

# JSPR-01 SPRING FRAMEWORK

# Prawa autorskie

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszych materiałów szkoleniowych w jakiejkolwiek postaci jest bezwzględnie zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną lub kopiowanie na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich.

Wszelkie znaki występujące w tekście są zastrzeżonymi znakami towarowymi lub firmowymi ich właścicieli.

Autor Łukasz Andrzejewski dołożył wszelkich starań, aby zawarte w materiałach szkoleniowych informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich.

Autor Łukasz Andrzejewski nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w niniejszych materiałach.

# Cele szkolenia

- Wprowadzenie i gruntowne poznanie możliwości framework'u Spring
- Zdobycie praktycznych umiejętności pozwalających na efektywne wykorzystanie frameworku w realizowanych projektach

# Grupa docelowa

- Osoby rozpoczynające naukę Spring framework
- Doświadczeni developerzy Java chcący poznać nowoczesne praktyki programowania
- Programiści Spring pragnący uporządkować/wzbogacić posiadaną wiedzę

# Oczekiwane przygotowanie słuchaczy

- Praktyczna umiejętność programowania w języku Java na poziomie średnio zaawansowanym
- Znajomość SQL na poziomie podstawowym

# Tematyka szkolenia

- Wprowadzenie do technologii
  - a) Spring framework
    - Informacje wstępne
    - Zastosowanie
    - Architektura
  - b) Wyjaśnienie pojęć:
    - Wstrzykiwanie zależności
    - Programowanie zorientowane aspektowo
- Kontener IoC
  - a) Budowa aplikacji opartej o Spring
  - b) Kontenery IoC i ich implementacje
  - c) Konfiguracja beanów (XML, anotacje, JavaConfig)
  - d) Wstrzykiwanie zależności
  - e) Edytory i konwersja typów złożonych
  - f) Cykl życia beanów
  - g) Generowanie i obsługa zdarzeń
  - h) Tworzenie i wykorzystanie obiektów specjalnych
  - i) Internacjonalizacja

# Tematyka szkolenia

- Spring Expression Language (SpEL)
  - a) Składnia, zastosowanie i przykłady użycia
- Programowanie aspektowe
  - a) Idea programowania zorientowanego na aspekty
  - b) Wyjaśnienie i poznanie najważniejszych pojęć
  - c) Weaving i sposoby jego realizacji
  - d) Wykorzystanie Spring AOP
- Wsparcie dla JDBC i DAO
  - a) Konfiguracja i zarządzanie połączeniem z bazą
  - b) Operacje na źródle danych przy użyciu dostępnych szablonów
- Współpraca z narzędziami ORM
  - a) Wprowadzenie do narzędzi mapowania obiektowo-relacyjnego
  - b) Integracja z frameworkiem Hibernate

# Tematyka szkolenia

- Zarządzanie transakcjami
  - a) Pojęcie transakcji
  - b) Menadżer transakcji i jego konfiguracja
  - c) Konfigurowanie mechanizmu transakcyjnego
  - d) Programowe i deklaratywne zarządzanie transakcjami
- Podstawy Spring Web MVC
  - a) Wzorzec Model-View-Controller
  - b) Wprowadzenie do Spring MVC
  - c) Cykl życia żądania
  - d) Konfiguracja aplikacji webowej
  - e) Budowa kontrolerów
  - f) Obsługa formularzy
  - g) Walidacja danych

# Tematyka szkolenia

- Bezpieczeństwo
  - a) Wyjaśnienie podstawowych pojęć
  - b) Wstęp do Spring Security
  - c) Autentykacja i autoryzacja dostępu do metod beanów
  - d) Zabezpieczanie aplikacji webowych

## MODUŁ 1 - WPROWADZENIE DO TECHNOLOGII

# Omawiane zagadnienia

- Wprowadzenie do technologii
  - a) Spring framework
    - Informacje wstępne
    - Zastosowanie
    - Architektura
  - b) Wyjaśnienie pojęć:
    - Wstrzykiwanie zależności
    - Programowanie zorientowane aspektowo

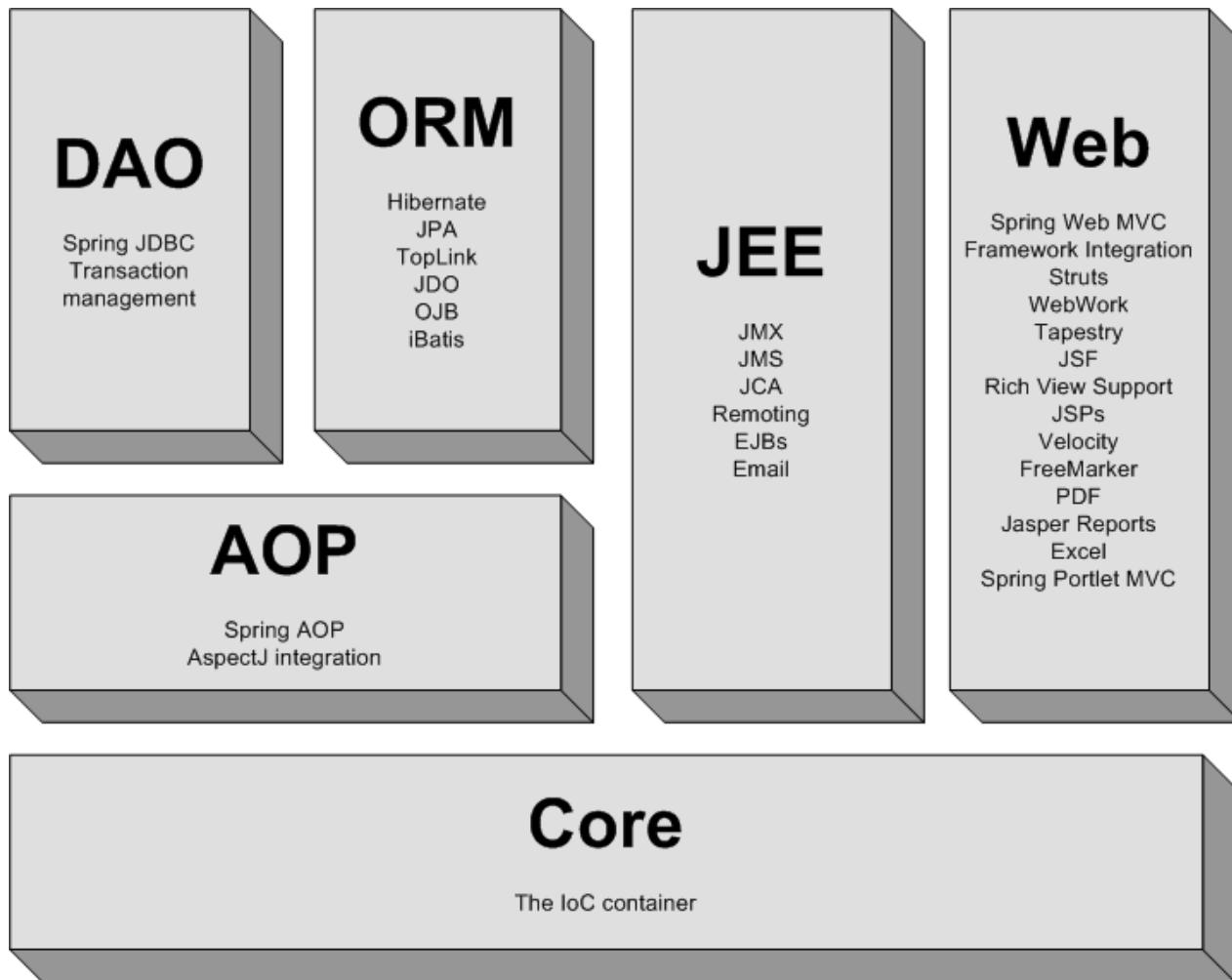
# Spring Framework

- Jeden z najpopularniejszych i najbardziej uniwersalnych frameworków języka Java
- Wykorzystuje mechanizm wstrzykiwania zależności oraz paradymat programowania aspektowego
- Pozwala na zastosowanie podejścia komponentowego w oparciu o lekkie obiekty typu POJO (Plain Old Java Object)
- Promuje programowanie przez interfejsy oraz użycie najlepszych praktyk programistycznych
- Umożliwia integrację z najpopularniejszymi technologiami i frameworkami

# Przyczyny powstania

- Wsparcie podczas tworzenia aplikacji korporacyjnych
- Zapewnienie łatwej i nieinwazyjnej integracji z istniejącymi technologiami
- Dostarczenie alternatywy dla ciężkich rozwiązań takich jak Enterprise JavaBeans 2.x

# Architektura Spring framework



# Wstrzykiwanie zależności (ang. dependency injection)

- Programy tworzone w języku Java realizują zadania poprzez zbiór współpracujących ze sobą obiektów, co wynika bezpośrednio z paradymatu programowania obiektowego
- Obiekty posiadają wzajemne zależności, które komplikują utrzymanie, modyfikowanie i testowanie kodu
- Spring pozwala na rozluźnienie powiązań między obiektami poprzez automatyczne zarządzanie zależnościami

# Programowanie aspektowe (ang. aspect oriented programming)

- Uzupełnia paradygmat programowania obiektowego
- Pozwala dynamicznie modyfikować statyczną hierarchię obiektów i wprowadzać do niej nowe funkcjonalności
- Umożliwia oddzielenie logiki biznesowej od dodatkowych zadań pobocznych takich jak np.: transakcje, logowanie, bezpieczeństwo

# Strategia Spring Framework

- Lekkie i nieinwazyjne programowanie z użyciem obiektów typu POJO
- Niskie sprzężenie uzyskiwane za pomocą wstrzykiwania zależności i programowania przez interfejsy
- Redukcja kodu typu „boilerplate” przez zastosowanie aspektów i szablonów
- Programowanie deklaratywne dzięki aop i przyjętym konwencjom

