



Lesson 6: Spring: Container & Dependency Injection

Maharishi University of Management
Computer Science Department
Orlando Arrocha
July 2017

Orlando Arrocha
July 2017

July 2017

CS544: Enterprise Architecture



© 2016 Maharishi University of Management, Fairfield, Iowa
All rights reserved.

No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi University of Management.

Course Overview Chart

Enterprise Architecture Actions in Accord with Laws of Nature

Structuring Software on Layers to Support Building Robust Enterprise Applications

Week		Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Theme I Hibernate Rest and Activity are the Steps of Progress	AM	Introduction to EA Wholeness is contained in every part	Entity Mapping, Inheritance, Collections Persistence API Outer depends on Inner	Associations & Complex Mapping Knowledge has organizing power	Transactions, Concurrency & JPQL Thought leads to action-achievement-fulfillment	Optimization & Integration Enjoy greater efficiency and accomplish more	Exam
	PM	Lab 1 Setup Env	Lab 2 Creating Entities	Lab 3 Associations	Lab 4 Query Data	Review	Extra Credit Hibernate
	Eve	Reading					
Theme II Spring Do Less and Accomplish More	AM	Container & DI Purification leads to progress	Service Layer & AOP Life is found in layers	Scheduling, Hibernate & Tx Harmony exists in diversity	Spring MVC & Validation Do Less and Accomplish More	Spring Security & Spring Data Seek the highest	Exam
	PM	Lab 5 DI	Lab 6 AOP	Lab 7 Transactions	Lab 8 Web MVC	Review	Extra Credit Spring
	Eve						
Theme III Integration The Whole is Greater than the Sum of the Parts	AM	JMS, AMQP & RMI Order is present everywhere	Spring Web Services Rest and activity are the steps of progress	Web Flow, Mobile & Integration Knowledge is gained from inside and outside	Microservices with Spring Knowledge is structured in consciousness	Project ... action leads to achievement ...	
	PM	Lab 9 Security & AMQP	Lab 10 REST WS	Lab 11 Web Flow	Project Thought leads to action		
	Eve						
Theme IV Project The Field of All Possibilities is the Source of All Solutions	AM	Project ...		Project Presentations ... and achievement leads to fulfillment			
	PM						
	Eve						

Wholeness of the Lesson

Lesson 6

SPRING CONTAINER & DEPENDENCY INJECTION

Purification leads to progress

Injecting the object dependencies based on a system configuration makes the system very flexible. The object initialization is taken care of by the container and the any change in the configuration will be reflected on the objects used by the application.

Science of Consciousness: *The inward stroke during the practice of the Transcendental Meditation technique results in deep rest that provides the momentum for the mind to come out, infused with greater creative awareness.*



Main Points

- 1) Through a configuration file, the Inversion of Control Container manages the lifecycle for the objects required by our application, allowing us to focus on the functionality of our logic and giving flexibility for future implementations.
Science of Consciousness: *by taking the holistic field of life, the pure nature of creative intelligence, we can enrich all aspects of life.*
- 2) Dependency injection allows us to support better separation of concerns and creates more malleable applications. ***Science of Consciousness:*** *a person growing in creative intelligence experiences purification of the nervous system, thus producing a better quality of consciousness.*



Overview

- We will cover
 - The application context, and the configuration of beans
 - We will talk about the basics of dependency injection
 - We will see various ways to perform dependency injection
 - Setter injection
 - Constructor injection
 - Auto-wiring
 - How to inject
 - Objects
 - Values
 - Collections
 - Also how to define DI
 - Inheritance
 - Using Annotations
 - Classpath Scanning



