BBM490

ENTERPRISE WEB ARCHITECTURE

Mert ÇALIŞKAN

Hacettepe University Spring Semester ’15

Recap of Week 03

- We defined the concept behind Maven Framework.
- We mentioned about the POM.
- We defined the simplest POM and how it's being merged with a super POM.
- We created a Maven Project with Eclipse Luna.
- We defined the Build LifeCycle.
- We mentioned about how Maven resolves dependencies.

Recap of Week 03

- We give samples on Transitive Dependencies.
- We defined the scopes for dependencies and how they affect the classpath (C,T,R) of the project.
- We defined the archetype concept.
- We detailed how Maven manages dependencies between Maven projects.

Recap of Week 03

- We defined the Repository approach.
- We gave a sample for plugin development with Maven.
- We defined the usage of settings.xml
- We defined the Maven profiles and the idea behind using it.
- To wrap up, we used the Maven goodness to create applications and get the dependencies instantly.

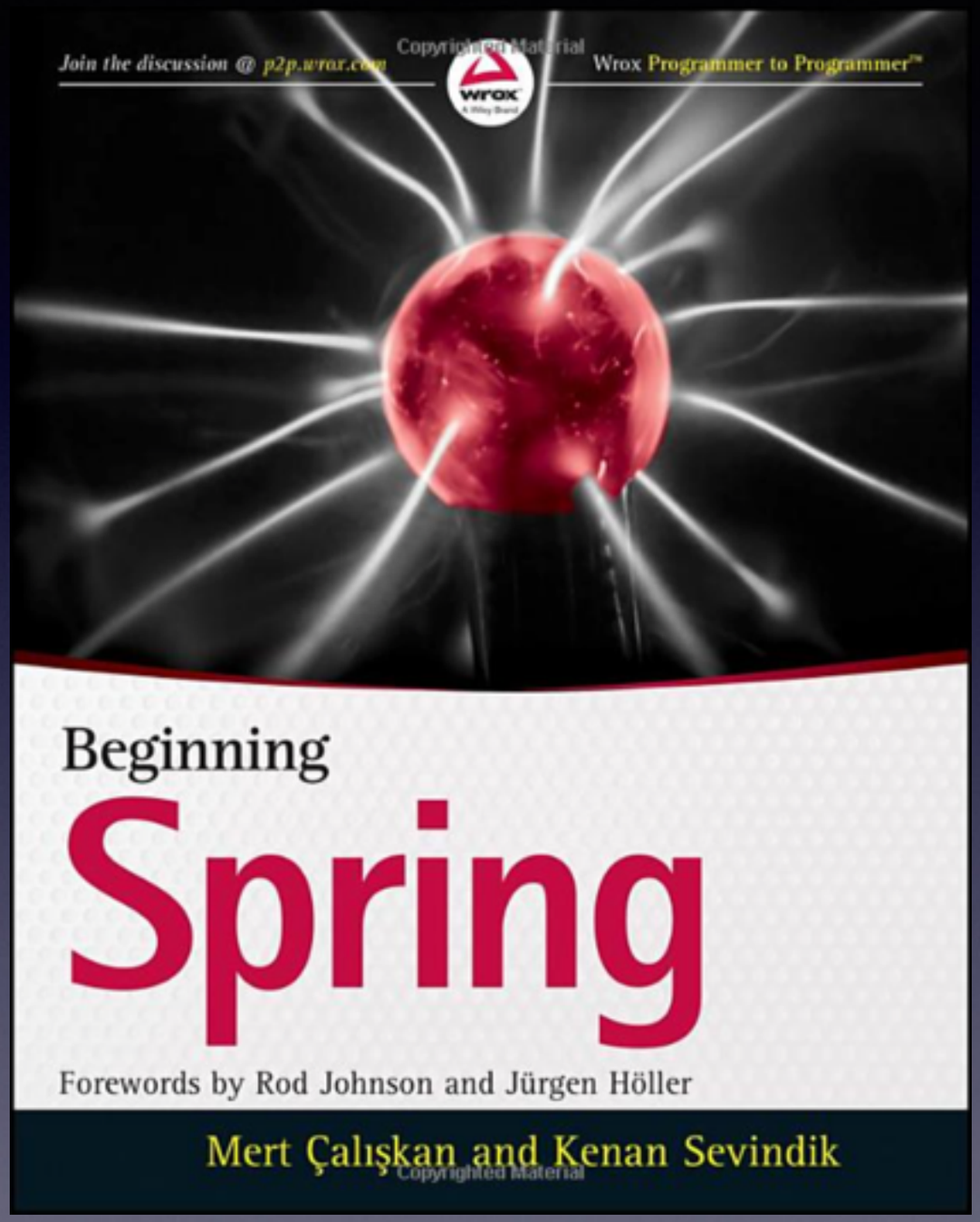
Week 4

Spring Framework

We'll examine Spring Framework with concepts like Dependency Injection / Inversion of Control.

Spring 4.1 is released in September'14.

We will examine the Spring Framework with its concepts like Dependency Injection / Inversion of Control, the Aspect Oriented Programming
and
we will focus on the latest features of the framework coming with the 4.x version (which maps to JDK8 and JavaEE7).



Spring Book focusing on topics:

- POJO programming model & DI
- Spring MVC
- JDBC and JPA Integration w/ Spring
- Transaction management in Spring
- TDD w/ Spring
- Expression Language, AOP, Caching w/ Spring
- RESTful Web Services w/ Spring
- Spring Security

with over 100+ working examples…

POJO Programming Model

- POJO stands for Plain Old Java Objects.
  - Idea founded by Martin Fowler, Rebecca Parsons, and Josh MacKenzie (Sept. 2000)
- Aim was to simplify the coding, testing, deployment phases of enterprise Java applications by doing the development in java objects instead of entity beans.
- It first addressed to solve the problems of EJB programming model so let's first take a look at what that means.

EJB Programming Model

- In 1997, J2EE spec offered the Enterprise Java Beans which tried to focus the developers on business logic rather than the middleware requirements like,
  - persistency stuff
  - transaction management
  - security
  - multi-threading,
  - distribution and etc.

EJB Programming Model

- But that didn't turned out well.
  First you need to implement home and remote interfaces:

```
public interface PetClinicService extends EJBObject {
    void saveOwner(Owner owner) throws RemoteException;
}

public interface PetClinicServiceHome extends EJBHome {
    PetClinicService create() throws RemoteException, CreateException;
}
```

- And a business object class following them.

EJB Programming Model

```
public class PetClinicServiceBean implements SessionBean {
    private SessionContext sessionContext;
    public void ejbCreate() {
    }
    public void ejbRemove() {
    }
    public void ejbActivate() {
    }
    public void ejbPassivate() {
    }
    public void setSessionContext(SessionContext sessionContext) {
       this.sessionContext = sessionContext;
    }
    public void saveOwner() throws java.rmi.RemoteException {
        //implementation of saving owner instance...
    }
}
```

- And that's all for just for one save method…and notice that the business object class is not implementing the interfaces :)

EJB Programming Model

- Testing EJBs outside the container was also a hassle to deal with and it was not like right-click and *Run as JUnit test*.
- XML Configuration turns deployment into hell, literally. 2 beans injected to each other with transaction management control.

