

Bachelor of Science (BSc) in Informatik
Modul Advanced Software Engineering 1 (ASE1)

LE 02 – Spring Framework

Spring Core Framework Basics Dependency Injection

Institut für Angewandte Informationstechnologie (InIT)

Walter Eich (eicw) / Matthias Bachmann (bacn)

<https://www.zhaw.ch/de/engineering/institute-zentren/init/>

Zürcher Fachhochschule

Das Spring Framework ist wohl die am meisten verbreitete Java-Bibliothek. Der Grund dafür ist relativ einfach, denn es nimmt viele häufige Programmieraufgaben auf deutlich weniger komplexe Art und Weise in Angriff. Binnen weniger Minuten können Java-Entwickler so komplexe Anwendungen von Enterprise-Niveau erstellen. Spring kann viel sein, aber auch wenig. Was genau das heissen soll und was Spring Framework konkret bietet, lernen Sie durch dieses Skript. Zunächst sollte man sich genau dafür noch einmal vor Augen führen, warum Spring eigentlich so beliebt in der Community ist.

Lernziele LE 02 – Dependency Injection

- Der Studierenden können ...,
 - mit Hilfe von Dependency Injection das IoC-Prinzip anwenden
 - die Konfiguration von Beans mit Hilfe von externen xml-Dateien oder mit Hilfe von Konfigurationen durchführen
 - den LifeCycle für das Erstellen oder Zerstören von Containern anwenden

Agenda

1. Das Spring Framework im Überblick
2. Dependency Injection
1. Grundlagen
2. Was ist eine Bean
3. Deklaration mittels XML
4. Weitere Konzepte
5. Deklaration mittels Annotationen
3. Aspektorientierte Programmierung mit Spring
4. Spring Data, Hibernate und JPA

Das Spring Framework im Überblick

Das Spring Framework und all seine zugehörigen Komponenten sind fester Bestandteil des Werkzeugs vieler Java-Entwickler geworden. Durch kontinuierliche Erweiterung bleibt es dauerhaft den Anforderungen moderner Entwicklung gewachsen.

Dependency Injection

Dependency Injection bezeichnet die Art und Weise, wie man die Initialisierung von Abhängigkeiten automatisieren kann und damit Redundanz vermeidet.

Aspektorientierte Programmierung mit Spring

Das Programmierparadigma der AOP erweitert die klassische objektorientierte Programmierung, in der Funktionen der Geschäftslogik in Modulen und Objekte gekapselt werden, um Aspekte. Aspekte drehen sich um die Handhabung von Aufgaben, die über das ganze Software-System hinweg an mehreren Stellen erledigt werden müssen: Logging oder Datenbankzugriffe gehören zu den charakteristischen Bestandteilen.

Spring Data, Hibernate und JPA

Mit Spring Data schafft das Framework eine Abstraktion über viele gängige Datenbankzugriffslösungen. Dank des einheitlichen Konzepts ist der Wechsel zwischen verschiedenen Lösungen unkomplizierter denn je.

Dependency Injection – Abkz: DI

- Wichtiges **Entwurfsmuster** - **Abhängigkeiten** werden zur **Laufzeit ausgelöst**
- Mit Dependency Injection ist es möglich – entsprechend dem **Single-Responsibility-Prinzip** – die Verantwortlichkeit für den Aufbau des Abhängigkeitsnetzes zwischen den Objekten eines Programmes aus den einzelnen Klassen in **eine zentrale Komponente** zu überführen.
- **Dependency Injection** überträgt die Verantwortung für das Erzeugen und die Verknüpfung von Objekten an eine **eigenständige Komponente**, wie beispielsweise ein **extern konfigurierbares Framework**. *Spring*
 - Dadurch wird der **Code des Objektes unabhängiger von seiner Umgebung**.
 - Das kann **Abhängigkeiten von konkreten Klassen beim Komplizieren vermeiden** und **erleichtert besonders die Erstellung von Unit-Tests**.
 - Bekannte Varianten: **Injection, Interface Injection und Setter Injection**
 - Konfigurations Varianten: **externe Datei** (zB. XML) oder **Annotationen (XML oder JAVA Programmierung)**

Wichtiges Entwurfsmuster

Nur, wer die Prinzipien von Dependency Injection bzw. Inversion of Control kennt, wird das Fundament des Spring Frameworks verstehen können. Das Spring Framework und die Dependency Injection sind zwei eng miteinander verknüpfte Begriffe. IoC – Inversion of Control bzw DI – Dependency Injection sind wichtiges **Entwurfsmuster**.

Zentrale Komponente

Abhängigkeiten werden zur **Laufzeit** ausgelöst. Die Grundlage ist das sogenannte Hollywoodprinzip: „Don't call us, we call you“. Das bedeutet, dass unsere Klasse die benötigten Komponenten von aussen injiziert bekommt, sie muss nur noch wissen, was sie braucht und sich nicht mehr selbst darum kümmern, wie sie die benötigten Komponenten bekommt.

Dependency Injection Framework

Der Dependency Injection Container garantiert, dass die Abhängigkeiten zur Laufzeit gesetzt werden und entkoppelt dadurch die Klasse von ihrer Laufzeitumgebung. Technisch funktioniert Dependency Injection so, dass die Klasse eine Membervariable für die benötigte Komponente beinhaltet und ein Konstruktorargument oder eine Setter-Methode bereitstellt, über die die Komponente dann injiziert werden kann.

Variante: Konfiguration der Komponenten über XML-Datei

In Spring erfolgt die Konfiguration der Komponenten über eine XML-Datei, in der die „Beans“ genannten Komponenten und ihre Abhängigkeiten zueinander deklariert werden. Spring liest dann beim Anwendungsstart diese Datei, erzeugt die „Beans“ (standardmäßig als Singletons) und verknüpft sie miteinander.

Variante: Konfiguration der Komponenten über Annotationen

Mittels Annotation kann der Konfigurationsaufwand über die XML-Dateien reduziert werden.

Beispiel – Benzin Teil des Autos

U: 21-Auto auto01

Zürcher Hochschule
für Angewandte Wissenschaften



- **Verantwortlichkeit**
 - Auto sollte egal sein, wie man Benzin herstellt
 - Besitzer besorgt Benzin an der Tankstelle
- **Probleme**
 - Hohe Kopplung (Benzin <-> fahren)
 - Schwierig zu testen (Benzin wird selber erzeugt)

```
public class Auto {  
  
    public void fahren () {  
        System.out.println("Auto fährt");  
        Benzin benzin = new Benzin();  
        benzin.verbrauchen();  
    }  
}  
  
Auto auto = new Auto();  
auto.fahren();
```

Verantwortlichkeiten

Viele Menschen fahren gerne Auto, weshalb es z. B. eine Klasse "Auto" gibt. Und die Methode "fahren" entsprechend. Zum Fahren braucht ein Auto immer "Benzin". Leider ist das Unglückliche an dem Fahren eines Autos, dass das Benzin auch verbraucht wird.

In der aktuellen Aufstellung ist das Ganze natürlich etwas unglücklich. In dem Moment, wo ich mir so ein neues Auto besorge, kann ich es fahren. Ich sehe jetzt allerdings nicht, wo das Benzin herkommt, was prinzipiell bedeutet: es gibt hier ein Verantwortlichkeitsproblem. Denn unglücklicherweise stellt das Auto im Beispiel sein Benzin selbst her. In der Realität besorgt der Besitzer das Benzin an der Tankstelle.

Probleme

Probleme, die sich hier ergeben sind, dass das Benzin und das Fahren direkt aneinander gekoppelt sind. Da diese Kopplung nach aussen unsichtbar ist, ist es auch schwierig zu testen, was unter bestimmten Verhältnissen passiert.

Beispiel – Benzin an Methode übergeben

U: 21-Auto auto02

- Auto kann auch ohne Benzin existieren
- Ohne Benzin kein Auto

```
public class Auto {  
  
    public void fahren(Benzin benzin) {  
        System.out.println("Auto fährt");  
        benzin.verbrauchen();  
    }  
}
```

```
Auto auto = new Auto();  
Benzin benzin = new Benzin();  
auto.fahren(benzin);
```

Auto kann auch ohne Benzin existieren

Der einfachste Weg ist, das Benzin schlicht und einfach zum Parameter der Methode zu machen. Nun erzeugt das Auto selbst kein Benzin mehr, sondern das Benzin wird vom Fahrer, bzw. demjenigen der das Auto nutzt, erzeugt und zum Fahren übergeben. Nichtsdestotrotz hat auch das einige Probleme, denn das Auto kann auch ohne Benzin existieren, man braucht es nur explizit zum Fahren.

Ohne Benzin kein Auto

Allerdings hat man ohne Benzin im Regelfall auch kein Auto, denn irgendwie muss es ja initial auch einmal fahren können, und sollte nicht jedes x-beliebige Benzin übergeben bekommen.

Beispiel – Benzin im Konstruktor

- Benzin ist fixer Bestandteil des Autos
- Instanzvariable definieren
- Übergabe von Benzin im Konstruktor

```
public class Auto {  
    private Benzin benzin;  
  
    public Auto (Benzin benzin) {  
        this.benzin = benzin;  
    }  
  
    public void fahren() {  
        System.out.println("Auto fährt");  
        benzin.verbrauchen();  
    }  
}  
  
Benzin benzin = new Benzin();  
Auto auto = new Auto(benzin);  
auto.fahren();
```

Um das Benzin zu einem fixen Bestandteil des Autos zu machen, deklariert man es schlicht und einfach als Variable und fordert es im Konstruktor. Zukünftig muss dann auch nicht im Aufrufen von "fahren" auf dem "Auto" dementsprechend irgendein "Benzin" mitgegeben werden. Die Anforderungen an das Auto können sich ändern, z. B. kann das Auto mit Diesel fahren bzw. ein Elektrofahrzeug benötigt elektrischen Strom.

