Spring & Hibernate Framework

seed®

beyond the obvious

Spring Framework

Compiled by SEED Infotech Ltd.

Index

Chapter 1 - Introduction to Spring framework

Spring Framework is a Java platform that provides comprehensive infrastructure support for developing enterprise applications. Spring handles the infrastructure so you can focus on your application.

Spring framework is an open source Java platform and it was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 2MB.

The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use and promote good programming practice by enabling a POJO-based programming model.

Spring enables you to build applications from “plain old Java objects” (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.

Benefits of Using Spring Framework:

Following is the list of few of the great benefits of using Spring Framework:

- Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
- Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about ones you need and ignore the rest.

- Spring does not reinvent the wheel instead, it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, other view technologies.
- Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBean-style POJOs, it becomes easier to use dependency injection for injecting test data.
- Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over engineered or less popular web frameworks.
- Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
- Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
- Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

Examples of how you, as an application developer, can use the Spring platform advantage:

- Make a Java method execute in a database transaction without having to deal with transaction APIs.
- Make a local Java method a remote procedure without having to deal with remote APIs.
- Make a local Java method a management operation without having to deal with JMX APIs.
- Make a local Java method a message handler without having to deal with JMS APIs.

Dependency Injection and Inversion of Control Background

Java applications -- a loose term that runs the gamut from constrained applets to n-tier server-side enterprise applications -- typically consist of objects that collaborate to form the application proper. Thus the objects in an application have *dependencies* on each other.

Although the Java platform provides a wealth of application development functionality, it lacks the means to organize the basic building blocks into a coherent whole, leaving that task to architects and developers. True, you can use design patterns such as *Factory*, *Abstract Factory*, *Builder*, *Decorator*, and *Service Locator* to compose the various classes and object instances that make up an application. However, these patterns are simply that: best practices given a name, with a description of what the pattern does, where to apply it, the problems it addresses, and so forth. Patterns are formalized best practices that *you must implement yourself* in your application.

The Spring Framework *Inversion of Control* (IoC) component addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use. The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into your own applications.

Dependency Injection (DI)

The technology that Spring is most identified with is the **Dependency Injection (DI)** flavor of Inversion of Control. The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control.

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing. Dependency Injection helps in gluing these classes together and same time keeping them independent.

What is dependency injection exactly? Let's look at these two

words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent on class B. Now, let's look at the second part, injection. All this means is that class B will get injected into class A by the IoC.

Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods. As Dependency Injection is the heart of Spring Framework, so I will explain this concept in a separate chapter with a nice example.

Aspect Oriented Programming (AOP)

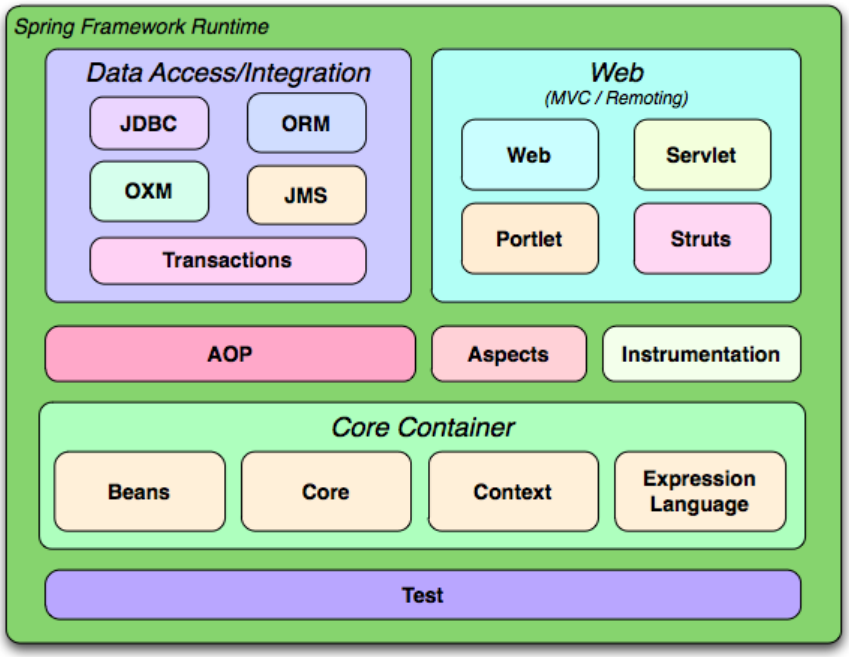
One of the key components of Spring is the **Aspect oriented programming (AOP)** framework. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, and caching etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Whereas DI helps you decouple your application objects from each other, AOP helps you decouple cross-cutting concerns from the objects that they affect.

The AOP module of Spring Framework provides aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. I will discuss more about Spring AOP concepts in a separate chapter.