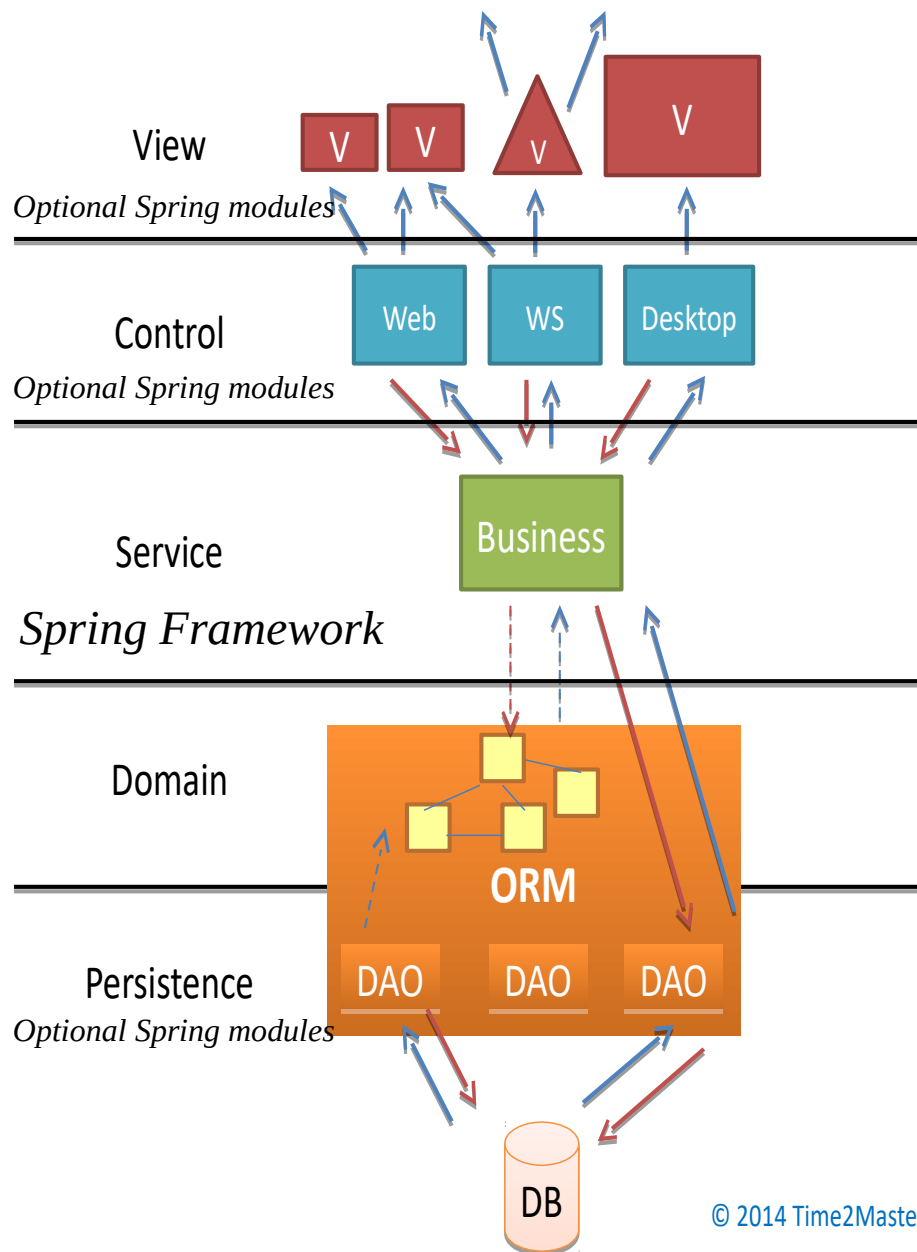
Spring Container



Any intelligent fool can make things bigger, more complex, and more violent.
It takes a touch of genius
– and a lot of courage –
to move in the opposite direction.

– **Albert Einstein**

Spring Framework



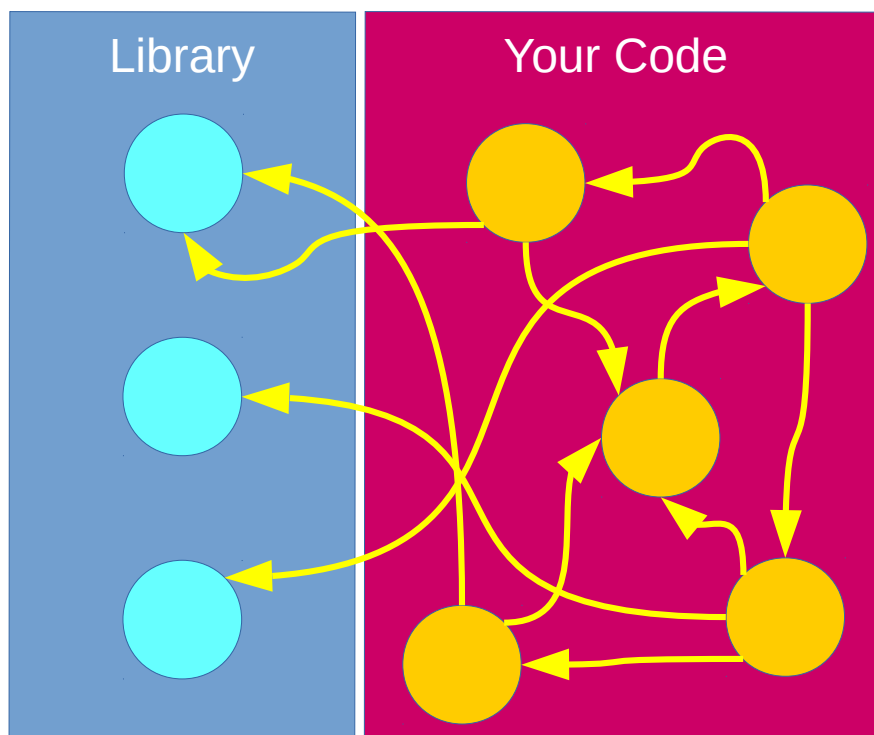
- A lightweight solution for building enterprise applications
- Promotes development best practices to improve design, productivity, portability, and maintainability of the code
- Modular, allowing you to use only those features required by your application
- Inversion of Control container
- POJO based programming
- Enterprise-class transaction management
- Spring leverages existing APIs and frameworks (including Java EE)
- Open Source – communities for every module