Beispiel – Generalisierung von Kraftstoff

- **Anforderungänderung**
 - Autos können auch mit Strom fahren
 - Benzin wird nicht mehr benötigt
- **Umsetzen der Anforderungen**
 - Autos können zukünftig die Kraftstoffart wechseln
 - Sicher ist: verbrauchen findet immer statt

Anforderungsänderung

Das Benzin wird nicht mehr benötigt oder auch nur alternativ genutzt. Um diese Anforderungen sicherstellen zu können, muss eine Generalisierung der Kraftstoffart vorgenommen werden. Denn nun kann jedes Auto mit verschiedenen Kraftstoffarten fahren. Die einzige Sicherheit, die es dabei gibt ist dass auf jeden Fall der Kraftstoff verbraucht wird.

Umsetzen der Anforderungen

Der typische Weg um dieses Problem zu lösen, ist das Interface. Es kapselt, bzw. generalisiert die Anforderungen des Treibstoffs. Klar ist auch: Treibstoff muss sich verbrauchen lassen.

Beispiel – Generalisierung von Kraftstoff

- **Interface**

- Generalisiert die Anforderung eines Treibstoffs
 - Lässt sich verbrauchen

- **Vorteile**

- Einfache Kopplung anhand des Interface
 - Sehr einfach durch Mockups zu testen

```
public interface Kraftstoff{  
    public void verbrauchen();  
}  
  
public class Benzin implements Kraftstoff  
public class Strom implements Kraftstoff
```

```
Benzin benzin = new Benzin();  
Strom strom = new Strom();  
  
Auto auto1 = new Auto(benzin);  
Auto auto2 = new Auto(strom);
```

Interface

Um Benzin und Strom zu konkretisieren muss dieses Interface von beiden implementiert werden. In der Folge ist es möglich verschiedene Autos mit dementsprechend anderen Treibstoffarten zu erstellen.

Vorteile

Die Vorteile, die sich aus der Austauschbarkeit des Treibstoffs ergeben sind, dass man zu Testzwecken auch einen sogenannten Testtreibstoff implementieren kann. Diesen würde man auch als Mockup bezeichnen. Außerdem ist die Kopplung weniger statisch dank des Interfaces. Trotz allem ist dieses Konstrukt noch immer nicht 100 Prozent ideal zu nutzen, denn es gibt immer noch eine Menge statische Bindungen.

Beispiel – Dependency Injection

- **Probleme**

- Wartbarkeit
- Skalierbarkeit
- Jeder muss das Auto neu bauen

- **Injektion**

- Jede Abhängigkeit wird in ein Objekt injiziert

- **Ziel**

- Konstruktor: Erstellen der Abhängigkeit
- Methodenaufruf: in die Methode falls benötigt
- Feld: direkte Definition

```
Benzol benzol= new Benzol();
Ethanol ethanol = new Ethanol();
Hexan hexan = new Hexan();
Benzin benzin = new Benzin(benzol,
                           ethanol, hexan);
Oel oel = new Oel();
Auto auto = new Auto(benzin, oel);
```

Probleme

Diesen Problemen nimmt sich die *Dependency-Injection* an. Wenn man das beschriebene Beispiel weiterverwendet und deutlich den Use-Case erweitert, hat man das Problem, dass allein das Benzin relativ viele Bestandteile hat. Diese müssten jetzt immer explizit initialisiert und dem Benzin übergeben werden. Und die Menge an Anforderungen, die übergeben werden müssen, kann sich auch kontinuierlich steigern. Die Probleme, die sich ergeben ist, dass die Wartbarkeit des Codes extrem schwierig ist.

Ausserdem skaliert er relativ ungünstig, je nachdem, wie viele Bestandteile z. B. das Benzin hat. Und der wohl schlimmste Punkt ist, dass jeder Ort, der das Benzin oder auch das Auto nutzt, de facto das Objekt komplett bauen muss. Auch samt Abhängigkeiten.

Injektion

Dependency Injection oder auch der Begriff *Injektion* sorgen dafür, dass die Abhängigkeiten in ein Objekt selbst injiziert werden. D. h. insofern deklariert ist, dass das Benzin aus Benzol, Ethanol oder Hexan besteht, wird die Erstellung von den Instanzen dieser Klassen übernommen und dem Benzin bereitgestellt.

Wenn das Auto Benzin braucht, könnte beispielsweise das Benzin automatisch erzeugt werden, was wiederum die Bestandteile automatisch erzeugt. Allein diese Kette von Erstellungsprozessen, die normalerweise manuell wären, sind ein grosser Vorteil der *Dependency-Injection*.

Ziel

Die *Dependency-Injection* kann zur Bereitstellung bestimmte *Orte* einer Klasse betreffen. Das kann der *Konstruktor* sein, d. h. dass alle Abhängigkeiten für den Konstruktor erstellt werden. Das kann genauso gut ein Methodenaufruf sein, so dass bestimmte Teile des Aufrufs automatisch besorgt werden. Und es kann genauso gut auch ein *Feld* aus der Klasse selbst sein, wofür die *Dependency-Injection* die konkreten Instanzen bereitstellt

Spring Dependency Injection mit XML

- **Definition**
 - Die Art, wie Objekte (Beans) von einem Container für einen Vorgang zusammengebracht werden
- **Inversion of Control**
 - Abgabe von Aufgaben der Erzeugung von Objekten (**Kontrollverlust**)
- **Container**
 - Verwaltet Abhängigkeiten
 - Objekterstellung
 - Verlinkung

"Don't call me, I will call you."

Definition

Dependency-Injection erleichtert die Erzeugung von Klasseninstanzen mit vielen Abhängigkeiten und macht daraus einen einmaligen Konfigurationsprozess. Spring macht das anhand von XML oder Annotations-Konfiguration möglich. Mit Hilfe der Dependency-Injection und des Spring Frameworks können Sie Ihr eigenes Programmiervorgehen zukünftig deutlich einfacher gestalten. Prinzipiell gilt es erstmals zu verstehen, was Dependency-Injection im Kontext des Spring Framework bedeutet. Es ist die Art und Weise, wie man Objekte von einem Container aus in Zusammenhang bringt. Da Objekte oft Abhängigkeiten zueinander haben, sorgt die Dependency-Injection dafür, dass diese Abhängigkeiten korrekt aufgelöst werden.

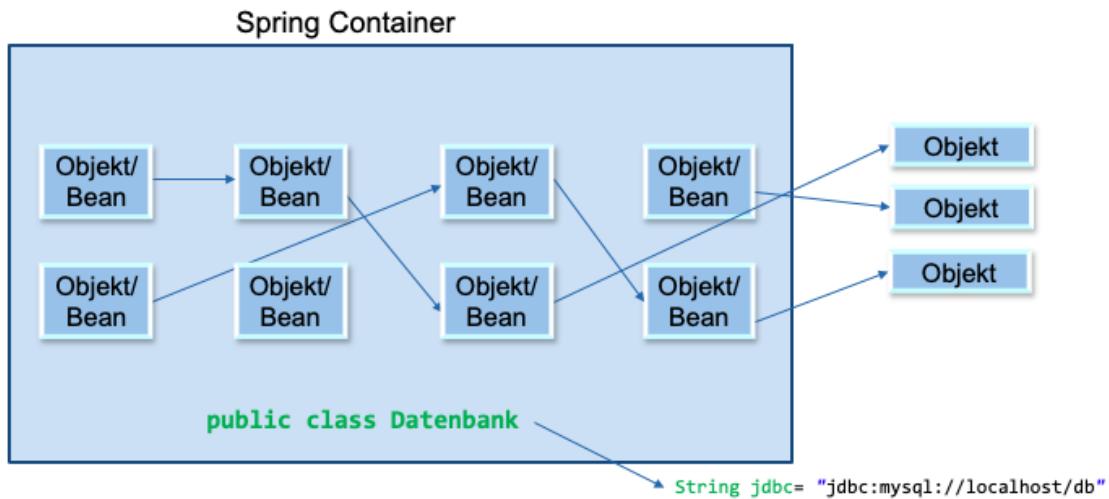
Inversion of Control

Oftmals spricht man auch von Inversion of Control. Das liegt daran, dass man die Aufgabe der Erzeugung von Objekten an eine externe Instanz übergibt. Dadurch hat man einen sogenannten Kontrollverlust.

Container

Der Container ist derjenige, der diese Abhängigkeiten verwaltet. Des Weiteren initialisiert er auch die Objekte, so dass sie genutzt werden können. Er verlinkt sie miteinander und damit ist der Container für den gesamten Injektionsprozess zuständig. Diesen Vorgang nutzt man auch oft mit dem in Hollywood typischen Ausspruch "Don't call me, I will call you."

Spring Container



BSc I Modul ASE1

LE 01 – Spring Framework | © 2020, InIT

12

Spring Container

Es gibt den Spring Container. Dieser erzeugt zur Laufzeit verschiedene Objekte, die man auch oftmals als Bean bezeichnet. Und genau diese Objekte/Bean haben oftmals auch Objekte, die ausserhalb des Spring Containers liegen, z. B. Instanzen eines Strings. Im Rahmen des Initialisierungsprozesses sorgt der Spring Container dafür, dass die Objekte/Beans mit den Objekten, die einerseits ausserhalb des Containers liegen verbunden werden und andererseits auch mit den Objekten/Beans, die innerhalb des Containers liegen.

Automatische Erzeugung anhand von Deklarationen

All das passiert vollkommen automatisch und anhand von Deklarationen. Betrachtet man die Beispiel-Bean-**Datenbank** hat diese beispielsweise einen jdbc-String der ausserhalb des Spring Containers liegt. Der Spring Container verlinkt beide Objekte bei der Initialisierung und sorgt weiterhin dafür, dass die Datenbank eine DataSource hat, die wiederum auch eine andere Bean im Container ist. Damit eine Klasse eine Bean sein kann, muss sie verschiedene Dinge erfüllen. Zunächst einmal handelt es sich bei einer Bean immer konkret um ein Objekt.

