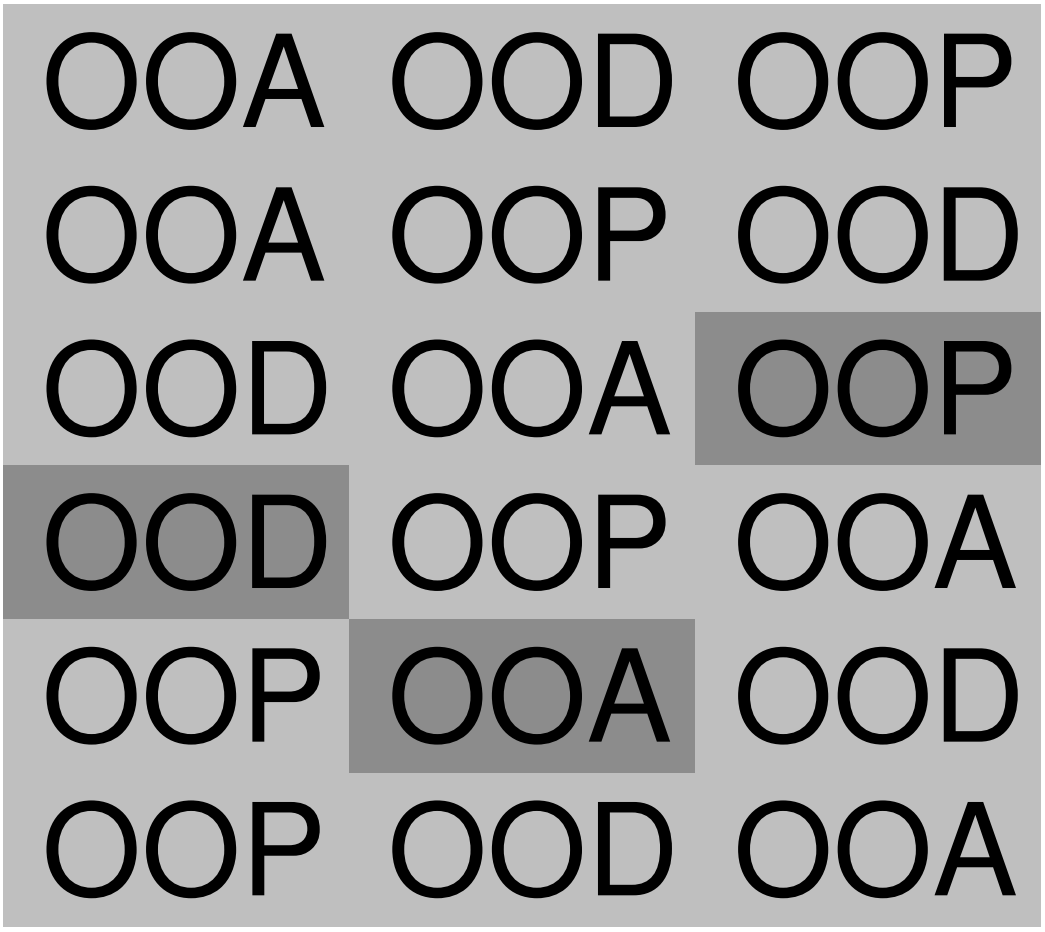
Java Springframework
Skript



Johannes Nowak

Johannes Nowak

johannes.nowak@t-online.de

Juni 2006
September 2007
November 2009
Juni 2011
Mai 2012
September 2015
August 2018
Oktober 2019

Inhalt

1 Einleitung

Spring ist in erster Linie ein Dependency-Injection-Container (ein DI-Container). Ein solcher Container wird auch als IOC-Container (Inversion of control) bezeichnet. Was ist ein DI- resp. IOC-Container? Wofür ist er verantwortlich?

Services und Beans

Ein nicht-triviales Programmsystem besteht aus vielen Komponenten (Objekten), die in irgendeiner Art und Weise kooperieren. Ein Objekt, das auf ein anderes angewiesen ist, sollte nicht direkt von der Klasse dieses anderen Objekts abhängig sein – Objekte, die Dienste für andere Objekte anbieten, sollten dies über Interfaces tun. Dann sind die Dienst-Objekt austauschbar; sie können gemockt werden; und schließlich können ihnen Proxy-Objekte vorgelagert werden.

Wir gehen also davon aus, dass alle Dienstklassen über Interfaces spezifiziert sind – und die Objekte sich nur über eben diese Interfaces aufeinander beziehen (diese Voraussetzung ist technisch gesehen allerdings nicht(!) erforderlich).

Hinweis: Was hier für "Dienstklassen" gesagt wird, gilt nicht für "dumme" Bean-Klassen: Objekte solcher Klassen dienen i.d.R. nur als Wertträger. Dienste benutzen solche Objekte, um sie als Parameter an Methoden anderer Objekte zu übergeben resp. um sie als Resultat zurückzuliefern. Im Gegensatz also zu Service-Klassen werden solche Bean-Klassen daher auch nicht über Interfaces spezifiziert.

Zusammenbau eines Systems

Um ein System in Betrieb zu nehmen, müssen eine Vielzahl von Dienst-Objekten erzeugt werden und miteinander – über Interfaces – verbunden werden. Um die Objekte zu verbinden (zu "verdrahten"), können Konstruktoren oder setter-Methoden verwendet werden.

Wichtig ist hier zu sehen, dass ein Objekt niemals ein anderes Objekt, von welchem es abhängig ist, selbst erzeugen darf – denn dass wäre es an die Implementierungsklasse dieses erzeugten Objekts verbunden. Einem Objekt, welches also von einem anderen abhängig ist, muss dieses andere Objekt also "von außen" übergeben werden – "eingespritzt" werden. Genau das meint Dependency Injection.

Wir können die an einem System beteiligten Objekte in einer eigenen "Assembler"-oder "Konfiguration"-Klasse selbst erzeugen und miteinander verdrahten – oder wir können diese Aufgabe en einen Automaten abgeben: an einen DI-Container.

DI (IOC)-Container

Ein DI-Container muss natürlich wissen, was zu tun ist: welche Objekte erzeugt werden müssen und wie diese Objekte miteinander verbunden werden müssen. Diese Informationen ("Metadaten") können wir einem Container deklarativ zur Verfügung stellen – etwa in Form einer XML-Datei oder (seit Java 5) in Form von Annotationen.

Die Java-Welt kennt eine Vielzahl solcher DI-Container. Das Herzstück eines EJB-Applikationsserver z.B. ist ein DI-Container. Ein Alternative zu EJB ist die Verwendung des CDI-Frameworks (ebenfalls im Kern ein DI-Container). Google benutzt einen eigenen DI-Container: Juice.

SE und EE

Das Spring-Framework kann in ebenso gut in einem SE-Kontext als auch im Kontext einer EE-Anwendung genutzt werden. Im Kontext einer EE-Anwendung können z.B. die Objekte, die eine mit JSF implementierte Web-Anwendung benötigt, von Spring bereitgestellt werden. Dabei ist Spring u.a. für den Lebenszyklus solcher Objekte verantwortlich: existieren sie während des gesamten Programmlaufs? Existieren sie nur für die Dauer einer http-Session? Oder existieren sie nur für die Dauer einer konkreten Requestverarbeitung? Aber Spring kann eben auch innerhalb einer einfachen Standalone-Anwendung (etwas einer Swing- oder JavaFX-Anwendung) eingesetzt werden.

Wir untersuchen im folgenden zuerst die Verwendung von Spring im SE-Kontext (dies ist der umfangreichste Part). Am Ende wenden wir und dem EE-Kontext zu.

Proxies und Interceptoren

Und Spring kann mehr, als nur Dienst-Objekte zu erzeugen und zu verbinden. Spring kann automatisch sog. Interzeptoren bereitstellen. Zu diesem Zwecke werden Proxies genutzt. Interzeptoren können fachübergreifende Aspekte implementieren (Logging, Performance-Überwachung, Transaktions-Steuerung etc.) – und solche Aspekte somit von der Fachlogik zu trennen.

Exporter-Klassen

Zudem enthält Spring eine Reihe von Hilfsklassen, welche die Benutzung der gewöhnlichen Java-APIs teilweise extrem vereinfachen. Spring unterstützt somit z.B. die JDBC und JPA-Entwicklung – insbesondere die Transaktionssteuerung. Weiterhin unterstützt Spring die Entwicklung von Anwendungen, die RMI, http, Web Services, JMS oder MBeans nutzen.

Events

Spring kann schließlich auch genutzt werden, um Events zu feuern und zu verarbeiten. Publisher können Events produzieren und feuern; Subscriber können sich für Events spezifischen Typs interessieren uns sich also für die Entgegennahme solcher Events registrieren. Eine Kommunikation der Komponenten via Events führt dazu, dass diese Komponenten nur sehr lose gekoppelt sind – und lose Kopplung ist bekanntlich ein wichtiges Architektur-Prinzip.

Technischer Hinweis:

Die Projekte des Workspace sind nach exakt demselben Schema nummeriert wie die Kapitel und die Abschnitte dieses Dokuments. Das Projekt `x0306-XmlCtx-Lifecycle` z.B. ist beschrieben im Kapitel 3, Abschnitt 6. Neben der Nummer des Kapitels und des Abschnitts enthält der Projektname dann den (i.d.R. verkürzten) Kapitelnamen und den Abschnittsnamen.

Servlet-Implantierung und MVC

Spring unterstützt nicht zuletzt die Entwicklung von Web-Anwendungen…

2 Services und Aspekte

Im folgenden demonstrieren wir die Spring-Features häufig anhand eines Systems von drei Services. Diese Services sind durch folgende Interfaces spezifiziert:

```
package ifaces;

public interface SumService {
    public abstract int sum(int x, int y);
}
```

```
package ifaces;

public interface DiffService {
    public abstract int diff(int x, int y);
}
```

```
package ifaces;

public interface MathService {
    public abstract int sum(int x, int y);
    public abstract int diff(int x, int y);
}
```

Mittels eines `SumService`s kann die Summe zweier Zahlen berechnet werden; mittels des `DiffService`s die Differenz zweier Zahlen; und mittels eines `MathService`s können sowohl die Summe als auch die Differenz berechnet werden.

In diesem Kapitel werden zwei der wichtigsten Features vorgesellt, die von Spring adressiert werden (aber ohne hier Spring bereits zu nutzten):

- Im ersten Abschnitt dieses Kapitels demonstrieren wir "manuelle" Dependency-Injection.
- Im zweiten Abschnitt zeigen wir, was es mit "Interception" auf sich hat.

Um Spring begreifen zu können, ist das Verständnis dieser beiden Mechanismen zumindest äußerst hilfreich...

2.1 Manuelle Erzeugung und Verdrahtung

Wie implementieren die o.g. Interfaces in einem Package namens `beans`.

Hinweis: Der Name ist missverständlich – implementieren wir doch gerade keine "dummen" "Bean"-Klassen, sondern "intelligente" Services. Da der Name `bean` aber auch von Spring benutzt wird, um Services zu bezeichnen, halten wir uns an diese Spring-Konvention…)

beans.DiffServiceImpl

```
package beans;

import ifaces.DiffService;
import jn.util.Log;

public class DiffServiceImpl implements DiffService {
    public DiffServiceImpl() {
        Log.log();
    }
    @Override
    public int diff(int x, int y) {
        return x - y;
    }
}
```

beans.SumServiceImpl

```
package beans;

import ifaces.SumService;
import jn.util.Log;

public class SumServiceImpl implements SumService {
    public SumServiceImpl() {
        Log.log();
    }
    @Override
    public int sum(int x, int y) {
        return x + y;
    }
}
```

beans.MathServiceImpl

```
package beans;

import ifaces.DiffService;
import ifaces.MathService;
import ifaces.SumService;
```

```
import jn.util.Log;

public class MathServiceImpl implements MathService {

    private final SumService sumService;
    private final DiffService diffService;

    public MathServiceImpl(SumService sumService, DiffService diffService) {
        Log.log(sumService, diffService);
        this.sumService = sumService;
        this.diffService = diffService;
    }

    @Override
    public int sum(int x, int y) {
        return this.sumService.sum(x, y);
    }

    @Override
    public int diff(int x, int y) {
        return this.diffService.diff(x, y);
    }
}
```

`SumServiceImpl` und `DiffServicIImpl` haben neben der `sum`- resp. der `diff`-Methode einen parameterlosen Konstruktor, dessen Aufgabe nur darin besteht, seinen Aufruf zu loggen (mittels der `Log.log`-Methode, die im `shared`-Paket hinterlegt ist).

`MathServiceImpl` hat zwei Attribute vom Typ `SumService` resp. `DiffService`. Diese Attribute werden über den Konstruktor initialisiert. Die `sum`- resp. die `diff`-Methoden von `MathServiceImpl` können also an die entsprechenden Methoden der "Hilfsmathematiker" delegieren.

Man beachte, dass `MathServiceImpl` die benötigten Hilfsmathematiker nicht selbst erzeugt – sie müssen also dem Konstruktor übergeben werden. Wir haben dem Chefmathematiker die beiden Hilfsmathematiker "untergejubelt" – injiziert. Würde die Klasse `MathServiceImpl` diese Hilfsmathematiker selbst erzeugen, müssten die Namen der Implementierungs-Klassen bekannt sein – `MathServiceImpl` wäre also von `SumServiceImpl` und `DiffServiceImpl` abhängig.

Genau solche Abhängigkeiten sollen aber vermieden werden. Aus vielerlei Gründen: die Implementierungen sollten austauschbar sein; man möchte Proxy-Objekte bauen, die den eigentlichen Implementierungen vorangestellt werden; man möchte auf einfache Art Mock-Objekte erzeugen können etc.

Wie definieren nun eine Klasse `Configuration`, innerhalb derer die Service-Objekte erzeugt und miteinander "verdrahtet" werden. Die einzige Methode dieser Klassen liefert eine Referenz des Interfaces-Typs `MathService` zurück:

appl.Configuration

```
package appl;
// ...
public class Configuration {

    private static MathService mathService;

    public static synchronized MathService getMathService() {
        if (mathService != null)
            return mathService;
        final SumService sumService = new SumServiceImpl();
        final DiffService diffService = new DiffServiceImpl();
        mathService = new MathServiceImpl(sumService, diffService);
        return mathService;
    }
}
```

Wie haben uns offensichtlich für Lazy-Creation entschieden (Erzeugung auf Verlangen). Man beachte, das die `getMathService`-Methode immer dasselbe `MathServiceImpl`-Objekt zurückliefert. (Service-Klassen sind konzeptionell i.d.R. Singleton-Klassen).

Hier die `Applicaiton`-Klasse, welche die Benutzung der Services demonstrieren soll:

appl.Application

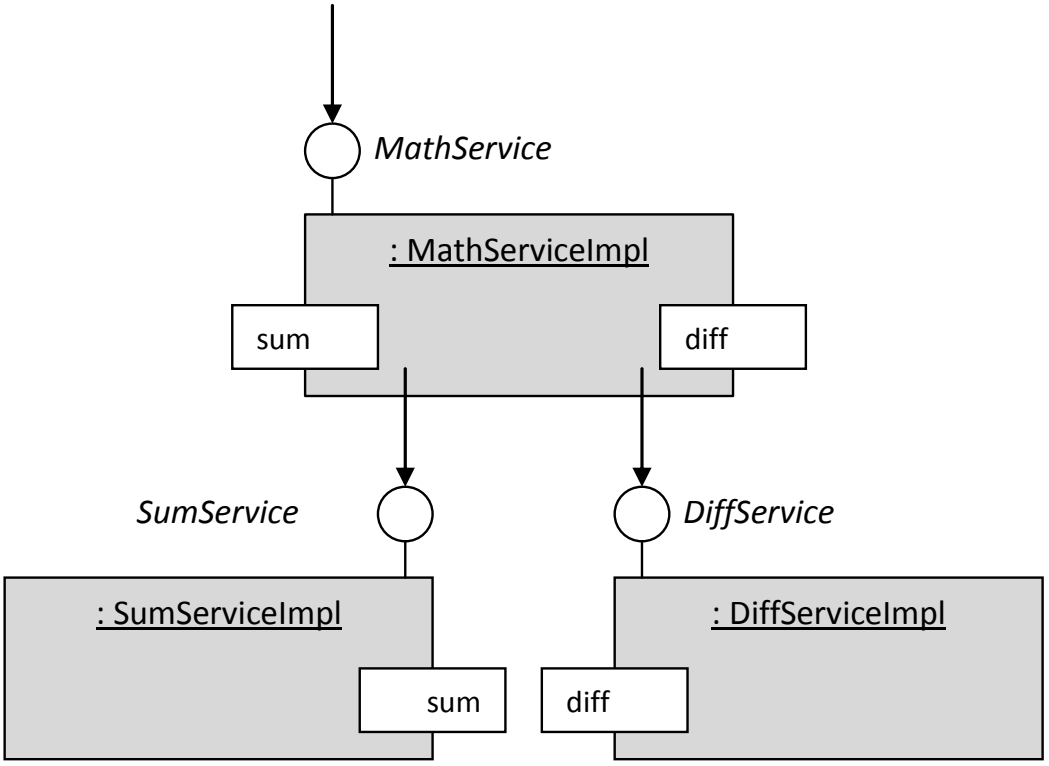
```
package appl;

import ifaces.MathService;

public class Application {
    public static void main(String[] args) {
        final MathService mathService = Configuration.getMathService();
        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
    }
}
```

Die Anwendung operiert nur mit einem Interface…

Die Ausgaben:

```
beans.SumServiceImpl.<init>()
beans.DiffServiceImpl.<init>()
beans.MathServiceImpl.<init>(
    beans.SumServiceImpl@2b193f2d, beans.DiffServiceImpl@355da254)
42
77
```

Hier das Resultat der Konfiguration in Form eines Objektdiagramms:



2.2 Manuelle Interception

Angenommen, wir wollen die Aufrufe aller Methoden des im letzten Abschnitt aufgebauten Systems tracen – und zwar sowohl den Einstieg als auch den Ausstieg in resp. aus diesen Methoden. Wir könnten die Methoden durch entsprechende Trace-Anweisungen erweitern – das würde aber den Code der Methoden "verschmutzen". Die Methoden würden dann sowohl die eigentliche Fachlichkeit implementieren als auch einen fachfremden "Aspekt": den Trace-Aspekt.

Wie kann die Implementierung solcher Aspekte (Tracen, Transaktionen steuern, Ausführungszeiten messen, Berechtigungen prüfen etc.) von der eigentlichen Fachlichkeit getrennt werden?

Wir benutzen eine einfache `Trace`-Klasse, die im `shared`-Paket implementiert ist, als Hilfsklasse:

jn.util.Trace

```
package jn.util;

import java.lang.reflect.Method;

// the class is only suitable for single-threaded systems...

public class Trace {

    private static int indent = 0;

    private static void printIndent() {
        for(int i = 0; i < indent; i++)
            System.out.print("\t");
    }

    public static void before(Method method, Object[] parameters) {
        printIndent();
        System.out.print(method.getDeclaringClass().getSimpleName());
        System.out.print(".");
        System.out.print(method.getName());
        System.out.print("(");
        if (parameters != null) {
            for (int i = 0; i < parameters.length; i++) {
                if (i > 0)
                    System.out.print(", ");
                System.out.print(parameters[i]);
            }
        }
        System.out.println(") {");
        indent++;
    }
```

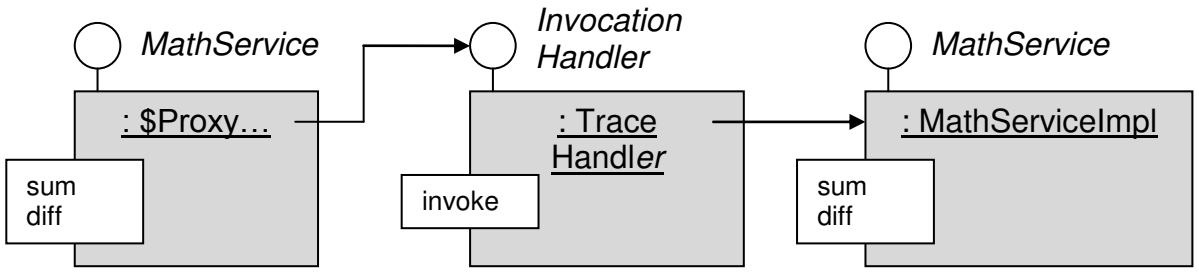
```
    public static void after(Method method, Object result) {
        indent--;
        printIndent();
        System.out.print("}");
        if (method.getReturnType() != void.class)
            System.out.print(" -> " + result);
        System.out.println();
    }

    public static void after(Method method, Throwable throwable) {
        indent--;
        System.out.println("} => " + throwable);
    }
}
```

Die Klasse enthält drei statische öffentliche Methoden: `before` und zwei überladene `after`-Methoden. before kann genutzt werden, um den Einstieg in eine Methode zu tracen; die erste `after`-Methoden kann genutzt werden, um den normalen Ausstieg zu tracen; die zweite `after`-Methoden kann genutzt werden, wenn ein exzeptioneller Ausstieg protokolliert werden soll.

Allen drei Methoden wird ein `Method`-Objekt übergeben – es handelt sich dabei um dasjenige `Method`-Objekt, welche die zu tracende Methode beschreibt. Alle drei Methoden geben den durch den Klassennamen präfixierten Namen der Methode aus – an die `before`-Methode wird zusätzlich die Liste der Aufrufparameter übergeben; den beiden `after`-Methoden zusätzlich der Returnwert resp. die Exception.

Ein `before`-Aufruf, der einem anderen `before`-Aufruf unmittelbar folgt, wird eingerückt dargestellt; die Ausgaben der entsprechenden `after`-Aufrufe werden "ausgerückt".

Wir benutzen im folgenden den Java-eigenen Dynamic-Proxy-Mechanismus. Das Resultat der folgenden Überlegungen sei zunächst einmal in Form eines Diagramms vorgestellt:



Der Dynamic-Proxy-Mechanismus verlangt die Implementierung des Interfaces `java.lang.reflect.InvocationHandler`. Die `invoke`-Methode eines solchen Handlers wird dann über ein Proxy-Objekt aufgerufen werden, dessen Klasse von Dynamic-Proxy automatisch zur Laufzeit generiert werden wird. Die Klasse dieses

Proxy-Objekts implementiert dabei dasselbe Interface wie die Klasse des eigentlichen Zielobjekts (im obigen Diagramm also das Interface `MathService`).

An das Proxy-Objekt kann jedes Objekt angeschlossen werden, dessen Klasse das Interface `InvocationHandler` implementiert (`TraceHandler`, `TransactionHandler` etc.); das Handler-Objekt hat eine Referenz auf das eigentliche Ziel-Objekt: eine Referenz vom allgemeinen Typ `Object`. Ein `TraceHandler` (ein `TransactionHandler`…) kann also einem Service-Objekt beliebigen Typs vorangestellt werden (er ist also allgemein nutzbar).

Hier die allgemein verwendbare Klasse `TraceHandler` (ebenfalls im `shared`-Projekt enthalten):

jn.util.TraceHandler

```
package jn.util;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TraceHandler implements InvocationHandler {

    private final Object target;

    public TraceHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        if (method.getDeclaringClass() != Object.class)
            Trace.before(method, args);
        try {
            final Object result;
            if (this.target instanceof InvocationHandler)
                result = ((InvocationHandler) this.target).invoke(
                    proxy, method, args);
            else
                result = method.invoke(this.target, args);
            if (method.getDeclaringClass() != Object.class)
                Trace.after(method, result);
            return result;
        }
        catch (InvocationTargetException e) {
            final Throwable te = e.getTargetException();
            if (method.getDeclaringClass() != Object.class)
                Trace.after(method, te);
            throw te;
        }
        catch (Throwable e) {
```

```
            if (method.getDeclaringClass() != Object.class)
                Trace.after(method, e);
            throw e;
        }
    }
}
```

Ein `TraceHandler` hat eine Referenz auf das Zielobjekt (`Object target`), die über den Konstruktor initialisiert wird. Die `invoke`-Methode des Handlers wird vom Proxy-Objekt aufgerufen werden – immer dann, wenn auf dieses Objekt eine der fachlichen Methoden (`sum`, `diff`) aufgerufen werden wird. Jeder Aufruf einer Proxy-Methode wird also an die `invoke`-Methode des Handlers delegiert.

Jede Proxy-Methode ist auf die gleiche Art und Weise implementiert: sie berechnet das `Method`-Objekt der aufgerufen Methode (genauer: der Interface-Methode) und trägt die individuellen Parameter, die an die Methode übergeben werden, in einem `Object`-Array ein. Dann wird die `invoke`-Methode des Handlers eben mit diesen beiden Parametern (`Method` und `Object`-Array) aufgerufen.

Die `invoke`-Methode des `TraceHandler` wird im folgenden nur grob beschrieben (es sei dem Leser / der Leserin überlassen, den Quellcode etwas genauer unter die Lupe zu nehmen).

Die `invoke`-Methode des Handlers implementiert einen "Dreisatz": Pre-Invoke, Invoke und Post-Invoke. Im Pre-Invoke-Schritt wird der Einstieg in die Methode getract (mittels `Trace.before`). Dann wird (im Invoke-Schritt) die `Method`-eigene `invoke`-Methode benutzt, um auf das Zielobjekt (`target`) genau diejenige Methode aufzurufen, die von dem `Method`-Objekt beschrieben wird. Anschließend wird im letzten Schritt (Post-Invoke) der Ausstieg aus der Methode getract (mittels `Trace.after`).

Alle Aufrufe auf das Zielobjekt geschehen also über die `invoke`-Methode des Handlers. Dieser kann daher als "Interceptor" bezeichnet werden. Diese `invoke`-Methode implementiert die Funktionalität des entsprechenden "Aspekts".

Nun muss nurmehr geklärt werden, wie die Proxy-Klasse zur Laufzeit generiert und diese generierte Klasse dann instanziiert wird. Hierzu wird die Java-eigene Methode `Proxy.newProxyInstance` genutzt. Dieser werden zwei Parameter übergeben: das Interface, das die zu generierende Klasse implementieren muss (damit eben dieses Klasse generiert werden kann) und der `InvocationHandler` (damit die generierte Klasse instanziiert werden kann).

Die folgende kleine Helper-Klasse `SimpleProxy` (ebenfalls im `shared`-Paket vereinbart) dient dazu, die Benutzung von `Proxy.newProxyInstance` zu vereinfachen:

jn.util.SimpleProxy

```
package jn.util;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

public class SimpleProxy {
    @SuppressWarnings("unchecked")
    public static <T> T create(Class<T> iface, InvocationHandler handler) {
        return (T) Proxy.newProxyInstance(
                Thread.currentThread().getContextClassLoader(),
                new Class<?>[] { iface },
                handler);
    }
}
```

Der `create`-Methode wird u.a. ein Interface des Typs `T` übergeben – und sie liefert ein Proxy zurück, welches ebenfalls vom Typ `T` ist (`T` implementiert).

Wir erweitern schließlich die Klasse `Configuration`. Vor jedes der drei `Impl`-Objekt wird ein Proxy mit angeschlossenem `TraceHandler` eingeschoben:

appl.Configuration

```
package appl;
// ...
public class Configuration {
    public static MathService createMathService() {
        final SumService sumService = SimpleProxy.create(
                SumService.class,
                new TraceHandler(
                    new SumServiceImpl()));
        final DiffService diffService = SimpleProxy.create(
                DiffService.class,
                new TraceHandler(
                    new DiffServiceImpl()));
        return SimpleProxy.create(
                MathService.class,
                new TraceHandler(
                    new MathServiceImpl(sumService, diffService)));
    }
}
```

Die Klasse `Application` wird unverändert aus dem letzten Abschnitt übernommen:

appl.Application

```
package appl;

import ifaces.MathService;

public class Application {
    public static void main(String[] args) {
        final MathService mathService = Configuration.createMathService();
        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
    }
}
```

Hier die Ausgaben:

```
beans.SumServiceImpl.<init>()
beans.DiffServiceImpl.<init>()
beans.MathServiceImpl.<init>(
    beans.SumServiceImpl@27c170f0, beans.DiffServiceImpl@5451c3a8)

MathService.sum(40, 2) {
    SumService.sum(40, 2) {
    } -> 42
} -> 42
42
MathService.diff(80, 3) {
    DiffService.diff(80, 3) {
    } -> 77
} -> 77
77
```

Resultat: Ohne dass die `Impl`-Klassen angepasst werden mussten und ohne dass die Client-Klasse (`Application`) geändert werden musste, konnten wir in unserem System nun Querschnitts-Funktionalität einfügen: den Trace-Aspekt. Dabei haben wir zwei wiederverwendbare Klassen verwenden: `TraceHandler` und `SimpleProxy`. Nur die Klasse `Configuration` musste angepasst werden…

3 Der ClassPathXmlApplicationContext

Spring muss konfiguriert werden. In der Konfiguration werden die zu erzeugenden Objekte und die Verdrahtung dieser Objekte und ggf. weitere Initialisierungen hinterlegt.

Mittlerweile existieren drei unterschiedliche Arten der Konfiguration:

- In den ersten Spring-Versionen wurden zum Zecke der Konfiguration XML-Dateien genutzt.
- Ab Java 5 können die zu instituierenden Klassen und deren Methoden durch Annotationen ausgezeichnet werden – mit dem Resultat, dass in vielen Fällen auf die XML-Dateien komplett verzichtet werden kann.
- Schließlich existiert noch eine weitere Variante, in welcher die Objekte direkt mit Java-Mitteln erzeugt und verdrahtet werden (dies geschieht in Klassen, die mit `@Configuration` gekennzeichnet sind und Methoden enthalten, die ihrerseits mit `@Bean` annotiert sind).

Alle Varianten haben Vor- und Nachteile.

Wenngleich mittlerweile die XML-Variante an Bedeutung verliert, wird sie im folgenden dennoch ausführlich vorgestellt – u.a. deshalb, weil viele "alte" Spring-basierte Systeme eben diese Variante verwenden. Und solche Systeme müssen natürlich nach wie vor verstanden und gepflegt werden können.

In diesem Kapitel geht's deshalb zunächst einmal darum, die XML-Konfiguration vorzustellen.

- Im ersten Abschnitt wird das Konzept der dynamischen Erzeugung der Objekte vorgestellt.
- Im zweiten Abschnitt wird die sog. Constructor-Injection vorgestellt.
- Im dritten Abschnitt geht's um die sog. Property-Injection.
- Im vierten Abschnitt geht's um das sog. Inlining
- Im fünften und sechsten Abschnitt wird demonstriert, wie Spring Objekte auch dann miteinander verknüpfen kann, wenn wie die Art und Weise dieser Verknüpfung überhaupt nicht explizit angeben (es geht um das `"autowired"`-Konzept).

- Im siebten Abschnitt geht's um sog. Lebenszyklus-Methoden: wie kann Spring z.B. den Objekten mitteilen, dass sie "betriebsbereit" sind?

- Im achten Abschnitt wird gezeigt, wie Objekten nicht nur Referenzen auf andere Objekte zugewiesen werden können, sondern auch Werte einfachen Typs (`String`s, `int`-Werte etc.)

- Im neunten Abschnitt zeigen wir, wie Beans von anderen Beans "abgeleitet" werden (über eine "`parent`"-Beziehung).

- In den Abschnitten 10 und 11 wird gezeigt, wie die Konfiguration auf mehrere Dateien verteilt werden kann.

- In den Abschnitten 12, 13 und 14 wird gezeigt, wie Referenzen auf einfache Objekte, Listen und Maps in Beans injiziert werden können.

- Die Abschnitte 15 und 16 geben einen kleinen Einblick in Spring-eigene Expression-Language.

- Abschnitt 17 zeigt, wie wir mittels Spring auf Property-Dateien zugreifen können.

- Im Abschnitt 18 zeigen wird, dass von Spring verwaltete Objekte auch bidirektional miteinander verknüpft werden können.

- Abschnitt 19 demonstriert die Scopes "`singleton`" und "`prototype`".

- Abschnitt 20 demonstriert Probleme der Mehrdeutigkeit.

- Abschnitt 21 zeigt den Unterschied zwischen dem eager- und dem lazy-Modus der Erzeugung von Objekten.

- Abschnitt 22 zeigt, dass `Closeable`s von Spring ordnungsgemäß geschlossen werden.

- Und Abschnitt 23 demonstriert eine einfache "realistische" Anwendung – eine Anwendung mit einem Konto-Service, einem Transfer-Service und einem Konto-DAO.

3.1 ApplicationContext

Wir wollen nun unsere `Configuration`-Klasse, in der wir im letzten Kapitel die Service-Objekte erzeugt und miteinander verdrahtet hatten, ersetzen durch einen Spring-eigenen Mechanismus.

Wir implementieren zunächst aber nur die Interfaces `SumService` und `DiffService` – der `MathService` wird erst in nächsten Abschnitt eingeführt.

Spring wird nun also die Service-Objekte erzeugen müssen – dies geschieht natürlich via Reflection (mittels `Class.getConstructor(...).newInstance(...)`). Da die Klassen der beiden Hilfsmathematiker (`SumServiceImpl` und `DiffServiceImpl`) einen parameterlosen Konstruktor haben, ist die Instanziierung dieser Klassen kein Problem.

Wir definieren folgende XML-Konfiguration:

spring.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-4.0.xsd">

    <bean id="sumService" class="beans.SumServiceImpl"/>
    <bean id="diffService" class="beans.DiffServiceImpl"/>

</beans>
```

Das Wurzelelement einer Spring-Konfiguration hat den Namen `beans`. Dieses Element kann u.a. beliebig viele `bean`-Elemente enthalten. Ein `bean`-Element hat die Attribute `id` und `class`. Der Wert des `class`-Attributs ist der voll-qualifizierte Name der zu instanziierenden Klasse.

Spring wird aufgrund dieser Konfiguration eine Registry aufbauen (einen sog. `ApplicationContext`), in dem die erzeugten Objekte unter der jeweiligen `id` registriert sind.

Hinweis zur Terminologie

Wir werden im folgenden davon sprechen, dass eine `SumServiceImpl`-Bean (oder noch einfacher: eine `SumService`-Bean) in der XML-Datei "registriert" wird.

Damit meinen wir dann folgendes:

Aufgrund des Eintrags in einer XML-Datei wird zur Laufzeit eine Objekt der angegebenen Implementierungs-Klasse erzeugt und initialisiert werden und unter dem angegebenen Namen in die Registratur des Spring-Kontexts eingetragen werden.

Da dieser Satz recht lang ist, werden wir die obige Kurzform verwenden - auch wenn diese missverständlich resp. unpräzise ist.

Die folgende Klasse zeigt, wie auf diese Spring-Konfiguration zugegriffen werden kann:

appl.Application

```
package appl;

import org.springframework.context.support.
                          ClassPathXmlApplicationContext;

import ifaces.DiffService;
import ifaces.SumService;

public class Application {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final SumService sumService1 =
                ctx.getBean(SumService.class);
            final SumService sumService2 =
                ctx.getBean(SumService.class);

            System.out.println(sumService1 == sumService2);

            System.out.println(sumService1.sum(40, 2));
            System.out.println(sumService2.sum(80, 3));

            final DiffService diffService =
                ctx.getBean(DiffService.class);
            System.out.println(diffService.diff(80, 3));
        }
    }
}
```

Die Ausgaben:

```
beans.SumServiceImpl.<init>()
beans.DiffServiceImpl.<init>()
true
42
83
77
```

Zunächst wird ein `ClassPathXmlApplicationContext` erzeugt (dieses ist `AutoCloseable`) – dabei wird der Name der Konfigurationsdatei übergeben (die ihrerseits über den `CLASSPATH` auffindbar sein muss). Der Konstruktor baut aufgrund der XML-Datei die interne Registry auf.

Mittels der Methode `getBean` können nun Referenzen auf die im `ApplicationContext` enthaltenen Objekte ermittelt werden. Diese Methode ist mehrfach überladen. Im obigen Beispiel wird diejenige Variante genutzt, welcher eine `Class`-Referenz übergeben wir. Wir übergeben die `Class`-Objekte der Interfaces `SumService` resp. `DiffService`.

Spring wird nun nachschauen, ob zu diesen Interfaces in der zuvor aufgebauten Registry Objekte gibt, deren Klassen jeweils das an `getBean` übergebene Interface implementieren. Für das Interface `SumService` wird ein `SumServiceImpl` gefunden; für `DiffService` wird ein `DiffServiceImpl` gefunden. Die Referenz auf diese Objekte wird dann jeweils von `getBean` zurückgeliefert.

Im obigen Beispiel wird `getBean` zweimal mit demselben `Class`-Objekt aufgerufen (`SumService.class`). Die Ausgaben zeigen, dass beide Aufruf die Referenz auf ein- und dasselbe Objekt zurückliefern.

Alternativ hätten wir auch eine überladene `getBean`-Methode aufrufen können, der eine ID übergeben wird – eine ID, unter der das entsprechende Objekte in der Registry eingetragen ist:

```
            final MathService mathService =
                (MathService) ctx.getBean("mathService");
```

Das funktioniert natürlich nur mit einem Downcast…

3.2 Constructor Injection

Neben den Klassen `SumServiceImpl` und `DiffServiceImpl` soll nun auch die `MathServiceImpl` von Spring instanziiert und die Instanz in der Spring-Registry eingetragen werden.

Im Gegensatz zu `SumServiceImpl` und `DiffServiceImpl` besitzt `MathServiceImpl` nun aber einen parametrisierten Konstruktor:

```
    public MathServiceImpl(SumService sumService, DiffService diffService) {
        this.sumService = sumService;
        this.diffService = diffService;
    }
```

Spring wird also bei der Erzeugung des `MethServiceImpl`-Objekts die Referenzen auf die bereits zuvor erzeugten Hilfsmathematiker übergeben müssen. Dies wird als Constructor-Injection bezeichnet.

Wir definieren folgende XML-Konfiguration:

spring.xml

```
<beans ...>

    <bean id="sumService" class="beans.SumServiceImpl"/>

    <bean id="diffService" class="beans.DiffServiceImpl"/>

    <bean id="mathService" class="beans.MathServiceImpl" >
        <constructor-arg ref="sumService"/>
        <constructor-arg ref="diffService"/>
    </bean>

    <alias alias="math" name="mathService"/>

</beans>
```

Das Wurzelelement einer Spring-Konfiguration hat den Namen `beans`. Dieses Element kann u.a. beliebig viele `bean`-Elemente enthalten.

Der dritte `bean`-Eintrag hat zwei `constructor-arg`-Sub-Elemente. Beide Einträge beziehen sich in ihrem `ref`-Attribut auf die `id`s der beiden anderen `bean`-Einträge. Wenn Spring also ein `MathServiceImpl` erzeugen wird, wird Spring dem Konstruktor die Referenzen auf die beiden bereits zuvor erzeugten Hilfsmathematiker übergeben.

Die folgende Klasse zeigt, wie auf diese Spring-Konfiguration zugegriffen werden kann:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            final MathService mathService1 =
                ctx.getBean(MathService.class);
            final MathService mathService2 =
                ctx.getBean(MathService.class);

            System.out.println(mathService1 == mathService2);

            System.out.println(mathService1.sum(40, 2));
            System.out.println(mathService2.diff(80, 3));
        }
    }
}
```

Die Ausgaben:

```
beans.SumServiceImpl.<init>()
beans.DiffServiceImpl.<init>()
beans.MathServiceImpl.<init>(
    beans.SumServiceImpl@50b494a6,
    beans.DiffServiceImpl@3cef309d)
true
42
77
```

Auch hier liefert der wiederholte Aufruf von `getBean(MathService.class)` stets dasselbe `MathServiceImpl`-Objekt zurück.

3.3 Property Injection

Die Dependency-Injection passierte im letzten Abschnitt über den Konstruktor (das hat den Vorteil, dass die somit initialisierten Referenzen `sumServie` und `diffService` als `final` definiert werden konnten).

Dependency-Injection kann aber auch via Properties erfolgen.

Wir erinnern uns: "Properties" sind get/set-Paare. Die folgenden beidem Methoden z.B. definieren eine Property namens `"schallUndRauch"` vom Typ `int`:

```
public void setSchallUndRauch(int x) { ... }
public int getSchallUndRauch() { ... }
```

Die folgende `MathServiceImpl`-Klasse enthält nurmehr einen parameterlosen Konstruktor – und zusätzlich aber zwei setter-Methoden: `setSumService` und `setDiffService`:

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {

    private SumService sumService;
    private DiffService diffService;

    public MathServiceImpl() {
        Log.log();
    }

    public void setSumService(SumService sumService) {
        Log.log(sumService);
        this.sumService = sumService;
    }

    public void setDiffService(DiffService diffService) {
        Log.log(diffService);
        this.diffService = diffService;
    }

    @Override
    public int sum(int x, int y) {
        return this.sumService.sum(x, y);
    }

    @Override
    public int diff(int x, int y) {
        return this.diffService.diff(x, y);
    }
}
```

Man beachte, dass die setter-Methoden unbedingt `public` sein müssen.

In der `spring.xml` müssen dann die `constructor-arg`-Einträge durch `property`-Einträge ersetzt werden (wobei als `name`-Attribut eben exakt der Name der Property angegeben werden muss):

spring.xml

```
<beans ...>

    <bean id="sumService" class="beans.SumServiceImpl"/>

    <bean id="diffService" class="beans.DiffServiceImpl"/>

    <bean id="mathService" class="beans.MathServiceImpl" >
        <property name="sumService" ref="sumService"/>
        <property name="diffService" ref="diffService"/>
    </bean>

    <alias alias="math" name="mathService"/>

</beans>
```

Die `Application`-Klasse ist unverändert aus dem letzten Abschnitt übernommen worden.

Die Ausgaben zeigen, dass Spring nun die `setSumService`- und `setDiffService`-Methoden aufruft:

Die Ausgaben:

```
beans.SumServiceImpl.<init>()
beans.DiffServiceImpl.<init>()
beans.MathServiceImpl.<init>()
beans.MathServiceImpl.setSumService(beans.SumServiceImpl@1ed4004b)
beans.MathServiceImpl.setDiffService(beans.DiffServiceImpl@ff5b51f)
true
42
77
```

3.4 Inlining

Bislang hätten wir in der Demo-Applikation auch direkt den `SumService` und den `DiffService` ermitteln können. Wir können die XML-Einträge zu diesen Services aber auch "verstecken":

```
<beans ...>

    <bean id="mathService" class="beans.MathServiceImpl" >

        <property name="sumService">
            <bean class="beans.SumServiceImpl"/>
        </property>

        <property name="diffService">
            <bean class="beans.DiffServiceImpl"/>
        </property>

    </bean>

</beans>
```

Statt die `property`-Elemente mit `ref`-Attributne zu versehen, welche auf andere bean-Einträge verweisen, kann auch ein inneres `bean`-Element verwendet werden – und zwar eines, welches nur ein `class`-Attribut hat.

Die beiden Hilfsmathematiker sind nun nicht nur "namenlos" – und sind auch nicht mehr direkt über die Registry ansprechbar. Wir können Objekte also auch "privatisieren" - die Anwendung kann nun nurmehr auf den `MathService` zugreifen.

3.5 Autowired By Type

In den obigen Beispielen haben wir ausdrücklich angegeben, was dem Konstruktor übergeben werden muss resp. welche Properties gesetzt werden müssen. Wir haben die erforderliche Verdrahtung also explizit beschrieben.

Spring unterstützt aber auch das Konzept des "automatischen Verdrahtens".

Angenommen, wir modifizieren unsere `MathServiceImpl`-Klasse – statt Properties implementieren wird zwei Methoden namens `setHello` und `setWorld` (der ersten Methode wird ein `SumSerice`, der zweiten ein `DiffService` übergeben):

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {

    private SumService sumService;
    private DiffService diffService;

    public void setHello(SumService sumService) {
        this.sumService = sumService;
    }

    public void setWorld(DiffService diffService) {
        this.diffService = diffService;
    }

    @Override
    public int sum(int x, int y) {
        return this.sumService.sum(x, y);
    }

    @Override
    public int diff(int x, int y) {
        return this.diffService.diff(x, y);
    }
}
```

Hier die neue `spring.xml`:

spring.xml

```
<beans ...>

    <bean class="beans.SumServiceImpl"/>
    <bean class="beans.DiffServiceImpl"/>
    <bean id="mathService" class="beans.MathServiceImpl" autowire="byType"/>

</beans>
```

Wir erweitern die Definition des `"mathService"`-Eintrags um das Attribut `autowire` mit dem Wert `"byType"`.

Spring wird dann die `MathSericeImpl`-Klasse daraufhin untersuchen, ob sie Methoden bereitstellt, denen eine andere registrierte Bean übergeben werden kann – eine andere Bean, deren Klasse kompatibel ist zu den Parametertypen dieser Methoden. Da Spring auf diese Weise die Methoden `setHello` und `setWorld` findet, werden eben diese aufgerufen.

3.6 Autowired By Name

Statt die automatische Verdrahtung über die Typen der Beans und die Typen der Methodenparameter zu bewerkstelligen, können auch die Namen herangezogen werden.

Wir verwenden dieselbe `MathServiceImpl`-Klasse wie im letzten Beispiel (mit den Methoden `setHello` und `setWorld`).

Die beiden Hilfsmathematiker registrieren wird unter den Namen "`hello`" und "`word`" – und ersetzten den `autowire`-Wert "`byType`" durch den Wert "`byName`":

spring.xml

```
<beans ...>

    <bean id="hello" class="beans.SumServiceImpl"/>
    <bean id="world" class="beans.DiffServiceImpl"/>
    <bean id="mathService" class="beans.MathServiceImpl" autowire="byName"/>

</beans>
```

Ob diese Variante wirklich empfehlenswert ist?

3.7 Lifecycle

Spring kann die vom Framework verwalteten Objekte über Ereignisse in ihrem Lebenszyklus unterrichten.

Wir erweitern alle `Impl`-Klassen um zwei Methoden: `postConstruct` und `preDestroy`. Beide Methoden sind parameterlos und `void`:

beans.SumServiceImpl

```
package beans;
// ...
public class SumServiceImpl implements SumService {
    public void postConstruct() {
        Log.log();
    }
    public void preDestroy() {
        Log.log();
    }
    // ...
}
```

beans.DiffServiceImpl

```
package beans;
// ...
public class DiffServiceImpl implements DiffService {
    // dito...
}
```

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {
    // dito...
}
```

In der `spring.xml` erweitern wir alle `bean`-Einträge um zwei Attribute: `init-method` und `destroy-method`. Die Werte beider Attribute sind die Namen der in der Implementierung genutzten Methoden (`"postConstruct"` und `"preDestroy"`):

spring.xml

```
<beans ...>

    <bean id="sumService" class="beans.SumServiceImpl"
            init-method="postConstruct"
            destroy-method="preDestroy">
    </bean>

    <bean id="diffService" class="beans.DiffServiceImpl"
            init-method="postConstruct"
            destroy-method="preDestroy">
    </bean>


    <bean id="mathService" class="beans.MathServiceImpl"
            init-method="postConstruct"
            destroy-method="preDestroy">
        <property name="sumService" ref="sumService"/>
        <property name="diffService" ref="diffService"/>
    </bean>

</beans>
```

Die `Application` besorgt sich wieder den Chefmathematiker und ruft auf diesen jeweils einmal die `sum`- und die `diff`-Methode auf.

Die Ausgaben:

```
beans.SumServiceImpl.<init>()
beans.SumServiceImpl.postConstruct()
beans.DiffServiceImpl.<init>()
beans.DiffServiceImpl.postConstruct()
beans.MathServiceImpl.<init>()
beans.MathServiceImpl.setSumService(beans.SumServiceImpl@1ed4004b)
beans.MathServiceImpl.setDiffService(beans.DiffServiceImpl@ff5b51f)
beans.MathServiceImpl.postConstruct()
42
77
beans.MathServiceImpl.preDestroy()
beans.DiffServiceImpl.preDestroy()
beans.SumServiceImpl.preDestroy()
```

Die `postContruct`-Methode wird erst dann aufgerufen, wenn alle Dependencies gesetzt wurden. Wenn diese Methode also aufgerufen wird, können wir sicher sein, dass die Bean "betriebsbereit" ist.

`preDestroy` wird dann aufgerufen, wenn der `ApplicationContext` geschlossen wird.

Die Namen dieser Methoden sind natürlich frei wählbar.

3.8 Value Properties

Bislang haben wir nur Referenzen in Objekte injiziert. Wir können aber auch einfache Werte injizieren.

Wir erweitern die `MathServiceImpl`-Klasse um ein `max`-Attribut. Sofern die `sum`- oder die `diff`-Methode mit Parameterwert aufgerufen werden, die größer als der `max`-Wert sind, werfen sie eine Exception.

Der `max`-Wert kann über die Property-Methode `setMax` eingestellt werden.

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {

    private SumService sumService;
    private DiffService diffService;
    private int max = Integer.MAX_VALUE;

    public void setSumService(SumService sumService) {
        Log.log(sumService);
        this.sumService = sumService;
    }

    public void setDiffService(DiffService diffService) {
        this.diffService = diffService;
    }

    public void setMax(int max) {
        this.max = max;
    }

    @Override
    public int sum(int x, int y) {
        if (x > this.max || y > this.max)
            throw new RuntimeException("too large numbers");
        return this.sumService.sum(x, y);
    }

    @Override
    public int diff(int x, int y) {
        if (x > this.max || y > this.max)
            throw new RuntimeException("too large numbers");
        return this.diffService.diff(x, y);
    }
}
```

spring.xml

```
<beans ...>

    <bean id="sumService" class="beans.SumServiceImpl" />
    <bean id="diffService" class="beans.DiffServiceImpl" />

    <bean id="mathService" class="beans.MathServiceImpl" >
        <property name="sumService" ref="sumService" />
        <property name="diffService" ref="diffService" />
        <property name="max" value="100"/>
    </bean>

</beans>
```

Die `"mathService"`-Bean ist um einen dritten `property`-Eintrag erweitert worden. Dieser bezieht sich aber nicht via `ref` auf eine andere Bean, sondern definiert das Attribut `value` – mit der Wert `"100"`.

Sollen also Werte einfachen Typs übergeben werden, so wird `ref` durch `value` ersetzt.

3.9 Parent Bean

Die `MathServiceImpl`-Klasse wird unverändert aus dem letzten Abschnitt übernommen.

Wir wollen zwei Beans vom Typ `MathServiceImpl` registrieren, die aber mit unterschiedlichen `max`-Werten ausgestattet sein sollen.

Das, was jeder der beiden Mathematiker benötigt, wird in einer "abstrakten" Bean deklariert - die unter "`mathService`" registriert wird.

Diese Bean kann nun als "Muster-Bean" fungieren.

Jeder der beiden "konkreten" Mathematiker wird dann von dieser Muster-Bean "abgeleitet" – und unter "`mathServiceA`" resp. "`mathServiceB`" registriert:

spring.xml

```
<beans ...>

    <bean id="sumService" class="beans.SumServiceImpl" />
    <bean id="diffService" class="beans.DiffServiceImpl" />

    <bean id="mathService" class="beans.MathServiceImpl">
        <property name="sumService" ref="sumService" />
        <property name="diffService" ref="diffService" />
    </bean>


    <bean id="mathServiceA" parent="mathService">
        <property name="max" value="100" />
    </bean>

    <bean id="mathServiceB" parent="mathService">
        <property name="max" value="10" />
    </bean>

</beans>
```

Per `parent`-Attribut wird hier festgelegt, dass diese Bean zunächst einmal genauso behandelt werden will wie die "Muster-Bean". Zusätzlich können dann weitere Dependency-Injections vorgenommen werden (Setzen der "`max`"-Property).