Inversion of Control

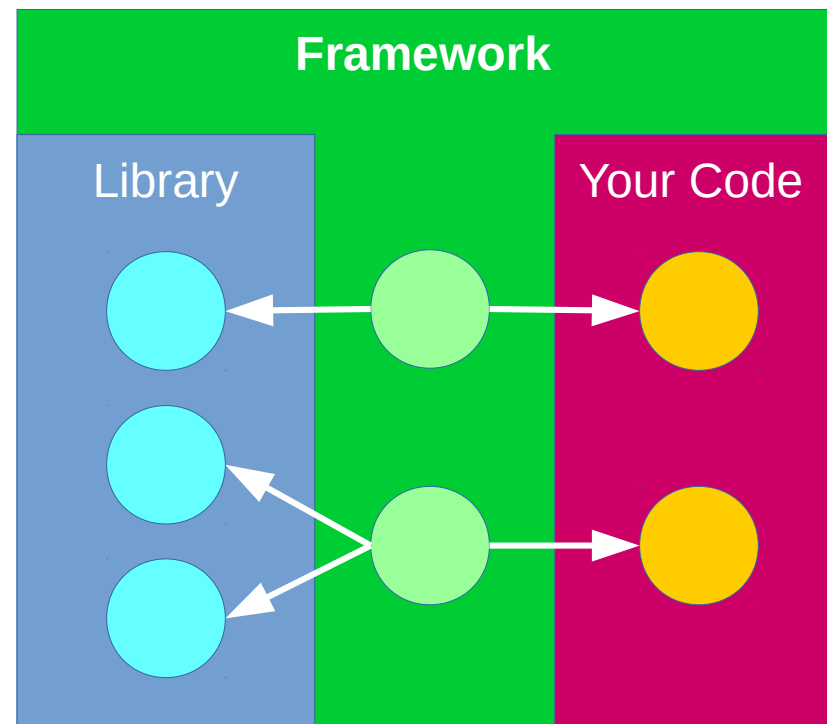
- The framework has control over your code
- You have to do less code to accomplish more
- Custom code has more flexibility



No Framework



Framework



Spring Example with XML Configuration

MainApp.java

```
public class MainApp {  
    public static void main(String[] args) {  
        ApplicationContext context =  
            New ClassPathXmlApplicationContext(  
                "application-context.xml");  
  
        HelloPerson hp = context.getBean("helloWorld");  
        hp.sayHello();  
    }  
}
```

HelloPerson.java

```
public class HelloPerson {  
    private Person person;  
  
    public void setPerson(Person person){  
        this.person = person;  
    }  
    public sayHello() {  
        System.out.println("Hello "  
            + person.getFirstname() + " "  
            + person.getLastname());  
    }  
    ...  
}
```

1. Create the application context
2. Get the beans

Person.java

```
public class Person {  
    private String firstname;  
    private String lastname;  
    private int id;  
  
    public Person() {  
    }  
    ...  
}
```

Output

Hello George Washington

application-context.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-4.1.xsd">  
  
    <bean id="president"  
        class="edu.mum.cs544.lecture.model.Person">  
        <property name="firstname" value="George"/>  
        <property name="lastname" value="Washington"/>  
    </bean>  
  
