### Spring Udemy Notes

**why spring:**

popular framework

simper and lightweight alternative to J2EE

large number of helper classes. make things easier

get rid of EJB, former bad reputation.

**New for Spring 5:**

java 8 or higher

Upgrade Spring MVC to use new versions of Servlet API 4.0

Add new reactive programming framework: spring webFlux

**Core Container:** Beans:Core:SpEL:Context:

**Infrastructure:** AOP: Aspect Oriented Programming

**Data Access Layer:** JDBC ORM Transactions OXM JMS:Java message service

**Web layer**: Servlet, websocket, web ,porlet

**Test layer:** unit, integration, mock

**Inversion of Control(IoC)**

The approach of outsourcing the construction and management of objects.

Software Engineering Best Practice: code to an interface

**Spring Container:**

Primary functions:

1. Create and manager objects(Inversion of Control)

2. Inject object's dependencies(Dependency Injection)

**Configuring Spring Container:**

1. XML configuration file(legacy)

2. Java Annotations(modern)

3. Java Source Code(modern)

**Spring Development Process**

1. Configure ur spring Beans: treat the beans as java objects

2. Create a Spring Container: generically known as ApplicationContext

3. Retrieve Beans from Spring Container.

**Dependency Injection:**

Injection types two common: 1. Constructor Injection. 2. Setter Injection !!// treat dependency as a helper

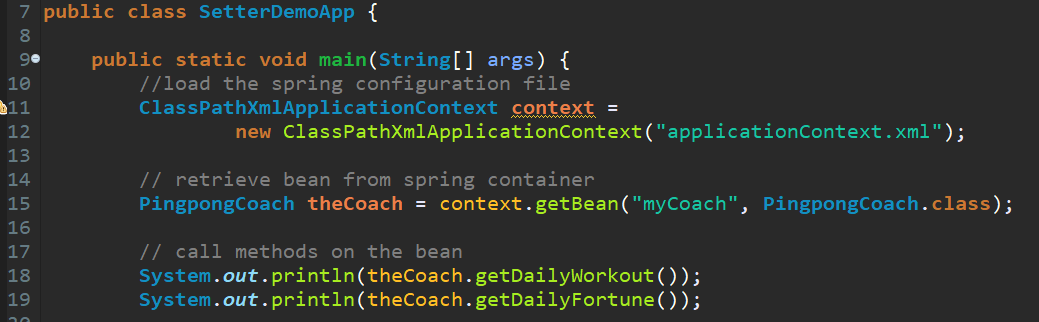
**Setter Injection:**

1. Create setter method(s) in class for injections. 2. Configure the dependency injection in Srping config file

**Injecting Literal Values:**

1. Create setter methods in class for injections.: create private fields, create setter methods.

2. Configure the injection in Spring config file.



**Singleton:**

1.Spring Container creates only one instance of the bean, by default.

2.It's cached in memory.

3.All requests for bean, will return a SHARED reference to the SAME bean.

**Bean Prototype Scope: @Scope(“prototype”)**

new bean object for each request: different address different object

IF not mentioned,would be considered by default: singleton scope: only one instance, different requests only share the bean.

**Default scope is singleton: @Scope(“singleton”)**

Spring Container creates only one instance of the bean, by default.

It is cached in memory.

All requests for the bean: will return a SHARED reference to the SAME bean.

**Bean lifecycle:**

container started--> Bean Instantiated--> Dependencies Injected --> Internal Spring Processing --> your custom Init Method --> Bean is ready for use --

-- container is shutdown --> your custom destroy method --> stop

**Bean lifecycle methods/ hooks:**

1. you can add custom code during bean initializaion, calling custom business logic methods, setting up handles to resources(db, sockets, file etc).

2. you can add custom code during bean destruction, calling custom business logic method, clean up handles to resources(db, sockets, files ect).

**Init:@PostConstruct**

method configuration:set up bean initialization(init-method="any method name").

Code will execute **after** constructor and **after** injection of dependencies.

**Destroy:@PreDestroy**

method configuration: set up bean destroy method(destroy-method="any method name").

Code will execute before bean is destroyed.

**Note: prototype scope Spring does not call the @PreDestroy method, does not manage the complete lifecycle of prototype bean.**

**Special note about** @PostConstruct and @PreDestroy method signatures

Access modifier: any type, public, protected, private

Return type: any type, void is common used.

Method name: any method name.

Arguments: can not accept any arguments.

**Development process:**

1. Define your methods for init and destroy.

2. Configure the method names on Spring config file

Or 2. Add annotations: @PostConstruct and @PreDestroy

**Java Annotations:**

1. Special labels/markers added to Java classes.
2. Provide meta-data about the class.
3. Processed at compile time or run-time for special processing.