41 lines of XML

```
<ejb-jar>
    <display-name>PetClinicEJB2</display-name>
    <enterprise-beans>
        <session>
        <ejb-name>PetClinicService</ejb-name>
        <home>com.example.PetClinicServiceHome</home>
        <remote>com.example.PetClinicService</remote>
        <ejb-class>
                com.example.PetClinicServiceImpl
        </ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
        <resource-ref>
            <res-ref-name>jdbc/ds</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
    </session>
    <message-driven>
        <ejb-name>MessageSubscriber</ejb-name>
        <ejb-class>com.example.MessageSubscriber</ejb-class>
        <transaction-type>Container</transaction-type>
        <message-destination-type>
          javax.jms.Topic
        </message-destination-type>
        <ejb-ref>
            <ejb-ref-name>ejb/PetClinicService</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <home>com.example.PetClinicServiceHome</home>
            <remote>com.example.PetClinicService</remote>
            <ejb-link>PetClinicService</ejb-link>
        </ejb-ref>
    </message-driven>
    </enterprise-beans>
    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>PetClinicService</ejb-name>
                <method-name>saveOwner</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

EJB Programming Model

- The motto for J2EE and EJB was:

  Write once and run everywhere!

- Which turned out to be:

  Write once and debug (and pray) everywhere!

  with the vendor lock-in on the application servers

- The code got tightly coupled with the API of the application servers.

EJB 3/3.1

- The current version of the EJB improved by addressing these problems.
  - POJO programming model applied.
  - We still have session and message driven beans but much simpler now.
  - Entity beans are built on JPA.
  - Testing and Deploying became much simpler.
- Nevertheless to say that these all inspired from POJO oriented frameworks like Spring & Hibernate.

What J2EE offered…

- Despite all the problems, J2EE also brought some cool stuff into the enterprise development world like,
  - Database connection management outside application code
  - Ready to use transaction management infrastructure
  - Component wiring and dependency management
  - Scheduling and Thread management and etc.

Now the Question is,

Do we need containers like from the era of J2EE to provide all those aforementioned features?

Expert one on one J2EE Development without EJB

- Rod Johnson and Juergen Hoeller founded the idea that the a lightweight container can also provide the features of J2EE without the burden of EJB.

- Life cycle management
- Dependency resolution
- Component lookup
- Application configuration
- Transaction management
- Security
- Thread management
- Object and resource pooling
- Remote access for components

and many others…

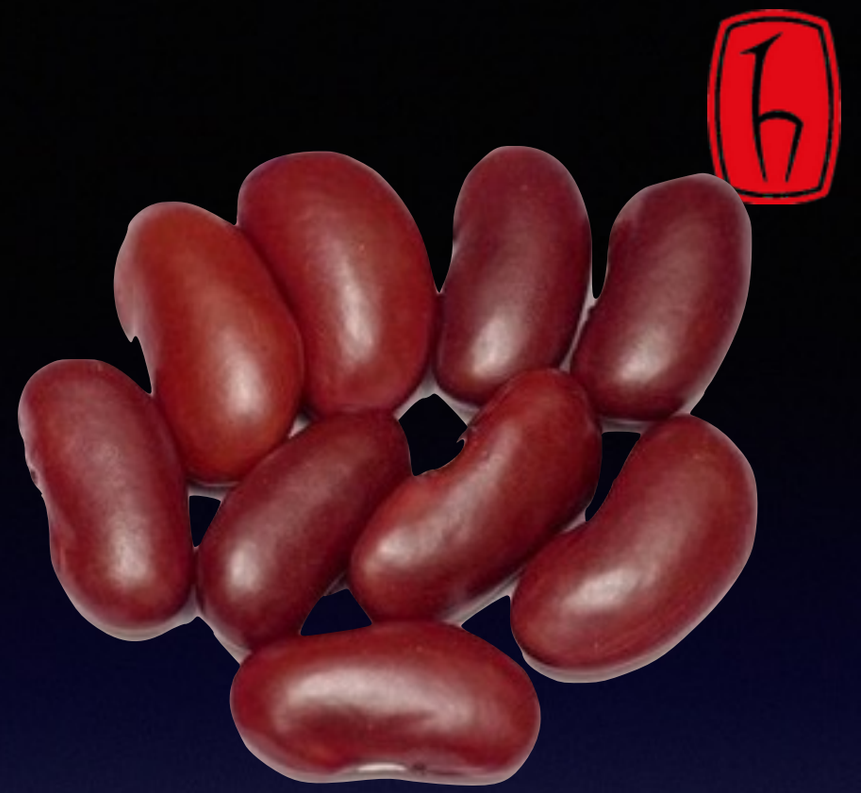
Spring Framework

- It's an IoC container based on Java and it's very light-weight.
- IoC is the most powerful part, we'll mention IoC in a bit.
- It supports the IoC features with the power of the AOP, Aspect Oriented Programming.
- It has no container requirements. You can run it on any application server.
- v1.0 released in March 2004. Now we have version 4.1.
- The first company behind Spring was *"Interface 21"* and then renamed to *"SpringSource"*.
- In 2009 VMWare bought SpringSource for 420 m$;)

IoC

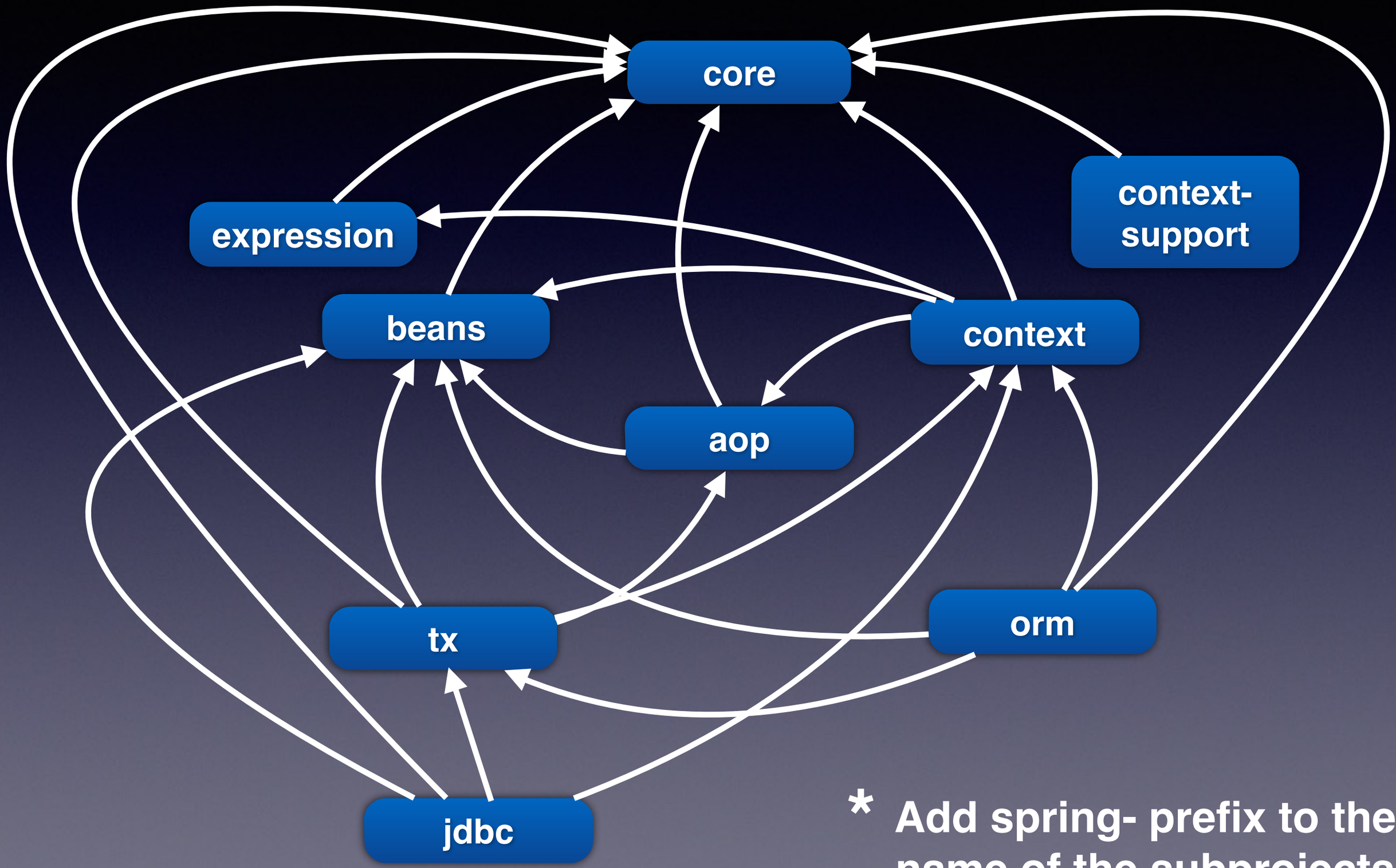
- IoC stands for Inversion of Control.
- It's either provided by Dependency Lookup or Dependency Injection (DI).
- Objects define their dependencies, the other objects they work with. The container then inject these dependencies when it's creating the bean.
- The process defined here is inverse, the bean does not control the instantiation by using direct construction of the classes but the other way around.
- It's also known as the *Hollywood Principle*
  (Don't call us, we'll call you)

Definition of a Bean



- The core of the Spring is the IoC container. Its job is to instantiate → initialise → wire up objects.
- The objects managed by Spring are called Beans. They are just POJOs but instantiated, assembled and maintained by the Spring container.

Spring Sub-Projects' Inter-Dependencies *



* Add spring- prefix to the name of the subprojects

Spring Maven Dependency

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.1.5.RELEASE</version>
    </dependency>
</dependencies>
```