    <bean id="helloWorld"  
        class="edu.mum.cs544.lecture.greeting.HelloPerson">  
        <property name="person" ref="president"/>  
    </bean>  
</beans>
```

Application Context

- The Application Context loads Spring configuration files (resources) in a generic fashion way
- Provides:
 - Bean factory methods for accessing application components
 - Instantiates objects declared in the Spring configuration file
 - Wires objects together with dependency injection
 - Creates proxy objects when needed (ie. transactions, caching, Java configuration)
 - The ability to publish events to registered listeners
 - The ability to resolve messages, supporting internationalization
- Implemented by many classes. Most commonly used:

ClassPathXmlApplicationContext("my-context.xml")

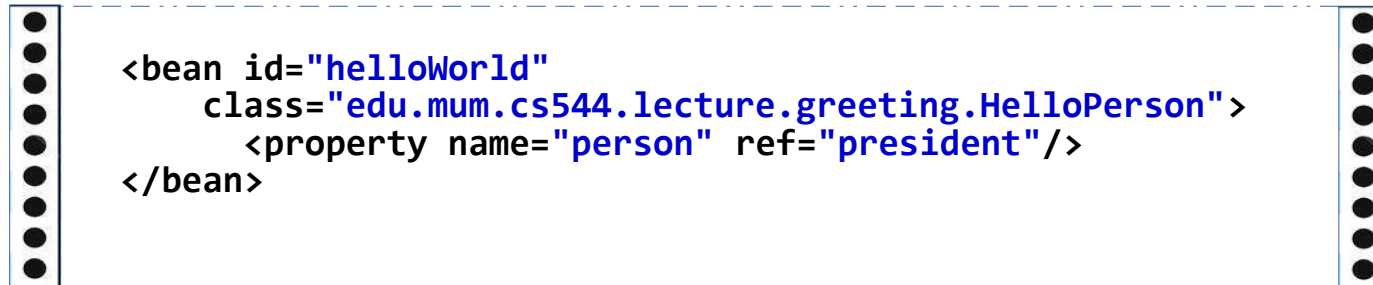
- Looks for the file on the class path

FileSystemXmlApplicationContext("C:/my-context.xml")

- Looks for the file on the file system

Beans

- A bean is an object that is instantiated, assembled, and managed by the Spring IoC container

A diagram showing an XML configuration element for a Spring bean. The element is enclosed in a dashed rectangular box. On the left and right sides of the box are vertical bars, each containing eight black dots. Inside the box, the XML code is as follows:

```
<bean id="helloWorld"  
      class="edu.mum.cs544.lecture.greeting.HelloPerson">  
    <property name="person" ref="president"/>  
</bean>
```

- Bean Name
 - Every bean has an identifier
 - A bean usually has only one identifier, but you may assign aliases
 - Identifiers must be unique within the container that hosts the bean
 - An identifier is not required, if not present the container will generate one
 - The naming convention is to use the standard Java convention for **instance field names**: start with lowercase letter and use camel-case
- Class
 - If you use XML-based configuration metadata use the **class** attribute of the **<bean>** element.
 - Tells the container which class to construct for the bean or which factory class to use to construct the object

Bean Scope

Scope	Description
SINGLETON	A single bean definition to a single object instance per container DEFAULT
PROTOTYPE	A single bean definition to any number of object instances NOT MANAGED
REQUEST	A single bean definition to the life cycle of a single HTTP request Each HTTP request has its own instance of a bean created Web-aware Spring Application Context
SESSION	A single bean definition to the life cycle of an HTTP Session Web-aware Spring Application Context
GLOBAL SESSION	A single bean definition to the life cycle of a global HTTP Session Only valid when used in a portlet context – The variables will be shared between all portlet sessions. Web-aware Spring Application Context
APPLICATION	Scopes a single bean definition to the lifecycle of a ServletContext Web-aware Spring Application Context

Singleton Vs Prototype Bean

customer-context.xml

```
...  
<bean id="customerOne" class="edu.mum.Customer" />  
  
<bean id="customerTwo" class="edu.mum.Customer" scope="prototype" />  
...
```

```
public class Application{  
    public static void main(String[] args) {  
        ApplicationContext context =  
            new ClassPathXmlApplicationContext("customer-context.xml");  
  
        Customer customerA = context.getBean("customerOne", Customer.class);  
        Customer customerB = context.getBean("customerOne", Customer.class);  
        Customer customerC = context.getBean("customerTwo", Customer.class);  
        Customer customerD = context.getBean("customerTwo", Customer.class);  
        System.out.println("Customer A =" + customerA);  
        System.out.println("Customer B =" + customerB);  
        System.out.println("Customer C =" + customerC);  
        System.out.println("Customer D =" + customerD);  
    }  
}
```

Output

```
Customer A =edu.mum.Customer@1a62e3  
Customer B =edu.mum.Customer@1a62e3  
Customer C =edu.mum.Customer@29e357  
Customer D =edu.mum.Customer@e95a56
```


Lazy Initialized Beans

lazy-customer-context.xml

```
...  
<bean id="customerOne" class="edu.mum.Customer" />  
<bean id="customerTwo" class="edu.mum.Customer" lazy-init="true" />  
...
```

```
public class Application{  
    public static void main(String[] args) {  
        System.out.println("Before Context Loading");  
        ApplicationContext context =  
            new ClassPathXmlApplicationContext("lazy-c-  
        System.out.println("After Context Loading");  
        System.out.println("Before getting One");  
        Customer customerA = context.getBean("customerOne", Customer.class);  
        System.out.println("After getting One");  
        System.out.println("Before getting Two");  
        Customer customerB = context.getBean("customerTwo", Customer.class);  
        Customer customerC = context.getBean("customerTwo", Customer.class);  
        System.out.println("After getting Two");  
    }  
}
```

By default, ApplicationContext eagerly creates and configure all singleton beans as part of the initialization process

Customer.java

