Lesson 10

Core Container

(Classpath Scanning and Managed Components)

Classpath Scanning and Managed Components

- All previous examples in lessons use XML to specify the configuration metadata that produces each BeanDefinition within the Spring container.
- The previous lesson (Annotation-based Configuration) demonstrates how to provide a lot of the configuration metadata through source-level annotations. Even in those examples, however, the "base" bean definitions are explicitly defined in the XML file.
- This lesson describes an option for implicitly detecting the candidate components by scanning the classpath. Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container.
- This removes the need to use XML to perform bean registration. Instead, you can use annotations (for example, @Component).

Classpath Scanning and Managed Components (Ex.)

- Started from Spring 2.5
- Spring provides stereotype annotations:
  - @Component is a generic stereotype for any Spring-managed component.
  - @Repository is specialized for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO).
  - @Service is specialized for service class.
  - @Controller is specialized for presentation layers.
- Therefore, you can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects.

Classpath Scanning and Managed Components (Ex.)

- Using context namespace introduced in Spring 2.5 :

```
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"
```

- Instead of using <annotation-config> Tag.

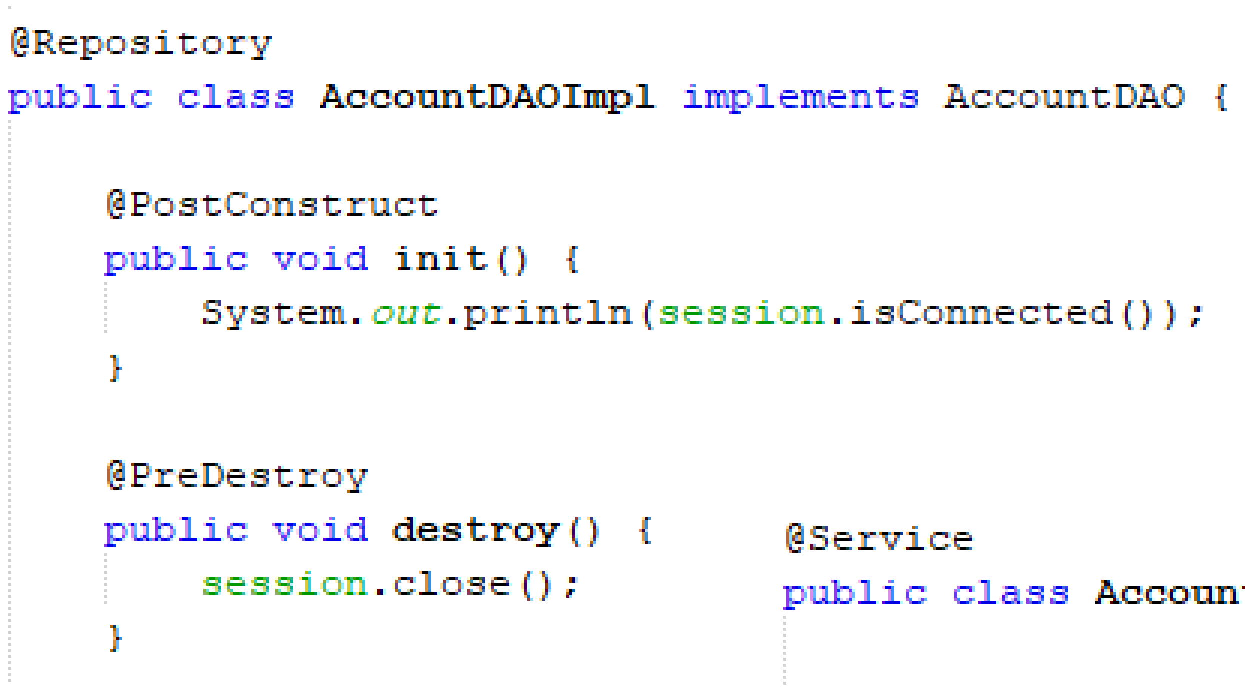
```
<context:annotation-config/>
```

- You use <context:component-scan> Tag and it automatically use <annotation-config> Tag internally.