Dieses Objekt wird im Regelfall immer anhand einer Konfiguration fremderzeugt. Fremderzeugt heisst in diesem Fall z. B. dass der Container diese Bean erzeugt. Eine Bean unterliegt auch unterschiedlichen Lebenszeiten und verschiedenen Events, die innerhalb ihrer Lebenszeit passieren, z. B. wird sie einerseits initialisiert und mit den jeweiligen Abhängigkeiten versorgt und andererseits wird auch wieder zerstört wenn der Kontext beendet wird.

- **Objekt**
 - Objekt, das fremderzeugt wird
- **Lebenszeit**
 - Unterliegt verschiedenen Lebenszyklen (kontrolliert vom Container)
- **Eigenschaften**
 - Öffentlicher argumentloser Konstruktor
 - Serialisierbarkeit
 - Öffentliche Setter/Getter

Objekt

Damit eine Klasse eine Bean sein kann, muss sie verschiedene Dinge erfüllen. Zunächst einmal handelt es sich bei einer Bean immer konkret um ein Objekt. Dieses Objekt wird im Regelfall immer anhand einer Konfiguration fremderzeugt. Fremderzeugt heisst in diesem Fall z. B. dass der Container diese Bean erzeugt.

Lebenszeit

Eine Bean unterliegt auch unterschiedlichen Lebenszeiten und verschiedenen Events, die innerhalb ihrer Lebenszeit passieren, z. B. wird sie einerseits initialisiert und mit den jeweiligen Abhängigkeiten versorgt und andererseits wird auch wieder zerstört wenn der Kontext beendet wird. Damit eine Klasse eine Bean werden kann, muss sie bestimmte Eigenschaften erfüllen.

Eigenschaften

Zunächst einmal ist es wichtig, einen öffentlichen und argumentlosen Konstruktor zu haben, denn sonst ist es für den Container nicht möglich, eine Instanz zu erstellen. Des Weiteren sollte sie im Regelfall serialisierbar sein, damit sie ausgetauscht werden kann. Das ist allerdings eine Eigenschaft, die nicht zwangsläufig für jede Bean gelten muss. Damit die Bean mit den jeweiligen Abhängigkeiten versorgt werden kann, ist es im Regelfall notwendig öffentliche Setter und Getter zu haben. Damit Spring weiß, dass es sich um eine Bean handelt, ist es notwendig, Spring darauf hinzuweisen mithilfe einer sogenannten Konfigurationsdatei bzw. Annotation.

Bean Konfiguration – Property

U: 22-HelloWorld

```
<?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/spring-  
       beans-3.0.xsd">  
  
    <bean id="helloWorld" class="edu.spring.HelloWorld">  
        <property name="message" value="Hello World!"/>  
    </bean>  
  
</beans>
```

initialisiert

```
public static void main(String[] args) {  
    ApplicationContext context = new  
        ClassPathXmlApplicationContext("beans.xml");  
    HelloWorld obj = (HelloWorld)  
        context.getBean("helloWorld");  
    obj.getMessage();  
}
```

BSc I Modul ASE1
Zürcher Fachhochschule

LE 01 – Spring Framework | © 2020, InIt

```
package edu.spring;  
  
public class HelloWorld {  
    private String message;  
  
    public void setMessage(String message){  
        this.message = message;  
    }  
  
    public void getMessage(){  
        System.out.println(  
            "Your Message : " + message);  
    }  
}
```

→ prints "Your Message : Hello World"

14

Konfigurationsdatei

Damit Spring weiß, dass es sich um eine Bean handelt, ist es notwendig, Spring darauf hinzuweisen. Dies geschieht mithilfe einer sogenannten Konfigurationsdatei bzw. einer Annotation. Im Regelfall handelt es sich dabei um eine XML-Datei. Diese beinhaltet die Information "beans". Wichtig hier ist, dass die jeweiligen Namensbereiche des XML-Files deklariert sind.

Main - HelloWorld

Hat man eine Klasse wie beispielsweise HelloWorld, ist es möglich mit der XML-Anweisung "bean" und der Vergabe einer bestimmten ID und den jeweiligen Klassenreferenz-Link eine Bean-Instanz dieser Klasse zu erstellen. Um die Abhängigkeit von Message umzusetzen gibt es die Möglichkeit die property zu deklarieren. Bei der Initialisierung wird Spring automatisch die Nachricht "Hello, World!" in die Instanz von HelloWorld übertragen.

Ruft man die Bean vom Applikationskontext ab und nutzt die Funktionalität, ist der jeweilige String bereits initialisiert.

XML-Namespaces

Zürcher Hochschule
für Angewandte Wissenschaften



Namensraum	Aufgabe
aop	Enthält Elemente, um Aspekte zu deklarieren und automatisch Proxes für @AspectJ-annotierte Klassen als Spring-Aspekte zu erstellen.
beans	Der primitive Kernnamensraum für Spring, mit dem die Deklaration von Beans und der Verschaltung ermöglicht wird.
context	Enthält Elemente für die Konfiguration des Spring-Applikationskontexts einschliesslich der Möglichkeit für die automatische Erkennung und Verschaltung von Beans und die Injektion von Objekten, die nicht direkt von Spring verwaltet werden.
jee	Bietet die Integration mit Java-EE-APIs wie JNDI und EJB.
jms	Bietet Konfigurationselemente zur Deklarierung von nachrichtengetriebenen POJOs.
lang	Ermöglicht die Deklaration von Beans, die als Groovy-, JRuby- oder Bean-Shell-Skripts implementiert sind.
mvc	Ermöglicht Spring-MVC-Fähigkeiten wie annotationsorientierte Controller, View-Controller und Interceptors.
oxm	Unterstützt die Konfiguration des Mappings von Objekten zu XML in Spring.
tx	Bietet deklarative Transaktionskonfiguration.
util	Eine Sammlung mit verschiedenen Utility-Elementen. Enthält die Möglichkeit, Collections als Beans zu deklarieren, und Support für Eigenschaftsplatzhalterelemente.

BSc I Modul ASE1

Zürcher Fachhochschule

LE 01 – Spring Framework | © 2020, IfIT

15

XML Namespaces

Seit Spring 3.0 gibt es zwei Möglichkeiten, Beans im SpringContainer zu konfigurieren: Traditionellerweise wird die Spring Konfiguration in einer oder mehreren XML-Dateien definiert. Doch Spring 3.0 bietet auch eine Javabasierte Konfigurationsoption. Wir konzentrieren uns hier auf die traditionelle XML-Option, die java-basierte Konfiguration von Spring sehen wir uns in später an. Wenn man Beans in XML deklariert, ist das Root-Element der Spring-Konfigurationsdatei das Element <beans> aus dem Beans-Schema von Spring.

In Spring sieht eine typische **XML Konfigurationsdatei** wie folgt aus:

```
<?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 declarations go here -->

</beans>
```

Innerhalb von <beans> können Sie Ihre gesamte Spring-Konfiguration einschliesslich der <bean>-Deklarationen platzieren. Doch der Beans-Namensraum ist nicht der einzige, dem Sie in Spring begegnen werden. Insgesamt hat das Kern-Framework von Spring aktuell zehn Konfigurationsnamensräume.

Neben den im Spring Framework enthaltenen Namensräumen bringen auch viele Mitglieder des Spring-Portfolios (z. B. Spring Security, Spring Web Flow und Spring Dynamic Modules) ihre eigenen Spring-Konfigurationsnamensräume mit. Wir werden weitere Namensräume von Spring im Verlauf kennenlernen. Doch nun wollen wir diesen auffallend leeren Raum in der Mitte der XML-

Konfiguration mit einigen <bean>Elementen innerhalb von <beans> füllen.

Mit einem Anwendungskontext arbeiten

- **ClassPathXmlApplicationContext** ressource
 - Lädt eine Kontextdefinition aus einer im Klassenpfad abgelegten XML-Datei. Hierbei werden Kontextdefinitionsdateien als Klassenpfadressourcen betrachtet.
- **FileSystemXmlApplicationContext** irgendwo im FS
 - Lädt eine Kontextdefinition aus der XML-Datei im Dateisystem.
- **XmlWebApplicationContext**
 - Lädt Kontextdefinitionen aus einer XML-Datei, die Bestandteil einer Webanwendung sind.

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("foo.xml");
```

ApplicationContext Varianten

Das Laden eines Anwendungskontexts aus dem Dateisystem oder dem Klassenpfad ähnelt dem Laden von Beans in eine BeanFactory. Einen FileSystemXmlApplicationContext laden Sie etwa wie folgt:

```
ApplicationContext context = new  
FileSystemXmlApplicationContext("c:/foo.xml");
```

Ähnlich können Sie einen Anwendungskontext mit ClassPathXmlApplicationContext aus dem Klassenpfad der Anwendung heraus laden:

```
ApplicationContext context = new  
ClassPathXmlApplicationContext("foo.xml");
```

Der Unterschied zwischen diesen Gebrauchsvarianten von FileSystemXmlApplication-Context und ClassPathXmlApplication-Context besteht darin, dass FileSystemXmlApplication-Context an einer bestimmten Position im Dateisystem nach foo.xml sucht, während ClassPathXmlApplication-Context den Klassenpfad (einschliesslich vorhandener JAR-Dateien) nach foo.xml durchstöbert.

Mit einem Anwendungskontext an der Hand können Sie Beans aus dem Spring-Container auslesen, indem Sie die getBean()-Methode des Kontexts aufrufen. Nachdem Sie die Grundlagen zur Erstellung eines Spring-Containers kennen, wollen wir den Lebenszyklus einer Bean im Bean-Container näher unter die Lupe nehmen.

Bean Konfiguration - Klasse

```

<?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="dreieck" class="edu.spring.dreieck01.Dreieck">
        <property name="start" ref="startPunkt"></property>
    </bean>
    <bean id="startPunkt" class="edu.spring.dreieck01.Punkt">
        <property name="x" value="0"></property>
        <property name="y" value="1"></property>
    </bean>
</beans>

Dreieck obj = (Dreieck) context.getBean("dreieck");

```

2 Beans

```

public class Dreieck {
    private Punkt start;

    public Punkt getStart() {
        return start;
    }

    public void setStart(Punkt
                         start) {
        this.start = start;
    }