```
public class Customer {  
    public Customer() {  
        System.out.println(">Creating Customer");  
    }  
    ...  
}
```

Output

```
Before Context Loading  
>Creating Customer  
After Context Loading  
Before getting One  
After getting One  
Before getting Two  
>Creating Customer  
After getting Two
```

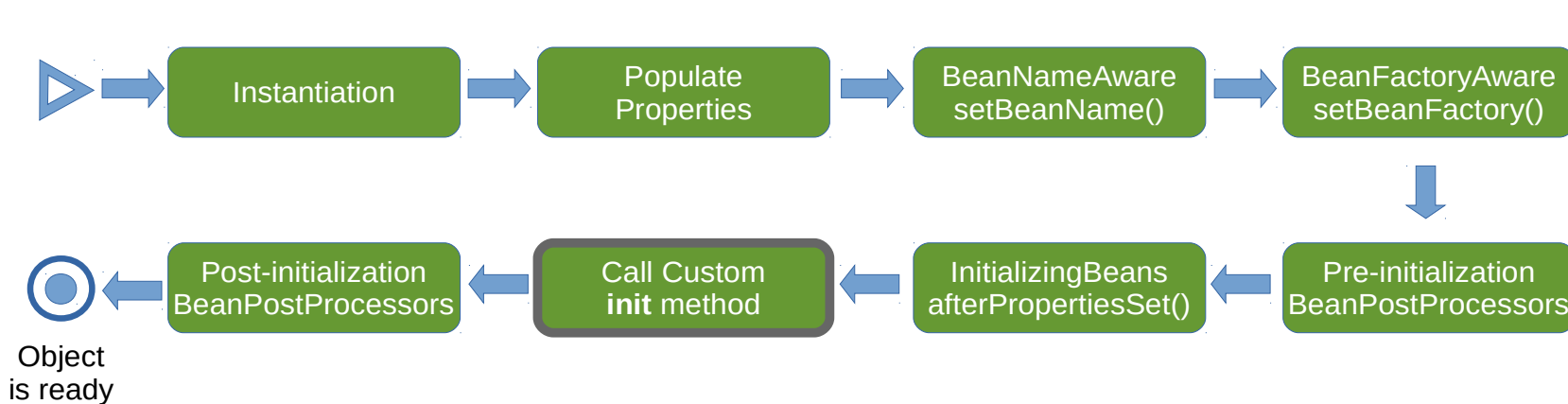
One

Two

Beans Life Cycle

- The beans life cycle allow us to add customizations through call back methods in two points:

Post initialization



Pre-destruction



- There are four ways to hook your custom code into the bean's life cycle:
 - custom **init-method** and **destroy-method** methods
 - @PostConstruct** and **@PreDestroy** annotations
 - InitializingBean** and **DisposableBean** callback interfaces
 - For other specific behavior you can use the **Aware** interfaces

init() / destroy()

```
public class Customer {
    @PostConstruct
    public void getReady() {
        System.out.println("I'm ready!");
    }
    @PreDestroy
    public void letsGo() {
        System.out.println("Bye!");
    }
    ...
}
```

or

customer-context.xml

```
<bean id="customerOne" class="edu.mum.Customer"
      init-method="getReady" destroy-method="letsGo" />
```

```
public class Customer {
    public void getReady() {
        System.out.println("I'm ready!");
    }
    public void letsGo() {
        System.out.println("Bye!");
    }
    ...
}
```

Default init() / destroy()

or

customer-context.xml

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

```
public class Customer {

    public void init() {
        System.out.println("I'm ready!");
    }
    public void destroy() {
        System.out.println("Bye!");
    }
    ...
}
```

Main Points

- 1) Through a configuration file, the Inversion of Control Container manages the lifecycle for the objects required by our application, allowing us to focus on the functionality of our logic and giving flexibility for future implementations.
Science of Consciousness: *by taking the holistic field of life, the pure nature of creative intelligence, we can enrich all aspects of life.*



Dependency Injection



Beauty is more important in computing than anywhere else in technology because software is so complicated.
Beauty is the ultimate defence against complexity.

– **David Gelernter**

Connecting Objects

Instantiate the Object Directly

```
public class AccountService {  
    private AccountDAO accountDAO;  
  
    public AccountService() {  
        accountDAO = new AccountDAO();  
    }  
  
    public void deposit(long accountNumber, double amount){  
        Account account =  
            accountDAO.loadAccount(accountNumber);  
        account.deposit(amount);  
        accountDAO.saveAccount(account);  
    }  
}
```

Decouple with an Interface

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public AccountService() {  
        accountDAO = new SecureAccountDAO();  
    }  
  
    public void deposit(long accountNumber, double amount){  
        Account account =  
            accountDAO.loadAccount(accountNumber);  
        account.deposit(amount);  
        accountDAO.saveAccount(account);  
    }  
}
```