Spring Modules

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the following diagram.



Overview of the Spring Framework

Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules.

The Core and Beans modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The Context module builds on the solid base provided by the Core and Beans modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The Context module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event-propagation, resource-loading, and the transparent creation of contexts by, for example, a servlet container. The Context module also supports Java EE features such as EJB, JMX ,and basic remoting. The ApplicationContext interface is the focal point of the Context module.

The Expression Language module provides a powerful expression language for querying and manipulating an object graph at runtime. It is an extension of the unified expression

language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection as well as common list aggregations.

Data Access/Integration

The *Data Access/Integration* layer consists of the JDBC, ORM, OXM, JMS and Transaction modules.

- The JDBC module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.
- The *ORM* module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis. Using the ORM package you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.
- The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service (JMS) module contains features for producing and consuming messages.
- The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for *all your POJOs (plain old Java objects)*.

Web

The *Web* layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules.

Spring's *Web* module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context. It also contains the web-related

parts of Spring's remoting support.

The *Web-Servlet* module contains Spring's model-view-controller (*MVC*) implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.

The *Web-Struts* module contains the support classes for integrating a classic Struts web tier within a Spring application. Note that this support is now deprecated as of Spring 3.0. Consider migrating your application to Struts 2.0 and its Spring integration or to a Spring MVC solution.

The *Web-Portlet* module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

AOP and Instrumentation

Spring's AOP module provides an AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Using source-level metadata functionality, you can also incorporate behavioral information into your code, in a manner similar to that of .NET attributes.

The separate Aspects module provides integration with AspectJ.

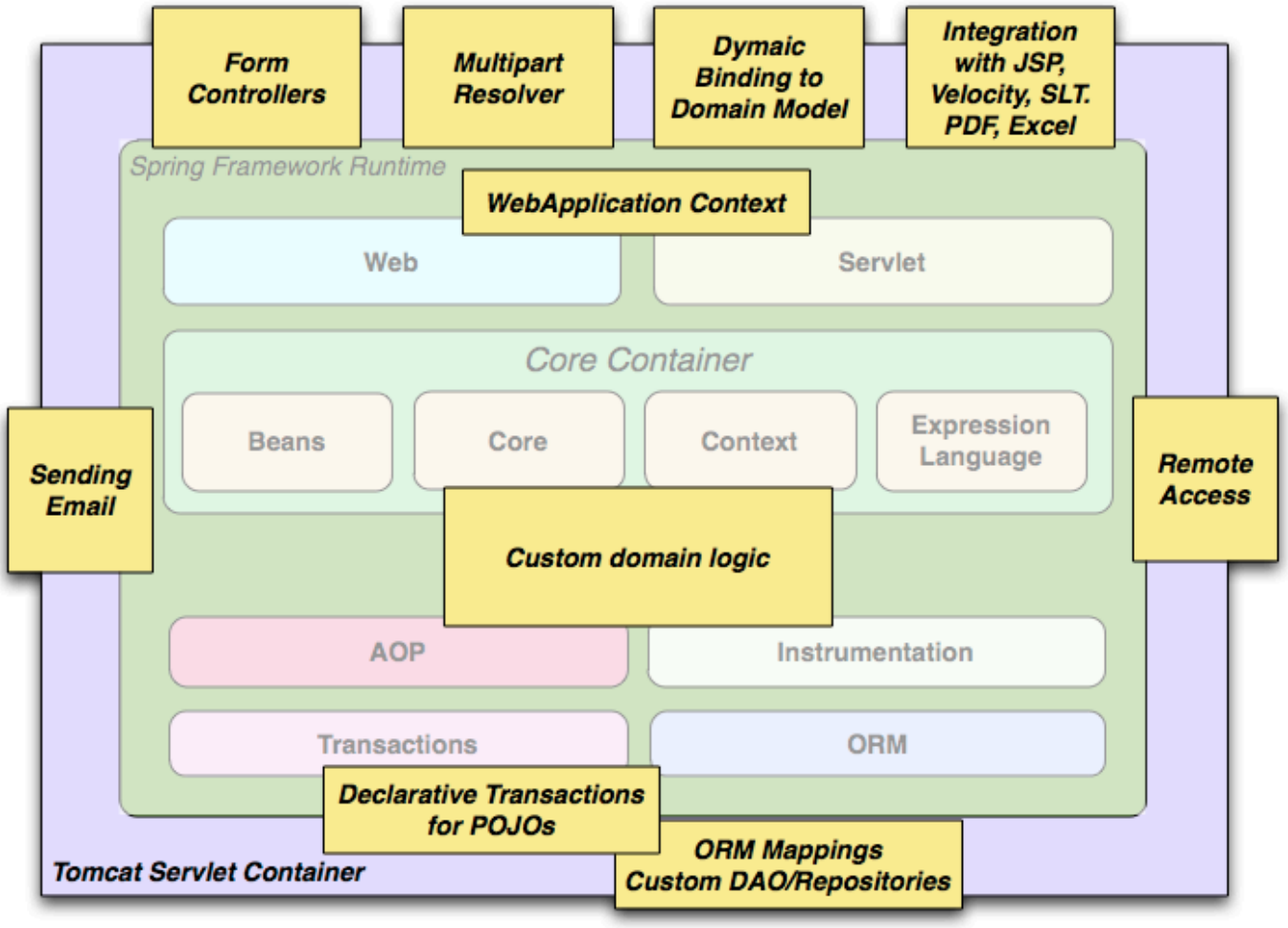
The Instrumentation module provides class instrumentation support and classloader implementations to be used in certain application servers.