- Application Context (depends on spring-core, spring-expression, spring-aop, spring-beans)
- This is the central artefact for Spring's Dependency Injection Container and is generally always defined.
- 4.1.5 is the current latest release of the framework.

Configuration Metadata

- Spring container demands information on which beans to instantiate, initialize and wire them together.
- This metadata information can either be XML based or Annotation based.
- They both can be used or mixed together, container will work regardless of this method.

Sample XML Config

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

    <bean id="accountService" class="mycomp.AccountServiceImpl">
        <property name="accountDao" ref="accountDao"/>
    </bean>

    <bean id="accountDao" class="mycomp.AccountDaoInMemoryImpl" />
</beans>
```

src:spring-xml-based-configuration

Bean Definition Configuration

- Within the applicationContext we can define the bean with the configurations (not all listed) given below.

Attribute	Definition
class	Class name to create the bean.
name / id	Both defines the bean as unique.
scope	Defines the scope of the bean and how the lifespan of it across the container.
constructor-arg	Defines the way to inject dependencies via constructor
properties	Defines the way to inject dependencies via setter methods

Scopes of the Bean

- Scope of the bean can be defined and this will affect its lifespan in the IoC container.

Values	Definition
prototype	Creates a new instance of the bean each time it's needed.
singleton*	Creates only one instance of the bean.
request	*lifespan of the bean is associated with an HTTP request*
session	*lifespan of the bean is associated with an HTTP session*

* is the default scope.

src:spring-webapp-scopes TBD

Going all the Annotation way

- We can remove all the XML config that we gave for the 2 examples and configure all of it with Java annotations.

```
@Configuration
public class SampleBeanConfiguration {

    @Bean
    public AccountService accountService() {
        AccountServiceImpl bean = new AccountServiceImpl();
        bean.setAccountDao(accountDao());
        return bean;
    }

    @Bean
    public AccountDao accountDao() {
        AccountDaoInMemoryImpl bean = new AccountDaoInMemoryImpl();
        //depedencies of accountDao bean will be injected here...
        return bean;
    }
}
```

src:spring-java-based-configuration

Going all the Annotation way

- @Configuration annotation tells Spring that the class annotated with it contains configuration metadata as well as itself as a bean.
- Within the configuration class, we have created two factory methods marked with @Bean. Those methods are called by Spring Container during bootstrap and their returning values are treated as Spring managed beans. Method name is accepted as bean name by default.

Configuration Best Practices

- Beans that will operate in web/presentation layer of application are defined in beans-web.xml file or ConfigurationForWeb class.
- Beans that will operate in service/business layer of application are defined in beans-service.xml file or ConfigurationForService class.
- Beans that will operate in data access layer of application are defined in beans-dao.xml file or ConfigurationForDao class.
- Beans that are necessary for several container specific features to be activated are defined in beans-config.xml file or ConigurationForConfig class.
- Beans that are used for security requirements of application are defined in beans-security.xml file or ConfigurationForSecurity class.

Bean Lookup

- BeanFactory interface provides method signatures to lookup Spring beans.
- It's the base interface for accessing the Spring Bean container. The signatures provided are:

```
<T> T getBean(Class<T> requiredType) throws BeansException;

Object getBean(String name) throws BeansException;

<T> T getBean(String name, Class<T> requiredType) throws BeansException;

Object getBean(String name, Object... args) throws BeansException;
```

Interface ApplicationContext

- extends *BeanFactory* interface.
- It's the central interface that provides ways to configure the application. It's responsible for instantiating, configuring, and assembling the beans.
- It does its job by reading configuration metadata. Configuration metadata could be in XML, Java Annotations or Java Code.

Interface ApplicationContext

- 3 important concrete implementation exist that load bean definitions,
  - FileSystemXmlApplicationContext: XML file in the file path.
  - ClassPathXmlApplicationContext: XML file that exist in the CLASSPATH.
  - AnnotationConfigApplicationContext: Accepting annotated classes as input for loading application context.

Startup your container

- Spring container is also a Java object and there are 2 ways to start it.
- In standalone applications we used the programmatic approach as we did in previous examples.