Decouple with Object Factory

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public AccountService() {  
        DAOFactory daoFactory = new DAOFactory();  
        accountDAO = daoFactory.getAccountDAO();  
    }  
  
    public void deposit(long accountNumber, double amount){  
        Account account =  
            accountDAO.loadAccount(accountNumber);  
        account.deposit(amount);  
        accountDAO.saveAccount(account);  
    }  
}
```

Decouple with Dependency Injection

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public void setAccountDAO(IAccountDAO accountDAO) {  
        this.accountDAO = accountDAO;  
    }  
    public void deposit(long accountNumber, double amount){  
        Account account =  
            accountDAO.loadAccount(accountNumber);  
        account.deposit(amount);  
        accountDAO.saveAccount(account);  
    }  
}
```

```
<bean id="accountService" class="AccountService">  
    <property name="accountDAO" ref="theSecureDAO" />  
</bean>  
<bean id="theSecureDAO" class="SecureAccountDAO" />
```

What is Dependency Injection?

- On Dependency Injection (DI)
 - An object [“bean”] define their dependencies (the other objects they work with) only through:
 - constructor arguments (**Constructor-based DI**),
 - arguments to a factory method,
 - or properties (**Setter-based DI**)
 - The relationship (“wiring”) between the beans is set by a configuration, either using annotations on the code or an external file
 - Then the container uses the configuration to create the beans and it injects the dependencies to the bean during initialization
- Benefits of using DI
 - Decoupling is more effective, changing the wiring is easy
 - Code is cleaner
 - Classes become easier to test, in particular when the dependencies are on interfaces or abstract

When to use Dependency Injection?

- When an object references another object whose implementation might change
- You want to be able to switch / plug-in another implementation of the dependency
- When an object references a plumbing object
 - An object that sends an email
 - A DAO object
- When an object references a resource (non-code asset)



Spring Dependency Injection Types

Constructor Based DI

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public AccountService(IAccountDAO accountDAO){  
        this.accountDAO = accountDAO;  
    }  
}
```

```
<bean id="accountService" class="AccountService">  
    <constructor-arg ref="accountDAO" />  
</bean>  
<bean id="accountDAO" class="AccountDAO" />
```

Setter Based DI

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public void setAccountDAO(IAccountDAO accountDAO){  
        this.accountDAO = accountDAO;  
    }  
}
```

```
<bean id="accountService" class="AccountService">  
    <property name="accountDAO" ref="accountDAO" />  
</bean>  
<bean id="accountDAO" class="AccountDAO" />
```

- Setter Based DI Considerations

- If the injection fails you may end up with an object in an invalid state
- If you want to execute initialization code that uses the injected attributes, then you have to write it on an `init()` method. You cannot place this code in the constructor

Constructor DI Multiple Arguments

Arguments of Different Types

```
public class PaymentService {  
    private IVisaVerifier visaVerifier;  
    private IMastercardVerifier mastercardVerifier;  
  
    public PaymentService(IVisaVerifier visaVerifier,  
        IMastercardVerifier mastercardVerifier){  
        this.visaVerifier=visaVerifier;  
        his.mastercardVerifier=mastercardVerifier;  
    }  
}
```

Sets by Type

```
<bean id="paymentService" class="products.PaymentService">  
    <constructor-arg ref="mastercardVerifier" />  
    <constructor-arg ref="visaVerifier" />  
</bean>  
<bean id="visaVerifier" class="VisaVerifier"/>  
<bean id="mastercardVerifier" class="MastercardVerifier"/>
```

Arguments of the Same Type

```
public class PaymentService implements IPaymentService{  
    private ICreditCardVerifier visaVerifier;  
    private ICreditCardVerifier mastercardVerifier;  
  
    public PaymentService(ICreditCardVerifier visaVerifier,  
        ICreditCardVerifier mastercardVerifier){  
        this.visaVerifier=visaVerifier;  
        this.mastercardVerifier=mastercardVerifier;  
    }  
}
```

Sets by Position
Be careful with the order
if not using index

```
<bean id="paymentService" class="products.PaymentService">  
    <constructor-arg index="0" ref="visaVerifier" />  
    <constructor-arg index="1" ref="mastercardVerifier" />  
</bean>  
<bean id="visaVerifier" class="VisaVerifier"/>  
<bean id="mastercardVerifier" class="MastercardVerifier"/>
```

Active Learning

- When a bean is declared with scope = prototype does it load eagerly or lazily?
- When will Spring call the destroy method on your class?