**Why Spring configuration with annotations:**

1. XML configuration can be verbose.
2. Configure Spring beans with annotations.
3. Annotations minimizes the XML configuration.

**Scan for component classes:**

1. Spring will scan Java classes for special annotations.
2. Automatically register beans in Spring container.

**Development Process:**

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

**Default Bean IDs**: the class name, make first letter lower-case

**Spring Dependency Injection with Annotations and Autowiring:**

**Spring Auto-wiring:**

1. For dependency injection, Spring can use auto wiring.
2. Spring will look for a class that matches the property: matches by type: class or interface
3. Spring will inject it automatically ... hence it is autowired

**Autowiring example:**

1. Injecting FortuneService into a Coach implementation.
2. Spring will scan @Components.
3. Any one implements FortuneService interface.
4. If so, let’s inject them. FOr example: HappyFortuneService.

**Autowiring Injection Types:**

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

**Development Process - Constructor Injection**

1. Define the dependency interface and class.
2. Create a constructor in class for injections. :constructor always have the same name as class.
3. Configure the dependency injection with @Autowired Annotation

**Setter Injection:** Inject dependencies by calling setter methods on class.

**Development Process - Setter Injection:**

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

**Field Injection:** Inject dependencies by setting field values on class directly(even private fields)

Accomplished by using Java Reflection

**Development Process - Field Injection:**

1. Configure the dependency injection with Autowired Annotation: applied directly to the field, no need for setter methods.

**Annotation Autowiring and Qualifiers:**

**Can apply @Qualifier annotation to:** 1. Constructor injection. 2. Setter injection methods. 3. Field injection

**Review *3* ways of Configuring Spring Container**:

1. Full XML Config.
2. XML Component Scan: <context: component-scan>
3. Java Configuration Class

**Java Source Code Configuration (No XML!) :**

**Development Process:**

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

**Define Spring Beans with Java Code:**

**Development Process:**

1. Define method to expose bean: @Bean and no component scan
2. Inject bean dependencies: still use @Bean pass in reference
3. Read Spring Java configuration class
4. Retrieve bean from Spring container: context.getBean(beanid, xx.class)

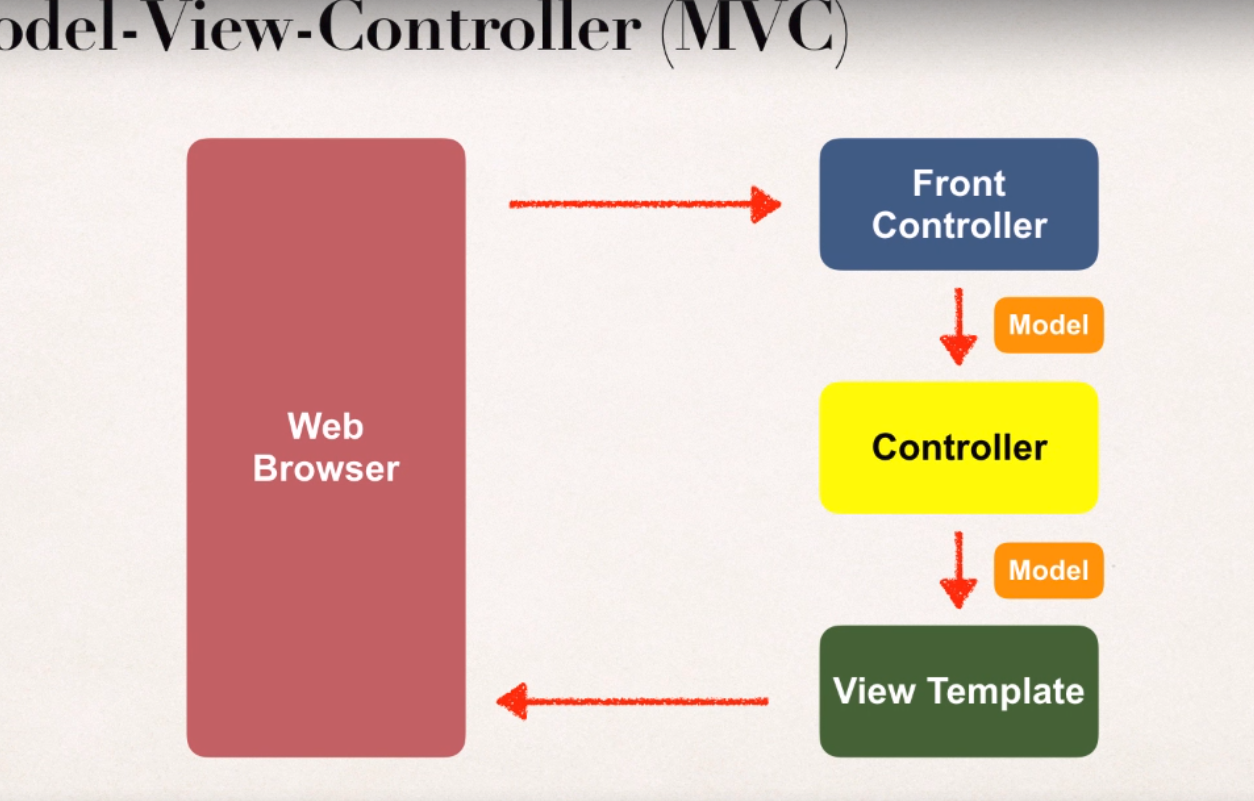
**Injecting Values from Properties file**