Der Lookup einer Anwendung muss sich stets auf "konkrete" Beans beziehen - ein Lookup mit "`mathService`" würde also fehlschlagen.

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext("spring.xml")) {

            final MathService mathService =
                ctx.getBean("mathService", MathService.class);

            final MathService mathServiceA =
                ctx.getBean("mathServiceA", MathService.class);

            // ...
        }
    }
}
```

3.10 Multiple Configuration Files

Die Spring-Konfiguration kann auf mehrere XML-Dateien verteilt werden.

Wir benutzen im folgenden nur die `SumService`- und die `DiffService`-Implementierung:

beans.SumServiceImpl

```
package beans;
// ...
public class SumServiceImpl implements SumService {
    // ...
}
```

beans.DiffServiceImpl

```
package beans;
// ...
public class DiffServiceImpl implements SumService {
    // ...
}
```

Aufgrund der ersten Konfigurations-Datei wird der `SumService` registriert:

spring1.xml

```
<beans ...>

    <bean id="sumService" class="beans.SumServiceImpl"
        destroy-method="preDestroy" />
</beans>
```

Aufgrund der zweiten Konfigurations-Datei dann der `DiffService`:

spring2.xml

```
<beans ...>

    <bean id="diffService" class="beans.DiffServiceImpl"
        destroy-method="preDestroy" />
</beans>
```

Dem Konstruktor von `ClassPathXmlApplicationContext` werden dann beide Konfigurations-Dateien übergeben (via varargs):

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {

        try (final ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext(
                "spring1.xml", "spring2.xml")) {

            final SumService sumService = ctx.getBean(SumService.class);
            System.out.println(sumService.sum(40, 2));

            final DiffService diffService = ctx.getBean(DiffService.class);
            System.out.println(diffService.diff(80, 3));
        }
    }
}
```

3.11 Parent Context

Wir verwenden wieder dieselben `ServiceImpl`-Klasse wie im letzten Abschnitt.

Auch die beiden Konfigurationsdateien (`spring1.xml`, `spring2.xml`) werden unverändert übernommen.

Statt eines einzigen Kontext-Objekts erzeugen wir nun aber zwei – wobei die erste Konfiguration von einem Vater-Kontext, die zweite Konfiguration von einem Kind-Kontext repräsentiert wird:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {

        try (final ClassPathXmlApplicationContext parent =
                new ClassPathXmlApplicationContext("spring1.xml")) {
            try (final ClassPathXmlApplicationContext ctx =
                    new ClassPathXmlApplicationContext()) {
                ctx.setParent(parent);
                ctx.setConfigLocations("spring2.xml");
                ctx.refresh();

                final SumService sumService =
                    ctx.getBean(SumService.class);
                System.out.println(sumService.sum(40, 2));
                final DiffService diffService =
                    ctx.getBean(DiffService.class);
                System.out.println(diffService.diff(80, 3));
            }
        }
    }
}
```

Wir benötigen beide Resource-Try-Blöcke, damit die Resourcen ordnungsgemäß freigegeben werden (auf beide Beans die `preDestroy`-Methode aufgerufen wird).

3.12 Simple Argument Types

Die Werte, die in einer Spring-Konfiguration hinterlegt werden (entweder als Werte von `value`-Attributen oder als innerer Text von `<value>`-Elementen), sind allesamt einfache `Strings`. Um solche `Strings` den Properties von Beans zuzuweisen, müssen sie natürlich entsprechend der Typen der Properties konvertiert werden. Hierzu benutzt Spring das `PropertyEditor`-Konzept des `java.beans`-Packages.

Das folgende Beispiel wird zeigen, dass z.B. nach `int` und `double` konvertiert werden kann - aber auch nach `Integer` und `Double`.

Bei `String`-, `Integer`- und `Double`-Argumenten stellt sich dann natürlich auch die Frage, wie ein `null`-Wert übergeben werden kann. Hierzu stellt Spring das Element `<null>` zur Verfügung (das überall dort verwendet werden kann, wo auch das `<value>`-Element benutzt werden kann).

Gegeben sei folgendes Interface:

ifaces.Foo

```
package ifaces;

import java.awt.Point;
import java.util.Date;

public interface Foo {
      public abstract String getS();
      public abstract int getI();
      public abstract double getD();
      public abstract Integer getIw();
      public abstract Double getDw();
      public abstract Date getDate();
      public abstract Point getPoint();
}
```

Und folgende Implementierung:

beans.FooImpl

```
package beans;
// ...
public class FooImpl implements Foo {

    private String s;
    private int i;
    private double d;
    private Integer iw;
    private Double dw = 0.0;
    private Date date;
    private Point point;

    // setter und getter ...
}
```

Da Foo eine `Point`-Property besitzt, für diese Klasse aber noch kein Standard-Editor existiert, benötigen wir eine eigenen Editor-Klasse:

util.PointEditor

```
package util;

import java.awt.Point;
import java.beans.PropertyEditorSupport;

import jn.util.Log;

public class PointEditor extends PropertyEditorSupport {
    private Point point;
    @Override
    public Object getValue() {
        Log.log();
        return this.point;
    }
    @Override
    public void setValue(Object value) {
        Log.log(value);
        this.point = (Point)value;
    }
    @Override
    public void setAsText(String text) {
        Log.log(text);
        try {
            String[] tokens = text.split(":");
            this.point = new Point(
                Integer.parseInt(tokens[1]),
                Integer.parseInt(tokens[0]));
        }
        catch(Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

spring.xml

```
<beans ...>

    <bean class="org.springframework.beans.factory
                 .config.CustomEditorConfigurer">
      <property name="customEditors">
        <map>
          <entry key="java.awt.Point" value="util.PointEditor"/>
        </map>
      </property>
    </bean>

    <bean id="foo1" class="beans.FooImpl">
        <property name="s" value="Hello"/>
        <property name="i" value="4711"/>
        <property name="d" value="3.14"/>
        <property name="iw" value="222"/>
        <property name="dw" value="2.71"/>
        <property name="date" value="2011/11/20"/>
        <property name="point" value="100:200"/>
    </bean>

    <bean id="foo2" class="beans.FooImpl">
        <property name="s" value="World"/>
        <property name="i" value="4711"/>
        <property name="d" value="3.14"/>
        <property name="iw" value="333"/>
        <property name="dw">
            <null/>
        </property>
        <property name="date">
            <null/>
        </property>
        <property name="point">
            <null/>
        </property>
    </bean>

</beans>
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            System.out.println();
            final Foo foo1 = ctx.getBean("foo1", Foo.class);
            print(foo1);

            System.out.println();
            final Foo foo2 = ctx.getBean("foo2", Foo.class);
            print(foo2);
        }
    }

    private static void print(Foo foo) {
        System.out.println(foo.getS());
        System.out.println(foo.getI());
        System.out.println(foo.getD());
        System.out.println(foo.getIw());
        System.out.println(foo.getDw());
        System.out.println(foo.getDate());
        System.out.println(foo.getPoint());
    }
}
```

Die Ausgaben:

```
util.PointEditor.setValue(null)
util.PointEditor.setAsText(100:200)
util.PointEditor.getValue()
util.PointEditor.setValue(null)
util.PointEditor.getValue()

Hello
4711
3.14
222
2.71
Sun Nov 20 00:00:00 CET 2011
java.awt.Point[x=200,y=100]

World
4711
3.14
333
null
null
null
```

3.13 List Types

Im folgenden zeigen wir, wie Listen, Sets und Arrays via Dependency Injection injiziert werden können.

Wir benutzen eine einfache Bean-Klasse (ohne Interface):

beans.Language

```
package beans;

public class Language {

    public final String name;
    public final String designer;

    public Language(String name, String designer) {
        this.name = name;
        this.designer = designer;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName() +
            " [" + name + ", " + designer + "]";
    }
}
```

Und wir benutzen einen Sprach-Zoo, der vier Properties besitzt (die über Spring initialisiert werden sollen):

beans.Zoo

```
package beans;
/7 ...
public class Zoo {

    private List<String> list;
    private Set<String> set;
    private String[] array;
    private List<Language> languageList;

    // setter, getter...

    public void print() {
        for (String s : this.list)
            System.out.println(s);
        System.out.println();
        for (String s : this.set)
            System.out.println(s);
        System.out.println();
```

```
        for (String s : this.array)
            System.out.println(s);
        System.out.println();
        for (Language l : this.languageList) {
            System.out.println(l);
        }
    }
}
```

Das `property`-Element der Spring-Konfiguration (oder auch das `constructor-arg`-Element) kann ein Subelement `list` (oder auch `set`) enthalten. Und dieses Element enthält beliebig viele `value`-Elemente (oder `ref`- resp. `bean`-Elemente):

spring.xml

```
<beans ...>

    <bean id="zoo" class="beans.Zoo">
        <property name="list">
            <list>
                <value>Pascal</value>
                <value>Modula</value>
                <value>Oberon</value>
            </list>
        </property>

        <property name="set">
            <list>
                <value>Pascal</value>
                <value>Modula</value>
                <value>Oberon</value>
            </list>
        </property>

        <property name="array">
            <list>
                <value>Pascal</value>
                <value>Modula</value>
                <value>Oberon</value>
            </list>
        </property>

        <property name="languageList">
            <list>
                <bean class="beans.Language">
                    <constructor-arg value="Pascal" />
                    <constructor-arg value="Wirth" />
                </bean>
                <bean class="beans.Language">
                    <constructor-arg value="Modula" />
                    <constructor-arg value="Wirth" />
                </bean>
                <bean class="beans.Language">
                    <constructor-arg value="Oberon" />
```

```
                    <constructor-arg value="Wirth" />
                </bean>
            </list>
        </property>
    </bean>
</beans>
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final Zoo zoo = ctx.getBean(Zoo.class);
            zoo.print();
        }
    }
}
```

Die Ausgaben:

```
Pascal
Modula
Oberon

Pascal
Modula
Oberon

Pascal
Modula
Oberon

Language [Pascal, Wirth]
Language [Modula, Wirth]
Language [Oberon, Wirth]
```

3.14 Map Types

Im folgenden zeigen wir, wie Map-Properties beschrieben werden können.

Wir verwenden wieder dieselbe `Language`-Klasse wie im letzten Abschnitt.

beans.Foo

```
package beans;

import java.util.Map;
import java.util.Properties;

public class Zoo {

    private Properties props;
    private Map<String,String> stringMap;
    private Map<String,Language> languageMap;

    public void setProps(Properties props) {
        this.props = props;
    }
    public void setStringMap(Map<String,String> stringMap) {
        this.stringMap = stringMap;
    }
    public void setLanguageMap(Map<String,Language> languageMap) {
        this.languageMap = languageMap;
    }

    public void print() {
        this.props.forEach(
            (k, v) -> System.out.println(k + " --> " + v));
        System.out.println();
        this.stringMap.forEach(
            (k, v) -> System.out.println(k + " --> " + v));
        System.out.println();
        this.languageMap.forEach(
            (k, v) -> System.out.println(k + " --> " + v));
        System.out.println();
    }
}
```

Eine `Map` wird in der Spring-Konfiguration mittels eines `map`-Elements beschrieben.

Das `map`-Element kann beliebig viele `entry`-Elemente enthalten. Ein `entry`-Element beschreibt einen Eintrag der zu erzeugenden Map und enthält daher zwei Subelemente: `key` und `value`. Ein `key`-Element enthält wiederum ein Subelement `value`. (Statt eines `value`-Elements sind hier natürlich auch `ref`- und `bean`-Elemente zulässig.)

swing.xml

```
<beans ...>

    <bean id="zoo" class="beans.Zoo">

        <property name="props">
            <props>
                <prop key="Pascal">Wirth</prop>
                <prop key="Eiffel">Meyer</prop>
            </props>
        </property>

        <property name="stringMap">
            <map>
                <entry>
                    <key>
                        <value>Pascal</value>
                    </key>
                    <value>Wirth</value>
                </entry>
                <entry key="Eiffel" value="Meyer" />
            </map>
        </property>

        <property name="languageMap">
            <map>
                <entry>
                    <key>
                        <value>Pascal</value>
                    </key>
                    <bean class="beans.Language">
                        <constructor-arg value="Pascal" />
                        <constructor-arg value="Wirth" />
                    </bean>
                </entry>
                <entry>
                    <key>
                        <value>Eiffel</value>
                    </key>
                    <bean class="beans.Language">
                        <constructor-arg value="Eiffel" />
                        <constructor-arg value="Meyer" />
                    </bean>
                </entry>
            </map>
        </property>
    </bean>

</beans>
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final Zoo zoo = ctx.getBean(Zoo.class);
            zoo.print();
        }
    }
}
```

Die Ausgaben:

```
Eiffel --> Meyer
Pascal --> Wirth

Pascal --> Wirth
Eiffel --> Meyer

Pascal --> Language [Pascal, Wirth]
Eiffel --> Language [Eiffel, Meyer]
```

3.15 Expression Language – Part 1

Wie können wir innerhalb einer Spring-Konfiguration auf Properties bereits registrierter Beans zugreifen?

Wie benutzen eine sehr einfache `MathServiceImpl`-Klasse mit einer `max`-Property (auf die schreibend und lesend(!) zugegriffen werden kann;

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {

    private int max = Integer.MAX_VALUE;

    public void setMax(int max) {
        this.max = max;
    }

    public int getMax() {
        return this.max;
    }

    @Override
    public int sum(int x, int y) {
        if (x > this.max || y > this.max)
            throw new RuntimeException("too large numbers");
        return x + y;
    }

    @Override
    public int diff(int x, int y) {
        if (x > this.max || y > this.max)
            throw new RuntimeException("too large numbers");
        return x - y;
    }
}
```

Wir wollen nun zwei Beans registrieren, von denen die zweite Bean einen doppelt so großen `max`-Wert zugewiesen bekommen soll wie die erste Bean:

spring.xml

```
<beans...>

    <bean id="mathService" class="beans.MathServiceImpl" />

    <bean id="mathServiceA" parent="mathService">
        <property name="max" value="100" />
    </bean>

    <bean id="mathServiceB" parent="mathService">
        <property name="max" value="#{2 * mathServiceA.max}" />
    </bean>

</beans>
```

Der `max`-Wert für die zweite Bean wird ermittelt über den Ausdruck

```
#{2 * mathServiceA.max}
```

Es handelt sich um einen Ausdruck der Spring-eigenen Expression-Language. Sie erlaubt es u.a., auf Properties anderer Beans mittels der üblichen Punkt-Notation zuzugreifen.

3.16 Expression Language – Part 2

Die Expression-Language erlaubt auch den Zugriff auf statische Elemente einer Klasse – sowohl auf statische Attribute als auch auf statische Methoden. Und sie erlaubt auch die Erzeugung von Objekten…

Zunächst zur Benutzung statischer Elemente.

Angenommen, wir wollen einen `javax.swing.JLabel` produzieren lassen – und dabei den Text und die Hintergrundfarbe einstellen (via `setText` und `setBackground`):

Wir stellen vier verschiedene Varianten vor – nur die letzten beiden nutzen die Möglichkeiten der Expression-Language.

Die erste Variante nutzt den Konstruktor der Klasse `java.awt.Color`:

```
    <bean id="label1" class="javax.swing.JLabel">
        <property name="text" value = "Label 1"/>
        <property name="background">
            <bean class="java.awt.Color">
                <constructor-arg type="int" value="255"/>
                <constructor-arg type="int" value="0"/>
                <constructor-arg type="int" value="0"/>
            </bean>

        </property>
    </bean>
```

Die Anwendung (`ctx` sei der `AppliationContext`):

```
            final JLabel label1 = ctx.getBean("label1", JLabel.class);
            System.out.println(
                label1.getText() + " " + label1.getBackground());
```

Die Ausgabe:

```
Label 1 java.awt.Color[r=255,g=0,b=0]
```

Die zweite Variante benutzt einen `CustomEditorConfigurer`:

```
    <bean class="org.springframework.beans.
                 factory.config.CustomEditorConfigurer">
      <property name="customEditors">
        <map>
          <entry key="java.awt.Color" value="util.ColorEditor"/>
        </map>
      </property>
    </bean>
```

```
    <bean id="label2" class="javax.swing.JLabel">
        <property name="text" value = "Label 2"/>
        <property name="background" value="0,255,0" />
    </bean>
```

Die Klasse `util.ColorEditor`:

```
package util;
// ...
public class ColorEditor extends PropertyEditorSupport {
    private Color color;
    @Override
    public Object getValue() {
        return this.color;
    }
    @Override
    public void setValue(Object value) {
        this.color = (Color)value;
    }
    @Override
    public void setAsText(String text) {
        try {
            final String[] tokens = text.split(",");
            this.color = new Color(
                    Integer.parseInt(tokens[0]),
                    Integer.parseInt(tokens[1]),
                    Integer.parseInt(tokens[2]));
        }
        catch(final Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

Die Anwendung und ihre Ausgaben:

```
            final JLabel label2 = ctx.getBean("label2", JLabel.class);
            System.out.println(
                label2.getText() + " " + label2.getBackground());
```

```
Label 2 java.awt.Color[r=0,g=255,b=0]
```

Die dritte Variante nutzt nun die Möglickeit, in einer Expression auf statische Elemente einer Klasse zuzugreifen – auf das Element `blue`:

```
    <bean id="label3" class="javax.swing.JLabel">
        <property name="text" value = "Label 3"/>
        <property name="background" value="#{T(java.awt.Color).blue}" />
    </bean>
```

Die Anwendung und ihre Ausgaben:

```
            final JLabel label3 = ctx.getBean("label3", JLabel.class);
            System.out.println(
                label3.getText() + " " + label3.getBackground());
```

```
Label 3 java.awt.Color[r=0,g=0,b=255]
```

Die vierte Variante schließlich nutzt nun die Möglickeit, in einer Expression statische Methoden einer Klasse aufzurufen – die Methode `decode`:

```
    <bean id="label4" class="javax.swing.JLabel">
        <property name="text" value = "Label 4"/>
        <property name="background"
            value="#{T(java.awt.Color).decode('0xffff00')}" />
    </bean>
```

Die Anwendung und ihre Ausgaben:

```
            final JLabel label4 = ctx.getBean("label4", JLabel.class);
            System.out.println(
                label4.getText() + " " + label4.getBackground());
```

```
Label 4 java.awt.Color[r=255,g=255,b=0]
```

Die Syntax solcher Zugriffe resp. Aufrufe sieht allgemein wie folgt aus:

```
T(<Klasse>).<statisches Element>
```

Ein letztes Beispiel: einem JLabel soll ein Font injiziert werden.

```
    <bean id="label5" class="javax.swing.JLabel">
        <property name="text" value = "Label 5"/>
        <property name="font">
            <bean class="java.awt.Font">
                <constructor-arg value="#{'Ari' + 'al'}" />
                <constructor-arg value=
                    "#{T(java.awt.Font).BOLD + T(java.awt.Font).ITALIC}" />
                <constructor-arg value="#{20 + 4}" />
            </bean>
        </property>
    </bean>
```

Die Anwendung und ihre Ausgaben:

```
            final JLabel label5 = ctx.getBean("label5", JLabel.class);
            System.out.println(label5.getText() + " " + label5.getFont());
```

```
Label 5 java.awt.Font[family=Arial,name=Arial,style=bolditalic,size=24]
```

Leider muss beim OR der Font-Stile der +-Operator verwendet werden (der |-Operator wird nicht unterstützt).

In einer Expression kann auch `new` verwendet werden:

```
    <bean id="label6" class="javax.swing.JLabel">
        <property name="text" value =
            "Label 6"/>
        <property name="background" value=
            "#{new java.awt.Color(255, 0, 255)}" />
        <property name="font" value=
            "#{new java.awt.Font('Arial', T(java.awt.Font).BOLD, 24)}" />
    </bean>
```

Die Anwendung und ihre Ausgaben:

```
            final JLabel label6 = ctx.getBean("label6", JLabel.class);
            System.out.println(label6.getText() + " " +
                label6.getBackground() + " " + label6.getFont());
```

```
Label 6 java.awt.Color[r=255,g=2,b=3]
    java.awt.Font[family=Arial,name=Arial,style=bold,size=24]
```

Die allgemeine Syntax:

```
new <Klassenname> (<Parameter>)
```

Und in einer Expression können wir auch wiederum Properties auf ein neu erzeugte Objekt aufrufen. Das folgende Beispiel ist zwar "sinnlos", aber es funktioniert:

```
    <bean id="label7" class="javax.swing.JLabel">
        <property name="text" value = "Label 7"/>
        <property name="background"
            value="#{new java.awt.Color(
                new java.awt.Color(255, 0, 0).red, 0, 255)}" />
    </bean>
```

Hier wird ein neues `java.awt.Color`-Objekt erzeugt und anschließend die `getRed`-Methode auf dieses Objekt aufgerufen. Der Wert, den dieser Aufruf liefert, wird dann als erster Parameter an die Erzeugung eines neuen `Color`-Objekts übergeben…

Daraus folgt, dass mittels der Expression-Language auch kompette Objekt-Grafen erzeugt werden können:

```
    <bean id="mathService" class="beans.MathServiceImpl">
        <constructor-arg value="#{new beans.SumServiceImpl()}"/>
        <constructor-arg value="#{new beans.DiffServiceImpl()}"/>
    </bean>
```

Eine Sprache in einer Sprache….

3.17 Property Files

Spring ermöglicht den einfachen Zugriff auf Properties-Dateien.

Wie bauen eine grafische Benutzeroberfläche – wobei wichtige Eigenschaften dieser Oberfläche sind in einer Properties-Datei hinterlegt sind:

gui.properties

```
title       My Window
x           100
y           200
width       700
height      100
```

Aufgrund der Einträge dieser Datei soll über die Spring-Konfiguration ein Objekt des Typs `GuiParams` erzeugt und initialisiert werden:

util.GuiParams

```
package util;
// ...
public class GuiParams {

    public final String title;
    public final int x;
    public final int y;
    public final int width;
    public final int height;

    public GuiParams(String title,
            int x, int y, int width, int height) {
        this.title = title;
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

In der `spring.xml` wird eine `PropertyPlaceholderConfigurer` genutzt, um die Property-Datei einzulesen. Die Einträge dieser Datei können dann über ihre Schlüssel angesprochen werden und in das `GuiParams`-Objekt injiziert werden:

spring.xml

```
<beans ...>

    <bean class="....beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location">
            <value>gui.properties</value>
        </property>
    </bean>

    <bean id="guiParams" class="util.GuiParams">
        <constructor-arg value="${title}" />
        <constructor-arg value="${x}" />
        <constructor-arg value="${y}" />
        <constructor-arg value="${width}" />
        <constructor-arg value="${height}" />
    </bean>

</beans>
```

Die `Application` besorgt sich die `GuiParams` (die dann zum Aufbau einer Swing-Oberfläche genutzt werden könnten):

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final GuiParams gui = ctx.getBean(GuiParams.class);
            System.out.println(gui);
        }
    }
}
```

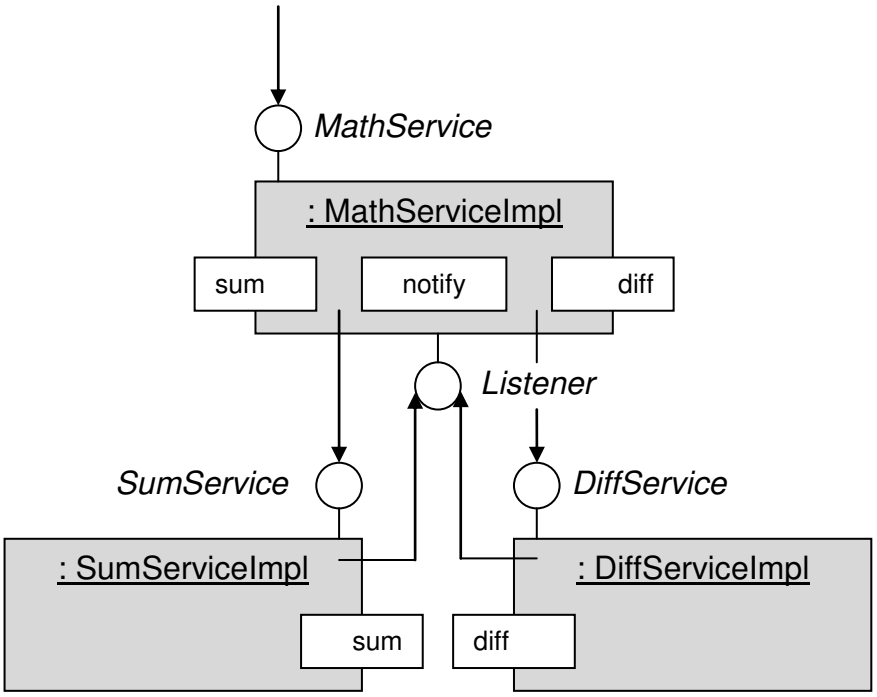
Die Ausgaben:

```
GuiParams [My Window, 100, 200, 700, 100]
```

3.18 Bidirectional Associations

Spring kann Objekte auch bidirektional miteinander verbinden.

Hier zunächst ein Objektdiagramm:



Die Klasse `MathServiceImpl` implementiert nun neben `MathService` zusätzlich ein Interface namens `Listener`. Jeder der beiden Hilfsmathematiker bekommt eine `Listener`-Referenz injiziert. Über diese Referenzen können dann die Hilfsmathematiker den Chefmathematiker über ihre Berechnungen informieren.

Das Interface `Listener` spezifiziert eine Benachrichtigungs-Methode `notify`:

ifaces.Listener

```
package ifaces;

public interface Listener {
    public abstract void notify(String msg);
}
```

Den beiden Hilfsmathematikern wird via Property-Injection jeweils ein `Listener` zugewiesen. In ihren `sum`- resp. `diff`-Methoden wird dann die `notify`-Methode auf diese `Listener` aufgerufen:

beans.SumServiceImpl

```
package beans;
// ...
public class SumServiceImpl implements SumService {
    private Listener listener;
    public void setListener(Listener listener) {
        Log.log(listener);
        this.listener = listener;
    }
    public int sum(int x, int y) {
        if (this.listener != null)
            this.listener.notify("sum(" + x + ", " + y + ")");
        return x + y;
    }
}
```

beans.DiffServiceImpl

```
package beans;
// ...
public class DiffServiceImpl implements DiffService {
    private Listener listener;
    public void setListener(Listener listener) {
        Log.log(listener);
        this.listener = listener;
    }
    public int diff(int x, int y) {
        if (this.listener != null)
            this.listener.notify("diff(" + x + ", " + y + ")");
        return x - y;
    }
}
```

Die Klasse `MathServiceImpl` implementiert nun zusätzlich das `Listener`-Interface:

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService, Listener {

    private SumService sumService;
    private DiffService diffService;

    public void setSumService(SumService sumService) {
        Log.log(sumService);
        this.sumService = sumService;
    }
    public void setDiffService(DiffService diffService) {
        Log.log(diffService);
        this.diffService = diffService;
    }
```

```
    public int sum(int x, int y) {
        return this.sumService.sum(x, y);
    }
    public int diff(int x, int y) {
        return this.diffService.diff(x, y);
    }
    public void notify(String msg) {
        System.out.println("notify: " + msg);
    }
}
```

In der XML-Datei müssen wir dann bei der Definition von `"sumService"` und `"diffService"` bereits auf den erst später definierten `"mathService"` zugreifen – diesen eben der `listener`-Property der Hilfsmathematiker zuweisen:

swing.xml

```
<beans...>

    <bean id="sumService" class="beans.SumServiceImpl">
        <property name="listener" ref="mathService" />
    </bean>

    <bean id="diffService" class="beans.DiffServiceImpl">
        <property name="listener" ref="mathService" />
    </bean>

    <bean id="mathService" class="beans.MathServiceImpl">
        <property name="sumService" ref="sumService" />
        <property name="diffService" ref="diffService" />
    </bean>

</beans>
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final MathService mathService = ctx.getBean(MathService.class);
            System.out.println(mathService.sum(40, 2));
            System.out.println(mathService.diff(80, 3));
        }
    }
}
```

Die Ausgaben zeigen, dass sowohl die `setSumService`- und `setDiffService`-Methoden auf den Chefmathematiker als auch die `setListener`-Methode auf die Hilfsmathematiker aufgerufen werden:

```
beans.DiffServiceImpl.setListener(beans.MathServiceImpl@56ef9176)
beans.MathServiceImpl.setSumService(beans.SumServiceImpl@ff5b51f)
beans.MathServiceImpl.setDiffService(beans.DiffServiceImpl@25bbe1b6)
beans.SumServiceImpl.setListener(beans.MathServiceImpl@56ef9176)
notify: sum(40, 2)
42
notify: diff(80, 3)
77
```

3.19 Scopes

Bislang waren wir stets davon ausgegangen, dass Services statuslos sind. Solche Services können von mehreren Clients gemeinsam genutzt werden.

Neben statuslosen Services benötigen wir zuweilen aber auch statusbehaftete Services. Solche Services können natürlich nicht gemeinsam genutzt werden: jeder Client benötigt eine eigene Instanz solcher Service-Klassen.

Wir definieren ein `Calculator`-Interface:

ifaces.Calculator

```
package ifaces;

public interface Calculator {
    public abstract void add(int value);
    public abstract void subtract(int value);
    public abstract int getValue();
}
```

Die Implementierung zeigt, dass es sich um einen stateful Service handelt:

beans.CalculatorImpl

```
package beans;
// ...
public class CalculatorImpl implements Calculator {

    private int value;

    public CalculatorImpl() {
        Log.log();
    }

    public void add(int value) {
        this.value += value;
    }

    public void subtract(int value) {
        this.value -= value;
    }

    public int getValue() {
        return this.value;
    }
}
```

Jedes Mal dann, wenn über `getBean` ein `Calculator` angefordert wird, muss Spring nun ein neues(!) `CalculatorImpl`-Objekt erzeugt werden.

Bei der Definition einer Bean kann zusätzlich ein `scope`-Attribut verwendet werden:

swing.xml

```
<beans ...>

    <bean id="mathService" class="beans.MathServiceImpl" scope="singleton"/>

    <bean id="calculator" class="beans.CalculatorImpl" scope="prototype"/>

</beans>
```

Beim `"mathService"` verwenden wird den `scope`-Wert `"singleton"`, beim `"calculator"` den `scope`-Wert `"prototype"`. Der default-Wert für das `scope`-Attribut ist `"singleton"` (alle bisherigen Beispiele benutzten also implizit `"singleton"`).

`"singleton"` bedeutet, dass wiederholte Aufruf von `getBean` stets dasselbe Objekt liefern; über `"prototpye"` weisen wir Spring an, bei jedem `getBean`-Aufruf ein neues Objekt zu erzeugen und zurückzuliefern.

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            final MathService mathService1 = ctx.getBean(MathService.class);
            final MathService mathService2 = ctx.getBean(MathService.class);
            System.out.println(mathService1 == mathService2);

            final Calculator calculator1 = ctx.getBean(Calculator.class);
            final Calculator calculator2 = ctx.getBean(Calculator.class);
            System.out.println(calculator1 == calculator2);

            calculator1.add(40);
            calculator2.add(80);
            calculator1.add(2);
            calculator2.subtract(3);
            System.out.println(calculator1.getValue());
            System.out.println(calculator2.getValue());
        }
    }
}
```

Die Ausgaben:

```
beans.MathServiceImpl.<init>()
true
beans.CalculatorImpl.<init>()
beans.CalculatorImpl.<init>()
false
42
77
```

Man beachte das Resultat der Referenz-Vergleiche.

3.20 Ambiguity

Ein Interface kann von unterschiedlichen Klassen implementiert sein – wie kann dann die Anwendung auf eine spezifische Implementierung zugreifen?

Das Interface `MathService` habe zwei Implementierungen (die zweite Implementierung operiert rekursiv):

beans.MathServiceImpl1

```
package beans;
// ...
public class MathServiceImpl1 implements MathService {

    public MathServiceImpl1() {
        Log.log();
    }

    @Override
    public int sum(int x, int y) {
        return x + y;
    }

    @Override
    public int diff(int x, int y) {
        return x - y;
    }
}
```

beans.MathServiceImpl2

```
package beans;
// ...
public class MathServiceImpl2 implements MathService {

    public MathServiceImpl2() {
        Log.log();
    }

    @Override
    public int sum(int x, int y) {
        if (y == 0)
            return x;
        return 1 + sum(x, y - 1);
    }

    @Override
    public int diff(int x, int y) {
        if (y == 0)
            return x;
        return diff(x, y - 1) - 1;
```

```
    }
}
```

In der XML sind zwei Beans registriert – eine vom Typ `MathServiceImpl1`, die zweite vom Typ `MathServiceImpl2`:

spring.xml

```
<beans...>

    <bean id="mathService1" class="beans.MathServiceImpl1" />
    <bean id="mathService2" class="beans.MathServiceImpl2" />

</beans>
```

Die `Application` muss nun eine überladene `getBean`-Methode nutzen. Wurde an `getBean` bislang stets einfach das `Class`-Objekt eines Interfaces übergeben, so muss nun zusätzlich der Name übergeben werden, unter dem die gewünschte Implementierung registriert ist:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            final MathService mathService1 =
                ctx.getBean("mathService1", MathService.class);
            System.out.println(mathService1.sum(40, 2));
            System.out.println(mathService1.diff(80, 3));

            final MathService mathService2 =
                ctx.getBean("mathService2", MathService.class);
            System.out.println(mathService2.sum(40, 2));
            System.out.println(mathService2.diff(80, 3));
        }
    }
}
```

3.21 Lazy Initialization

Beans können unmittelbar im Konstruktor des `ApplicationContext`s erzeugt und registriert werden; sie können aber auch erst bei Bedarf erzeugt und registriert werden (lazy creation – Erzeugung auf Verlangen).

Seien folgende zwei Implementierungsklassen gegeben (wobei jeweils der Konstruktor seinen Aufruf diagnostiziert):

beans.SumServiceImpl

```
package beans;
// ...
public class SumServiceImpl implements SumService {
    public SumServiceImpl() {
        Log.log();
    }
    // ...
}
```

beans.DiffService

```
package beans;
// ...
public class DiffServiceImpl implements DiffService {
    public DiffServiceImpl() {
        Log.log();
    }
    // ...
}
```

spring.xml

Bei der Definition eines `bean`-Elements kann zusätzlich das `lazy-init`-Attribut angegeben werden. Der default-Wert dieses Attributs ist `false` – defaultmäßig werden die Beans also direkt im Konstruktor des `ApplicationContext`s erzeugt.

Wir definieren das Initialisierungs-Verhalten nun explizit:

```
<beans ...>

    <bean id="sumService" class="beans.SumServiceImpl" lazy-init="false" />

    <bean id="diffService" class="beans.DiffServiceImpl" lazy-init="true" />

</beans>
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            System.out.println("-------------");
            final SumService sumService = ctx.getBean(SumService.class);
            System.out.println(sumService.sum(40, 2));

            System.out.println("-------------");
            final DiffService diffService = ctx.getBean(DiffService.class);
            System.out.println(diffService.diff(80, 3));
        }
    }
}
```

Die Ausgaben zeigen, dass das `SumServiceImpl`-Objekt sofort erzeugt wird, dass `DiffServiceImpl`-Objekt aber erst dann, wenn es via `getBean` angefordert wird:

```
beans.SumServiceImpl.<init>()
-------------
42
-------------
beans.DiffServiceImpl.<init>()
77
```

3.22 Closeables

Auf Beans, deren Klassen das Interface `Closeable` (oder `AutoCloseable`) implementieren, ruft Spring beim Schließen des `ApplicationContext`s automatisch deren `close`-Methode auf.

Die folgende Klasse ist `Closeable`:

beans.FileLogger

```
package beans;
// ...
public class FileLogger implements Closeable {

    private final PrintWriter writer;

    public FileLogger(String filename) throws IOException {
        this.writer = new PrintWriter(new FileOutputStream(filename), true);
    }

    public void log(String s) {
        this.writer.println(s);
    }

    @Override
    public void close() throws IOException {
        Log.log();
        this.writer.close();
    }
}
```

Die Klasse `MathService` nutzt einen `FileLogger`:

beans.MathService

```
package beans;

public class MathService {

    private final FileLogger logger;

    public MathService(FileLogger logger) {
        this.logger = logger;
    }

    public int sum(int x, int y) {
        this.logger.log("sum(" + x + ", " + y + ")");
        return x + y;
    }

    public int diff(int x, int y) {
```

```
        this.logger.log("diff(" + x + ", " + y + ")");
        return x - y;
    }
}
```

Über die `spring.xml` wird ein `MathService` erzeugt, wobei dem Konstruktor von `MathService` ein `FileLogger` übergeben wird:

spring.xml

```
<beans ...>
    <bean id="mathService" class="beans.MathService" >
        <constructor-arg>
            <bean class="beans.FileLogger" >
                <constructor-arg value="./log.txt"/>
            </bean>
        </constructor-arg>
    </bean>
</beans>
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final MathService sumService = ctx.getBean(MathService.class);
            System.out.println(sumService.sum(40, 2));
            System.out.println(sumService.diff(80, 3));
        }
    }
}
```

Die Ausgaben zeigen, dass beim Schließen des `ApplicationContext`s die `close`-Methode des `FileLogger`s aufgerufen wird.

4 Der AnnotationConfigApplicationContext

Neben der XML-basierten Konfigurations-Varianten existiert seit Java 5 die Möglichkeit, via Annotationen zu konfigurieren. Eine XML-Datei ist dann nicht erforderlich.

Wir können aber auch Annotations-basierte mit XML-basierten Konfigurationen mischen.

Hier eine Übersicht zu den Abschnitten dieses Kapitels:

- Abschnitt 1 zeigt, wie Klassen via `@Component` ausgezeichnet werden können. Zugleich wird auch demonstriert, wie bei dieser Konfiguration-Varianten Constructor-Injection funktioniert.
- Abschnitt 2 demonstriert die Method-Injection (Method-Injection ist nicht(!) dasselbe wie die im letzten Kapitel behandelte Property-Injection…) Hier wird die Annotation `@Autowired` genutzt werden.
- Abschnitt 3 demonstriert Field-Injection (bei der XML-Variante unbekannt). Auch hier wird `@Autowired` benutzt).
- Abschnitt 4 beschreibt das `required`-Attribut der `@Autowired`-Annotation.
- Abschnitt 5 zeigt, wie Lebenszyklus-Methoden definiert werden (via `@PostConstruct` und `@PreDestroy`).
- Im Abschnitt 6 wird gezeigt, wie die `@Scope`-Annotation verwendet werden kann.
- In den Abschnitten 7 und 8 wird gezeigt, dass auch die `Application` selbst als `@Component` fungieren kann – und wie wir auf `getBean`-Aufufe komplett verzichten können.
- Kapitel 9 demonstriert Probleme mit der Eindeutigkeit resp. Mehrdeutigkeit.
- Im Kapitel 10 zeigen wir, wie via Annotations die Einträge von Property-Dateien eingelesen werden können.
- Im Kapitel 11 und 12 schließlich werden zwei Möglichkeiten vorgestellt, XML-Konfigurationen und Annotation-basierte Konfigurationen zu mischen.

4.1 Constructor-Injection

Wir bauen im folgenden wieder unser `MathService`-System via Constructor-Injection auf.

Die Implementierungsklassen werden mit `@Component` (oder `@Service` – eine "Spezialisierung" von `@Component`) ausgezeichnet:

beans.SumServiceImpl

```
package beans;
// ...
import org.springframework.stereotype.Service;
import org.springframework.stereotype.Component;

@Service
//@Component
public class SumServiceImpl implements SumService {
    @Override
    public int sum(int x, int y) {
        return x + y;
    }
}
```

beans.DiffServiceImpl

```
package beans;
// ...
@Service
public class DiffServiceImpl implements DiffService {
    @Override
    public int diff(int x, int y) {
        return x - y;
    }
}
```

Der Konstruktor von `MathServiceImpl` kann mit `@Autowired` annotiert sein – sofern aber nur ein einziger Konstruktor existiert, kann `@Autowired` auch fehlen:

beans.MathServiceImpl

```
package beans;
// ...
@Component
public class MathServiceImpl implements MathService {

    private final SumService sumService;
    private final DiffService diffService;
```

```
    // @Autowired
    public MathServiceImpl(SumService sumService, DiffService diffService) {
        this.sumService = sumService;
        this.diffService = diffService;
    }
    @Override
    public int sum(int x, int y) {
        return this.sumService.sum(x, y);
    }
    @Override
    public int diff(int x, int y) {
        return this.diffService.diff(x, y);
    }
}
```

Die `@Component`- resp. `@Service`-Annotation kann mit einem `value` ausgestattet werden – unter dem dort angegebenen Namen werden dann die entsprechende Bean registriert – z.B. `@Component("math")`.

Die `Application` benutzt nun einen `AnnotationConfigApplicationContext`. Bei der Erzeugung eines solchen Kontexts wird dem Konstruktor der Name (oder die Namen) desjenigen Package übergeben, welche Spring nach `@Component`s scannen soll:

appl.Application

```
package appl;
// ...
import org.springframework.context.annotation
       .AnnotationConfigApplicationContext;

public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("beans")) {

            final MathService mathService = ctx.getBean(MathService.class);
            System.out.println(mathService.sum(40, 2));
            System.out.println(mathService.diff(80, 3));
        }
    }
}
```

4.2 Method-Injection

Das folgende Beispiel nutzt statt Constructor-Injection das Method-Injection-Verfahren.

Die Klassen `SumServiceImpl` und `DiffServiceImpl` werden unverändert aus dem letzten Abschnitt übernommen.

In MathServiceImpl wird eine Methode definiert, der zwei Parameter übergeben werden müssen: ein `Parameter` vom Typ `SumService` und ein zweiter vom Typ `DiffService`. Der Name der Methode ist `schallUndRauch` (sic!). Aufgrund der ihr übergebenen Parameter initialsiert sie die entsprechenden Instanzvariablen. Sie liefert void.

Die Methode muss mit `@Autowired` gekennzeichnet sein:

beans.MathServiceImpl

```
package beans;
// ...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MathServiceImpl implements MathService {

    private SumService sumService;
    private DiffService diffService;

    @Autowired // REQUIRED
    public void schallUndRauchh(
              SumService sumService,
              DiffService diffService) {
        this.sumService = sumService;
        this.diffService = diffService;
    }

    @Override
    public int sum(int x, int y) {
        return this.sumService.sum(x, y);
    }

    @Override
    public int diff(int x, int y) {
        return this.diffService.diff(x, y);
    }
}
```

Statt einer Methode mit zwei Parametern hätten wir auch zwei Methoden mit jeweils einem Parameter definieren können (und beide dieser Methoden als `@Autowired` auszeichnen können):

```
    @Autowired
    public void schall(SumService sumService) {
        this.sumService = sumService;
    }

    @Autowired
    public void rauch(DiffService diffService) {
        this.diffService = diffService;
    }
```

Statt `@Autowired` können wir auch die Standard-Java-Annotation `@Inject` verwenden können (die z.B. auch vom CDI-Framework verwendet wird):

```
    // @Inject // Alternative. Add to classpath: CDI-user-lib
```

Man beachte, dass diese Method-Injection sich von der im letzten Kapitel behandelten Property-Injection unterscheidet: wir können Methoden beliebigen Namens mit beliebig vielen Argumenten definieren. Sofern sie als `@Autowired` gekennzeichnet sind, wird Spring versuchen, diese Methoden mit entsprechenden Argumenten aufzurufen. Die Methoden dürfen nur auch `private` sein. Sie sollten `void` liefern.

4.3 Field-Injection

Im folgenden geht's um Field-Injection: Spring wird die benötigten Services direkt an die Instanzvariablen von `MahServiceImpl` zuweisen.

Die zu initialisierenden Instanzvariablen müssen mit `@Autowired` gekennzeichnet sein:

beans.MathServiceImpl

```
package beans;
// ...
@Component
public class MathServiceImpl implements MathService {

    @Autowired   // REQUIRED!
    private SumService schall;

    @Autowired   // REQUIRED!
    private DiffService rauch;

    @Override
    public int sum(int x, int y) {
        return this.schall.sum(x, y);
    }

    @Override
    public int diff(int x, int y) {
        return this.rauch.diff(x, y);
    }
}
```

Der Nachteil dieses Verfahrens liegt darin, dass wir in den Prozess der Initialisierung nicht mehr eingreifen können (sei es auch nur, dass wir die Initialisierung loggen wollen).

Einen zweiten Nachteil könnte man darin sehen, dass auf diese Weise eine Vielzahl von Instanzvariablen als `@Autowired` gekennzeichnet werden – und damit möglicherweise verdeckt wird, dass unsere Klasse von viel zu vielen weitern Klassen abhängig ist. Hätten wir stattdessen Constructor-Injection verwendet, hätten wir dieses Problem sofort gesehen (wir würden keinen Konstruktor mit zwanzig Parametern schreiben wollen…)

Aber die Einfachheit der Field-Injection hat natürlich auch ihren Reiz.

4.4 Required-Attribute of Autowired

Die `@Autowired`-Annotation kann mit einem `required`-Attribute ausgestattet werden. Der default von `required` ist `false`.

In unserem Beispiel gibt es zwar die Implementierung für das `DiffService`-Interface, aber keine Implementierung für das `SumService`-Interface. Die Klasse `MathServiceImpl` muss im Falle des Aufrufs der `sum`-Methode also selbst rechnen (der `diff`-Aufruf kann weiter an den `DiffServiceImpl` delegiert werden).

beans.MathServiceImpl

```
package beans;
// ...
@Component
public class MathServiceImpl implements MathService {

    private SumService sumService;
    private DiffService diffService;

    @Autowired(required = false)
    public void setSumService(SumService sumService) {
        this.sumService = sumService;
    }

    @Autowired(required = true)  // true is the default
    public void setDiffService(DiffService diffService) {
        this.diffService = diffService;
    }

    @Override
    public int sum(int x, int y) {
        if (this.sumService == null)
            return x + y;
        return this.sumService.sum(x, y);
    }

    @Override
    public int diff(int x, int y) {
        return this.diffService.diff(x, y);
    }
}
```

4.5 Lifecycle

Lifecycle-Methoden können wie auch im Falle der XML-Konfiguration beliebige Namen haben – sie müssen nur parameterlos sein.

Solche Methoden können mit den Standard-Java-Annotationen `@PostConstruct` und `@PreDestroy` ausgezeichnet sein:

beans.SumServiceImpl

```
package beans;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
// ...

@Component
public class SumServiceImpl implements SumService {
    public SumServiceImpl() {
        Log.log();
    }
    @PostConstruct
    public void postConstruct() {
        Log.log();
    }
    @PreDestroy
    public void preDestroy() {
        Log.log();
    }
    @Override
    public int sum(int x, int y) {
        return x + y;
    }
}
```

beans.DiffServiceImpl

```
package beans;
// ...
@Component
public class DiffServiceImpl implements DiffService {

    // wie in SumServiceImpl...

}
```

beans.MathServiceImpl

```
package beans;
// ...
@Component
public class MathServiceImpl implements MathService {

    private SumService sumService;
    private DiffService diffService;

    public MathServiceImpl() {
        Log.log();
    }

    @PostConstruct
    public void postConstruct() {
        Log.log();
    }

    @PreDestroy
    public void preDestroy() {
        Log.log();
    }

    @Autowired
    public void setSumService(SumService sumService) {
        Log.log();
        this.sumService = sumService;
    }

    @Autowired
    public void setDiffService(DiffService diffService) {
        Log.log();
        this.diffService = diffService;
    }

    // sum, diff
}
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("beans")) {
            final MathService mathService = ctx.getBean(MathService.class);
            System.out.println(mathService.sum(40, 2));
            System.out.println(mathService.diff(80, 3));
        }
    }
}
```

Die Ausgaben:

```
beans.DiffServiceImpl.<init>()
beans.DiffServiceImpl.postConstruct()
beans.MathServiceImpl.<init>()
beans.SumServiceImpl.<init>()
beans.SumServiceImpl.postConstruct()
beans.MathServiceImpl.setSumService()
beans.MathServiceImpl.setDiffService()
beans.MathServiceImpl.postConstruct()
42
77
beans.MathServiceImpl.preDestroy()
beans.SumServiceImpl.preDestroy()
beans.DiffServiceImpl.preDestroy()
```

Auch hier wird die mit `@PostConstruct` ausgezeichnete Method dann aufgerufen, wenn die Bean betriebsbereit ist (also erst dann, wenn alle Injections stattgefunden haben).

Die mit `@PreDestroy` annotierte Methode wird dann aufgerufen, wenn der Container (der `ApplicationContext`) geschlossen wird.

4.6 Scopes

Um Beans explizit dem Singleton- resp. dem Prototype-Scope zuzuweisen, kann die entsprechende Implementierungsklasse mit der `@Scope`-Annotation ausgestattet werden. Mögliche Werte für diese Annotation: `SCOPE_SINGLETON` und `SCOPE_PROTOTYPE`:

beans.MathServiceImpl

```
package beans;
// ...
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;

@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)  // the default
public class MathServiceImpl implements MathService {
    public int sum(int x, int y) {
        return x + y;
    }
    public int diff(int x, int y) {
        return x - y;
    }
}
```

beans.CalculatorImpl

```
package beans;
// ...
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class CalculatorImpl implements Calculator {
    private int value;
    public void add(int value) {
        this.value += value;
    }
    public void subtract(int value) {
        this.value -= value;
    }
    public int getValue() {
        return this.value;
    }
}
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("beans")) {

            final MathService mathService1 = ctx.getBean(MathService.class);
            final MathService mathService2 = ctx.getBean(MathService.class);
            System.out.println(mathService1 == mathService2);

            final Calculator calculator1 = ctx.getBean(Calculator.class);
            final Calculator calculator2 = ctx.getBean(Calculator.class);
            System.out.println(calculator1 == calculator2);

            calculator1.add(40);
            calculator2.add(80);
            calculator1.add(2);
            calculator2.subtract(3);
            System.out.println(calculator1.getValue());
            System.out.println(calculator2.getValue());
        }
    }
}
```

Die Ausgaben:

```
beans.MathServiceImpl.<init>()
true
beans.CalculatorImpl.<init>()
beans.CalculatorImpl.<init>()
false
42
77
```

4.7 Application as Component – Part 1

Natürlich könnte auch die Klasse `Application`, die bislang stets nur die statische `main`-Methode enthielt, instanziiert werden. Ist die Klasse dann mit `@Component` annotiert, könnte die `main`-Methode sich darauf beschränken, via `getBean` ein `Application`-Objekt zu erzeugen und auf dieses dann etwa eine Methode namens `run` aufzurufen. Und dem `Application`-Objekt könnten dann wiederum die benötigten Services injiziert werden.

Dem Konstruktor des `ApplicationContext`s müssen nun zwei Package-Namen übergeben werden: "`beans`" und "`appl`":

appl.Application

```
package appl;
// ...

