

A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front one is blue and the back one is light green. They are positioned diagonally, with the blue one partially covering the green one.

Spring DI

Jarosław Cierpich

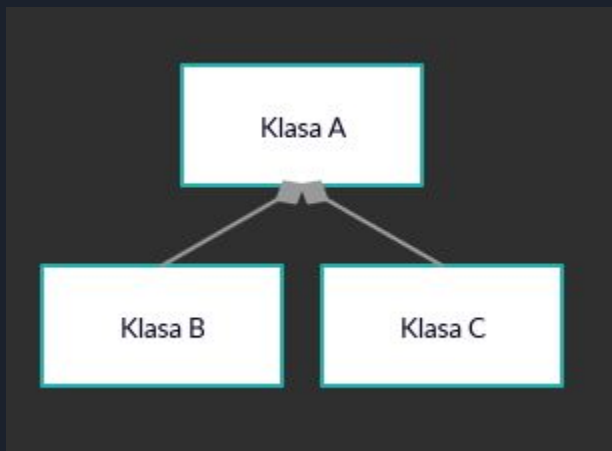


Dependency Injection

- Wzorzec projektowy oraz architektury oprogramowania, który polega na usuwaniu bezpośrednich zależności pomiędzy komponentami programu. Zastosowanie tzw. architektury “plug-in”.
- Jedna z technik realizujących paradygmat IoC (Inversion of Control)
- Pozwala na łatwe uzyskanie luźnych powiązań (Loose coupling)

Inversion of control (IoC)

Tradycyjne podejście - dependencje są wytwarzane przez obiekt, który je konsumuje





Inversion of control (IoC)

Tradycyjne podejście

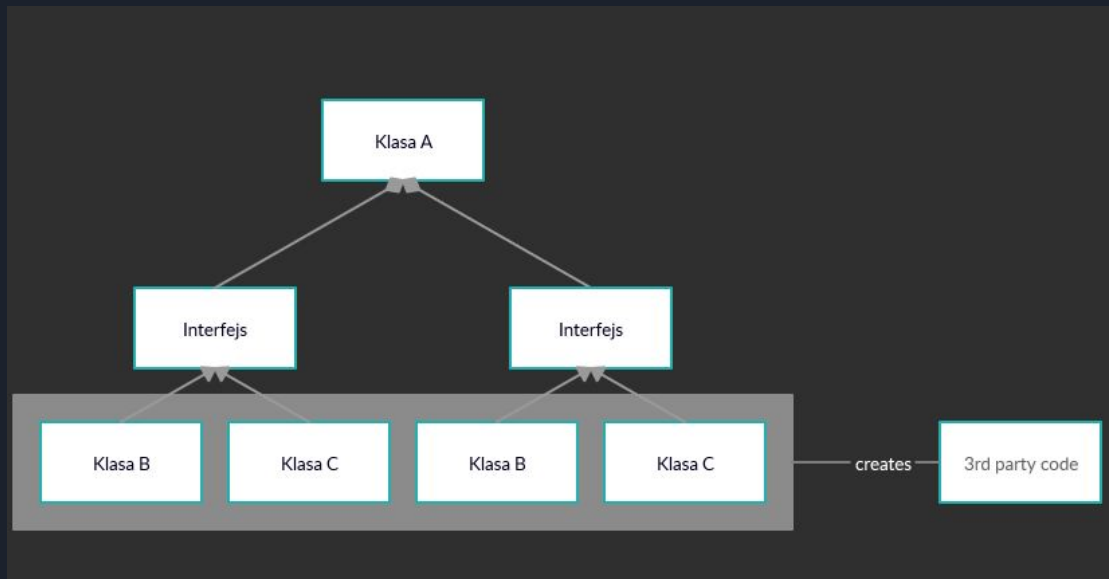
```
class KlasaA{  
  
    public void doWork(){  
        firstDependency.doWork();  
        secondDependency.doWork();  
    }  
  
    private KlasaB firstDependency = new KlasaB();  
    private KlasaC secondDependency = new KlasaC();  
}
```

```
class KlasaB{  
    public void doWork(){  
        System.out.println("Klasa B");  
    }  
}  
  
class KlasaC{  
    public void doWork(){  
        System.out.println("Klasa C");  
    }  
}
```

Inversion of control (IoC)

Podejście IoC

Podejście IoC - dependencje są wytwarzane przez zewnętrzny kod, a następnie dostarczane do obiektu który je konsumuje