**Development Process:**

1. Create Properties File
2. Load Properties file in Spring config: @PropertySource(“classpath:sport.properties”)
3. Reference values from Properties File: @Value(“${}”)

**Spring MVC:**

1. Framework for building web application in Java.
2. Based on Model-View-Controller design pattern
3. Leverages features of the Core Spring Framework (IoC, DI)



**Spring MVC Benefits:**

1. The 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 from 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,etc..)
3. Spring configuration(XML, Annotations or Java code.)

**Controller:**

1. Code created by developer
2. Contains business logic
3. Handle request
4. Store/retrieve data
5. Place data in model
6. Send to appropriate view template

**Model:**

1. Contains your data
2. Store/retrieve data via backend systems.
3. Database, web service, etc
4. Use a Spring bean if like.
5. Place your data in the model: data can be any Java object/collection

**View Template:**

1. Spring MVC is flexible: Supports many view templates.
2. Most common is JSP + JSTL, other view templates: Thymeleaf, Groovy, Velocity, Freemarker, etc.
3. Developer creates a page: Displays data

When app provides a “view” name, Spring MVC will:

Prepend the prefix

Append the suffix

**Spring MVC Configuration Process**

1. Add configurations to file: WEB-INF/web.xml:
2. Configure Spring MVC Dispatcher Servlet
3. Set up URL mappings to Spring MVC Dispatcher Servlet
4. Add configurations to file: WEB-INF/spring-mvc-demo-servlet.xml
5. Add support for Spring component scanning
6. Add support for conversion, formatting and validation
7. Configure Spring MVC View Resolver

**Simple page create:**

**Development Process:**

1. Create Controller class
2. Annotate class with @Controller
3. Controller inherits from @Component ... supports scanning
4. Define Controller method
5. Add request mapping to controller method: @RequestMapping
6. Return view name
7. Develop view page

**Reading HTML form data:**

**Development process:**

1. Create controller class
2. Show HTML form
   1. Create controller method to show HTML form
   2. Create view page for HTML form
3. Process HTML Form:
   1. Create controller method to process HTML form
   2. Develop view page for confirmation

**Spring Model: is used to pass data between controllers and views**

The Model is a container for application data.

In Controller:

1. Put anything in the model.
2. Strings, objects, info from database,etc...

View page(JSP) can access data from model.

**Reading HTML Form data with @RequestParam Annotation:**

1. **Create a new method to process form data.**
2. **Read the form data: student’s name passed in by annotation**
3. **Convert the name to Upper case**
4. **Add uppercase version to the model**

**@RequestParam: Spring will read param from request: studentName and bind it to the variable**

**Adding request mappings to controller**:

1. Serves as parent mapping for controller
2. All request mappings on methods in the controller are relative

RequestMapping conflict: create a @RequestMapping(“/xxxx”) parent mapping

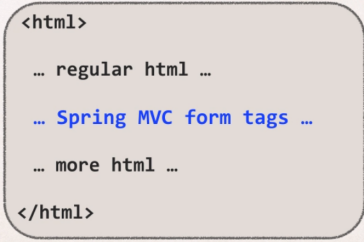
**Spring MVC Form tags:**

1. Are building block for a web page
2. Form tags are configurable and reusable for a web page

**Data Binding:**

1. Spring MVC Form tags can make use of data binding.
2. Automatically setting/ retrieving data from a Java object/bean

Web page structure: JSP page with special Spring MVC Form tags



**Showing Form: In Spring controller:**

1. Before show form, must add a model attribute
2. This is a bean that will hold form data for the data binding.

**Development Process:**

1. Create student class
2. Create student controller class
3. Create HTML form
4. Create form processing code
5. Create confirmation page

Label is for shown on the page, value is for passing

**Drop down Lists:**

**Input from form and pass in to the controller, finally show up in the page**.

Development process:

1. Update HTML form
2. Update student class - add getter/setter for new property
3. Update confirmation page

**Check box: <form:checkbox> multiple options, need array of Strings, add appropriate get/set methods**

**Loop in JSP file:** <%@ **taglib** uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

Use this in JSP and get loop from model object: <c:forEach var=”temp” items=”${student.operatingSystems}” >