@Component
public class Application {

    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl", "beans")) {
            final Application application = ctx.getBean(Application.class);
            application.run();
        }
    }

    @Autowired
    private MathService mathService;

    @Autowired
    private SumService sumService;

    @Autowired
    private DiffService diffService;

    private void run() {
        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
        System.out.println(sumService.sum(40, 2));
        System.out.println(diffService.diff(80, 3));
    }
}
```

4.8 Application as Component – Part 2

Im letzten Abschnitt benötigten wir nurmehr einen einzigen `getBean`-Lookup. Wie kommen auch ganz ohne `getBean`-Aufruf aus:

appl.Application

```
package appl;
// ...
@Component
public class Application {

    public static void main(String[] args) {
        new AnnotationConfigApplicationContext("appl", "beans").close();
    }

    @Autowired
    private MathService mathService;

    @Autowired
    private SumService sumService;

    @Autowired
    private DiffService diffService;

    @PostConstruct
    private void run() {
        System.out.println(this.mathService.sum(40, 2));
        System.out.println(this.mathService.diff(80, 3));
        System.out.println(this.sumService.sum(40, 2));
        System.out.println(this.diffService.diff(80, 3));
    }
}
```

Wir haben die `run`-Methode einfach mit `@PostConstruct` annotiert. Man beachte, dass der `ApplicationContext` natürlich erst dann geschlossen wird, wenn die `@PostContruct`-Methode komplett abgearbeitet ist.

4.9 Ambiguity

Angenommen, es existieren zwei Implementierungen des `MathService`-Interfaces: `MathServiceImpl1` und `MathServiceImpl2`. Die erste Implementierung wird unter dem Namen `"mathServiceSimple"`, die zweite unter `"mathServiceRecursive"` registriert:

beans.SumMathImpl1

```
package beans;
// ...
@Component("mathServiceSimple")
public class MathServiceImpl1 implements MathService {
      @Override
      public int sum(int x, int y) {
            return x + y;
      }
      @Override
      public int diff(int x, int y) {
            return x - y;
      }
}
```

beans.SumMathImpl2

```
package beans;
// ...
@Component("mathServiceRecursive")
public class MathServiceImpl2 implements MathService {
    @Override
    public int sum(int x, int y) {
        if (y == 0)
            return x;
        return 1 + sum(x, y - 1);
    }
    @Override
    public int diff(int x, int y) {
        if (y == 0)
            return x;
        return diff(x, y - 1) - 1;
    }
}
```

Um der Anwendung beide Referenzen zu injizieren, muss nun die Annotation `@Autowired` in Kombination mit `@Qualifier` verwendet werden. An `@Qualifier` wird der Name übergeben, unter welchem die Bean registriert ist.

Statt der Kombination von `@Autowired` und `@Qualifier` kann auch einfach die `@Resource`-Annotation verwendet werden (wobei an `@Resource` ebenfalls der Name übergeben wird).

appl.Application

```
package appl;
// ...
@Component
public class Application {

    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl", "beans")) {
            final Application application = ctx.getBean(Application.class);
            application.run();
        }
    }

    private MathService mathServiceSimple;

    @Resource(name = "mathServiceSimple")
    public void setMathServiceSimple(MathService mathService) {
        this.mathServiceSimple = mathService;
    }

    private MathService mathServiceRecursive;

    @Autowired
    @Qualifier("mathServiceRecursive")
    public void setMathServiceRecursive(MathService mathService) {
        this.mathServiceRecursive = mathService;
    }

    private void run() {
        System.out.println(mathServiceSimple.sum(40, 2));
        System.out.println(mathServiceSimple.diff(80, 3));
        System.out.println(mathServiceRecursive.sum(40, 2));
        System.out.println(mathServiceRecursive.diff(80, 3));
    }
}
```

4.10 Property Files

Auch mittels Annotations lassen sich die Daten von Property-Files einlesen.

Sei wieder die folgende Property-Datei gegeben:

gui.properties

```
title       My Window
x           100
y           200
width       700
height      100
```

Wir können folgende `GuiParams`-Klasse definieren:

```
package util;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.PropertySource;
import org.springframework.stereotype.Component;

@Component
@PropertySource("classpath:gui.properties")
public class GuiParams {

    @Value("${title}")
    public String title;

    @Value("${x}")
    public int x;

    @Value("${y}")
    public int y;

    @Value("${width}")
    public int width;

    @Value("${height}")
    public int height;

    @Override
    public String toString() {
        return this.getClass().getSimpleName() +
                " [" + this.title + ", " + this.x + ", " + this.y + ", " +
                this.width + ", " + this.height + "]";
    }
}
```

Die Klasse wird mit `@Component` und `@PropertySource` annotiert. Die einzelnen Felder der Klasse werden mit `@Value` annotiert – wobei das Argument von `@Value` sich über `"${<name>}` auf den jeweiligen Eintrag der Property-Datei bezieht. Die Felder werden dann von Spring initialisiert.

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("util")) {
            final GuiParams gui = ctx.getBean(GuiParams.class);
            System.out.println(gui);
        }
    }
}
```

Die Ausgaben:

```
GuiParams [My Window, 100, 200, 700, 100]
```

4.11 Mixing with XML – Variante 1

XML-Kontexte können mit Annotation-Kontexten gemischt werden.

Die Klassen `SumServiceImpl` und `DiffServiceImpl` besitzen keinerlei Annotationen (können also nur mittels eines XML-Kontexts instanziiert werden):

SumServiceImpl

```
package beans;
// ...
public class SumServiceImpl implements SumService {
    @Override
    public int sum(int x, int y) {
        return x + y;
    }
}
```

SumServiceImpl

```
package beans;
// ...
public class DiffServiceImpl implements SumService {
    @Override
    public int diff(int x, int y) {
        return x - y;
    }
}
```

Die Klasse `MathServiceImpl` aber ist mit `@Component` annotiert. Der Konstruktor der Klasse ist mit `@Autowired` annotiert (obwohl dies überflüssig ist) – und die Klasse enthält zwei mit `@PostConstruct` resp. `@PreDestroy` annotierte Methoden:

MathServiceImpl

```
package beans;
// ...
@Component
public class MathServiceImpl implements MathService {

    private SumService sumService;
    private DiffService diffService;

    @Autowired
    public MathServiceImpl(SumService sumService, DiffService diffService) {
        Log.log(sumService, diffService);
        this.sumService = sumService;
        this.diffService = diffService;
    }
```

```
    @PostConstruct
    public void postConstruct() {
        Log.log();
    }

    @PreDestroy
    public void PreDestroy() {
        Log.log();
    }

    @Override
    public int sum(int x, int y) {
        return this.sumService.sum(x, y);
    }

    @Override
    public int diff(int x, int y) {
        return this.diffService.diff(x, y);
    }
}
```

spring.xml

```
<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-4.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-4.0.xsd">

    <context:annotation-config/>
    <context:component-scan base-package="beans"/>

    <bean id="sumService" class="beans.SumServiceImpl"/>
    <bean id="diffService" class="beans.DiffServiceImpl"/>

</beans>
```

Via XML wird Spring angewiesen, das Package `"beans"` nach Klassen zu suchen, die mit `@Component` annotiert sind (`context:component-scan`). In diesem Package wird die `@Component`-Klasse `MathServiceBean` gefunden.

Und aufgrund der beiden expliziten `bean`-Definitionen können `SumMathBean` und `DiffMathBean` instanziiert werden.

Die `Application` erzeugt nur einen einzigen Kontext: einen XML-Kontext:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            final MathService mathService = ctx.getBean(MathService.class);
            System.out.println(mathService.sum(40, 2));
            System.out.println(mathService.diff(80, 3));

            final SumService sumService = ctx.getBean(SumService.class);
            System.out.println(sumService.sum(40, 2));

            final DiffService diffService = ctx.getBean(DiffService.class);
            System.out.println(diffService.diff(80, 3));
        }
    }
}
```

4.12 Mixing with XML – Variante 2

Alle Klassen außer der `Application`-Klasse werden unverändert aus dem letzten Abschnitt übernommen.

In der `spring.xml` wird auf den `component-scan`-Eintrag verzichtet:

spring.xml

```
<beans ...>
    <bean id="sumService" class="beans.SumServiceImpl"/>
    <bean id="diffService" class="beans.DiffServiceImpl"/>
</beans>
```

Die `Application` benutzt nun zwei Kontexte: eine Vater- und einen Kind-Kontext:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext parent =
                new ClassPathXmlApplicationContext("spring.xml")) {
            try (final AnnotationConfigApplicationContext ctx =
                    new AnnotationConfigApplicationContext()) {
                ctx.setParent(parent);
                ctx.scan("beans");
                ctx.refresh();

                final MathService mathService =
                    ctx.getBean(MathService.class);
                System.out.println(mathService.sum(40, 2));
                System.out.println(mathService.diff(80, 3));

                final SumService sumService =
                    ctx.getBean(SumService.class);
                System.out.println(sumService.sum(40, 2));

                final DiffService diffService =
                    ctx.getBean(DiffService.class);
                System.out.println(diffService.diff(80, 3));
            }
        }
    }
}
```

Bevor der Annotation-Kontext erzeugt wird, wird zunächst ein XML-Kontext erzeugt. Der Annotations-Kontext wird dann mittels des parameterlosen(!) Konstruktors erzeugt. Nach seiner Erzeugung ist dieser Kontext also zunächst einmal leer.

Über die Methode `setParent` wird dem Annotations-Kontext der zuvor erzeugt XML-Kontext als Vater übergeben. Somit kennt nun auch der Annotations-Kontext die Bean-Definitionen des XML-Kontexts (die Definitionen für `SumService` und `DiffService`).

Dann wird der Annotations-Kontext via `scan` gefüllt – mit der Bean-Definition für den `MathService`. Nun kennt der Annotations-Kontext alle drei Bean-Definitionen.

Mittels `refresh` wird der Annotation-Kontext nun zugeschnürt (`refresh` darf nur ein einziges Mal aufgerufen werden!).

Die eigentliche `Application` benutzt anschließend nurmehr den Annotation-Kontext.

Man beachte dabei: Beide(!) Resource-Try-Blöcke sind notwendig (jeder der beiden Kontexte muss explizit geschlossen werden – das Schließen des Kindes übernimmt also nicht der Vater (und auch nicht umgekehrt…)

5 Factories

In allen bisherigen Beispielen wurden die Beans, die Spring verwaltet, auch von Spring selbst erzeugt (dabei wurde entweder der parameterlose Konstruktor aufgerufen oder aber – im Falle der Constructor-Injection – ein parametrisierter Konstruktor.

Wie aber sollte Spring z.B. eine `Connection` erzeugen? Zum Erzeugen einer `Connection` bedarf es (im einfachsten Falle) des Aufrufs der statischen `DriverManager`-Methode `getConnection` (die mit drei Parametern aufgerufen werden muss: mit der URL, dem User und dem Password). `getConnection` liefert dann ein Objekt einer Klasse, die das Interface(!) `Connection` implementiert (die konkrete Klasse ist Datenbank-abhängig.) In solchen Fällen benötigen wir Factories. Statt bei Spring die Klasse eines "Produkts" zu registrieren, registrieren wie eine Factory (oder eine Factory-Methode), welche ein solches Produkt produziert. Dann muss Spring natürlich wissen, dass es sich bei der registrierten Bean eigentlich nicht um eine Bean, sondern um eine Bean-Factory handelt.

Hier eine Übersicht zu den Abschnitten dieses Kapitels:

- Im ersten Abschnitt stellen wir das `factory-method`-Attribut einer `bean`-Definition vor, mittels dessen eine statische Factory-Methode benutzt werden kann.
- Im zweiten Abschnitt zeigen wird, wie nicht statische Factory-Methoden benutzt werden können. Hier geht's um das `bean`-Attribut `factory-bean`.
- Im dritten Abschnitt wird eine Factory-Klasse benutzt, die vom Spring-eigenen Interface `FactoryBean` abgeleitet ist.
- Die Abschnitte 4, 5, 6 und 7 zeigen, wie Listen und Maps über Factories erzeugt werden können.
- Abschnitt 8 enthält ein kleines "praktisches" Beispiel: die Konfiguration einer einfachen Anwendung mit Operatoren.
- Im Abschnitt 9 zeigen wir, wie Factories via `@Bean` eingerichtet werden können.
- Im Abschnitt 10 geht's um die Möglichkeit, mittels der Expression-Language Objekte erzeugen zu lassen.
- Im letzten Abschnitt zeigen wir, wie eine GUI mittels Spring beschrieben und erzeugt werden kann.

5.1 Static Factory Methods

Um eine Datenbank-`Connection` zu besorgen, benutzen wir in EE-Anwendungen gewöhlich einen Connection-Pool (eine `DataSouce`). Wir können in einfachen SE-Anwendungen auf einen solchen Pool auch verzichten und eine `Connection` via `DriverManager.getConnection` zu erzeugen. Genau auf diesem Wege wollen wir uns im Folgenden eine Connection vom `ClassPathXmlApplicationContext` erzeugen lassen.
Die statische `DriverManager.getConnection`-Methode verlangt drei Parameter: die URL, den User und das Password.
Nun können wir aber in einem `ApplicationContext` keine statische Methode aufrufen. Aber wir können eine statische parameterlose Factory-Methode aufrufen lassen, die dann ihrerseits eine `Connection` erzeugt.
Wir bauen eine `ConnectionFactory` mit zwei parameterlosen Factory-Methoden – `createConnectionA` und `createConnectionB`:

beans.ConnectionFactory

```
package beans;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {

    public static Connection createConnectionA() throws SQLException {
        return createConnection(
                "jdbc:derby:../dependencies/derby/dataA;create=true",
                "user",
                "password");
    }

    public static Connection createConnectionB() throws SQLException {
        return createConnection(
                "jdbc:derby:../dependencies/derby/dataB;create=true",
                "user",
                "password");
    }

    private static Connection createConnection(
            String url, String user, String password) throws SQLException {
        final Connection con = DriverManager.getConnection(
            url, user, password);
        con.setAutoCommit(false);
        return con;
    }
}
```

Die beiden Methoden benutzen jeweils eine eigene URL (`...dataA...` und (`...dataB...`).

Die `spring.xml` enthält zwei `<bean>`-Einträge, die ihrerseits jeweils ein `factory-method`-Attribut besitzen:

spring.xml

```
<beans .../>

    <bean id="connectionA" class="beans.ConnectionFactory"
        factory-method="createConnectionA" scope="prototype"/>

    <bean id="connectionB" class="beans.ConnectionFactory"
        factory-method="createConnectionB" scope="prototype"/>

</beans>
```

Wenn wir nun die `getBean`-Methode z.B. mit der Id "`connectionA`" aufrufen, so wird die `createConnectionA`-Methode der `ConnectionFactory` aufgerufen – und wir erhalten das zurück, was diese Methode zurückliefert – nämlich eine `Connection`. (Man beachte den verwendeten `scope`!).

In der folgenden `main`-Mehtode lassen wir jeweils `Connection`s zu den beiden Datenbanken erzeugen:

appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) {

        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final Connection conA1 = ctx.getBean(
                "connectionA", Connection.class);
            System.out.println(conA1);
            final Connection conA2 = ctx.getBean(
                "connectionA", Connection.class);
            System.out.println(conA2);
            final Connection conB1 = ctx.getBean(
                "connectionB", Connection.class);
            System.out.println(conB1);
            final Connection conB2 = ctx.getBean(
                "connectionB", Connection.class);
            System.out.println(conB2);
        }
    }
}
```

Hier die Ausgaben:

```
org.apache.derby.impl.jdbc.EmbedConnection40@1632682988 (XID = 18),
    (SESSIONID = 0), (DATABASE = ../dependencies/derby/dataA)...
org.apache.derby.impl.jdbc.EmbedConnection40@793331940 (XID = 19),
    (SESSIONID = 1), (DATABASE = ../dependencies/derby/dataA)...)
org.apache.derby.impl.jdbc.EmbedConnection40@590845366 (XID = 18),
    (SESSIONID = 0), (DATABASE = ../dependencies/derby/dataB)...)
org.apache.derby.impl.jdbc.EmbedConnection40@116405378 (XID = 19),
    (SESSIONID = 1), (DATABASE = ../dependencies/derby/dataB)...)
```

Der Nachteil dieser Lösung liegt darin, dass die `getConnection`-Parameter in der Factory-Klasse fest verdrahtet sind…

5.2 Non-Static Factory Methods

Statt zur Produktion einer `Connection` eine statische Methode aufrufen zu lassen, können wir auch eine nicht-statische Methode aufrufen lassen.

Wir verwenden folgende Klasse:

beans.ConnectionFactory

```
package beans;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {

    private final String url;
    private final String user;
    private final String password;

    public ConnectionFactory(String url, String user, String password) {
        this.url = url;
        this.user = user;
        this.password = password;
    }

    private Connection createConnection() throws SQLException {
        final Connection con = DriverManager.getConnection(
            this.url, this.user, this.password);
        con.setAutoCommit(false);
        return con;
    }
}
```

Wir können via Spring diese Klasse instantiieren lassen – wobei der parametrisierte Konstruktor der Klasse verwendet wird. Die Parameter des Konstruktors werden Attributen der Klasse zugewiesen, auf welche dann die nicht-statische `createConnection`-Methode zugreifen kann.

In der Konfigurations-Datei definieren wir nun vier `<bean>`-Elemente – zwei für die beiden zu erzeugenden Factories und zwei für die mit deren Hilfe zu erzeugenden Connections.

In den `<bean>`-Einträgen für die Factories definieren wir die jeweils zu benutzende URL, den zu benutzenden User und das zu benutzende Password. In den `<bean>`-Einträgen für die zu erzeugenden `Connection` referenzieren wir via `factoy-bean` und `factory-method` das zu benutzende Factory-Objekt und die zu benutzenden Factory-Methode:

spring.xml

```
<beans ... />

    <bean id="connectionAFactory" class="beans.ConnectionFactory">
        <constructor-arg
            value="jdbc:derby:../dependencies/derby/dataA;create=true" />
        <constructor-arg value="user" />
        <constructor-arg value="password" />
    </bean>

    <bean id="connectionBFactory" class="beans.ConnectionFactory">
        <constructor-arg
            value="jdbc:derby:../dependencies/derby/dataB;create=true" />
        <constructor-arg value="user" />
        <constructor-arg value="password" />
    </bean>

    <bean id="connectionA" class="java.sql.Connection"
        factory-bean="connectionAFactory"
        factory-method="createConnection"
        scope="prototype" />

    <bean id="connectionB" class="java.sql.Connection"
        factory-bean="connectionBFactory"
        factory-method="createConnection"
        scope="prototype" />

</beans>
```

Alle konkreten `Connection`-Parameter sind nun in der `spring.xml` hinterlegt. Und es gibt nur noch eine einzige Factory-Methode, deren Verhalten durch die an den Kontruktor von `ConnectionFactory` übergebenen Parameter festgelegt wird.

Wir verwenden dieselbe `Application`-Klasse wie im letzten Abschnitt.

5.3 FactoryBean

Wir verwenden auch hier wiederum dieselbe `Application`-Klasse wie im letzten und vorletzten Abschnitt.
Diesmal aber ist die Factory-Klasse vom Spring-Interface `FactoryBean` abgeleitet. Dieses Interface ist mit einem Typ parametrisiert – mit dem Typ des von der Factory zu erzeugenden Produkts.
Das Interface `FactoryBean<T>` spezifiziert drei Methoden: ein Methode `getObject`, die ein `T` liefern muss; die Methode `getObjectType`, die das den `T` beschreibende `Class<T>`-Objekt liefern muss; und schließlich eine Methode `isSingleton`, die `true` liefern muss, wenn das Objekt als Singleton fungieren soll und `false`, falls bei jedem Aufruf von `getBean` (oder bei jeder Injection) ein neues Objekt erzeugt werden soll (Prototype):

beans.ConnectionFactory

```
package beans;
// ...
import org.springframework.beans.factory.FactoryBean;

public class ConnectionFactory implements FactoryBean<Connection> {

    private final String url;
    private final String user;
    private final String password;

    public ConnectionFactory(String url, String user, String password) {
        Log.log();
        this.url = url;
        this.user = user;
        this.password = password;
    }

    @Override
    public Connection getObject() throws Exception {
        final Connection con = DriverManager.getConnection(
            this.url, this.user, this.password);
        con.setAutoCommit(false);
        return con;
    }

    @Override
    public Class<?> getObjectType() {
        return Connection.class;
    }

    @Override
    public boolean isSingleton() {
        return false;
    }
}
```

In der `spring.xml` kann die Factory einfach genauso registriert werden wie eine gewöhnlich Bean. Spring erkennt, dass die dort angegebene Klasse das Interface `BeanFactory` implementiert und kann somit diese Factory mit der Produktion des Produkts beauftragen.

spring.xml

```
<beans .../>

    <bean id="connectionA" class="beans.ConnectionFactory">
        <constructor-arg
            value="jdbc:derby:../dependencies/derby/dataA;create=true" />
        <constructor-arg value="user" />
        <constructor-arg value="password" />
    </bean>

    <bean id="connectionB" class="beans.ConnectionFactory">
        <constructor-arg
            value="jdbc:derby:../dependencies/derby/dataB;create=true" />
        <constructor-arg value="user" />
        <constructor-arg value="password" />
    </bean>

</beans>
```

Man beachte, dass die `<bean>`-Einträge kein(!) `scope="prototype"`-Attribut haben – denn dieses Attribut würde sich auf die Factory beziehen, so dann wir bei jedem `getBean`-Aufruf eine neue Factory erzeugen würden. Stattdessen liefert die `isSingleton`-Methode der Factory-Klasse `false` – so dann bei jedem `getBean`-Aufruf ein neues Produkt(!) erzeugt wird (also eine neue `Connection`).

Obwohl als `class` z.B. für `"connectionA"` die Klasse `"benas.ConnectionFactory"` eingetragen ist, wird `getBean` nicht die Factory liefern, sondern das von ihr produzierte Produkt – eine `Connection`.

5.4 ListFactoryBean With Values

Wir hatten bereits im Kapitel zur XML-Konfiguration gezeigt, wie aufgrund einer `spring.xml` Listen erzeugt und wie diese in andere Objekte injiziert werden (wir erinnern uns an den `LanguageZoo`).

Im folgenden wird nun gezeigt, wie Listen direkt als Ergebnis eines `getBean`-Lookups zurückgeliefert werden können.

Wir wollen eine Liste mit vier Strings zurückliefern. Wir können eine Spring-eigene `ListFactoryBean` nutzen und diese über deren `sourceList`-Property mit einer String-Liste ausstatten. Da es sich um eine `BeanFactory` handelt, wird nur aber nicht die Factory zurückgeliefert, sondern die in ihr enthaltene Liste:

spring.xml

```
<beans ...>

    <bean id="languages"
        class="org.springframework.beans.factory.config.ListFactoryBean">
        <property name="sourceList">
            <list>
                <value>Pascal</value>
                <value>Modula</value>
                <value>Oberon</value>
                <value>Eiffel</value>
            </list>
        </property>
    </bean>
</beans>
```

Die Klasse `Application` erhält via `getBean(List.class)` nun die Liste der vier Strings:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final List<String> languages = ctx.getBean(List.class);
            languages.forEach(System.out::println);
        }
    }
}
```

Die Ausgaben:

```
Pascal
Modula
Oberon
Eiffel
```

Die von Spring erzeugte Liste ist vom Typ `java.util.ArrayList` – sie ist also mutable! Wir könnten in `Application` also weitere Sprachen zur Liste hinzügen:

```
        languages.add("Java");
```

5.5 ListFactoryBean With Objects

Die im letzten Abschnitt erzeugte Liste enthielt einfache Werte (Strings). Im folgenden wollen wir eine Liste zurückliefern, welche "richtige" Objekte enthält.

Wie verwenden die bereits bekannte Klasse Language:

```
package beans;

public class Language {

    public final String name;
    public final String designer;

    public Language(String name, String designer) {
        this.name = name;
        this.designer = designer;
    }

    @Override
    public String toString() { ... }
}
```

Statt der `value`-Einträge enthält die Liste nun `bean`-Einträge, wobei in jedem Eintrag das `class`-Attribut gesetzt ist:

spring.xml

```
<beans ...>

    <bean id="languages"
            class="org.springframework.beans.factory.config.ListFactoryBean">
        <property name="sourceList">
            <list>
                <bean class="beans.Language">
                    <constructor-arg value="Pascal" />
                    <constructor-arg value="Wirth" />
                </bean>
                <bean class="beans.Language">
                    <constructor-arg value="Modula" />
                    <constructor-arg value="Wirth" />
                </bean>
                <bean class="beans.Language">
                    <constructor-arg value="Oberon" />
                    <constructor-arg value="Wirth" />
                </bean>
                <bean class="beans.Language">
                    <constructor-arg value="Eiffel" />
                    <constructor-arg value="Meyer" />
                </bean>
            </list>
```

```
        </property>
    </bean>
</beans>
```

Bei der Erzeugung jedes `Language`-Objekts wird der Kontruktor der `Language`-Klasse aufgerufen, der das jeweile Objekte initialisiert.

appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final List<Language> languages =
                ctx.getBean("languages", List.class);
            languages.forEach(System.out::println);
        }
    }
}
```

Die Ausgaben:

```
Language [Pascal, Wirth]
Language [Modula, Wirth]
Language [Oberon, Wirth]
Language [Eiffel, Meyer]
```

Man beachte, dass die Namen aller ausgegebenen Sprachen die gleiche Länge haben. Und auch die Namen der beiden Designer sind gleich lang. Zufall? Bertand Meyer hat übrigens den ehemaligen Lehrstuhl von Niklaus Wirth an der ETH Zürich übernommen…

5.6 ListFactoryBean With References

Im letzten Projekt waren die in der `Language`-Liste enthaltenen `Language`-Objekte "inline" definiert.

Wie können in einer Listen-Definition aber auch auf bereits zuvor definierte Beans zu greifen:

spring.xml

```
<beans ...>

    <bean id="pascal" class="beans.Language">
        <constructor-arg value="Pascal"/>
        <constructor-arg value="Wirth"/>
    </bean>
    <bean id="modula" class="beans.Language">
        <constructor-arg value="Modula"/>
        <constructor-arg value="Wirth"/>
    </bean>
    <bean id="oberon" class="beans.Language">
        <constructor-arg value="Oberon"/>
        <constructor-arg value="Wirth"/>
    </bean>
    <bean id="eiffel" class="beans.Language">
        <constructor-arg value="Eiffel"/>
        <constructor-arg value="Meyer"/>
    </bean>
    <bean id="languages"
        class="org.springframework.beans.factory.config.ListFactoryBean">
        <property name="sourceList">
            <list>
                <ref bean="pascal"/>
                <ref bean="modula"/>
                <ref bean="oberon"/>
                <ref bean="eiffel"/>
            </list>
        </property>
    </bean>
</beans>
```

Wir benutzen in der Liste `<ref>`-Elemente, deren `bean`-Attribut sich auf die `id` einer "öffentlichen" Bean bezieht.

5.7 MapFactoryBean

Im folgenden liefern wir mittels der Spring-eigenen `MapFactoryBean` eine `Map` zurück. Jeder Eintrag der `Map` enthält als Schlüssel einen `String` und als Wert eine Operator-`bean`. Die `MapFactoryBean` hat eine Property `soruceMap`, der wir die erzeugte `Map` zuweisen:

spring.xml

```
<beans ...>

    <bean id="languages"
        class="org.springframework.beans.factory.config.MapFactoryBean">
        <property name="sourceMap">
            <map>
                <entry key="Pascal">
                    <bean class="beans.Language">
                        <constructor-arg value="Pascal"/>
                        <constructor-arg value="Wirth"/>
                    </bean>
                </entry>
                <entry key="Eiffel">
                    <bean class="beans.Language">
                        <constructor-arg value="Eiffel"/>
                        <constructor-arg value="Meyer"/>
                    </bean>
                </entry>
            </map>
        </property>
    </bean>
</beans>
```

Der `getBean`-Lookup liefert nun eine `Map`:

appl.Application

```
package appl;

import java.util.Map;
import java.util.function.IntBinaryOperator;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final Map<String, Language> languages =
                ctx.getBean("languages", Map.class);
            languages.forEach((name, language) ->
                System.out.println(name + " ==> " + language));        }
    }
}
```

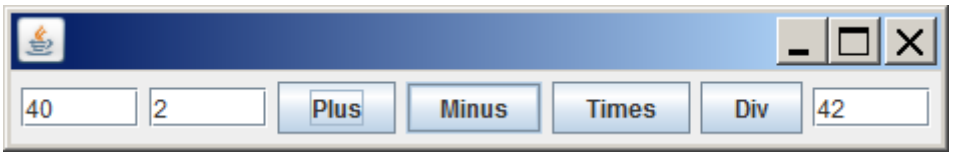
Die Ausgaben:

```
Pascal ==> Language [Pascal, Wirth]
Eiffel ==> Language [Eiffel, Meyer]
```

5.8 Example: Operators

Das folgende Beispiel ist zum Selbststudium gedacht. Es verwendet einige Operator-Klassen, die das Interface `IntBinaryOperator` implementieren.

Die Anwendung präsentiert sich wie folgt:



operators.PlusOperator

```
package operators;

import java.util.function.IntBinaryOperator;

public class PlusOperator implements IntBinaryOperator {
    @Override
    public int applyAsInt(int x, int y) {
        return x + y;
    }
}
```

Neben dieser Operator-Klasse existieren drei weitere:

operators.MinusOperator

operators.TimesOperator

operators.DivOperator

Die UI-Klasse wird eine `Map` übergeben, welche Strings (die Namen der Operatoren) auf `IntBinaryOperator`s abbildet:

appl.MathUI

```
package appl;

import java.awt.FlowLayout;
import java.util.Map;
import java.util.function.IntBinaryOperator;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;

public class MathUI extends JFrame {

    private static final long serialVersionUID = 1L;

    private final JTextField textFieldX = new JTextField(5);
    private final JTextField textFieldY = new JTextField(5);
    private final JTextField textFieldResult = new JTextField(5);

    public MathUI(Map<String, IntBinaryOperator> operators) {
        this.setLayout(new FlowLayout());
        this.add(this.textFieldX);
        this.add(this.textFieldY);
        this.addOperatorButtons(operators);
        this.add(this.textFieldResult);
        this.pack();
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    private void addOperatorButtons(
             Map<String, IntBinaryOperator> operators) {
        operators.forEach((name, op) -> {
            final JButton button = new JButton(name);
            this.add(button);
            button.addActionListener(e -> this.onCalc(op));
        });
    }

    private void onCalc(IntBinaryOperator op) {
        try {
            final int x = Integer.parseInt(this.textFieldX.getText());
            final int y = Integer.parseInt(this.textFieldY.getText());
            final int result = op.applyAsInt(x, y);
            this.textFieldResult.setText(String.valueOf(result));
        }
        catch(final NumberFormatException e) {
            this.textFieldResult.setText("Error");
        }
    }
}
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        final ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext("spring.xml");
        final Map<String, IntBinaryOperator> operators =
            ctx.getBean("operators", Map.class);
        new MathUI(operators).setVisible(true);
        Runtime.getRuntime().addShutdownHook(new Thread(() -> ctx.close()));
    }
}
```

Frage: Warum wird in der `Application` nicht der Resource-Try benutzt? Und was hat es mit `addShutDownHook` auf sich?

5.9 @Bean

Wir können Factories auch mittels Annotations definieren. Statt eines XML-Contexts kann dann ein Annotion-Context verwendet werden.

beans.ConnectionFactory

```
package beans;
// ...
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
public class ConnectionFactory {

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public Connection connectionA() throws Exception {
        return this.connection(
                "jdbc:derby:../dependencies/derby/dataA;create=true",
                "user",
                "password");
    }

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public Connection connectionB() throws Exception {
        return this.connection(
                "jdbc:derby:../dependencies/derby/dataB;create=true",
                "user",
                "password");
    }

    private Connection connection(
            String url, String user, String password) throws Exception {
        final Connection con =
            DriverManager.getConnection(url, user, password);
        con.setAutoCommit(false);
        return con;
    }
}
```

Wir haben die `ConnectionFactory` als `@Component` definiert – und die beiden Factory-Methoden mit `@Bean` (wobei via `@Scope` auch der Scope geklärt wird). Man beache, dass die Factory-Methoden sowohl als static als auch als nicht-static deklariert werden können.

Die Application benutzt `@Component`, `@PostConstruct`, `@Autowired` und `@Qualifier`:

appl.Application

```
package appl;
// ...
@Component
public class Application {

    public static void main(String[] args) {
        new AnnotationConfigApplicationContext("beans", "appl").close();
    }

    @Autowired
    @Qualifier("connectionA")
    private Connection conA1;

    @Autowired
    @Qualifier("connectionA")
    private Connection conA2;

    @Autowired
    @Qualifier("connectionB")
    private Connection conB1;

    @Autowired
    @Qualifier("connectionB")
    private Connection conB2;

    @PostConstruct
    public void run() {
        System.out.println(this.conA1);
        System.out.println(this.conA2);
        System.out.println(this.conB1);
        System.out.println(this.conB2);
    }
}
```

5.10 Expression Language

Wir können `DriverManager.getConnection` auch mittels der Expression-Language aufrufen:

spring.xml

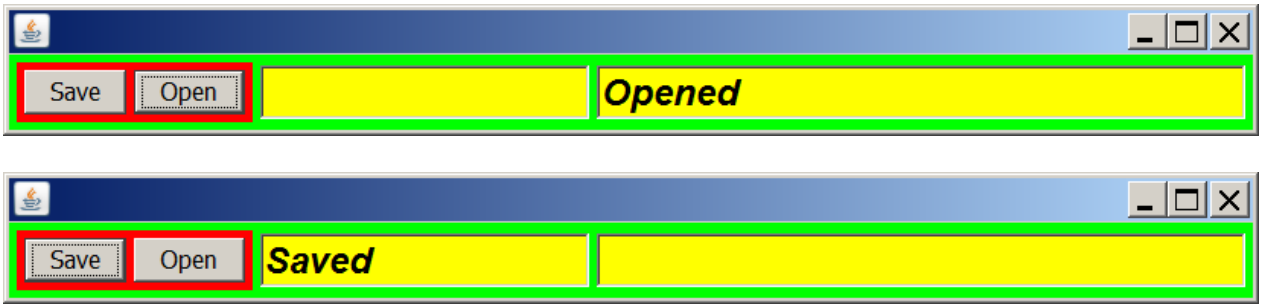
```
<beans ...>

    <bean id="connections"
            class="org.springframework.beans.factory.config.ListFactoryBean">
        <property name="sourceList">
            <list>
                <value>#{T(java.sql.DriverManager).getConnection(
                    'jdbc:derby:../dependencies/derby/dataA;create=true',
                    'user',
                    'password')}</value>
                <value>#{T(java.sql.DriverManager).getConnection(
                     'jdbc:derby:../dependencies/derby/dataB;create=true',
                     'user',
                     'password')}</value>
            </list>
        </property>
    </bean>
</beans>
```

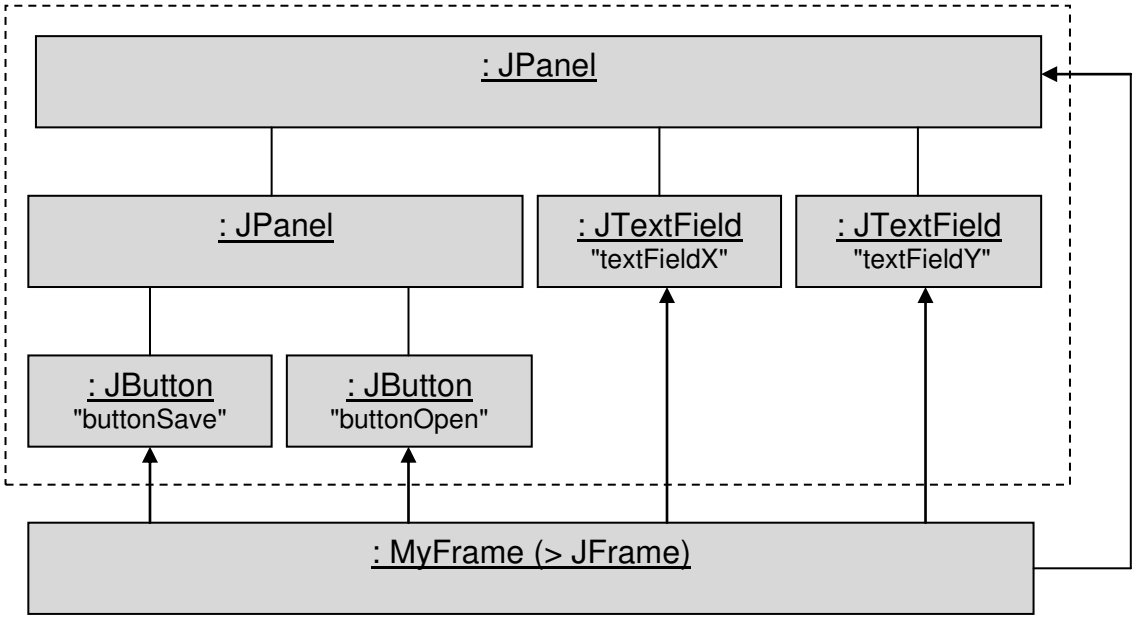
appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) {

        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final List<Connection> connections =
                ctx.getBean("connections", List.class);
            connections.forEach(System.out::println);
        }
    }
}
```

5.11 Example: GUI

Im Folgenden wird das Layout einer Swing-GUI mittels einer Spring-Konfiguration erstellt. Die Anwendung ist zum Selbststudium gedacht…





spring.xml

```
<beans ...>

    <bean id="arial" class ="java.awt.Font" >
        <constructor-arg type="java.lang.String" value="Arial"/>
        <constructor-arg type="int"
            value="#{T(java.awt.Font).BOLD + T(java.awt.Font).ITALIC}"/>
        <constructor-arg type="int" value="24"/>
    </bean>

    <bean id="panel" class="factories.JPanelFactory" scope="prototype">
        <property name="background" value="#{T(java.awt.Color).green}"/>
        <property name="components">
            <list>
                <bean class="factories.JPanelFactory">
                    <property name="background"
                        value="#{T(java.awt.Color).red}"/>
                    <property name="components">
                        <list>
                            <bean class="javax.swing.JButton">
                                <property name="name" value="buttonSave" />
                                <property name="text" value="Save" />
                            </bean>
                            <bean class="javax.swing.JButton">
                                <property name="name" value="buttonOpen" />
                                <property name="text" value="Open" />
                            </bean>
                        </list>
                    </property>

                </bean>
                <bean class="javax.swing.JTextField">
                    <constructor-arg type="int" value="10"/>
                    <property name="name" value="textFieldX" />
                    <property name="background"
                        value="#{T(java.awt.Color).yellow}"/>
                    <property name="font" ref="arial"/>
                </bean>
                <bean class="javax.swing.JTextField">
                    <constructor-arg type="int" value="20"/>
                    <property name="name" value="textFieldY" />
                    <property name="background"
                        value="#{T(java.awt.Color).yellow}"/>
                    <property name="font" ref="arial"/>
                </bean>
            </list>
        </property>
    </bean>
</beans>
```

factories.JPanelFactory

```
package factories;

import java.awt.Color;
import java.awt.Component;
import javax.swing.JPanel;
import org.springframework.beans.factory.FactoryBean;

public class JPanelFactory implements FactoryBean<JPanel> {

    private Component[] components;

    public JPanelFactory() {
        System.out.println("CTOR");
    }
    public void setComponents(Component[] components) {
        this.components = components;
    }

    private Color background;

    public void setBackground(Color background) {
        this.background = background;
    }

    @Override
    public JPanel getObject() throws Exception {
        final JPanel panel = new JPanel();
        if (this.components != null) {
            for (final Component component : this.components)
                panel.add(component);
        }
        if (this.background != null) {
            panel.setBackground(this.background);
        }
        return panel;
    }

    @Override
    public Class<?> getObjectType() {
        return JPanel.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

util.Binder

```
package util;

import java.awt.Component;
import java.awt.Container;
import java.lang.reflect.Field;
import org.springframework.context.ApplicationContext;

public class Binder {
    public static <T extends Component> T bind(ApplicationContext ctx,
            Class<T> type, Object target, String name) {
        final T component = ctx.getBean(name, type);
        traverse(target, component);
        return component;
    }

    private static void traverse(Object target, Component c) {
        try {
            if (c.getName() != null) {
                final Field field =
                    target.getClass().getDeclaredField(c.getName());
                field.setAccessible(true);
                field.set(target, c);
                System.out.println(c.getName());
            }
        }
        catch (final Exception e) {
            System.out.println(e);
        }
        if (c instanceof Container) {
            final Container container = (Container) c;
            for (int i = 0; i < container.getComponentCount(); i++) {
                traverse(target, container.getComponent(i));
            }
        }
    }
}
```

```
package appl;
// ...

import util.Binder;

public class MyFrame extends JFrame {

    private final JButton buttonSave = null;
    private final JButton buttonOpen = null;
    private final JTextField textFieldX = null;
    private final JTextField textFieldY = null;
    private final JPanel panel;

    public MyFrame(ApplicationContext ctx) {

        this.panel = Binder.bind(ctx, JPanel.class, this, "panel");
```

```
        this.add(this.panel);
        this.pack();
        this.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        this.setVisible(true);
        this.buttonSave.addActionListener(
            e -> this.textFieldX.setText("Saved"));
        this.buttonOpen.addActionListener(
            e -> this.textFieldY.setText("Opened"));
    }
}
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) throws Exception {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            new MyFrame(ctx).setLocation(100, 100);
            new MyFrame(ctx).setLocation(200, 400);
        }
    }
}
```

6 @Configuration

Im folgenden zeigen wir eine dritte Variante der Konfiguration.

Bei der Wahl dieser Variante können die von Spring verwalteten Bean-Klassen ganz ohne Spring-Annotationen auskommen. Und wir benötigen auch keine XML-Datei mehr.

Stattdessen vertrauen wir wieder mehr den Möglichkeiten der Sprache Java. Zum Zwecke der Konfiguration bauen wir eine Java-Klasse, die mit `@Configuration` annotiert ist und deren mit `@Bean` annotierte Methoden die Objekte erzeugen und zurückliefern. Die Erzeugung der Objekte geschieht mit den gewöhnlichen Java-Mitteln (also nicht mehr per Reflection – wie bei den beiden anderen Verfahren).

Hier eine Übersicht zu den folgenden Abschnitten:

- Im ersten Abschnitt präsentieren wir erneut unser `MathService`-System mit den beiden Hilfsmathematikern.
- Im Abschnitt 2 und 3 zeigen wir, wie `@Configuration`-Klassen und XML-Dateien gemischt werden können.
- Im Abschnitt 4 wird gezeigt, wie `@Configuration`-Klassen mit `@Component`-Annotationen gemischt werden können.
- Im Abschnitt 5 wird gezeigt, wie eine Anwendung mehrere `@Configuration`-Klassen nutzen kann.
- Im Abschnitt 6 demonstrieren wird das `@Qualifier`-Konzept.
- Im Abschnitt 7 geht's um `@Lazy`-Erzeugung.
- Im Abschnitt 8 geht's um `@Scope`s.

6.1 Simple

Wie erstellen auch hier zunächst wieder unser `MathService`-System (mit dem `SumService` und dem `DiffService`).

Die Implementierungs-Klassen benutzen keinerlei Spring-Annotationen. Wir können ihnen also ihre Verwendung als von Spring verwaltete Klassen nicht mehr ansehen.

Wir kommen zudem auch ohne XML-Dateien aus – und vermeiden somit die mit einer XML-Konfiguration verbundenen Nachteile.

Die Konstruktoren der Implementierungs-Klassen loggen ihren Aufruf:

beans.SumServiceImpl

```
package beans;
// ...
public class SumServiceImpl implements SumService {
    public SumServiceImpl() {
        Log.log();
    }
    // ...
}
```

beans.DiffServiceImpl

```
// ähnlich wie SumServiceImpl
```

Die Hilfsmathematiker-Referenzen der `MathServiceImpl`-Klasse werden über den Konstruktor initialisiert:

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {
    // ...
    public MathServiceImpl(SumService sumService, DiffService diffService) {
        Log.log(sumService, diffService);
        this.sumService = sumService;
        this.diffService = diffService;
    }
    // ...
}
```

Zusätzlich definieren wir nun eine Klasse, die mit der Annotation `@Configuration` ausgestattet ist:

appl.ApplConfig

```
package appl;
// ...
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ApplConfig {

    public ApplConfig() {
        Log.log();
        System.out.println(this.getClass().getName()); // !!!
    }

    @Bean(name="sum")
    public SumService sumService() {
        Log.log();
        return new SumServiceImpl();
    }

    @Bean
    public DiffService diffService() {
        Log.log();
        return new DiffServiceImpl();
    }

    @Bean
    public MathService mathService() {
        Log.log();
        return new MathServiceImpl(
            this.sumService(), this.diffService());
    }
}
```

Die Klasse enthält drei Methoden, die mit `@Bean` annotiert sind – wobei die erste `@Bean`-Annotation noch mit dem Attribut `name` ausgestattet ist.

Spring wird nun ein `ApplConfig`-Objekt erzeugen und die `@Bean`-Methoden dieser Klassen aufrufen, um die via `getBean` angeforderten Objekte zu erzeugen. Spring geht dabei von den Return-Typen der `@Bean`-Methoden aus.

- Die `sumService`-Methode wird aufgerufen werden, um eine Objekt einer das Interface `SumService` implementierenden Klasse zu erhalten.

- Die `diffService`-Methode wird aufgerufen, wenn Spring ein Objekt einer das Interface `DiffService` implementierenden Klasse benötigt.

- Und `mathService` wird dann aufgerufen, wenn ein `MathService`-kompatibles Objekt benötigt wird.

Bei der Implementierung von `mathService` ist nun zu beachten, dass die erforderliche Constructor-Injection in purem Java implementiert wird. Dabei werden die zwei anderen `@Bean`-Methoden aufgerufen, welche die beiden erforderlichen Hilfsmathematiker liefern.

Hier eine `Application`, welche zweimal den `MathService` ermittelt und jeweils einmal den `SumService` und den `DiffService`:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl")) {

            final MathService mathService1 =
                ctx.getBean(MathService.class);
            System.out.println(mathService1.sum(40, 2));
            System.out.println(mathService1.diff(80, 3));

            final MathService mathService2 =
                ctx.getBean(MathService.class);
            System.out.println(mathService2.sum(40, 2));
            System.out.println(mathService2.diff(80, 3));

            System.out.println(mathService1 == mathService2);

            final SumService sumService =
                ctx.getBean("sum", SumService.class);
            System.out.println(sumService.sum(40, 2));

            final DiffService diffService =
                ctx.getBean(DiffService.class);
            System.out.println(diffService.diff(80, 3));
        }
    }
}
```

Dem Konstruktor des `AnnotationConfigApplicationContext` wird wie gehabt der Name des Packages übergeben, welches von Spring nach annotierten Klassen gescannt werden soll (die einzige Klasse mit Spring-Annotationen, die in unserem `appl`-Paket gefunden wird, ist die Klasse `ApplConfig`).

Wie die Ausgaben zeigen, wird von jeder Klasse nur genau ein einziges Objekt erzeugt (also die entsprechende `@Bean`-Methode auch nur ein einziges Mal aufgerufen):

```
appl.ApplConfig.<init>()
appl.ApplConfig$$EnhancerBySpringCGLIB$$68e7f136
appl.ApplConfig.sumService()
beans.SumServiceImpl.<init>()
appl.ApplConfig.diffService()
beans.DiffServiceImpl.<init>()
appl.ApplConfig.mathService()
beans.MathServiceImpl.<init>(
     beans.SumServiceImpl@6b1274d2,
     beans.DiffServiceImpl@7bc1a03d)
42
77
42
77
true
42
77
```

Man beachte, dass nicht die Klasse `ApplConfig` instantiiert wird, sondern eine davon abgeleitete Klasse, die zur Laufzeit erzeugt wird.

Frage: Wie funktioniert diese Konstruktion?

6.2 Mixing With XML – Variant 1

Im folgenden zeigen wir, wie eine XML-basierte Konfiguration mit einer @Configuration-basierten Konfiguration gemischt werden kann.

In der spring.xml werden die `SumService`- und die `DiffService`-Beans vereinbart. Und zusätzlich wird auch die `ApplConfig` als einfache Bean registriert.

Und via `context:annotation-config` wird Spring angewiesen, zusätzlich Klassen daraufhin zu analysieren, ob sie mit Annotationen versehen sind (und dann entsprechende Aktionen zu unternehmen):

spring.xml

```
<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-4.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-4.0.xsd">

    <context:annotation-config />

    <bean class="appl.ApplConfig" />

    <bean id="sumService" class="beans.SumServiceImpl"/>
    <bean id="diffService" class="beans.DiffServiceImpl"/>

</beans>
```

Das `ApplConfig`-Objekt, welches aufgrund der XML-Konfiguration erzeugt wird, ist ein Objekt einer `@Configuration`-Klasse.

Dem Objekt werden via `@Autowired` zwei Beans injiziert: die aufgrund der XML-Konfiguration erzeugten `SumService`- und `DiffService`-Objekte.

Die `MathBean`, die über die `@Bean`-Methode `mathBean` erzeugt werden wird, kann dann mittels der injizierten Hilfsmathematiker erzeugt werden:

appl.ApplConfig

```
package appl;
// ...
@Configuration
public class ApplConfig {

    @Autowired
    private SumService sumService;

    @Autowired
    private DiffService diffService;

    @Bean
    public MathService mathService() {
        return new MathServiceImpl(this.sumService, this.diffService);
    }
}
```

appl.Application

Die Applikation erzeugt einen XML-Kontext – der Rest der Anwendung sieht genauso aus wie im letzten Abschnitt.

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            // ...
        }
    }
}
```

6.3 Mixing With XML – Variant 2

Aufgrund der spring.xml werden nur die Hilfsmathematiker registriert:

spring.xml

```
<beans ...>

    <bean id="sumService" class="beans.SumServiceImpl"/>
    <bean id="diffService" class="beans.DiffServiceImpl"/>

</beans>
```

appl.ApplConfig

Die Konfiguration benutzt `@ImportResource`, um die in der XML-Datei hinterlegten Bean-Definitionen nutzen zu können.

Via `@Autowired` werden die Instanzvariablen `sumService` und `diffService` initialisiert.

Bei der Erzeugung des `MathServiceImpl`-Objekts können dann eben diese Instanzvariablen genutzt werden.

```
package appl;
// ...
import org.springframework.context.annotation.ImportResource;

@Configuration
@ImportResource("spring.xml")
public class ApplConfig {

    @Autowired
    private SumService sumService;

    @Autowired
    private DiffService diffService;

    @Bean
    public MathService getMathService() {
        return new MathServiceImpl(this.sumService, this.diffService);
    }
}
```

appl.Application

Die Applikation erzeugt nun statt eines XML-Kontexts einen Annotations-Kontext – der Rest der Anwendung sieht genauso aus wie im letzten (und vorletzten) Abschnitt.

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl")) {
           // ...
       }
    }
}
```

6.4 Mixing With Annotations

Wir benutzten nun keine XML-Datei mehr.

Die Implementierungs-Klassen der Helper-Services (aber nur dieser Helper-Services!) werden mit `@Component` ausgezeichnet:

beans.SumServiceImpl

```
package beans;
// ...
@Component
public class SumServiceImpl implements SumService {
    // ...
}
```

beans.DiffServiceImpl

```
package beans;
// ...
@Component
public class DiffServiceImpl implements DiffService {
    // ...
}
```

beans.MathServiceImpl

Die `MathServiceImpl`-Klasse enthält keinerlei Annotationen (insbesondere fehlt `@Component` und `@Autowired`):

```
package beans;
// ...
public class MathServiceImpl implements MathService {

    private final SumService sumService;
    private final DiffService diffService;

    public MathServiceImpl(SumService sumService, DiffService diffService) {
        this.sumService = sumService;
        this.diffService = diffService;
    }

    // ...
}
```

Dem `ApplConfig`-Objekt werden wieder die beiden Hilfsmathematiker injiziert, aufgrund dessen dann in der `mathService`-Methode der `MathService` erzeugt werden kann:

appl.ApplConfig

```
package appl;
// ...
@Configuration
public class ApplConfig {

    @Autowired
    private SumService sumService;

    @Autowired
    private DiffService diffService;

    @Bean
    public MathService getMathService() {
        Log.log();
        return new MathServiceImpl(this.sumService, this.diffService);
    }
}
```

Die `Application` erzeugt einen `AnnotationConfigApplicationContext`:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl", "beans")) {

            // ...
        }
    }
}
```

6.5 Multiple Configurations

Keine der drei `ServiceImpl`-Klassen enthält nun irgendwelche Spring-Annotationen.

