**4. Inversion of control**

The approach of outsourcing the construction and management of objects.

Spring container. Primary functions:

* Create and manage objects (Inversion of control)
* Inject object’s dependencies (Dependency Injection)

Configuring Spring Container:

* XML configuration file (legacy)
* Java Annotations (modern)
* Java Source Code (modern)

Spring Development Process

* Configure your Spring Beans
* Create a Spring Container
* Retrieve Beans from Spring Container

*Step1*.

<beans … >

<bean id =”myCoach”

class=”com.luv2code.springdemo.BaseballCoach”>

</bean>

</beans>

*Step2*. The Spring container is generally known as ApplicationContext

Specialized implementations:

* ClassPathXmlApplicationContext
* AnnotationConfigApplicationContext
* GenericWebApplicationContext
* Others

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(“applicationContext.xml”);

*Step3*: Retrieve Beans from Container

Coach theCoach =context.getBean(“myCoach”, Coach.class);

Text

**5. Dependency Injection.**

The dependency inversion principle. The client delegates to calls to another object the responsibility of providing its dependencies.

Development Process – Constructor Injection:

1. Define the dependency interface and class
2. Create a constructor in your class of injections
3. Configure the dependency injection in Spring config file
   1. <bean id=”myCoach” … <constructor-arg ref=”DependencybeanId”>…

Development Process – Setter Injection:

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

Development Process – Inject literal values

1. Create setter method(s) in your class for injection
2. Configure the injection in Spring config file
   1. <property name=”team” value=”Sunrisers Hyderabad”>

Development Process – inject literal values from the properties file

1. Create Properties file
2. Load properties file in spring config file

<context:property-placeholder location=”classpath:sport.properties”/>

1. Reference values from Properties file
   1. ${the prop name}

**6**. **Spring Bean Scopes and Lifecycle**

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?

You can explicitly specify bean scope. If you want explicitly specify bean scope, then you make use of the scope attribute

Additional Spring Bean Scopes:

* Singleton – create a single shared instance of the bean. Default scope.
* Prototype – Creates a new bean instance for each container request
* Request – scoped to an HTTP web request. Only used for web apps.
* Session – scoped to an htttp web session. Only used for web apps.
* Global-session – scoped to a global HTTP web session. Only used for web apps.

Bean lifecycle:

1. Container started
2. Bean instantiated
3. Dependencies injected
4. Internal spring processing
5. Your custom init method (Bean is ready for use. Container is shutdown)
6. Custom destroy method

Bean lifecycle methods / Hooks: bean initialization, bean destruction:

<bean….

init-method=”doMyStartupStuff”

destroy-method=”doMyCleanupStuff”…

</bean>

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

When using XML configuration, I want to provide additional details regarding the method signatures of the init-method  and destroy-method .

**Access modifier**  
The method can have any access modifier (public, protected, private)

**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.

**Method name**  
The method can have any method name.

**Arguments**  
The method can not accept any arguments. The method should be no-arg.

 Although initialization lifecycle callback methods are called on all objects regardless of scope, ***in the case of prototypes, configured destruction lifecycle callbacks are not called***. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding.

**7. Spring Configuration with Java Annotations – Inversion of Control.**

Development Process:

1. *Enable component scanning in Spring config file*
   1. <beans … >

<context:component-scan base-package=”package” />

</beans

*Spring wiil recursively scan the package*

1. *Add the @Component Annotation to your Java classes*
   1. @Component(“thatSillyCoach”)

(beanId is thatSillyCoach)

1. *Retrieve bean from Spring container*

Spring supports Default Bean IDs. Default bean id: the class name, make first letter lower case.

**8. Spring Configuration with Java Annotations – Dependency Injection.**

What is Spring AutoWiring? For dependency injection, Spring can use auto wiring. Spring will look for a class that matches the property (mathes by type: class or interface). Once it finds, it will automatically inject the dependency.

Example:

* Injecting FortuneService into a Coach implementation
* Spring will scan Components
* Any one implements FortuneService Interface
* If so, let’s inject them. For example: HappyFortuneService

Autowiring Injection Types:

* Constructor Injection
* Setter Injection
* Field Injections

Development Process – 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. User @Autowired annotation before constructor, which has arg with dependency

Development Process –Setter Injection:

1. Create setter method(s) in your class for injections
2. Configure the dependency injection with @Autowired Annotation

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

Development Process – Fields Injection:

1. Configure the dependency injection with @Autowired Annotation
   1. Applied directly to the field
   2. No need for setter methods

@Qualifier(“beanId”) – The desired bean id, if you have multiple implementations. Can be applied to all defined injection types.