# Prezentacja praktyczna

- Idea i możliwości Spring Framework

## MODUŁ 2 - KONTENER IOC

# Omawiane zagadnienia

- Kontener IoC
  - a) Budowa aplikacji opartej o Spring
  - b) Kontenery IoC i ich implementacje
  - c) Konfiguracja beanów (XML, anotacje, JavaConfig)
  - d) Wstrzykiwanie zależności
  - e) Edytory i konwersja typów złożonych
  - f) Cykl życia beanów
  - g) Generowanie i obsługa zdarzeń
  - h) Tworzenie i wykorzystanie obiektów specjalnych
  - i) Internacjonalizacja

# Kontener inwersji kontroli (ang. inversion of control)

- Programy tworzone w języku Java składają się z obiektów współpracujących w celu realizacji założonych zadań
- Tradycyjne podejście do definiowania zależności między obiektami prowadzi do kodu trudnego w utrzymaniu i testowaniu (obiekty często wykonują więcej niż powinny i są silnie sprzężone)
- Aplikacje oparte o Spring Framework wykorzystują kontener IoC jako implementację mechanizmu wstrzykiwania zależności
- Kontener odpowiada za tworzenie, konfigurowanie i zarządzanie cyklem życia beanów, a także dostarcza im wszystkie niezbędne usługi

# Typy kontenerów i ich implementacje

- Spring dostarcza kilka implementacji kontenerów, które można sklasyfikować w następujący sposób:
  - a) Wywodzące się z interfejsu `org.springframework.beans.factory.BeanFactory`, zapewniają wsparcie dla mechanizmu wstrzykiwania zależności
  - b) Wywodzące się z interfejsu `org.springframework.context.ApplicationContext`, rozszerzają funkcjonalność kontenerów typu BeanFactory, dostarczają dodatkowe usługi takie jak: propagacja i obsługa zdarzeń, wsparcie dla internacjonalizacji, automatyczne rejestrowanie beanów specjalnych, wcześniejsza inicjalizacja beanów o zasięgu singleton
- Ze względu na oferowaną funkcjonalność najczęściej wykorzystuje się kontenery z rodziny ApplicationContentxt. Wybór konkretnej implementacji zależy od rodzaju aplikacji i sposobu konfiguracji beanów
- Najbardziej popularne implementacje kontenerów typu ApplicationContext to:
  - a) ClassPathXmlApplicationContext
  - b) FileSystemXmlApplicationContext
  - c) XmlWebApplicationContext

# Schemat budowy aplikacji opartej o framework Spring

- Stworzenie opisu beanów wchodzących w skład aplikacji (rejestr XML, anotacje, Java-configuration)
- Wybór implementacji i stworzenie instancji kontenera
- Pozyskanie referencji do beanów zarządzanych przez kontener

# Struktura rejestru XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="inMemoryRepository" class="pl.szkolenie.repository.InMemoryRepository">
        <!-- konfiguracja -->
    </bean>

    <bean id="..." class="...">
        <!-- konfiguracja -->
    </bean>

</beans>
```

# Tworzenie instancji kontenera i pobieranie beanów

```
package pl.szkolenie;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml");
        ((AbstractApplicationContext) ctx).registerShutdownHook();
        BankRepository bankRepository = ctx.getBean("inMemoryRepository", BankRepository.class);
    }
}
```

# Identyfikatory beanów

- Wszystkie bean'y zarządzane w ramach kontenera posiadają przynajmniej jeden, unikalny identyfikator
- Identyfikatory nadawane są za pomocą atrybutu `id` i/lub elementu `bean` lub automatycznie przez kontener np. w przypadku beanów wewnętrznych
- Wartość atrybutu `id` podlega walidacji zgodnie ze specyfikacją standardu XML dlatego jego użycie jest preferowane
- Atrybut `name` pozwala określić wiele identyfikatorów (oddzielonych przecinkiem, spacją lub średnikiem), które mogą zawierać znaki specjalne np. „//”
- Stosowane nazwy powinny mieć charakter opisowy i zwyczajowo pisane są z małej litery zgodnie z konwencją camel-case

# Aliases beanów

- W przypadku dużych systemów, w których konfiguracja jest rozproszona na wiele plików konfiguracyjnych, często zachodzi potrzeba wprowadzenia aliasu dla beana, który został zdefiniowany w innym miejscu
- Podczas konfiguracji opartej o XML nadanie nowej nazwy odbywa się za pomocą elementu **alias**

```
<alias name="identyfikatorBeana" alias="nowaNazwa"/>
```

# Tworzenie instancji beanów

- Definicja beana jest receptą pozwalającą na stworzenie jednego lub więcej obiektów
- W większości przypadków instancje tworzone są z pomocą mechanizmu refleksji, poprzez konstruktor klasy określonej atrybutem `class`
- W przypadku statycznych klas wewnętrznych jako wartość argumentu `class` wprowadzić należy pełną, binarną nazwę klasy np.:

```
<bean id="keyGenerator" class="pl.szkolenie.repository.JDBCRepository$KeyGenerator"/>
```

# Metody fabrykujące

- Framework Spring pozwala używać statycznej metody fabrykującej do tworzenia instancji beanów
- Typ zwracanego obiektu nie musi być tożsamy z typem klasy zawierającej metodę fabrykującą, a tym samym z typem określonym przez atrybut `class`

```
<bean id="identyfikator" class="nazwaKlasyZawierającejFabrykę" factory-method="metodaFabrykująca"/>
```

- Analogicznie istnieje możliwość użycia fabryk instancyjnych - metod fabrykujących zdefiniowanych w ramach innych beanów

```
<bean id="identyfikator" factory-bean="beanZawierającyFabrykę" factory-method="metodaFabrykująca"/>
```

- Wstrzykiwanie ewentualnych argumentów w obu powyższych przypadkach odbywa się analogicznie jak wstrzykiwanie przez konstruktor omówione w dalszej części modułu

# Wstrzykiwanie zależności

- Odbywa się przy użyciu znaczników `<constructor-arg>` i/lub `<property>` zagnieżdzanych wewnątrz znacznika `<bean/>`
- W przypadku wartości podawanych w postaci tekstu (typy prymitywne, String oraz inne) stosuje się atrybut `value`, a dla zależności będących referencją do innego beana atrybut `ref`
- Konwersja wartości podawanych w formie tekstu odbywa się za pomocą specjalnych obiektów typu PropertyEditor (wbudowanych lub zdefiniowanych przez programistę)

# Wstrzykiwanie zależności przez konstruktor

- Domyślnie argumenty przekazywane są w kolejności w jakiej zostały zdefiniowane

```
package x.y;  
public class Foo {  
    public Foo(Bar bar, Baz baz) {  
        // ...  
    }  
}
```

```
<beans>  
    <bean id="foo" class="x.y.Foo">  
        <constructor-arg ref="bar"/>  
        <constructor-arg ref="baz"/>  
    </bean>  
    <bean id="bar" class="x.y.Bar"/>  
    <bean id="baz" class="x.y.Baz"/>  
</beans>
```

# Wstrzykiwanie zależności przez konstruktor

- W przypadku argumentów będących typami prostymi należy określić dodatkowo atrybut **type**

```
package examples;
```

```
public class ExampleBean {  
    private int years;  
    private String ultimateAnswer;  
    public ExampleBean(int years, String ultimateAnswer) {  
        this.years = years;  
        this.ultimateAnswer = ultimateAnswer;  
    }  
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <constructor-arg type="int" value="7500000"/>  
    <constructor-arg type="java.lang.String" value="42"/>  
<bean/>
```

# Wstrzykiwanie zależności przez konstruktor

- Alternatywą pozwalającą na rozróżnienie argumentów jest użycie argumentu `index`, którego wartości rozpoczynają się od 0

package examples;

```
public class ExampleBean {  
    private int years;  
    private String ultimateAnswer;  
    public ExampleBean(int years, String ultimateAnswer) {  
        this.years = years;  
        this.ultimateAnswer = ultimateAnswer;  
    }  
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <constructor-arg index="0" value="7500000"/>  
    <constructor-arg index="1" value="42"/>  
</bean>
```

# Wstrzykiwanie zależności przez konstruktor

- Rozpoczynając od wersji Spring 3.0 istnieje możliwość wykorzystania atrybutu `name`, którego wartość przybiera poszczególne nazwy argumentów
- Opcja ta wymaga komplikacji kodu z flagą `debug` lub użycia adnotacji `@ConstructorProperties`

```
package examples;
```

```
public class ExampleBean {  
    private int years;  
    private String ultimateAnswer;  
    @ConstructorProperties({"years", "ultimateAnswer"})  
    public ExampleBean(int years, String ultimateAnswer) {  
        this.years = years;  
        this.ultimateAnswer = ultimateAnswer;  
    }  
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <constructor-arg name="years" value="7500000"/>  
    <constructor-arg name="ultimateAnswer" value="42"/>  
</bean>
```

# Użycie przestrzeni c

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c="http://www.springframework.org/schema/c"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="bar" class="x.y.Bar"/>
    <bean id="baz" class="x.y.Baz"/>
    <bean id="foo" class="x.y.Foo"
          c:bar-ref="bar"
          c:baz-ref="baz" />
    <bean id="foo2" class="x.y.Foo"
          c:_0-ref="bar"
          c:_1-ref="baz"/>
</beans>
```

# Wstrzykiwanie zależności przez metody typu setter

- Odbywa się po stworzeniu beana i uprzednim wywołaniu konstruktora
- Może być łączone razem z wstrzykiwaniem przez konstruktor

package examples;

```
public class ExampleBean {  
    private int length;  
    public setLength(int length) {  
        this.length = length;  
    }  
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <property name="length" value="42"/>  
</bean>
```

