Bean Definition Inheritance

- A child bean definition inherits configuration data from a parent definition.
- The child definition can override some values or add others as needed.
- Using parent and child bean definitions can save a lot of typing. Effectively, this is a form of templating.
- If you work with an ApplicationContext interface programmatically, child bean definitions are represented by the ChildBeanDefinition class.
- Most users do not work with them on this level. Instead, they configure bean definitions declaratively in a class such as the ClassPathXmlApplicationContext.
- When you use XML-based configuration metadata, you can indicate a child bean definition by using the parent attribute, specifying the parent bean as the value of this attribute.

Bean Definition Inheritance (Ex.)

```java
public class Parent {

    private String name;
    private int age;

    public String getName() {...3 lines }

    public void setName(String name) {...3 lines }

    public int getAge() {...3 lines }

    public void setAge(int age) {...3 lines }

    @Override
    public String toString() {...3 lines }

}
```

```java
public class Child {

    private String name;
    private int age;
    private String address;

    public String getName() {...3 lines }

    public void setName(String name) {...3 lines }

    public int getAge() {...3 lines }

    public void setAge(int age) {...3 lines }

    public String getAddress() {...3 lines }

    public void setAddress(String address) {...3 lines }

    @Override
    public String toString() {...3 lines }

}
```

Bean Definition Inheritance (Ex.)

Java™ Education and Technology Services

```
<bean id="parent" abstract="true"
      class="com.jediver.spring.core.bean.inheritance.Parent">
    <property name="name" value="Parent Name"/>
    <property name="age" value="20"/>
</bean>
```

```
<bean id="child" parent="parent"
      class="com.jediver.spring.core.bean.inheritance.Child">
    <property name="name" value="Child Name"/>
    <property name="address" value="Cairo"/>
</bean>
```

Bean Definition Inheritance (Ex.)

- A child bean definition uses the bean class from the parent definition if none is specified but can also override it.
- If the child bean class must be compatible with the parent:
  - It must accept the parent's property values.
- A child bean definition inherits from parent:
  - scope
  - constructor argument values
  - property values
  - initialization method
  - destroy method
  - static factory method

Bean Definition Inheritance (Ex.)

- A child bean definition can also override all inherited settings.
- The remaining settings are always taken from the child definition:
  - depends on
  - autowire mode
  - dependency check
  - singleton
  - lazy init.
- The preceding example explicitly marks the parent bean definition as abstract by using the abstract attribute.

Bean Definition Inheritance (Ex.)

- If the parent definition does not specify a class, explicitly marking the parent bean definition as abstract is required.

```
<bean id="parent" abstract="true">
    <property name="name" value="Parent Name"/>
    <property name="age" value="20"/>
</bean>
```

```
<bean id="child" parent="parent"
      class="com.jediver.spring.core.bean.inheritance.Child">
    <property name="name" value="Child Name"/>
    <property name="address" value="Cairo"/>
</bean>
```

Collections

- The Collections are:
  - <list/> element.
  - <set/> element.
  - <map/> element.
  - <props/> element.
- Supports merging collections.
- If you merge in case of a parent can have <list/>, <map/>, <set/> or <props/> element and child have <list/>, <map/>, <set/> or <props/> elements inherit and override values from the parent collection.
- That is, the child collection’s values are the result of merging the elements of the parent and child collections.

Collections <list/> element.

```
<bean id="user"
          class="com.jediver.spring.core.bean.di.User">
    <constructor-arg index="1" value="Medhat"/>
    <constructor-arg index="0" value="Ahmed"/>
</bean>
<bean id="parent" abstract="true"
      class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails2">
        <list>
            <value>Hello World</value>
            <ref bean="user"/>
        </list>
    </property>
</bean>
```

```
Hello World
User{name=AhmedMedhat, balance=0.0}
Hello World 2
User{name=AhmedMedhat, balance=0.0}
```

```
<bean id="child" parent="parent">
    <property name="adminEmails2">
        <list merge="true">
            <value>Hello World 2</value>
            <ref bean="user"/>
        </list>
    </property>
</bean>
```