```
ClassPathXmlApplicationContext applicationContext =
     new ClassPathXmlApplicationContext("/applicationContext.xml");
```

```
AnnotationConfigApplicationContext applicationContext =
     new AnnotationConfigApplicationContext(WorkshopBeanConfiguration.class);
```

- In web applications web.xml file can be used to startup the Spring container (remember the listeners that I mentioned?). We will get to that soon.

Bean Lifecycle with init-method and destroy-method

- Spring provides the hook for lifecycle callback methods that are invoked at the initialisation and destruction of a bean.

```
<bean name="myBean"
class="tr.edu.hacettepe.bbm490.MyBean"
init-method="init" destroy-method="destroy"/>
```

```
public void init() {
    System.out.println("Initialized");
}
```

```
public void destroy() {
    System.out.println("Destroyed");
}
```

src:spring-bean-lifecycle

InitializingBean / DisposableBean

- They are interfaces to perform certain actions upon bean initialisation and destruction.
- They are the same as the init and destroy methods but the order differs.

```
public interface InitializingBean {
   void afterPropertiesSet() throws Exception;
}
```

```
public interface DisposableBean {
   void destroy() throws Exception;
}
```

Order between interfaces and init-method / destroy-method

- 1. Constructor
- 2. InitializingBean - afterPropertiesSet method
- 3. method defined by init-method

  ....
- 4. DisposableBean - destroy method
- 5. method defined by destroy-method

Bean Injecting Method

- We can do the injection via Setter methods as:

```
<bean name="myBean" class="tr.edu.hacettepe.bbm490.MyBean">
   <property name="message" value="hello"></property>
</bean>
```

  - It's possible to inject other bean dependencies and primitive values, strings, classes, enums etc.
  - References to other beans are specified with ref attribute of <property> tag like,

```
<bean id="accountService"
class="tr.edu.hacettepe.bbm490.AccountServiceImpl">
    <property name="accountDao" ref="accountDao"/>
</bean>
```

Bean Injecting Method

- Constructor injection is performed during component creation.
- So it's possible to inject with Constructor arguments like,

```
<bean name="myBean" class="tr.edu.hacettepe.bbm490.MyBean">
   <constructor-arg name="message" value="hello" />
</bean>
```

```
public MyBean(String message) {
   this.message = message;
}
```

src:spring-constructor-injection

Bean Injecting Method

- If the name of the setter method or the name of the parameter of constructor do not match the definition in the applicationContext.xml, errors like given below will occur,

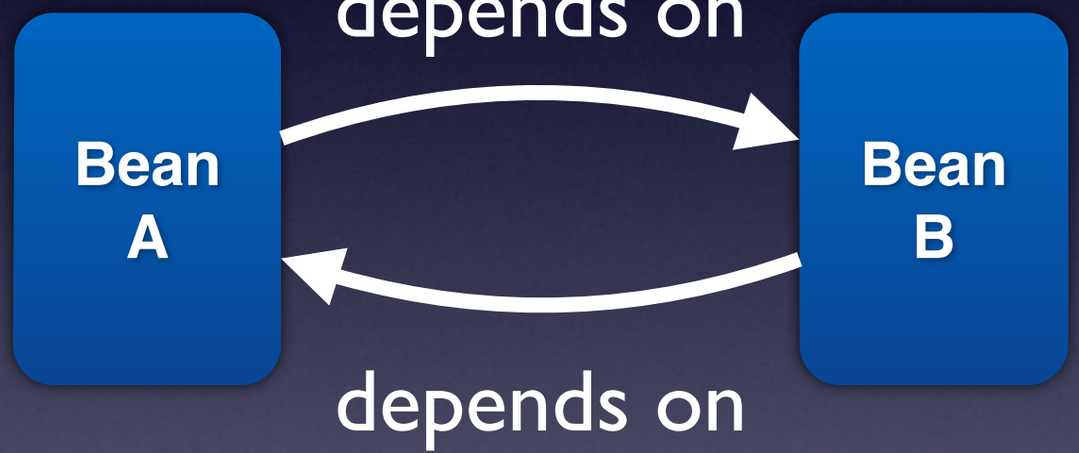
```
Bean property 'msg' is not writable or has an invalid setter
method.

Could not resolve matching constructor (hint: specify index/
type/name arguments for simple parameters to avoid type
ambiguities)
```

- We'll get to auto wiring soon, which would ease the development on topics like these.

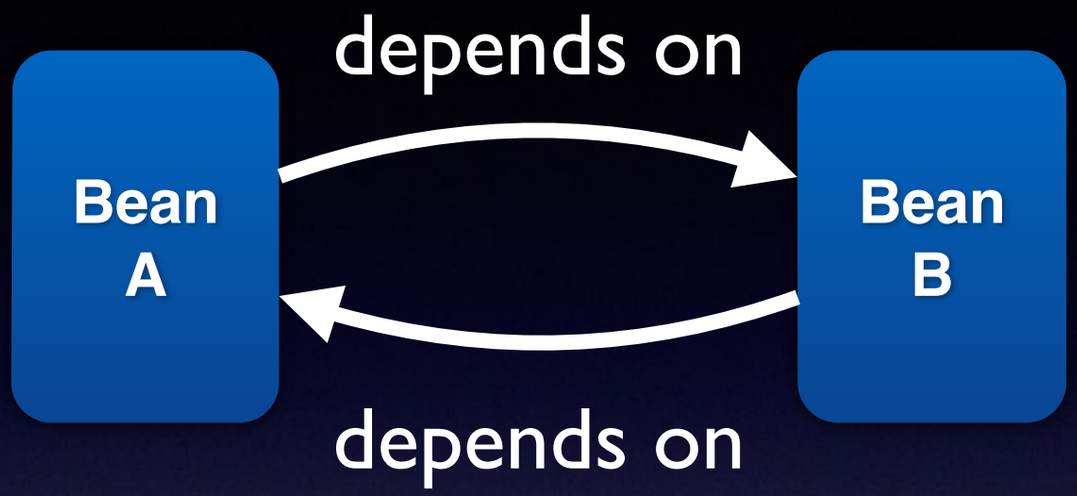
Circular dependencies

- What would happen if you have 2 beans depending on each other?



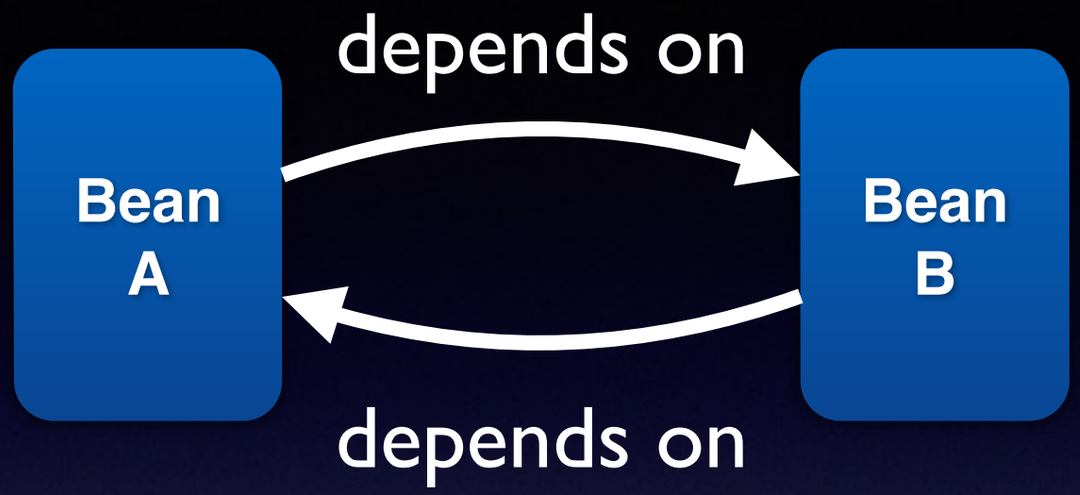
- Setter or Constructor injection would work?

Circular dependencies



Constructor injection would not work. It's because while creating beanA, container will look for beanB and while creating beanB, container will look for the beanA. And at that point since beanA will not be ready for injection *BeanCurrentlyInCreationException* will be thrown.

Circular dependencies



Setter injection will work. First beanA will be created by calling its default constructor and then beanB will be instantiated. At that point container will look for beanA and since it's created it will be available for injection.

One downside of this injection will be lack of features like transaction, validation, caching, security provided by the Spring Container since they get applied by BeanPostProcessors after the bean creation.

Injecting Collections

- Spring also provides to inject values as list, set, map.