# Użycie przestrzeni p

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean name="john-classic" class="com.example.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>
    <bean name="john-modern" class="com.example.Person"
        p:name="John Doe"
        p:spouse-ref="jane"/>
    <bean name="jane" class="com.example.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

## Beany wewnętrzne

- Beany wewnętrzne to beany definiowane w ramach znaczników `<constructor-arg>` lub `<property>`
- Pozwalają na zdefiniowanie zależności w miejscu jej deklaracji
- Nie można im przypisać nazwy (kontener ją ignoruje i generuje własną), a ich zasięg jest zawsze typu `prototype`

```
<bean id="outer" class="examples.PhoneBook">
    <property name="target">
        <bean class="examples.Person">
            <property name="name" value="Jan Kowalski"/>
            <property name="age" value="12"/>
        </bean>
    </property>
</bean>
```

# Kolekcje i mapy

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <property name="someList">
        <list>
            <value>a list element followed by a reference</value>
            <ref bean="myDataSource" />
        </list>
    </property>
    <property name="someMap">
        <map>
            <entry key="an entry" value="just some string"/>
            <entry key ="a ref" value-ref="myDataSource"/>
        </map>
    </property>
    <property name="someSet">
        <set>
            <value>just some string</value>
            <ref bean="myDataSource" />
        </set>
    </property>
</bean>
```

# Obiekty typu Properties

```
<bean id="mappings" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="properties">
        <value>
            jdbc.driver.className=com.mysql.jdbc.Driver
            jdbc.url=jdbc:mysql://localhost:3306/mydb
        </value>
    </property>
<bean/>

<bean id="moreComplexObject" class="example.ComplexObject">
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.org</prop>
            <prop key="support">support@example.org</prop>
            <prop key="development">development@example.org</prop>
        </props>
    </property>
<bean/>
```

## Wartości null

- Domyślnie pusty łańcuch jest traktowany jako pusty String
- W celu jawnego przypisania wartości null należy użyć specjalnego znacznika <null/>

```
<bean class="examples.ExampleBean">
  <property name="email"><null/></property>
</bean>
```

## Wartości złożone/zagnieżdżone

- Podczas definiowania wstrzykiwanej wartości istnieje możliwość użycia operatora „.” w celu wskazania właściwości zagnieżdżonej (żaden z elementów łańcucha ścieżki nie może mieć wartości null)

```
<bean id="foo" class="foo.Bar">
    <property name="a.b.c" value="123" />
</bean>
```

# Użycie depends-on

- Podczas tworzenia programu może dojść do sytuacji w której bean będzie zależał od istnienia innego beana, jednak nie będzie posiadał do niego bezpośredniej referencji
- Zależności tego typu definiuje się przy pomocy atrybutu `depends-on` (podawane wartości mogą być oddzielone przecinkami, spacjami lub średnikami)
- Określona zależność ma wpływ na kolejność tworzenia oraz niszczenia beanów

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>
<bean id="manager" class="ManagerBean" />
```

# Wstrzykiwanie automatyczne (autowiązanie)

- Kontener dostarcza możliwość automatycznego wstrzykiwania zależności, bez konieczności używania atrybutu ref

```
<bean id="nazwa" class="klasa" autowire="tryb"/>
```

- Istnieje możliwość ustawienia wstrzykiwania automatycznego na poziomie globalnym

```
<beans default-adutowire="tryb">
```

- Autowiązanie działa w następujących trybach
  - a) no - bez automatycznego wstrzykiwania, wartość domyślna
  - b) byName - aby nastąpiło wstrzyknięcie id beana musi być identyczne jak nazwa właściwości
  - c) byType - kandydat do wstrzyknięcia poszukiwany jest na podstawie typu; w kontenerze może istnieć tylko jeden bean poszukiwanego typu; wstrzyknięcie następuje poprzez metody set
  - d) constructor - analogiczne do byType tylko wstrzyknięcie następuje poprzez konstruktor

# Wstrzykiwanie automatyczne (autowiązanie)

- Bez automatycznego wstrzykiwania zależności

```
<bean id="car" class="pl.szkolenie.impl.CarImpl">
    <property name="engine">
        <ref bean="engine"/>
    </property>
</bean>
<bean id="engine" class="pl.szkolenie.impl.EnginelImpl"/>
```

- Z automatycznym wstrzykiwaniem zależności

```
<bean id="car" class="pl.szkolenie.impl.CarImpl" autowire="byName"/>
<bean id="engine" class="pl.szkolenie.impl.EnginelImpl"/>
```

# Dziedziczenie konfiguracji na poziomie XML

- Definicja beana może zawierać sporo informacji takich jak: zależności wstrzykiwane przez konstruktor i settery, konfigurację metody inicjalizującej i sprzątającej, tryb autowiązania i wiele innych
- Spring pozwala na dziedziczenie konfiguracji beanów na poziomie XML
- Rozwiążanie pozwala znacznie zmniejszyć ilość niezbędnej konfiguracji jednak może prowadzić do zmniejszenia jej czytelności
- Użycie atrybutu `abstract` uniemożliwia tworzenie instancji beana. Atrybut jest obowiązkowy w przypadku gdy nie zdefiniowano atrybutu `class`

```
<bean id="inheritedTestBean" abstract="true" class="org.springframework.beans.TestBean">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>
<bean id="inheritsWithDifferentClass" parent="inheritedTestBean" init-method="initialize">
    <property name="name" value="override"/>
</bean>
```

# Zasięg beanów

- Spring definiuje pięć standardowych zasięgów w których mogą występować beany: **singleton**, **prototype**, **request**, **session**, **global session** (trzy ostatnie wymagają dodatkowej konfiguracji)
- Istnieje możliwość definiowania zasięgów niestandardowych
- Domyślny zasięg to **singleton**
- Zmiana zasięgu odbywa się poprzez atrybut **scope**

```
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>
```

# Inicjalizacja poprzez zdefiniowanie metody w XML

- Użycie atrybutu **init-method** dla elementu **bean** pozwala na zdefiniowanie metody inicjalizującej wywoływanej po stworzeniu instancji i wstrzyknięciu jej zależności

```
<bean id="myService" class="pl.szkolenie.Service" init-method="init"/>
```

```
public class Service {  
    public void init() {  
        //...  
    }  
}
```

- Zaleca się używanie tej samej nazwy metody inicjalizującej dla każdego beana; w takim przypadku można ją zdefiniować globalnie na poziomie elementu **beans**:

```
<beans default-init-method="init">  
    ...  
</beans>
```

# Inicjalizacja poprzez implementację InitializingBean

- W celuinicjalizacji beana można także zaimplementować interfejs `InitializingBean` definiujący metodę `afterPropertiesSet()`
- Takie rozwiązanie wprowadza zależność aplikacji od API Spring i w związku z tym w wielu przypadkach może być nieakceptowane

```
public class SystemService implements InitializingBean {  
    public void afterPropertiesSet() throws Exception {  
        //...  
    }  
}
```

# Finalizacja poprzez zdefiniowanie metody w XML

- Użycie atrybutu `destroy-method` dla elementu `bean` pozwala na zdefiniowanie metody finalizującej wywoływanej przed zniszczeniem instancji (nie działa dla prototypów)

```
<bean id="myService" class="pl.szkolenie.Service" destroy-method="destroy"/>
```

```
public class Service {  
    public void destroy() {  
        //...  
    }  
}
```

- Zaleca się używanie tej samej nazwy metody finalizującej dla każdego beana; w takim przypadku można ją zdefiniować globalnie na poziomie elementu `beans`:

```
<beans default-destroy-method="destroy">  
    ...  
</beans>
```

# Inicjalizacja poprzez implementację DisposableBean

- W celu finalizacji beana można także zaimplementować interfejs `DisposableBean` definiujący metodę `destroy()` (nie działa dla prototypów)
- Takie rozwiązanie wprowadza zależność aplikacji od API Spring i w związku z tym w wielu przypadkach może być nieakceptowane

```
public class SystemService implements DisposableBean {  
    public void destroy() throws Exception {  
        //...  
    }  
}
```

# Obiekty specjalne typu BeanPostProcessor

- Interfejs `BeanPostProcessor` pozwala na zdefiniowanie metod typu callback wykonywanych przez kontener przed i po inicjalizacji każdego z beanów
- W celu zarejestrowania post procesora należy zdefiniować go jako jeden z beanów w kontekście kontenera
- Określenie kolejności wykonywania obiektów typu `BeanPostProcessor` jest możliwe poprzez implementację interfejsu `Ordered` i ustalenie odpowiedniej wartości pola `order`

# Obiekty specjalne typu BeanPostProcessor

```
package pl.szkolenie;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("Bean '" + beanName + "' created : " + bean.toString());
        return bean;
    }
}
```