Collections <set/> element.

```
<bean id="user"
      class="com.jediver.spring.core.bean.di.User">
    <constructor-arg index="1" value="Medhat"/>
    <constructor-arg index="0" value="Ahmed"/>
</bean>
<bean id="parent" abstract="true"
      class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails4">
        <set>
            <value>Hello World</value>
            <ref bean="user"/>
        </set>
    </property>
</bean>
```

```
Hello World
User{name=AhmedMedhat, balance=0.0}
Hello World 2
```

```
<bean id="child" parent="parent">
    <property name="adminEmails4">
        <set merge="true">
            <value>Hello World 2</value>
            <ref bean="user"/>
        </set>
    </property>
</bean>
```

Collections <map/> element.

```
<bean id="parent" abstract="true"
          class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails3">
        <map>
            <entry key="administrator" value="administrator@example.com"/>
            <entry key="support" value="support@example.com"/>
        </map>
    </property>
</bean>
```

```
administrator@example.com
support@example.co.uk
sales@example.com
```

```
<bean id="child" parent="parent">
    <property name="adminEmails3">
        <map merge="true">
            <entry key="sales" value="sales@example.com"/>
            <entry key="support" value="support@example.co.uk"/>
        </map>
    </property>
</bean>
```

Collections <props/> element.

```
<bean id="parent" abstract="true"
      class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.com</prop>
            <prop key="support">support@example.com</prop>
        </props>
    </property>
</bean>
```

```
support@example.co.uk
administrator@example.com
sales@example.com
```

```
<bean id="child" parent="parent">
    <property name="adminEmails">
        <!-- the merge is specified on the child collection definition -->
        <props merge="true">
            <prop key="sales">sales@example.com</prop>
            <prop key="support">support@example.co.uk</prop>
        </props>
    </property>
</bean>
```

Collections (Ex.)

- This merging behavior applies similarly to the <list/>, <map/>, and <set/> collection types.
- In <list/> The parent's values precede all of the child list's values in .
- In <map/> & <set/>, no ordering exists.
- **Limitations of Collection Merging**
  - You cannot merge different collection types (such as a Map and a List). If you do attempt to do so, an appropriate Exception is thrown.
  - The merge attribute must be specified on the lower, inherited, child definition.
  - Specifying the merge attribute on a parent collection definition is redundant and does not result in the desired merging

PropertyPlaceholderConfigurer

- The PropertyPlaceholderConfigurer is used to externalize property values from an XML configuration file.
- Into another separate file in the standard Java Properties format.
- This is useful to allow the person deploying an application to customize environment-specific properties,
- without the risk of modifying the main XML definition file or files for the container.

PropertyPlaceholderConfigurer (Ex.)

- You can convert the text inside the <value> element into a Properties instance using the PropertyEditor.

```
<bean id="mappings"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="properties">
        <value>
            jdbc.driverClassName= com.mysql.jdbc.Driver
            jdbc.url= jdbc:mysql://localhost:3306/biddingschema
            jdbc.username= root
            jdbc.password= root
        </value>
    </property>
</bean>
```

PropertyPlaceholderConfigurer (Ex.)

- A DataSource with placeholder values is defined.
- It will configure some properties from an external Properties file, and at runtime.
- The actual values come from another file in the standard Java Properties format.

```
jdbc.driverClassName= com.mysql.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3306/biddingschema
jdbc.username= root
jdbc.password= root
```

PropertyPlaceholderConfigurer (Ex.)

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <value>com/jediver/spring/core/cfg/jdbc.properties</value>
    </property>
</bean>
<bean id="dataSource" destroy-method="close"
     class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

PropertyPlaceholderConfigurer (Ex.)

- You can create it in two way
  - Direct way with define bean definition for PropertyPlaceholderConfigurer class in container
  - With the 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"
```

- Multiple locations may be provided as a comma-separated list for the location attribute.