Die Konfiguration wird nun aber auf zwei Klassen verteilt.

somewhere.SumDiffConfiguration

```
package somewhere;
// ...
@Configuration
public class SumDiffConfiguration {

    @Bean
    public SumService getSumService() {
        return new SumServiceImpl();
    }

    @Bean
    public DiffService getDiffService() {
        return new DiffServiceImpl();
    }
}
```

appl.ApplConfig

```
package appl;
// ...
@Configuration
@Import(SumDiffConfiguration.class)
    // nur noetig, wenn package nicht in scan-packages enthalten
public class ApplConfig {

    @Autowired
    private SumService sumService;

    @Autowired
    private DiffService diffService;

    @Bean
    public MathService getMathService() {
        return new MathServiceImpl(this.sumService, this.diffService);
    }
}
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl")) {

            // ...
        }
    }
}
```

6.6 Qualifier

Auch im folgenden enthalten die `ServiceImpl`-Klassen keinerlei Spring-Annotationen.

Das Interface `SumService` ist aber nun in zwei Klassen implementiert:

beans.SumServiceImplSimple

```
package beans;
// ...
public class SumServiceImplSimple implements SumService {
    @Override
    public int sum(int x, int y) {
        return x + y;
    }
}
```

beans.SumServiceImplRecursive

```
package beans;
// ...
public class SumServiceImplRecursive implements SumService {
    @Override
    public int sum(int x, int y) {
        return y == 0 ? x : 1 + sum(x, y - 1);
    }
}
```

Wir registrieren diese beiden Services unter verschiedenen Namen – in der `@Configurations`-Klasse `ApplConfigSumDiff`:

appl.ApplConfigSumDiff

```
package appl;
// ...
@Configuration
public class ApplConfigSumDiff {

    @Bean(name="sumServiceSimple")
    public SumService sumServiceSimple() {
        return new SumServiceImplSimple();
    }

    @Bean(name="sumServiceRecursive")
    public SumService sumServiceRecursive() {
        return new SumServiceImplRecursive();
    }

    @Bean
    public DiffService diffService() {
```

```
        return new DiffServiceImpl();
    }
}
```

appl.ApplConfigMath

```
package appl;
// ...
@Configuration
public class ApplConfigMath {

    @Autowired
    @Qualifier("sumServiceRecursive")
    private SumService sumService;

    @Autowired
    private DiffService diffService;

    @Bean
    public MathService getMathService() {
        return new MathServiceImpl(this.sumService, this.diffService);
    }
}
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl")) {

            final MathService mathService = ctx.getBean(MathService.class);

            final SumService sumService =
                ctx.getBean("sumServiceSimple", SumService.class);

            final DiffService diffService = ctx.getBean(DiffService.class);
        }
    }
}
```

6.7 Lazy

Auch über eine `@Configuration`-Klasse lassen sich Beans lazy erzeugen. Wir lassen den `"Hello"`-String eager, den `"World"`-String aber lazy erzeugen:

appl.ApplConfig

```
package appl;
// ...
import org.springframework.context.annotation.Lazy;

@Configuration
public class ApplConfig {

    @Bean(name = "Hello")
    public String hello() {
        return "Hello";
    }

    @Bean(name = "World")
    @Lazy
    public String world() {
        return "World";
    }
}
```

Die `Application` fordert den `"Hello"`- und den `"World"`-String an:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl")) {
            System.out.println("---------------------------------");
            final String hello = ctx.getBean("Hello", String.class);
            System.out.println(hello);
            final String world = ctx.getBean("World", String.class);
            System.out.println(world);
        }
    }
}
```

Die Ausgaben zeigen, das `"Hello"` bereits im Konstruktor der `ApplicationContext`s erzeugt wird, `"World"` aber erst dann, wenn es das erste Mal angefordert wird:

```
appl.ApplConfig.<init>()
appl.ApplConfig.hello()
----------------------------
Hello
appl.ApplConfig.world()
World
```

6.8 Scopes

Der `MathService` liegt im Singleton Scope (der default) – ein `Calculator` soll im Prototype-Scope angesiedelt sein. Die Bean-Implementierungen enthalten auch hier keinerlei Annotationen:

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {
    @Override
    public int sum(int x, int y) {
        return x + y;
    }
    @Override
    public int diff(int x, int y) {
        return x - y;
    }
}
```

beans.CalculatorImpl

```
package beans;
// ...
public class CalculatorImpl implements Calculator {
    private int value;
    @Override
    public void add(int value) {
        this.value += value;
    }
    @Override
    public void subtract(int value) {
        this.value -= value;
    }
    @Override
    public int getValue() {
        return this.value;
    }
}
```

appl.ApplConfig

In der `ApplConfig` wird nur die entsprechende Factory-Methode zusätzlich mit `@Scope` ausgezeichnet:

```
package appl;
// ...
@Configuration
public class ApplConfig {

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
    public MathService getMathService() {
        Log.log();
        return new MathServiceImpl();
    }

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public Calculator getCalculator() {
        Log.log();
        return new CalculatorImpl();
    }
}
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (final AnnotationConfigApplicationContext ctx =
                 new AnnotationConfigApplicationContext("appl")) {

            final MathService mathService1 = ctx.getBean(MathService.class);
            final MathService mathService2 = ctx.getBean(MathService.class);
            System.out.println(mathService1 == mathService2);

            final Calculator calculator1 = ctx.getBean(Calculator.class);
            final Calculator calculator2 = ctx.getBean(Calculator.class);
            System.out.println(calculator1 == calculator2);

            calculator1.add(40);
            calculator2.add(80);
            calculator1.add(2);
            calculator2.subtract(3);
            System.out.println(calculator1.getValue());
            System.out.println(calculator2.getValue());
        }
    }
}
```

Die Ausgaben zeigen, dass nur ein einziger `MathService`, aber zwei `Calculator`-Objekte erzeugt wurden.

7 Der ApplicationContext – Details

Im Folgenden geht's um einige Details des `ApplicationContext`s – Details, die aber gleichwohl wichtig werden können.

- Im ersten Abschnitt geht's darum, den "Inhalt" eines `ApplicationContext`s zu inspizieren: Welche Beans enthält dieser Kontext, sind die Beans Singleton- oder Prototype-Beans etc.

- Im zweiten Abschnitt geht's darum, dass es zuweilen wichtig ist, für die Terminierung eines `ApplicationContext`s einen sog. `ShutdownHook` einzurichten.

- Im dritten Abschnitt schließlich wird gezeigt, wie Spring das Publisher-Subscriber-Konzept unterstützt – wie Spring es also erlaubt, Events zu feuern und entsprechende Listener für solche Events zu registrieren.

7.1 Methods

Dieser Abschnitt zeigt, welche Methoden auf einen `ApplicationContext` aufgerufen werden können – neben den bislang natürlich immer schon benutzten getBean-Methoden.

Wir benutzen wieder die altbekannten Interfaces:

```
ifaces.SumService
ifaces.DiffService
ifaces.MathService
ifaces.Calculator
```

Das `MathService`-Interface ist nun allerdings wie folgt definiert:

ifaces.MathService

```
package ifaces;

public interface MathService extends SumService, DiffService {
}
```

Die Implementierungs-Klassen sind allesamt mit `@Service` resp. `@Component`-Annotation ausgezeichnet (wobei den `@Service`-Annotationen jeweils ein expliziter Name mitgegeben wird):

beans.SumServiceImpl

```
package beans;
// ...
@Service("sumService")
    public class SumServiceImpl implements SumService {
// ...
}
```

beans.DiffServiceImpl

```
package beans;
// ...
@Service("diffService")
public class DiffServiceImpl implements DiffService {
    // ...
}
```

beans.MathServiceImpl

```
package beans;
// ...
@Service("mathService")
public class MathServiceImpl implements MathService {
    // ...
}
```

beans.CalculatorImpl

```
package beans;
// ...
@Component
public class CalculatorImpl implements Calculator {
    // ...
}
```

Die Application definiert eine kleine Hilfsmethode `printArray`, um Arrays von Objects auszugeben. Die `main`-Methode ruft einige `demo`-Methoden auf:

appl.Application

```
package appl;
// ...
import org.springframework.context.ConfigurableApplicationContext;

public class Application {
    public static void main(String[] args) {
        try (final ConfigurableApplicationContext ctx =
                new AnnotationConfigApplicationContext("beans")) {
            demoBeanDefinitions(ctx);
            demoContains(ctx);
            demoScopes(ctx);
            demIsTypeMatch(ctx);
            demoGetType(ctx);
            demoBeanNamesFor(ctx);
            demoIsActiveIsRunning(ctx);
            findAnnotationOnBean(ctx);
        }
    }

    private static <T> void printArray(T[] elements) {
        for (final T element : elements)
            System.out.println(element);
    }

    // es folgen die demo-Methoden...

}
```

Die folgende Methode gibt alle Schlüssel aus, unter denen im `ApplicationContext` irgendeine Bean registriert ist:

```
    private static void demoBeanDefinitions(
            ConfigurableApplicationContext ctx) {
        System.out.println(ctx.getBeanDefinitionCount());
        printArray(ctx.getBeanDefinitionNames());
    }
```

```
10
...context.annotation.internalConfigurationAnnotationProcessor
...context.annotation.internalAutowiredAnnotationProcessor
...context.annotation.internalRequiredAnnotationProcessor
...context.annotation.internalCommonAnnotationProcessor
...context.event.internalEventListenerProcessor
...context.event.internalEventListenerFactory
calculatorImpl
diffService
mathService
sumService
```

Die folgende Methode demonstriert die `containsBean`-Methode:

```
    private static void demoContains(
            ConfigurableApplicationContext ctx) {
        System.out.println(ctx.containsBean("mathService"));
        System.out.println(ctx.containsBean("diffService"));
        System.out.println(ctx.containsBean("sumService"));
        System.out.println(ctx.containsBean("calculatorImpl"));
        System.out.println(ctx.containsBean("fooBar"));
    }
```

```
true
true
true
true
false
```

Die folgende Methode demonstriert die `isSingleton`- resp. die `isPrototype`-Methode:

```
    private static void demoScopes(
             ConfigurableApplicationContext ctx) {
        System.out.println(ctx.isPrototype("mathService"));
        System.out.println(ctx.isSingleton("mathService"));
        System.out.println(ctx.isPrototype("calculatorImpl"));
        System.out.println(ctx.isSingleton("calculatorImpl"));
    }
```

```
false
true
false
true
```

Die folgende Methode demonstriert die `isTypeMatch`-Methode:

```
    private static void demIsTypeMatch(
             ConfigurableApplicationContext ctx) {
        System.out.println(ctx.isTypeMatch(
            "mathService", MathService.class));
        System.out.println(ctx.isTypeMatch(
            "mathService", SumService.class));
        System.out.println(ctx.isTypeMatch(
            "mathService", DiffService.class));
        System.out.println(ctx.isTypeMatch(
            "sumService", SumService.class));
        System.out.println(ctx.isTypeMatch(
            "sumService", MathService.class));
        System.out.println(ctx.isTypeMatch(
            "calculatorImpl", Calculator.class));
    }
```

```
true
true
true
true
false
true
```

Die folgende Methode demonstriert die `getType`-Methode:

```
    private static void demoGetType(
             ConfigurableApplicationContext ctx) {
        System.out.println(ctx.getType("mathService").getName());
        System.out.println(ctx.getType("sumService").getName());
        System.out.println(ctx.getType("calculatorImpl").getName());
    }
```

```
beans.MathServiceImpl
beans.SumServiceImpl
beans.CalculatorImpl
```

Die folgende Methode demonstriert die Methoden `getBeanNamesForType` und `getBeanNamesForAnnotation`:

```
    private static void demoBeanNamesFor(
            ConfigurableApplicationContext ctx) {
        printArray(ctx.getBeanNamesForType(MathService.class));
        System.out.println();
        printArray(ctx.getBeanNamesForType(SumService.class));
        System.out.println();
        printArray(ctx.getBeanNamesForAnnotation(Service.class));
        System.out.println();
        printArray(ctx.getBeanNamesForAnnotation(Component.class));
    }
```

```
mathService

mathService
sumService

diffService
mathService
sumService

calculatorImpl
diffService
mathService
sumService
```

Die folgende Methode demonstriert die Methoden `isActive` und `isRunning`:

```
    private static void demoIsActiveIsRunning(
            final ConfigurableApplicationContext ctx) {
        System.out.println(ctx.isActive());
        System.out.println(ctx.isRunning());
    }
```

```
true
true
```

Die folgende Methode demonstriert die Methode `findAnnotationOnBean`:

```
    private static void findAnnotationOnBean(
            final ConfigurableApplicationContext ctx) {
        final Service service1 =
            ctx.findAnnotationOnBean("sumService", Service.class);
        if (service1 != null)
            System.out.println("\"" + service1.value() + "\"" );
        final Component component =
            ctx.findAnnotationOnBean("calculatorImpl", Component.class);
        if (component != null)
            System.out.println("\"" + component.value() + "\"" );
    }
```

```
"sumService"
""
```

7.2 Shutdown Hooks

Angenommen, in der main-Methode einer Anwendung wird ein `ApplicationContext` erzeugt. Angenommen weiter, die `main`-Methode startet eine separaten Thread, der diesen Kontext benötigt. Und angenommen schließlich, dass die `main`-Methode – nachdem sie den Thread gestartet hat, sofort terminiert.

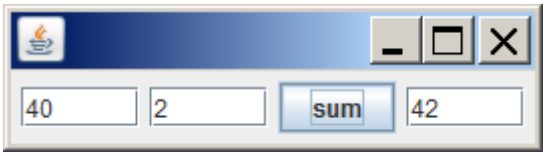
Dann ist klar, dass der `ApplicationContext` nicht in einem Resource-Try erzeugt werden kann (dann würde er ja spätestens beim Ende vom `main` (genauer: bereits beim Ende des `try`-Blocks) geschlossen werden.

Der `ApplicationContext` darf natürlich erst dann geschlossen, wenn die gesamte Anwendung terminiert – und dieses Schließen sollte auch garantiert erfolgen (egal, wo und wie die Anwendung terminiert).

Im folgenden Beispiel startet die `main`-Methode eine GUI – die in einem eigenen Thread (dem UI-Thread) weiterläuft.

In diesem Fall kann der `ApplicationContext` einen `ShutdownHook` registrieren.

Die GUI benötigt eine `MathService`:



Die `MathServiceImpl`-Klasse diagnostiziert, dass der Kontext geschlossen wird (dann nämlich wird die `@PreDestroy`-Methode aufgerufen werden):

beans.MathServiceImpl

```
package beans;
// ...
@Service
public class MathServiceImpl implements MathService {
	@PreDestroy
	private void preDestroy() {
		Log.log();
	}
	// ...
}
```

Die GUI benötigt den `MathService`, um die erforderlichen Rechnungen ausführen zu können:

appl.MathUI

```
package appl;
// ...
public class MathUI extends JFrame {

    private final JTextField textFieldX = new JTextField(5);
    private final JTextField textFieldY = new JTextField(5);
    private final JButton buttonSum = new JButton("sum");
    private final JTextField textFieldResult = new JTextField(5);

    private final ApplicationContext ctx;

    public MathUI(ApplicationContext ctx) {
        this.ctx = ctx;
        this.buildUI();
        this.buttonSum.addActionListener(e -> this.onSum());
    }

    private void onSum() {
        final MathService mathService = this.ctx.getBean(MathService.class);
        try {
            final int x = Integer.parseInt(this.textFieldX.getText());
            final int y = Integer.parseInt(this.textFieldY.getText());
            final int result = mathService.sum(x, y);
            this.textFieldResult.setText(String.valueOf(result));
        }
        catch(final NumberFormatException e) {
            this.textFieldResult.setText("Error");
        }
    }
    private void buildUI() {
        this.setLayout(new FlowLayout());
        this.add(this.textFieldX);
        this.add(this.textFieldY);
        this.add(this.buttonSum);
        this.add(this.textFieldResult);
        this.pack();
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Und die Anwendung schließlich ruft auf den `ApplicaitonContext` die Methode `registerShutdownHook` auf – und übergibt den Kontext dann an die GUI:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        final ConfigurableApplicationContext ctx =
                new AnnotationConfigApplicationContext("beans");
        ctx.registerShutdownHook();
        new MathUI(ctx).setVisible(true);
    }
}
```

Die Console-Ausgabe zeigt, dass beim Beenden der Anwendung der `ApplicationContext` terminiert:

```
beans.MathServiceImpl.preDestroy()
```

7.3 Events

Im folgenden wird gezeigt, wie Events gefeuert werden – und wie diese Events ihre Empfänger erreichen. Es geht um die Spring-Implementierung des Publisher-Subscriber-Konzepts.

Wir verwenden dieselbe GUI-Anwendung wie im letzten Abschnitt.

Wir definieren eine Klasse `MathEvent`, die von `ApplicationEvent` abgeleitet ist:

appl.MathEvent

```
package appl;

import org.springframework.context.ApplicationEvent;

public abstract class MathEvent extends ApplicationEvent {
    public MathEvent(Object source) {
        super(source);
    }
}
```

Von dieser Klasse sind wiederum die instanziierbare Klassen `SumEvent` und `ErrorEvent` abgeleitet:

appl.SumEvent

```
package appl;

public class SumEvent extends MathEvent {

    public final int x;
    public final int y;

    public SumEvent(Object source, int x, int y) {
        super(source);
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName() +
            " [" + this.x + ", " + this.y + "]";
    }
}
```

appl.ErrorEvent

```
package appl;

public class ErrorEvent extends MathEvent {
    public ErrorEvent(Object source) {
        super(source);
    }
    @Override
    public String toString() {
        return this.getClass().getSimpleName() + " []";
    }
}
```

Im Falle, dass gültige Benutzereingaben vorliegen und also der `SumService` seine Aufgabe erledigen kann, wird ein `SumEvent` gefeuert – im Falle fehlerhafter Eingaben wird ein `ErrorEvent` gefeuert.

Zum Feuern des jeweiligen Events wird die `ApplicationContext`-Methode `publishEvent` benutzt:

appl.MathUI

```
package appl;
// ...
public class MathUI extends JFrame {

    // ...

    private void onSum() {
        final MathService mathService = this.ctx.getBean(MathService.class);
        try {
            final int x = Integer.parseInt(this.textFieldX.getText());
            final int y = Integer.parseInt(this.textFieldY.getText());
            final int result = mathService.sum(x, y);
            this.textFieldResult.setText(String.valueOf(result));
            this.ctx.publishEvent(new SumEvent(this, x, y));
        }
        catch(final NumberFormatException e) {
            this.textFieldResult.setText("Error");
            this.ctx.publishEvent(new ErrorEvent(this));
        }
    }
}
```

Die `Application` startet die GUI und registriert drei Listener: einen, der auf alle `MathEvent`s horcht, einen zweiten, der nur auf `SumEvent`s und einen dritten, der nur auf `ErrorEvent`s horcht.

Die Registrierung erfolgt jeweils über die `ApplicationContext`-Methode `registerApplicationListener`, der ein `ApplicationListener` übergeben (ein Objekt einer Klasse, die das Interface `ApplicationListener` implementiert):

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        final ConfigurableApplicationContext ctx =
                new AnnotationConfigApplicationContext("beans");
        ctx.registerShutdownHook();

        new MathUI(ctx).setVisible(true);

        ctx.addApplicationListener((MathEvent event) ->
            System.out.println("MathEvent  : " + event));

        ctx.addApplicationListener((SumEvent event) ->
            System.out.println("SumEvent   : " + event.x + " " + event.y));

        ctx.addApplicationListener((ErrorEvent event) ->
            System.out.println("ErrorEvent : " + event));
    }
}
```

Die drei registrierten `ApplicationListener` sind mittels Lambdas implementiert.

Wir führen nun zwei Berechnungen aus: `40 + 2` und `40 + Hello` – also eine "richtige" und eine "falsche". Dann erscheinen auf der Console folgende Ausgaben:

```
MathEvent  : SumEvent [40, 2]
SumEvent   : 40 2

MathEvent  : ErrorEvent []
ErrorEvent : ErrorEvent []

beans.MathServiceImpl.preDestroy()
```

Der `MathEvent`-Listener wird zweimal aufgerufen (denn sowohl der `SumEvent` als auch der `ErrorEvent` sind `MathEvent`s…

8 AOP

Dieses Kapitel demonstriert, wie Spring aspektorientierte Programmierung unterstützt.

Der aspektorientierten Programmierung geht es darum, Aspekte (Logging, Transaktionen, Synchronisation etc.) von der Implementierung der eigentlichen Fachlichkeit zu trennen.

Spring benutzt zu diesem Zwecke Proxy-Mechanismen (entweder den bereits zu Anfang vorgestellten Java-eigenen Dynamic-Proxy-Mechanismus – sofern die Beans Interfaces implementieren – oder die GCLib (Code-Generation-Library) – sofern die Beans reine POJOs sind.

Wir verwenden in den folgenden Abschnitten Implementierungs-Klassen, die keinerlei Spring-Annotationen besitzen:

beans.SumServiceImpl

```
package beans;
// ...
public class SumServiceImpl implements SumService { ... }
```

beans.DiffServiceImpl

```
package beans;
// ...
public class DiffServiceImpl implements DiffService { ... }
```

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService { ... }
```

Wie demonstrieren APO jeweils in zwei Varianten (die im selben Projekt enthalten sind): mit dem XML-Kontext und mit dem Annotation-Kontext.

Im ersten Fall wird die XML-Datei herangezogen, im zweiten Fall eine `@Configation`-Klasse. (Und umgekehrt: im ersten Fall wird die `@Configation` -Klasse ignoriert, im zweiten Falle die Existenz der XML-Datei.)

Die `main`-Methode der folgenden Projekte hat stets denselben Aufbau. Sie erzeugt zunächst einen XML-Kontext und übergibt diesen an die `demo`-Methode (deren Inhalt jeweils Projekt-spezifisch ist); dann wird ein Annotation-Kontext erzeugt und dieser dann ebenfalls an die `demo`-Methode übergeben:

```
    public static void main(String[] args) {

        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            demo(ctx);
        }
        try (AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl")) {
            demo(ctx);
        }
    }

    static void demo(ApplicationContext ctx) {
        // ...
    }
```

Hier eine Übersicht zu den einzelnen Abschnitten:

- Im Abschnitt 1 zeigen wir, wie wir mit Spring ein einfaches DynamicProxy mit einem `InvocationHandler` erzeugen können.
- Im Abschnitt 2 zeigen wir, wie statt des Interfaces `InvocationHandler` das AOP-Interface `MethodInterceptor` genutzt werden kann.
- Im Abschnitt 3 geht's um Spring-Advices: `MethodBeforeAdvice`, `AfterReturningAdvice` und `ThrowsAdvice`.
- Abschnitt 4 zeigt, wie Proxies auch dann erzeugt werden können, wenn die Bean-Klassen nicht über Interfaces spezifiziert sind (dann wird die CGLib verwendet).
- In den Abschnitt 5 und 6 geht's um sog. Advisors. Ein Advisor bestimmt, für welche Methoden ein Advice herangezogen werden soll.
- Im Abschnitt 7 geht's darum, allen Beans (resp. allen via Advisor ausgewählten Beans) ein Proxy voranzustellen (`AutoProxyCreator`).
- Im letzten Abschnitts geht's um ein relativ komplexes Beispiel: um die Transaktions-Steuerung bei Datenbank-Anwendungen. Hier geht's um Services und Daos, um das `ThreadLocal`-Konzept und um eine `DelegatingConnection`...

8.1 DynamicProxy

Spring verwendet den Dynamic-Proxy-Mechanismus – kapselt ihn aber derart, dass man der Spring-Lösung diese Verwendung nicht mehr direkt ansieht.

Deshalb wird im folgenden zunächst gezeigt, wie dieser Mechanismus im Kontext einer Spring-Anwendung direkt genutzt werden kann.

Um ein Dynamic-Proxy erzeugen zu können, bauen wir eine Factory-Klasse. Dem Konstruktor der Factory-Klasse wird das zu implementierende Interface und ein `InvocationHandler` übergeben. Die `getObject`-Methode erzeugt via `Proxy.newProxyIntance` eine Klasse, die das Interface implementiert, instanziiert diese Klasse und liefert die Referenz auf das erzeugte Proxy-Objekt zurück:

util.DynamicProxyFactoryBean

```
package util;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

import org.springframework.beans.factory.FactoryBean;

public class DynamicProxyFactoryBean implements FactoryBean<Object> {

    private final InvocationHandler handler;
    private final Class<?> iface;

    public DynamicProxyFactoryBean(
            Class<?> iface, InvocationHandler handler)
            throws Exception {
        this.handler = handler;
        this.iface = iface;
    }

    @Override
    public Object getObject() throws Exception {
        return Proxy.newProxyInstance(
             Thread.currentThread().getContextClassLoader(),
             new Class<?>[] { this.iface },
             this.handler);
    }

    @Override
    public Class<?> getObjectType() {
        return this.iface;
    }
}
```

Wir implementieren das Interface `InvocationHandler` in der Klasse `TraceHandler`. Dem Konstruktor der Klasse wird das Zielobjekt übergeben. Die `invoke`-Method des Interfaces wird in Form des üblichen Dreisatzes implementiert: Pre-Invoke, Invoke und Post-Invoke (dabei werden die `before`- und `after`-Methode einer `Trace`-Klasse verwendet):

handlers.TraceHandler

```
package handlers;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

import jn.util.Trace;

public class TraceHandler implements InvocationHandler {

    private final Object target;

    public TraceHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        if (method.getDeclaringClass() == Object.class)
            return method.invoke(this.target, args);
        Trace.before(method, args);
        try {
            final Object result = method.invoke(this.target, args);
            Trace.after(method, result);
            return result;
        }
        catch (final InvocationTargetException e) {
            final Throwable t = e.getTargetException();
            Trace.after(method, t);
            throw t;
        }
        catch (final Exception e) {
            Trace.after(method, e);
            throw e;
        }
    }
}
```

In der `spring.xml` registrieren wird zunächst einen `TraceHandler` (mit einem daran angeschlossen `MathServiceImpl`-Objekts). Dann hinterlegen wird den Factory-Eintrag, wobei über `constructor-arg`s der Name des zu implementierenden Interfaces und der Handler übergeben werden:

spring.xml

```
<beans ...>

    <bean id="traceHandler" class="handlers.TraceHandler">
        <constructor-arg>
            <bean class="beans.MathServiceImpl" />
        </constructor-arg>
    </bean>

    <bean id="mathService" class="util.DynamicProxyFactoryBean">
        <constructor-arg value="ifaces.MathService"/>
        <constructor-arg ref="traceHandler"/>
    </bean>

</beans>
```

Hier die äquivalente Konfiguration mittels einer `@Configuration`-Klasse:

appl.ApplConfig

```
package appl;
// ...

@Configuration
public class ApplConfig {

    @Bean
    public MathService mathService() throws Exception {
        final TraceHandler handler =
            new TraceHandler(new MathServiceImpl());
        final DynamicProxyFactoryBean proxyFactoryBean =
            new DynamicProxyFactoryBean(MathService.class, handler);
        return (MathService)proxyFactoryBean.getObject();
    }
}
```

Hier die `demo`-Methode (die sowohl mit einem XML-Kontext als auch mit einem Annotation-Kontext aufgerufen werden kann):

appl.Application

```
package appl;
    // ...

    static void demo(ApplicationContext ctx) {
        final MathService mathService = ctx.getBean(MathService.class);
        System.out.println(mathService.getClass().getName());
        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
    }
}
```

Die Ausgaben (die fett-gedruckten Zeilen werden von der `Application` ausgegeben, die Zeilen mit normalen Font vom `TraceHandler`):

```
com.sun.proxy.$Proxy10
MathService.sum(40, 2) {
} -> 42
42
MathService.diff(80, 3) {
} -> 77
77
```

8.2 MethodInterceptor

Spring verwendet das Interface InvocationHandler nur unter der Hand. Das Spring-API benutzt stattdessen das AOP-Alliance-Interfaces `MethodInterceptor`.

Hier eine Klasse `TraceInterceptor`, welche dieses Interface implementiert:

interceptors.TraceInterceptor

```
package interceptors;

import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

import jn.util.Trace;

public class TraceInterceptor implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable {
        final Method method = invocation.getMethod();
        final Object[] args = invocation.getArguments();
        Trace.before(method, args);
        try {
            final Object result = invocation.proceed();
            Trace.after(method, result);
            return result;
        }
        catch (Exception e) {
            Trace.after(method, e);
            throw e;
        }
    }
}
```

`MethodInterceptor` spezifiziert genau eine Methode: invoke. Im Gegensatz zum Interface `InvocationHandler`, welches vom DynamicProxy-Mechanismus genutzt wird, hat das `MethodInterceptor` nicht drei, sondern nur einen einzigen Parameter: eine Parameter vom Typ `MethodInvocation`.

Mittels der `MethodInterceptor`-Methoden `getMethod` und `getArguments` können das `Method`-Objekt und die Parameter des aktuellen Aufrufs ermittelt werden. Mittels der Methode `proceed` wird die durch das `Method`-Objekt beschriebene Methode auf das eigentliche Ziel-Objekt aufgerufen.

Die obige `invoke`-Method implementiert wieder den obligatorischen "Dreisatz"

In der `spring.xml` wird der `TraceInterceptor` als gewöhnliche Bean registriert. Unter dem Namen `"mathService"` wird dann die Spring-eigene `ProxyFactoryBean`

registriert. Dieser wird (über die Property `interceptorNames`) die ID des `TraceInterceptor`s übergeben – und natürlich eine Bean der eigentlichen Implementierungs-Klasse (über `target`). Schließlich wird über `proxyInterfaces` die Liste der von der zu erzeugenden Proxy-Klasse zu implementierenden Interfaces hinterlegt:

spring.xml

```
<beans ...>

    <bean id="traceInterceptor" class="interceptors.TraceInterceptor" />

    <bean id="mathService"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="interceptorNames">
            <list>
                <value>traceInterceptor</value>
            </list>
        </property>
        <property name="target">
            <bean class="beans.MathServiceImpl" />
        </property>
        <property name="proxyInterfaces">
            <value>ifaces.MathService</value>
        </property>
    </bean>

</beans>
```

(Da das `list`-Element nur ein einziges `value`-Element besitzt, könnte die `list`-Klammer auch fehlen – die `interceptorNames`-Property hätte dann nur ein `value`-Element.)
Hier eine äquivalente Konfiguration über eine `@Configuration`-Klasse:

appl.ApplConfig

```
package appl;
// ...
import org.springframework.aop.framework.ProxyFactoryBean;

@Configuration
public class ApplConfig {

    @Bean
    public MathService mathService() {
        final ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
        proxyFactoryBean.addAdvice(new TraceInterceptor());
        proxyFactoryBean.addInterface(MathService.class);
        proxyFactoryBean.setTarget(new MathServiceImpl());
        return (MathService)proxyFactoryBean.getObject();
    }
}
```

Man beachte, dass die `mathService`-Methode nicht die `ProxyFactoryBean` selbst zurückliefert, sondern das Produkt dieser Factory-Bean (via `getObject()`).

Hier die `demo`-Methode der `Application`:

appl.Application

```
package appl;
// ...
public class Application {

    // ...

    static void demo(ApplicationContext ctx) {
        MathService mathService = ctx.getBean(MathService.class);
        System.out.println(mathService.getClass().getName());
        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
    }
}
```

Die Ausgaben:

```
com.sun.proxy.$Proxy4
MathService.sum(40, 2) {
} -> 42
42
MathService.diff(80, 3) {
} -> 77
77
```

Wie die erste Zeile der Ausgabe zeigt, verwendet auch Spring intern der Dynamic-Proxy-Mechanismus.

8.3 Before-After Advice

Statt das Interface `MethodInterceptor` zu implementieren, können wir auch die Spring-Interfaces `MethodBeforeAdvice`, `AfterReturningAdvice` und `ThrowsAdvice` implementieren (entweder aller oder nur ausgewählte):

advices.TraceBeforeAfterAdvice

```
package advices;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.ThrowsAdvice;

import jn.util.Trace;

public class TraceBeforeAfterAdvice implements
        MethodBeforeAdvice,
        AfterReturningAdvice,
        ThrowsAdvice {

    @Override
    public void before(Method method,
            Object[] parameters, Object target) {
        Trace.before(method, parameters);
    }

    @Override
    public void afterReturning(Object result, Method method,
            Object[] parameters, Object target) {
        Trace.after(method, result);
    }

    public void afterThrowing(Method method, Object[] parameters,
            Object target, RuntimeException e) throws Throwable {
        Trace.after(method, e);
    }

    public void afterThrowing(Method method, Object[] parameters,
            Object target, IllegalArgumentException e) throws Throwable {
        Trace.after(method, e);
    }
}
```

Das Interface `MethodBeforeAdvice` spezifiziert die Methode `before`; das Interface `AfterReturning` spezifiziert die Methode `after`; das Interface `ThrowsAdvice` aber ist ein reines Marker-Interface (es spezifiziert überhaupt keine Methode).

Neben den Implementierungen von `before` und `after` enthält die obige Klasse zwei `afterThrowing`-Methoden. Beiden Methoden wird ein `Method`-Objekt, ein Array der Aufrufparameter, ein Zielobjekt und eine Exception übergeben. Im ersten Falle ist der Exception-Parameter vom Typ `RuntimeException`, im zweiten Fall vom Typ `IllegalArgumentException`. Wirft nun die Zielmethode eine Exception, so wird diejenige `afterThrowing`-Methode aufgerufen, deren Parametertyp am besten zu der geworfenen Exception passt.

Die `spring.xml`-Datei sieht fast genauso aus wie im letzten Beispiel – mit dem einzigen Unterschied, dass statt eine `TraceInterceptors` nun ein `TraceBeforeAfterAdvice` registriert wird.

Die Liste der `"interceptorNames"` kann also neben `MethodInterceptor`s auch …`Advices` enthalten:

spring.xml

```
<beans ...>

    <bean id="traceBeforeAfterAdvice"
        class="advices.TraceBeforeAfterAdvice" />

    <bean id="mathService"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <list>
                <value>ifaces.MathService</value>
            </list>
        </property>
        <property name="interceptorNames">
            <list>
                <value>traceBeforeAfterAdvice</value>
            </list>
        </property>
        <property name="target">
            <bean class="beans.MathServiceImpl"/>
        </property>
    </bean>

</beans>
```

Auch die `@Configuration`-Variante sieht fast genauso aus wie im letzten Abschnitt:

appl.ApplConfig

```
package appl;
// ...
@Configuration
public class ApplConfig {

    @Bean
    public MathService mathService() {
        final ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
        proxyFactoryBean.addAdvice(new TraceBeforeAfterAdvice());
        proxyFactoryBean.addInterface(MathService.class);
        proxyFactoryBean.setTarget(new MathServiceImpl());
        return (MathService)proxyFactoryBean.getObject();
    }
}
```

Die Methoden der `MathSerivce`-Implementierung werfen nun bei negativen Parametern Exceptions. Die `sum`-Methode wirft eine `IllegalArgumentException`, die `diff`-Methode eine `RuntimeException`:

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {
    public int sum(int x, int y) {
        if (x < 0 || y < 0)
            throw new IllegalArgumentException(
                "parameters mustn't be negative");
        return x + y;
    }
    public int diff(int x, int y) {
        if (x < 0 || y < 0)
            throw new RuntimeException(
                "parameters mustn't be negative");
        return x - y;
    }
}
```

Die `demo`-Methode ruft nun die `sum`- und `diff`-Methoden sowohl mit gültigen als auch mit ungültigen Parametern auf (und fängt dabei die geworfenen Exception ab):

appl.Application

```
package appl;
// ...
public class Application {

    // ...

    private static void demo(ApplicationContext ctx) {
        final MathService mathService = ctx.getBean(MathService.class);
        System.out.println(mathService.getClass().getName());

        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));

        try {
            System.out.println(mathService.sum(-40, -2));
        }
        catch (Exception e) {
            System.out.println(e);
        }
        try {
            System.out.println(mathService.diff(-80, -3));
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Die Ausgaben:

```
com.sun.proxy.$Proxy4
MathService.sum(40, 2) {
} -> 42
42
MathService.diff(80, 3) {
} -> 77
77
MathService.sum(-40, -2) {
} => java.lang.IllegalArgumentException:
                parameters mustn't be negative
java.lang.IllegalArgumentException:
                parameters mustn't be negative
MathService.diff(-80, -3) {
} => java.lang.RuntimeException:
                parameters mustn't be negative
java.lang.RuntimeException:
                parameters mustn't be negative
```

Die Ausgabe zeigt, dass – sofern die Zielmethode eine Exception wirft – immer die am besten passende `afterThrows`-Methode unserer Advice-Klasse aufgerufen wird.

8.4 Before-After Advice with CGLib

Interfaces werden im folgenden nicht benötigt (daher gibt's auch kein `ifaces`-Package).

Es existiert also nur die Implementierungs-Klasse (die nun aber über keine `implements`-Klauseln mehr verfügt). Und diese Klasse heißt jetzt einfach `beans.MathService` (ohne `Impl`).

Auch in diesem Abschnitt wird die im letzten Abschnitt vorgestellte Klasse `TraceBeforeAfterAdvice` benutzt.

Und auch die `Application`-Klasse ist unverändert vom letzten Abschnitt übernommen worden.

Die einzigen Elemente, die geändert wurden, sind die `spring.xml` und die `ApplConfig`-Klasse. An die `target`-Property der FactoryBean wird nun `beans.MathService` übergeben; und da kein Interface mehr existiert, entfällt auch das Setzen der `proxyInterfaces`:

spring.xml

```
<beans ...>

    <bean id="traceBeforeAfterAdvice"
        class="advices.TraceBeforeAfterAdvice" />

    <bean id="mathService"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="interceptorNames">
            <list>
                <value>traceBeforeAfterAdvice</value>
            </list>
        </property>
        <property name="target">
            <bean class="beans.MathService"/>
        </property>
    </bean>

</beans>
```

appl.ApplConfig

```
package appl;
// ...
@Configuration
public class ApplConfig {

    @Bean
    public MathService mathService() {
        final ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
        proxyFactoryBean.addAdvice(new TraceBeforeAfterAdvice());
        proxyFactoryBean.setTarget(new MathService());
        return (MathService)proxyFactoryBean.getObject();
    }
}
```

Die Ausgaben zeigen, dass nun die GCLib verwendet wird. Hier der Name der genierierten Proxy-Klasse:

```
beans.MathService$$EnhancerBySpringCGLIB$$2480d72c
// ...
```

8.5 StaticMethodMatcherPointcutAdvisor

Auch in diesem Abschnitt wird wieder die Klasse `TraceBeforeAfterAdvice` benutzt.

Die Klasse `MathServiceImpl` implementiert nun wieder das Interface `MathService`.

Der Trace-Advice soll nun aber nur dann aktiv werden, wenn eine Methode aufgerufen wird, deren Name mit `"d"` beginnt.

Wir benötigen daher einen "Advisor", der bestimmt, wann der "Advice" herangezogen werden soll.

Wir definieren eine Klasse `MathServicePointcutAdvisor`, die von der Spring-Klasse `StaticMethodMatcherPointcutAdvisor` abgeleitet ist – und überschreiben dabei die `matches`-Methode:

pointcutAdvisors.MathServicePointcutAdvisor

```
package pointcutAdvisors;

import java.lang.reflect.Method;
import org.springframework.aop.support.StaticMethodMatcherPointcutAdvisor;

public class MathServicePointcutAdvisor
        extends StaticMethodMatcherPointcutAdvisor {

    @Override
    public boolean matches(Method m, Class<?> targetClass) {
        return m.getName().startsWith("d");
    }
}
```

Über die Spring-Konfiguration wird neben dem `TraceBeforeAfterAdvice` ein `MathServicePointcutAdvisor` erzeugt, dem der zuvor erzeugt Advice übergeben wird. Und satt des Advices wird in der Liste der `interceptorNames` nun der Advisor aufgeführt:

spring.xml

```
<beans ...>

    <bean id="traceBeforeAfterAdvice"
        class="advices.TraceBeforeAfterAdvice" />

    <bean id="mathServicePointcutAdvisor"
            class="pointcutAdvisors.MathServicePointcutAdvisor">
        <property name="advice" ref="traceBeforeAfterAdvice"/>
    </bean>
```

```
    <bean id="mathService"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <list>
                <value>ifaces.MathService</value>
            </list>
        </property>
        <property name="target">
            <bean class="beans.MathServiceImpl" />
        </property>
        <property name="interceptorNames">
            <list>
                <value>mathServicePointcutAdvisor</value>
            </list>
        </property>
    </bean>

</beans>
```

In der `ApplConfig` wird nun auf die `ProxyFactoryBean` statt der Methode `addAdvice` nun die Methode `addAdvisor` aufgerufen:

appl.ApplConfig

```
package appl;
// ...
import pointcutAdvisors.MathServicePointcutAdvisor;

@Configuration
public class ApplConfig {

    @Bean
    public MathService mathService() {
        final MathServicePointcutAdvisor advisor =
            new MathServicePointcutAdvisor();
        advisor.setAdvice(new TraceBeforeAfterAdvice());

        final ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
        proxyFactoryBean.addInterface(MathService.class);
        proxyFactoryBean.setTarget(new MathServiceImpl());
        proxyFactoryBean.addAdvisor(advisor);
        return (MathService)proxyFactoryBean.getObject();
    }
}
```

Die `Application` ruft auf den `MathService` die Methoden `sum` und `diff` auf:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {

    // ...

    static void demo(ApplicationContext ctx) {
        MathService mathService = ctx.getBean(MathService.class);
        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
    }
}
```

Wie man sieht, wird nur aber nur der Aufruf von `diff` diagnostiziert, nicht aber der von `sum`:

```
42
MathService.diff(80, 3) {
} -> 77
77
```

8.6 NameMatchMethodPointcutAdvisor

Statt einer eigenen Advisor-Klasse können wir auch vorgefertigte instanziierbare Advisor-Klassen nutzen.

Wir benutzen im folgenden die Spring-Klasse `NameMatchMethodPointcutAdvisor`. Neben dem Advice übergeben wird ein eine `mappedNames`-Liste:

spring.xml

```
<beans ...>

    <bean id="traceBeforeAfterAdvice"
        class="advices.TraceBeforeAfterAdvice" />

    <bean id="mathServiceTracePointcutAdvisor"
        class="....aop.support.NameMatchMethodPointcutAdvisor">
        <property name="advice" ref="traceBeforeAfterAdvice" />
        <property name="mappedNames">
            <list>
                <value>s*</value>
            </list>
        </property>
    </bean>

    <bean id="mathService"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces" value="ifaces.MathService" />
        <property name="target">
            <bean class="beans.MathServiceImpl" />
        </property>
        <property name="interceptorNames">
            <list>
                <value>mathServiceTracePointcutAdvisor</value>
            </list>
        </property>
    </bean>

</beans>
```

In der `AppConfig`-Klasse benutzen wir die Methode `addMethodName`:

appl.ApplConfig

```
package appl;
// ...
import org.springframework.aop.support.NameMatchMethodPointcutAdvisor;

@Configuration
public class ApplConfig {

    @Bean
    public MathService mathService() {
        Log.log();
        final NameMatchMethodPointcutAdvisor advisor =
            new NameMatchMethodPointcutAdvisor();
        advisor.setAdvice(new TraceBeforeAfterAdvice());
        advisor.addMethodName("s*");

        final ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
        proxyFactoryBean.addInterface(MathService.class);
        proxyFactoryBean.setTarget(new MathServiceImpl());
        proxyFactoryBean.addAdvisor(advisor);
        return (MathService)proxyFactoryBean.getObject();
    }
}
```

Wir verwenden die gleiche `Application`-Klasse wie im letzten Abschnitt.

Die Ausgaben sind dieselben wie im letzten Abschnitt:

```
MathService.sum(40, 2) {
} -> 42
42
77
```

8.7 AutoProxyCreator

Im folgenden benutzen wir wieder unser "komplexes" System – bestehend aus dem Chef- und seien beiden Hilfsmathematikern. Wir wollen sowohl die Aufrufe der Methoden des Chefmathematikers als auch die Aufrufe seiner Hilfsmathematiker tracen:

beans.SumServiceImpl

```
package beans;
// ...
public class SumServiceImpl implements SumService { ... }
```

beans.DiffServiceImpl

```
package beans;
// ...
public class DiffServiceImpl implements DiffService { ... }
```

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {
    // ...
    public MathServiceImpl(SumService sumService, DiffService diffService) {
        this.sumService = sumService;
        this.diffService = diffService;
    }
    // ...
}
```

Wir benötigen nun drei Proxies: eine für den Chef und jeweils eine weitere für den jeweiligen Hilfsmathematiker. Alle Proxies sollen dieselbe Funktionalität implementieren (sie sollen mit einem `TraceBeforeAfterAdvice` ausgestattet sein).

Wie können dann die Klasse `DefaultPointcutAdvisor` benutzen, welcher der Advice und ein `pointcut` übergeben wird. Als `pointcut` übergeben wird ein Objekt der Klasse `JdkRegexpMethodPointcut`, der ein `pattern` übergeben wird.

Zusätzlich wird eine Bean des Typs `DefaultAdvisorAutoProxyCreator` registriert. Diese Klasse wird dafür sorgen, dass allen Beans jeweils ein Proxy vorgeschaltet wird:

spring.xml

```
<beans ...>

    <bean
        class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <property name="advice">
            <bean class="advices.TraceBeforeAfterAdvice" />
        </property>
        <property name="pointcut">
            <bean class="....aop.support.JdkRegexpMethodPointcut">
                <property name="pattern" value="ifaces.*" />
            </bean>
        </property>
    </bean>

    <bean
        class="....aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" />

    <bean id="mathService" class="beans.MathServiceImpl">
        <constructor-arg>
            <bean class="beans.SumServiceImpl" />
        </constructor-arg>
        <constructor-arg>
            <bean class="beans.DiffServiceImpl" />
        </constructor-arg>
    </bean>

</beans>
```

Hier die äquivalente `@Configuration`-Variante:

appl.ApplConfig

```
package appl;

import ....aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.JdkRegexpMethodPointcut;

@Configuration
public class ApplConfig {

    @Bean
    public SumService sumService() {
        return new SumServiceImpl();
    }

    @Bean
    public DiffService diffService() {
        return new DiffServiceImpl();
    }

    @Bean
```

```
    public MathService mathService() {
        final SumService sumService = this.sumService();
        final DiffService diffService = this.diffService();
        return new MathServiceImpl(sumService, diffService);
    }

    @Bean
    public DefaultPointcutAdvisor advisor() {
        final JdkRegexpMethodPointcut pointcut =
            new JdkRegexpMethodPointcut();
        pointcut.setPattern("ifaces.*");
        final DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor();
        advisor.setAdvice(new TraceBeforeAfterAdvice());
        advisor.setPointcut(pointcut);
        return advisor;
    }

    @Bean
    public DefaultAdvisorAutoProxyCreator autoProxyCreator() {
        return new DefaultAdvisorAutoProxyCreator();
    }
}
```

appl.Application

```
package appl;
// ...
public class Application {

    // ...

    static void demo(ApplicationContext ctx) {
        MathService mathService = ctx.getBean(MathService.class);
        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
    }
}
```

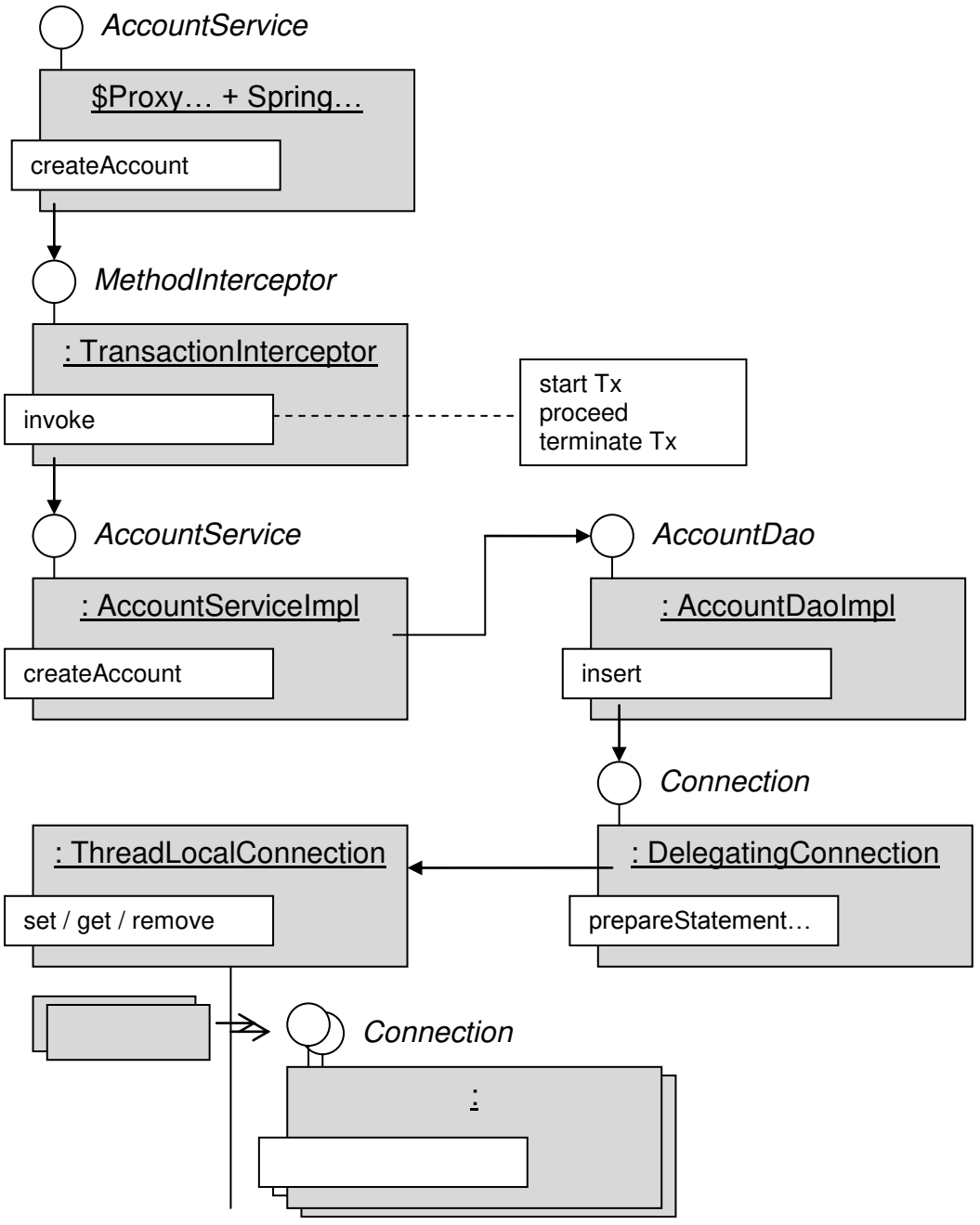
Die Ausgaben:

```
MathService.sum(40, 2) {
      SumService.sum(40, 2) {
      } -> 42
} -> 42
42
MathService.diff(80, 3) {
      DiffService.diff(80, 3) {
      } -> 77
} -> 77
77
```

8.8 Bank-Example

Im folgenden wird das Proxy-Konzept zur Steuerung von Transaktionen verwendet. Wir bauen eine kleine Bank-Anwendung, die auch in einem Multithreading-System verwendet werden kann.