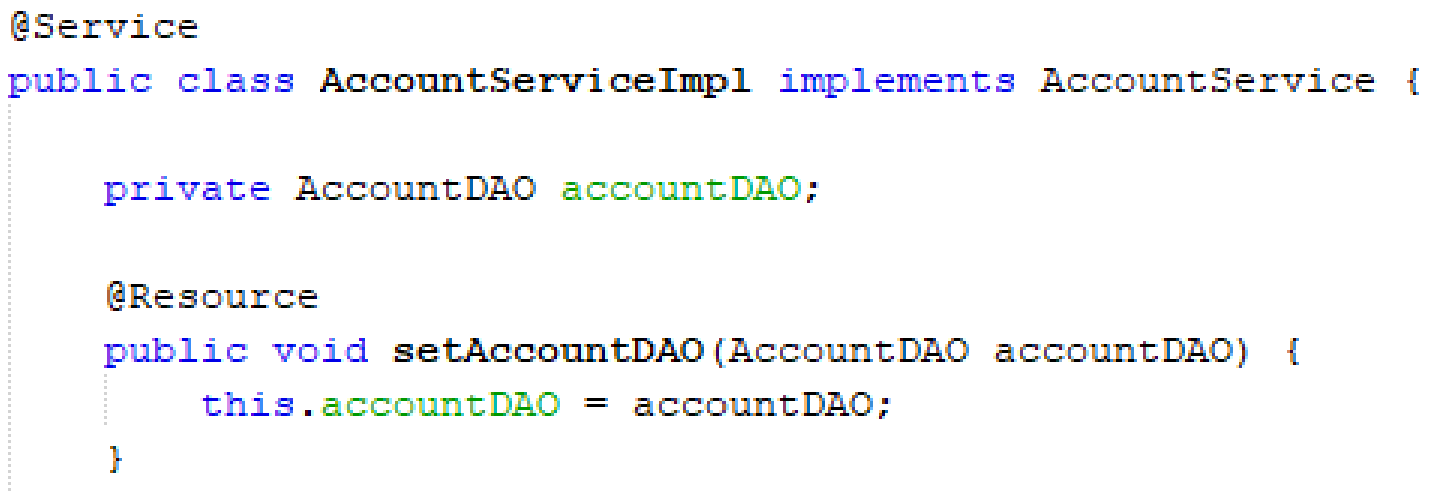
```
<context:component-scan base-package="com.jediver.spring.core"/>
```

Classpath Scanning and Managed Components (Ex.)

- Using context namespace introduced in Spring 2.5 :



```java
@Repository
public class AccountDAOImpl implements AccountDAO {

    @PostConstruct
    public void init() {
        System.out.println(session.isConnected());
    }

    @PreDestroy
    public void destroy() {
        session.close();
    }
```



```java
@Service
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Resource
    public void setAccountDAO(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

Classpath Scanning and Managed Components (Ex.)

- By default, classes annotated with @Component, @Repository, @Service, @Controller, or a custom annotation that itself is annotated with @Component are the only detected candidate components.
- However, you can modify and extend this behavior by applying custom filters.
  - Add them as includeFilters or excludeFilters parameters of the @ComponentScan annotation.
  - Or as include-filter or exclude-filter child elements of the component-scan element.

Classpath Scanning and Managed Components (Ex.)

Filter Type	Example Expression	Description
annotation (default)	org.example.SomeAnnotation	An annotation to be present at the type level in target components.
assignable	org.example.SomeClass	A class (or interface) that the target components are assignable to (extend or implement).
aspectj	org.example..*Service+	An AspectJ type expression to be matched by the target components.
regex	org\.example\.Default.*	A regex expression to be matched by the target components class names.
custom	org.example.MyTypeFilter	A custom implementation of the org.springframework.core.type .TypeFilter interface.

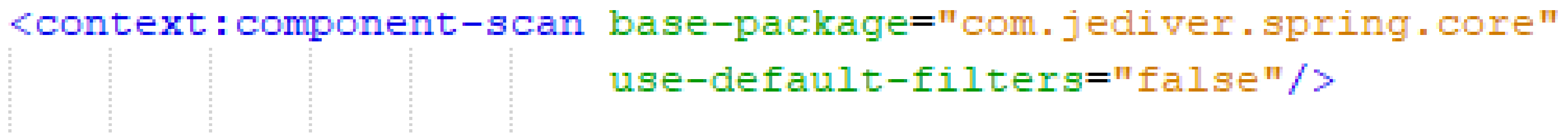
Classpath Scanning and Managed Components (Ex.)

- The following example shows the configuration ignoring all @Repository annotations and using "stub" repositories instead:

```
<context:component-scan base-package="com.jediver.spring.core">
    <context:include-filter type="regex"
            expression=".*Stub.*Repository"/>
    <context:exclude-filter type="annotation"
            expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

Classpath Scanning and Managed Components (Ex.)