```
<context:property-placeholder
        location="com/jediver/spring/core/cfg/jdbc.properties"/>
```

PropertyPlaceholderConfigurer (Ex.)

- The PropertyPlaceholderConfigurer look for properties in
  - The Properties file.
  - Checks the Java System properties.
    - If it cannot find a property you are trying to use.

PropertyPlaceholderConfigurer (Ex.)

- Set how to check system properties: as (fallback, override, or never).
- For example, will resolve ${user.dir} to the "user.dir" system property.
  - The default is "fallback":
    - If not being able to resolve a placeholder with the specified properties, a system property will be tried.
  - "override"
    - will check for a system property first, before trying the specified properties.
  - "never"
    - will not check system properties at all.

Lesson 9

Core Container
(Annotation-based Configuration)

Annotations VS. XML for configuring Spring?

- The introduction of annotation-based configuration raised the question of whether this approach is "better" than XML.
  - The short answer is "it depends".
  - The long answer is that each approach has its pros and cons, and usually, it is up to the developer to decide which strategy suits them better. Due to the way they are defined,
  - Annotations provide a lot of context in their declaration, leading to shorter and more concise configuration.
  - However, XML excels at wiring up components without touching their source code or recompiling them.
  - Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.
- No matter Your choice, Spring can accommodate both styles and even mix them together.

Annotation-based Configuration

- Spring 2.0 introduced
  - @Required annotation to enforce required properties.
- Spring 2.5 introduced
  - @Autowired annotation provides the capabilities of autowiring between dependencies as replacement to "autowire" attribute in xml but with more fine-grained control and wider applicability.
  - Also added support for JSR-250 annotations, such as
    - @PostConstruct and @PreDestroy.
- Spring 3.0 introduced
  - Added support for JSR-330 annotations (Dependency Injection for Java) annotations contained in the javax.inject package such as @Inject and @Named.

Annotation-based Configuration (Ex.)

- Use of these annotations also requires that certain specialized classes be registered within the Spring container.
- These can be registered as individual bean definitions:
  - RequiredAnnotationBeanPostProcessor.
  - AutowiredAnnotationBeanPostProcessor.
  - CommonAnnotationBeanPostProcessor.
  - PersistenceAnnotationBeanPostProcessor.
- OR implicitly registered by including the following tag in an XML-based
  - <annotation-config> in context namespace this implicitly registered all above BeanPostProcessor except RequiredAnnotationBeanPostProcessor.

Annotation-based Configuration (Ex.)

- These can be registered as individual bean definitions:
  - RequiredAnnotationBeanPostProcessor.

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"/>
```

  - AutowiredAnnotationBeanPostProcessor.

```
<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>
```

  - CommonAnnotationBeanPostProcessor.

```
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>
```

  - PersistenceAnnotationBeanPostProcessor.

```
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>
```

Annotation-based Configuration (Ex.)

- OR 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"
```

- <annotation-config> Tag.

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

Annotation-based Configuration (Ex.)

- @Required these can be registered as bean for spring 5.1:
  - RequiredAnnotationBeanPostProcessor.

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"/>
```

  - This annotation indicates that the affected bean property must be populated at configuration time.
  - The @Required annotation applies to bean property setter methods, as in the following example:

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

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

Annotation-based Configuration (Ex.)

- @Autowired :
  - @Autowired annotation provides the capabilities of autowiring but with more fine-grained control.
  - You can apply the @Autowired annotation to constructors.

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Autowired
    public AccountServiceImpl(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

  - Starting from Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean defines only one constructor to begin with.
  - Autowiring are applied by type.

Annotation-based Configuration (Ex.)

- @Autowired :
  - Also You can apply the @Autowired annotation to property itself.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDAO accountDAO;
```

  - Also you don't have to implement setter for this property
  - Autowiring are applied by type.

```
<bean id="accountDao"
      class="com.jediver.spring.core.dal.dao.impl.AccountDAOImpl">
    <constructor-arg ref="session"/>
</bean>
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - Also You can apply the @Autowired annotation to "traditional" setter methods.

