**Spring Core**

**How to start Spring Project?**

1. Create new Java Project
2. Add JAR files to lib folder (create lib folder in src folder)
3. Right click-> Properties
4. Java Build Path -> classpath -> add jars -> select jars from lib folder -> apply and close

**ApplicationContext.xml in src directory**

**IOC -> Outsource to an object factory**

**The approach of outsourcing the construction and management of obje**

**Spring Container:**

Primary Function-

1. Create and manage objects (inversion of control)

2. Inject object dependency (Dependency Injection)

**Spring Development Process**

1. Configure your Spring Beans

2. Create a Spring Container

3. Retrieve Beans from Spring Container

**Dependency Injection:**

**The client delegates to calls to another object the responsibility of providing its dependencies.**

Dependency same thing as “helper objects”.

Injection Types:

1. Constructor Injection
2. Setter Injection

**Constructor Injection development process:**

1. Define the dependency interface and class
2. Create a constructor in your class for injections
3. Configure the dependency injection in Spring config file

**Setter injection development process:**

1. Create setter methods in your class for injections
2. Configure the dependency injection in Spring config file

Property Injection

<property name=”StudentID” value=”Neha234e” />

**Bean Scopes:**

Scope refers to the lifecycle of a bean

How long does the bean live?

How many instances are created

How is the bean shared?

1. **Singleton**

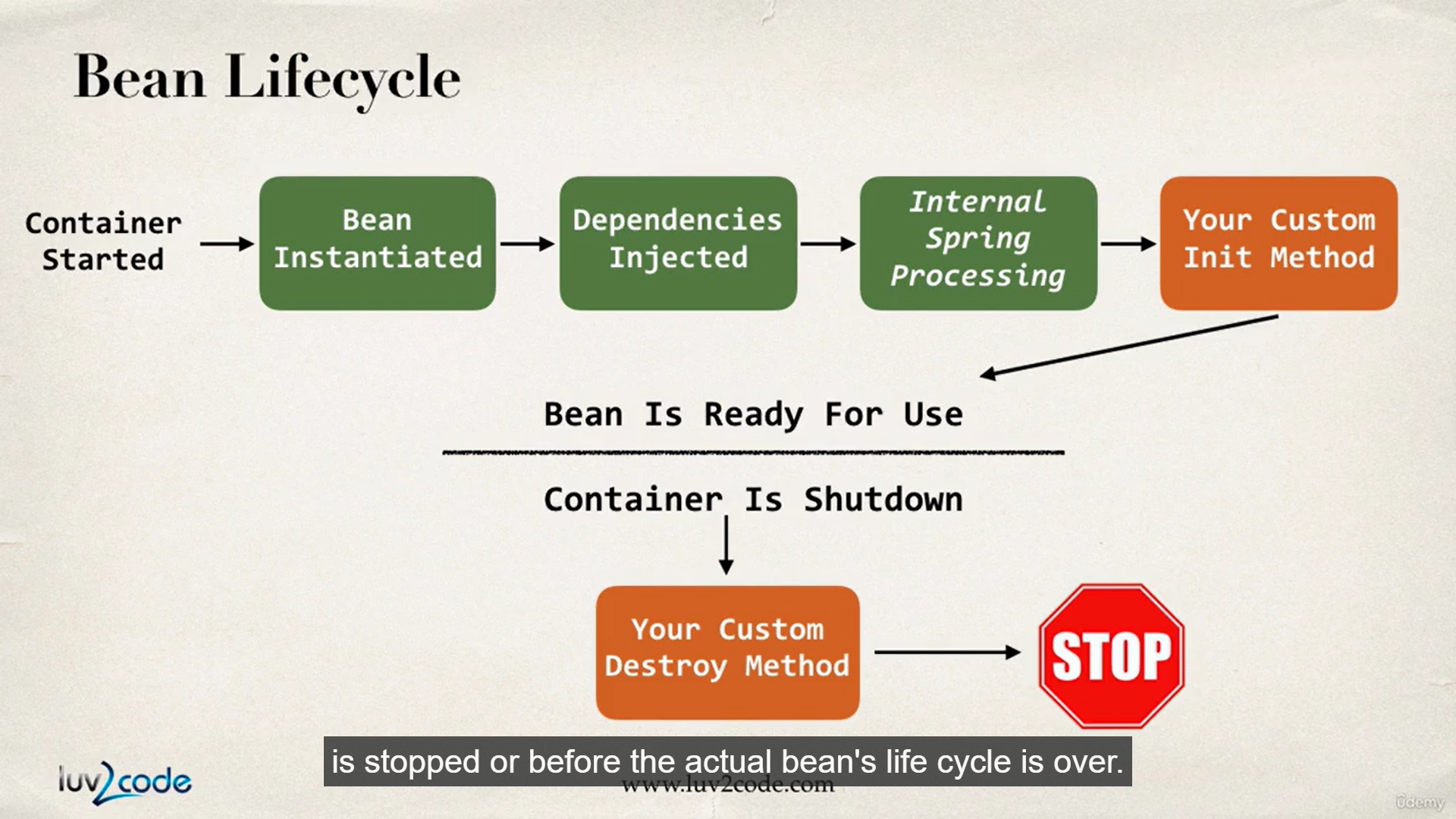
Create a single shared instance of bean. Default scope