Note 1: As of Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean only defines one constructor to begin with. However, if several constructors are available, at least one must be annotated to teach the container which one to use.

Note 2: Qualifier with constructors:

@Autowired  
    public TennisCoach(@Qualifier("randomFortuneService") FortuneService theFortuneService) {  
  
         System.out.println(">> TennisCoach: inside constructor using @autowired and @qualifier");  
          
         fortuneService = theFortuneService;  
    }

Note 3: This solution will show you how inject values from a properties file using annotatons. The values will no longer be hard coded in the Java code.

**1. Create a properties file to hold your properties. It will be a name value pair.**

New text file:  src/sport.properties

foo.email=myeasycoach@luv2code.com

foo.team=Silly Java Coders

Note the location of the properties file is very important. It must be stored in src/sport.properties

**2. Load the properties file in the XML config file.**

File: applicationContext.xml

Add the following lines:

    <context:property-placeholder location="classpath:sport.properties"/>

This should appear just after the <context:component-scan .../> line

**3. Inject the properties values into your Swim Coach: SwimCoach.java**

@Value("${foo.email}")

private String email;

@Value("${foo.team}")

private String team;

---

**9. Spring configuration with Java Annotations – Bean Scopes and Lifecycle Methods.**

Explicitly specify bean scope using the @Scope annotation:

@Scope("singleton");

Scope methods/hooks. Development process:

1. Define your methods for init and destroy
2. Add annotations: @PostConstruct and @PreDestroy

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, and otherwise assembles a prototype object, and hands it to the client, with no further record of that prototype instance.

Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding.

To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.

**10. Spring configuration with Java Code.**

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 (AnnotationConfigApplicationContext)

Development Process:

1. Define method to expose bean (use @Bean annotation)
2. Inject bean dependencies
3. Read Spring Java configuration class
4. Retrieve bean from Spring container

Injecting values from the properties file:

1. Create properties file
2. Load properties file in Spring configuration (@PropertySource(“”))
3. Reference values from properties file (@Value (“${name}”) private field…

**11. Spring MVC – Building Spring Web Apps.**

What is Spring MVC?

* Framework for building web applications in Java
* Based on Model-View-Controller design pattern
* Leverages features of the Core Spring Framework (IoC, DI)

Spring MVC documentation:

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html

Components of a Spring MVC Application:

* A set of web pages to layout UI components
* A collection of Spring beans (controllers, services, etc…)
* Spring configuration (XML, Annotations or Java)

Front controller known as DispatcherServlet:

* Part of the Spring Framework
* Already developed by Spring Dev Team

You will create:

* Model objects
* View templates (JSP page)
* Controller classes (Business logic)

Controller contains business logic:

* Handle the request
* Store / retrieve data
* Place data in model

Send to appropriate view

Model contains your data:

* Store / retrieve data via backend systems
* Place your data in the model

View Template. Most common view template is JSP + JSTL. Developer creates a page. Displays data. Other templates supported (Thymeleaf, Groovy, Velocity, Freemarker)

Spring MVC Configuration Process – Part 1:

* Add configuration to file: WEB-INF/web.xml

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

* Add configuration to file WEB-INF/spring-mvc-demo-servlet.xml

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

Step 1. In our web.xml file, we need to add an entry for Spring DispatcherServlet, or the Front Controller:

<servlet>  
 <servlet-name>dispatcher</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

Once you have the servlet reference, then you set up initial parameter, so you basically tell where your Spring context configuration file is located.

<init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>/WEB-INF/spring-mvc-demo-servlet.xml</param-value>  
</init-param>

The next thing is set up the URL mappings for the Spring MVC DispatcherServlet:

<servlet-mapping>  
 <servlet-name>dispatcher</servlet-name>  
 <url-pattern>/</url-pattern>  
</servlet-mapping>

In this case our url patter will be slash, meaning all web requests, coming in, should be handled by the DispatcherServlet. Servlet name must match with the servlet reference.