# Obiekty specjalne typu BeanFactoryPostProcessor

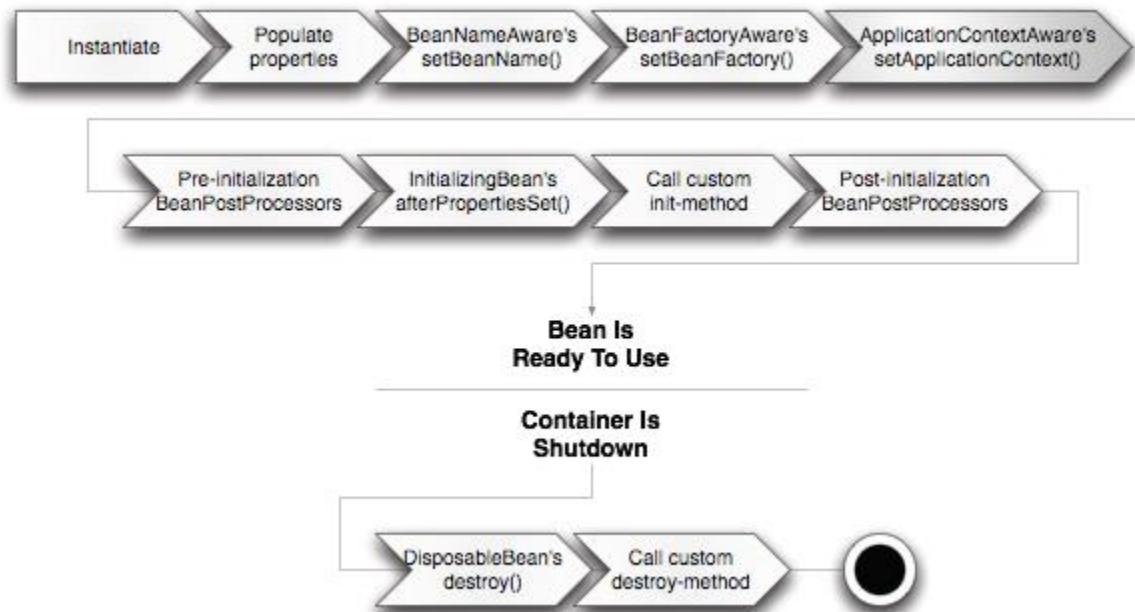
- Interfejs `BeanFactoryPostProcessor` pozwala na stworzenie obiektów mających dostęp do rejestru beanów
- Mogą one zarówno czytać jak i dokonywać modyfikacji konfiguracji beanów zanim jakikolwiek z nich zostanie stworzony
- W celu zarejestrowania post procesora należy zdefiniować go jako jeden z beanów w kontekście kontenera
- Określenie kolejności wykonywania obiektów typu `BeanFactoryPostProcessor` jest możliwe poprzez implementację interfejsu `Ordered` i ustalenie odpowiedniej wartości pola `order`

# Obiekty specjalne typu BeanFactoryPostProcessor

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations" value="classpath:com/foo/jdbc.properties"/>
</bean>
<bean id="dataSource" destroy-method="close" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

# Cykl życia beanów

- Cykl życia beanów zarządzanych przez kontener Spring jest dużo bardziej skomplikowany niż w przypadku zwykłych obiektów języka Java. Poniżej przedstawiono jego poszczególne fazy



# Prezentacja praktyczna

- Konfiguracja kontenera w oparciu o anotacje i Java-config
- Edytory i konwersja typów złożonych
- Generowanie i obsługa zdarzeń
- Internacjonalizacja

## MODUŁ 3 - SPRING EXPRESSION LANGUAGE

# Omawiane zagadnienia

- Spring Expression Language (SpEL)
  - a) Składnia
  - b) Zastosowanie
  - c) Przykłady użycia

# Spring Expression Language (SpEL)

- Samowystarczalny język wyrażeń umożliwiający dostęp do obiektów oraz ich modyfikację w czasie działania aplikacji
- Stworzony, aby zapewnić jeden spójny język wyrażeń dla wszystkich produktów z rodziny Spring
- Pozwala między innymi na:
  - a) dostęp do właściwości obiektów, a także elementów tablic, list oraz map
  - b) wywoływanie metod i konstruktorów
  - c) stosowanie literałów, zmiennych, wartości logicznych, operatorów i wyrażeń regularnych

# SpEL API - parsowanie i ewaluacja wyrażeń

- Klasy i interfejsy związane z API języka SpEL zostały umiejscowione w pakiecie `org.springframework.expression`
- Interfejs `ExpressionParser` definiuje metody pozwalające na parsowanie wyrażeń podanych w postaci tekstu
- Ewaluację sparsowanych wyrażeń umożliwia interfejs `Expression` udostępniając między innymi metody: `getValue()` i `setValue(kontekst, wartość)`

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("Hello World");
String message = (String) exp.getValue();
```

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("new String('hello world').toUpperCase()");
String message = exp.getValue(String.class);
```

# SpEL API - kontekst ewaluacji

- Ewaluacja wyrażeń może odbywać się w wybranym kontekście np. określonej instancji obiektu
- Proces tworzenia obiektu `StandardEvaluationContext` jest dość kosztowny dlatego tam gdzie to możliwe zaleca się jego reużywanie

```
User user = new User("user", "123");
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("login");
EvaluationContext context = new StandardEvaluationContext(user);
String name = (String) exp.getValue(context);
System.out.println(name);
```

```
User user = new User("user", "123");
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("login");
String name = (String) exp.getValue(user);
System.out.println(name);
```

# SpEL jako wsparcie podczas konfiguracji beanów

- Wyrażenia SpEL mogą być używane podczas konfiguracji beanów (rejestr oparty o XML oraz adnotacja `@Value`)
- Ogólna składnia wyrażeń ma postać `#{ wyrażenie }`

```
<bean id="numberGuess" class="org.springframework.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 10.0 }"/>
</bean>
```

```
<bean id="shapeGuess" class="org.springframework.samples.ShapeGuess">
    <property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>
</bean>
```

# SpEL jako wsparcie podczas konfiguracji beanów

- Annotacja `@Value` może być używana na poziomie pól, metod oraz parametrów metod i konstruktorów w celu przypisania wartości domyślnej

```
public static class UserLocale
    @Value("#{ systemProperties['user.region'] }")
    private String defaultLocale;
    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }
    public String getDefaultLocale() {
        return this.defaultLocale;
    }
}
```

## Składnia języka SpEL - literały

- SpEL rozpoznaje literały tekstowe, daty, wartości numerycznych (int, real, hex), logiczne (true, false) oraz null
- W celu zapisania pojedynczego cudzysłowa należy powtórzyć go dwukrotnie

```
ExpressionParser parser = new SpelExpressionParser();
String helloWorld = (String) parser.parseExpression("\"Hello World\"").getValue();
double avogadrosNumber = (Double) parser.parseExpression("-3.131415E+3").getValue();
int maxValue = (Integer) parser.parseExpression("0x1FB2CFFF").getValue();
boolean trueValue = (Boolean) parser.parseExpression("true").getValue();
Object nullValue = parser.parseExpression("null").getValue();
```

# Składnia języka SpEL - nawigacja po właściwościach obiektów

- Dostęp do dowolnie zagnieżdżonych właściwości obiektów realizowany jest za pomocą operatora kropki (żadna składowa łańcucha nie może mieć wartości null)

```
User user = new User("user", "123");
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("login.bytes");
EvaluationContext context = new StandardEvaluationContext(user);
byte[] name = (byte []) exp.getValue(context);
```

## Składnia języka SpEL - wywoływanie metod

- Wywoływanie metod w wyrażeniach SpEL odbywa się analogicznie jak w przypadku standardowej składni języka Java
- Dopuszczalne jest wywoływanie metod na literałach, a także użycie metod o zmiennej liczbie argumentów

```
String c = parser.parseExpression("'abc'.substring(2, 3)").getValue(String.class);
```

## Składnia języka SpEL - tworzenie obiektów

- Tworzenie obiektów odbywa się standardowo przy użyciu operatora new przy czym poza typami prostymi i klasą String wymagane jest podanie pełnej nazwy pakietowej klasy

```
User user = parser.parseExpression("new pl.szkolenie.User('user', '123')").getValue(User.class);
```

# Składnia języka SpEL - tablice, listy i mapy

- SpEL pozwala na bezpośrednie definiowanie list oraz tablic wraz z ich inicjalizacją

```
List numbers = (List) parser.parseExpression("{}").getValue(context);
```

```
List numbers = (List) parser.parseExpression("{1,2,3,4}").getValue(context);
```

```
List listOfLists = (List) parser.parseExpression("{{'a','b'}, {'c','d'}}").getValue(context);
```

```
int[] numbers1 = (int[]) parser.parseExpression("new int[3]").getValue(context);
```

```
int[] numbers2 = (int[]) parser.parseExpression("new int[]{1,2,3}").getValue(context);
```

```
int[][] numbers3 = (int[][][]) parser.parseExpression("new int[5][2]").getValue(context);
```

- W celu pozyskania elementu z tablicy, listy lub mapy należy zastosować notację z nawiasami kwadratowymi (dla tablic i list podaje się indeks elementu, a w przypadku map literał określający wartość klucza)

```
String name = parser.parseExpression("Users[0].name").getValue(usersContext, String.class);
```

```
String name = parser.parseExpression("Users[0].phones['home']").getValue(usersContext, String.class);
```

# Składnia języka SpEL - filtrowanie (ang. collection selection)

- Mechanizm filtrowania pozwala przekształcić kolekcję/mapę źródłową w nową, zawierającą tylko wyselekcjonowane elementy
- Ogólna składnia wyrażeń filtrujących ma postać `?[selectionExpression]`
- W przypadku list ewaluacja kryteriów dotyczy przechowywanych elementów natomiast dla map obiektów typu Map.Entry
- Zapis `^[selectionExpression]` zwraca wyłącznie pierwszy, a `$_[selectionExpression]` ostatni pasujący element

```
List<Integer> list = (List<Integer>) parser.parseExpression("Results.?["Score > 60]").getValue(resultsContext);  
Map map = parser.parseExpression("map.?["value<29]").getValue();
```

# Składnia języka SpEL - projekcja (ang. collection projection)

- Projekcja polega na stworzeniu nowej kolekcji poprzez ewaluację wyrażenia SpEL na elementach kolekcji już istniejącej
- W przypadku list ewaluacja wyrażenia dotyczy przechowywanych elementów natomiast dla map obiektów typu Map.Entry

```
List placesOfBirth = (List) parser.parseExpression("Users.!{placeOfBirth.city}");
```

# Składnia języka SpEL - operatory relacyjne

- Wspierane są wszystkie standardowe operatory relacyjne znane z języka Java
- Każdy z operatorów występuje w wersji symbolicznej oraz słownej (wielkość liter nie ma znaczenia) odpowiednio: lt ('<'), gt ('>'), le ('<='), ge ('>='), eq ('=='), ne ('!='), div ('/'), mod ('%'), not ('!')

```
boolean trueValue = parser.parseExpression("1 == 1").getValue(Boolean.class);
boolean falseValue = parser.parseExpression("-6 < 5.0").getValue(Boolean.class);
```