**Validation:** check user input form for required fields, valid numbers in a range, valid format

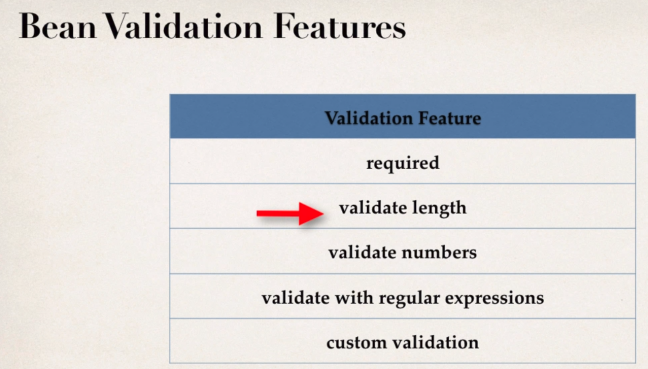
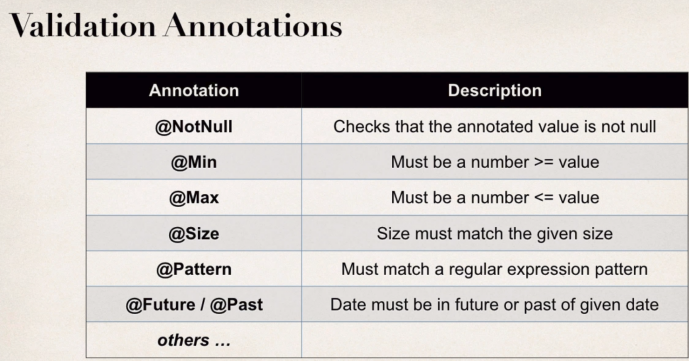
Java’s Standard Bean validation API: **constrain once, validate everywhere**

1. Java has a standard Bean Validation API (JSR-303/309).
2. Defines a metadata model and API for entity validation
3. Not tied to either the web tier or the persistence tier
4. Available for server-side apps and also client-side JavaFX/Swing apps

**Spring and Validation:**

1. Spring version 4 and higher supports Bean Validation API
2. Preferred method for validation when building Spring apps
3. Simply add Validation JARS to project

**Bean Validation Features:**



**Road map:**

1. Set up our development environment
2. Required field
3. Validate number range: min, max
4. Validate using regular expression(regexp)
5. Custom validation

**Hibernate: comes in to rescue and implement help validation**

1. Started as an ORM project
2. Recent years, expanded into other areas
3. Have full compliant JSR-303/309 implementation: not tied to ORM or database work.. seperate project.

**Development process:**

1. Download Validation JAR files from Hibernate
2. Add JAR files to project

**Checking for required field:**

1. Pass in value, show in the confirmation page if validated. Or return warming.

**Development process:**

1. Add validation rule to customer class
2. Display error messages on HTML form
3. Perform validation in the controller class
4. Update confirmation page.

**White space issues:**

Validator cannot distinguish it.

**@InitBinder:advanced annotation , not many ppl knw this!!!**

works as a pre-processor

It will pre-process each web request to our controller

Method annotated with @InitBinder is executed

**Pre-process every String form data**

Remove leading and trailing white space

If String only has white space.. trim it to null

**StringTrimmerEditor is a class defined in Spring API**

Xxxx = new StringTrimmerEditor(ture); true - means trim to null if there are all white space

**Validate a number range:**

Add a new input field on our form for: free passes

User can only enter a ranger: 0 to 10

**Development process:**

1. Add validation rule to customer class
2. Display error messages on HTML form
3. Perform validation in the controller class
4. Update confirmation page

**Using Regular Expression to validate Post Code:**

**Development process:**

1. Add validation rule to Customer class
2. Display error messages on HTML form
3. Update confirmation page

**Deal String input for Integer Fields - custom message:**

**Development process:**

1. Create custom error message: src/resources/messages.properties
2. Load custom messages resource in Spring config file: WebContent/WEB\_INF/spring-mvc-demo-servlet.xml

**Custom Validation rules by your own: new annotations**

1. Perform custom validation based on ur business rules
2. Spring MVC calls custom validation
3. Custom validation returns boolean value for pass/fail (true/false)
4. For custom validation, create a Custom Java Annotations

**Development process:**

1. Create custom validation rule
2. Create @CourseCode annotation
3. Create CourseCodeConstraintValidator
4. Add validation rule to Customer class
5. Display error messages on HTML form
6. Update confirmation page

Some details:

@Constraint(validatedBy = CourseCodeConstriantValidator.class): helper class that contains business rules/ validation logic

@Target( {ElementType.METHOD, ElementType.FIELD} ): where can we apply, on methods and fields

@Rentention(RetentionPolicy.RUNTIME) means: retain this annotation in the JAVA class file, and process it at runtime.