Add support for conversion, formatting and validation in config.xml:

<mvc:annotation-driven/>

Define Spring MVC view resolver in config.xml:

<bean  
 class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/view/" />  
 <property name="suffix" value=".jsp" />  
</bean>

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

* Prepend the prefix
* Append the suffix

**12. Spring MVC – Creating Controllers and Views.**

Development Process:

1. Create Controller class:
   1. Annotate class with @Controller (extends from @Component)
2. Define Controller method (Any method name)
3. Add request Mapping to Controller method
   1. Add @RequestMapping(“/”) to the method
4. Return View Name
   1. Inside the controller method we need to return view name
5. Develop View Page

Form. 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.** The Model is a container for your application data. In your controller code you can put anything in the model. Your View Page (JSP) can access data from the model.

When we make use of our spring controller, we can actually pass the model to our controller, and we can also read form data in our controller. If you need to read form data in your controller code, then you pass in the HttpServletRequest and it’s working like the normal servlet request. You can pass Model. Model is just a container that can hold your form data. When the model comes in initially, it’s empty and you can add data to it.

Ex, Read the request parameter from the HTML form:

request.getParameter(“studentName”);

To add something to the model, I say model.addAttribute and the attribute I give the actual name of the attribute, comma, the value:

model.addAttribute(“message”, result);

Now in JSP page data from the model is accessible: ${message}

**13. Spring MVC – Request Params and Request Mappings.**

Spring has a special annotation called @RequestParam. This will allow you to read form data, and automatically bind it to a parameter coming into your method:

public String letsDo(  
    @RequestParam("student name") String theName, Model model) {  
    // use the variable  
}

Behind the scenes:

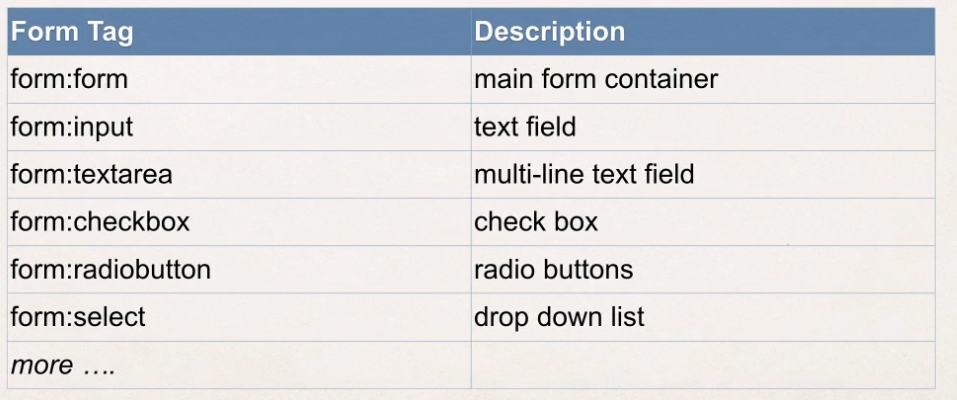
* Spring will read param from request
* Bind it to the variable

Adding Request Mappings to Controller. You can define a request mapping at the controller level. So, it basically serves as like the parent mapping for the controller. All of the request mappings on the methods are relative to the controller’s path.

**14. Spring MVC – Form Tags and Data Binding.**

Spring MVC Form Tags are the building block for a web page. Form Tags are configurable and reusable for a web page. Spring MVC Form Tags can make use of data binding. Automatically setting/retrieving data from Java Beans.

MVC Form Tags:



At the beginning of the jsp page, you make use of this tag lib reference. Your prefix equals form, and then you give the uri and solicit on the slide:

<%@ taglib prefix=”form” uri=<http://www.springframework.org/tags/form> %>

In your Spring Controller before you show the form, you must add a model attribute. This is a bean that will hold form data for the data binding.

**Handling submission in the Controller.** We can simply make use of a new Spring annotation called @ModelAttribute, and we give the name of the attribute.