1. **Prototype**

Creates a new bean instance for each container request

1. **Request**
2. **Session**
3. **Global-session**

**Bean Life Cycle:**



Spring allows to add and call custom code during bean initialization and destruction. These are called hooks.

Hook in code during initialization and destruction.

Init-method = “fooBar”

Destroy-method=”fooDes”

1. Define your methods for init and destroy
2. Configure the method names in Spring config file

**Special Note about init and destroy Method Signatures**

1. When using XML configuration, I want to provide additional details regarding the method signatures of the init-method  and destroy-method .
2. **Access modifier**  
   The method can have any access modifier (public, protected, private)
3. **Return type**  
   The method can have any return type. However, "void' is most commonly used. If you give a return type just note that you will not be able to capture the return value. As a result, "void" is commonly used.
4. **Method name**  
   The method can have any method name.
5. **Arguments**  
   The method can not accept any arguments. The method should be no-arg.

**Java Annotations**

Special labels/ markers added to Java classes

Provide meta-data about the class

Processed at compile time or run-time for special processing

**Why Spring Configuration with annotations?**

Xml configuration can be verbose

Configure your Spring beans with Annotations

Annotations minimizes the XML configuration

1. Enable component scanning in Spring config file
2. Add the @Component Annotation to your Java classes
3. Retrieve bean from Spring container

<context:component-scan base-packgage=””>

**Default Bean Id**

The class name, make first letter lower-case

@Component

**Spring AutoWiring?**

For dependency injection, Spring can use auto wiring

Spring will look for a class that matches the property

* Matches by type: class or interface

Spring will inject it automatically…hence it is autowired

**Autowiring Injection Types**

1. Constructor Injection
2. Setter Injection
3. Field Injection

**Constructor Injection**

1. Define the dependency interface and class
2. Create a constructor in your class for injections
3. Configure the dependency injection with @Autowired Annotation

**Setter Injection**

1. Create setter methods in your class for injections
2. Configure the dependency injection with @Autowired Annotation

**Method Injection**

Inject dependencies by calling any method on your class, simply give @Autowired

**Field Injection**

Inject dependencies by setting field values on your class directly (even private fields)

Accomplished by using Java Reflection

1. Configure the dependency injection with Autowired Annotation

Applied directly to the field

No need for setter methods

**@Qualifier**

If there are multiple FortuneService implementations…then which one to pick when we use autowired. So to specify which bean to use, qualifier is used.

1. Constructor Injection
2. Setter Injection
3. Field Injection

**@Qualifier with Field**

**@Autowired**

**@ Qualifier(“happyFortuneService”)**

**private FortuneService fortuneService;**

**@Qualifier with Constructors**

**@Autowired  
    public TennisCoach(@Qualifier("randomFortuneService") FortuneService theFortuneService) {**

**System.out.println(">> TennisCoach: inside constructor using @autowired and @qualifier");  
          
        fortuneService = theFortuneService;  
    }**

**@Qualifier with Setter Injection**

@Autowired

public void setFortuneService(@Qualifier("randomFortuneService") FortuneService theFortuneService) {

System.out.println(">> TennisCoach: inside setFortuneService() method");

this.fortuneService = theFortuneService;

}

**Scope Annotations**

1. @Scope(“singleton”)
2. @Scope(“prototype”)

**Lifecycle methods**

* @PostConstruct
* @PreDestroy

**Three ways of configuring Spring Container**

1. Full XML Config
2. XML Component Scan
3. Java Configuration Class

**No XML Configuration**

**Spring Configuration in java class**

1. Create a Java class and annotate as @Configuration
2. Add component scanning support: @ComponentScan (Optional)
3. Read Spring Java Configuration class
4. Retrieve bean from Spring container