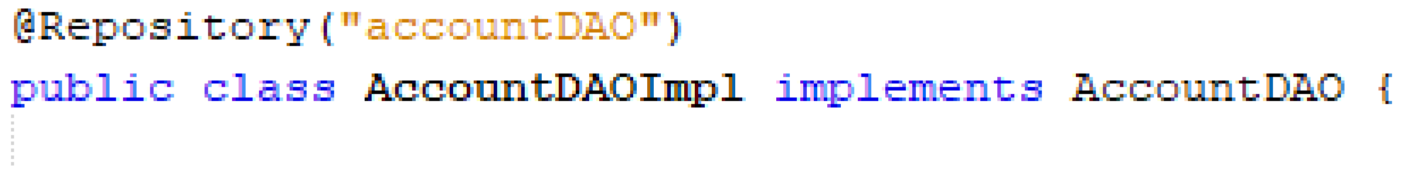
- It is also possible to disable the default filters
  - by providing use-default-filters="false" as an attribute of the <component-scan/> element.
- This will in effect disable automatic detection of classes annotated with @Component, @Repository, @Service, or @Controller.



```xml
<context:component-scan base-package="com.jediver.spring.core"
                        use-default-filters="false"/>
```

Naming autodetected components

- When a component is autodetected as part of the scanning process.
- If 'stereotype' annotation (@Component, … etc) contains a name value, corresponding bean will have that name
- If not, the bean name will be the un-capitalized non-qualified class name.
- Example,
  - if the class name is : Employee
  - Bean name will be : employee



```
@Repository("accountDAO")
public class AccountDAOImpl implements AccountDAO {
```

Autodetected Components Scope

- The default scope is 'singleton'.
- However, there are times when other scopes are needed.
- Therefore Spring 2.5 introduces a new @Scope annotation as well.
- Simply provide the name of the scope within the annotation, such as:

```
@Scope("prototype")
@Repository("accountDAO")
public class AccountDAOImpl implements AccountDAO {
```

Autodetected Components Scope

- Note:
  - If you would like to provide a custom strategy for scope resolution rather than relying on the annotation-based approach.
  - Implement the ScopeMetadataResolver interface, and be sure to include a default no-arg constructor.
  - Then, provide the fully-qualified class name when configuring the scanner:

```java
public class CustomScope implements ScopeMetadataResolver {

    @Override
    public ScopeMetadata resolveScopeMetadata(BeanDefinition bd) {
        //do your logic here and return the scope metadata
        return null;
    }
}
```

```xml
<context:component-scan base-package="com.jediver.spring.core"
    scope-resolver="com.jediver.spring.core.dal.cfg.CustomScope"/>
```

Lesson 11

Core Container
(Using JSR 330 Standard Annotations)

