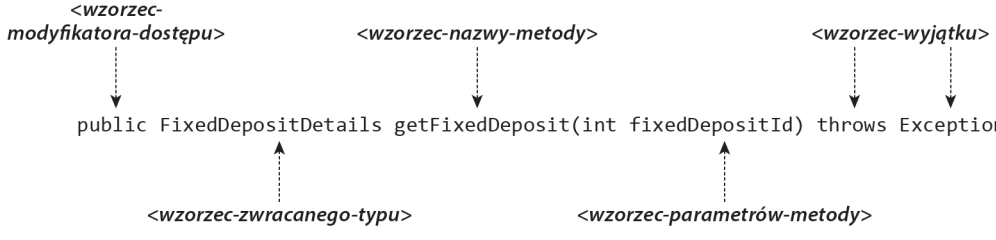
## Desygnatory execution i args

Desygnator `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*

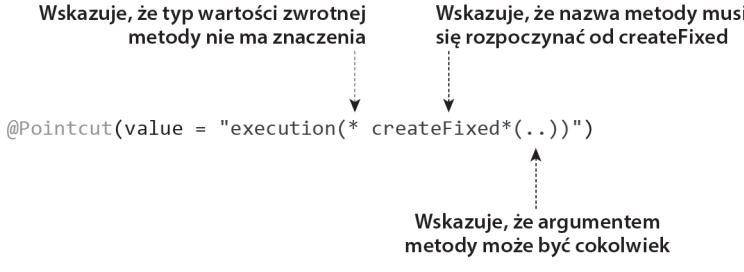
| Fragment wyrażenia | Opis |
|---|---|
| *wzorzec-modyfikatora-dostępu* | Określa modyfikator dostępu metody docelowej. W Springu AOP jedyna wartość, jaką można określić dla tej części wyrażenia, to `public`. Ta część wyrażenia `execution` jest *opcjonalna*. |
| *wzorzec-zwracanego-typu* | Określa w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

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 wyrażeń punktu przecięcia używających desygnatorów `execution` i `args`.

### Przykład 1.

Na rysunku 9.3 zostało pokazane wyrażenie używające wzorca nazwy metody.

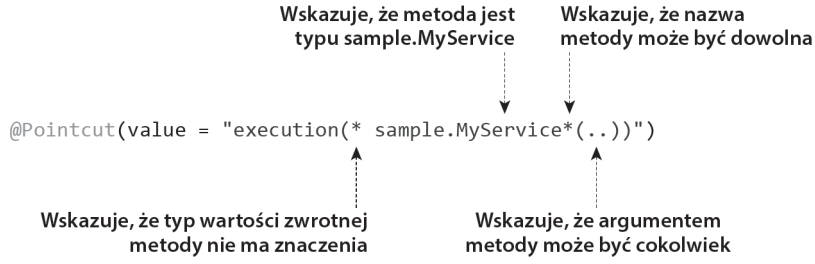



**Rysunek 9.3.** *Wyrażenie punktu przecięcia używające wzorca nazwy metody*

Przedstawione wyrażenie punktu przecięcia powoduje dopasowanie metod o nazwach rozpoczynających się od `createFixed`. Jako typ wartości zwrotnej 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.

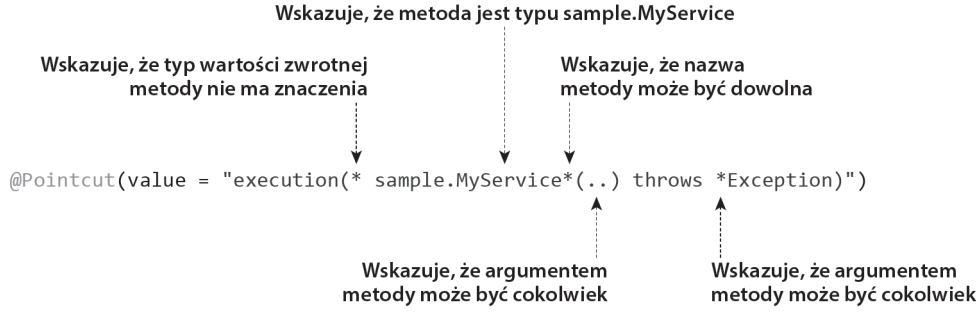



**Rysunek 9.4.** *Wyrażenie punktu przecięcia wskazujące typ (klasa lub interfejs) zawierający metodę docelową*

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.