Hier zunächst ein Objekt-Diagramm:



Der Aufruf von `getBean(MathService.class)` liefert ein Proxy (und einen Spring-Adapter) zurück, das (der) an einen `TransactionInterceptor` delegiert.

Die `invoke`-Methode des `TransactionInterceptor`s erzeugt eine neue `Connection`, trägt diese in einen `ThreadLocal` ein (in ein `ThreadLocalConnections`-Objekt) und delegiert dann an weiter an die entsprechende Methode des `AccountServiceImpl`-Objekts. Nach Rückkehr aus dieser Methode wird die Transaktion terminiert, die `Connection` geschlossen und aus dem `ThreadLocalConnections`-Objekt ausgetragen. Dem `AccountServiceImpl` wird ein `AccountDaoImpl` injiziert. Die `createAccount`-Methode des Services (in welcher die Fachlichkeit implementiert ist) kann somit an die `insert`-Methode des Daos delegieren (letztere enthält nur die Persistenzlogik).

Dem `AccountDaoImpl` wird eine `Connection` injiziert – eine `DelegatingConnection`. Die Methoden dieses `DelegatingConnection`s-Objekts (z.B. `prepareStatement`) delegieren an die entsprechenden Methode desjenigen "realen" `Connection`-Objekts, das mit dem aktuellen Thread assoziiert ist (also an eine der im `ThreadLocalConnection`s-Objekt enthaltenen Connections.

Wir verwenden folgende `db.properties`-Datei:

db.properties

```
db.driver           org.apache.derby.jdbc.EmbeddedDriver
db.url              jdbc:derby:../dependencies/derby/data
db.user             user
db.password         password
db.schema           USER
```

Die Datenbank besitzt nur eine einzige Tabelle:

create.sql

```
create table account (
    number integer,
    balance integer,
    primary key (number)
)
```

Das Dao-Interface spezifiziert eine `insert`-Methode:

ifaces.AccountDao

```
package ifaces;

public interface AccountDao {
    public abstract void insert(int number, int balance);
}
```

Das Service-Interface spezifiziert eine `createAccount`-Methode:

ifaces.AccountService

```
package ifaces;

public interface AccountService {
    public abstract void createAccount(int number);
}
```

`AccountDaoImpl` implementiert `AccountDao`. Via Constructor-Injection wird einem `AccountDaoImpl`-Objekt eine `Connection` (eine `DelegatingConnection`!) übergeben:

beans.AccountDaoImpl

```
package beans;
// ...
public class AccountDaoImpl implements AccountDao {

    private final Connection con;

    public AccountDaoImpl(Connection con) {
        this.con = con;
    }

    public void insert(int number, int balance) {
        System.out.println("\t>> insert() " +
            Thread.currentThread().getId());
        try {
            Thread.sleep(1000);
            final PreparedStatement ps = con.prepareStatement(
                "insert into account (number, balance) values (?, ?)");
            ps.setInt(1, number);
            ps.setInt(2, balance);
            ps.execute();
            ps.close();
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
        System.out.println("\t<< insert() " +
            Thread.currentThread().getId());
    }
}
```

Man beachte, dass die obige `insert`-Methode "harte Arbeit" simuliert (`Thread.sleep(1000)`) – dieser sleep wird zur Demonstration des Multithreadings verwendet werden (siehe weiter unten).

`AccountServiceImpl` implementiert `AccountService`. Via Constructor-Injection wird einem `AccountServiceImpl`-Objekt die Referenz auf ein `AccountDao`-Objekt übergeben:

beans.AccountServiceImpl

```
package beans;

import ifaces.AccountDao;
import ifaces.AccountService;

public class AccountServiceImpl implements AccountService {

    private AccountDao dao;

    public AccountServiceImpl(AccountDao dao) {
        this.dao = dao;
    }

    public void createAccount(int number) {
        System.out.println(">> createAccount() thread = " +
            Thread.currentThread().getId());
        // Fachlogik ...
        this.dao.insert(number, 0);
        System.out.println("<< createAccount() thread = " +
            Thread.currentThread().getId());
    }
}
```

Im `util`-Package ist die Klasse `ThreadLocalConnections` implementiert, welche einen `ThreadLocal<Connection>` besitzt. Mittels der `set`-Methode kann eine `Connection` in den `ThreadLocal` eingetragen werden (mit dem aktuellen Thread assoziiert werden); mittels `get` wird diejenige `Connection` ermittelt, die mit dem aktuellen Thread assoziiert ist; und mittels `remove` kann der Eintrag für den aktuellen Thread entfernt werden.

Die Methode `set` und `remove` werden vom `TransactionInterceptor` aufgerufen; die `get`-Methode vom `DelegatingConnection`-Objekt.

util.ThreadLocalConnections

```
package util;

import java.sql.Connection;

public class ThreadLocalConnections {

    private final ThreadLocal<Connection> connections =
        new ThreadLocal<Connection>();

    public void set(Connection con) {
        if (this.connections.get() != null)
```

```
            throw new RuntimeException(
                "Connection for current thread already set");
        this.connections.set(con);
    }
    public Connection get() {
        final Connection con = this.connections.get();
        if (con == null)
            throw new RuntimeException(
                "Connection for current thread not set");
        return con;
    }
    public void remove() {
        final Connection con = this.connections.get();
        if (con == null)
            throw new RuntimeException(
                "Connection for current thread not set");
        this.connections.remove();
    }
    public boolean isAvailable() {
        return this.connections.get() != null;
    }
}
```

Die `util`-Klasse `DelegatingConnection` implementiert das `Connection`-Interface. Alle Methoden werden an die entsprechenden Methoden desjenigen realen `Connection`-Objekts delegiert, welches mit dem aktuellen Thread assoziiert ist.

Via Constructor-Injection wird dem `DelegatingConnection`-Objekt die Referenz auf das `ThreadLocalDonnection`-Objekt übergeben:

util.DelegatingConnection

```
package util;
// ...
import java.sql.Connection;
import java.sql.PreparedStatement;

public class DelegatingConnection implements Connection {

    private final ThreadLocalConnections connections;

    public DelegatingConnection(ThreadLocalConnections connections) {
        this.connections = connections;
    }

    @Override
    public void close() throws SQLException {
        // do nothing...
    }

    @Override
    public void commit() throws SQLException {
        throw new RuntimeException("close is forbidden");
    }
```

```
    @Override
    public void rollback() throws SQLException {
        throw new RuntimeException("close is forbidden");
    }

    @Override
    public void setAutoCommit(boolean autoCommit) throws SQLException {
        this.connections.get().setAutoCommit(autoCommit);
    }


    @Override
    public PreparedStatement prepareStatement(String sql)
            throws SQLException {
        System.out.println("\t\t" + Integer.toHexString(
            System.identityHashCode(this.connections.get())));
        return this.connections.get().prepareStatement(sql);
    }

    // ETC ...

}
```

Dem `TransactionInterceptor` wird ein `Properties`-Objekt (welches die JDBC-Verbindungsdaten enthält) und das `ThreadLocalConnections`-Objekt übergeben.

Die `inovke`-Methode erzeugt im Pre-Invoke-Schritt eine neue `Connection`, setzt Autocommit auf `false` und trägt die `Connection` im `ThreadLocalConnections`-Objekt ein.

Im Post-Invoke-Schritt wird die Transaktion terminiert (`commit` resp. `rollback`), die `Connection` geschlossen und aus dem `ThreadLocalConnections`-Objekt entfernt:

interceptors.TransactionInterceptor

```
package interceptors;

import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import util.ThreadLocalConnections;

public class TransactionInterceptor implements MethodInterceptor {

    private final Properties properties;
    private final ThreadLocalConnections connections;

    public TransactionInterceptor(
            Properties properties, ThreadLocalConnections connections) {
        this.properties = properties;
```

```
        this.connections = connections;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        try {
            final String url = properties.getProperty("db.url");
            final String user = properties.getProperty("db.user");
            final String password = properties.getProperty("db.password");
            final Connection con =
                DriverManager.getConnection(url, user, password);
            con.setAutoCommit(false);
            this.connections.set(con);
            Object result = invocation.proceed();
            this.connections.get().commit();
            return result;
        }
        catch (Exception e) {
            if (this.connections.isAvailable())
                this.connections.get().rollback();
            throw e;
        }
        finally {
            if (this.connections.isAvailable()) {
                final Connection con = this.connections.get();
                this.connections.remove();
                con.close();
            }
        }
    }
}
```

Hier die Spring-Konfiguration (ohne weitere Erklärungen…):

spring.xml

```
<beans ...>

    <bean id="dbProperties"
            class="....beans.factory.config.PropertiesFactoryBean">
        <property name="locations" value="classpath*:db.properties"/>
    </bean>

    <bean id="threadLocalConnections" class="util.ThreadLocalConnections" />

    <bean id="transactionInterceptor"
            class="interceptors.TransactionInterceptor">
        <constructor-arg ref="dbProperties" />
        <constructor-arg ref="threadLocalConnections" />
    </bean>

    <bean id="delegatingConnection" class="util.DelegatingConnection">
        <constructor-arg ref="threadLocalConnections"/>
    </bean>

    <bean id="accountService"
```

```
            class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="interceptorNames">
            <list>
                <value>transactionInterceptor</value>
            </list>
        </property>
        <property name="target">
            <bean id="accountServiceImpl" class="beans.AccountServiceImpl">
                <constructor-arg>
                    <bean class="beans.AccountDaoImpl">
                        <constructor-arg ref="delegatingConnection" />
                    </bean>
                </constructor-arg>
            </bean>
        </property>
    </bean>

</beans>
```

Eine äquivalente `@Configuration`-Klasse:

appl.ApplConfig

```
package appl;
// ...
@Configuration
public class ApplConfig {

    @Bean
    protected Properties properties() {
        try {
            final Properties properties = new Properties();
            properties.load(
                ClassLoader.getSystemResourceAsStream("db.properties"));
            return properties;
        }
        catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Bean
    protected ThreadLocalConnections threadLocalConnections() {
        return new ThreadLocalConnections();
    }

    @Bean
    protected TransactionInterceptor transactionInterceptor() {
        return new TransactionInterceptor(
            this.properties(),
            this.threadLocalConnections());
    }

    @Bean
    protected DelegatingConnection delegatingConnection() {
```

```
        return new DelegatingConnection(threadLocalConnections());
    }

    @Bean
    protected AccountDao accountDao() {
        return new AccountDaoImpl(this.delegatingConnection());
    }

    @Bean
    public AccountService accountService() {
        final ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
        proxyFactoryBean.addAdvice(this.transactionInterceptor());
        proxyFactoryBean.addInterface(AccountService.class);
        proxyFactoryBean.setTarget(
            new AccountServiceImpl(
                this.accountDao()));
        return (AccountService)proxyFactoryBean.getObject();
    }
}
```

Und hier schließlich die komplette Anwendung, welche den gleichzeitigen Zugriff zweier Threads demonstriert:

appl.Application

```
package appl;
// ...
import db.util.appl.Db;

public class Application {

    public static void main(String[] args) throws Exception {
        Db.aroundAppl();

        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            demo(ctx, 4711, 4712);
        }
        try (AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl")) {
            demo(ctx, 4713, 4714);
        }
    }

    static void demo(ApplicationContext ctx, int nr1, int nr2)
            throws Exception {
        final AccountService service = ctx.getBean(AccountService.class);

        final Thread t1 = new Thread(() -> service.createAccount(nr1));
        final Thread t2 = new Thread(() -> service.createAccount(nr2));

        t1.start();
        Thread.sleep(500);
        t2.start();
```

```
        t1.join();
        t2.join();
    }
}
```

Die Ausgaben zeigen, dass beide Threads mit einer jeweils eigenen Connection operieren:

```
>> createAccount() thread = 17
     >> insert() 17
>> createAccount() thread = 18
     >> insert() 18
          2377a40a
     << insert() 17
<< createAccount() thread = 17
          413d644
     << insert() 18
<< createAccount() thread = 18

>> createAccount() thread = 19
     >> insert() 19
>> createAccount() thread = 20
     >> insert() 20
          21667d94
     << insert() 19
<< createAccount() thread = 19
          10e55fa6
     << insert() 20
<< createAccount() thread = 20
```

Der abschließende Zustand der Datenbank:

```
ACCOUNT
NUMBER BALANCE
---------------
4711   0
4712   0
4713   0
4714   0
---------------
```

9 AspectJ

In allen Beispielen dieses Kapitels wird das `MathService`-Interface und folgende einfache Implementierung verwendet:

```
package beans;
// ...
public class MathServiceImpl implements MathService {
    public int sum(int x, int y) {
        return x + y;
    }
    public int diff(int x, int y) {
        return x - y;
    }
}
```

Weiterhin benutzen wir eine `JoinPointTrace`-Klasse aus dem `shared`-Projekt. Der `trace`-Methode dieser Klasse wird u.a. ein `JoinPoint` übergeben (`JoinPoint` ist ein Interface von AspectJ):

```
package jn.util;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.Signature;

public class JoinPointTrace {
    public static void trace(String text,
            JoinPoint joinPoint, Object result) {

        final Signature signature = joinPoint.getSignature();
        System.out.print(text +
            signature.getDeclaringTypeName() + "." +
            signature.getName() + "(");
        final Object[] args = joinPoint.getArgs();
        if (args != null)
            for (int i = 0; i < args.length; i++) {
                if (i > 0)
                    System.out.print(", ");
                System.out.print(args[i]);
            }
        System.out.print(")");
        if (result != null)
            System.out.print(" --> " + result);
        System.out.println();
    }
}
```

Spring benutzt von AspectJ nur die Syntax der sog. Joinpoint-Expressions. Spring benutzt also nicht(!) das ByteCode-Enhancement (Weaven) von AspectJ. Spring wird also wieder Proxies erzeugen (DynamicProxies oder CGLib-Proxies).

Hier eine Übersicht zu den Abschnitten dieses Kapitels:

- Im ersten Abschnitt werden die AspectJ-Annotationen `@Aspect`, `@Before` und `@After` vorgestellt.
- Im zweiten Abschnitt geht's um die `@Pointcut`-Annotation.
- Im dritten Abschnitt geht's um die Annotationen `@AfterReturning` und `@AfterThrowing`.
- Im vierten Abschnitt geht's um die `@Around`-Annotation.
- Im Abschnitt fünf und sechs werden die sog. Pointcut-Designators näher untersucht.

In den folgenden Abschnitten wird i.d.R. nur der Quellcode und die Ausgaben der Demo-Programme vorgestellt – auf eine nähere Erläuterung wir verzichtet. Den Lesern / den Leserinnen sei empfohlen, mit den Beispielen einfach ein wenig zu spielen…

9.1 @Advice

advices.TraceAdvices

```
package advices;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

import jn.util.JoinPointTrace;

@Aspect
public class TraceAdvices {

    @Before("within(beans..*)")
    public void before(JoinPoint joinPoint) {
        JoinPointTrace.trace(">> ", joinPoint, null);
    }

    @After("within(beans..*)")
    public void after(JoinPoint joinPoint) {
        JoinPointTrace.trace("<< ", joinPoint, null);
    }
}
```

spring.xml

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

    <aop:aspectj-autoproxy/>

    <bean id="traceAdvices" class="advices.TraceAdvices" />

    <bean id="mathService" class="beans.MathServiceImpl" />

</beans>
```

appl.Application

```
package appl;
// ...
import org.springframework.aop.Advisor;
import org.springframework.aop.SpringProxy;
import org.springframework.aop.framework.Advised;

public class Application {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            final MathService mathService = ctx.getBean(MathService.class);
            System.out.println(mathService.getClass().getName());

            System.out.println(mathService.sum(40, 2));
            System.out.println(mathService.diff(80, 3));

            final Class<?>[] ifaces = mathService.getClass().getInterfaces();
            System.out.println("interfaces");
            for (final Class<?> iface : ifaces)
                System.out.println("\t" + iface.getName());
        }
    }
}
```

```
com.sun.proxy.$Proxy6

>>  iface.MathService.sum(40, 2)
<<  iface.MathService.sum(40, 2)
42
>>  iface.MathService.diff(80, 3)
<<  iface.MathService.diff(80, 3)
77
```

Eine Annotations-basierte Variante

Die `@Adivce`-Klasse wird zusätzlich mit `@Component` annotiert:

```
package advices;
// ...
@Aspect
@Component
public class TraceAdvices {
    // ...
}
```

Und auch `MathServiceImpl` wird mit `@Component` ausgezeichnet:

```
package beans;
// ...
@Component
public class MathServiceImpl implements MathService {
    // ...
}
```

Hier die Spring-Konfiguration:

```
<beans ...>
    <context:component-scan base-package="advices"/>
    <context:component-scan base-package="beans"/>
    <aop:aspectj-autoproxy/>
</beans>
```

9.2 Pointcut

pointcuts.Pointcuts

```
package pointcuts;

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

@Aspect
public class Pointcuts {
    @Pointcut("within(beans..*)")
    public void withinBeans() {   // "Pointcut signature"
    }
}
```

advices.TraceAdvice

```
package advices;
// ...

@Aspect
public class TraceAdvices {

    @Before("pointcuts.Pointcuts.withinBeans()")
    public void before(JoinPoint joinPoint) {
        JoinPointTrace.trace(">> ", joinPoint, null);
    }

    @After("pointcuts.Pointcuts.withinBeans()")
    public void after(JoinPoint joinPoint) {
        JoinPointTrace.trace("<< ", joinPoint, null);
    }
}
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            final MathService mathService = ctx.getBean(MathService.class);
            System.out.println(mathService.sum(40, 2));
            System.out.println(mathService.diff(44, 2));
        }
    }
}
```

Die Ausgaben:

```
>> iface.MathService.sum(40, 2)
<< iface.MathService.sum(40, 2) --> 42
42
>> iface.MathService.diff(80, 3)
<< iface.MathService.diff(80, 3) --> 77
77
```

9.3 AfterReturning / AfterThrowing

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {
    final String MESSAGE = "only positive parameters allowed";
    @Override
    public int sum(int x, int y) {
        if (x <= 0 || y <= 0)
            throw new RuntimeException(this.MESSAGE);
        return x + y;
    }
    @Override
    public int diff(int x, int y) {
        if (x <= 0 || y <= 0)
            throw new RuntimeException(this.MESSAGE);
        return x - y;
    }
}
```

adivices.TraceAdvice

```
package advices;
// ...
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class TraceAdvices {

    @Before("within(beans..*)")
    public void before(JoinPoint joinPoint) {
        JoinPointTrace.trace(">> ", joinPoint, null);
    }

    @AfterReturning(pointcut="within(beans..*)", returning="result")
    public void afterReturning(JoinPoint joinPoint, Object result) {
        JoinPointTrace.trace("<< ", joinPoint, result);
    }

    @AfterThrowing(pointcut="within(beans..*)", throwing="exception")
    public void afterThrowing(JoinPoint joinPoint, Object exception) {
        JoinPointTrace.trace("<< ", joinPoint, exception);
    }
}
```

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx = new
                ClassPathXmlApplicationContext("spring.xml")) {
            final MathService mathService =
                ctx.getBean(MathService.class);

            System.out.println(mathService.sum(40, 2));
            try {
                System.out.println(mathService.diff(-80, -3));
            }
            catch (final Exception e) {
                System.out.println(e);
            }

        }
    }
}
```

Die Ausgaben:

```
>> iface.MathService.sum(40, 2)
<< iface.MathService.sum(40, 2) --> 42
42
>> iface.MathService.diff(-80, -3)
<< iface.MathService.diff(-80, -3) --> java.lang.RuntimeException:
                                    only positive parameters allowed
java.lang.RuntimeException: only positive parameters allowed
```

9.4 Around

```
package advices;
// ...
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class TraceAdvices {

    @Around("within(beans..*)")
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
        JoinPointTrace.trace(">> ", joinPoint, null);
        final Object result = joinPoint.proceed();
        JoinPointTrace.trace("<< ", joinPoint, result);
        return result;
    }
}
```

9.5 Execution Pointcut-Designator

```
package advices;
// ...
@Aspect
public class TraceAdvices {

    @Around("execution(public int sum(int,int))")
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
        JoinPointTrace.trace(">> ", joinPoint, null);
        final Object result = joinPoint.proceed();
        JoinPointTrace.trace("<< ", joinPoint, result);
        return result;
    }
}
```

Hier wird nur der Aufruf solcher Methoden geloggt werden, die "`sum`" heißen, zwei `int`-Parameter und den Return-Typ `int` besitzen.

Bei der folgenden solchen Joinpoint-Expression würden alle Aufrufe aller öffentlichen Methoden geloggt werden:

```
    @Around("execution(public * *(..))")
```

9.6 Annotation Pointcut-Designator

annotations.Logged

```
package annotations;
// ...
@Retention(RUNTIME)
@Target(METHOD)
public @interface Logged {
}
```

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {
    @Override
    @Logged
    public int sum(int x, int y) {
        return x + y;
    }
    @Override
    public int diff(int x, int y) {
        return x - y;
    }
}
```

```
package advices;
// ...
@Aspect
public class TraceAdvices {

    @Around("@annotation(annotations.Logged)")
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
        JoinPointTrace.trace(">> ", joinPoint, null);
        final Object result = joinPoint.proceed();
        JoinPointTrace.trace("<< ", joinPoint, result);
        return result;
    }
}
```

Die Aufrufe aller mit `@Logged` ausgezeichneten Methoden werden geloggt werden.

10 JDBC

Spring vereinfacht die Implementierung von JDBC-basierten Anwendungen.

- Im Abschnitt 1 wird vorgestellt, wie eine JDBC-basierte Anwendung mit einer in der Spring-Konfiguration hinterlegten `DataSource` arbeiten kann.

- Im Abschnitt 2 wird die Spring-Klasse `JdbcTemplate` vorgestellt. Diese erleichtert die Implementierung von Inserts, Updates, Deletes und Queries – wir müssen uns die Erstellung von `PreperedStatement`s und `ResultSet`s nicht weiter kümmern.

- Im Abschnitt 3 geht's um das Spring-eigene Interface `RowMapper`. Die Verwendung dieses Interfaces vereinfacht die Formulierung von Queries noch weiter.

- Im Abschnitt 4 wird die Basisklasse `JdbcDaoSupport` vorgestellt, eine Klasse, von der konkrete DAO-Klassen abgeleitet werden können. Bei der Verwendung dieser Basisklasse müssen wir uns auch um die Erstellung von `JdbcTemplate`s nicht mehr kümmern.

- Im Abschnitt 5 wird ein Transaktions-Manager vorgestellt.

- Im Abschnitt 6 geht's um das Spring-eigene `TransactionTemplate`, welches die Programmierung von Transaktionen radikal vereinfacht.

- Im Abschnitt 7 geht's um Datenbanken und Multithreading.

- Im Abschnitt 8 und 9 geht's um Transaktions-Proxies.

- Im Abschnitt 10 schließlich zeigen wir, wie Transaktionen deklarativ (via Annotations) gesteuert werden können.

Wir können die Abschnitte zu den Transaktions-Mechanismen unverändert übernehmen, wenn wir statt auf der JDBC- auf der JPA-Ebene operieren (siehe das nächste Kapitel).

Alle folgenden Beispiele benutzen XML-Konfigurationen; für einige der Beispiele wird auch die `@Configuration`-Variante vorgestellt.

10.1 DataSource

Im folgenden wird zunächst gezeigt, wie eine einfache JDBC-basierte Anwendung mit Hilfe einer von Spring verwalteten `DataSource` aufgebaut sein könnte.

Wir benutzen alternativ eine von zwei `DataSource`-Implementierungen – die `c3p0-DataSource` und eine kleine einfache Implementierung.

Hier die triviale Implementierung (im `shared`-Projekt angesiedelt):

util.SimpleDateSource

```
package jn.util;

import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.SQLFeatureNotSupportedException;
import java.util.logging.Logger;

import javax.sql.DataSource;

public class SimpleDataSource implements DataSource {

    private PrintWriter writer;
    private int loginTimeout;

    private final String driverClass;
    private final String url;
    private final String user;
    private final String password;

    public SimpleDataSource(String driverClass, String url,
            String user, String password) {
        this.driverClass = driverClass;
        this.url = url;
        this.user = user;
        this.password = password;
    }
    @Override
    public Connection getConnection() throws SQLException {
        return DriverManager.getConnection(
            this.url, this.user, this.password);
    }
    @Override
    public Connection getConnection(String user, String password)
            throws SQLException {
        throw new SQLException();
    }
    @Override
```

```
    public PrintWriter getLogWriter() throws SQLException {
        return this.writer;
    }
    @Override
    public void setLogWriter(PrintWriter writer) throws SQLException {
        this.writer = writer;
    }
    @Override
    public int getLoginTimeout() throws SQLException {
        return this.loginTimeout;
    }
    @Override
    public void setLoginTimeout(int loginTimeout) throws SQLException {
        this.loginTimeout = loginTimeout;
    }
    @Override
    public boolean isWrapperFor(Class<?> cls) throws SQLException {
        throw new UnsupportedOperationException();
    }
    @Override
    public <T> T unwrap(Class<T> cls) throws SQLException {
        throw new UnsupportedOperationException();
    }
    @Override
    public Logger getParentLogger() throws SQLFeatureNotSupportedException {
        return null;
    }
}
```

Eine "richtige" `DataSource`-Implementierung (z.B. `c3p0`) verwaltet einen Pool von `Connection`s.

Die JDBC-Verbindungsdaten sind in einer Properties-Datei hinterlegt:

db.properties

```
db.driver           org.apache.derby.jdbc.EmbeddedDriver
db.url              jdbc:derby:../dependencies/derby/data
db.user             user
db.password         password
db.schema           USER
```

Die Datei `db.properties` befindet sich im `shared`-Projekt (letzteres ist daher im `CLASSPATH` des aktuellen Projekts enthalten).

spring.xml

```
<beans ...>

    <bean
    class="org.springframework.beans.factory.config.
                                PropertyPlaceholderConfigurer">
      <property name="location">
```

```
            <value>db.properties</value>
        </property>
    </bean>

    <bean id="simpleDataSource" class="jn.util.SimpleDataSource">
        <constructor-arg value="${db.driver}"/>
        <constructor-arg value="${db.url}"/>
        <constructor-arg value="${db.user}"/>
        <constructor-arg value="${db.password}"/>
    </bean>

    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${db.driver}"/>
        <property name="jdbcUrl" value="${db.url}"/>
        <property name="user" value="${db.user}"/>
        <property name="password" value="${db.password}"/>
    </bean>
</beans>
```

Unter dem Namen `"simpleDataSource"` ist unsere eigene Implementierung registriert, unter dem Namen `"dataSource"` die `c3p0`-Implementierung. Erstere benutzt Constructor-Injection, letztere Property-Injection. Die injizierten Werte werden der `db.properties`-Datei entnommen.

Das Datenbank-Schema wird durch folgenden `CREATE TABLE` beschrieben:

create.sql

```
create table account (
    number integer,
    balance integer,
    primary key (number)
)
```

Wir verwenden also einen fachlichen Schlüssel (`number`) als Primary-Key – typischerweise würde ein generierter Schlüssel verwendet werden…

Zeilen der `ACCOUNT`-Tabelle sollen auf Objekte der Klasse `Account` abgebildet werden:

domain.Account

```
package domain;

import java.io.Serializable;

public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    private int number;
    private int balance;
```

```
    public Account() {
    }
    public Account(int number) {
        this.number = number;
    }

    // getter, setter, toString...
}
```

Die Klasse `Account` gehorcht der Bean-Spezifikation (`Serializable`, parameterloser Konstruktor, setter/getter).

Hier die Klasse `Application`:

appl.Application

```
package appl;
// ...
import javax.sql.DataSource;

import db.util.appl.Db;
import domain.Account;

public class Application {
    public static void main(String[] args) {
        Db.aroundAppl();

        try (final ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext("spring.xml")) {
            final DataSource dataSource =
                (DataSource) ctx.getBean("dataSource");
            try (final Connection con = dataSource.getConnection()) {
                demo(con);
            }
            catch (Exception e) {
                System.out.println(e);
            }
        }
    }
    // ...
```

Mittels des Aufrufs von `Db.aroundAppl` wird die Datenbank aufgebaut (die `create.sql`-Datei wird ausgeführt).

Anschließend beschaffen wir uns mittels einen Spring-Lookups die `c3p0-DataSource`. Mittels dieser `DataSource` besorgen wir uns schließlich eine `Connection` (und nutzen dabei die Eigenschaft, dass `Connection` das `AutoCloseable`-Interface erweitert). Die `Connection` wird dann an `demo` übergeben:

```
    private static void demo(final Connection con) throws SQLException {
        insertAccount(con, new Account(4711));
```

```
        insertAccount(con, new Account(4712));
        final Account account1 = getAccount(con, 4711);
        account1.setBalance(account1.getBalance() + 5000);
        updateAccount(con, account1);
        final Account account2 = getAccount(con, 4712);
        account2.setBalance(account2.getBalance() + 6000);
        updateAccount(con, account2);
        for (final Account account : findAllAccounts(con))
            System.out.println(account);
        deleteAccount(con, account1);
    }
```

`demo` ruft die Methoden `insertAccount`, `updateAccount`, `deleteAccount` und `findAllAccounts` auf (und bildet hierbei `Account`-Objekte auf `ACCOUNT`-Zeilen und umgekehrt ab):

```
    private static void insertAccount(Connection con, Account account)
            throws SQLException {
        final String sql =
            "insert into account (number, balance) values (?, ?)";
        try (final PreparedStatement ps = con.prepareStatement(sql)) {
            ps.setInt(1, account.getNumber());
            ps.setInt(2, account.getBalance());
            if (ps.executeUpdate() != 1)
                throw new RuntimeException(
                    "account not inserted: " + account.getNumber());
        }
    }

    private static void updateAccount(Connection con, Account account)
            throws SQLException {
        final String sql = "update account set balance = ? where number = ?";
        try (final PreparedStatement ps = con.prepareStatement(sql)) {
            ps.setInt(1, account.getBalance());
            ps.setInt(2, account.getNumber());
            if (ps.executeUpdate() != 1)
                throw new RuntimeException(
                    "account not updated: " + account.getNumber());
        }
    }

    private static void deleteAccount(Connection con, Account account)
            throws SQLException {
        final String sql = "delete from account where number = ?";
        try (final PreparedStatement ps = con.prepareStatement(sql)) {
            ps.setInt(1, account.getNumber());
            if (ps.executeUpdate() != 1)
                throw new RuntimeException(
            "account not deleted: " + account.getNumber());
        }
    }

    private static Account getAccount(Connection con, int number)
            throws SQLException {
        final Account account = findAccount(con, number);
```

```
        if (account == null)
            throw new RuntimeException(
                "account with number " + number + " not found");
        return account;
    }

    private static Account findAccount(Connection con, int number)
            throws SQLException {
        final String sql =
            "select number, balance from account where number = ?";
        try (final PreparedStatement ps = con.prepareStatement(sql)) {
            ps.setInt(1, number);
            try (final ResultSet rs = ps.executeQuery()) {
                if (!rs.next())
                    return null;
                final Account a = new Account();
                a.setNumber(rs.getInt(1));
                a.setBalance(rs.getInt(2));
                return a;
            }
        }
    }

    private static List<Account> findAllAccounts(Connection con)
            throws SQLException {
        String sql = "select number, balance from account";
        try (final PreparedStatement ps = con.prepareStatement(sql)) {
            try (final ResultSet rs = ps.executeQuery()) {
                final List<Account> list = new ArrayList<>();
                while (rs.next()) {
                    final Account a = new Account();
                    a.setNumber(rs.getInt(1));
                    a.setBalance(rs.getInt(2));
                    list.add(a);
                }
                return list;
            }
        }
    }
}
```

Nach Ausführung der obigen `Application` sieht die Datenbank wie folgt aus:

```
ACCOUNT
NUMBER BALANCE
---------------
4712   6000
---------------
```

Die obige Anwendung ist reichlich geschwätzig. Spring wird Abhilfe schaffen…

Eine @Configuration-basierte Lösung

Die `spring.xml` kann durch folgende `ApplConfig`-Klasse ersetzt werden:

```
package appl;
// ...
@Configuration
@PropertySource("classpath:db.properties")
public class ApplConfig {

    @Value("${db.driver}")
    private String driver;

    @Value("${db.url}")
    private String url;

    @Value("${db.user}")
    private String user;

    @Value("${db.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        Log.log();
        return new SimpleDataSource(
            this.driver, this.url, this.user, this.password);
    }
}
```

In der `Application` müsste dann ein `AnnotationConfigApplicationContext` erzeugt werden.

10.2 JdbcTemplate

Mittels der Spring-Klasse `JdbcTemplate` können Datenbank-Operationen wesentlich knapper formuliert werden.

Bei der Erzeugung eines `JdbcTemplate`s wird eine `DataSource` übergeben.

`JdbcTemplate` hat eine `update`-Methode, mittels derer Inserts, Updates und Deletes formuliert werden können – und eine `query`-Methode, mittels derer Abfragen formuliert werden können.

Die `JdbcTemplate`-Methoden fangen die `SQLException`s ab und verpacken diese in `RuntimeException`s (Spring definiert eine Reihe von eigenen `Exception`-Klassen, die allesamt von `RuntimeException` abgeleitet sind).

Wir registrieren das `JdbcTemplate` in der Spring-Konfiguration:

spring.xml

```
<beans ...>

    // ...

    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
       // ...
    </bean>

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

Die `Application` des letzten Abschnitts kann dann wie folgt formuliert werden:

appl.Application

```
package appl;
// ...
import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.dao.DataRetrievalFailureException;
import org.springframework.jdbc.core.JdbcTemplate;

public class Application {
    public static void main(String[] args) {
        Db.aroundAppl();

        try (final ClassPathXmlApplicationContext ctx =
```

```
            new ClassPathXmlApplicationContext("spring.xml")) {
        final TransactionTemplate tt =
            ctx.getBean(TransactionTemplate.class);
        demo(template);
    }
}
```

An die `demo`-Methode wird nun statt der `DataSoruce` ein `JdbcTemplate` übergeben (welches seinerseits die `DataSource` kennt).

Die `demo`-Methode reicht nun das ihr übergebene `JdbcTemplate` an die von ihr aufgerufen Sub-Methoden weiter:

```
private static void demo(final JdbcTemplate template) {
    insertAccount(template, new Account(4711));
    insertAccount(template, new Account(4712));
    final Account account1 = getAccount(template, 4711);
    account1.setBalance(account1.getBalance() + 5000);
    updateAccount(template, account1);
    final Account account2 = getAccount(template, 4712);
    account2.setBalance(account2.getBalance() + 6000);
    updateAccount(template, account2);
    for (final Account account : findAllAccounts(template))
        System.out.println(account);
    deleteAccount(template, account1);
}
```

`insertAccount` benutzt nun die update-Methode von `JdbcTemplate`, der das `sql`-Statement und die für den `INSERT` erforderlichen Werte (`number`, `balance`) übergeben werden (letztere in Form eines VarArgs-Arrays):

```
private static void insertAccount(
        JdbcTemplate template, Account account) {
    final String sql =
        "insert into account (number, balance) values (?, ?)";
    final Object result = template.update(
        sql, account.getNumber(), account.getBalance());
    if ((Integer) result != 1)
        throw new DataRetrievalFailureException(account.toString());
}
```

Auch die `updateAccount` und `deleteAccount`-Methode benutzen dieselbe `update`-Methode:

```
private static void updateAccount(
        JdbcTemplate template, Account account) {
    final String sql =
        "update account set balance = ? where number = ?";
    final Object result = template.update(
        sql, account.getBalance(), account.getNumber());
    if ((Integer) result != 1)
        throw new DataRetrievalFailureException(account.toString());
```

```
    }

    private static void deleteAccount(
            JdbcTemplate template, Account account) {
        final String sql = "delete from account where number = ?";
        final Object result = template.update(sql, account.getNumber());
        if ((Integer) result != 1)
            throw new DataRetrievalFailureException(account.toString());
    }
```

Die `findAccount`- und `findAllAccounts`-Methoden benutzen die `JdbcTemplate`-Methode `query`. An `query` wird der SQL-String, ein Array von `Object`s und ein `ResultSetExtractor` übergeben. Dieses Spring-Interface `ResultSetExtractor` ist wie folgt definiert:

```
package org.springframework.jdbc.core;

@FunctionalInterface
public interface ResultSetExtractor<T> {
    public abstract T extractData(ResultSet rs)
           throws SQLException, DataAccessException;
}
```

Das Interface wird im folgenden mittels Lambda-Ausdrücken implementiert.

Der in `findAccount` verwendete `ResultSetExtractor` liefert aufgrund eines `ResultSet`s (`rs`) ein `Account`-Objekt zurück:

```
    private static Account findAccount(JdbcTemplate template, int number) {
        final String sql =
            "select number, balance from account where number = ?";
        return template.query(sql, new Object[] { number }, rs -> {
            if (!rs.next())
                return null;
            final Account a = new Account();
            a.setNumber(rs.getInt(1));
            a.setBalance(rs.getInt(2));
            return a;
        });
    }
```

Der in `findAllAccounts` verwendete `ResultSetExtractor` liefert aufgrund eines `ResultSet`s (`rs`) eine `List<Account>` zurück:

```
    private static List<Account> findAllAccounts(JdbcTemplate template) {
        final String sql = "select number, balance from account";
        return template.query(sql, new Object[] {}, rs -> {
            final List<Account> list = new ArrayList<>();
            while (rs.next()) {
                final Account a = new Account();
                a.setNumber(rs.getInt(1));
                a.setBalance(rs.getInt(2));
                list.add(a);
            }
            return list;
        });
    }
}
```

Die Verwendung der `JdbcTemplate`-Klasse entlastet uns somit von der expliziten Erzeugung von `PreparedStatement`s und `ResultSet`s. Und weil alle `SQLException`s in `RuntimeException`s transformiert werden, wird auch die Fehlerbehandlung wesentlich angenehmer…

10.3 RowMapper

Mittels der Verwendung der Spring-eigenen `RowMapper`-Interfaces können wir Abfragen noch einfacher formuliert werden. Das Interface ist wie folgt definiert:

```
package org.springframework.jdbc.core;

@FunctionalInterface
interface RowMapper<T> {
    public abstract T mapRow(ResultSet rs, int rowNum)
        throws SQLException;
}
```

appl.Application

Hier eine Lambda-Implementierung des Interfaces:

```
    private static RowMapper<Account> mapper = (rs, rowNumber) -> {
        final Account a = new Account();
        a.setNumber(rs.getInt(1));
        a.setBalance(rs.getInt(2));
        return a;
    };
```

Dieses Lambda-Objekt können wir nun sowohl in der Implementierung von `findAccount` als auch von `findAllAccounts` verwenden. Der `RowMapper` wird in `findAccount` an die `JdbcTemplate`-Methode `queryForObject` übergeben und in `findAllAccounts` an eine überladene `query`-Methode:

```
    private static Account findAccount(JdbcTemplate template, int number) {
        final String sql =
            "select number, balance from account where number = ?";
        return template.queryForObject(sql, mapper, number);
    }
```

```
    private static List<Account> findAllAccounts(JdbcTemplate template) {
        final String sql = "select number, balance from account";
        return template.query(sql, mapper);
    }
```

`queryForObject` liefert eine Spring-Exception, wenn die abzubildende Ergebnismenge keinen oder mehrere Treffer enthält (`EmptyResultDataAccessException` resp. `IncorrectResultSizeDataAccessException`). `query` liefert immer eine Liste (die möglicherweise leer ist).

10.4 JdbcDaoSupport

Typischerweise werden Datenbank-Zugriffe in DAO-Klassen implementiert (Data Access Objects).

Für solche Implementierungen bietet Spring eine Basisklasse an: `JdbcDaoSupport`. Diese Klasse stellt bereits das einer DAO-Implementierung erforderliche `JdbcTempate` zur Verfügung – es kann via `getJdbcTemplate` ermittelt werden. Wir brauchen uns also um die Erzeugung des `JdbcTemplate`s nicht mehr kümmern – und benötigen also auch keinen Lookup mehr auf die Spring-Konfiguration.

Sei z.B. folgendes Interface definiert:

daos.AccountDao

```
package daos;
// ...
public interface AccountDao {
    public abstract void insert(Account account);
    public abstract void update(Account account);
    public abstract void delete(Account account);
    public abstract Account find(int number);
    public abstract List<Account> findAll();
}
```

Dann kann dieses mittels der Basisklasse `JdbcDaoSupport` wie folgt implementiert werden:

daos.jdbc.AccountDaoImpl

```
package daos.jdbc;
// ...
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class AccountDaoImpl
        extends JdbcDaoSupport implements AccountDao {

    @Override
    public void insert(Account account) {
        final String sql =
            "insert into account (number, balance) values (?, ?)";
        final Object result = this.getJdbcTemplate().update(
            sql, account.getNumber(), account.getBalance());
        if ((Integer) result != 1)
            throw new RuntimeException("cannot insert: " + account);
    }
    // ...
}
```

In der Spring-Konfiguration müssen nurmehr eine `DataSource` und natürlich das DOA-Objekt registriert werden:

spring.xml

```
<beans ...>

    <bean class="org.springframework.beans.factory.config.
                                    PropertyPlaceholderConfigurer">
        <property name="location">
             <value>db.properties</value>
        </property>
    </bean>

    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${db.driver}" />
        <property name="jdbcUrl" value="${db.url}" />
        <property name="user" value="${db.user}" />
        <property name="password" value="${db.password}" />
    </bean>

    <bean id="accountDao" class="daos.jdbc.AccountDaoImpl">
        <property name="dataSource" ref="dataSource" />
    </bean>

</beans>
```

Man beachte die Definition von `"accountDao"`: `dataSource` ist eine Property der Basisklasse `JdbcDaoSupport`. Aufgrund der hier angegebenen `DataSource` kann Spring hinter unserem Rücken das `JdbcTemplate` erzeugen.

Die Klasse `Application` muss von `DataSource`s und `JdbcTemplate`s nichts mehr wissen:

appl.Application

```
package appl;
// ...
public class Application {
    public static void main(String[] args) {
        Db.aroundAppl();
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final AccountDao accountDao = ctx.getBean(AccountDao.class);

            accountDao.insert(new Account(4711));
            accountDao.insert(new Account(4712));
            final Account account1 = accountDao.find(4711);
            account1.setBalance(account1.getBalance() + 5000);
            accountDao.update(account1);
            final Account account2 = accountDao.find(4712);
            accountDao.delete(account2);
            accountDao.findAll().forEach(System.out::println);
        }
    }
}
```

Nach Ausführung des obigen Programms hat die Datenbank den folgenden Zustand:

```
domain.Account [4711, 5000]
```

10.5 TransactionManager

Im folgenden untersuchen wir, wie mittels des Spring-eigenen `TransactionManager`s Transaktionen gesteuert werden können.

Wir bauen zu diesem Zweck eine kleine Anwendung, die aus drei Schichten besteht: einer DAO-Schicht, die für die Persistenz zuständig ist, eine Service-Schicht, in welcher die Fachlichkeit implementiert ist und einer Controller-Schicht (die durch ein einfache main-Methode repräsentiert wird). Die Service-Schicht nutzt die DAO-Schicht, die Controller-Schicht nutzt ausschließlich die Service Schicht.

Wir verwenden auch hier wieder die bereits bekannt `Account`-Klasse:

domain.Account

```
package domain;
// ...
public class Account implements Serializable {
    // ...
}
```

In allen folgenden Beispielen verwenden wir folgendes DAO-Interface:

daos.AccountDao

```
package daos;
// ...
public interface AccountDao {
    public abstract void insert(Account account);
    public abstract void update(Account account);
    public abstract void delete(Account account);
    public abstract Account get(int number);
    public abstract Account find(int number);
    public abstract List<Account> findAll();
}
```

Die `get`-Methode liefert bei erfolgloser Suche eine Exception; die `find`-Methode liefert bei einer erfolglosen Suche `null` zurück.

Hier ein Ausschnitt aus der Implementierung des obigen Interfaces:

daos.jdbc.AccountDaoImpl

```
package daos.jdbc;
// ...
public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {

    @Override
```

```
    public void insert(Account account) {
        final String sql =
            "insert into account (number, balance) values (?, ?)";
        final Object result = this.getJdbcTemplate().update(
            sql, account.getNumber(), account.getBalance());
        if ((Integer) result != 1)
            throw new RuntimeException("cannot insert: " + account);
    }

    // ...
}
```

Die DAO-Klasse ist abgeleitet von `JdbcDaoSupport`, um auf einfache Weise das `JdbcTemplate` nutzen zu können.

Während die DAO-Schicht nur Persistenzlogik implementiert, stellt die Service-Schicht die eigentliche fachliche Funktionalität zur Verfügung. Der Konto-Service ist mittels des Interfaces `AccountService` spezifiziert:

services.AccountService

```
package services;
// ...
public interface AccountService {
    public abstract void createAccount(int number);
    public abstract void deleteAccount(int number);
    public abstract Account getAccount(int number);
    public abstract Account findAccount(int number);
    public abstract List<Account> findAllAccounts();
    public abstract void deposit(int number, int amount);
    public abstract void withdraw(int number, int amount);
    public abstract void transfer(
        int fromNumber, int toNumber, int amount);
}
```

Die Methoden-Spezifikationen sind selbsterklärend.

Das obige Interface wird von `AccountServiceImpl` implementiert:

services.AccountServiceImpl

```
package services;
// ...
public class AccountServiceImpl implements AccountService {

    private final AccountDao accountDao;

    public AccountServiceImpl(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    @Override
```

```
    public void createAccount(int number) {
        this.accountDao.insert(new Account(number));
    }

    @Override
    public void deleteAccount(int number) {
        final Account account = this.accountDao.get(number);
        if (account.getBalance() != 0)
            throw new RuntimeException("balance must be 0");
        this.accountDao.delete(account);
    }

    @Override
    public Account getAccount(int number) {
        return this.accountDao.get(number);
    }

    @Override
    public Account findAccount(int number) {
        return this.accountDao.find(number);
    }

    @Override
    public List<Account> findAllAccounts() {
        return this.accountDao.findAll();
    }

    @Override
    public void deposit(int number, int amount) {
        if (amount <= 0)
            throw new RuntimeException("bad amount: " + amount);
        final Account account = this.accountDao.get(number);
        account.setBalance(account.getBalance() + amount);
        this.accountDao.update(account);
    }

    @Override
    public void withdraw(int number, int amount) {
        if (amount <= 0)
            throw new RuntimeException("bad amount: " + amount);
        final Account account = this.accountDao.get(number);
        if (amount > account.getBalance())
            throw new RuntimeException("cannot withdraw: " + amount);
        account.setBalance(account.getBalance() - amount);
        this.accountDao.update(account);
    }

    @Override
    public void transfer(int fromNumber, int toNumber, int amount) {
        this.deposit(toNumber, amount);
        this.withdraw(fromNumber, amount);
    }
}
```

Über den Konstruktor wird ein `AccountDao` injiziert werden.

Man beachte, dass die `withdraw`-Methode eine Exception wirft, wenn der abzuhebende Betrag größer ist als der Bestand des Kontos.

Man beachte weiterhin, dass in `transfer` zunächst die `deposit`-Methode für das Zielkonto und dann erst die `withdraw`-Methode für das Quellkonto aufgerufen wird. Diese Reihenfolge ist natürlich kontra-intuitiv. Da die Methode aber im Kontext einer Transaktion aufgerufen wird, muss auch diese Reihenfolge funktionieren (die Wirkung des `deposit` muss ggf. zurückgenommen werden, wenn der folgende `withdraw` fehlschlägt).

Hier die Spring-Konfiguration:

spring.xml

```
<beans ...>

    // ...

    <bean id="dataSource"
        class="com.mchange.v2.c3p0.ComboPooledDataSource">
        // ...
    </bean>

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


    <bean id="accountDao" class="daos.jdbc.AccountDaoImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="accountService" class="services.AccountServiceImpl">
        <constructor-arg ref="accountDao"/>
    </bean>

</beans>
```

Unter dem Namen `"transactionManager"` wird ein `DataSourceTransactionManager` registriert. Über dessen Property `"dataSource"` wird ihm die zuvor registrierte `DataSource` injiziert.

Zusätzlich wird eine `AccountDao` und ein `AccountService` registriert.

Dem `AccountDao` wird über die `"dataSource"`-Property der Basisklasse `JdbDaoSupport` die `DataSource` injiziert (damit das `AccountDao` mit dem `JdbcTemplate` operieren kann); dem `AccountService` wird das `AccountDao` injiziert (via Constructor-Injection).

In der folgenden Anwendung rufen wir nun einige Service-Methoden auf. Jede Aufruf einer solchen Methode wird in einer eigenen Transaktion ausgeführt werden.

appl.Application

```
package appl;
// ...
import org.springframework.context.support.
                                    ClassPathXmlApplicationContext;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.
                                    DefaultTransactionDefinition;

public class Application {

    public static void main(String[] args) {
        Db.aroundAppl();

        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            final PlatformTransactionManager tm =
                ctx.getBean(PlatformTransactionManager.class);
            final AccountService accountService =
                ctx.getBean(AccountService.class);

            createAccount(tm, accountService, 4711);
            createAccount(tm, accountService, 4712);
            deposit(tm, accountService, 4711, 5000);
            deposit(tm, accountService, 4712, 6000);
            withdraw(tm, accountService, 4711, 2000);
            transfer(tm, accountService, 4711, 4712, 500);

            final List<Account> list =
                findAllAccounts(tm, accountService);
            list.forEach(System.out::println);
        }
    }
    // ...
```

Neben dem `AccountService` besorgen wir uns von der Spring-Konfiguration einen `PlatformTransactionManager` – genauer: ein Objekt, dessen Klasse das Interface `PlatformTransactionManager` implementiert.

Es wird sich dabei um ein Objekt der Klasse `DataSourceTransactionManager` handeln (siehe die `spring.xml`). Es wird also der JDBC-Transaktionsmechanismus verwendet werden (im Unterschied zu einem JPA-Transaktionsmechanismus, der im Kontext von JPA verwendet wird – siehe das Kapitel zu JPA).

Allen aufgerufenen Demo-Methoden wird dann u.a. der Transaktions-Manager und der `AccountService` übergeben

In jeder der Demo-Methoden wird zunächst ein Objekt des Typs `DefaultTransactionDefinition` erzeugt. Ein solches Objekt dient u.a. zur Fein-Einstellung des Transaktionsverhaltens (via `setPropagationBehavior`).

Das Objekt wird an die Methode `getTransaction` übergeben, die auf den Transaktions-Manager aufgerufen wird. Letztere startet eine Transaktion und liefert ein Objekt des Typs `TransactionStatus` zurück. Dieses repräsentiert den Zustand der Transaktion und wird an die Transaktions-Manager-Methoden `commit` und `rollback` übergeben. `commit` wird aufgerufen, wenn die aufgerufene Service-Methode normal zurückkehrt; liefert sie dagegen eine Exception, wird `rollback` aufgerufen.

Alle Demo-Methoden sind nach demselben Schema aufgebaut:

```
    // ...
    private static void createAccount(PlatformTransactionManager tm,
            AccountService accountService, int number) {
        final DefaultTransactionDefinition td =
            new DefaultTransactionDefinition();
        td.setPropagationBehavior(
            TransactionDefinition.PROPAGATION_REQUIRED);
        final TransactionStatus ts = tm.getTransaction(td);
        try {
            accountService.createAccount(number);
            tm.commit(ts);
        }
        catch (RuntimeException e) {
            tm.rollback(ts);
            throw e;
        }
    }

    private static void deposit(PlatformTransactionManager tm,
            AccountService accountService, int number, int amount) {
        final DefaultTransactionDefinition td =
            new DefaultTransactionDefinition();
        td.setPropagationBehavior(
            TransactionDefinition.PROPAGATION_REQUIRED);
        final TransactionStatus ts = tm.getTransaction(td);
        try {
            accountService.deposit(number, amount);
            tm.commit(ts);
        }
        catch (RuntimeException e) {
            tm.rollback(ts);
            throw e;
        }
    }

    private static void withdraw(PlatformTransactionManager tm,
```

```
            AccountService accountService, int number, int amount) {
        final DefaultTransactionDefinition td =
            new DefaultTransactionDefinition();
        td.setPropagationBehavior(
            TransactionDefinition.PROPAGATION_REQUIRED);
        final TransactionStatus ts = tm.getTransaction(td);
        try {
            accountService.withdraw(number, amount);
            tm.commit(ts);
        }
        catch (RuntimeException e) {
            tm.rollback(ts);
            throw e;
        }
    }

    private static void transfer(PlatformTransactionManager tm,
             AccountService accountService,
             int fromNumber, int toNumber, int amount) {
        final DefaultTransactionDefinition td =
            new DefaultTransactionDefinition();
        td.setPropagationBehavior(
            TransactionDefinition.PROPAGATION_REQUIRED);
        final TransactionStatus ts = tm.getTransaction(td);
        try {
            accountService.transfer(fromNumber, toNumber, amount);
            tm.commit(ts);
        }
        catch (RuntimeException e) {
            tm.rollback(ts);
            throw e;
        }
    }

    private static List<Account> findAllAccounts(
            PlatformTransactionManager tm, AccountService accountService) {
        final DefaultTransactionDefinition td =
            new DefaultTransactionDefinition();
        td.setPropagationBehavior(
            TransactionDefinition.PROPAGATION_SUPPORTS);
        final TransactionStatus ts = tm.getTransaction(td);
        try {
            final List<Account> list = accountService.findAllAccounts();
            tm.commit(ts);
            return list;
        }
        catch (RuntimeException e) {
            tm.rollback(ts);
            throw e;
        }
    }
}
```

Man beachte, dass man es den Methoden nicht ansieht, welcher tatsächlicher Transaktions-Manager benutzt wird (ein JDBC- oder ein JPA-basierter) – alle Methoden beziehen sich auf den Transaktions-Manager ausschließlich über das Interface `PlatformTransactionManager`.

Um die Korrektheit des oben implementierten Transaktions-Verhaltens zu demonstrieren, könnte die `transfer`-Methode mit einem zu hohen Überweisungsbetrag aufgerufen werden – etwa wie folgt:

```
transfer(tm, accountService, 4711, 4712, 500000);
```

Dann müsste die Wirkung des Aufrufs der `deposit`-Methode, der ja vor(!) dem Aufruf von `withdraw` erfolgt, wieder zurückgenommen worden sein (weil `withdraw` in dieser Situation eine Exception werfen würde).

Jede der Demo-Methoden hat denselben formalen Aufbau. Für diesen immer gleichen Aufbau bietet Spring eine entsprechende Abstraktion an. Diese Abstraktion wird im nächsten Abschnitt vorgestellt.

10.6 TransactionTemplate

Mittels des Spring-eigenen `TransactionTemplate`s können wir Transaktionen wesentlich knapper und konziser formulieren.

Alle Klassen – bis auf die `Application`-Klasse – werden vom letzten Abschnitt unverändert übernommen.

In der Spring-Konfiguration wird nun das `TransactionTemplate` registriert:

spring.xml

```
<beans ...>

    // ...