Dropdown list is represented by the tag “select”:



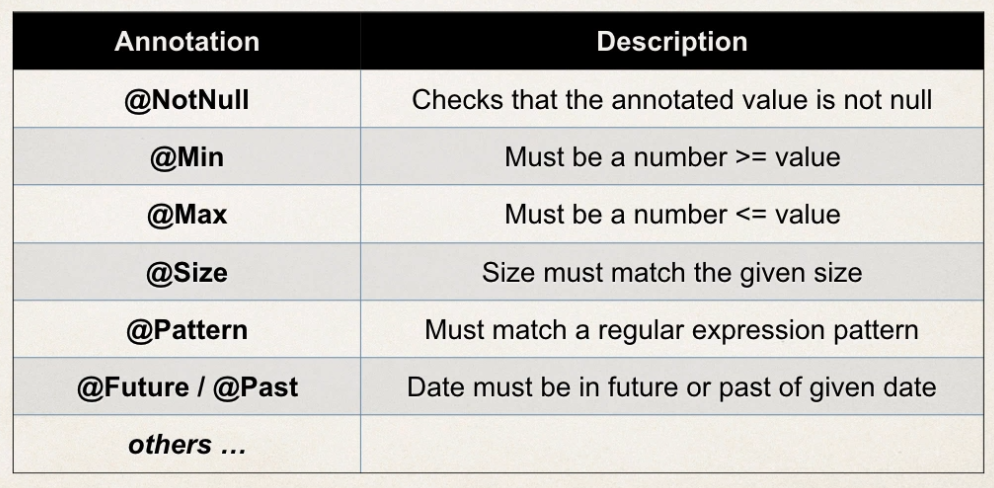
**15. Spring MVC Form Validation – Applying Built-In Validation Rules.**

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 (beanvalidation.org). Spring version 4 and higher supports Bean Validation API.

Validation feature:

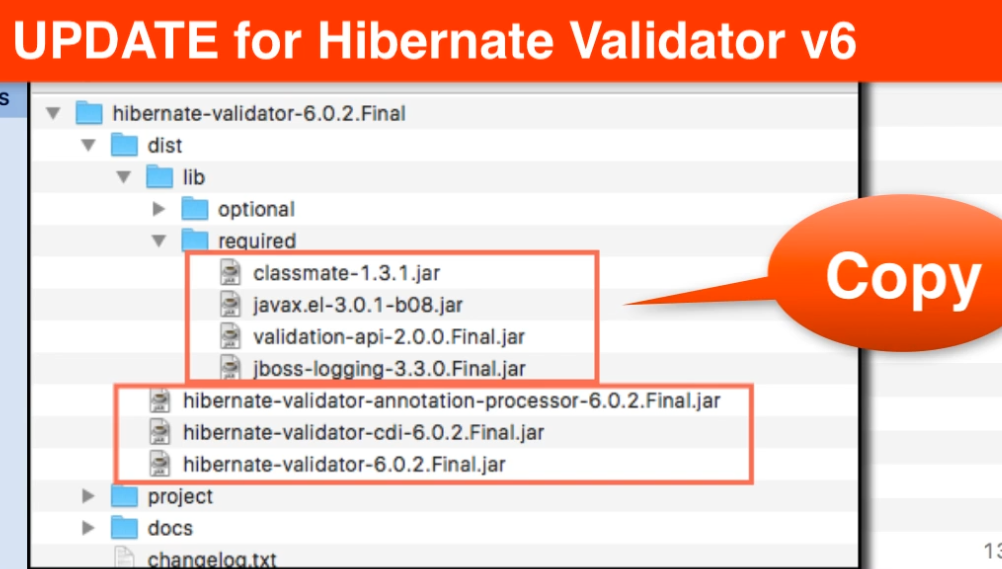
* Required
* Validate length
* Validate numbers
* Validate with regular expressions
* Custom validation

Validation annotations:



Hibernate.org/validator

Development process:

* Download jar
* Add jar to the project
* 

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

**16. Spring MVC Form Validation – Validating Number Ranges and Regular Expressions.**

**17. Spring MVC Form Validation – Creating Custom Validation Rules.**

@Constraint(validatedBy = CourseCodeConstraintValidator.**class**)

@Target({ ElementType.***METHOD***, ElementType.***FIELD*** } )

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** CourseCode {

// define default course code

**public** String value() **default** "LUV";

// define default error message

**public** String message() **default** "must start with LUV";

// define default groups

**public** Class<?>[] groups() **default** {};

//define default payloads

**public** Class<? **extends** Payload>[] payload() **default** { };

}

**public** **class** CourseCodeConstraintValidator

**implements** ConstraintValidator<CourseCode, String>{

**private** String coursePrefix;

@Override

**public** **boolean** isValid(String theCode, ConstraintValidatorContext arg1) {

**if** (theCode != **null**) {

**boolean** result = theCode.startsWith(coursePrefix);

**return** result;

}

**return** **true**;

}

@Override

**public** **void** initialize(CourseCode theCourseCode) {

coursePrefix = theCourseCode.value();