**Hibernate:** a framework for persisting/ saving data Java objects in a database

**Benefits of Hibernate:**

1. Hibernate handles all of the low-level SQL
2. Minimizes the amount of JDBC code you have to develop
3. Hibernate provides the Object-to-Relational Mapping(ORM): make it easy to create app to store and retrieve.

**Hibernate CRUD Apps**: create, read, update, delete objects.

Hibernate and JDBC:

Hibernate usese JDBC in the background for all database communications.

**Set up Hibernate in Eclipse:**

1. Create Eclipse project
2. Download Hibernate Files
3. Download MySQL JDBC Driver
4. Add JAR files to Eclipse Project... Build Path: and there are referenced libraries showing up upon lib

**Hibernate Dev process:**

1. Add Hibernate Configuration file
2. Annotate Java class
3. Develop Java Code to perform database operations.

**Two options for mapping:**

1. Option 1: XML config file (legacy)
2. Option 2: Java annotations (modern, preferred)

**Java Annotations:**

Step 1: Map class to database table

Step 2: Map fields to database columns

Hibernate two key players:

1. SessionFactory:
   1. Reads hibernate config file, get connect with database
   2. Creates session objects
   3. Heavy-weight object
   4. Only create once in ur app
   5. Re-use it over and over again
2. Session:
   1. Wraps a JDBC connection
   2. Main object used to save/retrieve objects
   3. Short-lived object
   4. Retrieved from sessionFactory

**Primary key:**

* Uniquely identifies each row in a table,
* must be a unique value,
* Cannot contain NULL values

ID Generation Strategies:

1. GenerationType.AUTO
2. GenerationType.IDENTITY : most common for MySQL and leverage auto-increment
3. GenerationType.SEQUENCE
4. GenerationType.TABLE

Bonus:

Can define ur own CUSTOM generation strategy

Create subclass of org.hibernate.id.SequenceGenerator

Override the method: public Serializable generate(...)

For hibernate: always remember to begin Transaction to commit and read

CRUD with hibernate: using session factory to create session.

Session a framework to manipulate object with database

Create object:

1. create session factory: session factory = new Configuration.configure(“hibernate.cfg.xml”)

.addAnnotatedClass(Student.class)

.buildSessionFactory();

1. Create session:Session.session = factory.getCurrentSession();
2. Create a student object: Student tempStudent = new Student(“”, “”, “”);
3. Start a transaction: session.beginTransaction();
4. **Save the object**: session.save(tempStudent);
5. Commit transaction: session.getTransaction().commit();

Read object:

1. Get a new session and start transaction: session = factory.getCurrentSession();
2. Session.beginTransaction();
3. Retrieve object:Student myStudent = session.get(Student.class, studentId);
4. Commit the transaction: session.getTransaction().commit();

Query object:

1. Start transaction: session.beginTransaction();
2. Query students: session.createQuery(“from Student”).getResultList();
3. Display the students: displayStudents(theStudents);
4. Commit the transaction: session.getTransaction().commit();

Update object method1:

1. Retrieve object:Student myStudent = session.get(Student.class, studentId);
2. Update the student: myStudent.setFirstName(“Scooby”)
3. Commit the transaction: session.getTransaction().commit();

Update object method 2:

1. Session = factory.getCurrentSession();
2. Session.beginTransaction();
3. Session.createQuery(“update student set xxxx”).executeUpdate();
4. Commit the transaction: session.getTransaction().commit();

Delete 1 method:

1. retrieve object: Student myStudent = session.get(Student.class, studentId);
2. Delete the student session.delete(myStudent);
3. Commit the transaction: session.getTransaction().commit();

Delete method 2:

1. Session.createQuery(“delete from Student where id=2”).executeUpdate();
2. Commit the transaction: session.getTransaction().commit();

Hibernate advanced mappings:

In databases, most likely will have to use with hibernate: multiple tables, and relationships between tables.

One-to-one, one-to-many, many-to-many mappings

Primary key and foreign key

Cascade: apply the same operation to related entities FOR EXAMPLE: cascade delete.