```
<bean name="myBean" class="tr.edu.hacettepe.bbm490.MyBean">
    <property name="countries">
        <list>
            <value>Turkey</value>
            <value>USA</value>
            <value>Germany</value>
        </list>
    </property>
</bean>
```

```
<bean name="myBean" class="tr.edu.hacettepe.bbm490.MyBean">
    <property name="countriesMap">
        <map>
            <entry key="1" value="Turkey"/>
            <entry key="2" value="USA"/>
            <entry key="3" value="Germany"/>
        </map>
    </property>
</bean>
```

src:spring-inject-collections

Defining Inner Bean

- We can define a bean while defining a bean…!

```
<bean name="myBean" class="tr.edu.hacettepe.bbm490.MyBean">
   <property name="myInnerBean">
      <bean class="tr.edu.hacettepe.bbm490.MyInnerBean"></bean>
   </property>
</bean>
```

- We can define inner beans within,
  <property/>
  <constructor-arg/>
- It's like inner classes that are defined within of other Java classes.

Abstract Beans

- While defining a bean we can set the property as abstract="true" and that means,

  The bean itself is not meant to be instantiated but rather just serving as parent for concrete child bean definitions. It's like abstraction in Java but we don't even need to map a Java class to the bean. It's vital to group common properties, so you avoid repetition in XML.

```
<bean name="myService" class="tr.edu.hacettepe.bbm490.MyService" />

<bean name="abstractParent" abstract="true">
   <property name="service" ref="myService"/>
</bean>

<bean name="myBean"
     class="tr.edu.hacettepe.bbm490.MyBean" parent="abstractParent" />
```

Bean Autowiring

- Too much XML 'till this slide right?
- Spring container can autowire bean dependencies to cut down this XML configuration.
- We can use the *autowire* attribute of the <bean/> to configure it.
- One drawback on this is we cannot auto-wire simple properties and Strings.
- *autowire* attribute can either get byName, byType, constructor or no (no is the default behaviour).

Bean Autowiring (byName)

- The injection would be as follows automagically:

```
<bean id="accountService"
class=“tr.edu.hacettepe.bbm490.AccountServiceImpl"
autowire="byName"/>

<bean id="accountDao"
class=“tr.edu.hacettepe.bbm490.AccountDaoInMemoryImpl”/>

<bean id="accountDaoJdbc"
class=“tr.edu.hacettepe.bbm490.AccountDaoJdbcImpl"/>
```

- Here, we don’t need to set any property into service bean.

src:spring-xml-based-autowiring

Bean Autowiring - Exception

- If there is more than one type, exception gets thrown like,

```
No qualifying bean of type
[tr.edu.hacettepe.bbm490.AccountDaoInMemoryImpl] is defined:
expected single matching bean but found 2:
```

Bean Autowiring (byType)

- With this option Spring investigates properties of a bean definition by looking at its class via Java reflection, and tries to inject available beans in the container to the matching properties by their types.

```
<bean id="accountService"
class=“tr.edu.hacettepe.bbm490.AccountServiceImpl"
autowire="byType"/>

<bean id="accountDao"
class=“tr.edu.hacettepe.bbm490.AccountDaoInMemoryImpl"/>
```

- If there is more than one bean instance as autowiring candidates, injection will fail. *autowire-candidate* attribute can be set to overcome this problem.

autowire-candidate

- Here the accountDao bean will be opt out for injection and accountDaoJdbc will be selected for injection into accountService bean.

```
<bean id="accountService" class=“tr.edu.hacettepe.bbm490.AccountServiceImpl"
autowire="byType"/>

<bean id="accountDao" class=“tr.edu.hacettepe.bbm490.AccountDaoInMemoryImpl”
autowire-candidate="false"/>

<bean id="accountDaoJdbc" class=“tr.edu.hacettepe.bbm490.AccountDaoJdbcImpl"/>
```

Bean Autowiring (constructor)

- It's similar to byType but Spring container tries to find beans whose types matching with constructor arguments of the bean class.

```
<bean id="accountService"
class="tr.edu.hacettepe.bbm490.AccountServiceImpl"
autowire="constructor"/>

<bean id="accountDao"
class="tr.edu.hacettepe.bbm490.AccountDaoInMemoryImpl"/>
```

- If there is more than one candidate bean for the constructor argument, injection will fail.

src:spring-autowiring-constructor

Bean Autowiring (no)

- What happens if there is no candidate bean found in Spring Container?
- no bean will be injected into specified property if there is no candidate for injection.
- The same will also applies if the autowire attribute is set to no.
- And the property will have its default value (most probably null).

depends-on

- myBeanA depends on myBeanB, Spring container will create the myBeanB first and then myBeanA. That's certain.
- But if there are 2 beans and have no relationship between each other, the creation order is not guaranteed by the Spring container.
- The definition is as follows:

```
<bean id="beanA"
      class="tr.edu.hacettepe.bbm490.BeanA" depends-on="beanB, beanC"/>
```

- We can list several bean names in depends-on attribute.
- Annotation approach for this feature can be provided by @DependsOn, it should be put on class level and will be picked up during the component scan phase.

Annotation Based Config

- Spring version 2.5 brought the goodness of Java Annotations to the container.
- It made possible to define the beans via annotations.
- First we need to enable this configuration via annotation in applicationContext.xml (coming up in next slides).
- Then we need to annotate our beans with **@Component**.
- Then we can wire our beans into others with **@Autowired**.

Annotation Based Config

- The @Component annotation is the base annotation and
    - @Controller
    - @Repository
    - @Service
- annotations are deriving from it. They all offer the same feature, creating a Spring managed bean via annotations.

Annotation Based Config

- The XML configuration would be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<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.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="tr.edu.hacettepe.bbm490"/>

</beans>
```