}

}

**18. Introduction to Hibernate.**

Hibernate.org

**19. Setting Up Hibernate Development Environment.**

**20. Hibernate Configuration with Annotations.**

Development process:

* Add Hibernate Configuration file
* Annotate Java Class
* Develop Java Code to perform database operations

Hibernate.cfg.xml:

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<!-- JDBC Database connection settings -->

<property name=*"connection.driver\_class"*>com.mysql.jdbc.Driver</property>

<property name=*"connection.url"*>jdbc:mysql://localhost:3306/hb\_student\_tracker?useSSL=false</property>

<property name=*"connection.username"*>hbstudent</property>

<property name=*"connection.password"*>hbstudent</property>

<!-- JDBC connection pool settings ... using built-in test pool -->

<property name=*"connection.pool\_size"*>1</property>

<!-- Select our SQL dialect -->

<property name=*"dialect"*>org.hibernate.dialect.MySQLDialect</property>

<!-- Echo the SQL to stdout -->

<property name=*"show\_sql"*>true</property>

<!-- Set the current session context -->

<property name=*"current\_session\_context\_class"*>thread</property>

</session-factory>

</hibernate-configuration>

**Entity Class –** Java class that is mapped to a database table.

**Two options for Mapping:**

* XML (legacy)
* Java Annotations (modern)

Java annotations:

* Map class to database table:

**import** javax.persistence.Entity;

**import** javax.persistence.Table;

@Entity

@Table(name="student")

**public** **class** Student {

* Define no-arg constructor
* Map fields to database columns:

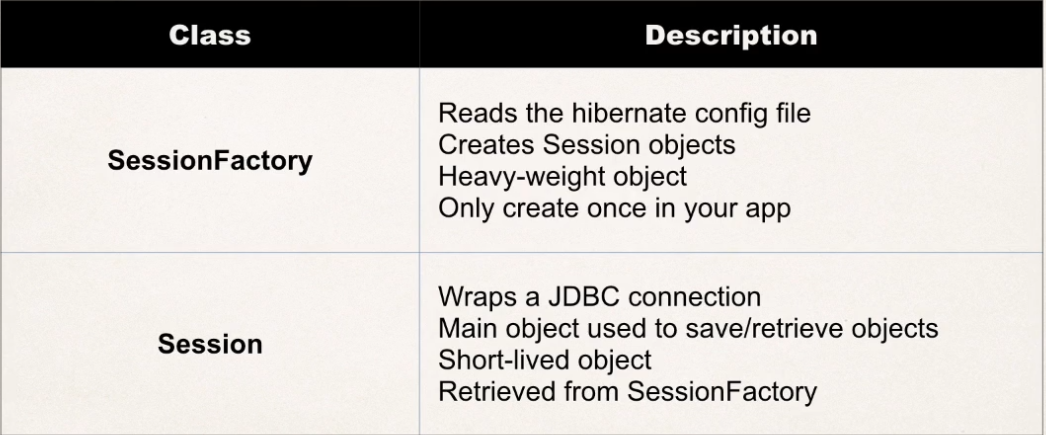
@Id

@Column(name="id")

**private** **int** id;

**21. Hibernate CRUD Features Create Read Update and Delete.**

**Two key players:**



**Example:**

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.cfg.Configuration;

**import** com.luv2code.hibernate.demo.entity.Student;

**public** **class** CreateStudentDemo {

**public** **static** **void** main(String[] args) {

// create session factory

// create session

**try** (SessionFactory factory = **new** Configuration()

.configure("hibernate.cfg.xml")

.addAnnotatedClass(Student.**class**)

.buildSessionFactory();

Session session = factory.getCurrentSession()) {

// create a student object

Student tempStudent = **new** Student("Paul", "Wall", "paul@luv2code.com");

// start a transaction

session.beginTransaction();

// save the student object

session.save(tempStudent);