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

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

```
<bean id="accountDao"
      class="com.jediver.spring.core.dal.dao.impl.AccountDAOImpl">
    <constructor-arg ref="session"/>
</bean>
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - Also You can apply the @Autowired annotation to any setup methods.
  - These methods are called automatically called from spring container and autowire all parameters in these method.

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Autowired
    public void setup(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;

    }
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - Also You can apply the @Autowired annotation to any setup methods.
  - These methods are called automatically called from spring container and autowire all parameters in these method.

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;
    private Connection connection;

    @Autowired
    public void setup(AccountDAO accountDAO, Connection connection) {
        this.accountDAO = accountDAO;
        this.connection = connection;
    }
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - You can also provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method.
  - Expects an array of that type.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private Account[] accounts;
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - You can also provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method.
  - Expects a List of that type.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private List<Account> accounts;
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - You can also provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method.
  - Expects a Set of that type.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private Set<Account> accounts;
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - You can also provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method.
  - Expects a Map of that type.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private Map<String, Account> accounts;
```

  - By default, the autowiring fails whenever zero candidate beans are available.

Annotation-based Configuration (Ex.)

- The default behavior is to treat annotated methods, constructors, and fields as indicating required dependencies.

- You can change this behavior by define required attribute.

```
public class AccountServiceImpl implements AccountService {

    @Autowired(required = false)
    private AccountDAO accountDAO;
```

Annotation-based Configuration (Ex.)

- Only one annotated constructor per-class can be marked as required, but multiple non-required constructors can be annotated.
  - In that case, each is considered among the candidates and Spring uses the greediest constructor whose dependencies can be satisfied — that is, the constructor that has the largest number of arguments.
- Ex. Other constructors marked @Autowired(required=false).
- The required attribute of @Autowired is recommended over the @Required annotation.

Annotation-based Configuration (Ex.)

- Alternatively, you can express the non-required nature of a particular dependency through Java 8's java.util.Optional

```
public class AccountServiceImpl implements AccountService {

    private Optional<AccountDAO> accountDAO;

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

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

Annotation-based Configuration (Ex.)

- Starting from Spring Framework 5.0, you can also use a @Nullable annotation (of any kind in any package.
- Ex. jakarta.annotation.Nullable from JSR-305):

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

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

Annotation-based Configuration (Ex.)

- You can also use @Autowired for interfaces that are well-known resolvable dependencies:
  - BeanFactory
  - ApplicationContext
  - Environment
  - ResourceLoader
  - ApplicationEventPublisher
  - MessageSource.
- These interfaces and their extended interfaces, such as ConfigurableApplicationContext or ResourcePatternResolver, are automatically resolved, with no special setup necessary.

Annotation-based Configuration (Ex.)

- The following example autowires an ApplicationContext object:

```
public class AccountDAOImpl implements AccountDAO {

    @Autowired
    private ApplicationContext context;
```

Annotation-based Configuration (Ex.)

- Since autowiring by type may lead to multiple candidates.
- It is necessary to have more control over the selection process.
  - You can do that by define primary attribute for bean definition.
- The other way is to accomplish this is with Spring's @Qualifier annotation.
- When you need more control over the selection process, you can use Spring’s @Qualifier annotation.

Annotation-based Configuration (Ex.)

- @Qualifier By id or name of bean definition:

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    @Qualifier("accountDao")
    private AccountDAO accountDAO;
```

- @Qualifier By qualifier value specified in bean definition:

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    @Qualifier("mainAccountDao")
    private AccountDAO accountDAO;
```

```
<bean id="accountDao"
      class="com.jediver.spring.core.dal.dao.impl.AccountDAOImpl">
    <constructor-arg ref="session"/>
    <qualifier value="mainAccountDao"/>
</bean>
```

Annotation-based Configuration (Ex.)

- @Qualifier also applied for setup methods parameters:

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Autowired
    @Qualifier("mainAccountDao")
    public void setup(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

```
<bean id="accountDao"
      class="com.jediver.spring.core.dal.dao.impl.AccountDAOImpl">
    <constructor-arg ref="session"/>
    <qualifier value="mainAccountDao"/>
</bean>
```

Annotation-based Configuration (Ex.)

- You may create your own custom qualifier annotations as well.
- Simply define an annotation and provide the @Qualifier annotation with your own attributes.

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface DAOQualifier {

    String name();

    Mobile mobile();

}
```

- Mobile is an enum:

```
public enum Mobile {
    Etisalat, Vodafone, Orange
}
```

Annotation-based Configuration (Ex.)