Annotation Based Config

```
@Service("accountService")
public class AccountServiceImpl implements AccountService {
    private AccountDao accountDao;

    @Autowired
    @Qualifier("accountDao")
    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
}

@Repository
@Qualifier("accountDao")
public class AccountDaoInMemoryImpl implements AccountDao {

}
```

src:spring-annotation-based-autowiring

@Qualifier

- This annotation can be used when there is ambiguity like more than one bean that can't be mapped byType.

```
public interface BeanMarkerInterface {
    void sayHelo();
}

@Component
public class MyDependencyBean implements BeanMarkerInterface {

    public void sayHelo() {
        System.out.println("Hello from dependency");
    }
}

@Component
public class MyOtherDependencyBean implements BeanMarkerInterface {

    public void sayHelo() {
        System.out.println("Hello from other dependency");
    }
}
```

@Qualifier

```
@Component
public class MyBean {

   @Autowired
   private BeanMarkerInterface bean;

   public BeanMarkerInterface getBean() {
      return bean;
   }

   public void setBean(BeanMarkerInterface bean) {
      this.bean = bean;
   }
}
```

- It will give an exception like,

```
No qualifying bean of type [tr.edu.hacettepe.bbm490.BeanMarkerInterface]
is defined: expected single matching bean but found 2:
myDependencyBean,myOtherDependencyBean
```

@Qualifier

```
@Component
public class MyBean {

   @Autowired
   @Qualifier("myOtherDependencyBean")
   private BeanMarkerInterface bean;

   public BeanMarkerInterface getBean() {
      return bean;
   }

   public void setBean(BeanMarkerInterface bean) {
      this.bean = bean;
   }
}
```

@PostConstruct
@PreDestroy

- @PostConstruct annotation is used on a method that needs to be executed after dependency injection is done to perform any initialisation.
- @PreDestroy annotation is used on methods as a callback notification to signal that the instance is in the process of being removed by the container.
- These annotations were coming from the JSR250 that contains the annotations under javax.annotation package, but now it's shipping with JDK. (Use JDK8!)

Bean Definition Override

- It is possible to override a bean definition in Spring Container.
- Beans are identified by their names.
- If 2 beans exist with the same identification, the second definition will override the first definition.
- This type of bean override is only possible if bean definitions with the same name are placed into different configuration metadata sources. Spring doesn't allow redefining a bean in the same configuration metadata file or class.

src:spring-bean-definition-override

Accessing Beans outside of the Spring Container

- At some point, we might need to access Spring beans outside of the Spring Container.
- This is doable in a couple of ways.
- For web applications, Spring container provides WebApplicationContextUtils. ApplicationContext can be reached through here.

default-lazy-init
lazy-init

- Spring Container, by default, creates beans during its startup, named as eager bean initialisation.
- Spring also supports lazy bean initialisation.
- With this approach bean creation will be delayed until they are used (referenced from another bean or looked up from context).
- Lazy initialisation can either be set from default-lazy-init attribute in <beans> or lazy-init attribute in <bean>.

default-lazy-init
lazy-init

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

    <bean id="accountService"
class="“tr.edu.hacettepe.bbm490.AccountServiceImpl”>
        <property name="accountDao" ref=“accountDao"/>
    </bean>

    <bean id="accountDao"
class="tr.edu.hacettepe.bbm490.AccountDaoInMemoryImpl"
       lazy-init="false">
    </bean>

</beans>
```

@Lazy

- If @Lazy annotation is present on class level with @Component (or its derivatives) annotation, then that bean definition will be lazy.

```
@Service("accountService")
@Lazy(true)
public class AccountServiceImpl implements AccountService {
//...
}
```

Bean Definition Profiles

- Profiling bean definitions enables the instantiation to be dynamic.
- In development we may use JDBC connection to database and in production we can use a JNDI lookup to connect to our data source defined on weblogic server.
- Since it is forbidden to have 2 bean definitions with same name in a single configuration metadata file, we can use profiling to define a bean with name “datasource” within the same configuration file/class.

Bean Definition Profiles

```
<beans …>
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <beans profile="dev,test">
        <bean id="dataSource"
        class="org.springframework.jdbc.datasource.SingleConnectionDataSource">
            <property name="driverClassName" value="org.h2.Driver" />
            <property name="url" value="jdbc:h2:mem:test" />
            <property name="username" value="sa" />
            <property name="password" value="" />
        </bean>
    </beans>
    <beans profile="prod">
        <bean id="dataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">
            <property name="jndiName" value="java:comp/env/jdbc/DS" />
        </bean>
    </beans>
</beans>
```

Let's create a Spring-ified WebApp Project

- We created Spring-ified Standalone applications 'till now.
- Now let's create a web application that loads Spring's application context while running on Tomcat.
- We'll create a web application as a maven project.
- We'll define spring-web dependency and then add context listener to get the application context up and running.

src:spring-sample-webapp

web dependencies

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.1.5.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>4.1.5.RELEASE</version>
    </dependency>

</dependencies>
```

web.xml

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

    <context-param>
       <param-name>contextConfigLocation</param-name>
       <param-value>/WEB-INF/applicationContext.xml</param-value>
    </context-param>

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

applicationContext.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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-
context-4.0.xsd">