// commit transaction

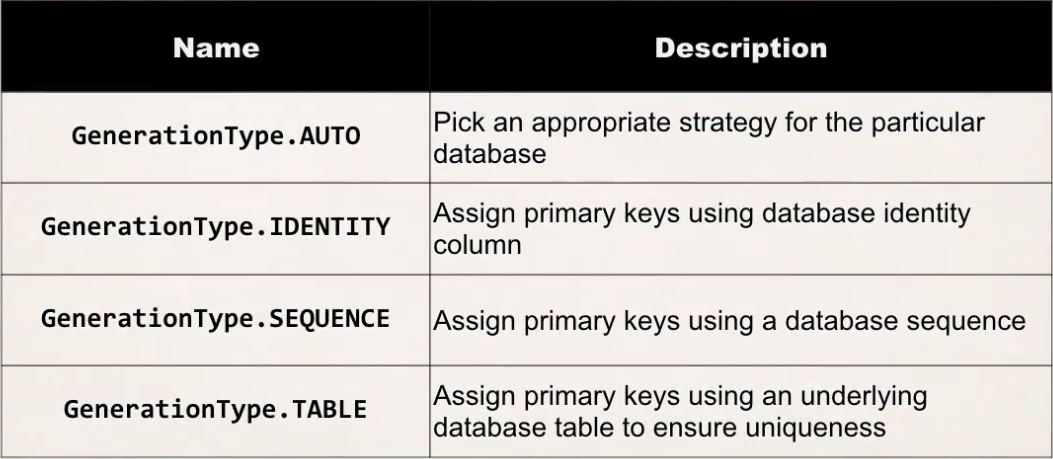
session.getTransaction().commit();

}

}

}

**ID Generation Strategies:**



You can define your own custom generation strategy:

* Create subclass of org.hibernate.id.SequenceGenerator
* Override the method: public Serializable generate(…)

Retrieving:

session.get(Student.**class**, tempStudent.getId());

session.createQuery("from Student s where s.lastName='Doe'").list();

**22. Hibernate Advanced Mappings.**

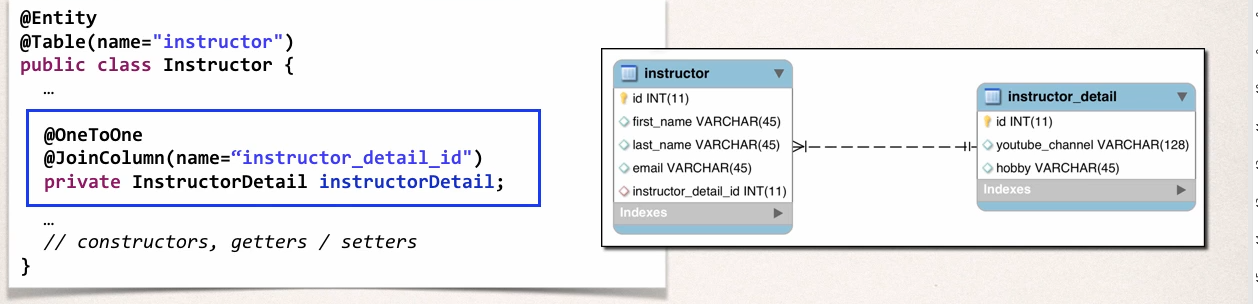
Advanced mappings:

* One-to-one (instructor -> instructor detail)
* One-to-Many, Many-to-One (instructor can have many courses, many courses can have one instructor)
* Many-to-Many (a course can have many students, a student can have many courses)

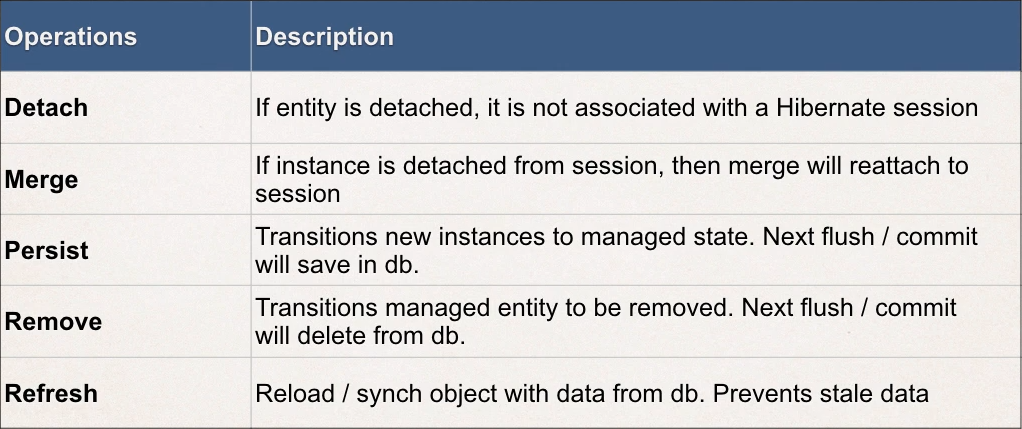
You can cascade operations. Apply the same operation to related entities.