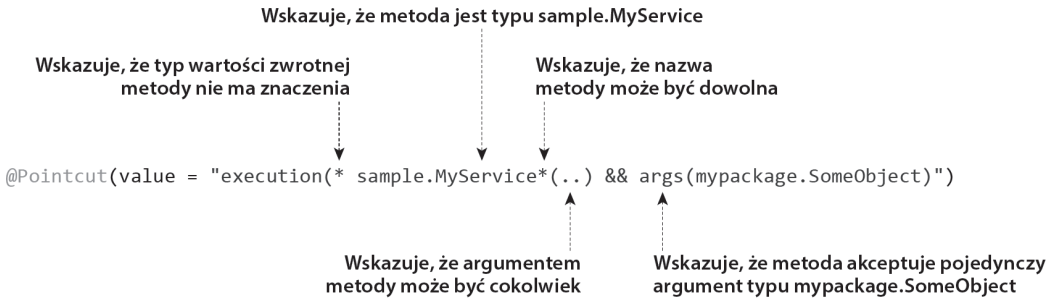



**Rysunek 9.5.** *Wyrażenie punktu przecięcia wskazujące wzorzec metody*

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

### Przykład 4.

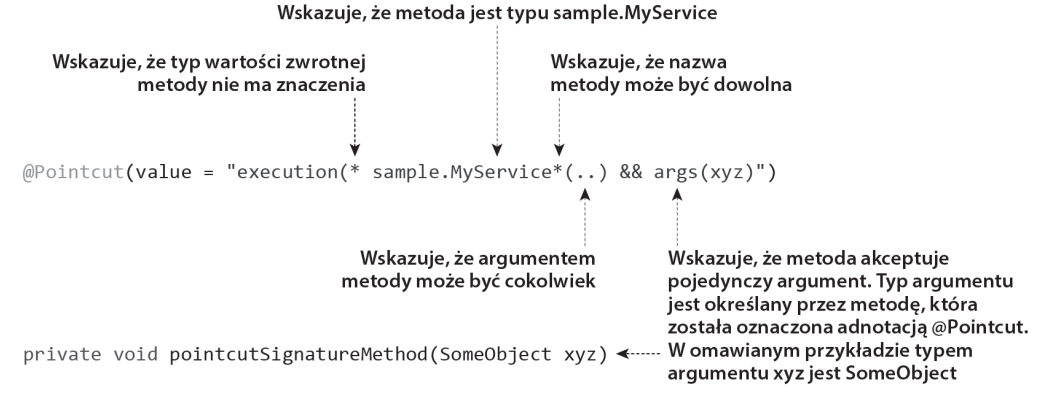
Na rysunku 9.6 zostało pokazane wyrażenie określające egzemplarz obiektu przekazywany metodzie docelowej.




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

Przedstawione wyrażenie punktu przecięcia pokazuje użycie desygnatorów `execution` i `args`. Jeżeli chcesz jednocześnie użyć więcej niż jednego desygnatora, 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 desygnatory `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.** *Desygnator args określa argumenty metody docelowej, które muszą być dostępne dla rady*

W pokazanym na rysunku 9.7 wyrażeniu punktu przecięcia desygnator `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 radzie.

### 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 `Fixed`
`↪DepositDetails`. Ponieważ wyrażenie `args` dostarcza metodę `log()` wraz z egzemplarzem obiektu `FixedDepositDetails`, metoda `log()` została zmodyfikowana w taki sposób, aby akceptowała dodatkowy argument typu `FixedDepositDetails`.