Fetch types: eager vs lazy loading

* + - 1. Eager will retrieve everything
      2. Lazy will retrieve on request

Uni-directional vs bi-directional

One-to-one & uni-directional：

Development process: one-to-one:

Pre work – define database tables

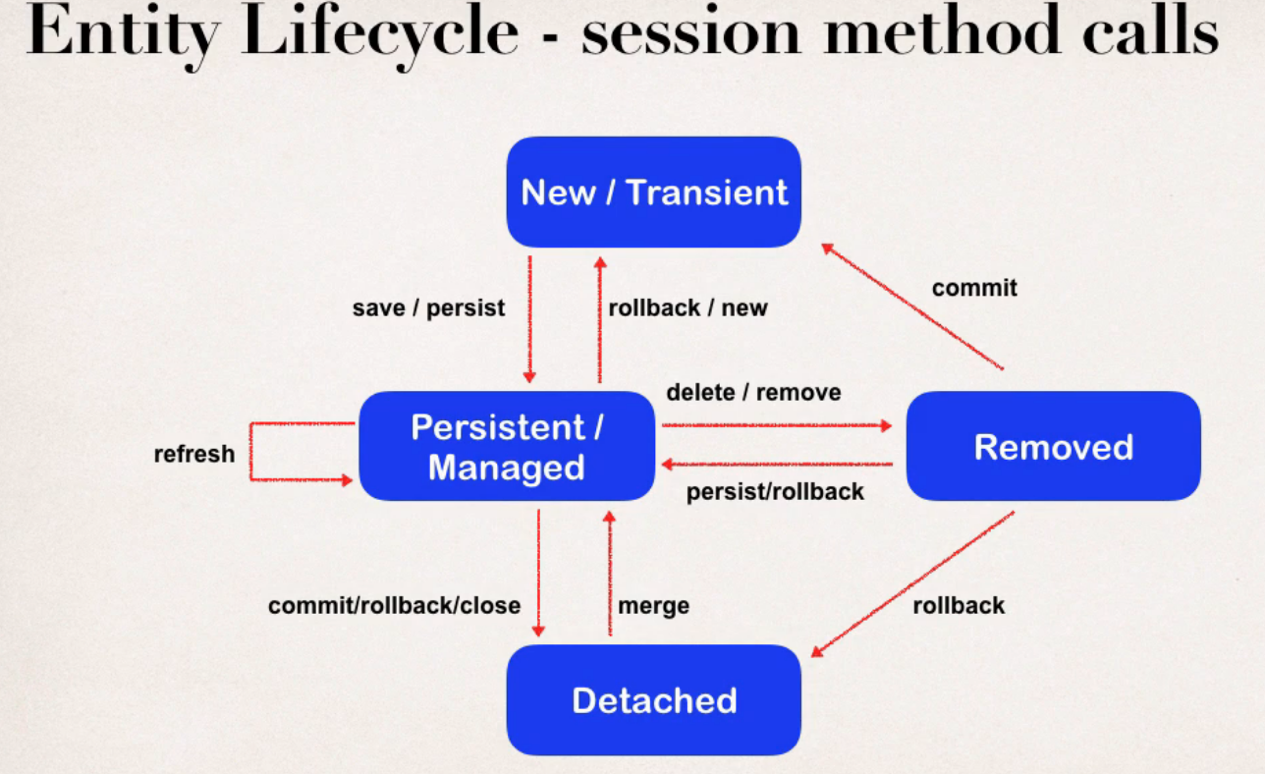
Create instructorDetails class

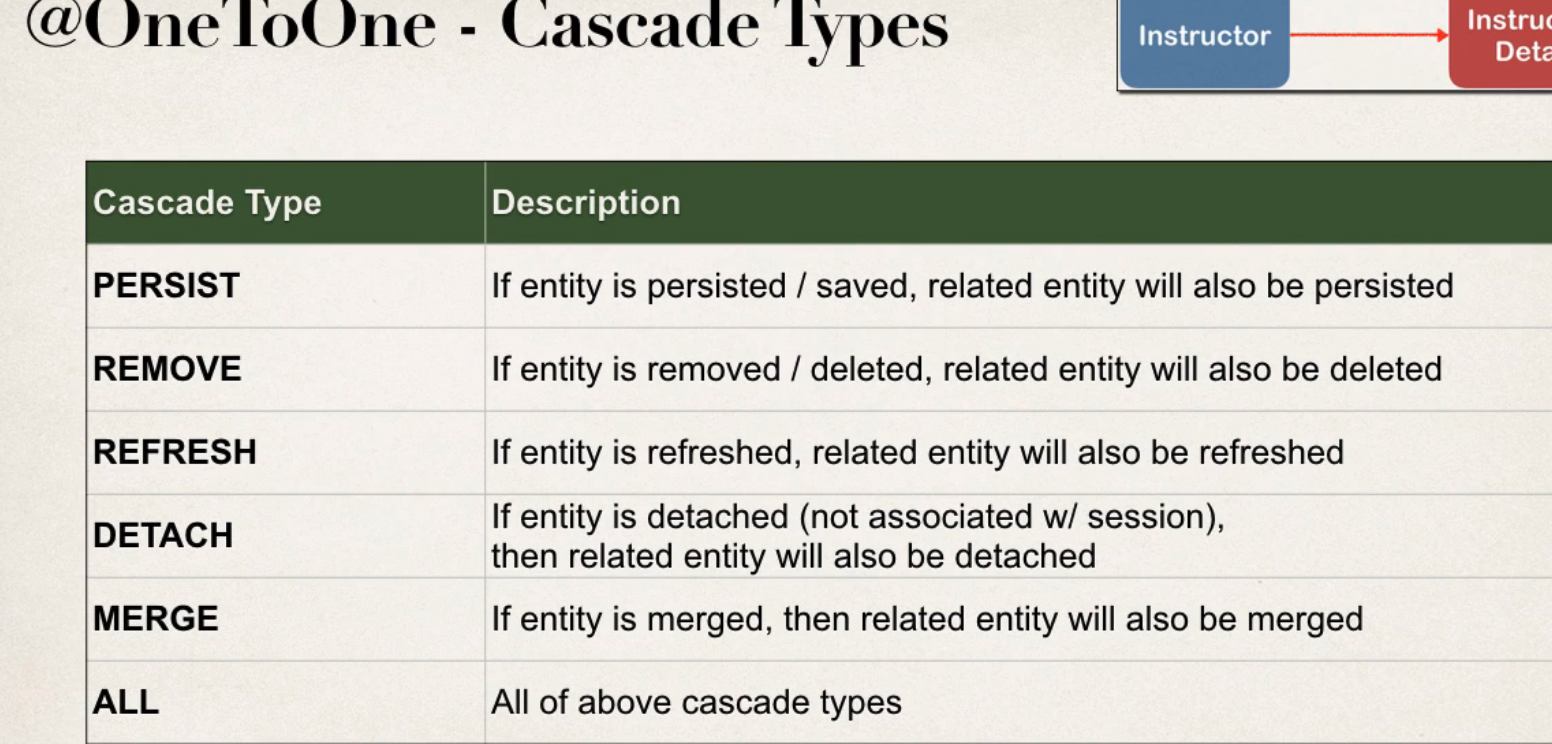
Create instructor class

Create main app

Entity lifecycle:

* + - 1. Detach
      2. Merge
      3. Persist
      4. Remove
      5. Refresh





set up “setter and getter” method in order to get those information in our app.

Set up “toString()” method in order to printout those information.

**One-to-one mapping**: the one we doen before is for uni-direction, you cannot manipulate from IntructorDetail object. SO bi-direction comes.

To use bi-direction we can keep the existing database shcema, no changes required to database, simply update the Java code.

Development process: one-to-one (Bi-Directional): mappedBy + CascadeType.ALL

Make updates to InstructorDetail class:

1. Add new field to reference Instructor
2. Add getter/setter methods for Instructor
3. Add @OneToOne annotation

Create the app

mappedBy: tells Hibernate: which class/object should I hook up with for this time

Look at the InstructorDetail property in the Instructor class

Use information from the Instructor class@JoinColum

NO Cascase delete(a real-world consideration): One more trial: remove the cascade relationship from Detail class to Instructor class: with doing so: Cascade.ALL to Cascade.DETACH …….just without Cascade.REMOVE