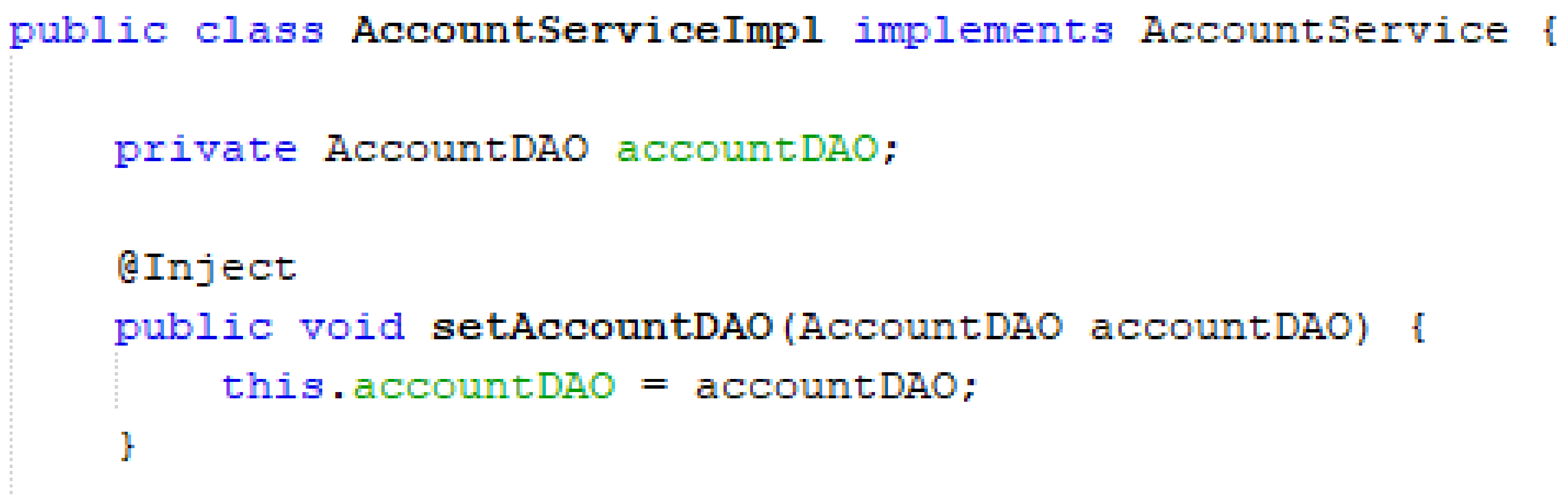
Using JSR 330 Standard Annotations

- Introduced in Spring 3.0,
- Spring offers support for JSR-330 standard annotations (Dependency Injection).
- Those annotations are scanned in the same way as the Spring annotations.
- To use them, you need to have the relevant jars in your classpath.

```
<dependency>
        <groupId>jakarta.inject</groupId>
        <artifactId>jakarta.inject-api</artifactId>
        <version>2.0.1</version>
 </dependency>
```

Dependency Injection with @Inject and @Named

- Instead of @Autowired, you can use @jakrta.inject.Inject as follows:



```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Inject
    public void setAccountDAO(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

Dependency Injection with @Inject and @Named (Ex.)

- As with @Autowired, you can use @Inject at
  - The field level
  - The method level
  - The constructor-argument level.
- Furthermore, you may declare your injection point as a Provider, allowing for on-demand access to beans of shorter scopes or lazy access to other beans through a Provider.get() call.

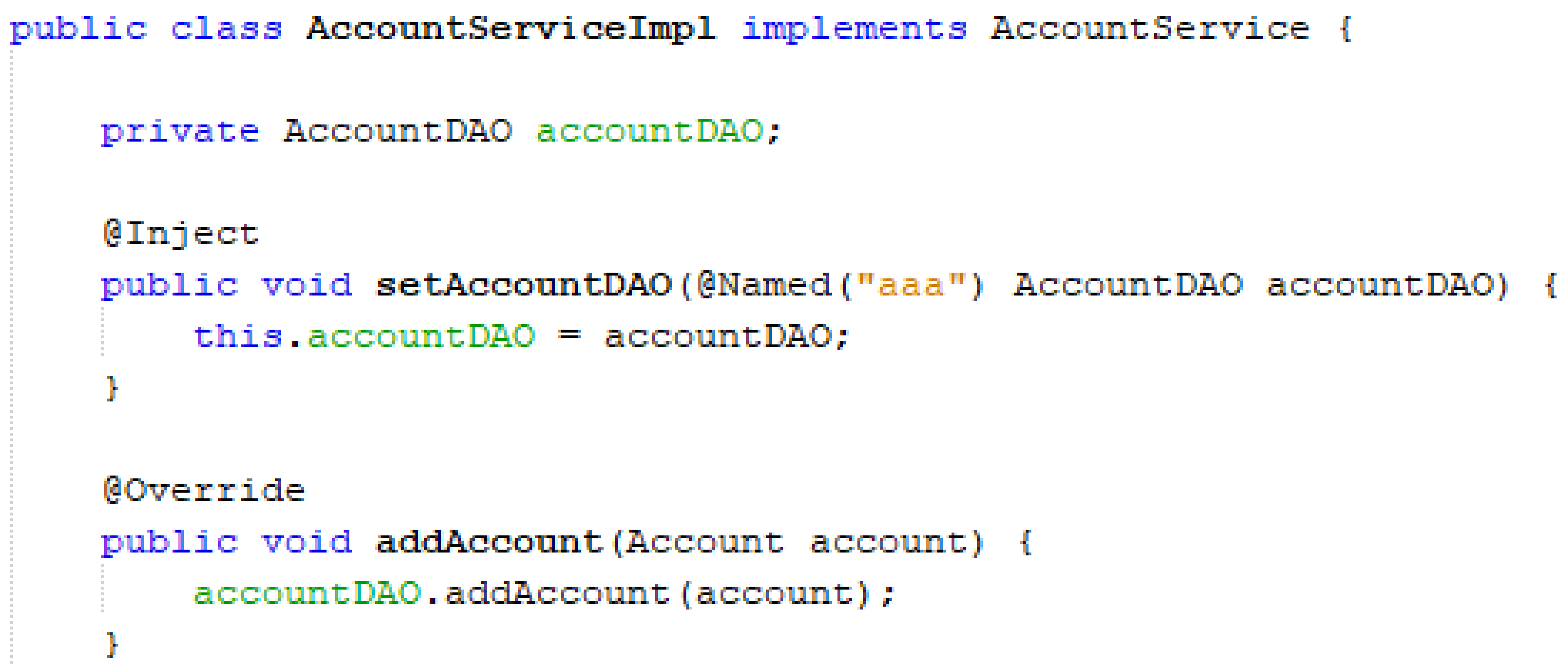
Dependency Injection with @Inject and @Named (Ex.)

- As the Following Example:

```
public class AccountServiceImpl implements AccountService {

    private Provider<AccountDAO> accountDAO;

    @Inject
    public void setAccountDAO(Provider<AccountDAO> accountDAO) {
        this.accountDAO = accountDAO;
    }

    @Override
    public void addAccount(Account account) {
        accountDAO.get().addAccount(account);
    }
```

Dependency Injection with @Inject and @Named (Ex.)

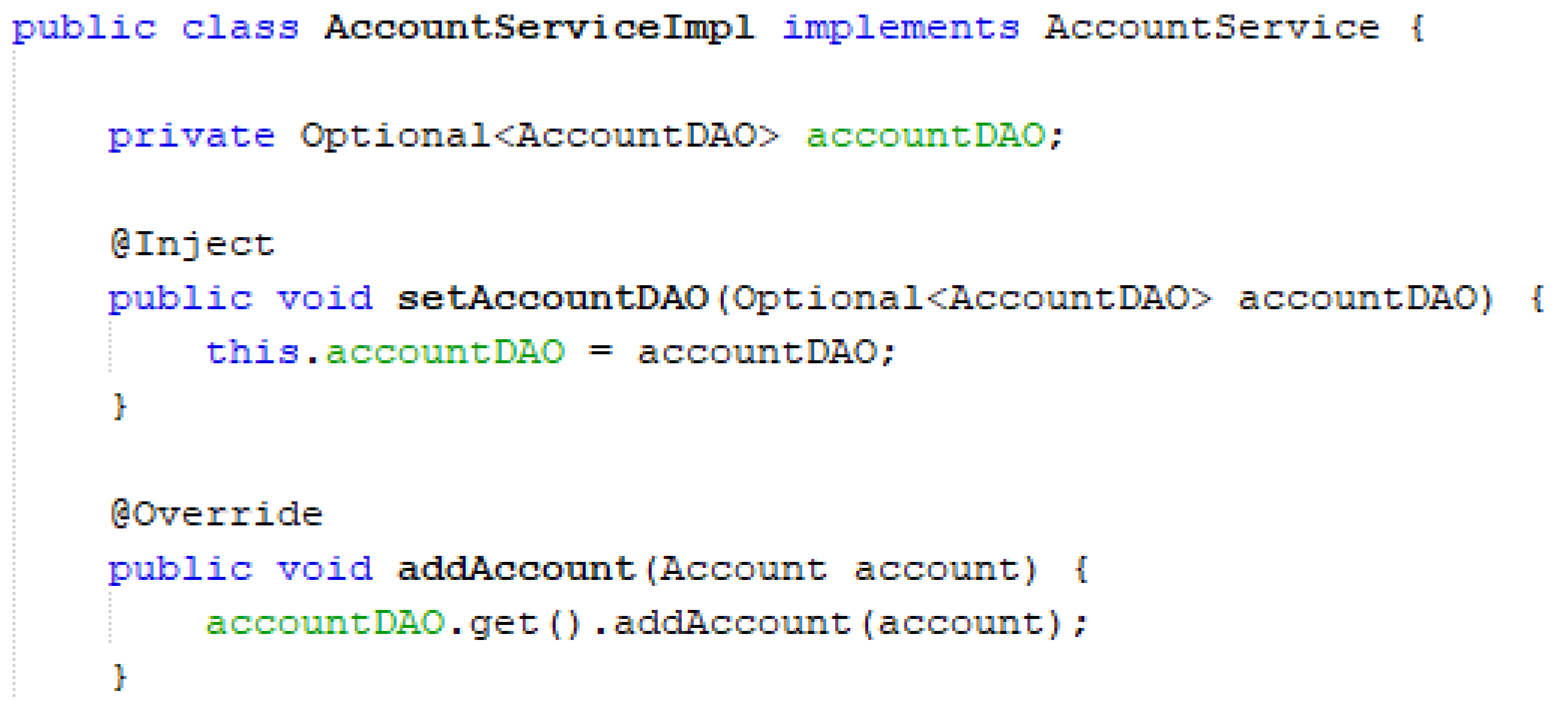
- If you would like to use a qualified name for the dependency that should be injected, you should use the @Named annotation