- Dodatkowo poprzez operator matches obsługiwane są wyrażenia regularne, a także dozwolone jest użycie instanceof

```
boolean falseValue = parser.parseExpression("'abc' instanceof T(int)").getValue(Boolean.class);
boolean trueValue = parser.parseExpression("'60-595' matches '^\\d{2}-\\d{3}$'").getValue(Boolean.class);
```

# Składnia języka SpEL - operatory logiczne

- Obsługiwane operatory logiczne to `and`, `or` i `not`

```
boolean falseValue = parser.parseExpression("true and false").getValue(Boolean.class);
boolean trueValue = parser.parseExpression("true or false").getValue(Boolean.class);
boolean falseValue = parser.parseExpression("!true").getValue(Boolean.class);
```

# Składnia języka SpEL - operatory matematyczne

- SpEL „rozumie” standardowe operatory matematyczne czyli: „\*”, „/”, „+”, „-”, „%”
- Dodatkowo występuje operator potęgowania „^”
- Operatory „+” i „-” zmieniają znaczenie w zależności od kontekstu użycia. („+” może być stosowany do liczb, ciągów znakowych i dat, a „-” do liczb oraz dat)

```
int five = parser.parseExpression("4 + 1").getValue(Integer.class);
String testString = parser.parseExpression("test' + ' + 'string'").getValue(String.class);
int two = parser.parseExpression("1 - -2").getValue(Integer.class);
double d = parser.parseExpression("900.00 - 1e3").getValue(Double.class);
int six = parser.parseExpression("-3 * -2").getValue(Integer.class);
int minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class);
```

# Składnia języka SpEL - pozostałe operatory

- Operator T pozwala na odwołanie się do typu - odpowiedniej instancji java.lang.Class
- Umożliwia dostęp do elementów statycznych
- Użycie operatora wymaga podania pełnej nazwy pakietowej klasy (poza elementami pakietu java.lang)

```
Class stringClass = parser.parseExpression("T(String)").getValue(Class.class);
Double pi = parser.parseExpression("T(Math).PI").getValue(Double.class);
```

- Operator trójargumentowy (ang. ternary operator)

```
String falseString = parser.parseExpression("false ? 'trueExp' : 'falseExp'").getValue(String.class);
```

# Składnia języka SpEL - pozostałe operatory

- Operator elvisowy (ang. elvis operator) - rozwija wyrażenie przy wartości null

```
User user = new User(null, "123");
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("login?:'brak'");
EvaluationContext context = new StandardEvaluationContext(user);
String name = exp.getValue(context, String.class);
```

- Operator bezpiecznej nawigacji - zwraca null zamiast rzucać wyjątek

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("new pl.szkolenie.User(null, '123')?.login");
String name = exp.getValue(String.class);
System.out.println(name);
```

# Składnia języka SpEL - szablony

- Szablony umożliwiają mieszanie literałów tekstowych z blokami podlegającymi ewaluacji
- Każdy rozwijany dynamicznie blok musi być określony przez prefiks i suffix

```
String result = parser.parseExpression("random number is #{T(java.lang.Math).random()}",
    new TemplateParserContext()).getValue(String.class);
```

```
public class TemplateParserContext implements ParserContext {
    public String getExpressionPrefix() {
        return "#{";
    }
    public String getExpressionSuffix() {
        return "}";
    }
    public boolean isTemplate() {
        return true;
    }
}
```

# Składnia języka SpEL - definiowanie zmiennych

- Interfejs `StandardEvaluationContext` pozwala definiować zmienne w ramach wskazanego kontekstu
- Dostęp do zmiennych realizowany jest przez zapis `#nazwaZmiennej`

```
StandardEvaluationContext context = new StandardEvaluationContext(context);
context.setVariable("serverName", "localhost");
parser.parseExpression("server = #serverName").getValue(context);
```

# Składnia języka SpEL - definiowanie funkcji niestandardowych

- Interfejs `StandardEvaluationContext` pozwala rejestrować nowe metody, a tym samym rozszerzać standardową funkcjonalność języka
- Dostęp do zdefiniowanych metod odbywa się przez zapis `#nazwaMetody(parametry)`

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.registerFunction("reverseString", StringUtils.class.getDeclaredMethod("reverseString",
    new Class[] { String.class }));
String helloWorldReversed = parser.parseExpression("#reverseString('hello')").getValue(context, String.class);

public abstract class StringUtils {
    public static String reverseString(String input) {
        return new StringBuilder(input).reverse().toString();
    }
}
```

# Prezentacja praktyczna

- Wykorzystanie SpEL

## MODUŁ 4 - PROGRAMOWANIE ASPEKTOWE

# Omawiane zagadnienia

- Programowanie aspektowe
  - a) Idea programowania zorientowanego na aspekty
  - b) Wyjaśnienie i poznanie najważniejszych pojęć
  - c) Weaving i sposoby jego realizacji
  - d) Wykorzystanie Spring AOP

# Programowanie aspektowe

- Paradygmat programowania pozwalający odseparować logikę biznesową od funkcjonalności pobocznych (ang. cross-cutting concerns)
- Logowanie, bezpieczeństwo, obsługa transakcji oraz inne funkcjonalności zamknięte są w ramach specjalnych klas nazywanych aspektami, a następnie deklaratywnie aplikowane do wybranych modułów aplikacji (bez konieczności ich modyfikacji)
- Programowanie aspektowe przyczynia się do:
  - a) Zwiększenia czytelności kodu - usługi skupią się wyłącznie na realizacji logiki biznesowej
  - b) Zmniejszenia kosztów utrzymania i modyfikacji kodu - funkcjonalności poboczne zamknięte są w ramach aspektów, a nie rozsiane po całej aplikacji

# Najważniejsze pojęcia związane z AOP

- **Aspect** - moduł/jednostka skupiająca się na realizacji funkcjonalności używanej w wielu miejscach aplikacji
- **Join point** - punkt w wykonywanym programie, w którym potencjalnie można dołączyć aspekt
- **Advice** - akcja wykonywana przez aspekt w określonym punkcie joint point
- **Pointcut** - wyrażenie określające miejsca (wybrane join points), w których powinna być dodana funkcjonalność zaimplementowana w aspekcie
- **Target object** - obiekt docelowy do którego dodawana jest funkcjonalność aspektu
- **Weaving** - proces dodawania aspektów do obiektów docelowych
- **Introduction** - dodawanie nowych metod i właściwości do istniejących klas

## Rodzaje advice

- **Before advice** - implementowana funkcjonalność ma pierwszeństwo przed wykonaniem metody docelowej
- **After advice** - implementowana funkcjonalność jest realizowana po wykonaniu metody docelowej niezależnie od zwróconego rezultatu
- **After returning advice** - implementowana funkcjonalność jest realizowana tylko po poprawnym zakończeniu metody docelowej
- **After throwing advice** - implementowana funkcjonalność jest realizowana tylko po wyrzuceniu wyjątku z metody docelowej
- **Around advice** - implementowana funkcjonalność opakowuje całe wykonanie metody docelowej

# Weaving - sposoby realizacji

- Proces dodawania aspektów może być realizowany w różnych fazach cyklu życia obiektów docelowych:
  - a) Przy komplikacji
  - b) Podczas ładowania klasy do maszyny wirtualnej
  - c) W czasie wykonywania programu (obiekty proxy)

- Zaimplementowany w języku Java
- Używa elementów składni AspectJ
- Umożliwia definiowanie aspektów przy użyciu plików XML lub adnotacji
- Pozwala dołączać aspekty wyłącznie do metod publicznych
- Dla każdego z obiektów docelowych tworzy automatycznie obiekt proxy

# Spring AOP - konfiguracja przez adnotacje

- W celu włączenia obsługi adnotacji związanych z aop należy:
  - a) Umieścić znacznik `<aop:aspectj-autoproxy/>` w pliku konfiguracyjnym kontenera
  - b) Dołączyć do classpath biblioteki `aspectjweaver.jar` oraz `aspectjrt.jar`
- Od tej pory każdy bean zarządzany przez kontener, którego klasa została oznaczona adnotacją `@Aspect`, zostanie automatycznie wykryty przez Spring
- Określenie kolejności wykonywania aspektów jest możliwe poprzez implementację interfejsu `Ordered` i ustalenie odpowiedniej wartości pola `order`

# Konfiguracja przez adnotacje - Pointcuts

- Deklaracja elementów typu pointcut składa się z dwóch części:
  - a) Metody określającej nazwę elementu pointcut (musi zwracać void)
  - b) Wyrażenia definiowanego przy użyciu adnotacji `@Pointcut(desygnator(wyrażenie))`

```
@Pointcut("execution(* transfer(..))")  
private void transfercash() {}
```

- Spring AOP wspiera następujące desygnatory: `execution`, `within`, `@within`, `this`, `target`, `@target`, `args`, `@args`, `@annotation`, `bean`
- Wyrażenia pointcut mogą być kombinowane przy użyciu operatorów `&&`, `||`, `!`

```
@Pointcut("execution(public * *(..))")  
private void wszystkiePubliczne() {}  
@Pointcut("within(pl.szkolenie.*)")  
private void wPakiecieSzkolenie() {}  
@Pointcut("wszystkiePubliczne() && wPakiecieSzkolenie()")  
private void wszystkiePubliczneWSzkolenie() {}
```

# Konfiguracja przez adnotacje - Before advice

```
@Aspect  
public class BeforeExample {  
    @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")  
    public void doAccessCheck() {  
        //...  
    }  
}
```

```
@Aspect  
public class BeforeExample {  
    @Before("execution(* com.xyz.myapp.dao.*.*(..))")  
    public void doAccessCheck() {  
        //...  
    }  
}
```