Auto Wiring By Name

```
public class CustomerService {  
    private EmailService emailService;  
  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
}
```

```
<bean id="customerService" class="mypackage.CustomerService" autowire="byName"/>  
<bean id="emailService" class="mypackage.EmailService"/>
```

Auto-Wiring by Name using Annotations

```
public class CustomerService {  
    private EmailService emailService;  
  
    @Autowired  
    @Qualifier("myEmailService")  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
}
```

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

```
<bean id="customerService" class="mypackage.CustomerService"/>  
<bean id="myEmailService" class="mypackage.EmailService"/>
```

*<context:annotation-config/>
tells Spring to look for
annotations in the classes*

Auto Wiring By Type

```
public class CustomerService {  
    private EmailService emailService;  
  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
}
```

```
<bean id="customerService" class="mypackage.CustomerService" autowire="byType"/>  
<bean id="myEmailService" class="mypackage.EmailService"/>
```

Auto-Wiring by Type using Annotations

```
public class CustomerService {  
  
    @Autowired  
    private EmailService emailService;  
  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
}
```

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

```
<bean id="customerService" class="mypackage.CustomerService"/>  
<bean id="myEmailService" class="mypackage.EmailService"/>
```

Autowire by type has the restriction that only ONE bean of the given type can be defined in the configuration

Auto Wiring By Constructor

```
public class CustomerService {  
    private EmailService emailService;  
  
    CustomerService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
}
```

```
<bean id="customerService" class="mypackage.CustomerService" autowire="constructor"/>  
<bean id="myEmailService" class="mypackage.EmailService"/>
```

Auto-Wiring by Constructor using Annotations

```
public class CustomerService {  
    private EmailService emailService;  
  
    @Autowired  
    public CustomerService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
}
```

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

```
<bean id="customerService" class="mypackage.CustomerService"/>  
<bean id="myEmailService" class="mypackage.EmailService"/>
```

Injecting Values

```
<bean id="customerService" class="mypackage.CustomerService" >
  <property name="region" value="English" />
  <property name="defaultdiscount" value="4.5" />
</bean>
```

Auto-Wiring Values using Annotations

```
public class CustomerService implements
ICustomerService{
    @Value("#{systemProperties['user.language']" )
    private String region;
    @Value("4.5" )
    private double defaultdiscount;
    ...
}
```

- Spring configuration can also make use of the Spring Expression Language (SPEL)
- Looks a lot like JSF expression language

```
<bean id="customerService" class="mypackage.CustomerService" >
  <property name="region"
    value="#{ systemProperties['user.language'] }" />
  <property name="defaultdiscount"
    value="#{ discountCalculator.defaultdiscount }" />
</bean>
```

Spring Expression Language (SpEL)

- Supports querying and manipulating an object graph at runtime
- Similar to Unified EL, lets you access bean properties, method invocation and it has basic string templating functionality
- Support for defining bean definitions in XML or @ annotation-based configuration
- The variables **systemProperties** and **systemEnvironment** are predefined to access operative system and JVM environment variables
- As in Unified EL, you can use `${}` or `#{}` to evaluate an expression. In Spring, they will be evaluated in the context they are used - there is no deferred evaluation.

`#{Expression}`



Expression Language

- Similar to JavaScript expressions:
 - no typecasting
 - implicit type conversion
 - Null friendly
 - **object.property** means the same as **object['property']**, **object.get("property")**, **object.getProperty("property")** or **object.getProperty()**; thus liberates the web-content designers from having to know how the values must be accessed on the objects
- In Spring, to read a properties file, you will add an object of type **PropertyPlaceholderConfigurer** in your configuration to let upload the values and make them accessible through EL

`@Value("${database.url}")`

- A properties file is a text file with a **key=value** structure (one key/value per line)

```
database.url = jdbc:derby://server/myDB
database.user=sa
```

- For internationalization in Spring, you must include an object of type **ReloadableResourceBundleMessageSource** to load the values, which will also be accessible through EL

Injecting Lists and Sets

```
public class ShippingService {  
    public List<IShipper> shippers;  
    public Set<IMover> movers;  
  
    public ShippingService(List<IShipper> shippers, Set<IMover> movers) {  
        this.shippers = shippers;  
        this.movers = movers;  
    }  
}
```

```
<bean id="shippingService" class="mypackage.ShippingService" >  
    <constructor-arg>  
        <list>  
            <bean id="upsShipper" class="mypackage.UPS" />  
            <bean id="uspostalShipper" class="mypackage.USPostal" />  
        </list>  
    </constructor-arg>  
    <constructor-arg>  
        <set>  
            <bean id="bigM" class="mypackage.BigMover" />  
            <bean id="mediumM" class="mypackage.MediumMover" />  
            <bean id="smallM" class="mypackage.SmallMover" />  
        </set>  
    </constructor-arg>  
</bean>
```