```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Inject
    public void setAccountDAO(@Named("aaa") AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }

    @Override
    public void addAccount(Account account) {
        accountDAO.addAccount(account);
    }
```

Dependency Injection with @Inject and @Named (Ex.)

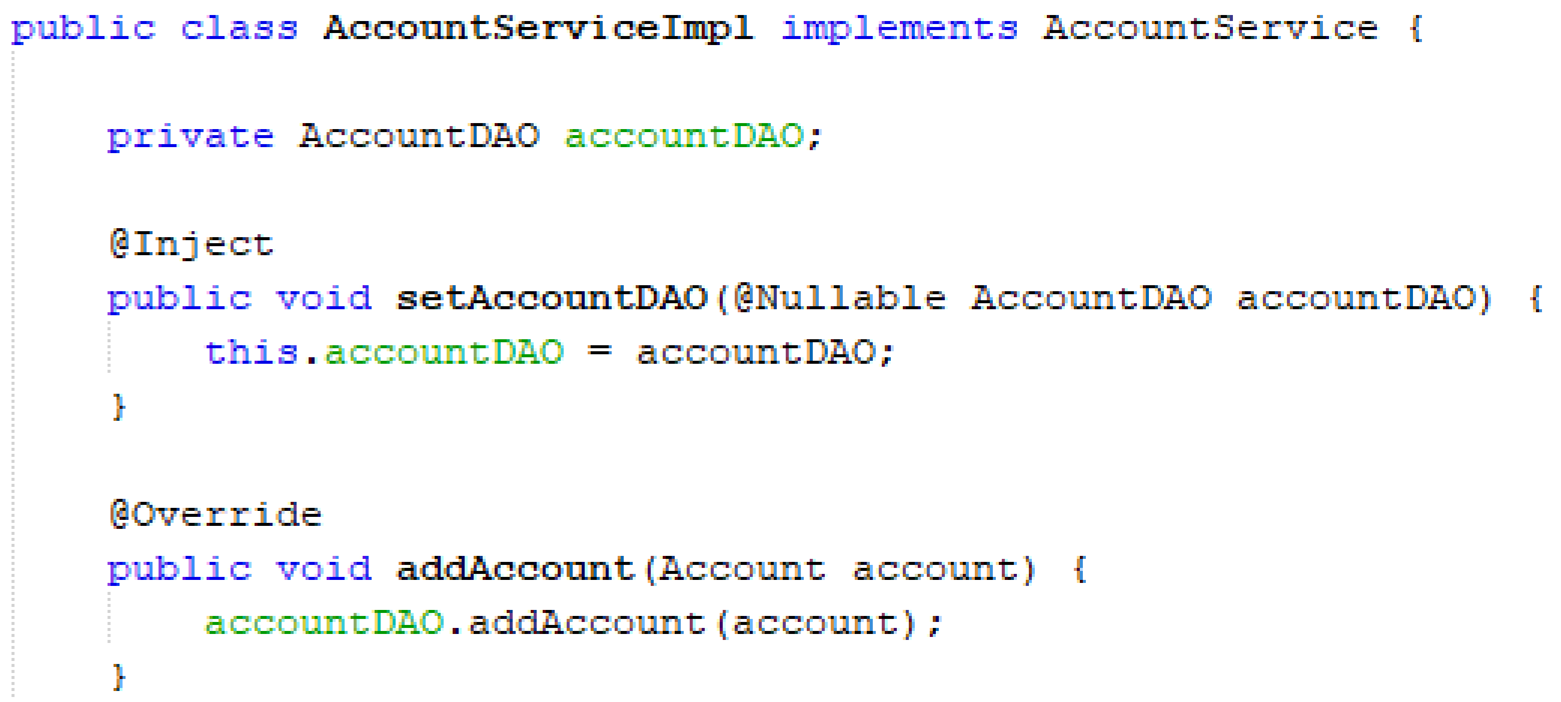
- As with @Autowired, @Inject can also be used with java.util.Optional.
- This is even more applicable here, since @Inject does not have a required attribute.