    <bean id="dataSource"
        class="com.mchange.v2.c3p0.ComboPooledDataSource">
        // ...
    </bean>

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

    <bean id="transactionTemplate"
            class="org.springframework.transaction.support.
                        TransactionTemplate">
        <constructor-arg ref="transactionManager"/>
    </bean>

    <bean id="accountDao" class="daos.jdbc.AccountDaoImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="accountService" class="services.AccountServiceImpl">
        <constructor-arg ref="accountDao"/>
    </bean>

</beans>
```

Bei der Erzeugung eines `TransactionTemplate`s muss dem Konstruktor ein Transaktions-Manager übergeben werden. Das `TransactionTemplate` kennt anschließend also den Transaktion-Manager und kann diesen nutzen.

Hier die neue `Application`:

appl.Application

```
package appl;
// ...
import org.springframework.transaction.support.TransactionCallback;
import ...transaction.support.TransactionCallbackWithoutResult;
import ...transaction.support.TransactionTemplate;

public class Application {

    public static void main(String[] args) {
        Db.aroundAppl();

        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            final TransactionTemplate tt =
                ctx.getBean(TransactionTemplate.class);
            final AccountService accountService =
                ctx.getBean(AccountService.class);

            createAccount(tt, accountService, 4711);
            createAccount(tt, accountService, 4712);
            deposit(tt, accountService, 4711, 5000);
            deposit(tt, accountService, 4712, 6000);
            withdraw(tt, accountService, 4711, 2000);
            transfer(tt, accountService, 4711, 4712, 500);

            final List<Account> list =
                findAllAccounts(tt, accountService);
            list.forEach(System.out::println);
        }
    }
    // ...
```

Wir besorgen uns von Spring ein `TransactionTemplate`. Den Demo-Methoden übergeben wir dann statt des Transaktions-Managers das Template.

Die Klasse `TransactionTemplate` hat eine Methode `execute`:

```
package org.springframework.transaction.support;

public class TransactionTemplate … {
    // ...
    public <T> T execute(TransactionCallback<T> action)
          throws TransactionException { … }
}
```

Diese `execute`-Methode bereitet die Transaktion vor, ruft dann die an ihr übergebene `action` auf und terminiert schließlich die Transaktion.

`TransactionCallback` ist ein funktionales Interface:

```
package org.springframework.transaction.support;

@FunctionalInterface
public interface TransactionCallback<T> {
    T doInTransaction(TransactionStatus status);
}
```

Neben diesem Interface existiert folgende Klasse:

```
package org.springframework.transaction.support;

public abstract class TransactionCallbackWithoutResult
         implements TransactionCallback<Object> {

    @Override
    public final Object doInTransaction(TransactionStatus status) {
        doInTransactionWithoutResult(status);
        return null;
    }
    protected abstract void doInTransactionWithoutResult(
        TransactionStatus status);
}
```

Auch ein `TransactionCallbackWithoutResult` ist also ein `TransactionCallback` – ein `TransactionCallback`, dessen `doInTransaction` bereits implementiert ist. Diese Methode delegiert an eine `doInTransactionWithoutResult`, die in abgeleiteten Klassen implementiert werden muss.

`createAccount`, `deposit`, `withdraw` und `transfer` rufen die `execute`-Methode mit einem `TransactionCallbackWithoutResult` auf (denn die jeweils aufgerufene Service-Methode liefert nichts zurück). Die Methoden implementieren die jeweilige Callback-Klasse in Form anonymer Klassen:

```
    // ...
    private static void createAccount(TransactionTemplate tt,
            AccountService accountService, int number) {
        tt.execute(new TransactionCallbackWithoutResult() {
            public void doInTransactionWithoutResult(TransactionStatus ts) {
                accountService.createAccount(number);
            }
        });
    }

    private static void deposit(TransactionTemplate tt,
            AccountService accountService, int number, int amount) {
        tt.execute(new TransactionCallbackWithoutResult() {
            public void doInTransactionWithoutResult(TransactionStatus ts) {
                accountService.deposit(number, amount);
            }
        });
```

```
    }

    private static void withdraw(TransactionTemplate tt,
            AccountService accountService, int number, int amount) {
        tt.execute(new TransactionCallbackWithoutResult() {
            public void doInTransactionWithoutResult(TransactionStatus ts) {
                accountService.withdraw(number, amount);
            }
        });
    }

    private static void transfer(TransactionTemplate tt,
            AccountService accountService,
            int fromNumber, int toNumber, int amount) {
        tt.execute(new TransactionCallbackWithoutResult() {
            public void doInTransactionWithoutResult(TransactionStatus ts) {
                accountService.transfer(fromNumber, toNumber, amount);
            }
        });
    }
```

Die `findAllAccounts`-Methode übergibt an `execute` ein Objekt, dessen Klasse das Interface `TransactionCallback` implementiert (also sinngemäß ein Objekt der Klasse "TransactionCallbackWithResult"). Das, was die in `doInTransaction` aufgerufene Service-Methode liefert, wird dann von `excute` an die Anwendung zurückgeliefert:

```
    private static List<Account> findAllAccounts(TransactionTemplate tt,
            AccountService accountService) {
        return tt.execute(new TransactionCallback<List<Account>>() {
            public List<Account> doInTransaction(TransactionStatus ts) {
                return accountService.findAllAccounts();
            }
        });
    }
}
```

Da in der Methode `findAllAccounts` einfach ein funktionales Interface (`TransactionCallback`) implementiert wird, könnten wir diese Implementierung auch durch einen Lambda-Ausdruck ersetzen:

```
    private static List<Account> findAllAccounts(TransactionTemplate tt,
            AccountService accountService) {
        return tt.execute(ts -> accountService.findAllAccounts());
    }
```

Und auch die anderen Methoden (`createAccount`, `deposit` etc.) könnten mittels Lambdas implementiert werden – sofern wir uns eines kleinen Tricks bedienen:

```
    private static void createAccount(TransactionTemplate tt,
            AccountService accountService, int number) {
        tt.execute((TransactionCallback<Void>)(ts -> {
            accountService.createAccount(number);
```

```
            return null;
        }));
    }
```

(Ob das aber die Sache übersichtlicher macht?)

Dass die Lambda-Implementierung hier auf gewisse Schwierigkeiten stößt, ist natürlich der historischen Entwicklung des Spring-Frameworks geschuldet: Spring ist nun einmal älter als Java-8…

Resultat:

Das gesamte "Drumherum" einer Transaktion wird durch das `TransactionTemplate` abstrahiert. Die Anwendung kann sich darauf konzentrieren, was innerhalb einer Transaktion fachlich ausgeführt werden muss.

Eine @Configuration-basierte Lösung

Die `spring.xml` kann durch folgende `ApplConfig`-Klasse ersetzt werden:

appl.ApplConfig

```
package appl;
// ...
@Configuration
@PropertySource("classpath:db.properties")
public class ApplConfig {

    @Value("${db.driver}")
    private String driver;

    @Value("${db.url}")
    private String url;

    @Value("${db.user}")
    private String user;

    @Value("${db.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        Log.log();
        return new SimpleDataSource(
            this.driver, this.url, this.user, this.password);
    }
    @Bean
    public PlatformTransactionManager transactionManager() {
        return new DataSourceTransactionManager(this.dataSource());
    }
```

```
    @Bean
    public TransactionTemplate transactionTemplate() {
        return new TransactionTemplate(this.transactionManager());
    }
    @Bean
    public AccountDao accountDao() {
        final AccountDaoImpl dao = new AccountDaoImpl();
        dao.setDataSource(this.dataSource());
        return dao;
    }
    @Bean
    public AccountService accountService() {
        return new AccountServiceImpl(this.accountDao());
    }
}
```

In der `Application` muss dann natürlich ein `AnnotationConfigApplicationContext` erzeugt werden.

10.7 Multithreading

Die folgende Anwendung demonstriert, dass das `TransactionTemplate` auch dann korrekt funktioniert, wenn mehrere Threads gleichzeitig die Service-Methoden aufrufen.

Die Klasse `AccountDaoImpl` nutzt eine kleine Hilfsmethode, welche die aktuell verwendete `Connection` ausgibt und dann eine Sekunde lang schläft:

daos.jdbc.AccountDaoImpl

```
    private void traceAndSleep(String where) {
        System.out.println(where + " " +
                Thread.currentThread().getId() + " " +
                this.getConnection());
        try {
            Thread.sleep(1000);
        }
        catch (final Exception e) {
            throw new RuntimeException(e);
        }
    }
```

In der `update`-Methode wird diese Hilfsmethode aufgerufen:

```
    // ...
    @Override
    public void update(Account account) {
        this.traceAndSleep("update");
        final String sql =
            "update account set balance = ? where number = ?";
        final Object result = this.getJdbcTemplate().update(sql,
            account.getBalance(), account.getNumber());
        if ((Integer) result != 1)
            throw new RuntimeException("cannot update: " + account);
    }
    // ...
```

Die `main`-Methode erzeugt zwei Threads, innerhalb derer jeweils die `deposit`- und die `withdraw`-Methode aufgerufen wird. Die beiden Threads werden im Abstand von einer halben Sekunde gestartet:

appl.Application

```
    // ...
    public static void main(String[] args) throws Exception {
        Db.aroundAppl();

        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final TransactionTemplate tt =
                ctx.getBean(TransactionTemplate.class);
            final AccountService accountService =
                ctx.getBean(AccountService.class);

            createAccount(tt, accountService, 4711);
            createAccount(tt, accountService, 4712);

            Thread t1 = new Thread(() -> {
                deposit(tt, accountService, 4711, 5000);
                withdraw(tt, accountService, 4711, 2000);
            });
            Thread t2 = new Thread(() -> {
                deposit(tt, accountService, 4712, 6000);
                withdraw(tt, accountService, 4712, 3000);
            });

            t1.start();
            Thread.sleep(500);
            t2.start();

            t1.join();
            t2.join();

            findAllAccounts(tt, accountService).forEach(System.out::println);
        }
    }
    // ...
```

Die Ausgaben zeigen, dass jeder Thread seine eigene `Connection` nutzt:

```
insert 1 com.mchange.v2.c3p0.impl.NewProxyConnection@4f0f2942
insert 1 com.mchange.v2.c3p0.impl.NewProxyConnection@2e570ded

get 21 com.mchange.v2.c3p0.impl.NewProxyConnection@7762b905
get 22 com.mchange.v2.c3p0.impl.NewProxyConnection@4d648482
update 21 com.mchange.v2.c3p0.impl.NewProxyConnection@7762b905
update 22 com.mchange.v2.c3p0.impl.NewProxyConnection@4d648482

get 21 com.mchange.v2.c3p0.impl.NewProxyConnection@31d55daf
get 22 com.mchange.v2.c3p0.impl.NewProxyConnection@2c6982a3
update 21 com.mchange.v2.c3p0.impl.NewProxyConnection@31d55daf
update 22 com.mchange.v2.c3p0.impl.NewProxyConnection@2c6982a3
```

10.8 TransactionProxy

Alternativ zum Template-Mechanismus kann auch der Proxy-Mechanismus zur Steuerung von Transaktionen genutzt werden.

Die `getBean`-Methode wird nicht den "richtigen" `AccountService` liefern, sondern ein Proxy. Über den an das Proxy angeschlossenen `InvocationHandler` wird dann (Dreisatz!) die Transaktion gestartet, die entsprechende Methode auf den "richtigen" Service aufgerufen und nach deren Rückkehr die Transaktion geschlossen.

Hier zunächst die `spring.xml`:

spring.xml

```
<beans ...>
    // ...

    <bean id="dataSource"
             class="com.mchange.v2.c3p0.ComboPooledDataSource">
        // ...
    </bean>

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


    <bean id="accountDao" class="daos.jdbc.AccountDaoImpl">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="accountService"
        class="org.springframework.transaction.interceptor
                          .TransactionProxyFactoryBean">
        <property name="target">
            <bean class="services.AccountServiceImpl">
                <constructor-arg ref="accountDao" />
            </bean>
        </property>
        <property name="transactionManager" ref="transactionManager" />
        <property name="transactionAttributeSource">
            <bean
                class="org.springframework.transaction.interceptor.
                            MatchAlwaysTransactionAttributeSource" />
        </property>
    </bean>

</beans>
```

In der Konfiguration wird eine `TransactionProxyFactoryBean` registriert. Dieser Beans werden folgende Properties injiziert:

- An `target` wird das eigentliche `AccountServiceImpl`-Objekt zugewiesen.
- An `transactionManager` wird der zuvor erzeugt `TransactionManger` zugewiesen.
- An `transactionAttributeSource` wird ein Objekt zugewiesen, welches dafür sorgt, dass jeder Aufruf einer der Service-Methoden im Kontext einer neuen Transaktion laufen wird.

Die Application erhalt von `getBean` die Referenz auf ein Proxy zurück. Jede Methode, die auf das Proxy aufgerufen wird, läuft im Kontext einer jeweils neuen Transaktion:

appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) {
        Db.aroundAppl();

        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final AccountService accountService =
                ctx.getBean(AccountService.class);
            System.out.println(accountService.getClass().getName());
            accountService.createAccount(4711);
            accountService.createAccount(4712);
            accountService.deposit(4711, 5000);
            accountService.deposit(4712, 6000);
            accountService.withdraw(4711, 2000);
            accountService.transfer(4711, 4712, 500);
            final List<Account> list = accountService.findAllAccounts();
            list.forEach(System.out::println);
        }
    }
}
```

10.9 AutoProxy

Im folgenden Beispiel wird eine weitere Proxy-Variante vorgestellt.

Über die `spring.xml` wird ein `TransactionInterceptor` erzeugt (dem u.a. der `TransactionManager` mitgegeben wird).

Weiterhin wird ein `BeanNameAutoProxyCreator` erzeugt. Dieser Bean wird eine Liste von Interceptoren und eine Liste von Bean-Namen übergeben. Sie sorgt dafür, dass jeder der in der Bean-Liste enthaltene Bean alle Interceptoren vorangestellt werden:

spring.xml

```
<beans ...>
    // ...
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        // ...
    </bean>

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

    <bean id="accountDao" class="daos.jdbc.AccountDaoImpl">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="accountService" class="services.AccountServiceImpl">
        <constructor-arg ref="accountDao" />
    </bean>

    <bean id="transactionInterceptor"
            class="org.springframework.transaction.interceptor.
                        TransactionInterceptor">
        <property name="transactonManager" ref="transactionManager"/>
        <property name="transactionAttributeSource">
            <bean class="org.springframework.transaction.interceptor.
                            MatchAlwaysTransactionAttributeSource"/>
        </property>
    </bean>

    <bean class="org.springframework.aop.framework.autoproxy.
                                BeanNameAutoProxyCreator">
        <property name="interceptorNames">
            <list>
                <value>transactionInterceptor</value>
            </list>
        </property>
        <property name="beanNames">
```

```
            <list>
                <value>accountService</value>
            </list>
        </property>
    </bean>

</beans>
```

Wir können dieselbe `Application` nutzen wie im letzten Abschnitt.

10.10 Annotations

Der folgende Abschnitt stellt eine weitere Variante des Proxy-Mechanismus der Transaktionsteuerung vor. Die Transaktionssteuerung geschieht hier rein deklarativ – mittels Annotationen in der Service-Klasse.

Die `spring.xml` ist erheblich geschrumpft – in ihr ist von Interceptoren nun keine Rede mehr. Statt dessen enthält sie den Eintrag `<tx:annotation-driven>`:

spring.xml

```
<beans ...>

    <context:annotation-config/>
    <tx:annotation-driven/>

    // ...

    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        // ...
    </bean>

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

    <bean id="accountDao" class="daos.jdbc.AccountDaoImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="accountService" class="services.AccountServiceImpl">
        <constructor-arg ref="accountDao"/>
    </bean>

</beans>
```

Die Service-Implementierung benutzt nun die Annotation `@Transactional`. Diese Annotation hat ein Attribut namens `propagation`. Über dieses Attribut kann das Transaktionsverhalten genau beschrieben werden.

`@Transactional` kann sowohl auf Klassen- als auch auf Methodenebene verwendet werden. Eine `@Transactional`-Annotation auf Methodenebene "überschreibt" die auf der Klassenebene verwendete Annotation.

Als `propagation` kann z.B. `REQUIRED` verwendet werden (das ist auch der Default). `REQUIRED` besagt folgendes: Existiert für den aktuellen Thread im Augenblick bereits eine Transaktion, so wird die Methode im Kontext eben dieser Transaktion ausgeführt.

Ansonsten wird eine neue Transaktion gestartet – und nach Beendigung des Aufrufs der Methode auch wieder terminiert.

`SUPPORT` z.B. besagt, dass die Methode ohne Transaktion oder aber im Kontext einer bereits existierenden Transaktion laufen kann.

Weitere Werte für das `propagation`-Attribut: `MANDANTORY`, `NEVER`, `NESTED`, `REQUIRES_NEW`, `NOT_SUPPORTED`.

Die Klasse ist mit `REQUIRED` markiert – alle Methoden ohne `@Transactional`-Annotation laufen also in diesem Modus. Die `get`- und die `find`-Methoden sind mit einer eigenen Annotation ausgestattet – mit dem `propagation`-Wert `SUPPORTS`:

services.AccountServiceImpl

```
package services;
// ...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@Transactional(propagation=Propagation.REQUIRED)
public class AccountServiceImpl implements AccountService {

    private final AccountDao accountDao;

    public AccountServiceImpl(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    @Override
    public void createAccount(int number) {
        this.accountDao.insert(new Account(number));
    }

    @Override
    public void deleteAccount(int number) {
        final Account account = this.accountDao.get(number);
        if (account.getBalance() != 0)
            throw new RuntimeException("balance must be 0");
        this.accountDao.delete(account);
    }

    @Override
    @Transactional(propagation=Propagation.SUPPORTS)
    public Account getAccount(int number) {
        return this.accountDao.get(number);
    }

    @Override
    @Transactional(propagation=Propagation.SUPPORTS)
    public Account findAccount(int number) {
        return this.accountDao.find(number);
```

```
    }

    @Override
    @Transactional(propagation=Propagation.SUPPORTS)
    public List<Account> findAllAccounts() {
        return this.accountDao.findAll();
    }

    @Override
    public void deposit(int number, int amount) {
        if (amount <= 0)
            throw new RuntimeException("bad amount: " + amount);
        final Account account = this.accountDao.get(number);
        account.setBalance(account.getBalance() + amount);
        this.accountDao.update(account);
    }

    @Override
    public void withdraw(int number, int amount) {
        if (amount <= 0)
            throw new RuntimeException("bad amount: " + amount);
        final Account account = this.accountDao.get(number);
        if (amount > account.getBalance())
            throw new RuntimeException("cannot withdraw: " + amount);
        account.setBalance(account.getBalance() - amount);
        this.accountDao.update(account);
    }

    @Override
    public void transfer(int fromNumber, int toNumber, int amount) {
        this.deposit(toNumber, amount);
        this.withdraw(fromNumber, amount);
    }
}
```

Eine @Configuration-basierte Variante

Damit die `@Transactional`-Annotationen bei der Transaktionssteuerung herangezogen werden, muss die `@Configuration`-Klasse zusätzlich annotiert sein mit `@EnableTransactionManagement`:

```
package appl;
// ...
@Configuration
@PropertySource("classpath:db.properties")
@EnableTransactionManagement
public class ApplConfig {

    @Value("${db.driver}")
    private String driver;

    @Value("${db.url}")
    private String url;
```

```
    @Value("${db.user}")
    private String user;

    @Value("${db.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        Log.log();
        return new SimpleDataSource(
            this.driver, this.url, this.user, this.password);
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        return new DataSourceTransactionManager(this.dataSource());
    }

    @Bean
    public AccountDao accountDao() {
        final AccountDaoImpl dao = new AccountDaoImpl();
        dao.setDataSource(this.dataSource());
        return dao;
    }
    @Bean
    public AccountService accountService() {
        return new AccountServiceImpl(this.accountDao());
    }
}
```

11 JPA

Dieses Kapitel demonstriert, wie die JPA-Programmierung von Spring unterstützt wird.

Dabei werden uns viele Mechanismen bereits bekannt vorkommen: sie wurden auch im letzten Kapitel bereits bei der JDBC-Programmierung genutzt.

Hier eine Übersicht zu den folgenden Abschnitten:

- Im Abschnitt 1 wird eine einfache JPA-Anwendung vorgestellt, die ganz ohne auskommt.
- Im Abschnitt 2 wird gezeigt, wie das Template-Muster zur Steuerung der Transaktionen und der `EntityManager` genutzt werden kann.
- Im Abschnitt 3 wird statt des Template-Musters das Proxy-Muster zur Steuerung verwendet.
- Im letzten Abschnitt wird gezeigt, was es mit dem `JpaRepository` auf sich hat: wir müssen die Zugriffsmethoden nicht mehr im implementieren, sondern nurmehr spezifizieren…

11.1 Eine einfache JPA-Anwendung

Im Folgenden bauen wir eine kleine JPA-Anwendung ohne jegliche Spring-Unterstützung –um zunächst einmal zu zeigen, wie JPA eigentlich funktioniert.

Wir benutzen folgende Datenbank:

create.sql

```
create table ACCOUNT (
    NUMBER integer,
    BALANCE integer,
    primary key (NUMBER)
)
```

Zeilen der `ACCOUNT`-Tabelle sollen auf Objekte des Typs `Account` abgebildet werden (und umgekehrt).

Die `Account`-Klasse wird mit `@Entity` annotiert; das Feld, das dem Primärschlüssel der Datenbank-Tabelle repräsentiert (`number`), wird mit `@Id` annotiert; alle anderen Felder, die auf Spalten der Tabelle abgebildet werden sollen, werden mit `@Basic` annotiert (hier: `balance`).

Wir setzen dabei voraus, dass der Tabellenname dem Klassennamen und die Spaltennamen den Attributnamen entsprechen (ansonsten müssten diese Namen via `@Table` resp. `@Column` explizit aufeinander abgebildet werden).

domain.Account

```
package domain;
// ...
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private int number;

    @Basic
    private int balance;

    public Account() {
    }
```

```
    public Account(int number) {
        this.number = number;
    }

    // setter, getter, toString...
}
```

Eine JPA-Anwendung benötigt eine `persistence.xml`. In dieser wird die `persistence-unit` festgelegt, der sog. JPA-Provider (hier: Hibernate), die persistenten Klassen, die JDBC-Verbindungsdaten und einige weitere Properties.

Bei einer Standalone-Anwendung muss diese Konfigurationsdatei im Verzeichnis `META-INF` liegen:

META-INF/persistence.xml

```
<persistence ...>

    <persistence-unit name="bank">

        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <class>domain.Account</class>

        <properties>
            <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.EmbeddedDriver" />
            <property name="javax.persistence.jdbc.url"
                value="jdbc:derby:../dependencies/derby/data" />
            <property name="javax.persistence.jdbc.user"
                value="user" />
            <property name="javax.persistence.jdbc.password"
                value="password" />
            <property name=" javax.persistence.jdbc.dialect"
                value="org.hibernate.dialect.DerbyDialect" />
            <property name="hibernate.show_sql" value="false" />
            <property name="hibernate.format_sql" value="false" />
        </properties>

    </persistence-unit>

</persistence>
```

Und hier eine kleine `Application`, die einige Sätze in die Datenbank einfügt (im Kontext einer Transaktion) und anschließend lesend auf die Tabelle zugreift:

appl.Application

```
package appl;
// ...
import javax.persistence.EntityManager;
```

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

public class Application {
    public static void main(String[] args) throws Exception {
        Db.aroundAppl();

        final EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("bank");

        try {
            demoPersist(factory);
            demoFind(factory);
            demoQuery(factory);
        }
        finally {
            factory.close();
        }
    }
```

Den Einstiegspunkt in eine JPA-Anwendung liefert die statische `Persistence`-Methode `createEntityManagerFactory` (welcher der Name der `persistence-unit` übergeben wird).

Die Methode liest die JPA-Konfiguration ein und erzeugt eine `EntityManagerFactory`. Eine solche Factory fungiert logisch als Singleton; sie enthält einen `Connection`-Pool und ist threadsafe.

Diese Factory wird den drei aufgerufenen `demo`-Methoden übergeben.

Am Ende der Anwendung wird die Factory (und damit alle in ihrem Pool enthaltenen `Connection`s) geschlossen. (`EntityManagerFactory` ist leider nicht `Closeable`).

Die `demoPersist`-Methode fügt einige Konten in die Datenbank ein:

```
    static void demoPersist(EntityManagerFactory factory) {
        final EntityManager manager = factory.createEntityManager();
        final EntityTransaction transaction = manager.getTransaction();
        try {
            transaction.begin();
            manager.persist(new Account(4711));
            manager.persist(new Account(4712));
            manager.persist(new Account(4713));
            transaction.commit();
        }
        catch (final RuntimeException e) {
            System.out.println(e);
            if (transaction.isActive())
                transaction.rollback();
            throw e;
```

```
        }
        finally {
            manager.close();
        }
    }
```

Für die eigentlichen Datenbank-Aktionen benötigen wir einen `EntityManager`. Diesen besorgen wir uns via `createEntityManager` von der Factory.

Hinweis: Sowohl `EntityManagerFactory` als auch `EntityManager` sind nur Interfaces(!).

Ein `EntityManager` bekommt von der Factory eine `Connection` zugewiesen, die er exklusiv nutzen kann. Weil `Connection` nicht threadsafe ist, ist auch ein `EntityManager` nicht threadsafe. Nachdem ein `EntityManager` genutzt wurde, muss er via `close` geschlossen werden (u.a. deshalb, um seine `Connection` in den Pool der Factory zurückzustellen).

Mittels des `EntityManager`s wird zunächst eine `EntityTransaction` erzeugt, die am Ende (bevor der Manager geschlossen wird) via `commit` oder `rollback` terminiert wird.

Der Manager wird dann benutzt, um drei Konten via `persist` zu persistieren. Nach Abschluss der Transaktion wird die `ACCOUNT`-Tabelle also drei Zeilen enthalten.

Die `demoFind`-Methode greift mittels des Primärschlüssels direkt auf eine Zeile der Tabelle zu. JPA erzeugt ein entsprechendes `Account`-Objekt, initialisiert dieses und liefert es zurück.

Der grundsätzlich Aufbau der Methode ist identisch mit demjenigen der `demoPersist`-Methode:

```
    static void demoFind(EntityManagerFactory factory) {
        final EntityManager manager = factory.createEntityManager();
        final EntityTransaction transaction = manager.getTransaction();
        try {
            transaction.begin();
            final Account account = manager.find(Account.class, 4711);
            System.out.println(account);
            transaction.commit();
        }
        catch (final RuntimeException e) {
            System.out.println(e);
            if (transaction.isActive())
                transaction.rollback();
            throw e;
        }
        finally {
            manager.close();
        }
    }
```

Auch die folgende `demoQuery`-Methode ist vom Aufbau her identisch mit dem Aufbau von `demoPersist` resp. `demoFind`. Die Methode benutzt ein `TypedQuery`-Objekt, um alle Zeilen der Tabelle einzulesen. Die `getResultList`-Methode liefert eine Liste von drei `Account`s zurück:

```
    static void demoQuery(EntityManagerFactory factory) {
        final EntityManager manager = factory.createEntityManager();
        final EntityTransaction transaction = manager.getTransaction();
        final List<Account> accountList;
        try {
            transaction.begin();
            String jpql = "select a from Account a";
            final TypedQuery<Account> query =
                manager.createQuery(jpql, Account.class);
            accountList = query.getResultList();
            transaction.commit();
        }
        catch (final RuntimeException e) {
            System.out.println(e);
            if (transaction.isActive())
                transaction.rollback();
            throw e;
        }
        finally {
            manager.close();
        }
        accountList.forEach(System.out::println);
    }
}
```

Resultat: Das "Drumherum" ist in allen drei Methoden identisch. Spring bietet natürlich entsprechende Mittel an, um dieses Drumherum zu abstrahieren. Ebenso wie bei JDBC stellt Spring einen Template- und einen Proxy-Mechanismus zur Verfügung

11.2 TransactionTemplate

Wir benutzen dasselbe DAO-Interface wie im JDBC-Kapitel:

daos.AccountDao

```
package daos;
// ...
public interface AccountDao {
    // siehe: Kapitel "JDBC" Abschnitt "TransactionTemplate"
}
```

Aber natürlich eine andere Implementierung.

Die `AccountDaoImpl`-Klasse definiert eine Instanzvariable `manager` (vom Typ `EntityManager`), die via `setEntityManager` injiziert wird – mittels einer Methode, die mit `@PersistenceContext` annotiert ist (natürlich hätten wir auch Field-Injection verwenden können).

Diese `setEntityManager`-Methode wird nur ein einziges Mal aufgerufen werden – es wird kein "richtiger" `EntityManager` injiziert, sondern ein DynamicProxy, welches ebenfalls das `EntityManager`-Interface implementiert – und an denjenigen realen Manager delegiert, der mit dem aktuellen Thread assoziiert ist.

Die `insert`-Methode der DAO-Klasse delegiert einfach an die `persist`-Method des `EntityManager`s; die `update`-Methode an `merge`; die `delete`-Methode an `merge` und `remove`; die `find`-Methode an `find` und die `findAll`-Methode schließlich an `createQuery` / `getResultList`:

daos.jpa.AccountDaoImpl

```
package daos.jpa;
// ...
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.springframework.dao.DataRetrievalFailureException;

public class AccountDaoImpl implements AccountDao {

    private EntityManager manager;

    @PersistenceContext
    public void setEntityManager(EntityManager manager) {
        Log.log(manager.getClass().getName());
        this.manager = manager;
    }

    @Override
```

```
    public void insert(Account account) {
        this.manager.persist(account);
    }

    @Override
    public void update(Account account) {
        this.manager.merge(account);
    }

    @Override
    public void delete(Account account) {
        this.manager.merge(account);
        this.manager.remove(account);
    }

    @Override
    public Account get(int number) {
        final Account account = this.find(number);
        if (account == null)
            throw new DataRetrievalFailureException(
                "account with number " + number + " not found");
        return account;
    }

    @Override
    public Account find(int number) {
        return this.manager.find(Account.class, number);
    }

    @Override
    public List<Account> findAll() {
        return this.manager.createQuery("select a from Account a",
            Account.class).getResultList();
    }
}
```

Auch das `AccountService`-Interface und dessen Implementierung sind unverändert aus dem JDBC-Kapitel übernommen worden (dem `AccountServiceImpl` wird via Constructor-Injection das DAO injiziert):

services.AccountService

```
package services;
// ...
public interface AccountService {
    // siehe: Kapitel "JDBC" Abschnitt "TransactionTemplate"
}
```

services.AccountServiceImpl

```
package services;
// ...
public class AccountServiceImpl implements AccountService {
```

```
    private final AccountDao accountDao;

    public AccountServiceImpl(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    // siehe: Kapitel "JDBC" Abschnitt "TransactionTemplate"
}
```

Über die `spring.xml` wird eine `LocalEntityManagerFactoryBean` erzeugt. Dann wird ein `JpaTransactionManager` erzeugt, dem die zuvor erzeugte `ManagerFactoryBean` übergeben wird. Dann wird ein `TransactionTemplate` erzeugt, dem der `JpaTransactionManager` übergeben wird.

Und schließlich wird ein DAO-Objekt und ein Service-Objekt erzeugt (wobei letzterem das DAO übergeben wird). Dem DAO wird dabei der delegierende `EntityManager` übergeben - hierzu sind die beiden Einträge `<context.annotation-config>` und `<context.component-scan>` erforderlich.

spring.xml

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

    <context:annotation-config />
    <context:component-scan base-package="daos.jpa"/>

    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    </bean>

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

    <bean id="transactionTemplate"
            class="org.springframework.transaction.support.
                                    TransactionTemplate">
        <constructor-arg ref="transactionManager"/>
    </bean>

    <bean id="accountDao" class="daos.jpa.AccountDaoImpl">
    </bean>

    <bean id="accountService" class="services.AccountServiceImpl">
        <constructor-arg ref="accountDao"/>
    </bean>
```

```
</beans>
```

Die `Application` ist unverändert aus dem JDBC-Kapitel übernommen worden:

appl.Application

```
package appl;
// ...
public class Application {

    // siehe: Kapitel "JDBC" Abschnitt "TransactionTemplate"
}
```

Eine @Configuration-basierte Lösung

Die `spring.xml` kann durch folgende `ApplConfig`-Klasse ersetzt werden:

```
package appl;
// ...
@Configuration
public class ApplConfig {

    @Bean
    public LocalEntityManagerFactoryBean managerFactory() {
        return new LocalEntityManagerFactoryBean();
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        final JpaTransactionManager manager = new JpaTransactionManager();
        manager.setEntityManagerFactory(this.managerFactory().getObject());
        return manager;
    }

    @Bean
    public TransactionTemplate transactionTemplate() {
        return new TransactionTemplate(this.transactionManager());
    }

    @Bean
    public AccountDao accountDao() {
        // don't call setEntityManager -
        // this was done via @PersistenceContext
        return new AccountDaoImpl();
    }

    @Bean
    public AccountService accountService() {
        return new AccountServiceImpl(this.accountDao());
    }
}
```

Man beachte, dass die mit `@PersistenceContext` annotierte DAO-Methode `setEntityManager` nicht explizit aufgerufen werden darf (dies geschieht hinter unserem Rücken) – woher sollten wir auch den delegierenden Manager nehmen?

In der `Application` muss dann natürlich ein `AnnotationConfigApplicationContext` erzeugt werden.

11.3 AutoProxy

Alternativ zur Template-basierten Lösung können wir auch hier wieder die Proxy-Variante verwenden. Alle Service-Methoden werden in der Anwendung auf ein Proxy aufgerufen werden, dessen Klasse das `AccountService`-Interface implementiert.

Dieses Proxy delegiert an einen Spring-eigenen `TransactionInterceptor`, der die Transaktions-Steuerung bereitstellt.

Wir demonstrieren hier nur den AutoProxy-Mechanismus:

spring.xml

```
<beans ...>

    <context:annotation-config />
    <context:component-scan base-package="daos.jpa"/>

    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    </bean>

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

    <bean id="accountDao" class="daos.jpa.AccountDaoImpl">
    </bean>

    <bean id="accountService" class="services.AccountServiceImpl">
        <constructor-arg ref="accountDao" />
    </bean>

    <bean id="transactionInterceptor"
        class="org.springframework.transaction.interceptor
                                    .TransactionInterceptor">
        <property name="transactionManager" ref="transactionManager" />
        <property name="transactionAttributeSource">
            <bean
                class="org.springframework.transaction.interceptor.
                       MatchAlwaysTransactionAttributeSource" />
        </property>
    </bean>

    <bean class="org.springframework.aop.framework.autoproxy.
                                    BeanNameAutoProxyCreator">
        <property name="interceptorNames">
            <list>
                <value>transactionInterceptor</value>
            </list>
```

```
        </property>
        <property name="beanNames">
            <list>
                <value>accountService</value>
            </list>
        </property>
    </bean>

</beans>
```

Jede in der `Application` aufgerufene Service-Methode läuft nun innerhalb einer jeweils neuen Transaktion:

appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) {
        Db.aroundAppl();

        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final AccountService accountService =
                ctx.getBean(AccountService.class);

            System.out.println(accountService.getClass().getName());

            accountService.createAccount(4711);
            accountService.createAccount(4712);
            accountService.deposit(4711, 5000);
            accountService.deposit(4712, 6000);
            accountService.withdraw(4711, 2000);
            accountService.transfer(4711, 4712, 500);

            final List<Account> list = accountService.findAllAccounts();
            list.forEach(System.out::println);
        }
    }
}
```

11.4 Transactional-Annotations

Genauso wie bei der JDBC-Programmierung kann auch bei JPA-Anwendungen die `@Transactional`-Annotation verwendet werden.

Die DAO-Klasse wir mit `@Component` annotiert:

daos.jpa.AccountDaoImpl

```
package daos.jpa;

// ...
import org.springframework.stereotype.Component;

@Component
public class AccountDaoImpl implements AccountDao {

    private EntityManager manager;

    @PersistenceContext
    public void setEntityManager(EntityManager manager) {
        this.manager = manager;
    }

    // ...
}
```

Die `AccountServiceImpl`-Klasse ebenfalls mit `@Component` und zusätzlich mit der `@Transactional(porpagation=REQUIRED)`-Annotation (und die `get`- und `find`-Methoden "überschreiben" diese `@Transactional`-Annotation mit dem `propagation`-Wert `SUPPORTS`):

services.AccountServiceImpl

```
package services;
// ...
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@Component
@Transactional(propagation=Propagation.REQUIRED)
public class AccountServiceImpl implements AccountService {

    private final AccountDao accountDao;

    public AccountServiceImpl(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
```

```
    @Override
    public void createAccount(int number) {
        this.accountDao.insert(new Account(number));
    }

    @Override
    public void deleteAccount(int number) {
        final Account account = this.accountDao.get(number);
        if (account.getBalance() != 0)
            throw new RuntimeException("balance must be 0");
        this.accountDao.delete(account);
    }

    @Override
    @Transactional(propagation=Propagation.SUPPORTS)
    public Account getAccount(int number) {
        return this.accountDao.get(number);
    }

    @Override
    @Transactional(propagation=Propagation.SUPPORTS)
    public Account findAccount(int number) {
        return this.accountDao.find(number);
    }

    @Override
    @Transactional(propagation=Propagation.SUPPORTS)
    public List<Account> findAllAccounts() {
        return this.accountDao.findAll();
    }
    @Override
    public void deposit(int number, int amount) {
        if (amount <= 0)
            throw new RuntimeException("bad amount: " + amount);
        final Account account = this.accountDao.get(number);
        account.setBalance(account.getBalance() + amount);
        this.accountDao.update(account);
    }

    @Override
    public void withdraw(int number, int amount) {
        if (amount <= 0)
            throw new RuntimeException("bad amount: " + amount);
        final Account account = this.accountDao.get(number);
        if (amount > account.getBalance())
            throw new RuntimeException("cannot withdraw: " + amount);
        account.setBalance(account.getBalance() - amount);
        this.accountDao.update(account);
    }

    @Override
    public void transfer(int fromNumber, int toNumber, int amount) {
        this.deposit(toNumber, amount);
        this.withdraw(fromNumber, amount);
    }
}
```

In der `spring-xml` muss nurmehr die `LocalEntityManagerFactoryBean` und der `JpaTransactionManager` eingetragen werden (und natürlich die `context:-` und `tx:-` Einträge):

spring.xml

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

    <context:annotation-config/>
    <tx:annotation-driven/>
    <context:component-scan base-package="daos.jpa"/>
    <context:component-scan base-package="services"/>

    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    </bean>

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

</beans>
```

Auch hier läuft jede in der `Application` aufgerufene Service-Methode nun innerhalb einer jeweils neuen Transaktion:

appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) {
        Db.aroundAppl();

        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final AccountService accountService =
                ctx.getBean(AccountService.class);

            System.out.println(accountService.getClass().getName());

            accountService.createAccount(4711);
            accountService.createAccount(4712);
            accountService.deposit(4711, 5000);
            accountService.deposit(4712, 6000);
            accountService.withdraw(4711, 2000);
            accountService.transfer(4711, 4712, 500);

            final List<Account> list = accountService.findAllAccounts();
            list.forEach(System.out::println);
        }
    }
}
```

11.5 Repositories

Mit Spring können sog. Repositories gebaut werden, die standardmäßig bereits `save`- und `delete`-Methoden enthalten.

Diese default-Methoden, die Spring bereits automatisch bereitstellt, können ergänzt werden durch zusätzliche `find`-Methoden, deren Namen einen spezifischen Aufbau haben müssen: aus den Namen muss exakt hervorgehen, was der entsprechende Aufruf leisten soll.

Diese zusätzlichen Methoden müssen nurmehr in einem Interface spezifiziert werden – für die Implementierung der Methoden ist Spring zuständig (Spring wird zur Laufzeit die entsprechende Implementierungs-Klasse generieren).

In der `spring.xml` findet sich nun ein `jpa:repositories`-Element:

spring.xml

```
<beans ...
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/data/jpa
        http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

    <jpa:repositories base-package="repositories" />

    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean" />

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

</beans>
```

Hier eine beispielhafte Erweiterung des Basis-Interfaces `JpaResository` (eine Klasse, die mit dem Typ der persistenten Klasse und mit dem Typ des `@Id`-Elements dieses Klasse parametrisiert ist):

repositories.AccountRepository

```
package repositories;

// ...
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
```

```
public interface AccountRepository extends JpaRepository<Account, Integer> {

    public abstract List<Account> findByBalance(int balance);

    public abstract List<Account> findByBalanceLessThan(int balance);

    public abstract Account findByNumberAndBalance(int number, int balance);

    @Query("select a from Account a where a.balance > 0")
    public abstract List<Account> findByPositiveBalance();
}
```

Das Interface spezifiziert vier zusätzliche `find`-Methoden:

- Die Methode `findByBalance` liefert eine Liste derjenigen `Account`s, deren `balance` gleich dem Wert des sie übergebenen Parameters ist.
- Die `findByBalanceLessThen` liefert alle `Account`s, deren `balance` kleiner als der Wert des an sie übergebenen Parameters ist.
- Die Methode `findByNumberAndBalance` benutzt als Selektionskriterium die `number` und die `balance`.
- Und die vierte Methode hat einen Namen, dessen Bedeutung für Spring unklar ist. Deshalb muss der zu benutzende Query-String mittels einer `@Query`-Annotation hinterlegt werden.

Hie eine beispielhafte Application (die neben den `find`-Methoden auch die Standard-Methoden `save` und `delete` nutzt):

appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) {

        Db.aroundAppl();
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final AccountRepository repository =
                ctx.getBean(AccountRepository.class);

            System.out.println(repository.getClass().getName());

            final Account a1 = repository.save(new Account(4711));
            final Account a2 = repository.save(new Account(4712));
            final Account a3 = repository.save(new Account(4713));
            final Account a4 = repository.save(new Account(4714));
```

```
            final Account a5 = repository.save(new Account(4715));

            a1.setBalance(5000);
            repository.save(a1);

            a2.setBalance(10000);
            repository.save(a2);

            a3.setBalance(-2000);
            repository.save(a3);

            repository.delete(a4);

            System.out.println("\nfindAll");
            repository.findAll().forEach(System.out::println);

            System.out.println("\nfindByNumber");
            System.out.println(repository.findOne(4712));

            System.out.println("\nfindByBalance");
            repository.findByBalance(5000)
                .forEach(System.out::println);

            System.out.println("\nfindByBalanceLessThan");
            repository.findByBalanceLessThan(5000)
                .forEach(System.out::println);

            System.out.println("\nfindByNumberAndBalance");
            System.out.println(
                 repository.findByNumberAndBalance(4711, 5000));
            System.out.println(
                 repository.findByNumberAndBalance(4711, 500000));

            System.out.println("\nfindByPositiveBalance");
            repository.findByPositiveBalance()
                 .forEach(System.out::println);        }
    }
}
```

Die Ausgaben:

```
com.sun.proxy.$Proxy23

findAll
domain.Account [4711, 5000]
domain.Account [4712, 10000]
domain.Account [4713, -2000]
domain.Account [4715, 0]

findByNumber
domain.Account [4712, 10000]

findByBalance
domain.Account [4711, 5000]
```

```
findByBalanceLessThan
domain.Account [4713, -2000]
domain.Account [4715, 0]

findByNumberAndBalance
domain.Account [4711, 5000]
null

findByPositiveBalance
domain.Account [4711, 5000]
domain.Account [4712, 10000]
```

12 Exporters

Spring vereinfacht und vereinheitlicht den Umgang mit zahlreichen weiteren Java-Mechanismen. Zu diesem Zweck stellt Spring sog. Exporter zur Verfügung.

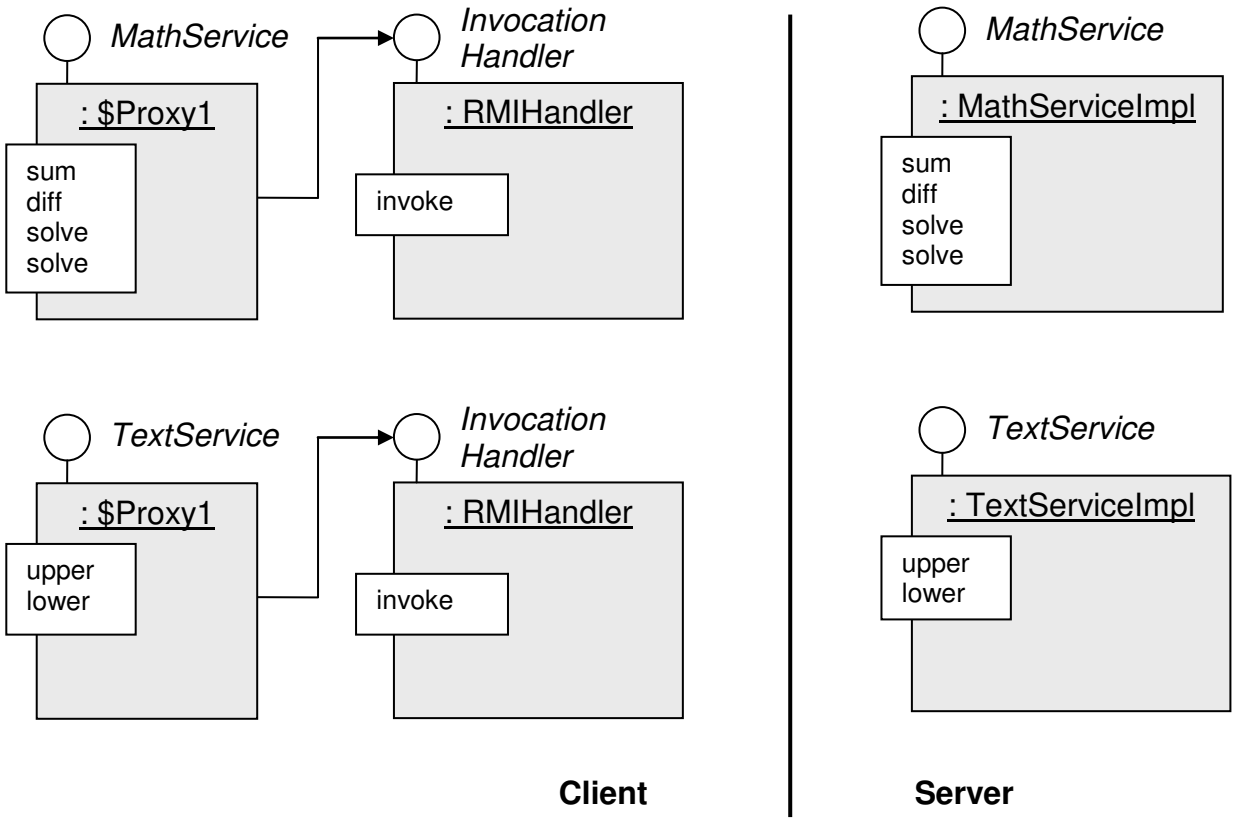
- Im ersten Abschnitt geht's um den RMI-Exporter.
- Der zweite Abschnitt demonstriert den HTTP-Exporter.
- Im dritten Abschnitt wird der WebService-Exporter vorgestellt.
- Im vierten Abschnitt wird ein JMS-Exporter vorgestellt.
- Und im letzten Abschnitt geht’s um den MBean-Exporter.

Die Voraussetzung für das Verständnis dieses Kapitels ist die ungefähre Vorstellung davon, wie RMI-, HTTP-, WS-, JMS- und MBean-Anwendungen ohne Nutzung der Spring-Exporter implementiert werden können.

Zu jedem der folgenden Abschnitte existieren i.d.R. zwei Eclipse-Projekte: ein Server- und ein Client-Projekt. Die `main`-Methoden sind jeweils in den Klassen `server.Server` resp. `client.Client` enthalten.

12.1 RMI

Die im folgenden entwickelte RMI-Anwendung kann wie folgt skizziert werden (sofern wir nur den RMI-Kern betrachten – und das, was Spring zu dieser Konstruktion hinzufügt, zunächst einmal außer Acht lassen:



Die Aufgabe des RMI-Handlers besteht darin, die client-seitigen Methodenaufrufe zu serialisieren und zum Server zu übertragen. Auf der Server-Seite muss dann eine Instanz existieren, welche diese serialisierten Aufrufe entgegennimmt, sie deserialisiert und die entsprechenden Methoden auf das serverseitige Dienst-Objekt aufruft.