**One-to-Many mappings:** like an instructor has many courses.

Real-world project requirement or consideration:

1. If you delete an instructor, DO NOT delete the courses.
2. If you delete a course, DO NOT delete the instructor.
3. AKA which means: DO NOT APPLY cascade delete

Development process: one to many:

1. Prep work – define database tables
2. Create course class
3. Update instructor class
4. Create main app

**Fetch types: eager vs lazy loading**

Eager will retrieve everything: will load all dependent entities, load instructor and all of their courses at once. Would be a nightmare when with mega data.

Best practice: only load data when absolutely needed. Prefer lazy loading instead of eager loading.

Lazy will retrieve on request: lazy loading will load the main entity first and load dependent entities on demond.

Real-world use case:

In master view, use lazy loading for search results: only load instructors…not their courses.

In detail view, retrieve the entity and necessary dependent entities: load instructors and courses.

Default Fetch types: of course you can override it.

@OneToOne: FetchType.EAGER

@OneToMany: FetchType.LAZY

@ManyToOne: FetchType.EAGER

@ManyToMany: FetchType.LAZY

More about Lazy Loading:

When you lazy load, the data is only retrieved on demand.

However, this requires an open Hibernate session, need an connection to database to retrieve data.

If Hibernate session is closed, and you attempt to retrieve lazy data, Hibernate wil throw an exception.

To retrieve lazy data, you will need to open a Hibernate session.