```java
public class AccountServiceImpl implements AccountService {

    private Optional<AccountDAO> accountDAO;

    @Inject
    public void setAccountDAO(Optional<AccountDAO> accountDAO) {
        this.accountDAO = accountDAO;
    }

    @Override
    public void addAccount(Account account) {
        accountDAO.get().addAccount(account);
    }
```

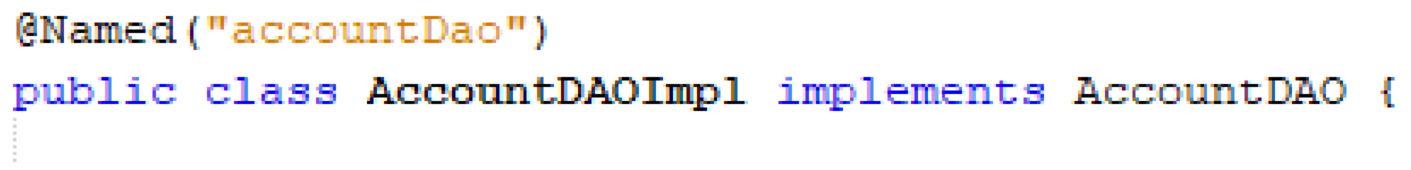
Dependency Injection with @Inject and @Named (Ex.)

- As with @Autowired, @Inject can also be used with @Nullable.
- This is even more applicable here, since @Inject does not have a required attribute.

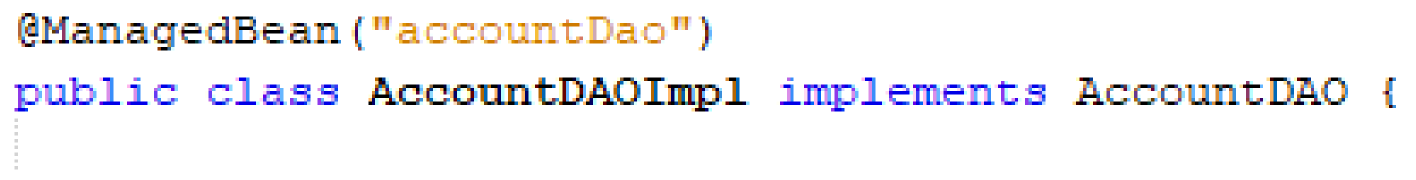

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Inject
    public void setAccountDAO(@Nullable AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }

    @Override
    public void addAccount(Account account) {
        accountDAO.addAccount(account);
    }
```

@Named and @ManagedBean

- Standard Equivalents to the @Component Annotation
- Instead of @Component, you can use @jakrta.inject.Named or jakarta.annotation.ManagedBean.



```
@Named("accountDao")
public class AccountDAOImpl implements AccountDAO {
```



```
@ManagedBean("accountDao")
public class AccountDAOImpl implements AccountDAO {
```

Limitations of JSR-330 Standard Annotations

Spring	jakrta.inject.*	jakrta.inject restrictions / comments
@Autowired	@Inject	• @Inject has no 'required' attribute. • Can be used with Java 8’s Optional instead.
@Component	@Named / @ManagedBean	• JSR-330 does not provide a composable model • Only a way to identify named components.
@Scope("singleton")	@Singleton	• The JSR-330 default scope is like Spring’s prototype. • In order to use a scope other than singleton, you should use Spring’s @Scope annotation. • javax.inject also provides a @Scope annotation. Nevertheless, this one is only intended to be used for creating your own annotations.

Limitations of JSR-330 Standard Annotations

Spring	jakrta.inject.*	jakrta.inject restrictions / comments
@Qualifier	@Qualifier / @Named	• javax.inject.Qualifier is just a meta-annotation for building custom qualifiers. • Concrete String qualifiers (like Spring’s @Qualifier with a value) can be associated through javax.inject.Named.
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent
ObjectFactory	Provider	• javax.inject.Provider is a direct alternative to Spring’s ObjectFactory, only with a shorter get() method name. • It can also be used in combination with Spring’s @Autowired or with non-annotated constructors and setter methods.