# Konfiguracja przez adnotacje - Before advice

```
@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && " + "args(account,..)")  
private void accountDataAccessOperation(Account account) {}  
  
@Before(value="accountDataAccessOperation(account)", argNames="account")  
public void validateAccount(Account account) {  
    // ...  
}  
  
@Before(value="com.xyz.myapp.SystemArchitecture.dataAccessOperation() && " + "args(account,..)",  
       argNames="account")  
public void validateAccount(Account account) {  
    // ...  
}
```

# Konfiguracja przez adnotacje - After returning advice

```
@Aspect  
public class AfterReturningExample {  
    @AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")  
    public void doAccessCheck() {  
        //...  
    }  
}  
  
@Aspect  
public class AfterReturningExample {  
    @AfterReturning(  
        pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()", returning="retVal")  
    public void doAccessCheck(Object retVal) {  
        //...  
    }  
}
```

# Konfiguracja przez adnotacje - After throwing advice

```
@Aspect  
public class AfterThrowingExample {  
    @AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")  
    public void doRecoveryActions() {  
        //...  
    }  
}  
  
@Aspect  
public class AfterThrowingExample {  
    @AfterThrowing(pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()", throwing="ex")  
    public void doRecoveryActions(DataAccessException ex) {  
        //...  
    }  
}
```

# Konfiguracja przez adnotacje - After advice

```
@Aspect  
public class AfterFinallyExample {  
    @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")  
    public void doReleaseLock() {  
        //...  
    }  
}
```

# Konfiguracja przez adnotacje - Around advice

```
@Aspect  
public class AroundExample {  
    @Around("com.xyz.myapp.SystemArchitecture.businessService()")  
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {  
        Object retVal = pjp.proceed();  
        return retVal;  
    }  
}
```

# Spring AOP - konfiguracja przez XML

- Wszystkie elementy związane z konfiguracją aop w XML muszą zostać zamknięte w ramach znacznika `<aop:config>`
- Deklaracja klasy aspektu odbywa się poprzez znacznik `<aop:aspect>`

```
<aop:config>
    <aop:aspect id="myAspect" ref="aBean">
        ...
    </aop:aspect>
</aop:config>
```

```
<bean id="aBean" class="...">
    ...
</bean>
```

- Określenie kolejności wykonywania aspektów jest możliwe poprzez implementację interfejsu `Ordered` i ustalenie odpowiedniej wartości pola `order`

# Konfiguracja przez XML - Pointcuts

- Deklaracja elementów typu pointcut odbywa się za pomocą znacznika `<aop:pointcut>` stosowanego na poziomie `<aop:config>` lub `<aop:aspect>`

```
<aop:config>
  <aop:pointcut id="businessService" expression="execution(* com.xyz.myapp.service.*.*(..))"/>
</aop:config>
```

- Wyrażenia pointcut mają identyczną postać jak w przypadku konfiguracji przez adnotacje
- Ze względu na składnię XML wyrażenia pointcut mogą być kombinowane przy użyciu operatorów `and`, `or`, `not`

# Konfiguracja przez XML - Before advice

```
<aop:aspect id="beforeExample" ref="aBean">
    <aop:before pointcut-ref="dataAccessOperation" method="doAccessCheck"/>
    ...
</aop:aspect>

<aop:aspect id="beforeExample" ref="aBean">
    <aop:before pointcut="execution(* com.xyz.myapp.dao.*.*(..))" method="doAccessCheck"/>
    ...
</aop:aspect>

<aop:aspect id="beforeExample" ref="aBean">
    <aop:before pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"
        method="audit" arg-names="auditable"/>
</aop:aspect>
```

# Konfiguracja przez XML - After returning advice

```
<aop:aspect id="afterReturningExample" ref="aBean">
    <aop:after-returning pointcut-ref="dataAccessOperation" method="doAccessCheck"/>
    ...
</aop:aspect>

<aop:aspect id="afterReturningExample" ref="aBean">
    <aop:after-returning pointcut-ref="dataAccessOperation" returning="retVal"  method="doAccessCheck"/>
    ...
</aop:aspect>
```

# Konfiguracja przez XML - After throwing advice

```
<aop:aspect id="afterThrowingExample" ref="aBean">
    <aop:after-throwing pointcut-ref="dataAccessOperation" method="doRecoveryActions"/>
    ...
</aop:aspect>

<aop:aspect id="afterThrowingExample" ref="aBean">
    <aop:after-throwing pointcut-ref="dataAccessOperation" throwing="dataAccessEx"
        method="doRecoveryActions"/>
    ...
</aop:aspect>
```

# Konfiguracja przez XML - Around advice

```
<aop:aspect id="aroundExample" ref="aBean">
    <aop:around pointcut-ref="businessService" method="doBasicProfiling"/>
    ...
</aop:aspect>
```

## Parametr JoinPoint

- Metody pełniące rolę advice mogą przyjmować jako pierwszy argument obiekt typu `org.aspectj.lang.JoinPoint` (wyjątkiem jest around advice)
- Interfejs `JointPoint` definiuje metody zwracające szereg przydatnych informacji związanych z obiektem docelowym, wywoływaną metodą oraz obiektem proxy

# Prezentacja praktyczna

- Wykorzystanie programowania przy użyciu aspektów

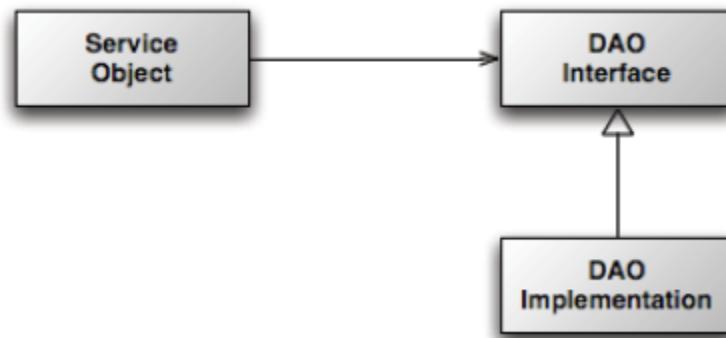
## MODUŁ 5 - WSPARCIE DLA JDBC I DAO

# Omawiane zagadnienia

- Wsparcie dla JDBC i DAO
  - a) Konfiguracja i zarządzanie połączeniem z bazą
  - b) Operacje na źródle danych przy użyciu dostępnych szablonów

# Obiekty typu DAO (ang. data access objects)

- Odpowiadają za realizację dostępu do danych
- Pozwalają uniezależnić system od użytej technologii i sposobu utrwalania
- Ułatwiają testowanie



# Obsługa wyjątków związanych z utrwalaniem

- Spring dostarcza mechanizm konwersji wyjątków specyficznych dla najpopularniejszych technologii utrwalania do spójnej hierarchii wyjątków opartej o typ `DataAccessException`
- Wyrzucane wyjątki nie muszą być przechwytywane w blokach `catch` ponieważ wywodzą się z typu `RuntimeException`

# Strategia utrwalania danych

- Spring separuje stałe i zmienne elementy procesu dostępu do danych
- Klasy typu Templates odpowiadają za realizację typowych, powtarzających się operacji (otwieranie i zamykanie połączenia, obsługa wyjątków, zarządzanie transakcjami), a także dostarczają api upraszczające operowanie na danych
- Klasy typu Callback zawierają zmienną logikę dostępu do danych (tworzenie wyrażeń statement, podstawianie parametrów, mapowanie wyników)
- Framework Spring dostarcza kilka implementacji szablonów zależnych od technologii utrwalania, a także klas typu DAO support mogących służyć jako baza dla obiektów dostępowych

# Konfiguracja połączenia z bazą danych

- Wszystkie szablony i klasy typu DAO support oczekują skonfigurowanego połączenia z bazą danych w postaci obiektu DataSource
- Spring oferuje kilka opcji konfiguracji źródła danych m.in.:
  - a) Z użyciem lokalnego sterownika JDBC
  - b) Poprzez pobranie referencji przy użyciu usługi JNDI
  - c) Z wykorzystaniem puli połączeń

# Konfiguracja połączenia z bazą danych - lokalny sterownik

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:hsq://localhost/myDatabase" />
    <property name="username" value="user" />
    <property name="password" value="123" />
</bean>

<context:property-placeholder location="classpath:jdbc.properties"/>

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="url" value="${database.url}" />
    <property name="driverClassName" value="${database.driver}" />
    <property name="username" value="${database.user}" />
    <property name="password" value="${database.password}" />
</bean>
```

# Konfiguracja połączenia z bazą danych - zdalny obiekt JNDI

- Znacznik `<jee:jndi-lookup>` pozwala pobrać dowolny zasób udostępniony w ramach usługi JNDI i zwrócić w formie beana
- Atrybut `jndi-name` wskazuje nazwę zasobu w rejestrze

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/mysql" />
```

# Konfiguracja połączenia z bazą danych - pula połączeń

- Projekt Jakarta Commons Database Connection Pools (DBCP)