    @Override
    public String toString() {
        return "Dreieck [start=" +
               start + "]";
    }
}

```

Klassen-Initialisierung

Ein wenig komplizierter ist das Ganze, wenn es sich um Klassen handelt, die initialisiert werden müssen. In diesem Fall gibt es ein Dreieck, das abhängig ist von einem Punkt. Um diesen Punkt aufzubauen, würde man zunächst einmal die Namespaces deklarieren und hätte dann entsprechend eine Klasse "Punkt". Diese wird auch über eine Bean initialisiert.

Und um diesen Punkt für das Dreieck zu nutzen, kann man anhand der Referenz die Verbindung herstellen. Damit befinden sich beide Instanzen innerhalb des Spring Containers. Durch die simple Abfrage `getBean("dreieck")` wird das vollständig initialisierte Dreieck abgerufen.

Bean Konfiguration – Klasse (**Innere Bean**)

- Ref-Tag ersetzen
 - Bean unmittelbar in Bean erzeugen
- Empfehlung
 - Wenn nur eine Property genutzt wird, direkt erzeugen

```
<?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/spring-beans-3.0.xsd">

    <bean id="dreieck" class="edu.spring.dreieck01.Dreieck">
        <property name="start">
            <bean id="startPunkt" class="edu.spring.dreieck01.Punkt">
                <property name="x" value="0"/>
                <property name="y" value="1"/>
            </bean>
        </property>
    </bean>
</beans>
```

BSc I Modul ASE1
Zürcher Fachhochschule

LE 01 – Spring Framework | © 2020, IfIT

18

Ref-Tag

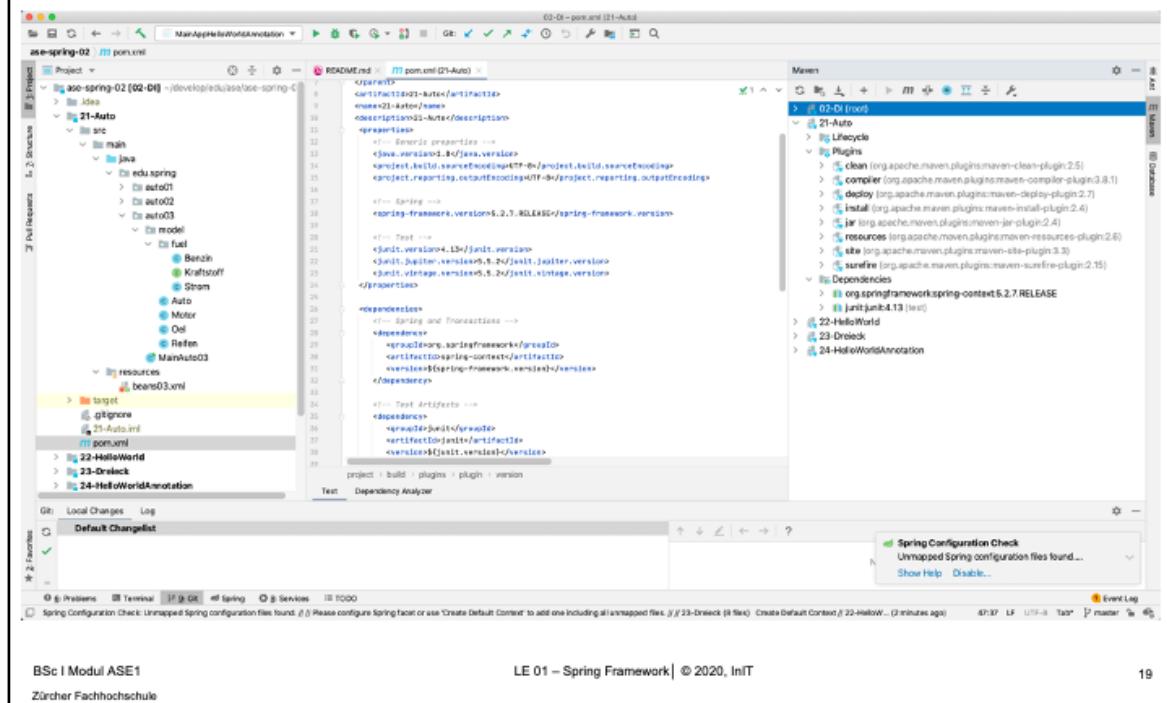
Alternativ ist es auch möglich, die Bean direkt zu deklarieren, also ohne die Referenz. In diesem Beispiel deklariert man also das Dreieck und man deklariert die Bean selbst innerhalb der Property. Dadurch erspart man sich den Ref-Tag. Der Ref-Tag kann ersetzt werden.

Empfehlung

Das Vorgehen führt allerdings zu einer unglaublich tiefen Verschachtelung und je nach Anwendungszweck sollte man sich überlegen, ob man genau diese Möglichkeiten nutzen möchte.

Dependency Injection – Beispiel

U: 21-Auto auto03



POM-Datei

Zunächst einmal sollte sichergestellt sein, dass die korrekten Abhängigkeiten eingebunden sind. Die richtige Abhängigkeit ist hier der Spring-Kontext, der hauptsächlich für die Dependency Injektion zuständig ist.

```
<!-- Spring -->
<spring-framework.version>4.3.2.RELEASE</spring-framework.version>

<!-- Spring and Transactions -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring-framework.version}</version>
</dependency>
```

Dependency Injection – Beispiel

U: 21-Auto auto03

```
<?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="A3" class="edu.spring.auto03.model.Auto">
        <property name="name" value="A3"></property>
        <property name="reifen">
            <list>
                <bean class="edu.spring.auto03.model.Reifen" />
                <bean class="edu.spring.auto03.model.Reifen" />
                <bean class="edu.spring.auto03.model.Reifen" />
                <bean class="edu.spring.auto03.model.Reifen" />
            </list>
        </property>
        <property name="motor" ref="a3_motor" />
        <property name="kraftstoff" ref="benzin" />
    </bean>
    <bean id="a3_motor" class="edu.spring.auto03.model.Motor" autowire="byName" />
    <bean id="benzin" class="edu.spring.auto03.model.fuel.Benzin" scope="prototype" />
    <bean id="oel" class="edu.spring.auto03.model.Oel" scope="prototype" />
</beans>
```

BSc I Modul ASE1
Zürcher Fachhochschule

LE 01 – Spring Framework | © 2020, InIT

20

Ohne scope=prototype
ist die erzeugte Instanz
ein Singleton!

Motor benutzt Oel welches das
Property oel hat, wird anhand
dessen Namens automatisch
gefunden

beans.xml

Die gesamte Konfiguration der Beans befindet sich in der XML-Datei.

Dort gibt es eine Bean mit dem Namen "A3", die aus dem Auto erstellt wird. Weiterhin wird ein Property definiert, dass vier Reifen erzeugt über direkte Bean-Einbindung innerhalb des Properties. Und dann gibt es noch eine Referenz auf den "a3_motor" der entsprechend hier initialisiert wird. Über "autowire" besorgt er sich seine Abhängigkeiten. Und der Kraftstoff wird entsprechend hier unten initialisiert, mit "prototype", so dass immer ein neuer Kraftstoff entsteht. Um sich einen solchen "A3" erstellen zu können, ist das Vorgehen relativ einfach.

Dependency Injection – Beispiel

U: 21-Auto auto03

```
public class Auto {  
  
    private String name;  
    private Motor motor;  
    private Kraftstoff kraftstoff;  
    private List <Reifen> reifen;  
  
    public void fahren (Benzin benzin){  
        System.out.println("Auto fährt");  
        benzin.verbrauchen();  
    }  
    ...  
}
```

```
public class Motor {  
    private Oel oel;  
  