Der RMI-Handler und der Reflection-basierte Aufrufmechanimus auf der Server-Seite wird von der RMI-Technologie zur Verfügung gestellt.

Die `solve`-Methode des `MathService`-Interfaces ist überladen. Der ersten `solve`-Methode wird ein `Task`-Objekt übergeben (eine Mathematik-Aufgabe) – sie liefert ein `Result`-Objekt zurück (die Lösung dieser Aufgabe). Der zweiten `solve`-Methode wird eine Liste von `Task`s übergeben; sie liefert eine Liste von `Result`s zurück.

Diese beiden Methoden sollen demonstrieren, dass nicht nur "triviale" Parameter übergeben werden können (bzw. triviale Resultate geliefert werden können), sondern beliebige, serialisierbare Objekte (`Task` und `Result` sind `Serializable`):

dto.Task

```
package dto;

import java.io.Serializable;

public class Task implements Serializable {

    private static final long serialVersionUID = 1L;

    public final int x;
    public final int y;

    public Task(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // toString...
}
```

dto.Result

```
package dto;

import java.io.Serializable;

public class Result implements Serializable {

    private static final long serialVersionUID = 1L;

    public final int sum;
    public final int diff;

    public Result(int sum, int diff) {
        this.sum = sum;
        this.diff = diff;
    }

    // toString...
}
```

Nun zu den Interfaces. Sofern wir direkt mit RMI arbeiten würden, müssten die Interfaces der Dienste abgeleitet werden von dem Marker-Interfaces `java.rmi.Remote`. Und jede der im Interface spezifizierten Methoden muss eine Exception des Typs `java.rmi.RemoteException` ankündigen (eine Exception, die bei technischen Problemen der Kommunikation geworfen wird). Dadurch werden die fachlichen Interfaces natürlich "verschmutzt" von der RMI-Technik. (Hinweis: Die Methoden der

Implementierungsklassen werden natürlich keine `RemoteException` werfen – das Werfen von `RemoteExceptions` ist der RMI-Infrastruktur vorbehalten.)

Spring erlaubt es, ganz gewöhnliche Interfaces zu benutzen (Interfaces, die also weder von `Remote` abgeleitet sind und deren Methoden auch keine `RemoteException`s ankündigen müssen). Aufgrund dieser Interfaces wird Spring zur Laufzeit entsprechende RMI-Interfaces generieren, auf welche die gewöhnlichen Interfaces abgebildete werden.

ifaces.MathService

```
package ifaces;
// ...
public interface MathService {
    public abstract int sum(int x, int y);
    public abstract int diff(int x, int y);
    public abstract Result solve(Task task)
    public abstract List<Result> solve(List<Task> taskList);
}
```

ifaces.TextService

```
package ifaces;
// ...
public interface TextService {
    public abstract String upper(String s);
    public abstract String lower(String s);
}
```

Dieselben(!) `dto`-Klassen `Task` und `Result` und die Interfaces sind sowohl auf der Server- als auch auf der Client-Seite enthalten.

Die Implementierungs-Klassen (die nur auf der Server-Seite benötigt werden):

beans.MathServiceImpl

```
package beans;
// ...
public class MathServiceImpl implements MathService {

    @Override
    public int sum(int x, int y) {
        return x + y;
    }

    @Override
    public int diff(int x, int y) {
        return x - y;
    }
```

```
    @Override
    public Result solve(Task task) {
        return resultOfTask(task);
    }

    @Override
    public List<Result> solve(List<Task> taskList) {
        final List<Result> resultList = new ArrayList<Result>();
        for (final Task task : taskList)
            resultList.add(resultOfTask(task));
        return resultList;
    }

    private static Result resultOfTask(Task task) {
        return new Result(task.x + task.y, task.x - task.y);
    }
}
```

beans.TextServiceImpl

```
package beans;
// ...
public class TextServiceImpl implements TextService {

    @Override
    public String upper(String s) {
        return s.toUpperCase();
    }

    @Override
    public String lower(String s) {
        return s.toLowerCase();
    }
}
```

Sofern wir direkt mit der RMI-Technologie arbeiten würden, müssten wir im Server die Dienst-Objekte erzeugen und in der sog. RMI-Registry eintragen.

Spring vereinfacht diese Registrierung. Die Dienste, die der Server anbietet, können wir nun über die Spring-Konfiguration definieren:

spring.xml

```
<beans ...>

    <bean id="mathService" class="beans.MathServiceImpl"/>
    <bean id="textService" class="beans.TextServiceImpl"/>

    <bean class="org.springframework.remoting.rmi.RmiServiceExporter">
        <property name="serviceName" value="mathService" />
        <property name="service" ref="mathService" />
        <property name="serviceInterface" value="ifaces.MathService" />
        <property name="registryPort" value="1099" />
```

```
    </bean>

    <bean class="org.springframework.remoting.rmi.RmiServiceExporter">
        <property name="serviceName" value="textService" />
        <property name="service" ref="textService" />
        <property name="serviceInterface" value="ifaces.TextService" />
        <property name="registryPort" value="1099" />
    </bean>

</beans>
```

Die `RmiServiceExporter`-Property `serviceName` wird initialisiert mit dem Namen des Dienstes (denjenigen Namen, welcher der Client in seinem Lookup verwendet). Die `service`-Property wird initialisiert mit einer Referenz auf das Dienst-Objekt (`MathServiceImpl`, `TextServiceImpl`). `serviceInterface` wird initialisiert mit dem Namen des Interfaces. Und `registryPort` wird schließlich initialisiert mit der Nummer des Ports, auf welchem der Dienst zur Verfügung stehen wird.

Für den eigentlichen Server bleibt kaum etwas zu tun:

server.Server

```
package server;
// ...
public class Server {

    @SuppressWarnings("resource")
    public static void main(String[] args) {
        try {
            System.out.println("Server starts ...");
            new ClassPathXmlApplicationContext("spring.xml");
            System.out.println("Server running ...");
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Wir können uns darauf beschränken, die Spring-Konfiguration zu laden…

Würden wir direkt mit der RMI-Technologie arbeiten, müssten wir im Client einen RMI-Lookup implementieren.

Sofern wir nun aber Spring benutzen, sieht auch die Seite etwas einfacher aus:

spring.xml

```
<beans ...>
    <bean id="mathService"
        class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
```

```
        <property name="serviceUrl"
            value="rmi://localhost:1099/mathService" />
        <property name="serviceInterface" value="ifaces.MathService" />
    </bean>
    <bean id="textService"
        class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
        <property name="serviceUrl"
            value="rmi://localhost:1099/textService" />
        <property name="serviceInterface" value="ifaces.TextService" />
    </bean>
</beans>
```

Eine `RmiProxyFactoryBean` wird ein Dynamic-Proxy liefern, welche das im `serviceInterface`-Eintrag spezifizierte Interface implementiert. Dieses Proxy wird Verbindung aufnehmen mit dem im `serviceUrl`-Eintrag spezifizierten Remote-Dienst.

Hier schließlich der Client selbst:

client.Client

```
package client;
// ...
public class Client {

    public static void main(String[] args) throws Exception {
        final Client client = new Client();
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            client.run(ctx.getBean(MathService.class));
            client.run(ctx.getBean(TextService.class));
        }
    }

    public void run(MathService mathService) {
        System.out.println(mathService.getClass().getName());
        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
        final Result result = mathService.solve(new Task(55, 11));
        System.out.println(result);
        final List<Task> taskList = new ArrayList<Task>();
        taskList.add(new Task(100, 1));
        taskList.add(new Task(100, 2));
        taskList.add(new Task(100, 3));
        final List<Result> resultList = mathService.solve(taskList);
        System.out.println(resultList);
    }

    public void run(TextService textService) {
        System.out.println(textService.getClass().getName());
        System.out.println(textService.upper("hello"));
        System.out.println(textService.lower("WORLD"));
    }
}
```

Hier die Client-seitigen Ausgaben:

```
com.sun.proxy.$Proxy4
42
77
Result [66, 44]
[Result [101, 99], Result [102, 98], Result [103, 97]]
com.sun.proxy.$Proxy5
HELLO
world
```

12.2 Http

Spring stellt eine eigene Servlet-Klasse für die Bedienung des HTTP-Protokolls zur Verfügung. Weiterhin vereinfacht Spring natürlich die Implementierung eines HTTP-Client.

Als Server wird im folgenden Beispiel Tomcat verwendet. In das `lib`-Verzeichnis von Tomcat sind die benötigten `jar`-Dateien von Spring kopiert worden.

Wir benutzen `ant` zum Deployment der Server-Anwendung (`build.xml`).

Sowohl auf der Client- als auch auf der Server-Seite existieren wieder dieselben Interfaces wie im RMI-Abschnitt: `ifaces.MathService` und `ifaces.DiffService`.

Und auf der Server-Seite existieren auch wieder exakt dieselben Implementierungen dieser Interfaces wie im RMI-Beispiel: `beans.MathServiceImpl` und `beans.TextServiceImpl`.

In der `web.xml` wird das Spring-eigene `DispatcherServlet` eingetragen; zusätzlich bedarf es eines `ContextLoaderServlets`:

WEB-INF/web.xml

```
<web-app ...>

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

    <servlet>
        <servlet-name>DispatcherServlet</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>DispatcherServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Wir benötigen weiterhin eine Konfigurations-Datei für das `DispatcherServlet`:

WEB-INF/DispatcherServlet-servlet.xml

```
<beans ...>

    <bean id="mathService" class="beans.MathServiceImpl"/>
    <bean id="textService" class="beans.TextServiceImpl"/>

    <bean name="/mathServiceHtml"
            class="org.springframework.remoting.httpinvoker.
                                HttpInvokerServiceExporter">
        <property name="service" ref="mathService"/>
        <property name="serviceInterface" value="ifaces.MathService"/>
    </bean>

    <bean name="/textServiceHtml"
            class="org.springframework.remoting.httpinvoker.
                                HttpInvokerServiceExporter">
        <property name="service" ref="textService"/>
        <property name="serviceInterface" value="ifaces.TextService"/>
    </bean>

</beans>
```

Und eine leere `applicationContext.xml`:

WEB-INF/applicationContext.xml

```
<beans ...>

</beans>
```

(Die `"mathService"`- und die `"textService"`-Registrierungen hätten wir statt in der ersten auch in dieser zweiten Konfigurationsdatei hinterlegen können.)

Nun zum Client – er sieht exakt genauso aus wie im RMI-Beispiel….

Technischer Hinweis: Der Client setzt `POST`-Requests ab. Die Parameter des Requests werden im Body des Requests resp. im Body der Responses in Java-serialisierter Form übertragen (daher müssen auch hier die Klassen `Task` und `Result` das Interface Serializable implementieren).

12.3 WebServices

Zunächst zur Server-Seite.

Auf der Server-Seite existiert eine `Task`-Klasse, die mit `@XmlAccossorType` annotiert ist:

server.dto.Task

```
package server.dto;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;

@XmlAccessorType(XmlAccessType.FIELD)
public class Task {

    private int x;
    private int y;

    // getter, setter, toString...
}
```

Und es existiert eine `Result`-Klasse, die ebenfalls entsprechend annotiert ist:

server.dto.Result

```
package server.dto;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;

@XmlAccessorType(XmlAccessType.PROPERTY)
public class Result {

    private int sum;
    private int diff;

    public Result() {  // required for Spring
    }
    public Result(int sum, int diff) { // used in MathService
        this.sum = sum;
        this.diff = diff;
    }

    // getter, setter, toString...
}
```

Auf der Server-Seite existiert zusätzlich der `MathService`. Die `MathService`-Klasse ist u.a. mit `@WebService` annotiert; die Methoden der Klasse mit `@WebMethod`. Diese Klasse ist nicht(!) über ein Interface spezifiziert. Sie enthält einige Methoden mit primitiven

Parametern und Rückgaben; sie enthält aber Methoden, denen `Task`s übergeben werden und die `Result`s liefern:

server.MathService

```
package services;
// ...
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

import server.dto.Result;
import server.dto.Task;

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC, parameterStyle =
SOAPBinding.ParameterStyle.WRAPPED)
public class MathService {

    @WebMethod
    public int sum(int x, int y) {
        return x + y;
    }

    @WebMethod(operationName = "difference")
    public int diff(int x, int y) {
        return x - y;
    }

    @WebMethod
    public Result solve(Task task) {
        return resultOfTask(task);
    }

    @WebMethod
    public Result[] solveArray(Task[] tasks) {
        final Result[] results = new Result[tasks.length];
        for (int i = 0; i < tasks.length; i++)
            results[i] = resultOfTask(tasks[i]);
        return results;
    }

    private static Result resultOfTask(Task task) {
        return new Result(
            task.getX() + task.getY(), task.getX() - task.getY());
    }
}
```

In der `main`-Methode des Servers wird ein `SimpleJaxWsServiceExporter` angefordert (aber nicht weiter genutzt…). Das ist alles:

server.Server

```
package server;
// ...
import org.springframework.remoting.jaxws.
               SimpleJaxWsServiceExporter;

public class Server {
    public static void main(String[] args) throws Exception {

        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            System.out.println("Server starts ...");
            final SimpleJaxWsServiceExporter exporter =
                ctx.getBean(SimpleJaxWsServiceExporter.class);
            System.out.println("Server running ...");
            System.out.println("type ENTER to schutdown server....");
            System.in.read();
            exporter.destroy();
            System.out.println("Server terminated!");
        }
    }
}
```

In der `spring.xml` wird der `SimpleJaxWsServiceExporter` registriert und mittels des Ports, auf dem er lauschen soll, initialisiert. Zudem wird natürlich der `MathService` registriert:

spring.xml

```
<beans ...>

    <bean id="exporter" class="org.springframework.remoting.jaxws.
                                SimpleJaxWsServiceExporter">
        <property name="baseAddress" value="http://localhost:9999/services" />
    </bean>

    <bean
        class="services.MathService">
    </bean>
</beans>
```

Nun zur Client-Seite. Zunächst die `client.dto`-Klassen. Diese sind nicht identisch mit den Server-seitigen Klassen – aber strukturgleich:

client.dto.Task

```
package client.dto;

public class Task {

    private int x;
    private int y;

    public Task() {  // required for Spring
```

```
    }
    public Task(int x, int y) { // convenience for Application
        this.x = x;
        this.y = y;
    }

    // getter, setter, toString...
}
```

client.dto.Result

```
package client.dto;

public class Result {

    private int sum;
    private int diff;

    // only the default-constructor is required

    // getter, setter, toString...
}
```

Auf der Client-Seite wird zudem ein Interface definiert – namens `MathService`. Das Interface wird mit `@WebService` und deren Methoden mit `@WebMethod` ausgestattet (genauso wie die Server-seitige Implementierungsklasse). Das Interface enthält für jede der vom Service angebotenen Methoden eine entsprechend äquivalente Methode.

Mit der Implementierung dieser Methode haben wir nichts zu schaffen – Spring wird zur Laufzeit eine entsprechende Implemtierungsklasse erzeugen (lassen).

client.MathService

```
package client;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

import client.dto.Result;
import client.dto.Task;

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC, parameterStyle =
SOAPBinding.ParameterStyle.WRAPPED)
public interface MathService {
    @WebMethod
    public abstract int sum(int x, int y);
    @WebMethod
    public abstract int difference(int x, int y);
    @WebMethod
    public abstract Result solve(Task task);
    @WebMethod
```

```
    public abstract Result[] solveArray(Task[] tasks);
}
```

In der `spring.xml` wird eine `JaxWsPortProxyFactoryBean` registriert und initialisiert – unter dem Namen `"mathService"`. Die FactoryBean liefert in unserem Fall ein Objekt der von Spring erzeugten Implementierungsklasse des `MathService`-Interfaces (siehe die Initialisierung der `serviceInterface`-Property):

spring.xml

```
<beans ...>

    <bean id="mathService" class="org.springframework.remoting.jaxws.
                                  JaxWsPortProxyFactoryBean">
        <property name="serviceInterface" value="client.MathService" />
        <property name="wsdlDocumentUrl"
            value="http://localhost:9999/services?wsdl" />
        <property name="namespaceUri" value="http://services/" />
        <property name="serviceName" value="MathServiceService" />
        <property name="portName" value="MathServicePort" />
    </bean>
</beans>
```

Der Client kann sich via `getBean` eine `MathService`-Referenz besorgen und diese dann genauso nutzen, als handele es sich um einen lokalen `MathService`:

client.Client

```
package client;
// ...
import client.dto.Result;
import client.dto.Task;

public class Client {
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {

            final MathService mathService =
                ctx.getBean(MathService.class);

            System.out.println(mathService.getClass().getName());

            final int sum = mathService.sum(40, 2);
            System.out.println(sum);

            final int diff = mathService.difference(80, 3);
            System.out.println(diff);

            final Result result = mathService.solve(new Task(55, 11));
            System.out.println(result);
```

```
            final Task[] tasks = new Task[] {
                new Task(100, 1), new Task(100, 2), new Task(100, 3)
            };
            final Result[] results = mathService.solveArray(tasks);
            System.out.println(results);
        }
    }
}
```

12.4 JMS

Dieses Kapitel demonstriert die Unterstützung des JMS (Java Message System).

JMS kennt sog. Queues (die das point-to-point-Prinzip implementieren) und sog. Topics (die das publish-subcribe-Prinzip implementieren). Im folgenden Beispiel beschränken wir und auf Queues.

Als Message-Queue wird die Apache-Implementierung ActiveMQ benutzt

Zunächst muss die Message-Queue gestartet werden:

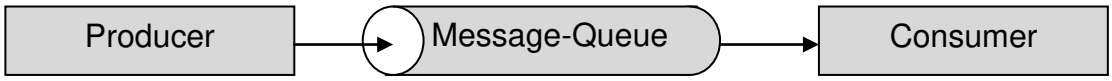
`dependencies / apache-activemq-5.10.0 / bin / win64 / activemq.bat`

Wir können die Message-Queue auch überwachen: http://localhost:8161/admin/

Als User und auch als Password wird "admin" eingegeben.

Es gibt ein `producer` und ein `consumer`-Package. Beide packages enthalten jeweils eine startbare Klasse (`Producer`, `Consumer`).

Der `Producer` stellt jeweils vier Messages in der Message-Queue ab; der `Consumer` liest diese Messages aus und protokolliert sie auf der Console:



Würden wie die Klassen ohne Spring implementieren, müssten beide Klassen sich mit einer Menge von Objekten herumplagen:

- eine `ConnectionFactory`
- eine `Connection`
- eine `Session`
- eine `Destination`
- einen `MessageProducer` resp. `MessageConsumer`

Spring stellt ein `JmsTemplate` zur Verfügung, welches die Implementierung sowohl des Producers als auch des Consumers wesentlich vereinfacht.

In der `spring.xml` wird eine `ActiveMQConnectionFactory` registriert und u.a. mit dem Port initialisiert, der für die `Connection` genutzt wird.

Weiterhin wird ein `JmsTemplate` registriert und mit dem Verweis auf die `ConnectionFactory` und dem Namen der Queue registriert:

spring.xml

```
<beans ...>

    <bean id="jmsFactory"
            class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL">
            <value>tcp://localhost:61616</value>
            <!-- <value>failover://tcp://localhost:61616</value> -->
        </property>
    </bean>

    <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory" ref="jmsFactory" />
        <property name="defaultDestinationName" value="QUEUE-A" />
    </bean>

</beans>
```

Der `Producer` nutzt das `JmsTemplate` (welches dem Entwickler die Erzeugung der o.g. erforderlichen Objekte abnimmt). Er ruft auf das `JmsTemplate` einfach die `send`-Methode auf, der eine `Function` übergeben. Mittels des Inputs der `Function` (`session`) kann dann die zu sendende Nachricht erzeugt werden und als Resultat der `Function` zurückgeliefert werden.

producer.Producer

```
package producer;
// ...
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

public class Producer {

    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            System.out.println("Producer starts");
            final JmsTemplate template =
                ctx.getBean(JmsTemplate.class);

            final String[] lines = new String[] {
                "Spring", "Summer", "Autumn", "Winter"
            };
            for (final String line : lines) {
                template.send((Session session) -> {
```

```
                    System.out.println("Producer <<<< " + line);
                    return session.createTextMessage(line);
                });
                Thread.sleep(2000);
            }
            System.out.println("Producer terminates");
        }
        catch (final Exception e) {
            e.printStackTrace();
        }
    }
}
```

Der `Consumer` nutzt die `JmsTemplate`-Methode `receive`, um auf eine Nachricht zu warten. Trifft eine Nachricht ein, so kehrt `receive` mit der eingetroffenen Nachricht zurück:

consumer.Consumer

```
package consumer;
// ...
import javax.jms.Message;
import javax.jms.TextMessage;
import org.springframework.jms.core.JmsTemplate;

public class Consumer {

    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")){
            System.out.println("Consumer starts");
            final JmsTemplate template =
                ctx.getBean(JmsTemplate.class);
            while (true) {
                System.out.println("Consumer waiting...");
                final Message message = template.receive();
                if (message instanceof TextMessage) {
                    final String text =
                        ((TextMessage) message).getText();
                    System.out.println("Consumer >>>>> " + text);
                }
                else {
                    System.out.println("Consumer >>>>> " + message);
                }
            }
        }
        catch (final Exception e) {
            System.out.println(e);
        }
    }
}
```

Die `Producer`-Ausgaben:

```
Producer starts
Producer <<<< Spring
Producer <<<< Summer
Producer <<<< Autumn
Producer <<<< Winter
Producer terminates
```

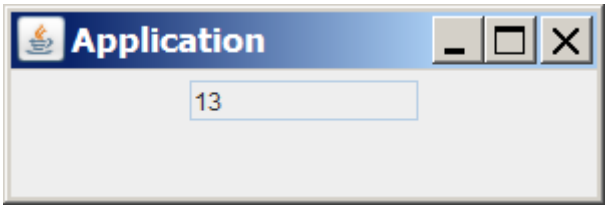
Die `Consumer`-Ausgaben:

```
Consumer starts
Consumer waiting...
Consumer >>>>> Spring
Consumer waiting...
Consumer >>>>> Summer
Consumer waiting...
Consumer >>>>> Autumn
Consumer waiting...
Consumer >>>>> Winter
Consumer waiting...
```

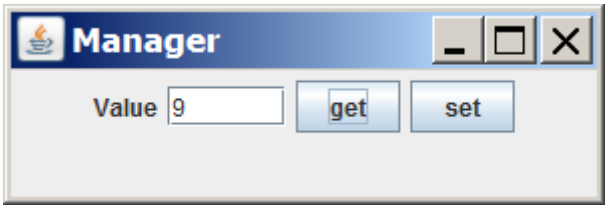
12.5 MBeans

Mittels der MBeans-Technik können Anwendung fernüberwacht und ferngesteuert werden.

Im folgenden entwickeln wir eine Anwendung, die einfach einen Zähler sekündlich hochzählt und diesen dann jeweils auf einer GUI präsentiert. Diese Anwendung soll überwacht und ferngesteuert werden:



Wir entwickeln weiterhin einen Manager, der die Überwachung und Steuerung der ersten Anwendung übernimmt. Mittels des get-Buttons kann der aktuelle Zählerstand der überwachten Anwendung angezeigt werden; mittels des set-Buttons kann er auf einen neuen Wert gesetzt (zurück- oder vorgesetzt) werden.



Die Applikation (ein `JFrame`) läuft in einer Endlosschleife, in welcher jeweils der aktuell angezeigte Wert inkrementiert wird. Sie besitzt eine Methoden, mittels derer der aktuelle Zählerstand ermittelt werden kann (`getValue`); und sie besitzt eine zweite Methode, mittels derer dieser Wert neu gesetzt werden kann (`setValue`):

application.MainFrame

```
package application;
// ...
public class MainFrame extends JFrame {

    private static final long serialVersionUID = 1L;

    private final JTextField textFieldValue = new JTextField(10);

    private int value = 0;

    public MainFrame() {
        super("Application");
        this.setLayout(new FlowLayout());
```

```
        this.add(this.textFieldValue);
        this.textFieldValue.setEditable(false);
        final Thread thread = new Thread(() -> {
            while (true) {
                try {
                    Thread.sleep(1000);
                }
                catch (final InterruptedException e) {
                    throw new RuntimeException(e);
                }
                this.value++;
                this.setValue(this.value);
            }
        });
        thread.setDaemon(true);
        thread.start();
        this.setBounds(600, 100, 300, 100);
        this.setVisible(true);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    synchronized public int getValue() {
        return this.value;
    }

    synchronized public void setValue(int value) {
        this.value = value;
        SwingUtilities.invokeLater(() ->
            this.textFieldValue.setText(
                String.valueOf(this.value)));
    }
}
```

Zur Überwachung resp. Steuerung wird ein Interface definiert, dessen Name mit `MBean` enden muss. Das Interface hat zwei Methoden:

application.DemoMBean

```
package application;

public interface DemoMBean {
    public abstract int getValue();
    public abstract void setValue(int value);
}
```

Die Klasse `Demo` enthält eine Referenz auf die eigentliche Anwendung (`mainFrame`) und implementiert das obige `MBean`-Interface – und zwar derart, dass die jeweils namensgleiche Methode des `MainFrame` aufgerufen wird:

application.Demo

```
package application;
```

```
public class Demo implements DemoMBean {

    private final MainFrame mainFrame;

    public Demo(MainFrame mainFrame) {
        this.mainFrame = mainFrame;
    }

    @Override
    public int getValue() {
        return this.mainFrame.getValue();
    }

    @Override
    public void setValue(int value) {
        this.mainFrame.setValue(value);
    }
}
```

Die `main`-Methode der Application erzeugt eine RMI-Registry und einen Spring-XML-Kontext:

application.Main

```
package application;
// ...
import java.rmi.registry.LocateRegistry;

public class Main {
    @SuppressWarnings("resource")
    public static void main(String[] args) throws Exception {
        LocateRegistry.createRegistry(1099);
        new ClassPathXmlApplicationContext("spring.xml");
        Thread.sleep(Long.MAX_VALUE);
    }
}
```

In der Spring-Konfiguration werden ein `MainFrame` und ein `Demo`-Objekt registriert. Zusätzlich werden ein `MBeanExporter` und eine `ConnectorServerFactoryBean` registriert:

spring.xml

```
<beans ...>

    <bean id="exporter"
            class="org.springframework.jmx.export.MBeanExporter"
            lazy-init="false">
        <property name="beans">
            <map>
                <entry key="jn.abc:type=xyz" value-ref="demo"/>
            </map>
        </property>
```

```
    </bean>

    <bean id="mainFrame" class="application.MainFrame"/>

    <bean id="demo" class="application.Demo">
        <constructor-arg ref = "mainFrame"/>
    </bean>

    <bean id="connector"
            class="org.springframework.jmx.support
                      .ConnectorServerFactoryBean">
        <property name="objectName" value="connector:name=rmi"/>
        <property name="serviceUrl" value=
            "service:jmx:rmi://localhost/jndi/rmi:
            //localhost:1099/logconnector"/>
    </bean>

</beans>
```

Die zu überwachende Anwendung muss nun mit folgenden JVM-Parametern gestartet werden:

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

Soviel zur Applikation, die überwacht und gesteuert werden soll.

Auch auf der Manager-Seite benötigen wir ein Interface `DemoMBean`, welches in seiner Struktur äquivalent sein muss zu demjenigen Interface, welches wir auf der Seite der Application implementiert haben (dieses Interface wird nun aber automatisch zur Laufzeit implementiert werden):

manager.DemoMBean

```
package manager;

public interface DemoMBean {
    public abstract int getValue();
    public abstract void setValue(int value);
}
```

Dem `JFrame` des Managers wird ein `DemoMBean`-Objekt übergeben (via Constructor-Injection). Aufgrund der Betätigung des get- resp. des set-Buttons werden dann die MBean-Methoden `getValue` und `setValue` aufgerufen.

```
package manager;
// ...
public class MainFrame extends JFrame {
```

```
    private static final long serialVersionUID = 1L;

    private final JTextField textFieldValue = new JTextField(5);
    private final JButton buttonGetValue = new JButton("get");
    private final JButton buttonSetValue = new JButton("set");

    final private DemoMBean demoMBean;

    public MainFrame(DemoMBean demoMBean) {
        super("Manager");
        this.demoMBean = demoMBean;
        this.setLayout(new FlowLayout());
        this.add(new JLabel("Value"));
        this.add(this.textFieldValue);
        this.add(this.buttonGetValue);
        this.add(this.buttonSetValue);
        this.buttonGetValue.addActionListener(e ->
            this.textFieldValue.setText(
                String.valueOf(MainFrame.this.demoMBean.getValue())));
        this.buttonSetValue.addActionListener(e ->
            this.demoMBean.setValue(
                Integer.parseInt(this.textFieldValue.getText())));
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setBounds(200, 100, 300, 100);
        this.setVisible(true);
    }
}
```

Die `main`-Methode des Managers besorgt sich von der Spring-Konfiguration eine `MBeanServerConnection` – und mittels dieser Connection dann eine `DemoMBean`:

manager.Main

```
package manager;

import javax.management.JMX;
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
// ...

public class Main {
    public static void main(String[] args) {
        try (final ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            final MBeanServerConnection connection =
                (MBeanServerConnection) ctx.getBean("connector");
            final ObjectName name =
                new ObjectName("jn.abc:type=xyz");
            final DemoMBean demoMBean = JMX.newMBeanProxy(
                connection, name, DemoMBean.class, true);
            new MainFrame(demoMBean);
            Thread.sleep(Long.MAX_VALUE);
        }
        catch (final Exception e) {
            System.err.println(e);
```

```
            System.err.println("Please start Application...");
        }
    }
}
```

In der `spring.xml` des Managers ist nur der `connector` registriert:

spring.xml

```
<beans ...>

    <bean id="connector" class="org.springframework.jmx.support.
                                MBeanServerConnectionFactoryBean">
        <property name="serviceUrl" value=
             "service:jmx:rmi://localhost/jndi/rmi:
                     //localhost:1099/logconnector"/>
    </bean>

</beans>
```

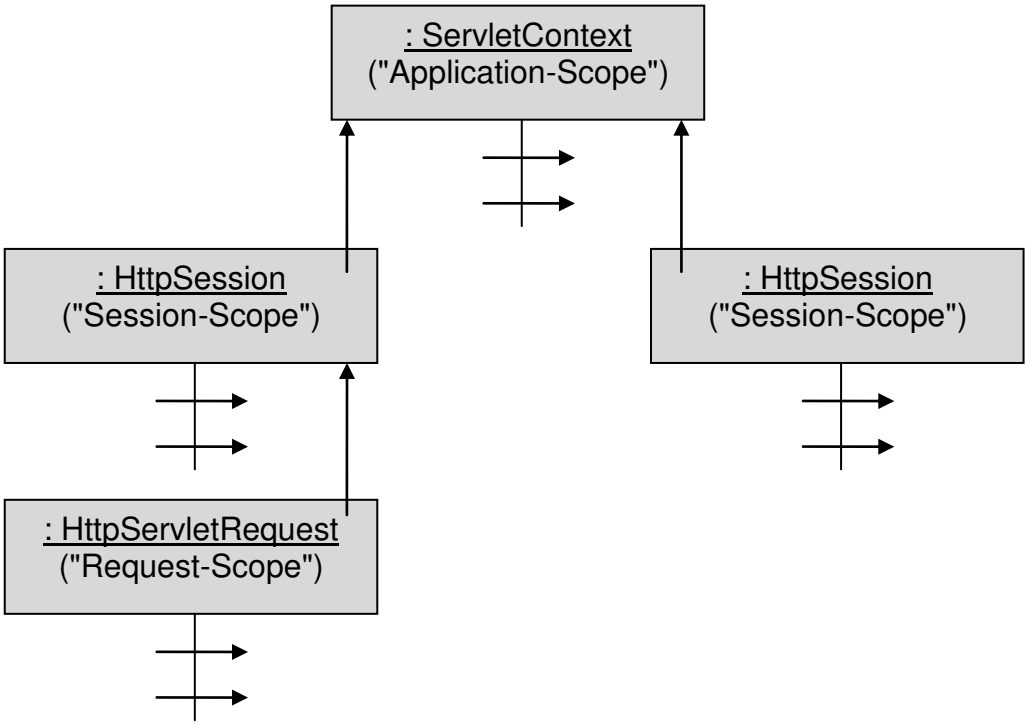
13 Beans in einem Servlet-Container

Dieses Kapitel zeigt, wie die Servlet-Technologie mit Spring verheiratet werden kann.

Mittels des Listener-Konzepts klinkt Spring sich in den Servlet-Container (z.B. Tomcat) ein. Die Spring-Listener wird über das Starten einer Anwendung und über das Eintreffen von Requests informiert. Spring kann dann auf die Scopes der Anwendung zugreifen und in diesen Scopes die gewünschten Beans installieren.

Bekanntlich gibt's drei Scopes: den Application-Scope, den Session-Scope und den Request-Scope. Ein Scope stellt eine Map zur Verfügung, in welcher unter Schlüsseln vom Typ String beliebige Objekte registriert werden können. Für eine Anwendung existiert genau ein einziger Application-Scope; bei Start einer Session wird ein Session-Scope erzeugt, welcher dieser Session exklusiv zugeordnet ist; und beim Eintreffen eines jeden Requests wird ein Request-Scope erzeugt. Der Application-Scope lebt solange wie die Anwendung; ein Session-Scope lebt bis zum Ende der Session, welcher er zugeordnet ist; ein Request-Scope existiert nur während der Bearbeitung des aktuellen Requests.

Ein kleines Schaubild (man beachte dabei, dass `ServletContext`, `HttpSession` und `HttpServletRequest` nur Interfaces sind, die vom jeweiligen Servlet-Container implementiert sind):



Zu den einzelnen Abschnitten dieses Kapitels:

- Im ersten Abschnitt zeigen wir, wie sich Spring (in Form eines `WebApplicationContext`s) in eine Servlet-Anwendung einklinkt.

- Im zweiten Abschnitt zeigen wir, wie Spring aufgrund des `scope`-Attribute der `bean`-Definitionen die benötigen Objekte erzeugt und in den entsprechenden Scopes einträgt.

- Im dritten Abschnitt demonstrieren wird, wie Beans einander referenzieren können: Request-Beans können Session-Beans referenzieren und diese dann Application-Beans.

- Im vierten Abschnitt demonstrieren wir, dass auch langlebige Objekte kurzlebige Objekte referenzieren können (also etwa eine Application-Bean eine Session- oder eine Request-Bean) – obwohl das scheinbar unsinnig ist (aber eben nur scheinbar!). Hierbei werden – wer hätte es gedacht – wiederum Proxies verwendet.

- Im letzten Abschnitt präsentieren wir eine kleine "realistische" Anwendung: ein `MathServlet`. Dieses Servlet kann die Summe jeweils zweier Zahlen berechnen und führt eine Historie dieser Berechnungen. Zusätzlich zeigt das Servlet die Anzahl der bislang erfolgten Besuche an.

Wir stellen zunächst jeweils die XML-Konfiguration vor. Ab und wann wird auch eine Annotations-basierte Lösung vorgestellt.

13.1 WebApplicationContext

Im folgenden wird dargestellt, wie Spring sich in eine Servlet-Anwendung einklinkt.

Wir benötigen einen `WebApplicationContext`, der im `ServletContext` registriert werden muss. Ein solcher `WebApplicationContext` ermöglicht dann die Registrierung von Beans in den Servlet-Scopes. Und mittels dieses Kontext ist dann auch der Zugriff auf diese Beans möglich.

Wie benutzen eine denkbar einfache Bean-Klasse (ohne Interface):

beans.MessageBean

```
package beans;

public class MessageBean {
    public String getMessage() {
        return this.toString()
    }
}
```

Die Spring-Configuration, die bislang stets in einer Datei `spring.xml` hinterlegt wurde, muss nun in einer `applicationContext.xml` hinterlegt werden. Diese muss im `WEB-INF`-Verzeichnis der Anwendung enthalten sein (im selben Verzeichnis, in welchem auch die `web.xml` enthalten ist):

WEB-INF/applicationContext.xml

```
<beans ...>
    <bean id="messageBean"
            class="beans.MessageBean" scope="singleton">
    </bean>
</beans>
```

Die Klasse `MyServlet` zeit, was es mit dem Spring-eigenen `WebApplicationContext` auf sich hat: wie dieser Kontext ermittelt und benutzt werden kann (wie er eingerichtet wird, ist Sache der `web.xml` – siehe weiter unten).

servlets.MyServlet

```
package servlets;

import java.io.IOException;
import java.io.PrintWriter;
@WebServlet(urlPatterns="/*")
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
```

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import org.springframework.web.context.ContextLoader;
import org.springframework.web.context.WebApplicationContext;

import beans.MessageBean;

@WebServlet(urlPatterns="/*")
public class MyServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    public void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {

        response.setContentType("text/plain");
        final PrintWriter out = response.getWriter();

        final HttpSession session = request.getSession();
        final ServletContext servletContext =
            session.getServletContext();
        final String applContextName = WebApplicationContext
                        .ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE;
        final WebApplicationContext ctx1 = (WebApplicationContext)
            servletContext.getAttribute(applContextName);

        final WebApplicationContext ctx2 =
            ContextLoader.getCurrentWebApplicationContext();

        out.println(ctx1 == ctx2);

        final MessageBean bean = ctx1.getBean(MessageBean.class);
        out.println(bean.getMessage());
        out.println(servletContext.getAttribute("messageBean"));
    }
}
```

Die `doGet`-Methode, die beim Eintreffen eines `GET`-Requests vom Servlet-Container aufgerufen wird, ermittelt zunächst – ausgehend vom dem ihr übergebenen `HttpServletRequest` - die `HttpSession`- und die `ServletContext`-Objekte.

Mittels des Schlüssels `appContextName` greift sie dann auf das im `ServletContext` eingetragene `WebApplicationContext`-Objekt zu (`webApplicationContext1`). Wie das `WebApplication` im `ServletContext` eingetragen wird, ist Sache der `web.xml` (siehe weiter unten). Der `WebApplicationContext` ist dann über `ctx1` referenzierbar.

Dann wird die `ContextLoader`-Methode `getCurrentWebApplicationContext` aufgerufen – auch sie liefert eben dieses `WebApplicationContext`-Objekt zurück (`ctx2`).

Beide `webApplicationContext`-Objekte zeigen auf dasselbe Objekt – der Referenzvergleich `ctx1 == ctx2` liefert also `true`.

Schließlich wie die im `WebApplicationContext` registrierte `MessageBean` ermittelt – und auf diese die Methode `getMessage` aufgerufen.

Und zuallerletzt wird ermittelt, ob die Bean auch im Application-Scope (im `ServletContext`) eingetragen wurde. Wie die Ausgabe zeigt, ist dies nicht(!) der Fall.

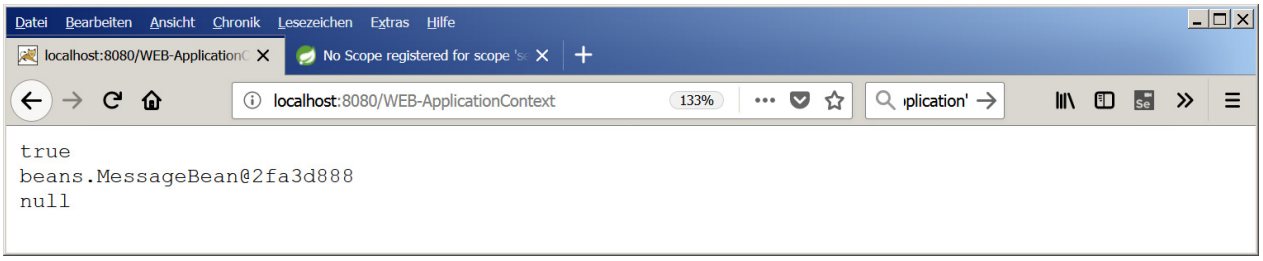
Hier die `web.xml`:

web.xml

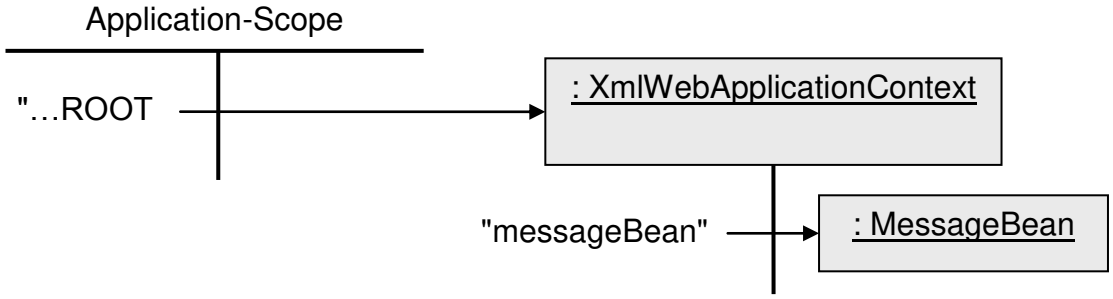
```
<web-app ...>

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

</web-app>
```

In der `web.xml` muss ein `ContextLoaderListener` als Listener registriert werden. Dieser Listener sorgt dafür, dass aufgrund der `applicationContext.xml` ein `XmlWebApplicationContext` erzeugt wird, welcher im Application-Scope eingehängt wird (unter dem Schlüssel `ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE`.

Hie die Ausgaben der obigen Test-Anwendung:



Der `WebApplicaitonContext` ist nun im Application-Scope registriert – und die `MessageBean` im `WebApplicationContext`:



Die Bean ist nur im `WebApplicationContext` registriert - nicht aber im Application-Scope (im `ServletContext`).

Eine Annotations-basierte Variante

Die `applicationContext.xml` beschränkt sich auf die Angabe des Packages, in welchem nach `@Component`s gesucht wird:

```
<beans ...>
    <context:annotation-config/>
    <context:component-scan base-package="beans"/>
</beans>
```

Die Bean-Klasse wird um eine `@Component`- und eine `@Scope`-Annotation erweitert:

```
package beans;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("singleton")
public class MessageBean {
    public String getMessage() {
        return this.toString();
    }
}
```

Die `MyServlet`-Klasse kann unverändert übernommen werden.

Eine WebApplicationInitializer-Variante

Die `web.xml` kann radikal verkürzt werden:

```
<web-app ...>

</web-app>
```

Wir schreiben eine Klasse, die das Spring-Interface `WebApplicationInitializer` implementiert. In der `onStartup`-Methode dieser Methode wird ein Objekt des Typs `AnnotationConfigWebApplication` erzeugt und damit beauftragt, im Package "`beans`" nach annotierten Beans zu suchen:

```
package servlets;

import javax.servlet.ServletContext;
import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.request.RequestContextListener;
import org.springframework.web.context.support
                          .AnnotationConfigWebApplicationContext;

public class MyWebAppicationInitializer
        implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) {

        final AnnotationConfigWebApplicationContext ctx =
            new AnnotationConfigWebApplicationContext();
        ctx.scan("beans");
        ctx.refresh();

        servletContext.addListener(new ContextLoaderListener(ctx));

        final ServletRegistration.Dynamic servlet =
        servletContext.addServlet("MyServlet", new MyServlet());
        servlet.setLoadOnStartup(1);
        servlet.addMapping("/*");
    }
}
```

Zusätzlich wird beim an `onStartup` übergebenen `ServletContext` ein `ContextLoaderListener` installiert (was zuvor in der `web.xml` passierte). Und schließlich wird auch ein `MyServlet` erzeugt und beim `ServletContext` registriert (welches bislang ebenfalls über die `web.xml` eingerichtet wurde).
Dabei wird die Klasse `MyWebApplicationInitializer` automatisch über den Java-eigenen `ServiceLoader`-Mechanismus instanziiert. (Nähere Einzelheiten zu diesem Mechanismus können der Klasse `SpringServletContainerInitializer` entnommen werden.)

13.2 Scopes

Beans, die im Spring-Scope "singleton" eingetragen sind, werden vom Spring-ApplicationContext verwaltet. In einer Spring-Konfiguration können aber auch Beans spezifiziert werden, die die Scopes "application", "session" und "request" besitzen. Solche Beans werden von Spring zwar erzeugt und initialisiert, sie werden aber nicht im ApplicationContext gehalten. Statt dessen werden sie von Spring in die Standard-Scopes des Servlet-Containers eingetragen (Session-Scope resp. Request-Scope).

Ein Objekt der folgenden MyBean-Klasse weiß, wann es erzeugt wurde und in welchem Scope es angesiedelt wurde:

beans.MyBean

```
package beans;
// ...
public class MyBean {

    private final String springScope;
    private final Date dateCreated;

    public MyBean(String springScope) {
        this.springScope = springScope;
        this.dateCreated = new Date();;
    }
    @Override
    public String toString() {
        return Integer.toHexString(System.identityHashCode(this)) +
            " " + this.springScope
            DateFormat.getTimeInstance().format(this.dateCreated);
    }
}
```

In der Spring-Konfiguration werden vier MyBean-Einträge hinterlegt:

WEB-INF/applicationContext.xml

```
<beans ...>

    <bean id="requestBean" class="beans.MyBean" scope="request">
        <constructor-arg value="request" />
    </bean>

    <bean id="sessionBean" class="beans.MyBean" scope="session">
        <constructor-arg value="session" />
    </bean>

    <bean id="applicationBean" class="beans.MyBean" scope="application">
        <constructor-arg value="application" />
    </bean>
```

```
    <bean id="singletonBean" class="beans.MyBean" scope="singleton">
        <constructor-arg value="singleton" />
    </bean>

</beans>
```

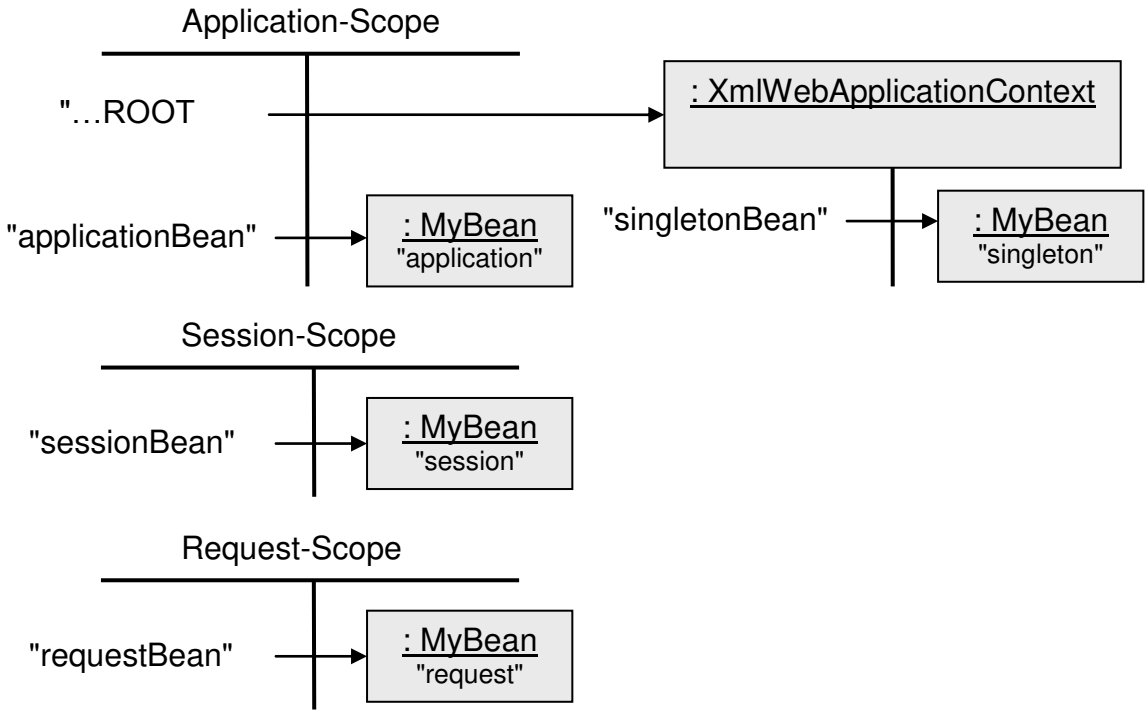
Der "requestBean"-Eintrag führt dazu, dass Spring bei jedem Request eine MyBean-Instanz erzeugt (mit "request" initialisiert wird), und im Request-Scope des Servlet-Containers eingetragen wird.

Die unter "sessionBean" eingetragene Bean-Beschreibung führt dazu, dass zu Beginn jeder Session eine neue MyBean-Instanz erzeugt wird (mit "session" initialisiert wird) und im Session-Scope des Servlet-Containers eingetragen wird.

Die unter "applicationBean" eingetragene Bean-Beschreibung führt dazu, dass beim Starten der Anwendung eine entsprechende Bean-Instanz erzeugt wird und im Application-Scope des Servlet-Containers eingetragen wird.

Die unter "singletonBean" eingetragene Bean-Beschreibung führt dazu, dass beim Starten der Anwendung eine entsprechende Bean-Instanz erzeugt wird und im WebApplicationContext eingetragen wird.

Das folgende Bild veranschaulicht dieses Verhalten:



Damit Spring beim Eintreffen eines Request resp. beim Beginn einer neuen Session entsprechend reagieren kann, muss neben dem `ContextLoaderListener` ein weiterer Listener in der `web.xml` registriert werden: ein `RequestContextListener`.

web.xml

```
<web-app ...>

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

    <listener>
        <listener-class>
            org.springframework.web.context.request
                                .RequestContextListener
        </listener-class>
    </listener>

</web-app>
```

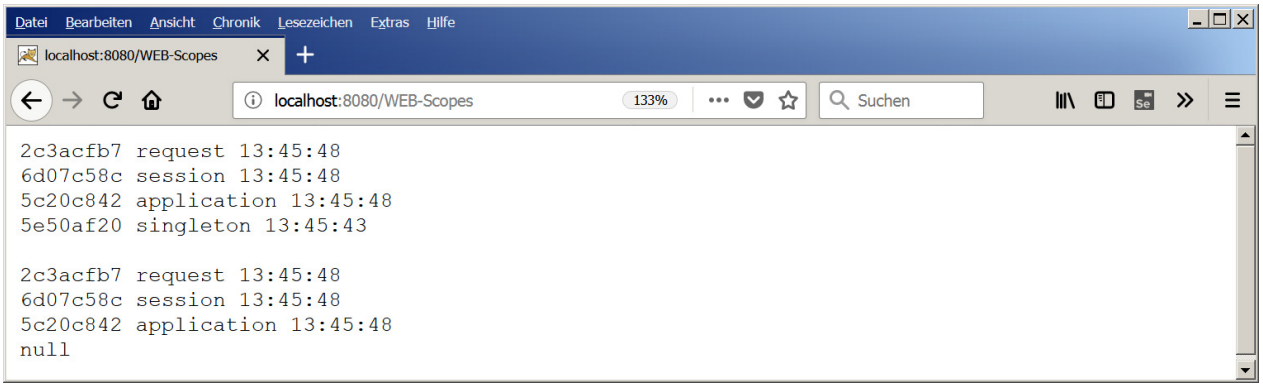
Das folgende Servlet soll die hier dargestellten Zusammenhänge verdeutlichen:

servlets.MyServlet