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:hsq://localhost/myDatabase" />
    <property name="username" value="user" />
    <property name="password" value="123" />
    <property name="initialSize" value="4" />
    <property name="maxActive" value="8" />
</bean>
```

# Użycie SimpleJdbcTemplate

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
    <constructor-arg ref="dataSource" />
</bean>

<bean id="myDao" class="pl.szkolenie.myDao">
    <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>

public class myDAO implements DAO {
    private SimpleJdbcTemplate jdbcTemplate;
    public void setJdbcTemplate(SimpleJdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

# Prezentacja praktyczna

- Użycie szablonów JDBC do budowy obiektów typu DAO

## MODUŁ 6 - WSPÓŁPRACA Z NARZĘDZIAMI ORM

# Omawiane zagadnienia

- Współpraca z narzędziami ORM
  - a) Wprowadzenie do narzędzi mapowania obiektowo-relacyjnego
  - b) Integracja z frameworkiem Hibernate

# Wprowadzenie do narzędzi ORM

- Mapowanie obiektowo-relacyjne (ang. object-to-relational mapping) to zautomatyzowany proces zapewniania trwałości obiektów przy użyciu relacyjnej bazy danych
- ORM przekształca dane z modelu obiektowego do modelu relacyjnego i vice-versa
- ORM nie wymaga od programisty pisania tradycyjnego kodu SQL
- W skład frameworków ORM wchodzą:
  - a) API obsługujące podstawowe operacje CRUD
  - b) Język lub API służące do wykonywania bardziej skomplikowanych zapytań
  - c) Narzędzie pozwalające na definiowanie metadanych
  - d) „Maszyneria” zapewniająca obsługę pamięci podręcznej, leniwe ładowanie, propagację kaskadową itd.

# Integracja Spring z produktami ORM

- Spring nie implementuje własnego mechanizmu ORM, ale zapewnia integrację z najpopularniejszymi rozwiązaniami takimi jak Hibernate, JPA, Toplink, iBatis
- Ze swojej strony zapewnia:
  - a) Zintegrowany mechanizm obsługi transakcji
  - b) Zarządzanie i obsługę wyjątków
  - c) Zarządzanie niezbędnymi zasobami

- Najpopularniejszy i najbardziej rozbudowany framework ORM
- Zapoczątkowany przez Gavina Kinga
- Od roku 2003 rozwijany przez JBoss Inc.

# Sesja Hibernate

- Interfejs `org.hibernate.Session` jest używany do komunikacji z frameworkm
- Obsługuje operacje zapisu i odczytu stanu obiektów trwałych
- Sesję uzyskujemy przy pomocy fabryki `org.hibernate.SessionFactory`
- Po zakończeniu operacji na obiektach trwałych sesję należy zamknąć
- W sesji znajduje się m.in. połączenie do bazy danych i pamięć podręczna dla obiektów trwałych

# Obiekty trwałe

- Obiekty trwałe to obiekty, które mogą zostać odtworzone po zatrzymaniu i ponownym uruchomieniu aplikacji
- Każdy obiekt trwały może znajdować się w jednym z trzech stanów względem sesji
  - a) Przechodni (ang. transient) - instancja nie jest związana z sesją, nie posiada wartości klucza głównego
  - b) Trwały (ang. persistent) - instancja jest związana z sesją, posiada przypisaną wartość klucza głównego, może być reprezentowana przez rekord w bazie danych
  - c) Rozłączony (ang. detached) - instancja nie jest związana z sesją, posiada przypisaną wartość klucza głównego, może być reprezentowana przez rekord w bazie danych

# Prezentacja praktyczna

- Integracja Spring i Hibernate

## MODUŁ 7 - ZARZĄDZANIE TRANSAKCJAMI

# Omawiane zagadnienia

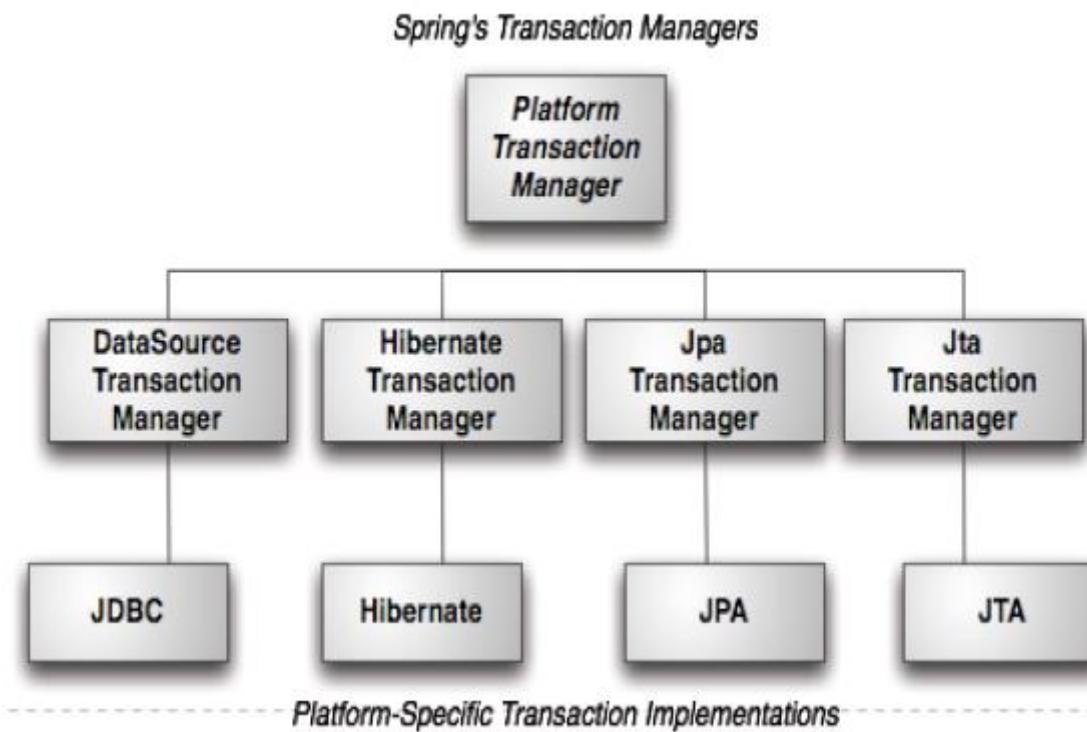
- Zarządzanie transakcjami
  - a) Pojęcie transakcji
  - b) Menadżer transakcji i jego konfiguracja
  - c) Konfigurowanie mechanizmu transakcyjnego
  - d) Programowe i deklaratywne zarządzanie transakcjami

# Pojęcie transakcji

- Transakcja to zbiór czynności podejmowanych w celu realizacji założonego zadania, z których każda musi zostać zakończona w sposób prawidłowy (w innym przypadku wprowadzone zmiany są wycofywane)
- Transakcje są mechanizmem pozwalającym na zachowanie integralności danych na których operuje jednocześnie wiele użytkowników/systemów
- Prawidłowa transakcja powinna spełniać kryteria ACID czyli być:
  - a) atomowa - każda z czynności składających się na realizowane zadanie musi zostać wykonana bezbłędnie i w całości w przeciwnym przypadku transakcja jest przerywana, a wykonane zmiany są cofane
  - b) spójna - składowe systemu muszą zachować integralność w czasie i po zakończeniu transakcji
  - c) izolowana - dane używane podczas transakcji nie mogą być wykorzystywane przez inne elementy systemu w czasie trwania transakcji
  - d) trwała - zmiany wprowadzone podczas transakcji muszą zostać utrwalone w pamięci fizycznej

# Realizacja mechanizmu transakcyjnego w Spring

- Framework Spring nie zarządza transakcjami w sposób bezpośredni lecz dostarcza zbiór gotowych menadżerów transakcji (ang. transaction managers), które delegują odpowiedzialność do konkretnych implementacji mechanizmów transakcyjnych, zależnych od używanej platformy (JDBC, Hibernate, JPA, JTA)



# Deklaracja menadżera transakcji

```
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>

<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManagerName" value="java:/TransactionManager" />
</bean>
```

# Programowa obsługa transakcji

- Wymaga jawnego rozpoczęcia i zatwierdzenia/wycofania transakcji
- Podobnie jak w przypadku obiektów typu DAO, Spring dostarcza specjalny szablon realizujący niezmienne elementy obsługi transakcji
- Programista jest odpowiedzialny za przygotowanie klasy typu Callback

```
<bean id="myService" class="pl.szkolenie.myService">
    <property name="transactionTemplate ">
        <bean class="org.springframework.transaction.support.TransactionTemplate">
            <property name="transactionManager" ref="transactionManager" />
        </bean>
    </property>
</bean>
```

# Programowa obsługa transakcji

```
<bean id="myService" class="pl.szkolenie.myService">
    <property name="transactionTemplate ">
        <bean class="org.springframework.transaction.support.TransactionTemplate">
            <property name="transactionManager" ref="transactionManager" />
        </bean>
    </property>
</bean>