    ...  
}
```

```
Auto obj = (Auto) context.getBean("A3");  
System.out.println(obj.toString());
```

Um sich einen solchen "A3" erstellen zu können, ist das Vorgehen relativ einfach.

Man lädt den Applikationskontext und die Konfigurationsdatei dazu, lässt sich die Bean "A3" erzeugen und gibt diese, in diesem Fall auf der Konsole aus. Bei der Ausführung wird automatisch das Auto erzeugt, das entsprechend die Abhängigkeiten beinhaltet. Die Besonderheit anhand der XML-Konfiguration ist, dass in den *Beans* selbst nichts weiter mit Annotationen definiert ist. Alle Abhängigkeiten werden anhand der XML-Datei aufgelöst. Sie haben gesehen, wie einfach es durch das Zusammenspiel von *Dependency-Injection* und Spring ist, jetzt selbst komplexeste Objekte auf einfachste Art und Weise zu erzeugen.

Bean Scope

- Scope

- Bestimmt, wie Beans von Spring zurückgegeben werden

- Standardscope

- Singleton

- Normalfall

- Singleton & Prototype

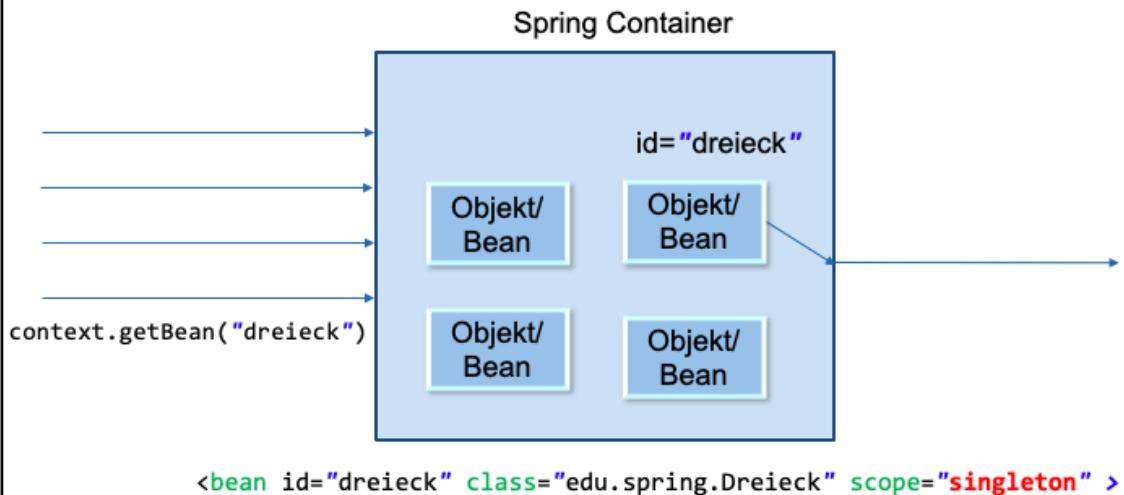
Scope	Beschreibung
singleton	Eine einzige Instanz über die gesamte Anwendung (default)
prototype	Erzeugt eine beliebige Menge Instanzen
request	Erzeugt eine Instanz für einen HTTP-Request
session	Erzeugt eine Instanz für eine HTTP-Session
global-session	Erzeugt eine Instanz für eine globale HTTP-Session

Für die Entwicklung mit *Dependency-Injection* und dem Spring Framework werden in den folgenden Folien noch einige Tipps und Tricks vermittelt. Zunächst einmal, wann immer Sie eine Bean erstellen, gibt es einen sogenannten Bean Scope. Der Bean Scope bezeichnet, wann und wie Beans von Spring zurückgegeben werden und ob neue Instanzen für dieses Beans erzeugt werden müssen. Es gibt fünf verschiedenen Arten des Bean-Scopes, allen voran der Singleton. Beim Singleton handelt es sich um den standardmäßig von Spring zurückgegebenen Scope d. h. wann immer Sie eine Bean deklarieren wird nur eine einzige Instanz zur Laufzeit erzeugt und bei jeder Abfrage diese Instanz zurückgegeben.

Bei der Nutzung des Scopes Prototype wird für jeden Fall der Abfrage eine neue Instanz zurückgegeben. Die anderen drei Fälle beziehen sich vor allem auf das Verhalten im Web. Wenn ein Request beispielsweise an das Spring Framework gestellt wird, so erzeugt es für jeden dieser Requests eine neue Instanz. Für den Request selber bleibt er aber über den gesamten Programmcode hinweg gleich. Bei einer Session, die eben für einen User der z. B. eine Internetseite aufruft, vergeben wird, bleibt für genau diesen User innerhalb seiner Session der Scope auch immer gleich auf dem jeweiligen Bean.

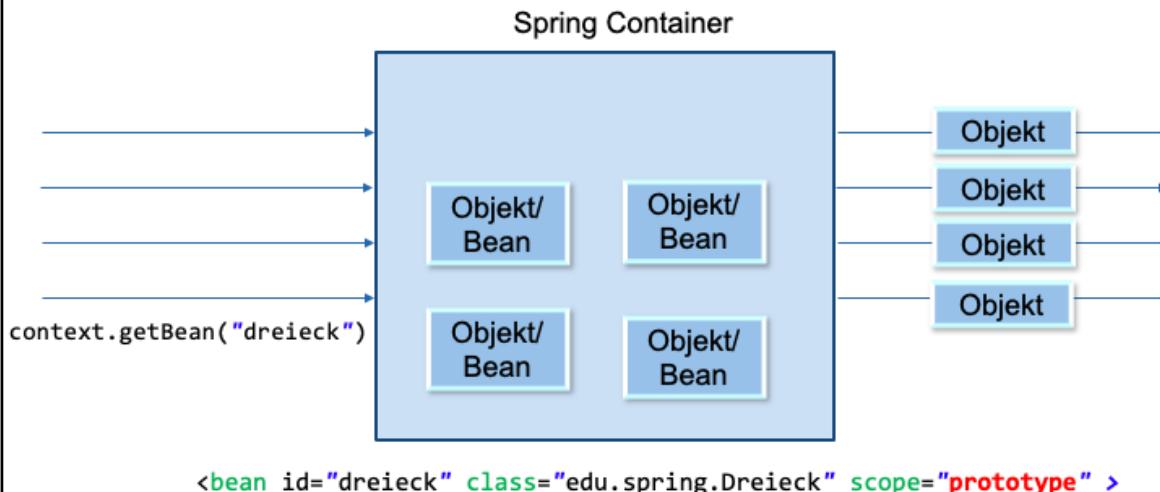
Respektive ähnlich funktioniert das für eine sogenannte Globale Session, die dann oftmals anwendungsübergreifend angesiedelt ist.

Spring Container – Scope Singleton



Wenn man sich nochmal diesen Singleton Scope illustrieren möchte, so gibt es den sogenannten Spring Container in dem sich die jeweiligen Objekte und Beans befinden. Wenn man von aussen eine Abfrage gegen diesen Spring Container macht, wird mit Hilfe der getBean-Methode und einer jeweiligen ID vom Container im Fall des Singleton-Scopes die bestimmte Bean gesucht.

Spring Container – Scope Prototype



Scope Prototype

Beim anderen Fall, dem Prototype-Scope kommen wieder mehrere Anfragen gegen den Spring Container und statt aus der vorhandenen Menge entsprechend die Bean zu **suchen initialisiert er für jede Abfrage das Objekt neu**. Die Angabe findet dann entsprechend durch "scope =prototype" statt. Für den Namen einer Bean kann mit Hilfe des Alias-Ausdrucks auch ein alternativer Name vergeben werden unter dem die Bean dann auch ansprechbar ist. Möglich ist das Ganze über eine Angabe im XML.

Spring Collection

- Listen erzeugen

Element	Description
<list>	Unterstützt die Injektion von Werten, doppelte Werte sind erlaubt.
<set>	Unterstützt die Injektion von Werten, doppelte Werte sind NICHT erlaubt.
<map>	Unterstützt die Injektion einer Collection von Name-Value-Paaren, wobei Name und Value jeder beliebiger Daten-Typ sein kann.
<props>	Unterstützt die Injektion einer Collection von Name-Value-Paaren, wobei Name und Value vom Typ String sind.

```
public class Dreieck {  
    private List<Punkt> punkte;
```

```
<bean id="dreieck"  
      class="edu.spring.Dreieck">  
    <property name="punkte">  
      <list>  
        <ref bean="punkt1" />  
        <ref bean="punkt2" />  
      </list>  
    </property>  
</bean>  
  
<bean id="punkt1"  
      class="edu.spring.Punkt">  
    <property name="x" value="1" />  
    <property name="y" value="1" />  
</bean>  
...
```

Listen erzeugen

Wenn man z. B. ein Dreieck initialisiert und diesem Dreieck mehrere Punkte übergibt, würde man dies mit Hilfe sogenannter Listentypen machen. Diese sind ähnlich wie in Java selbst: Set, List, Map und entsprechend auch Properties. Anhand der Angabe List innerhalb des XML, kann die jeweilige Menge an Beans entsprechend angegeben werden.

- Konfigurationsaufwand
 - Automatisierte Auflösung von Referenzen
- **Typen**
 - byName
 - byType
 - Constructor

```
public class Dreieck {  
    private Punkt punktA, PunktB, PunktC;  
  
<bean id="dreieck"  
      class="edu.spring.Dreieck"  
      autowire="byName"  
/>  
  
<bean id="punktA"  
      class="edu.spring.Punkt">  
  <property name="x" value="1" />  
  <property name="y" value="1" />  
</bean>  
...
```

Konfigurationsaufwand

Spring ist dazu ausserdem in der Lage automatisch Zusammenhänge zu erraten, auch als Autowiring bezeichnet. Durch Autowiring werden automatisch Referenzen auf andere Objekte aufgelöst.

Typen

Das Ganze kann entweder byName passieren, also durch Referenz auf den gleichen Namen wie in diesem Beispiel würde jede ID "punktA" aufgelöst werden und entsprechend als Variable auf dem Dreieck hinterlegt. Andere Möglichkeiten sind auch via Constructor oder byType.

Spring Vererbung

U: 23-Dreieck inheritance

- Parent
 - Übernimmt Deklaration von einer Bean gleichen Typs

```
public class Dreieck {  
  
    protected Punkt punktA;  
    ...  
  
    public class Dreieck2 extends Dreieck {  
  
        private Punkt punktB;  
        ...
```

```
<bean id="dreieck"  
      class="edu.spring.Dreieck">  
    <property name="punktA" ref="punktA" />  
  </bean>  
  
<bean id="dreieck2"  
      class="edu.spring.Dreieck2"  
      parent="dreieck">  
    <property name="punktB" ref="punktB" />  
  </bean>  
  
<bean id="punktA" class="edu.spring.Punkt">  
  <property name="x" value="1" />  
  <property name="y" value="1" />  
</bean>  
  
<bean id="punktB" class="edu.spring.Punkt">  
  <property name="x" value="2" />  
  <property name="y" value="2" />  
</bean>
```

Parent

Weiterhin ist es mit Spring-XML auch möglich Vererbung abzubilden und sogenanntes Templating. Durch die Angabe eines parents kann man eine Bean einmalig deklarieren und dementsprechend auch wieder verwenden. Im hier gezeigten Fall würde ein Dreieck erzeugt werden, dass bereits den Punkt "A" definiert, das Dreieck "2" würde diese Angabe übernehmen und zusätzlich den Punkt "B" definieren.

- BeanNameAware
 - Enthält den eigenen Namen
- ApplicationContextAware
 - Enthält den übergeordneten Applikationskontext via Setter

```
public class Dreieck implements  
    BeanNameAware,  
    ApplicationContextAware {  
    private ApplicationContext  
        applicationContext;  
    private String beanName;  
    private List<Punkt> punkte;  
  
    public void setApplicationContext () ..  
    public void setBeanName () ..  
  