Injecting Maps

```
public class ShippingService {  
    public Map<String,IShipper> shippers;  
  
    public ShippingService(Map<String,IShipper> shippers) {  
        this.shippers = shippers;  
    }  
}
```

```
<bean id="shippingService" class="shipping.ShippingService" >  
    <constructor-arg>  
        <map>  
            <entry key="one" value-ref="upsShipper"/>  
            <entry key="two" value-ref="uspostalShipper"/>  
            <entry key="three" value-ref="dhlShipper"/>  
        </map>  
    </constructor-arg>  
</bean>  
<bean id="upsShipper" class="shipping.UPS" />  
<bean id="uspostalShipper" class="shipping.USPostal" />  
<bean id="dhlShipper" class="shipping.DHL" />
```

Bean Definition Inheritance

- A bean definition can contain a lot of configuration information
- A child bean definition inherits configuration data from a parent definition
- The child definition can override some values, or add others, as needed

```
<bean id="mySuperClass" abstract="true"
      class="myPackage.SuperClass">
  <property name="lastname" value="Doe"/>
  <property name="age" value="1"/>
</bean>

<bean id="mySubClass"
      class="myPackage.SubClass"
      parent="mySuperClass" >
  <property name="lastname" value="James"/>
</bean>
```

Stereotype Annotations

Annotation	Description
@Component	Generic annotation for any Spring-managed component
@Controller	Indicates that the component is a Spring MVC controller
@Repository	The component is a data access repository. Also indicates that SQLExceptions thrown from the component's methods should be translated into Spring DataAccessExceptions
@Service	Indicates that the component is a service.

```
@Service ("customerService")
public class CustomerServiceImpl implements CustomerService{
    @Autowired
    private EmailService emailService;

    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }
    ...
}
```

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

    <context:component-scan base-package="edu.mum.example1"/>
    <context:annotation-config />
</beans>
```

<context:component-scan...
Indicates in which package
Spring has to search for
annotations

Spring and JSR Annotations

- Spring also supports injection using Java Standard Annotations defined by JSR-250 and JSR-330

Spring	javax.inject.*	Comments
@Autowired	@Inject @Resource	@Inject has no required attribute @Resource allows you to specify the name of the bean as an attribute
@Qualifier	@Named	Equivalent
@Scope("singleton")	@Singleton	The JSR-330 default scope is prototype. However, in order to keep it consistent with Spring's general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring's @Scope annotation.
@Component	@Named	Equivalent
@Controller	@Named	Equivalent
@Repository	@Named	Equivalent
@Service	@Named	Equivalent
@Value	-	There is no equivalent
@Required	-	There is no equivalent
@Lazy	-	There is no equivalent

XML or Annotations?

- XML Configuration
 - Pros
 - Configuration is separated from code
 - All the configuration can be in one file, or divided into files that group related beans
 - Tools can use the XML for graphical views
 - Cons
 - Large verbose XML file(s)
 - No compile time type safety
 - May become difficult to maintain
- Annotations
 - Pros
 - All information (configuration and logic) in one place: the Java code
 - More type safe
 - Cons
 - Configuration in the code requires recompiling to change



Active Learning

- What are the disadvantages of setter injection?
- Why is annotation configuration called 'classpath scanning'?



Main Points

2. Dependency injection allows us to support better separation of concerns and creates more malleable applications. **Science of Consciousness**: *a person growing in creative intelligence experiences purification of the nervous system, thus producing a better quality of consciousness.*



Summary

In this lesson we covered

- The ApplicationContext instantiates all the beans declared in the configuration
- Spring beans are eagerly instantiated singletons by default
- Spring allows you to call your init methods and destroy methods
- Spring beans are unaware of the fact that they are organized by Spring, regardless of what their configuration is
- Dependency injection is all about doing less (code), and accomplishing more (flexibility)
- Allowing Spring to inject more than just bean references opens up a completely new area of flexibility and configuration options

References:

<http://docs.spring.io/spring/docs/4.3.0.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#overview-core-container>

<http://docs.spring.io/spring/docs/4.3.0.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#beans-introduction>

<https://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html>

UNITY CHART

CONNECTING THE PARTS OF THE KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Separation of concerns has to be applied on all levels of the architecture

- 1) The Inversion of Control container facilitates the integration of the system through dependency injection.
 - 2) Abstraction of the implementation is achieved on the classes, allowing the implementation of the algorithms to be changed over time.
-
- 3) **Transcendental Consciousness:** is the common base of all knowledge.
 - 4) **Impulses in the Transcendental Field:** The first expression of knowledge is the non-expressed state of pure awareness, in the pure potentiality of knowledge, but flowing withing itself in its unmanifest nature.
 - 5) **Wholeness moving within itself:** Seeing the absolute value of the object on the surface of the object.