Inversion of control (IoC)

Podejście IoC

```
class KlasaA{  
  
    public KlasaA(KlasaInterface firstDependency, KlasaInterface secondDependency){  
        this.firstDependency = firstDependency;  
        this.secondDependency = secondDependency;  
    }  
  
    public void doWork(){  
        firstDependency.doWork();  
        secondDependency.doWork();  
    }  
  
    private KlasaInterface firstDependency;  
    private KlasaInterface secondDependency;  
}
```

```
interface KlasaInterface{  
    public void doWork();  
}  
  
class KlasaB implements KlasaInterface{  
    public void doWork(){  
        System.out.println("Klasa B");  
    }  
}  
  
class KlasaC implements KlasaInterface{  
    public void doWork(){  
        System.out.println("Klasa C");  
    }  
}
```



Inversion of Control (IoC)

Co zyskujemy

- Odpowiednio zaprojektowany kod wykorzystujący podejście IoC jest znacznie lepiej dostosowany do modyfikacji.
- Testowanie kodu, czy też stosowanie podejścia TDD jest znacznie prostsze.
- Zmiany w kodzie wymagają mniejszej ilości dokonywanych kompilacji.
- Przy poprawnie zastosowanym IoC jesteśmy w stanie modyfikować kod nawet bez kompilacji



Inversion of Control (IoC)

Przykład 1. - HelloWorldIoC



Spring Bean

Co to jest

- Bean jest to koncept powszechnie stosowany w technologii JAVA.
- Według standardu jest to klasa, która:
 - posiada publiczny konstruktor bezargumentowy
 - wszystkie jego atrybuty mają prywatny zakres dostępu
 - dostęp do atrybutów realizowany jest za pomocą getterów/setterów
- Inaczej jest w przypadku frameworku Spring. Bean jest to klasa/obiekt zarządzany przez kontener Spring IoC. (instancjonowanie, składanie, zarządzanie)



Spring IoC

Przykład 2. - HelloWorldIoCSpring



Rodzaje IoC

Dependency Pull - potrzebne dependencje są pobierane (pull) z rejestru.

```
ApplicationContext applicationContext = new ClassPathXmlApplicationContext( configLocation: "/spring/hello-world.xml");  
MessageRenderer renderer = applicationContext.getBean( s: "renderer", MessageRenderer.class);
```



Rodzaje IoC

Contextualized Dependency Lookup - klasa, która wymaga jakiejś zależności implementuje interfejs pozwalający na pobranie tejże zależności z kontenera

```
public class ContextualizedDependencyLookup implements ManagedComponent {  
    private Dependency dependency;  
  
    public void performLookup(Container container) {  
        this.dependency = (Dependency) container.getDependency("myDependency");  
    }  
  
    public String toString() {  
        return dependency.toString();  
    }  
}
```



Rodzaje IoC

Setter Dependency Injection

```
public class SystemErrMessageRenderer implements MessageRenderer {  
  
    private MessageProvider provider;  
  
    @Override  
    public void setProvider(MessageProvider provider) {  
        this.provider = provider;  
    }  
  
    @Override  
    public void render() {  
        System.err.println("ERROR: " + provider.getMessage());  
    }  
}
```

Constructor Dependency Injection

```
class KlasaA{  
  
    public KlasaA(KlasaInterface firstDependency, KlasaInterface secondDependency){  
        this.firstDependency = firstDependency;  
        this.secondDependency = secondDependency;  
    }  
  
    public void doWork(){  
        firstDependency.doWork();  
        secondDependency.doWork();  
    }  
  
    private KlasaInterface firstDependency;  
    private KlasaInterface secondDependency;  
}
```



BeanFactory

- jest to interfejs definiujący metody, które musi implementować każdy rejestr w SpringFramework. Metody te odpowiadają za zarządzanie komponentami, ich zależnościami oraz ich cyklem życia.
- Informacje do BeanFactory trafiają dzięki wywołaniu odpowiednich metod klas, które implementują interfejs **BeanDefinitionReader**. Jest wiele implementacji tego interfejsu, np.:
 - xml - XmlBeanDefinitionReader
 - properties - PropertiesBeanDefinitionReader

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
XmlBeanDefinitionReader rdr = new XmlBeanDefinitionReader(factory);  
rdr.loadBeanDefinitions(new ClassPathResource("spring/xml-bean-factory-config.xml"));  
MessageRenderer renderer = (MessageRenderer) factory.getBean( name: "renderer");
```



ApplicationContext

- Jest to interfejs rozszerzający BeanFactory. Pozwala na wczytanie definicji Bean bez potrzeby korzystania z innej klasy
- Oprócz wsparcia dla DI zapewnia również m.in. transakcje, serwis AOP, event handling.