    @Override  
    public String toString() {  
        return "Dreieck [punkte=" + punkte +  
            ", applicationContext=" +  
            applicationContext +  
            ", beanName=" + beanName + "]";  
    }  
}
```

```
Dreieck [punkte=[Punkt [x=1, y=1], Punkt [x=2, y=2]],  
applicationContext=org.springframework.context.support.ClassPathXmlApplicationContext@f  
fa5d: startup date [Fri Aug 26 11:07:01 CEST 2016];  
root of context hierarchy, beanName=dreieck]
```

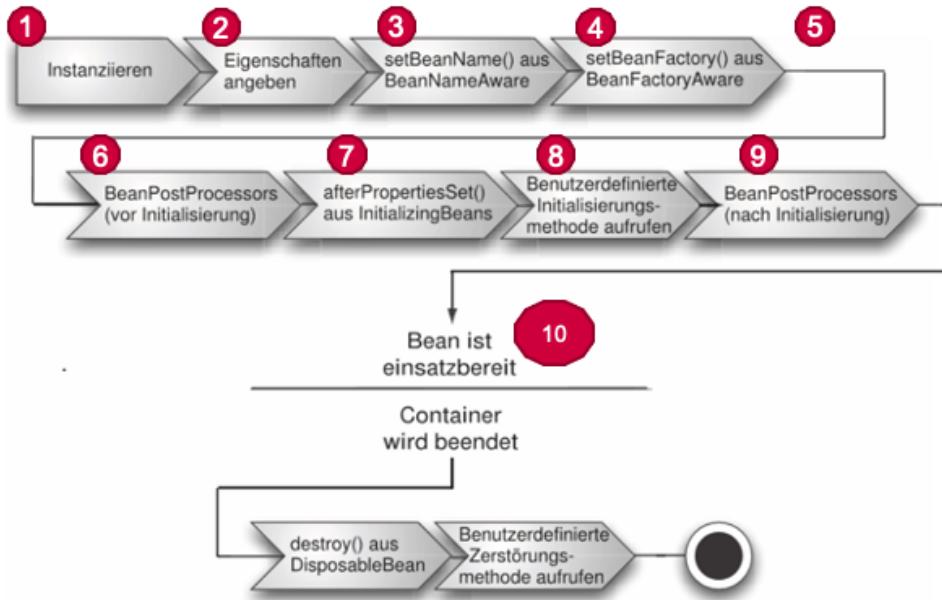
BeanNameAware

Jede Bean kann sich auch über bestimmte Statusveränderungen informieren lassen, z. B. ist es möglich mit dem Interface BeanNameAware den Namen der Bean selbst zu erhalten.

ApplicationContextAware

Mit ApplicationContextAware ist es möglich den Applikationskontext sich in die eigene Klasse geben zu lassen.

LifeCycle Events



In einem SpringContainer ist der Lebenszyklus einer Bean abwechslungsreich. Sie müssen den Lebenszyklus einer Spring-Bean kennen, denn schliesslich wollen Sie ja auch von dem einen oder anderen Vorteil profitieren, den Spring bei der Erstellung einer Bean bietet. Die Grafik zeigt den Beginn des Lebenszyklus einer normalen Bean, wenn diese in einen Spring-Anwendungskontext geladen wird. Wie Sie sehen, führt eine BeanFactory verschiedene Konfigurationsschritte durch, bis eine Bean einsatzbereit ist.

1. Spring instanziiert die Bean.
2. Spring injiziert Werte und Bean-Verweise in die Eigenschaften der Bean.
3. Wenn die Bean *BeanNameAware* implementiert, vergibt Spring die ID der Bean an die Methode *setBeanName()*.
4. Wenn die Bean *BeanFactoryAware* implementiert, ruft Spring die *setBeanFactory()* Methode auf und über gibt die *BeanFactory* selbst.
5. Wenn die Bean *ApplicationContextAware* implementiert, wird Spring die Methode *setApplicationContext()* aufrufen und eine Referenz an den einkapselnden Anwendungskontext übergeben.
6. Wenn irgendeine der Beans die *BeanPostProcessor*-Schnittstelle implementiert, ruft Spring dessen Methode *postProcessBeforeInitialization()* auf.
7. Wenn irgendeine der Beans die *InitializingBean*-Schnittstelle implementiert, ruft Spring dessen Methode *afterPropertiesSet()* auf.
8. Wenn die Bean entsprechend mit einer *init-method* deklariert wurde, wird dann die spezifizierte Initialisierungsmethode aufgerufen.
9. Falls es Beans gibt, die *BeanPostProcessor* implementieren, ruft Spring dessen Methode *postProcessAfterInitialization()* auf.
10. An dieser Stelle ist die Bean bereit für den Einsatz durch die Anwendung. Sie verbleibt im Anwendungskontext, bis der Anwendungskontext zerstört wird.
11. Wenn irgendeine der Beans die *DisposableBean*-Schnittstelle implementiert, ruft Spring dessen *destroy()* Methoden auf.
12. Wenn die Bean entsprechend mit einer *destroy-method* deklariert wurde, wird dann die

spezifizierte Initialisierungsmethode aufgerufen.

- Ereignisse
 - Initialisierung und Cleanup
- Varianten
 - Via Interface: Initializing Bean, Disposable Bean
 - Via XML: init-method, destroy-method
 - (erst Interface, dann XML)

```
public class Dreieck implements  
InitializingBean, DisposableBean {  
  
    private List<Punkt> punkte;  
  
    public void afterPropertiesSet() ...  
    public void destroy() ...  
    public void init() ...  
    public void mydestroy() ...
```

```
<bean id="dreieck" class="edu.spring.Dreieck"  
    init-method="init"  
    destroy-method="mydestroy">
```

```
afterPropertiesSet  
init  
Dreieck [punkte=[Punkt [x=1, y=1], Punkt [x=2,  
y=2]]]  
doClose  
destroy  
mydestroy
```

```
AbstractApplicationContext context = ...  
Dreieck obj=(Dreieck)context.getBean("dreieck");  
System.out.println(obj.toString());  
context.registerShutdownHook();  
context.close();
```

BSc I Modul ASE1
Zürcher Fachhochschule

LE 01 – Spring Framework | © 2020, InIT

30

Ereignisse

Eine Bean unterliegt auch verschiedenen Ereignissen in ihrem Lebenszyklus. Diese werden auch als *Lifecycle-Events* bezeichnet. Zum Beispiel, wenn die Bean initialisiert wird oder auch wenn die Bean zerstört wird.

Varianten

Dafür gibt es unterschiedliche Möglichkeiten die Bean entweder via Interface, über InitializingBean oder DisposableBean zu informieren, als auch mit XML, durch die Deklarationen der *init-method* und der *destroy-method*, die jeweils auf der Klasse deklariert sein muss.

Bei der Reihenfolge des Aufrufs sollte man beides verwenden. Hier wir zuerst das Interface bevorzugt und danach XML.

BeanPostProcessor

U: 23-Dreieck beanspostprocessor

- Wird von und nach der Initialisierung von Beans aufgerufen

```
public class InitDreieckApp implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        System.out.println("BeforeInitialization : " + beanName);
        return bean; // you can return any other object as well
    }

    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        System.out.println("AfterInitialization : " + beanName);
        return bean; // you can return any other object as well
    }
}

<bean class="edu.spring.dreieck.xml.beanspostprocessor.InitDreieckApp" />
```

BeforeInitialization : punktA
AfterInitialization : punktA
BeforeInitialization : punktB
AfterInitialization : punktB

Spring DI mit Annotationen

- **Annotationen**
 - Definition im Code
 - Definition ausserhalb in XML
- **Kominierbarkeit**
 - Annotationen vor XML DI (XML überschreibt)

Annotationen

Anders als mit XML kann mithilfe von Annotationen die Konfiguration von Dependency Injection auch unmittelbar im Code vorgenommen werden: mit @Component, @Autowired und @Configuration.

XML ist gerade für Programmierer nicht immer die angenehmste Art und Weise die Anwendung zu konfigurieren. Aus genau diesem Grund lässt sich Spring auch auf verschiedene Arten und Weisen konfigurieren, zum Beispiel über die Möglichkeit direkt im Code mit Hilfe von Annotationen. Grundsätzlich gibt es zwei verschiedene Arten und Weisen, wie Spring definiert werden kann. Einerseits im Code und andererseits ausserhalb mit Hilfe von XML.

Kominierbarkeit

Wenn man beide kombiniert, dann wird im Regelfall die Annotation immer vor der XML-Datei benutzt. Alles was die XML zusätzlich deklariert wird dann überschrieben. Damit bestünde die Möglichkeit einerseits auf der Programmierer-Ebene Spring selbst zu deklarieren und anderen Nutzern es ausserhalb zu ermöglichen Komponenten anhand von XML-Konfigurationen auszutauschen.

Aktivierung – XML mit Annotationen

```
<context:annotation-config />
<context:component-scan base-package=edu.spring.xxxxx
```

- Annotation Config
 - Konfiguriert anhand der spezifizierten Annotationen
- Component Scan
 - Sucht nach Klassen mit Annotationen

Annotation Config

Wie aktiviert man diese Vorgänge? Wenn man eine Teils-Teils-Lösung benutzt, würde man einerseits im XML mithilfe von **context:annotation-config** einen Prozess anstoßen, der anhand von Annotationen bestimmte Dinge initialisiert.

Component Scan

Oft in diesem Zusammenhang verwendet ist auch der Kontext Component-Scan. Dieser sucht über die verschiedenen Klassen hinweg nach den bestimmten Annotationen, wie z.B. "component".

```
<context:component-scan base-package=edu.spring.xxxxx
```

Container Konfiguration

- **Zweck**
 - Konfigurieren einer Bean Registry **ausserhalb von XML**
- **@Configuration**
 - Container der Beans beinhaltet
 - Wird an AnnotationConfigApplicationContext übergeben
- **@Bean**
 - Deklaration einer Bean

Zweck

Um konkret im Code einen Container zu erstellen, so wie er in XML existiert, würde man die Container-Konfiguration nutzen. Sie sammelt, genau wie die XML-Container-Deklaration Beans innerhalb einer eigenen Registry.

@Configuration

Die Annotation hierfür ist @Configuration. Damit wird aus der jeweiligen Klasse ein Container, der Beans beinhaltet. Statt beispielsweise einen ClassPathApplicationContext zu verwenden würde man an dieser Stelle den AnnotationConfigApplicationContext verwenden.

@Bean

Wenn man verschiedene Felder innerhalb dieser Klasse spezifiziert kann mit der Annotation @Bean festgelegt werden, dass es sich dabei um eine Bean handelt.

@Component

- **Zweck**
 - Generische Bezeichnung für Spring-gemanagtes Objekt
- **Spezialisierungen**
 - `@Controller`
 - `@Service`
 - ...