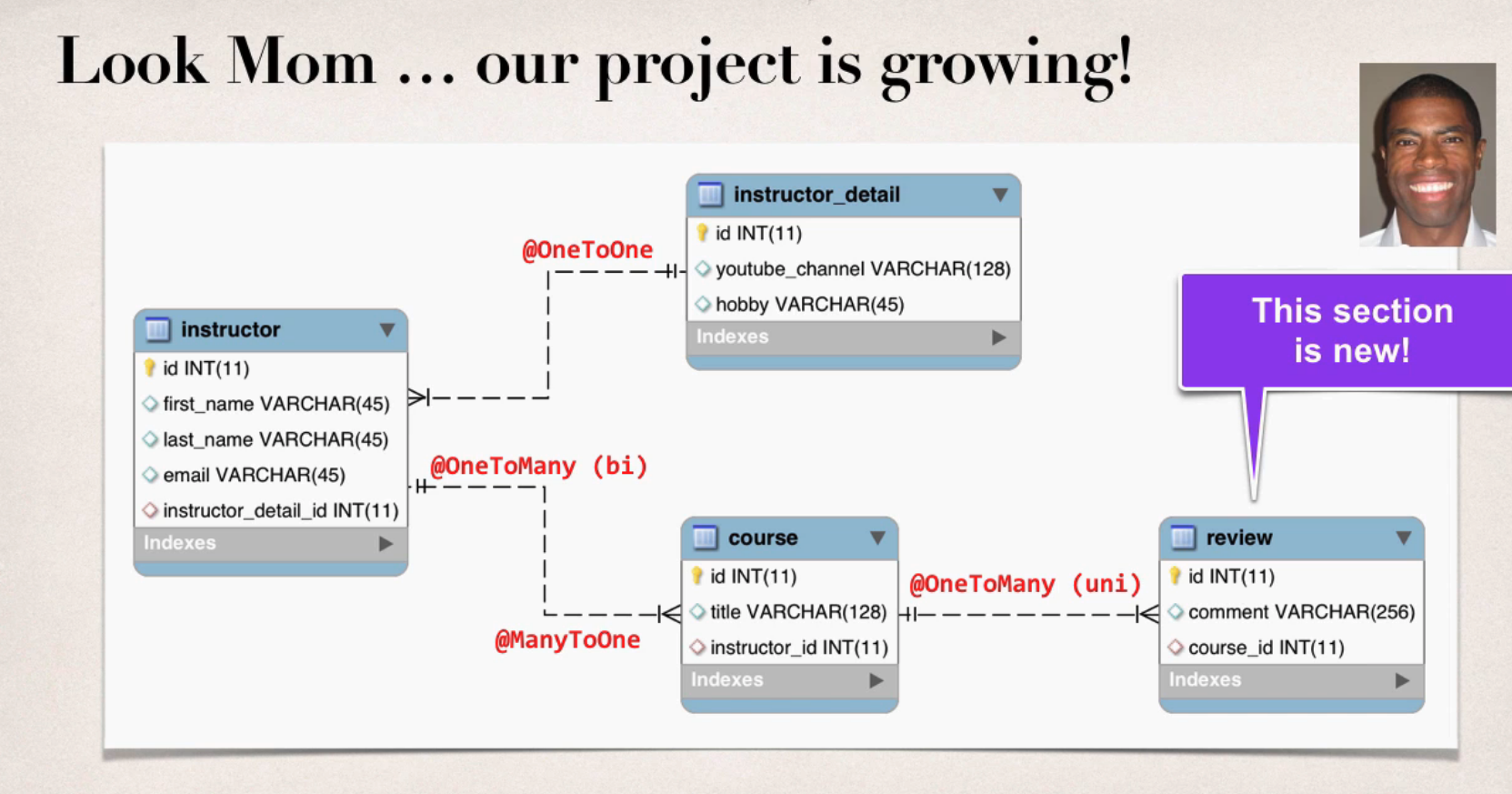
Retrieve lazy data using:

1. Option1: session.get and call appropriate getter method
2. Option2: Hibernate query with HQL : join fetch HQL
3. Many other techniques available but the two above are most common.

One-to-many mapping: like course has many reviews is uni-direction:

Real-world requirement:

* If delete a course, also delete the reviews.
* Reviews without a course… have no meaning.1



development process: one-to-many

1. Prep work –define database tables
2. Create review class
3. Update course class
4. Create main app

More:@JoinColumn:

In this scenario, @JoinColumn tells Hibernate:

* Look at the course\_id column in the review table
* Use this information to help find associated reviews for a course.

Many-to-many mapping: a course can have many students, a student can have many courses.

Keep track of relationships.

Join table: that provides a mapping between two tables. It has foreign keys for each table to define the mapping relationship.

Tells Hibernate:

* Look at the course\_id column in the course\_student table
* For other side(inverse), look at the student\_id column in the course\_student table
* Use this information to find relationship between course and students.

@ManyToMany: a join table coming out.

More on “inverse”:

* We are defining the relationship in the Course class
* The student class is on the “other side” … so it is considered the “inverse”
* “inverse” refers to the “other side” of the relationship

Real-world project requirement:

If you delete a course, DO NOT delete the students. DO NOT apply cascade delete.