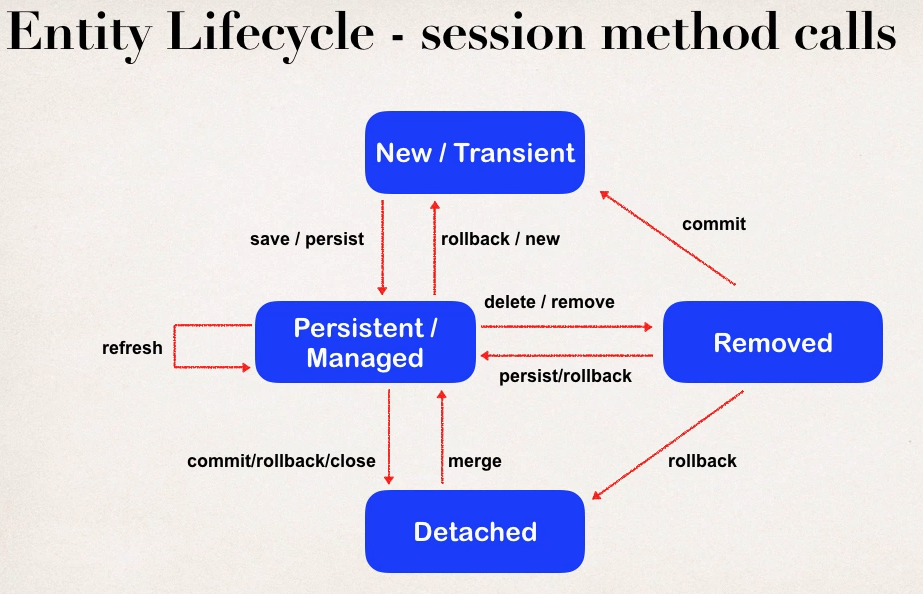
**23. Hibernate Advanced Mappings – OneToOne.**

OneToOne example:



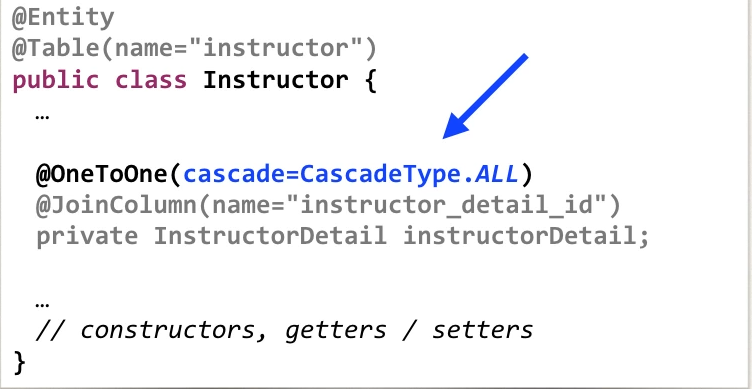
Entity lifecycle:





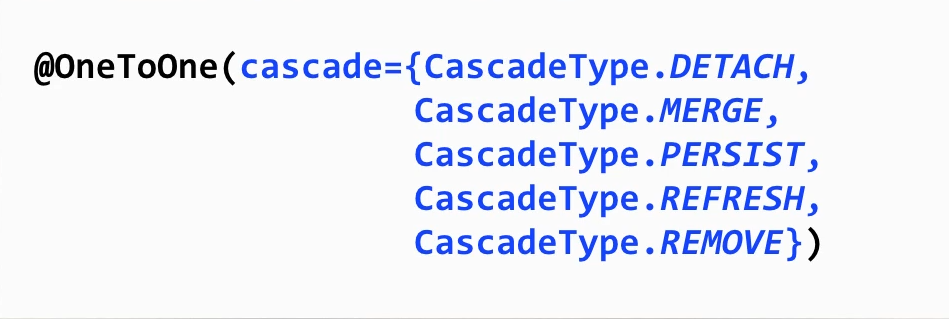
Cascade Types:





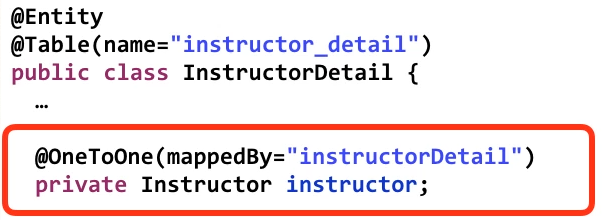
By default, no operations are cascaded.