```
ApplicationContext applicationContext = new ClassPathXmlApplicationContext( configLocation: "/spring/hello-world.xml");  
MessageRenderer renderer = applicationContext.getBean( s: "renderer", MessageRenderer.class);
```



Konfiguracja Spring DI

- za pomocą pliku xml
- za pomocą kodu Java
- za pomocą adnotacji



Konfiguracja Spring DI

za pomocą pliku 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.xsd">

    <bean id="provider" class="examples.example1.HelloWorldMessageProvider"/>
    <bean id="renderer" class="examples.example1.SystemErrMessageRenderer"/>

</beans>
```

Przykład 2.

Konfiguracja Spring DI

za pomocą kodu Java

```
public class HelloWorldAnnotation {  
    public static void main(String[] args){  
        ApplicationContext applicationContext = new AnnotationConfigApplicationContext(HelloWorldConfiguration.class);  
        MessageRenderer renderer = applicationContext.getBean("renderer", MessageRenderer.class);  
        renderer.render();  
    }  
}  
  
@Configuration  
class HelloWorldConfiguration {  
  
    @Bean  
    public MessageProvider provider() { return new HelloWorldMessageProvider(); }  
  
    @Bean  
    public MessageRenderer renderer(){  
        MessageRenderer renderer = new SystemOutMessageRenderer();  
        renderer.setProvider(provider());  
        return renderer;  
    }  
}
```

Przykład 3.



Konfiguracja Spring DI

za pomocą adnotacji

```
public class HelloWorldAnnotation {  
    public static void main(String[] args){  
        ApplicationContext applicationContext = new AnnotationConfigApplicationContext(HelloWorldConfiguration.class);  
        MessageRenderer renderer = applicationContext.getBean("renderer", MessageRenderer.class);  
        renderer.render();  
    }  
}  
  
@Configuration  
@ComponentScan(basePackages = "examples.example4")  
class HelloWorldConfiguration { }
```

Przykład 4.



Konfiguracja Spring DI

mieszany

```
public class HelloWorldComponentScan {  
    public static void main(String[] args){  
        ApplicationContext applicationContext = new AnnotationConfigApplicationContext(HelloWorldConfigurationXml.class);  
        MessageRenderer renderer = applicationContext.getBean("renderer", MessageRenderer.class);  
        renderer.render();  
    }  
}  
  
@Configuration  
@ImportResource("classpath:spring/hello-world-5.xml")  
class HelloWorldConfigurationXml{}
```

Przykład 5.



Konfiguracja Spring DI

Więcej o XML

- zależności możemy zdefiniować również w konfiguracji xml

```
<bean id="renderer" class="examples.example1_2.SystemErrMessageRenderer">  
  <property name="provider" ref="provider"/>  
</bean>
```

```
<bean id="renderer" class="examples.example1_2.SystemErrMessageRenderer" p:provider-ref="provider"/>
```

- możliwa jest inicjalizacja parametrów konstruktora typami prostymi, czy też kolekcjami

Przykład 6.

- Podobne możliwości zapewniają pozostałe dwa sposoby konfiguracji



Zagnieżdżanie konfiguracji

- Aplikacja może wczytać więcej niż jedną konfigurację.
- Spring pozwala na hierarchizację wczytanych konfiguracji, dzięki czemu łatwiejsze jest zarządzanie dużymi projektami.

Przykład 7



Tryby instancjonowania

- Domyślnie wszystkie Bean zarządzane przez Spring IoC Container są **SINGLETONAMI**
- Tryb instancjonowanie można zmienić definiując konfigurację:

- xml

```
<bean id="renderer" class="examples.example1_2.SystemErrMessageRenderer" scope="prototype"/>
```

- adnotacją

```
@Component("provider")
@Scope("prototype")
public class HelloWorldMessageProvider implements MessageProvider {
    @Override
    public String getMessage() { return "Hello World!"; }
}
```



Tryby instancjonowania

- Wspierane tryby instancjonowania:
 - Singleton
 - Prototype - przy każdym zapytaniu o Bean będzie tworzona nowa instancja.
 - Request - do użytku przy aplikacjach webowych. Instancja jest tworzona dla każdego HTTP Request. Po zakończeniu obsługi HTTP Request instancja jest usuwana.
 - Session - j.w. tylko, że dla sesji.
 - Global Session - portlet-based web applications. Jedna instancja jest współdzielona pomiędzy wiele portletów.
 - Thread - jedna instancja dla jednego wątku.
 - Custom - można również utworzyć własny tryb instancjonowania.