- You can refer now to your bean definition by your custom qualifier.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    @DAOQualifier(name = "ahmed", mobile = Mobile.Etisalat)
    private Account firstAccount;

    @Autowired
    @DAOQualifier(name = "ahmed", mobile = Mobile.Vodafone)
    private Account secondAccount;

    @Autowired
    @DAOQualifier(name = "mohamed", mobile = Mobile.Etisalat)
    private Account thirdAccount;
```

Annotation-based Configuration (Ex.)

```
<bean id="account1" class="com.jediver.spring.core.dal.entity.Account">
    <qualifier type="com.jediver.spring.core.dal.dao.DAOQualifier">
        <attribute key="name" value="ahmed"/>
        <attribute key="mobile" value="Etisalat"/>
    </qualifier>
</bean>
<bean id="account2" class="com.jediver.spring.core.dal.entity.Account">
    <qualifier type="com.jediver.spring.core.dal.dao.DAOQualifier">
        <attribute key="name" value="ahmed"/>
        <attribute key="mobile" value="Vodafone"/>
    </qualifier>
</bean>
<bean id="account3" class="com.jediver.spring.core.dal.entity.Account">
    <qualifier type="com.jediver.spring.core.dal.dao.DAOQualifier">
        <attribute key="name" value="mohamed"/>
        <attribute key="mobile" value="Etisalat"/>
    </qualifier>
</bean>
```

Annotation-based Configuration (Ex.)

- Spring also supports injection by using the JSR-250 @Resource annotation (jakarta.annotation.Resource) on fields or bean property setter methods.
- @Resource is a common pattern in Java EE (Ex. EJB, JSF-managed beans, JAX-WS endpoints).
- Also spring supports this pattern for Spring-managed objects as well.
- @Resource if a name specified,
  - Spring interprets that value as the bean name to be injected.

```
public class AccountServiceImpl implements AccountService {

    @Resource(name = "accountDao")
    private AccountDAO accountDAO;
```

Annotation-based Configuration (Ex.)

- @Resource :
  - Also You can apply the @Resource annotation to "traditional" setter methods.

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

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

```
<bean id="accountDao"
      class="com.jediver.spring.core.dal.dao.impl.AccountDAOImpl">
    <constructor-arg ref="session"/>
</bean>
```

Annotation-based Configuration (Ex.)

- @Resource If no name is specified, the default name is derived from the field name or setter method.
  - In case of a field, it takes the field name.
  - In case of a setter method, it takes the bean property name.

```
public class AccountServiceImpl implements AccountService {

    @Resource
    private AccountDAO accountDAO;
```

- @Resource If no name is specified and didn't find match for fieldName or proertyName, then finds a primary type match instead of a specific named bean.

Annotation-based Configuration (Ex.)

- You can also use @Resource for interfaces that are well-known resolvable dependencies:
  - BeanFactory
  - ApplicationContext
  - Environment
  - ResourceLoader
  - ApplicationEventPublisher
  - MessageSource.
- These interfaces and their extended interfaces, such as ConfigurableApplicationContext or ResourcePatternResolver, are automatically resolved, with no special setup necessary.

Annotation-based Configuration (Ex.)

- The following example autowires an ApplicationContext object by @Resource:

```
public class AccountDAOImpl implements AccountDAO {

    @Resource
    private ApplicationContext context;
```

Annotation-based Configuration (Ex.)

- The CommonAnnotationBeanPostProcessor not only recognizes the @Resource annotation but also the JSR-250 lifecycle annotations:
  - jakarta.annotation.PostConstruct
  - jakarta.annotation.PreDestroy.

- Introduced in Spring 2.5, the support for these annotations offers an alternative to the lifecycle callback mechanism described in initialization callbacks and destruction callbacks.

Annotation-based Configuration (Ex.)

```
public class AccountDAOImpl implements AccountDAO {

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

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

- Like @Resource, the @PostConstruct and @PreDestroy annotation types were a part of the standard Java libraries from JDK 6 to 8.
- However, the entire jakarta.annotation package got separated from the core Java modules in JDK 9 and eventually removed in JDK 11. If needed, the jakarta.annotation-api artifact needs to be obtained via Maven Central now.