public void saveUser(final User user) {
    txTemplate.execute(new TransactionCallback<Void>() {
        public Void doInTransaction(TransactionStatus txStatus) {
            try {
                userDao.saveUser(user);
            } catch (RuntimeException e) {
                txStatus.setRollbackOnly();
                throw e;
            }
            return null;
        }
    });
}
```

# Parametry transakcji

- Poziom izolacji (ang. isolation level) - definiuje stopień, w jakim transakcja ma dostęp do niezatwierdzonych wyników działania innej transakcji pracującej na tych samych danych
- Propagacja (ang. propagation) - określa sposób propagowania kontekstu transakcji w ramach wywoływanych metod
- Czas ważności (ang. timeout) - wskazuje dopuszczalny czas wykonywania transakcji
- Status tylko do odczytu (ang. readonly status) - informuje czy mogą być stosowane mechanizmy optymalizacyjne związane z faktem, że dane są tylko odczytywane
- Reguły wycofywania (ang. rollback rules) - opisuje jakie działania powinny być podjęte przez mechanizm transakcyjny w przypadku wystąpienia określonych wyjątków

# Poziomy izolacji

- DEFAULT - zgodny z konfiguracją na poziomie źródła danych
- READ\_UNCOMMITTED - pozwala na odczyt danych zmienionych w ramach innych transakcji, ale jeszcze nie zatwierdzonych. Może powodować odczyty brudne, fantomowe i niepowtarzalne
- READ\_COMMITTED - pozwala na odczyt danych zmienionych w ramach innych, zatwierdzonych już transakcji. Może powodować odczyty fantomowe i niepowtarzalne
- REPEATABLE\_READ - zapewnia ochronę przed odczytami niepowtarzalnymi jednak nie chroni przed wystąpieniem fantomów
- SERIALIZABLE - zapewnia pełną ochronę (pełne wsparcie ACID) i najwyższy poziom izolacji kosztem małej wydajności

# Sposoby propagacji

- MANDATORY - metoda musi działać w ramach transakcji, w przypadku jej braku wyrzucony zostaje wyjątek
- NESTED - jeśli transakcja istnieje metoda będzie działać w transakcji zagnieżdżonej, która może być niezależnie zatwierdzana/wycofywana
- NEVER - metoda nie może być wywoływana w ramach transakcji, jeśli tak się stanie dojdzie do wyjątku
- NOT\_SUPPORTED - metoda nie może być wywoływana w ramach transakcji, jeśli tak się stanie bieżąca transakcja zostanie zawieszona
- REQUIRED - metoda musi działać w ramach transakcji, jeśli jej nie ma jest ona tworzona, jeśli jest wykonanie następuje w jej kontekście
- REQUIRES\_NEW - zawsze tworzona jest nowa transakcja, jeśli transakcja już istnieje zostaje zawieszona
- SUPPORTS - metoda nie wymaga transakcji, jeśli transakcja już istnieje metoda będzie wykonywana w jej kontekście

# Deklaratywna obsługa transakcji

- Preferowany sposób obsługi transakcji
- Konfiguracja może odbywać się przez pliki XML lub adnotacje

# Deklaratywna obsługa transakcji - pliki XML

- Konfiguracja transakcji odbywa się przez znacznik `<tx:advice>` oraz znaczniki zagnieżdżone `<tx:attributes>` i `<tx:method>`
- Parametry transakcji dla wskazanej metody określone są przy użyciu atrybutów `isolation`, `propagation`, `read-only`, `rollback-for`, `no-rollback-for` i `timeout` znacznika `<tx:method>`
- Domyślnie zakładane jest istnienie menadżera transakcji o id równym "transactionManager" (jeśli jego nazwa jest inna należy ją zdefiniować przy użyciu atrybutu `transaction-manager`)

```
<tx:advice id="txAdvice" transaction-manager="txManager" >
  <tx:attributes>
    <tx:method name="add*" propagation="REQUIRED" />
    <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
  </tx:attributes>
</tx:advice>
```

# Deklaratywna obsługa transakcji - pliki XML

- W celu przypisania aspektu realizującego obsługę transakcji konkretnym metodom aplikacji stosuje się znacznik `<aop:advisor>`

```
<aop:config>
  <aop:advisor pointcut="execution(* *..Szkolenie.*(..))" advice-ref="txAdvice"/>
</aop:config>
```

# Deklaratywna obsługa transakcji - adnotacje

- Włączenie obsługi transakcji przy użyciu adnotacji odbywa się przez zadeklarowanie elementu **annotation-driven**
- Podobnie jak w przypadku XML, wskazanie id beanu pełniącego rolę menadżera transakcji jest konieczne jeśli jego nazwa jest różna od "transactionManager"
- Adnotacja może być stosowana na poziomie klasy i/lub wybranych metod publicznych, posiada takie same atrybuty jak znacznik **<tx:method>**

```
<tx:annotation-driven transaction-manager="txManager" />
```

```
@Service
public class BankTransferService {
    @Transactional(propagation=Propagation.REQUIRES_NEW)
    public void doTransfer() {
        //...
    }
}
```

# Prezentacja praktyczna

- Obsługa transakcji

## MODUŁ 8 - PODSTAWY SPRING WEB MVC

# Omawiane zagadnienia

- Podstawy Spring Web MVC
  - a) Wzorzec Model-View-Controller
  - b) Wprowadzenie do Spring MVC
  - c) Cykl życia żądania
  - d) Konfiguracja aplikacji webowej
  - e) Budowa kontrolerów
  - f) Obsługa formularzy
  - g) Walidacja danych

# Spring Web MVC

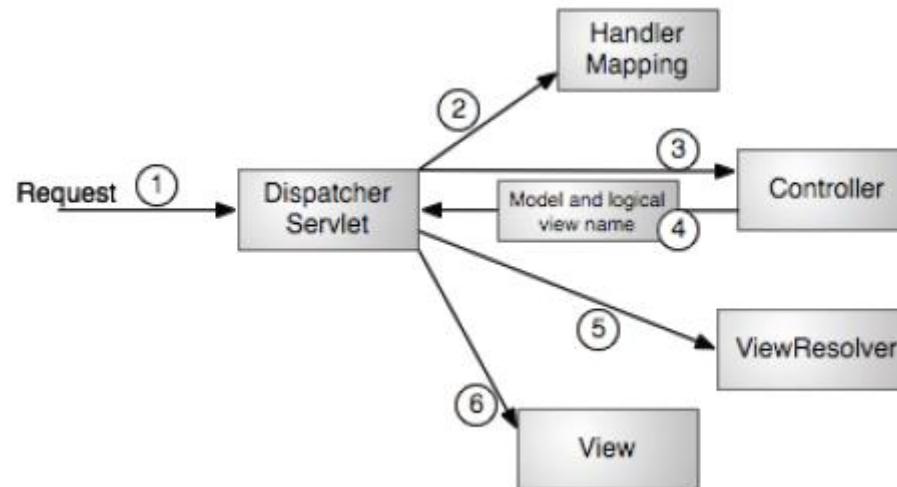
- Framework webowy rozwijany w ramach niezależnego projektu Spring
- Stanowi implementację wzorca Model-View-Controller
- Zapewnia walidację i automatyczne mapowanie danych formularzy na obiekty
- Wspiera użycie wielu technologii widoku
- Pozwala obsługiwać żądania rozciągające się na kilka stron
- Umożliwia internacjonalizację aplikacji

# Wzorzec Model-View-Controller

- Implementacja wzorca MVC polega na logicznym wydzieleniu w aplikacji następujących warstw:
  - a) Warstwa modelu - przechowuje stan aplikacji, reprezentuje struktury danych prezentowane użytkownikom
  - b) Warstwa widoku - prezentuje dane przechowywane w modelu, pozwala na interakcję z systemem i powiadamia kontrolerów o podjętych akcjach użytkownika
  - c) Warstwa kontrolera - definiuje zachowanie aplikacji, przekształca żądania użytkownika na akcje wykonywane na modelu, decyduje o użyciu konkretnego widoku
- Cechy wzorca MVC:
  - a) Zapewnia swobodę w sposobie prezentowania danych
  - b) Centralizuje kontrolę przepływu sterowania w aplikacji
  - c) Separuje komponenty odpowiedzialne za przechowywanie, pozyskiwanie i zapis danych od komponentów widoku

# Cykl życia żądania

- Za każdym razem kiedy użytkownik kliką wybrany link albo zatwierdza formularz na serwer wysłane zostaje żądanie HTTP (ang. HTTP request)
- Poniżej przedstawiony został schemat obsługi żądania po stronie serwera



# Konfiguracja aplikacji webowej - front controller

- Głównym elementem każdej aplikacji opartej o Spring MVC jest `DispatcherServlet` pełniący rolę front controller'a
- Jego konfiguracja odbywa się tradycyjnie w pliku `web.xml`
- Nazwa servletu ma istotne znaczenie - podczas ładowania servletu następuje próba wczytania kontekstu Spring z pliku `<nazwa>-servlet.xml`

```
<servlet>
  <servlet-name>bank</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
```

```
</servlet>
```

```
<servlet-mapping>
  <servlet-name>bank</servlet-name>
  <url-pattern>*.mvc</url-pattern>
</servlet-mapping>
```

# Konfiguracja aplikacji webowej - ładowanie kontekstu

- Do poprawnego załadowania kontekstu Spring wymagane jest zdefiniowanie obiektu listenera typu `ContextLoaderListener`
- Na podstawie deskryptora `applicationContext.xml` utworzy on główny kontekst aplikacji, a plik `<nazwa>-servlet.xml` posłuży jako kontekst potomny
- Wskazanie innych plików umożliwia parametr `contextConfigLocation`

```
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/bank-repository.xml /WEB-INF/bank-security.xml</param-value>
</context-param>
```

# Prezentacja praktyczna

- Budowa kontrolerów
- Obsługa formularzy
- Walidacja danych

## MODUŁ 9 - BEZPIECZEŃSTWO

# Omawiane zagadnienia

- Bezpieczeństwo
  - a) Wyjaśnienie podstawowych pojęć
  - b) Wstęp do Spring Security
  - c) Autentykacja i autoryzacja dostępu do metod beanów
  - d) Zabezpieczanie aplikacji webowych

# Bezpieczeństwo - wyjaśnienie podstawowych pojęć

- Autentykacja (ang. authentication) - identyfikacja użytkownika najczęściej realizowana przy użyciu loginu i hasła
- Autoryzacja (ang. authorization) - pozwala sprawdzić czy dany użytkownik ma prawo wykonywania żądanych przez niego operacji
- Integralność danych (ang. data integrity) - polega na zagwarantowaniu spójności danych przekazywanych podczas komunikacji między nadawcą i odbiorcą
- Poufność (ang. confidentiality) - proces ograniczający dostęp do istotnych informacji niepowołanym osobom

# Wstęp do Spring Security

- Spring Security jest osobnym modułem skupiającym się na aspektach związanych z bezpieczeństwem
- Pozwala na zabezpieczanie aplikacji webowych oraz poszczególnych metod beanów
- Konfiguracja odbywa się w sposób deklaratywny przy użyciu plików XML lub adnotacji
- Bezpieczeństwo jest realizowane przez zbiór gotowych implementacji menadżerów, zależnych od sposobu przechowywania i pobierania informacji niezbędnych do wykonania autentykacji/autoryzacji

# Prezentacja praktyczna

- Autentykacja i autoryzacja dostępu do metod beanów
- Zabezpieczanie aplikacji webowych