**Defining Spring Beans with Java code**

1. Define method to expose bean
2. Inject bean dependencies
3. Read Spring Java configuration class
4. Retrieve bean from Spring container

**@Bean**

//define bean for our sad fortune service

@Bean

**public** FortuneService sadFortuneService() {

**return** **new** SadFortuneService();

}

//define bean for our swim coach AND inject dependency

@Bean

**public** Coach swimCoach() {

**return** **new** SwimCoach(sadFortuneService());

}

**Injecting values from properties file - @PropertySource(“classpath:sport.properties”)**

1. Create Properties File
2. Load Properties File in Spring Config
3. Reference values from Properties File

**Spring MVC**

Framework for building web applications

Based on MVC design pattern

Leverages features of the Core Spring Framework

Benefits

1. Spring way of building web app UIs in Java
2. Leverage a set of reusable UI components
3. Help manage application state for web requests
4. Process form data: validation, conversion etc
5. Flexible configuration for the view layer

Components of a Spring MVC application

1. A set of web pages to layout UI components
2. A collection of Spring beans (controllers, services)
3. Spring configuration (XML, Annotations or Java)

MVC Configuration

Add configurations to file: WEB-INF/web.xml

1. Configure Spring MVC Dispatcher Servlet
2. Set up URL mappings to Spring MVC Dispatcher Servlet

Add configurations to file: WEB-INF/spring-mvc-demp-servlet.xml

1. Add support for Spring component scanning
2. Add support for conversion, formatting and validation
3. Configure Spring MVC View Resolver

@Controller

@RequestMapping

@RequestParam

**Spring MVC Framework**follows the Model-View-Controller design pattern. It is used to develop web applications. It works around DispatcherServlet. DispatcherServlet handles all the HTTP requests and responses. It dispatches the requests to handlers. It uses @Controller and @RequestMapping as default request handlers. The @Controller annotation defines that a particular class is a controller. @RequestMapping annotation maps web requests to Spring Controller methods. The terms model, view, and controller are as follows:

* **Model**: The Model encapsulates the application data.
* **View**: View renders the model data and also generates HTML output that the client’s browser can interpret.
* **Controller**: The Controller processes the user requests and passes them to the view for rendering.

Spring MVC Framework works as follows:

1. All the incoming requests are intercepted by the DispatcherServlet that works as the front controller.
2. The DispatcherServlet then gets an entry of handler mapping from the XML file and forwards the request to the controller.
3. The object of ModelAndView is returned by the controller.
4. The DispatcherServlet checks the entry of the view resolver in the XML file and invokes the appropriate view component.

**@ModelAttritbute**

In Spring MVC, the @ModelAttribute annotation binds a method parameter or method return value to a named model attribute and then exposes it to a web view. It refers to the property of the Model object.

For example, if we have a form with a form backing object that is called “Student” then we can have Spring MVC supply this object to a Controller method by using the @ModelAttribute annotation:

@RequestMapping("/home")

public String showHomePage(@ModelAttribute("studentInfo") StudentInfoDTO studentInfoDTO) {

return "something";

}

**Spring MVC Form Validation:**

Java’s Standard Bean Validation API

* Java has a standard Bean Validation API
* Defines a metadata model and API for entity validation
* Not tied to either the web tier or the persistence tier
* Available for server-side apps and also client-side JavaFX/ Swing apps

**Validation Annotations**

@NotNull

@Min

@Max

@Size

@Pattern

@Future/ @Past

Others…

@Valid: Perform validation rules on object

@BingingResult: results of validation placed int the Binding Result.

**Special Note about BindingResult Parameter Order**

When performing Spring MVC validation, the location of the BindingResult parameter is very important. In the method signature, **the BindingResult parameter must appear immediately after the model attribute**.

If you place it in any other location, Spring MVC validation will not work as desired. In fact, your validation rules will be ignored.

1. @RequestMapping("/processForm")
2. public String processForm(
3. @Valid @ModelAttribute("customer") Customer theCustomer,
4. BindingResult theBindingResult) {
5. ...
6. }

**@InitBinder**

* @InitBinder annotation works as a pre-processor
* It will pre-process each web request to our controller
* Method annotated with @InitBinder is executed

Hibernate

Hibernate handles all of the low-level SQL

Minimizes the amount of JDBC code you have to develop

Hibernate provides the Object-to-Relational Mapping (ORM)

**How to start hibernate project?**

https://www.udemy.com/course/spring-hibernate-tutorial/learn/lecture/6424824#overview