   <context:component-scan base-package="tr.edu.hacettepe.bbm490"/>
   <context:annotation-config />
</beans>
```

Spring AOP

- Spring provides a Java solution for defining method execution join points on Spring beans. Let's go first with the terms.
- We can think of an aspect as the common feature that implements the cross cutting parts of as software system in separate entities. Like logging method execution times.
- AspectJ is a powerful framework for doing AOP.
- We will give the definition of the AOP terms along with working examples.

Spring AOP

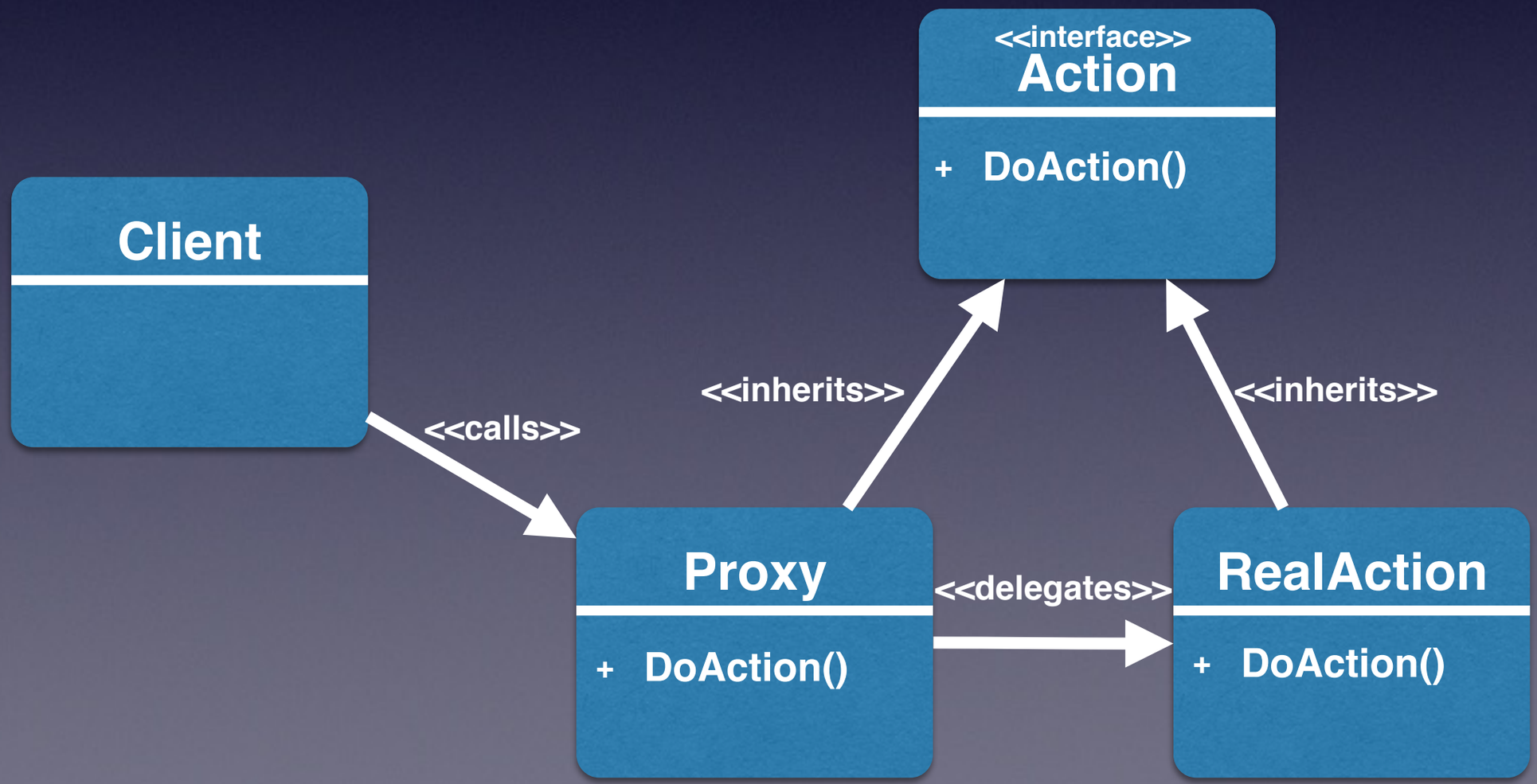
- Join-point: The point within the actual code where the aspect gets executed to insert additional logic into the application.
- Advice: The action—or the chunk of code—that is executed by the aspect at a specific join-point.
- Point-cut: An expression that selects one or more join-points for execution. You can think of a point-cut as a group of join-points.
- Target: The object where its execution flow is modified by an aspect so it is meant to be the actual business logic.

Spring AOP

- Weaving: The process of wiring aspects to the target objects, which can be done at three different levels: compile-time, load-time, or run-time:
- Compile-time weaving is the simplest method. The compiler passes through the source code of the application and creates woven classes.
- Load-time weaving is the process where the specific class loaders weave the class while loading it.
- Run-time weaving is a more dynamic approach compared to the compile-time and load-time weaving processes. Spring AOP uses this method by utilizing the Proxy pattern, which is covered in the next slide.

Spring AOP

- Spring follows the Proxy Pattern.
- You can think of the *proxy objects* as the wrappers around the actual objects, so the features can be introduced *before*, *after,* or *around* the method calls of the originator objects.



Logging Method Execution Times

```
public class ExecutionTimeLoggingSpringAOP implements MethodBeforeAdvice,
AfterReturningAdvice {

    long startTime = 0;

    @Override
    public void before(Method method, Object[] args, Object target) throws
Throwable {
        startTime = System.nanoTime();
    }

    @Override
    public void afterReturning(Object returnValue, Method method, Object[]
args, Object target) throws Throwable {
        long elapsedTime = System.nanoTime() - startTime;
        String className = target.getClass().getCanonicalName();
        String methodName = method.getName();
        System.out.println("Execution of " + className + "#" + methodName
+ " ended in " + new BigDecimal(elapsedTime).divide(new
BigDecimal(1000000)) + " milliseconds");
    }
}
```

Logging Method Execution Times

```
<bean id="executionTimeLoggingSpringAop"
class="tr.edu.hacettepe.bbm490.aspect.ExecutionTimeLoggingSpringAOP" />
```

```
<aop:config>
    <aop:pointcut id="executionTimeLoggingPointcut"
                  expression="execution(public * *(..))" />