Przykład 8.



Autowire

- Spring wspiera pięć trybów automatycznego wiązania zależności:
 - *byName* - Spring powiąże wszystkie metody, pola, oraz parametry konstruktorów na podstawie nazwy parametru oraz nazw dostępnych Beans
 - *byType* - Spring dokonuje wiązania na podstawie typów, wykorzystuje metody set/get
 - *constructor* - działa podobnie jak w przypadku *byType*, natomiast wykorzystuje w tym przypadku konstruktor. Jeżeli istnieje kilka konstruktorów wybiera ten, który ma najwięcej parametrów, które można dopasować
 - *default* - Spring automatycznie wybierze tryb *byType* lub *constructor* faworyzując ten pierwszy
 - *no* - automatyczne wiązanie wyłączone. Domyślne dla wszystkich Beans

Przykład 9.



Co dalej?

Ściąga dodatkowych tematów dotyczących Spring DI

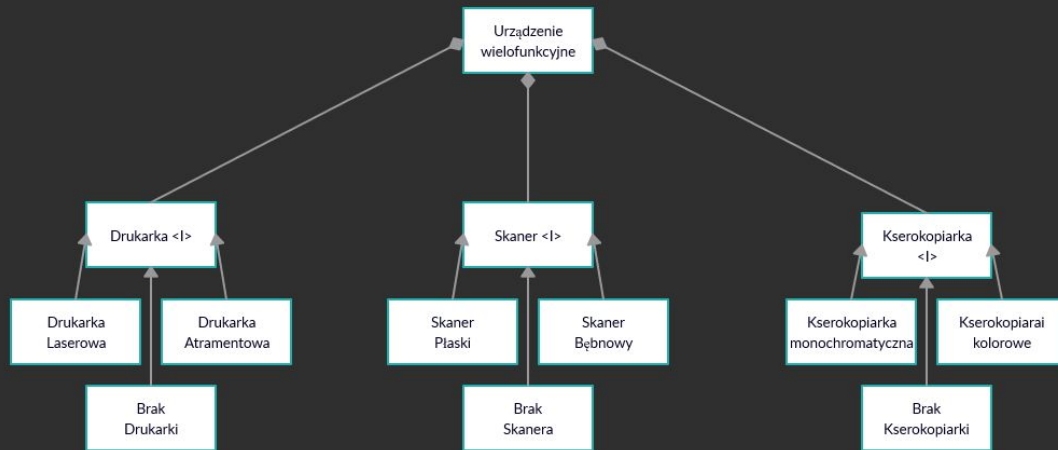
- Wstrzykiwanie kolekcji za pomocą Spring DI
- SpEL - Spring Expression Language
- Wstrzykiwanie Metod/Podmienianie Metod
- Nazewnictwo oraz aliasy dla Beans
- Rozwiązywanie zależności



Dziękuję za uwagę

Zadanie

- Zaimplementować na dwa sposoby: konfiguracja XML oraz Adnotacje
- Każda z klas powinna mieć zaimplementowany toString() tak, aby klasa urządzenia wielofunkcyjnego mogła je wywołać (w celu sprawdzenia poprawnego działania).
- Przykładowy output (nie musi być 1:1):
MultifunctionDevice{printer=LaserPrinter, scanner=FlatScanner, xero=NoXero}





Jak uruchomić w Eclipse

- Sklonować repozytorium <https://github.com/Loniowsky/PrezentacjaZTI>
- Uruchomić eclipse -> Utworzyć nowy workspace -> Wybrać "Import Existing Projects" -> Jako Root Directory podać sklonowane repozytorium -> Finish
- Kliknąć prawym przyciskiem myszy korzeń projektu -> Configure -> Convert to Maven project
- Poczekać aż Maven pobierze potrzebne zależności
- Gotowe :)