```
@Component("helloWorldPrinter")
public class HelloWorldPrinter {
```

Zweck

Bei Verwendung der Annotation Component kann man eine Klasse auch automatisch als Bean deklarieren lassen.

Spezialisierungen

Prinzipiell kann man Component für jede Klasse verwenden, allerdings gibt es auch Spezialisierungsfälle von Component, beispielsweise der Controller.

Diese wird oft im Zusammenhang mit Web-Anwendungen verwendet, die den Prinzipien von Model-View-Controller entsprechen. Des Weiteren gibt es die Annotation-Service die auch von Component abstammt. Diese wird bei Webservices verwendet. Eine weitere Spezialisierung ist das Repository. Dieses wird z. B. für Datenbanken verwendet. Es gibt noch viele weitere Sonderfälle, wo statt Component eine Spezialisierung verwendet werden kann. Prinzipiell ist aber Component immer gültig.

@Autowired

- **Zweck**
 - Spezifiziert DI-Abhängigkeit zu einer Bean
- **Gilt für**
 - Felder
 - Setter-Methoden
 - Methoden
 - Konstruktoren

```
@Autowired  
HelloWorldService helloWorldService;
```

Zweck

Mithilfe der Annotation @Autowired können automatisch Abhängigkeiten in die Klassen injiziert werden.

Gilt für

Autowired lässt sich anwenden für Felder als auch für Setter-Methoden als auch für normale Methoden, die bestimmte Argumente beinhalten.

@Qualifier für Autowired

- **Zweck**
 - Bestimmung von Kriterien, um eine Bean auszuwählen
 - Fehlende Eindeutigkeit im DI-Prozess
- **Eigene Qualifier**
 - Eigene Kriterien zum Abruf von Beans bestimmen

```
@Autowired  
@Qualifier("student1")  
private Student student;
```

```
<!-- Definition for student1 bean -->  
<bean id="student1" class="edu.spring.Student" />
```

Variante XML

```
@Bean(name = " student1 ")  
public Student student1(){  
    return new Student();  
}
```

Variante @Bean

Zweck

In manchen Fällen möchte man aber auch bestimmte Beans injizieren. Für genau diese Fälle würde man den Qualifier verwenden. Dieser schränkt die Injektion durch Autowired anhand eines Namenskriteriums standardmäßig ein.

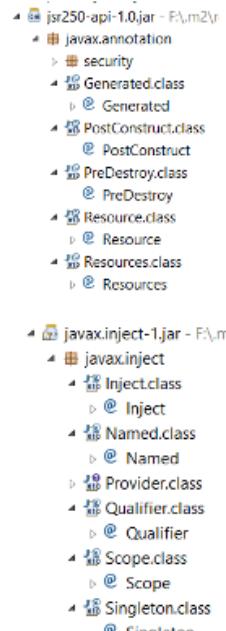
Eigene Qualifier

An der Stelle ist es auch möglich, einen eigenen Qualifier zu definieren, der nach bestimmten eigenen Kriterien und Gesichtspunkten den Autowired-Befehl einschränken kann. Dafür würde man dann ein eigenes Kriterium anhand eines Aspektes einführen.

JSR 250 und JSR 330 Kompatibilität

JSR 250

- **@PreDestroy und @PostConstruct**
 - PostConstruct kurz nach der Erzeugung einer Bean
 - PreDestroy: Aufruf kurz vor dem Zerstören einer Bean
- **@Resource**
 - Injiziert eine Bean beim Namen (ähnlich Autowired & Qualifier)



JSR 330

- **@Inject**
 - Injiziert eine Bean anhand des Typs
 - Ähnlich wie Autowired

BSc I Modul ASE1
Zürcher Fachhochschule

LE 01 – Spring Framework | © 2020, InIT

38

JSR 250

Neben vielen der eigen-implementierten Annotationen unterstützt Spring auch einige des Java-Standards. Beispielsweise der Java Standard Recommendation 250 sieht die Annotation PreDestroy und PostConstruct vor. Diese deklarieren jeweils Methoden, die einerseits wie PostConstruct kurz nach der Erzeugung einer Bean aufgerufen werden oder wie PreDestroy, nachdem die Bean zerstört wird, also der Applikationskontext beendet.

Mit Hilfe von Resource lässt sich auch eine Bean anhand des Namens injizieren. Dies entspricht in etwa den Prinzipien von Autowired und Qualifier kombiniert.

JSR 330

Des Weiteren unterstützt Spring auch die Java Standard Recommendation 330, die die Inject-Annotation vorsieht. Diese funktioniert zu 100 Prozent wie Autowired indem sie eine Bean eines bestimmten Typs direkt injiziert. Aufgrund der Kompatibilität mit den Java Standard Recommendations, kann das Spring Framework auch alternativ zur Java Enterprise Edition in bestimmten Fällen eingesetzt werden.

Vergleich Spring Annotationen mit javax

Spring	javax.inject.*	javax.inject restrictions / comments
@Autowired	@Inject	@Inject has no 'required' attribute; can be used with Java 8's Optional instead.
@Component	@Named	JSR-330 does not provide a composable model, just a way to identify named components.
@Scope("singleton")	@Singleton	The JSR-330 default scope is like Spring's prototype. However, in order to keep it consistent with Spring's general defaults, a JSR-330 bean declared in the Spring container is a singleton by default.
@Qualifier	@Qualifier / @Named	javax.inject.Qualifier is just a meta-annotation for building custom qualifiers. Concrete String qualifiers (like Spring's @Qualifier with a value) can be associated through javax.inject.Named.
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent
ObjectFactory	Provider	javax.inject.Provider is a direct alternative to Spring's ObjectFactory, just with a shorter get() method name. It can also be used in combination with Spring's @Autowired or with non-annotated constructors and setter methods.

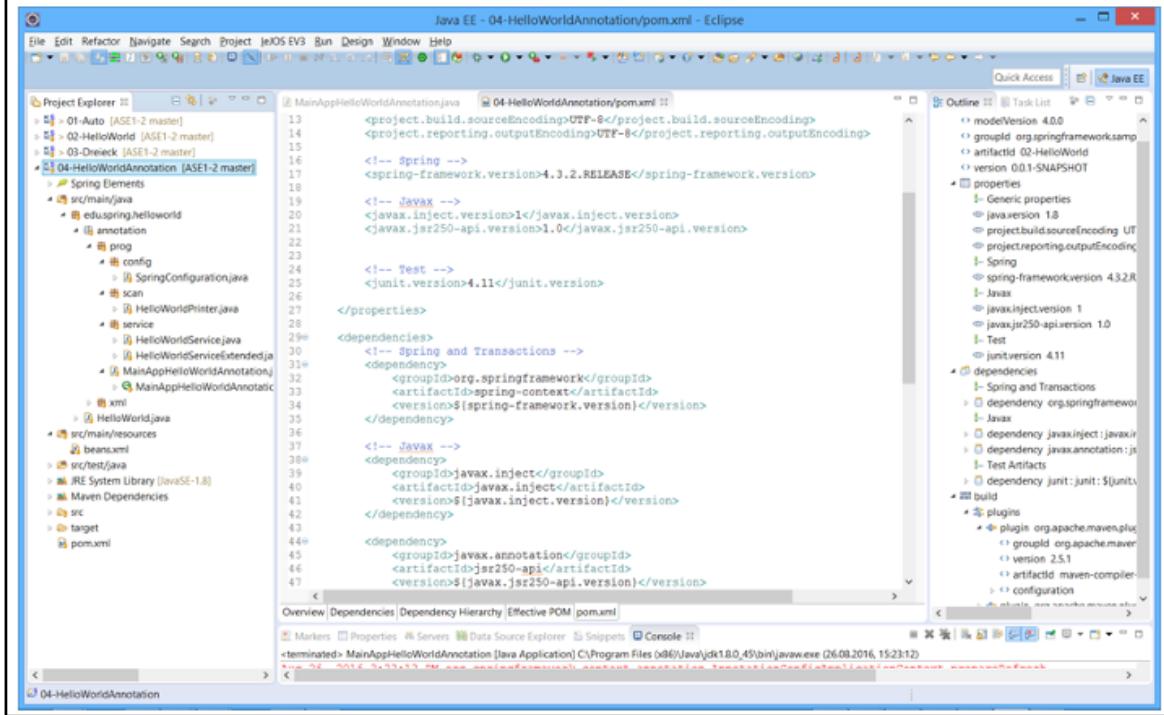
Die Tabelle zeigt den Vergleich zwischen den Spring und Javax Annotationen. In der Praxis findet man öfters in Spring Anwendungen aus JSR250 und 330 folgende Annotationen:

- @Inject
- @PostConstruct
- @PreDestroy
- @Singleton
- @Resource

Achtung: beim Import darauf achten, dass die gewünschte Klasse ausgewählt wird

Beispiel (DI ohne XML)

U: 24-HelloWorldAnnotation annotation.prog



Diese ganzen Konstrukte werden anhand eines praktischen Beispiels veranschaulicht.

pom.xml

Zunächst einmal ist es wichtig, dass die richtigen Abhängigkeiten definiert sind. Das sind einerseits die Abhängigkeit zum klassischen Spring Framework und der Dependency Injection im Spring Kontext und die jeweiligen Abhängigkeiten zur Java Standard Recommendation.