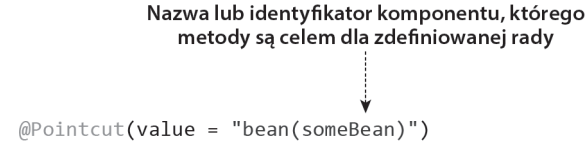
Desygnatory punktu przecięcia, takie jak `execution`, `args`, `within`, `this`, `target` itd., są zdefiniowane przez AspectJ. Spring AOP definiuje charakterystyczny dla tego frameworku desygnator `bean`. Warto więc przeanalizować pokrótce desygnator `bean` punktu przecięcia.

## Desygnator bean

Desygnator `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 desygnatora `bean`.

### Przykład 1.

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

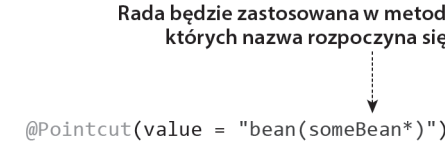



**Rysunek 9.8.** *Desygnator bean określa nazwę komponentu zawierającego metody, w których będzie stosowana rada*

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

**Przykład 2.**

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




**Rysunek 9.9.** *Desygnator 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 desygnatorów punktu przecięcia, istnieje możliwość połączenia desygnatora `bean` z innymi za pomocą operatorów `&&` i `||`. Takie połączenie umożliwia tworzenie skomplikowanych wyrażeń.

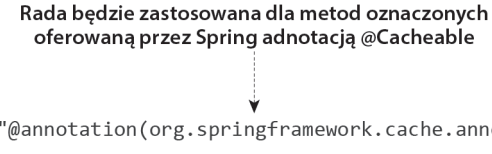
Teraz przechodzimy do desygnatorów, które przeprowadzają dopasowanie na podstawie adnotacji.

## Desygnatory oparte na adnotacjach

AspectJ dostarcza desygnatory 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 desygnatorów.

**Przykład 1.**

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

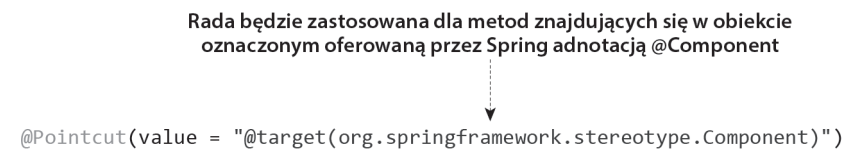



**Rysunek 9.10.** *Przedstawiony desygnator @annotation określa, że rada będzie zastosowana dla metod oznaczonych oferowaną przez Spring adnotacją @Cacheable*

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

**Przykład 2.**

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




**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** **`Rozdzial09/ch09-aop-advices`**. (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-advices* przedstawia aspekt AOP. W listingu 9.8 pokazano, że klasa definiuje radę *after returning* wyświetlającą wartość zwróconą przez metodę `createBank ↪Account()` klasy `BankAccountService`.

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

*Projekt: ch09-aop-advices*
*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 adnotację `@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 `afterRe ↪turningAdvice()` 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-advices*
*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-advices*
*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-advices*.

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

*Projekt: ch09-aop-advices*
*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 `Proceeding↪JoinPoint` metody `aroundAdvice()` jest przeznaczony do kontrolowania, czy metoda docelowa w ogóle zostanie wykonana. Trzeba koniecznie zwrócić uwagę na fakt, że argument `ProceedingJoin↪Point` *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 zwrotną metody `proceed()` klasy `ProceedingJoinPoint` rzutować na typ zwrotny metody docelowej, a następnie ją zmodyfikować. Metoda wywołująca będzie widziała wartość zwracaną przez metodę. Dlatego typ wartości zwrotnej 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** **`Rozdzial09/ch09-aop-xml-schema`**. (Ten projekt jest taki sam jak przedstawiony wcześniej *ch09-aop-advices*, 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 `Sample`
`↪Aspect` 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 wyrażeń punktu 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 zwrotną 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 wyrażeń 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

## B



## C



## D



## E

## F



## H



## I

## J



## K

## L



## M

## R



## S

## Ś



## T



## U



## W

## Z

## Ż

**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/nowosci

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

ISBN 978-83-283-0929-6

9 788328 309296

Informatyka w najlepszym wydaniu

cena: 69,00 zł