    <aop:advisor id="executionTimeLoggingAdvisor"
                 advice-ref="executionTimeLoggingSpringAop"
                 pointcut-ref="executionTimeLoggingPointcut" />
</aop:config>
```

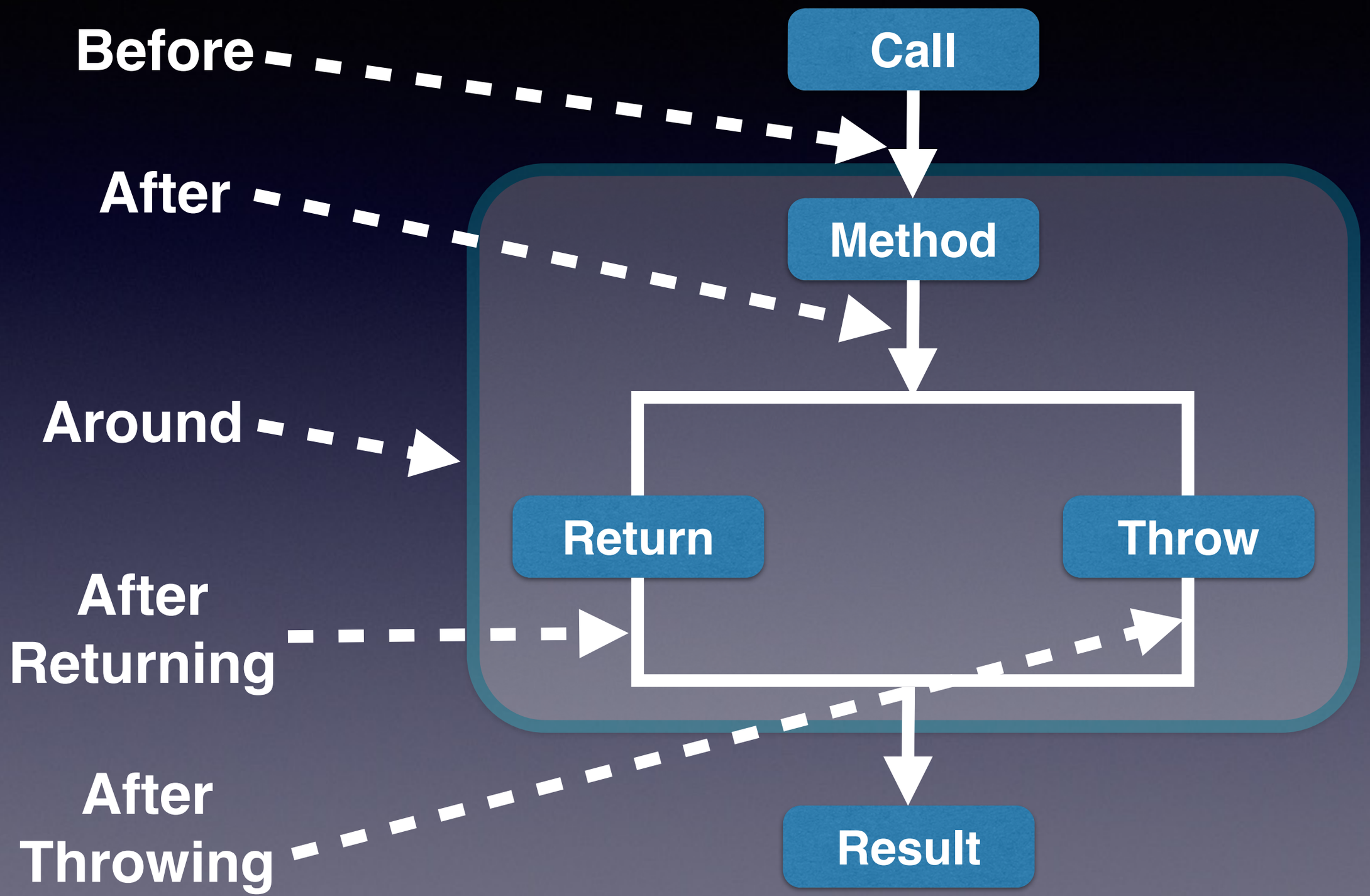
src:spring-exection-time-logging

The List of Advices

Type	Interface	Execution point
Before	`MethodBeforeAdvice`	The advice gets executed before the join-point.
After Returning	`AfterReturningAdvice`	The advice gets executed after the execution of the join-point finishes.
After Throwing	`ThrowsAdvice`	The advice gets executed if any exception is thrown from the join-point.
After (Finally)	N/A	The advice gets executed after the execution of the join-point whether it throws an exception or not.
Around	N/A	The advice gets executed around the join-point, which means that it is invoked before the join-point and after the execution of the join-point.

src:spring-exection-time-logging-around

Execution Flow for Advice Types



Pointcut Designators

- Spring AOP provides various matcher expressions in order to filter methods for applying the advices to Spring beans.
- This matcher expressions are also called pointcut designators.
- For filtering methods according to its types—like interfaces, class names, or package names—Spring AOP provides the `within` keyword.
- For filtering according to the method signatures, the `execution` keyword can be used.
- It's possible to filter beans according to their names with the `bean` keyword.
- It's possible to filter the methods according to an annotation applied on.

within

- within(tr.edu..*):
  This advice will match for all the methods in all classes of the tr.edu package and all of its subpackages.
- within(tr.edu.hacettepe.bbm490.MyService):
  This advice will match for all the methods in the MyService class.
- within(MyServiceInterface+):
  This advice will match for all the methods of classes that implement the MyServiceInterface.
- within(tr.edu.hacettepe.bbm490.MyBaseService+):
  This advice will match for MyBaseService class and for all of its subclasses.

execution

- execution(* tr.edu.hacettepe.bbm490.MyBean.*(..)):
  This advice will match for all the methods of MyBean.
- execution(public * tr.edu.hacettepe.bbm490.MyBean.*(..)):
  This advice will match for all the public methods of MyBean.
- execution(public String tr.edu.hacettepe.bbm490.MyBean.*(..)):
  This advice will match for all the public methods of MyBean that return a String.
- execution(public * tr.edu.hacettepe.bbm490.MyBean.*(long, ..)):
  This advice will match for all the public methods of MyBean with the first parameter defined as long.

bean & annotation

- bean(*Service): The point-cut expression given here will match for the beans that have the suffix Service in their names.
- @annotation(tr.edu.hacettepe.bbm490.MarkerMethodAnnotation): The expression here states that the methods that have the MarkerMethodAnnotation annotation will be advised.

- It's also possible to blend the expressions with grammatical operators: and, or and not (or with &&, ||, and !).

Wildcard	Definition
. .	This wildcard matches any number of arguments within method definitions, and it matches any number of packages within the class
+	This wildcard matches any subclasses of a given class.
*	This wildcard matches any number of characters.

Capitalising on the Power of Annotations

- Doing aspect-oriented development becomes an easy job with the help of Spring AOP and AspectJ cooperation.
- To define an aspect, advice or a point-cut, we did some XML configuration in our examples.
- Since Spring AOP also employs the annotations provided by AspectJ, the definition of the aspects can be done with pure Java code instead of the bloated XML.

- @Before

```
@Before("execution(public * *(..))")
public void before(JoinPoint joinPoint) {
}
```

- @Pointcut

```
@Pointcut("execution(public * *(..))")
public void anyPublicMethod()  {
}

@Before("anyPublicMethod()")
public void beforeWithPointcut(JoinPoint joinPoint) {
}
```

- @Pointcut - Here we stated two point-cut definitions with two different expressions; one applies for any public method with any return type, method name, and parameters, and the other one applies for the methods annotated with the `@MarkerAnnotation`.

```
@Pointcut("execution(public * *(..))")
public void anyPublicMethod()  {
}

@Pointcut("@annotation(tr.edu.hacettepe.bbm490.MarkerAnnotation)")
public void annotatedWithMarkerAnnotation()  {
}

@After(value = "anyPublicMethod() && annotatedWithMarkerAnnotation()")
public void afterWithMultiplePointcut(JoinPoint joinPoint) {
}
```

- @After

```
@After("execution(public * *(..))")
public void after(JoinPoint joinPoint) {
}
```

- @AfterReturning

```
@AfterReturning(value = "execution(public * *(..))")
public void after(JoinPoint joinPoint) {
    System.out.println("After Returning Advice.");
}
```

- @AfterThrowing

```
@AfterThrowing(value = “within(tr.edu.hacettepe.bbm490.MyBean)”,
               throwing = "t")
public void afterThrowing(JoinPoint joinPoint, Throwable t) {
}
```

- @Aspect

```
@Component
@Aspect
public class ExecutionOrderBefore {

    @Before(value = "execution(public * *(..))")
    public void before(JoinPoint joinPoint) {
    }
}
```

- @Around

```
@Around("execution(public * *(..))")
public void around(ProceedingJoinPoint jp) throws Throwable {
    System.out.println("Before proceeding part of the Around advice.");
    jp.proceed();
    System.out.println("After proceeding part of the Around advice.");
}
```