```

<!-- Spring and Transactions -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring-framework.version}</version>
</dependency>

<!-- JavaX -->
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>${javax.inject.version}</version>
</dependency>

<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>jsr250-api</artifactId>
    <version>${javax.jsr250-api.version}</version>
</dependency>

```

Ausführen

U: 24-HelloWorldAnnotation annotation.prog

Zürcher Hochschule
für Angewandte Wissenschaften

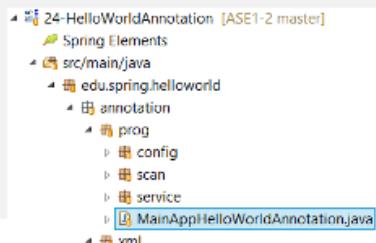


```
INFORMATION: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
Hello World Printer initialized..
Hello World!
Hello World Extended!
Aug 26, 2016 3:23:13 PM
org.springframework.context.annotation.AnnotationConfigApplicationContext doClose
INFORMATION: Closing
org.springframework.context.annotation.AnnotationConfigApplicationContext@18fd1ac:
startup date [Fri Aug 26 15:23:13 CEST 2016]; root of context hierarchy
Shutting down Hello World Printer
```

Wenn man diese Anwendungen ausführt, sieht man, dass zunächst ein HelloWorldPrinter initialisiert wird (Hello World Printer initialized). Es gibt eine Nachricht (Hello World!), es gibt eine weitere Nachricht (Hello World Extended!) und dann wird die Anwendung heruntergefahren (Shutting down Hello World Printer).

- SpringConfiguration wird explizit mittels **AnnotationConfigApplicationContext** geladen
- Mehrere Configs laden mit context.register und context.refresh

```
public static void main(String[] args) {  
  
    AnnotationConfigApplicationContext context = new  
        AnnotationConfigApplicationContext(SpringConfiguration.class);  
    HelloWorldPrinter helloWorldPrinter = (HelloWorldPrinter)  
        context.getBean("helloWorldPrinter");  
  
    helloWorldPrinter.print();  
  
    context.registerShutdownHook();  
    context.close();  
}
```



Main-Klasse – main()

Das Ganze spielt sich in der Main-Klasse ab, wo zunächst der Annotations-Applikationskontext anhand der SpringConfiguration.class initialisiert wird. Danach wird der HelloWorldPrinter abgeholt mithilfe von getBean und dementsprechend die Print-Methode ausgelöst. Als nächstes wird dann der Kontext beendet und dementsprechend auch alles heruntergefahren.

Beachtet werden sollte auch noch, dass in der Hauptklasse statt dem normalen ApplicationContext-Interface der sogenannte AbstractApplicationContext verwendet wird, denn ApplicationContext ist für Web-Anwendungen gedacht die automatisch beendet werden können.

Nur der AbstractApplicationContext ermöglicht es einen ShutdownHook zu setzen. Sie haben die Möglichkeiten gesehen auch zukünftig ohne XML-Konfiguration und nur mit dem Code arbeiten zu können.

- **HelloWorldPrinter** liegt im Package xxxx.scan und wird somit automatisch gefunden
- **HelloWorldService** und **HelloWorldServiceExtended** werden explizit initialisiert

```
@Configuration
@ComponentScan(basePackages = "edu.spring.helloworld.annotation.prog.scan")
public class SpringConfiguration {

    @Bean(name = "helloWorldService")
    public HelloWorldService helloWorldService(){
        return new HelloWorldService();
    }

    @Bean(name = "helloWorldServiceExtended")
    public HelloWorldServiceExtended helloWorldServiceExtended() {
        return new HelloWorldServiceExtended();
    }
}
```

@Configuration und @Bean

In der SpringConfiguration befindet sich die Konfiguration für die Beans, gekennzeichnet durch die Angabe von **@Configuration** wird mithilfe der Angabe von **@Bean** die jeweilige Bean initialisiert, in dem Fall **HelloWorldService** und **HelloWorldServiceExtended**.

Zusätzlich durch die Angabe von Component-Scan wird das Paket scan, das hier den **HelloWorldPrinter** beinhaltet nach Beans durchsucht. Der **HelloWorldPrinter** wird dadurch gefunden, dass die Angabe **@Component** existiert.

```
@Component("helloWorldPrinter")
public class HelloWorldPrinter {

    @Inject
    HelloWorldService helloWorldService;

    @Resource(name = "helloWorldServiceExtended")
    HelloWorldServiceExtended helloWorldServiceExtended;

    public void print() {
        System.out.println(helloWorldService.getMessage());
        System.out.println(helloWorldServiceExtended.getMessage());
    }

    @PostConstruct
    public void onInit() {
        System.out.println("Hello World Printer initialized..");
    }

    @PreDestroy
    public void onDestroy() {
        System.out.println("Shutting down Hello World Printer");
    }
}
```

Zürcher Fachhochschule

@Inject

Mithilfe von @Inject und der Angabe der jeweiligen Klasse wird automatisch der HelloWorldService in den HelloWorldPrinter hineingeladen.

@Resource

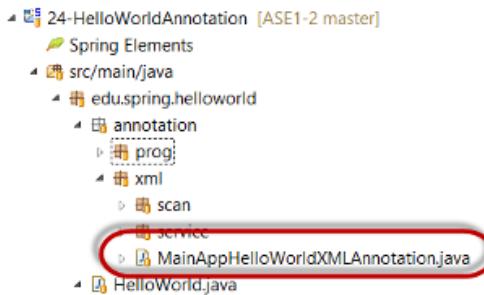
Durch die Nutzung von Resource und der Angabe des Namens wird eine ganz bestimmte Bean hineingeladen, die HelloWorldServiceExtended. Die Bezeichnung dafür entstammt dem Namen.

Der HelloWorldPrinter nutzt beide Abhängigkeiten und gibt die Nachricht aus. Zum Schluss wird, wenn er heruntergefahren wird, eine Nachricht ausgegeben. Eine Nachricht wird ebenfalls beim Hochfahren ausgegeben. Damit ist es möglich, alle Abhängigkeiten gegebenenfalls zu beenden.

Beispiel DI mit Annotation und XML

U: 24-HelloWorldAnnotation annotation.xml

- Eine Datei beans.xml wird benötigt
- Die Datei SpringConfiguration.java wird nicht benötigt
- Die Datei mit main() muss angepasst werden Klasse:
[MainAppHelloWorldXMLAnnotation](#)



Anbei noch ein Beispiel, welches die Konfiguration des Containers für die Anwendung von Annotationen mittels XML-Datei zeigt. Dazu wird eine beans.xml Datei mit den Konfigurationen benötigt. Diese muss in der Main()-Funktion mittels ClassPathXmlApplicationContext geladen werden. Dadurch erübrigert sich die Klasse SpringConfiguration.java.

Im Beispiel 24-HelloWorldAnnotation befinden sich beide Varianten: ohne und mit beans.xml Datei.

- Base-Package und Bean-Definitionen hinzufügen
- Den Namespace xmlns:context hinzufügen

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

    <context:annotation-config />
    <context:component-scan base-package="edu.spring.helloworld.annotation.xml.scan" />

    <bean id="helloWorldService"
          class="edu.spring.helloworld.annotation.xml.service.HelloWorldService" />
    <bean id="helloWorldServiceExtended"
          class="edu.spring.helloworld.annotation.xml.service.HelloWorldServiceExtended"
    />
</beans>
```

Für die Konfiguration mittels XML Datei werden zunächst die Einträge `context:annotation-config` und `context:component-scan` benötigt. Auch die Beans müssen deklariert werden. Im obenstehenden Beispiel sind das HelloWorldService und HelloWorldServiceExtended.

Main mit Annotationen und XML

U: 24-HelloWorldAnnotation annotation.xml

- Beans.xml wird mittel **ClassPathXmlApplicationContext** geladen
- Der Rest ist identisch

```
public static void main(String[] args) {  
    AbstractApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");  
    HelloWorldPrinter helloWorldPrinter = (HelloWorldPrinter)  
        context.getBean("helloWorldPrinter");  
  
    helloWorldPrinter.print();  
  
    context.registerShutdownHook();  
    context.close();  
    context.close();  
}
```

In der Main Funktion muss die beans.xml Datei geladen werden. Der Rest ist identisch wie im Beispiel mit der Spring-Konfigurationsklasse.

Zusammenfassung DI

- Was ist eine **Bean**
- Injektion einer **Inner-Class**
- **Bean Scope** (singleton, prototype, request, session, global-session)
- Injektion von **Collections** (list, map, set, props)
- Injektion einer **Vererbung**
- Injektion der **Aware Interfaces** und BeanPostProcessor
- **LifeCycleEvents** (via Interfaces oder XML)
- **Annotationen** (programmatisch oder mittels XML-Datei)
 - Spring Annotationen (@Bean, @Component, @Controller, @Service, @Autowired, @Qualifier, @Configuration)
 - JSR250 (@PreDestroy, @PostConstruct, @Resource)
 - JSR330 (@Inject) Achtung: @Qualifier von javax und spring

Wir haben das Thema Dependency Injection (Verschaltung) von Spring im Detail angeschaut und die Theorie mit Hilfe von Beispielen und Übungen gefestigt.

Der Container ist der Kern des Spring-Frameworks. Spring bietet mehrere Implementierungen des Containers, die sich aber alle jeweils einer von zwei Kategorien zuordnen lassen. Eine Bean-Factory ist die einfachste Form des Containers. Sie bietet einfache Dienste für DI und Bean-Verschaltung. Werden fortgeschrittene Framework-Dienste benötigt, ist der Einsatz des Containers Application-Context zu bevorzugen. Wir haben in diesem Kapitel gesehen, wie man Beans innerhalb des Spring-Containers miteinander verschaltet. Das Verschalten kann im Spring-Container unter Verwendung einer XML-Datei erfolgen. Diese XML-Datei enthält Konfigurationsangaben für alle Komponenten einer Anwendung sowie Informationen, mit deren Hilfe der Container die DI durchführen kann, um Beans mit anderen für sie notwendigen Beans zu verknüpfen. Mittels Annotation kann der Aufwand der XML-Konfiguration in einer Spring-Applikation reduziert werden.