```
package servlets;
// ...
@WebServlet(urlPatterns="/*")
public class MyServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    public void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {

        final ApplicationContext ctx =
            ContextLoader.getCurrentWebApplicationContext();

        response.setContentType("text/plain");
        final PrintWriter out = response.getWriter();

        out.println(ctx.getBean("requestBean"));
        out.println(ctx.getBean("sessionBean"));
        out.println(ctx.getBean("applicationBean"));
        out.println(ctx.getBean("singletonBean"));

        out.println();

        out.println(request.getAttribute("requestBean"));
```

```
        out.println(request.getSession().getAttribute("sessionBean"));
        out.println(request.getSession().getServletContext()
            .getAttribute("applicationBean"));
        out.println(request.getSession().getServletContext()
            .getAttribute("singletonBean"));
    }
}
```

Hier die Ausgaben:



Alle Beans können über den `WebApplicationContext` ermittelt werden.

Aber nur Beans mit den Scopes `"application"`, `"session"` und `"request"` können über die entsprechenden Attribute-Maps der Standard-Kontexte (`ServletContext`, `HttpSession`, `HttpServletRequest`) ermittelt werden. `"singlton"`-Beans können nur über den Spring-Kontext ermittelt werden.

Eine @Configuration-basierte Variante

Die `applicationContext.xml` beschränkt sich auf die Angabe des Packages, in welchem die mit `@Configuration` ausgezeichnete Konfigurations-Klasse enthalten ist:

```
<beans ...>
    <context:annotation-config/>
    <context:component-scan base-package="config"/>
</beans>
```

Und hier die `@Configuration`-Klasse:

```
package config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;

import beans.MyBean;

@Configuration
public class ApplConfig {

    @Bean(name ="requestBean")
    @Scope("request")
    public MyBean requestBean() {
        return new MyBean("request");
    }

    @Bean(name ="sessionBean")
    @Scope("session")
    public MyBean sessionBean() {
        return new MyBean("session");
    }

    @Bean(name ="applicationBean")
    @Scope("application")
    public MyBean applicationBean() {
        return new MyBean("application");
    }

    @Bean(name ="singletonBean")
    @Scope("singleton")
    public MyBean singletonBean() {
        return new MyBean("singleton");
    }
}
```

13.3 Dependencies

Natürlich kann Spring auch Beans innerhalb eines Servlet-Containers miteinander verdrahten.

Wenn eine `request`-Bean erzeugt wird, kann dieser eine Referenz auf eine `session`- oder auf eine `application`-Bean injiziert werden (oder eine Referenz auf eine andere `request`-Bean); und wenn eine `session`-Bean erzeugt wird, kann ihr eine Referenz auf eine andere `session`-Bean oder auf eine `application`-Bean injiziert werden.

Die folgende Anwendung verwendet drei Bean-Klassen:

beans.SingletonBean

```
package beans;

public class ApplicationBean {

    public ApplicationBean() {
    }
}
```

beans.SessionBean

```
package beans;

public class SessionBean {

    private final ApplicationBean applicationBean;

    public SessionBean(ApplicationBean applicationBean) {
        this.applicationBean = applicationBean;
    }

    public ApplicationBean getApplicationBean() {
        return this.applicationBean;
    }
}
```

Eine `RequestBean` wird in der Spring-Konfiguration als `request`-Bean definiert werden. Wenn Spring beim Starten einer Session (genauer: beim ersten `getBean`-Lookup nach dem Starten einer Session) eine neue `SessionBean` erzeugt, wird er sie per Constructor-Injection mit einer Referenz auf die `ApplicationBean` ausstatten.

beans.RequestBean

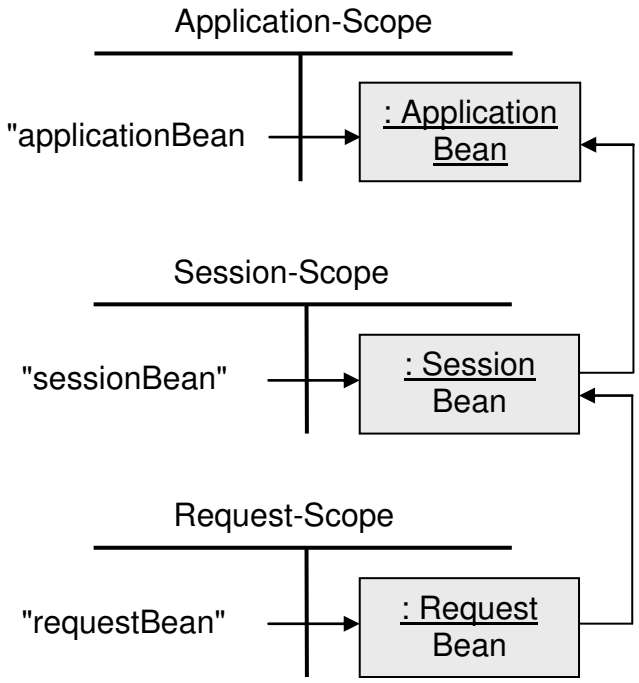
```
package beans;


public class RequestBean {

    private final SessionBean sessionBean;

    public RequestBean(SessionBean sessionBean) {
        this.sessionBean = sessionBean;
    }

    public SessionBean getSessionBean() {
        return this.sessionBean;
    }
}
```

Eine `RequestBean` wird in der Spring-Konfiguration als `request`-Bean definiert werden. Wenn Spring beim Eintreffen eines Requests (genauer: beim ersten `getBean`-Lookup nach Eintreffen eines Requests) eine neue `RequestBean` erzeugt, wird sie per Constructor-Injection mit einer Referenz auf die mit der aktuellen Session assoziierte `SessionBean` ausgestattet.

WEB-INF/applicationContext.xml

```
<beans ...>

    <bean id="applicationBean"
            class="beans.ApplicationBean" scope="application"/>

    <bean id="sessionBean"
            class="beans.SessionBean" scope="session">
        <constructor-arg ref="applicationBean"/>
    </bean>

    <bean id="requestBean"
            class="beans.RequestBean" scope="request">
        <constructor-arg ref="sessionBean"/>
    </bean>
</beans>
```

Das folgende Bild zeigt das Resultat:



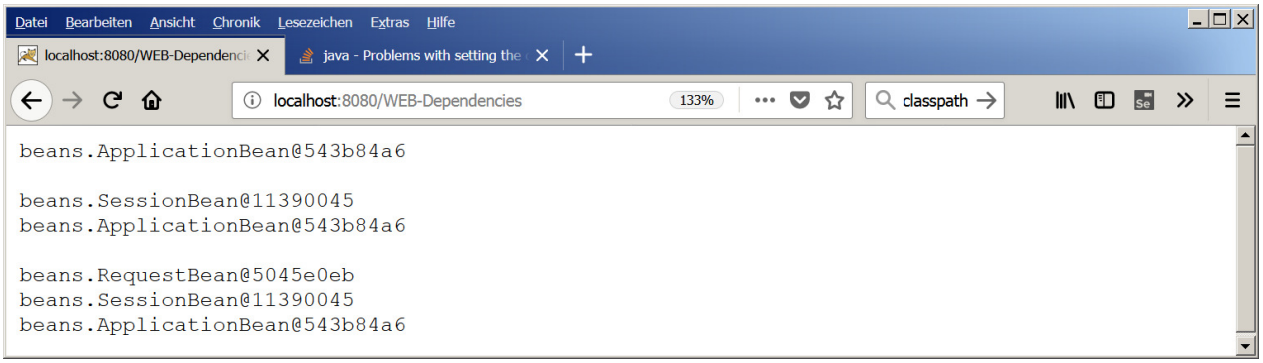
Das folgende Servlet zeigt, dass Spring die Objekte korrekt verknüpft:

servlets.MyServlet

```
package servlets;
// ...
@WebServlet(urlPatterns="/*")
public class MyServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    public void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {

        final ApplicationContext ctx =
            ContextLoader.getCurrentWebApplicationContext();
        response.setContentType("text/plain");
        final PrintWriter out = response.getWriter();

        final ApplicationBean applicationBean =
            ctx.getBean(ApplicationBean.class);
        final SessionBean sessionBean =
            ctx.getBean(SessionBean.class);
        final RequestBean requestBean =
```

```
            ctx.getBean(RequestBean.class);

        out.println(applicationBean);
        out.println();

        out.println(sessionBean);
        out.println(sessionBean.getApplicationBean());
        out.println();

        out.println(requestBean);
        out.println(requestBean.getSessionBean());
        out.println(requestBean.getSessionBean().getApplicationBean());

        ServletUtils.printScopes(System.out, request,
            (k, v) -> k.contains("Bean"));
    }
}
```

Hier die Ausgaben des obigen Servlets:



Auf der Server-Console erscheinen die Ausgaben von `ServletUtils.printScopes`:

```
Application-Scope
        applicationBean ==> ApplicationBean (543b84a6)
Session-Scope
        sessionBean ==> SessionBean (11390045)
Request-Scope
        requestBean ==> RequestBean (5045e0eb)
```

Eine Annotations-basierte Variante

Die Bean-Klassen werden um `@Component`- und `@Scope`-Annotationen erweitert:

```
package beans;
// ...
@Component
@Scope(WebApplicationContext.SCOPE_APPLICATION)
public class ApplicationBean {
    // ...
}
```

```
package beans;
// ...
@Component
@Scope(WebApplicationContext.SCOPE_SESSION)
public class SessionBean {
    // ...
}
```

```
package beans;
// ...
@Component
@Scope(WebApplicationContext.SCOPE_REQUEST)
public class RequestBean {
    // ...
}
```

Die `WEB-INF/applicationContext.xml` beschränkt sich auf die Angabe des zu scannenden Verzeichnisses:

```
<beans ...>
    <context:annotation-config/>
    <context:component-scan base-package="beans"/>
</beans>
```

13.4 Dependencies mit Proxies

Wie im letzten Abschnitt erläutert wurde, können Beans mit einer kurzen Lebensdauer auf Beans mit einer längeren Lebensdauer verweisen (z.B. eine `request`-Bean auf eine `session`-Bean), aber offensichtlich nicht umgekehrt. Dieser Abschnitt zeigt, dass mittels Proxies aber auch scheinbar widersinnige Verbindungen konstruiert werden können.

beans.ApplicationBean

```
package beans;

public class ApplicationBean {

    private final SessionBean sessionBean;

    public ApplicationBean(SessionBean sessionBean) {
        this.sessionBean = sessionBean;
    }

    public SessionBean getSessionBean() {
        return this.sessionBean;
    }
}
```

beans.SessionBean

```
package beans;

public class SessionBean {

    private RequestBean requestBean;

    public void setRequestBean(RequestBean requestBean) {
        this.requestBean = requestBean;
    }

    public RequestBean getRequestBean() {
        return this.requestBean;
    }
}
```

beans.RequestBean

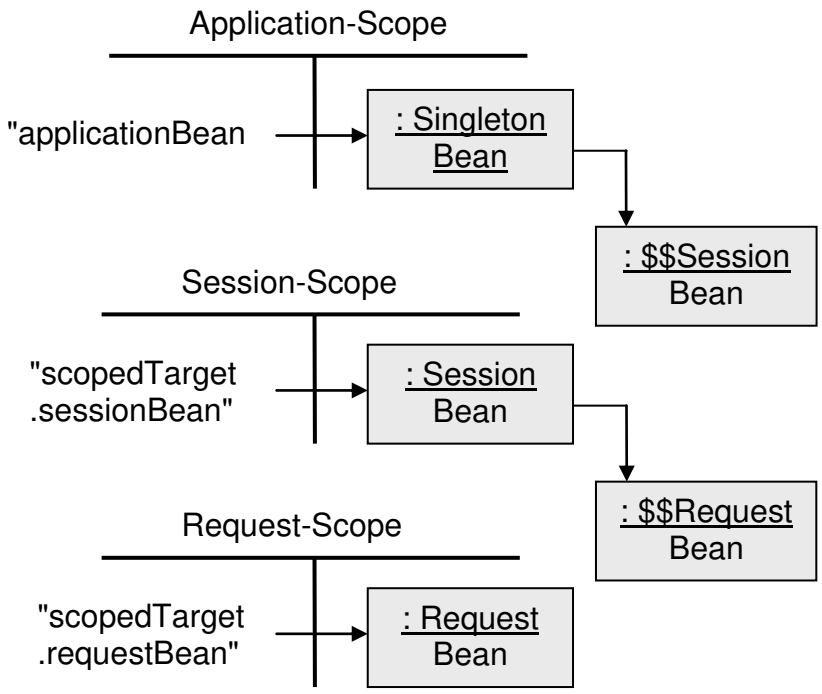
```
package beans;

public class RequestBean {
}
```

applicationContext.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <bean id="requestBean"
            class="beans.RequestBean" scope="request">
        <aop:scoped-proxy/>
    </bean>

    <bean id="sessionBean"
            class="beans.SessionBean" scope="session">
        <constructor-arg ref="requestBean" />
        <aop:scoped-proxy/>
    </bean>

    <bean id="applicationBean"
            class="beans.ApplicationBean" scope="application">
        <constructor-arg ref="sessionBean" />
    </bean>

</beans>
```

Der `application`-Bean wird via Constructor-Injection eine Referenz auf die `session`-Bean zugewiesen; der `session`-Bean wird via Property-Injection eine Referenz auf die `request`-Bean zugewiesen. Man beachte, dass die `request`- und die `session`-Beans jeweils mit dem inneren Element `<aop:scoped-proxy>` versehen sind.

Das folgende Bild verdeutlicht die Zusammenhänge:



Zur Laufzeit werden von der GCLib zwei Proxy-Klassen erzeugt: `$$SessionBean` und `$$RequestBean`. Diese Klassen sind abgeleitet von `SessionBean` resp. `RequestBean`. In die `applicationBean` wird eine Instanz von `$$SessionBean` injiziert; in die `SessionBean` eine Instanz von `$$RequestBean`. Diese Injection findet – wie üblich – nur ein einziges Mal statt (beim ersten Lookup per `getBean`).

Beim Beginn einer neuen Session wird aber natürlich stets ein neues "normales" `SessionBean`-Objekt erzeugt und in den Session-Scope eingetragen. Und beim Eintreffen eines neuen Requests wird ebenfalls natürlich jeweils eine neue "normale" `RequestBean` erzeugt und in den Request-Scope des Servlet-Containers eingetragen.

Zur Laufzeit wird es also für jede laufende Session genau eine `SessionBean` geben; und für jeden aktuellen Requests wird es eine `RequestBean` geben. Aber es gibt nur eine einzige `$$SessionBean` (die in der `AppicationBean` eigetragen ist) und pro laufender Session nur eine einzige `$$RequestBean` – eine Bean, die in der `SessionBean` eingetragen ist. Die `$$...Beans` können also keinerlei feste Verbindungen zu den den entsprechenden "normalen" Beans haben.

Wird nun über das `ApplicationBean`-Objekt eine Methode auf das mit dieser Bean assoziierte `$$SessionBean`-Objekt aufgerufen, so wird diese Methode die mit dem aktuellen Request verbundene Session ermitteln und über diese Session das "normale"

`SessionBean`-Objekt. Der Methodenaufruf wird dann an diese "normale" `SessionBean` weitergeleitet.

Und wird über eine `SessionBean` eine Methode auf das mit ihr assoziierte `$$RequestBean`-Objekt aufgerufen, so wird diese Methode den aktuellen Request und über diesen Request das "normale" mit dem Request verbundene `RequestBean`-Objekt ermitteln – und auch hier den Methodenaufruf an dieses Objekt weiterleiten.

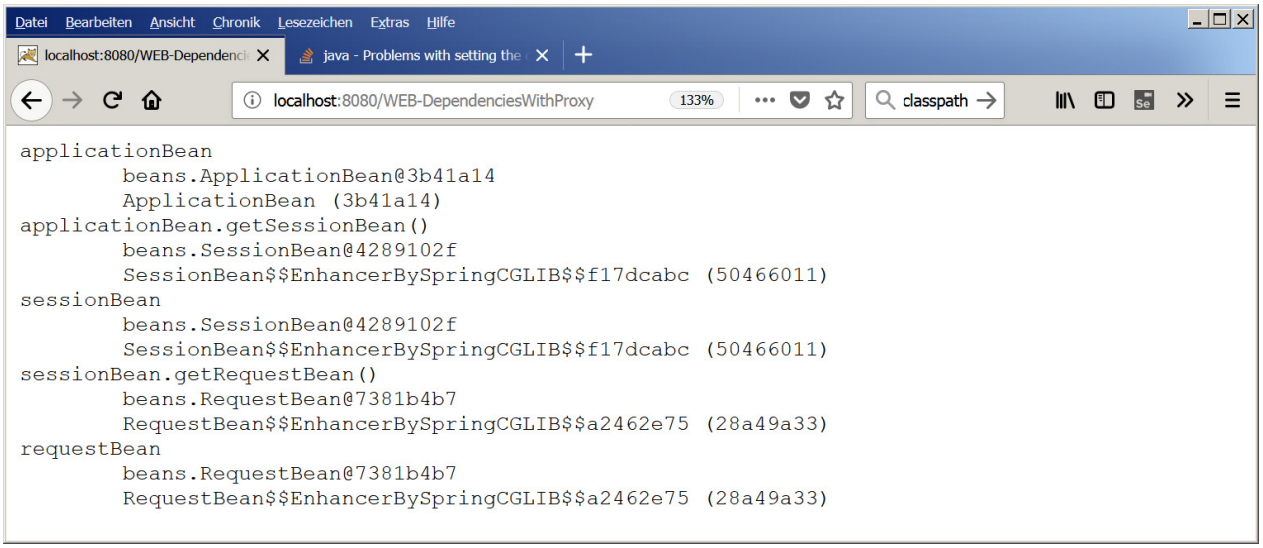
Hier die Test-Anwendung:

servlets.MyServlet

```
package servlets;
// ...
@WebServlet(urlPatterns="/*")
public class MyServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    public void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {

        response.setContentType("text/plain");
        final PrintWriter out = response.getWriter();

        final ApplicationContext ctx =
            ContextLoader.getCurrentWebApplicationContext();

        final ApplicationBean applicationBean =
            ctx.getBean(ApplicationBean.class);
        final SessionBean sessionBean =
            ctx.getBean(SessionBean.class);
        final RequestBean requestBean =
            ctx.getBean(RequestBean.class);

        out.println("applicationBean");
        out.println("\t" + applicationBean);
        out.println("\t" + Utils.identityOf(applicationBean));

        out.println("applicationBean.getSessionBean()");
        out.println("\t" + applicationBean.getSessionBean());
        out.println("\t" +
            Utils.identityOf(applicationBean.getSessionBean()));

        out.println("sessionBean");
        out.println("\t" + sessionBean);
        out.println("\t" + Utils.identityOf(sessionBean));

        out.println("sessionBean.getRequestBean()");
        out.println("\t" + sessionBean.getRequestBean());
        out.println("\t" +
```

```
            Utils.identityOf(sessionBean.getRequestBean()));

        out.println("requestBean");
        out.println("\t" + requestBean);
        out.println("\t" + Utils.identityOf(requestBean));

        ServletUtils.printScopes(System.out, request,
            (k, v) -> k.contains("Bean"));
    }
}
```

Hier die Client-Ausgaben:



Auf der Server-Console erscheinen die Ausgaben von `ServletUtils.printScopes`:

```
Application-Scope
        applicationBean ==> ApplicationBean (3b41a14)
Session-Scope
        scopedTarget.sessionBean ==> SessionBean (4289102f)
Request-Scope
        scopedTarget.requestBean ==> RequestBean (7381b4b7)
```

Sofern es sich bei den Bean-Klassen um Klassen handeln würde, die über ein Interface spezifiziert wären, würde statt der CGLib der übliche DynamicProxy-Mechanismus verwendet, um die Proxy-Klassen zu generieren.

Eine Annotations-basierte Variante

Die Bean-Klassen werden um eine `@Component`- und eine `@Scope`-Annotation erweitert:

```
package beans;

@Component
@Scope(WebApplicationContext.SCOPE_APPLICATION)
public class ApplicationBean {
    // ...
}
```

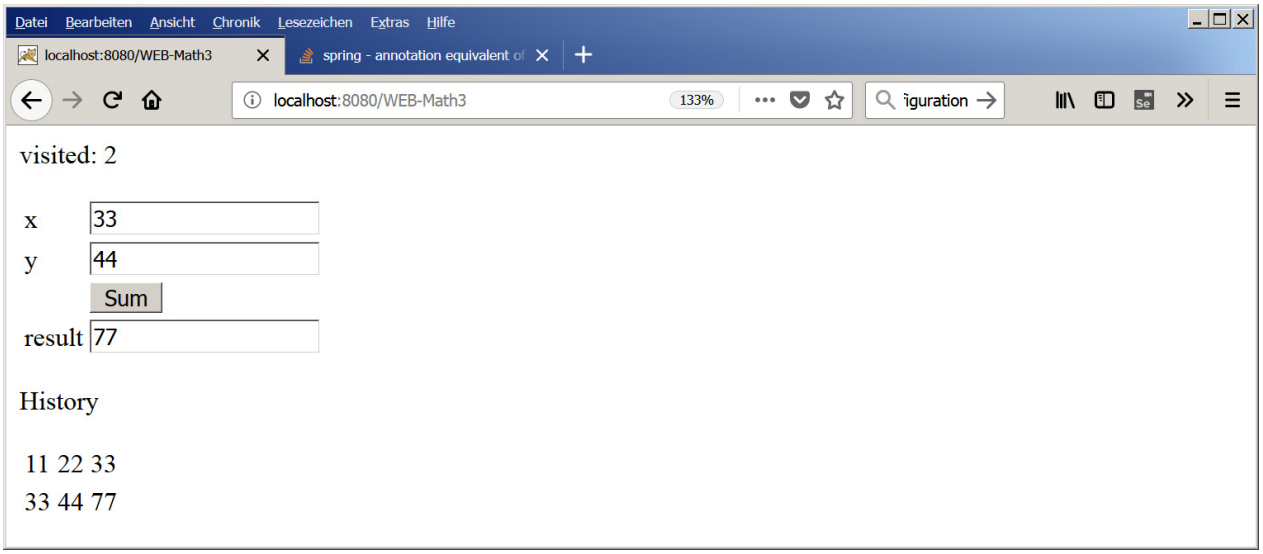
```
package beans;

@Component
@Scope(value = WebApplicationContext.SCOPE_SESSION,
       proxyMode=ScopedProxyMode.TARGET_CLASS)
public class SessionBean {
    // ...
}
```

```
package beans;

@Component
@Scope(value = WebApplicationContext.SCOPE_REQUEST,
       proxyMode=ScopedProxyMode.TARGET_CLASS)
public class RequestBean {
    // ...
}
```

Die `applicationContext.xml` beschränkt sich auf die Angabe des scan-Verzeichnisses:

```
<beans ...>
    <context:annotation-config/>
    <context:component-scan base-package="beans"/>
</beans>
```

13.5 Example: MathServlet



Das Feld "visited" zeigt an, wie häufig die Seite insgesamt besucht wurde. Mittels des "Sum"-Buttons kann eine Berechnung angestoßen werden, deren Ergebnis im "result"-Feld angezeigt wird. Unter "History" sind die in der aktuellen Session bereits ausgeführten Berechnungen zu sehen.

Die Anwendung definiert drei Bean-Klassen:

Eine `CounterBean` wird im Application-Scope eingehängt werden. Sie dient zur Speicherung der Anzahl Besuche. Bei jedem Besuch wird diese Anzahl inkrementiert:

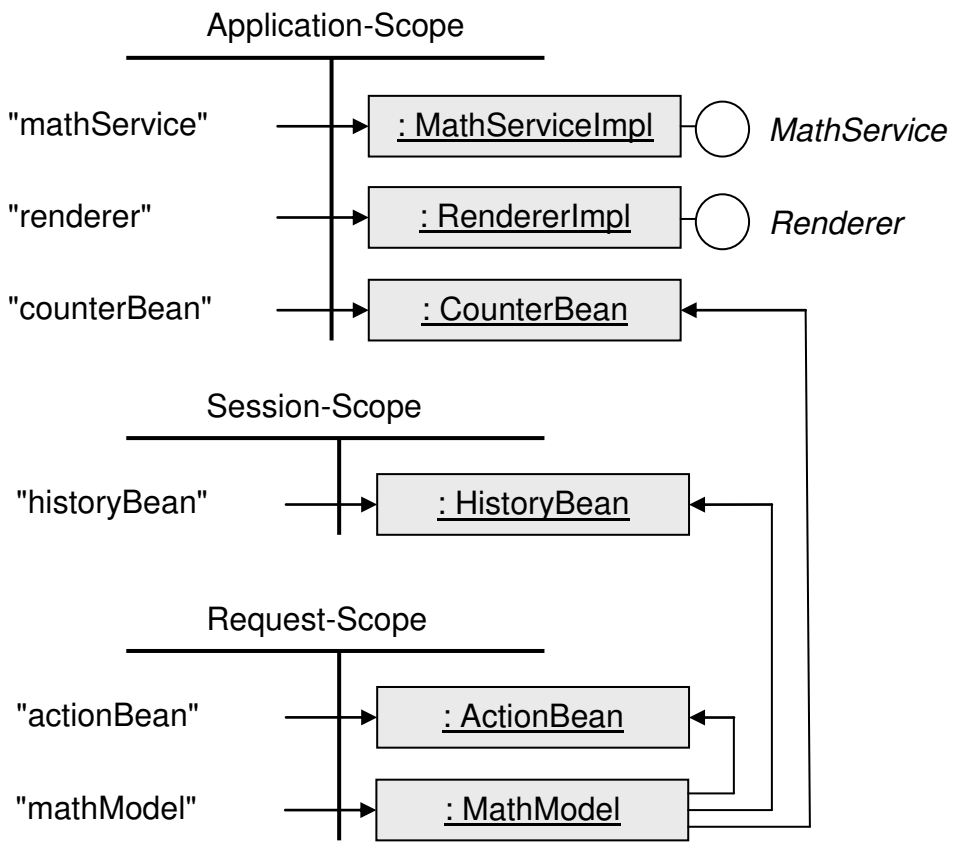
Eine `HistoryBean` wird in den Session-Scope eingehängt werden. Sie speichert in Form von Entry-Objekten die in der aktuellen Session erfolgten Berechnungen:

Eine `ActionBean` wird im Request-Scope eingehängt. Sie speichert die Input-Parameter und das Berechnungsergebnis:

Zusätzlich existiert eine `MathModel`, welches in den Request-Scope eingehängt wird und Referenzen auf die drei obigen Beans enthält (welches also die Gesamtheit dieser Beans repräsentiert).

Weiterhin wird ein `MathServiceImpl` und ein `RendererImpl` benutzt. Die Klassen dieser Objekte sind über Interfaces (`MathService`, `Renderer`) spezifiziert. Die Objekte der Implementierungs-Klassen werden im Application-Scope eingehängt. Das Servlet muss nur die Interfaces kennen.

Zunächst eine Übersicht zu der Belegung der Servlet-Scopes:



Hier die Klassen des `beans`-Pakets:

beans.CounterBean

```
package beans;
// ...
public class CounterBean implements Serializable {

    private static final long serialVersionUID = 1L;

    private int counter;

    public void inc() {
        this.counter++;
    }
    public int getCounter() {
        return this.counter;
    }
}
```

beans.HistoryBean

```
package beans;
// ...
public class HistoryBean implements Serializable {

	private static final long serialVersionUID = 1L;

	public static class Entry implements Serializable {

		private static final long serialVersionUID = 1L;

		private final int x;
		private final int y;
		private final int result;

		public Entry(int x, int y, int result) {
			this.x = x;
			this.y = y;
			this.result = result;
		}

		// getter ...
	}

	private final List<Entry> entries = new ArrayList<>();

	public List<Entry> getEntries() {
		return Collections.unmodifiableList(this.entries);
	}

	public void add(Entry entry) {
		this.entries.add(entry);
	}
}
```

beans.ActionBean

```
package beans;

public class ActionBean {

    private String x = "";
    private String y = "";
    private String result = "";

    // getter, setter...
}
```

Hier die `MathModel`-Klasse:

models.MathModel

```
package models;
// ...
public class MathModel {

    private final CounterBean counterBean;
    private final HistoryBean historyBean;
    private final ActionBean actionBean;

    public MathModel(CounterBean counterBean,
            HistoryBean historyBean, ActionBean actionBean) {
        this.counterBean = counterBean;
        this.historyBean = historyBean;
        this.actionBean = actionBean;
    }

    // getter...
}
```

Hier das Service-Interface und die Service-Klasse:

services.MathService

```
package services;

public interface MathService {
    public abstract int sum(int x, int y);
}
```

services.impl.MathServiceImpl

```
package services.impl;
// ...
public class MathServiceImpl implements MathService {
    @Override
    public int sum(int x, int y) {
        return x + y;
    }
}
```

Und hier schließlich das (allgemein verwendbare!) `Renderer`-Interface und die `RendererImpl`-Klasse (ein Renderer ist für die Produktion der HTML-Seite verantwortlich):

utils.Renderer

```
package utils;
// ...
public interface Renderer {
    public abstract void render(
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException;
}
```

Das Interface ist deshalb allgemein verwendbar, weil die `render`-Methode über keinerlei Applications-spezifische Parameter verfügt.

Die Implementierung dieses Interfaces besorgt sich den `ApplicationContext` und mittels dieses Kontexts dann die Referenz auf das `MathModel`. Über dieses `MathModel` schließlich gelangt die Anwendung dann zu den einzelnen Beans.

Die eigentliche Produktion der HTML-Seite ist hier nur in ihren wesentlichen Elementen dargestellt:

servlets.RendererImpl

```
package servlets;
// ...
public class RendererImpl implements Renderer {

    @Override
    public void render(
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {

        final MathModel model =
            (MathModel)request.getAttribute("mathModel");

        final CounterBean counterBean = model.getCounterBean();
        final HistoryBean historyBean = model.getHistoryBean();
        final ActionBean actionBean = model.getActionBean();

        response.setContentType("text/html");
        final PrintWriter out = response.getWriter();
        out.printf("<html>");
        out.printf("<body>");

        out.printf("visited: %d", counterBean.getCounter());

        out.printf("<form method='post' action='" +
            response.encodeURL("") + "'>");

        out.printf("<input name='x' type='text' value='%s'/>",
            actionBean.getX());
```

```
        out.printf("<input name='y' type='text' value='%s'/>",
            actionBean.getY());

        out.printf(
            "<input name='sum' type='submit' value='Sum'/>");

        out.printf(
            "<input name='result' type='text' value='%s' />",
            actionBean.getResult());

        out.printf("</form>");

        for (HistoryBean.Entry entry : historyBean.getEntries()) {
            out.printf("%d", entry.getX());
            out.printf("%d", entry.getY());
            out.printf("%d", entry.getResult());
        }

        out.printf("</body>");
        out.printf("</html>");
    }
}
```

Man beachte, dass die Implementierung keinerlei Abhängigkeiten von Spring aufweist.

(Hinweis: natürlich hätten wir auch z.B. die JSP-Technik zur Produktion der HTML-Seite verwenden können…)

Hier die Spring-Konfiguration:

WEB-INF/applicationContext.xml

```
<beans ...

    <bean id="counterBean"
          class="beans.CounterBean" scope="application"/>

    <bean id="historyBean"
          class="beans.HistoryBean" scope="session"/>

    <bean id="actionBean"
          class="beans.ActionBean" scope="request"/>

    <bean id="mathModel"
          class="models.MathModel" scope="request">
        <constructor-arg ref="counterBean"/>
        <constructor-arg ref="historyBean"/>
        <constructor-arg ref="actionBean"/>
    </bean>

    <bean id="mathService"
          class="services.impl.MathServiceImpl" scope="application"/>
```

```
    <bean id="renderer"
          class="servlets.RendererImpl" scope="application"/>

</beans>
```

Und hier schließlich das Servlet:

servlets.MathServlet

```
package servlets;
// ...
@WebServlet(urlPatterns="/*")
public class MathServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    public void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {

        final ApplicationContext ctx =
            ContextLoader.getCurrentWebApplicationContext();
        ctx.getBean(MathModel.class);
        final Renderer renderer = ctx.getBean(Renderer.class);
        renderer.render(request, response);
    }

    @Override
    public void doPost(
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {

        final ApplicationContext ctx =
            ContextLoader.getCurrentWebApplicationContext();

        final MathModel model = ctx.getBean(MathModel.class);

        final CounterBean counterBean = model.getCounterBean();
        final HistoryBean historyBean = model.getHistoryBean();
        final ActionBean actionBean = model.getActionBean();

        final MathService mathService =
            ctx.getBean(MathService.class);

        final Renderer renderer =
            ctx.getBean(Renderer.class);

        counterBean.inc();

        actionBean.setX(request.getParameter("x").trim());
        actionBean.setY(request.getParameter("y").trim());

        try {
```

```
            final int x = Integer.parseInt(actionBean.getX());
            final int y = Integer.parseInt(actionBean.getY());
            final int result = mathService.sum(x, y);
            final HistoryBean.Entry entry =
                new HistoryBean.Entry(x, y, result);
            historyBean.add(entry);
            actionBean.setResult(String.valueOf(result));
        }
        catch (final NumberFormatException e) {
            actionBean.setResult("bad input");
        }
        renderer.render(request, response);
    }
}
```

`doGet` wird beim erstmaligen Aufruf der Seite aufgerufen. In `doGet` wird der `Renderer` ermittelt und dessen `render`-Methode aufgerufen.

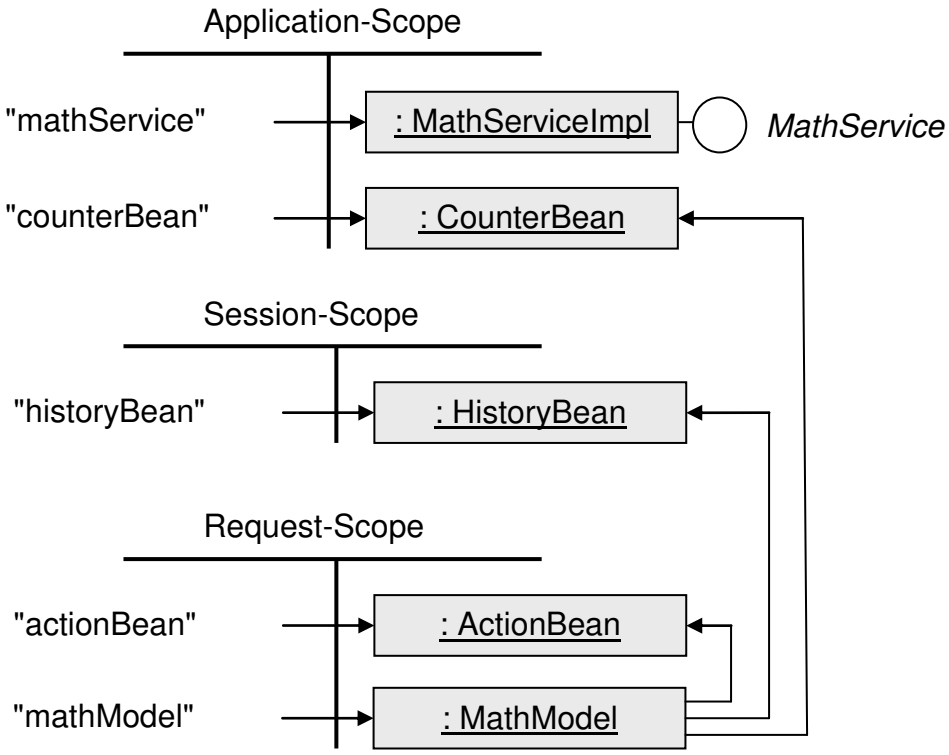
`doPost` wird jedes Mal dann aufgerufen, wenn der Benutzer eine Berechnungsaufgabe sendet. Hier werden die Beans ermittelt (über das `MathModel`) und zusätzlich die Referenzen auf den `MathService` und den `Renderer`. Die von der `CounterBean` verwaltete Anzahl der Besucht wird inkrementiert. Die `ActionBean` wird mit den Eingabeparametern versorgt. Dann findet unter Zuhilfenahme des `MathService` die eigentliche Berechnung statt. Das Ergebnis dieser Berechnung wird in die `ActionBean` eingetragen. Zur `HistoryBean` wird ein neuer `Entry` hinzugefügt. Und schließlich wird auch hier der `Renderer` gestartet.

14 MVC

Spring-MVC (Model-View-Controller) ist ein Framework für Servlet-basierte Anwendungen, welches die Entwicklung solcher Anwendungen vereinfacht.

Im folgenden bauen wir vier verschiedene Varianten derjenigen Anwendung, die bereits im letzten Abschnitt des letzten Kapitels entwickelt wurde (Varianten der Calculator-Anwendung).

Hier ein Diagramm der von Spring verwalteten Objekte (es handelt sich um dieselben Objekte, die auch im letzten Kapitel verwendet wurde. Nur das `Renderer`-Objekt fehlt – dessen Rolle wird nun eine JSP-Seite übernehmen.



Die `application.xml` sieht in allen Varianten gleich aus:

WEB-INF/application.xml

```
<beans ...>

    <bean id="counterBean" class="beans.CounterBean" scope="application"/>
    <bean id="historyBean" class="beans.HistoryBean" scope="session"/>
```

```
    <bean id="actionBean" class="beans.ActionBean" scope="request"/>

    <bean id="mathModel" class="models.MathModel" scope="request">
        <constructor-arg ref="counterBean"/>
        <constructor-arg ref="historyBean"/>
        <constructor-arg ref="actionBean"/>
    </bean>

    <bean id="mathService"
        class="services.impl.MathServiceImpl" scope="application"/>

</beans>
```

In allen Varianten wird dieselbe `jsp`-Seite verwendet. Hier der Quellcode dieser Seite – wobei EL-Ausdrücke fett gedruckt sind:

WEB-INF/jsp/math.jsp

```
<%@ taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core' %>
<%@ page pageEncoding='UTF-8' %>

<html>
    <head>
        <title>Calculator</title>
    </head>
    <body>
        <p>
        Visited: ${model.counterBean.counter}
        <p>
        <form method='post'>
            <table>
                <tr>
                    <td>X</td>
                    <td><input name='x' type='text'
                        value='${model.actionBean.x}' /></td>
                </tr>
                <tr>
                    <td>X</td>
                    <td><input name='y' type='text'
                        value='${model.actionBean.y}' /></td>
                </tr>
                <tr>
                    <td></td>
                    <td><input name='sum' type='submit'
                        value='sum' /> </td>
                </tr>
                <tr>
                    <td>Result</td>
                    <td><input name='result' type='text'
                        value='${model.actionBean.result}'
                        readonly /> </td>
                </tr>
            </table>

            History
```

```
            <p>
            <table>
                <c:forEach var='entry'
                        items='${model.historyBean.entries}'>
                    <tr>
                        <td>${entry.x}</td>
                        <td>${entry.y}</td>
                        <td>${entry.result}</td>
                    </tr>
                </c:forEach>
            </table>

        </form>
    </body>
</html>
```

Hier eine Übersicht zu den folgenden Abschnitten:

- Im Abschnitt 1 wird das `Controller`-Interface und das `ModelAndView`-Konzept vorgestellt.
- Im Abschnitt 2 werden die `@Controller`-Annotation und das automatische Request-Mapping demonstriert.
- Im Abschnitt 3 geht's um die Zustellung der Request-Parameter.
- Im Abschnitt 4 schließlich geht's erneut um die Request-Parameter – hier aber in Kombination mit der `@ModelAttribute`-Annotation.

Diese Einführung soll nur ein Gefühl dafür entwickeln, wie sich eine Spring-MVC-Anwendung grundsätzlich anfühlt. Sie enthält längst nicht alle Features, die Spring für eine solche Anwendung zur Verfügung stellt. Es gibt also noch viel zu studieren…

14.1 Das Interface Controller

In der web.xml wird ein `ContextLoaderListener` und ein `RequestContextListener` registriert. Als Servlet wird das Spring-eigene `DispatcherServlet` eingetragen - wobei dieses für Requests mit dem URL-Pattern `"/math"` zuständig ist:

WEB-INF/web.xml

```
<web-app ...>

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

    <listener>
        <listener-class>
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>calculator</servlet-name>
        <!-- also heisst die config datei: "calculator-servlet.xml" -->
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>calculator</servlet-name>
        <url-pattern>/math</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

</web-app>
```

Da in der `web.xml` als Servlet-Name `"calculator"` gewählt wurde, muss es eine zusätzliche Spring-Konfiguration mit dem Namen `calculator-servlet.xml` geben. In dieser ist der `MathController` registriert (der weiter unten beschrieben wird):

WEB-INF/calculator-servlet.xml

```
<beans ...>
    <bean name="/math" class="controller.MathController"/>
</beans>
```

Das `DispatcherServet` wird die Request-Verarbeitung delegieren an einen `MathController`. Die Klasse `MathController` implementiert das Spring-Interface `Controller`.

Das `Controller`-Interface spezifiziert eine Methode namens `handleRequest`, der ein `HttpServletRequest`- und ein `HttpServletResonse`-Objekt übergeben werden. Die Methode muss ein `ModelAndView` zurückliefern. Ein solches Objekt enthält zwei Dinge: das "Model", welches die zu rendernden Daten enthält, und ein Verweis auf diejenige Seite, die für das Rendern zuständig ist (im folgenden Beispiel ist dies eine JSP-Seite – Spring kann aber auch mit anderen Render-Technologien zusammenarbeiten).

controllers.MathController

```
package controller;
// ...
import org.springframework.http.HttpMethod;
import org.springframework.web.context.ContextLoader;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class MathController implements Controller {

    private static String VIEW = "/WEB-INF/jsp/math.jsp";

    @Override
    public ModelAndView handleRequest(
            HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {

        if (request.getMethod().equals(HttpMethod.GET.name()))
            return new ModelAndView(VIEW);

        final ApplicationContext ctx =
            ContextLoader.getCurrentWebApplicationContext();
        final MathModel model = ctx.getBean(MathModel.class);

        final CounterBean counterBean = model.getCounterBean();
        final HistoryBean historyBean = model.getHistoryBean();
        final ActionBean actionBean = model.getActionBean();

        final MathService mathService =
            ctx.getBean(MathService.class);

        counterBean.inc();

        actionBean.setX(request.getParameter("x").trim());
        actionBean.setY(request.getParameter("y").trim());

        try {
            final int x = Integer.parseInt(actionBean.getX());
            final int y = Integer.parseInt(actionBean.getY());
            final int result = mathService.sum(x, y);
```

```
            final HistoryBean.Entry entry =
                new HistoryBean.Entry(x, y, result);
            historyBean.add(entry);
            actionBean.setResult(String.valueOf(result));
        }
        catch (final NumberFormatException e) {
            actionBean.setResult("bad input");
        }

        return new ModelAndView(VIEW, "model", model);
    }
}
```

Im Falle es initialen Requests der Seite (im Falle also eines `GET`-Requests) wird ein `ModelAndView` mit einem leeren (nicht vorhandenen!) Model zurückgeliefert.

Im Falle eines `POST`-Requests werden die Eingabedaten ermitteln und in der `ActionBean` eingetragen, die Berechnung ausgeführt (und deren Resultat ebenfalls in der `ActionBean` eingetragen), die `MathHistory` um einen weiteren Eintrag ergänzt und der Zähler der `CounterBean` inkrementiert.

Schließlich wird ein `ModelAndView`-Objekt zurückgeliefert, welches unter dem Namen "`model`" das `MathModel` enthält (und dieses enthält die `ActionBean`, die `MathHistory` und die `CounterBean`). Die JSP-Seite kann also via "`model`" auf alle benötigten Daten zugreifen.

Resultat: Die `Controller`-Abstraktion ist eine recht "dünne" Abstraktion. Wir bewegen uns im Prinzip immer noch auf derselben Ebene, auf der wir uns auch dann bewegen würden, wenn wir statt des `DispatcherServlet`s einfach eine eigene Servlet-Klasse schreiben würden. Diese würde allerdings von einer Klasse abgeleitet (`HttpServlet`) – während wir bei der Ableitung von `Controller` ein Interface implementieren.

14.2 @Controller und @RequestMapping

Im folgenden leiten wird die `MathController` nicht mehr von `Controller` ab, sondern bauen eine Klasse, die mit `@Controller` annotiert ist.

Zunächst aber entkoppeln wie die Implementierung der View von der eigentlichen Anwendung – die Anwendung soll von `/WEB-INF/jsp/...jsp` nichts mehr sehen (die View-Technologie sollte also austauschbar sein. Deshalb wird in der `servlet.xml` ein `InternalResourceViewResolver` registriert. In der Anwendung kann die View nun einfach als `"math"` angesprochen werden.

Schließlich benötigen wir noch eine `component-scan`-Eintrag, damit der `MathController` gefunden und von Spring verwaltet werden kann:

WEB-INF/calculator-servlet.xml

```
<beans ...>

    <context:component-scan base-package="controller"/>

    <bean id="viewResolver"
          class="org.springframework.web.servlet.view.
                              InternalResourceViewResolver">
        <property name="viewClass"
            value="org.springframework.web.servlet.view.JstlView"/>
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

Nun zum `MathController`. Die Klasse ist eine einfache POJO-Klasse. Sie ist mit `@Controller` annotiert. Und sie besitzt eine `@RequestMapping`-Annotation.

Die Klasse hat zwei Methoden: `handleGet` und `handlePost`. Die erste Methode wird bei einem `GET`-Request, die zweite bei einem `POST`-Request aufgerufen (dies wird durch die jeweilige `@RequestMapping`-Annotation spezifiziert).

Über `@RequestMapping`-Annotationen legen wir also fest, welche Methoden aufgrund welcher Requests aufgerufen werden sollen.

Die erste Methode (`handleGet`) ist parameterlos; die zweite Methode (`handlePost`) hat einen `HttpServletRequest`-Parameter.

Methoden können im Prinzip beliebige Parameter haben – vorausgesetzt, Spring kann dies Parameter bedienen…

controllers.MathController

```
package controller;
// ...
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/math")
public class MathController {

    private static String VIEW = "math";

    @RequestMapping(method=RequestMethod.GET)
    public ModelAndView handleGet() {
        return new ModelAndView(VIEW);
    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView handlePost(HttpServletRequest request) {

        final ApplicationContext ctx =
            ContextLoader.getCurrentWebApplicationContext();
        final MathModel model = ctx.getBean(MathModel.class);

        final CounterBean counterBean = model.getCounterBean();
        final HistoryBean historyBean = model.getHistoryBean();
        final ActionBean actionBean = model.getActionBean();

        final MathService mathService =
            ctx.getBean(MathService.class);

        counterBean.inc();

        actionBean.setX(request.getParameter("x").trim());
        actionBean.setY(request.getParameter("y").trim());

        // Berechung...

        return new ModelAndView(VIEW, "model", model);
    }
}
```

Resultat: Wir benötigen keinerlei Fallunterscheidungen mehr, um die zum Request passende Aktion auszuführen. Statt dessen können wir für jeden "Request-Typ" einfach eine eigene Methode hinterlegen (wobei diese mit `@RequestMapping` ausgestattet ist). Außerdem brauchen wir die Methoden nur noch mit denjenigen Parametern ausstatten, die tatsächlich in der Methode auch benötigt werden.

14.3 @RequestParam

Statt die Eingabedaten aus dem `HttpServletRequest` zu ermitteln und diese dann entsprechend auf die verlangten Typen hin zu konvertieren, können wir die `handlePost`-Methode auch einfach mit Parametern jeweils speziellen Typs ausstatten, denen die Eingaben dann bereits in konvertierter Form übergeben werden.

Statt die `handlePost`-Methode mit einem `HttpServletRequest` zu parametrisieren, können wir sie einfach mit zwei `int`-Parametern (`x` und `y`) ausstatten. Diese müssen dann aber jeweils mit `@RequestParam` annotiert sein – wobei dieser Annotation der Name des erwarteten Request-Parameters übergeben wird.

Dabei stellt sich dann die Frage, was passiert, wenn die erforderlichen Konvertierungen scheitern. Für diesen Fall können wir eine Methode definieren, welche mit der Annotation `@ExceptionHandler` ausgestattet ist, welche mit dem Typ der zu bearbeitenden Exception ausgestattet ist. Diese Methode kann als Parameter u.a. die geworfene Exception und den `HttpServletReqeust` verlangen. Im Falle eines Konvertierungsfehlers wird dann automatisch eben dieser Exception-Handler aufgerufen.

controllers.MathController

```
package controller;
// ...
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.method.annotation.
                        MethodArgumentTypeMismatchException;

@Controller
@RequestMapping("/math")
public class MathController {

    private static String VIEW = "math";

    @RequestMapping(method = RequestMethod.GET)
    public ModelAndView handleGet() {
        return new ModelAndView(VIEW);
    }

    @RequestMapping(method = RequestMethod.POST)
    public ModelAndView handlePost(
            @RequestParam("x") int x,
            @RequestParam("y") int y) {

        // Berechung...
```

```
        return new ModelAndView(VIEW, "model", model);
    }

    @ExceptionHandler(MethodArgumentTypeMismatchException.class)
    public ModelAndView handleTypeMismatch(
            MethodArgumentTypeMismatchException ex,
            HttpServletRequest request) {
        final ApplicationContext ctx =
            ContextLoader.getCurrentWebApplicationContext();
        final MathModel model = ctx.getBean(MathModel.class);
        final ActionBean actionBean = model.getActionBean();
        actionBean.setX(request.getParameter("x"));
        actionBean.setY(request.getParameter("y"));
        actionBean.setResult(ex.getMessage());
        return new ModelAndView(VIEW, "model", model);
    }
}
```

Resultat: Wir müssen uns nicht mehr um die erforderlichen Konvertierungen der Request-Parameter kümmern – die Eingaben werden uns sofort in der passenden Form übergeben. Und auf einen `HttpServletRequest`-Parameter können wir komplett verzichten. Wir haben uns also erneut ein Stück weit von der zugrundeliegenden Servlet-Technologie entfernt.

14.4 @ModelAttribute

Statt eine `handlePost` mit zwei mit `@RequestParam` annotierten `int`-Parametern zu definieren, können wir auch eine Klasse schreiben, welche für jeden der erwarteten Eingaben ein Attribut (und eine entsprechende Property) enthält:

```
package controller;

public class MathRequestParams {
    private int x;
    private int y;

    // getter, setter...
}
```

Die `handleRequest`-Methode kann dann einfach ein `MathRequstParams`-Objekt verlangen. Und sie kann einen weiteren Parameter verlangen: eine Parameter vom Typ `BindingResult`. Dieses `BindingResult` enthält alle Fehler, die bei der Konvertierung der Request-Parameter erkannt wurden.

```
package controller;
// ...
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;

@Controller
@RequestMapping("/math")
public class MathController {

    private static String VIEW = "math";

    @RequestMapping(method = RequestMethod.GET)
    public ModelAndView handleGet() {
        return new ModelAndView(VIEW);
    }

    @RequestMapping(method = RequestMethod.POST)
    public ModelAndView handlePost(
            @ModelAttribute MathRequestParams mathRequestParams,
            BindingResult bindingResult) {

        final ApplicationContext ctx =
            ContextLoader.getCurrentWebApplicationContext();

        final MathModel model = ctx.getBean(MathModel.class);

        final CounterBean counterBean = model.getCounterBean();
        final HistoryBean historyBean = model.getHistoryBean();
        final ActionBean actionBean = model.getActionBean();

        final MathService mathService = ctx.getBean(MathService.class);
```

```
        counterBean.inc();

        if (bindingResult.hasErrors()) {
            bindingResult.getFieldValue("x");
            actionBean.setX(bindingResult.getFieldValue("x").toString());
            actionBean.setY(bindingResult.getFieldValue("y").toString());
            actionBean.setResult("bad input");
            return new ModelAndView(VIEW, "model", model);
        }

        final int x = mathRequestParams.getX();
        final int y = mathRequestParams.getY();

        // Berechnung...

        return new ModelAndView(VIEW, "model", model);
    }
}
```

Resultat: Wir müssen nicht für jedes Eingabefeld einen eigenen Parametern definieren. Spring kann uns ein Objekt übergeben, welches alle Eingabedaten in bereits konvertierter Form enthält.