Configure Multiple Cascade Types:





**Bi-Directional relationship.** Use@OneToOne annotation.

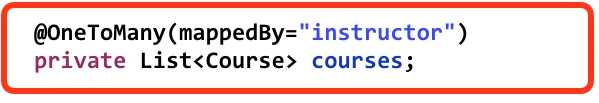


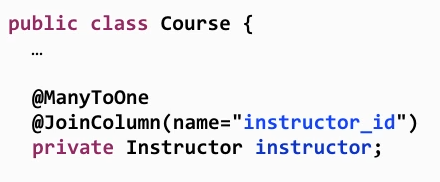
Set related field to null first, to delete detail record.

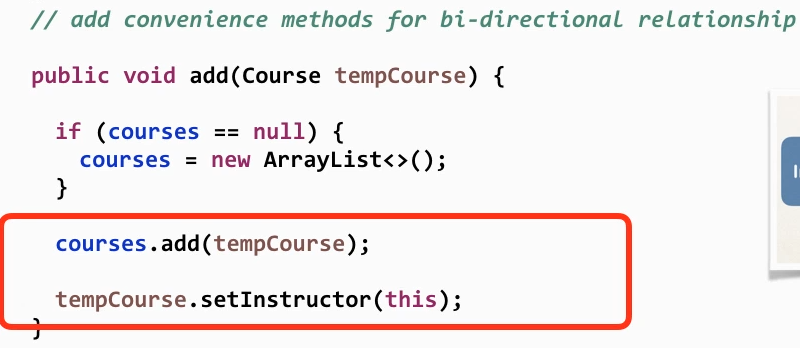
**24. Hibernate Advanced Mappings – One to Many.**

Real world requirement: Do not apply cascading delete!

Use @OneToMany annotation:







**25. Hibernate Advanced Mappings - Eager vs Lazy Loading.**

**Eager** will retrieve everything

**Lazy** will retrieve on request

Eager loading will load all dependent entities.

Lazy loading will load the main entity first. Load dependent entities on demand.

When you define the mapping relationship. You can specify fetch type: FetchType.EAGER or FetchType.LAZY.



When you lazy load, the data is only retrieved on demand. However, this requires an open Hibernate session (need connection to database). If the hibernate session is closed and you attempt to retrieve lazy data Hibernate will throw an exception. To retrieve lazy data you will need to open a Hibernate session. Retrieve lazy data using:

* Session.get and call appropriate getter method(s)
* Hibernate query with HQL:

Query<Instructor> query =

session.createQuery("select i from Instructor i "

+ "JOIN FETCH i.courses "

+ "where i.id=:theInstructorId",

Instructor.**class**);

query.setParameter("theInstructorId", theId);

Instructor tempInstructor = query.getSingleResult();

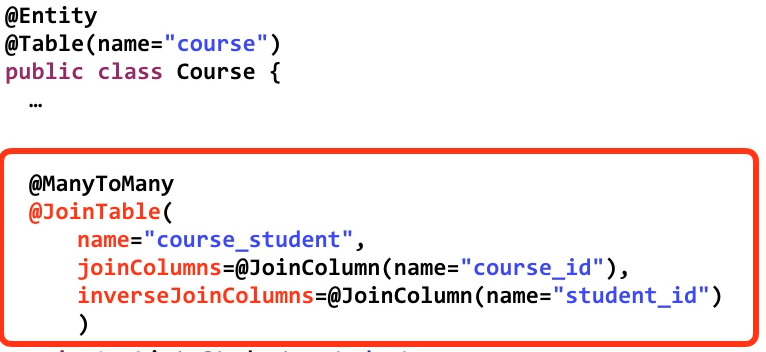
Many other techniques are available.

**26. Hibernate Advanced Mappings – One to Many – Unidirectional.**

For unidirectional relationship, do not set column info in detail entity.

**27. Hibernate Advanced Mappings – Many to Many.**

Add annotations:



Inverse refers to the “other side” of the relationship.

Real world project requirement: Do not cascade delete