Test

The *Test* module supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

Usage scenarios

The building blocks described previously make Spring a logical choice in many scenarios, from applets to full-fledged enterprise applications that use Spring's transaction management functionality and web framework integration.

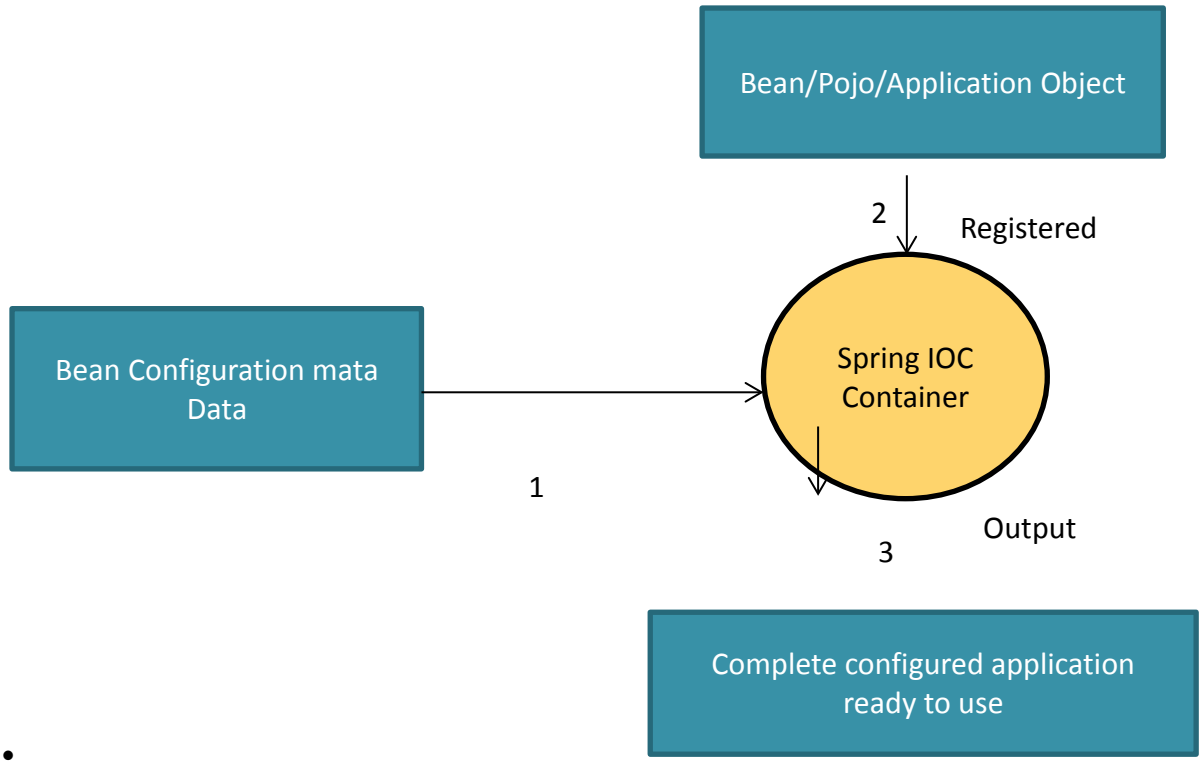


Spring Container

Primary function of Spring Container is

- Create and manage the Object (Inversion of control)
- Inject the dependencies (Dependency injection)

Spring Development Process steps



-
- Step-1 Create a bean /Pojo class
- Step-2 Configure a bean class
- Step-3 Create a Spring container
- Step-4 Retrieved bean from Spring Container

Step-1 (bean/Pojo) class

```
Class Employee{
        private int employeeId;
  private String name;
      public Employee()
      {
      System.out.println("From
  constructor");
                          }
                                getter()/setter()
      ……
}
```

Step-2 XML configuration file applicationContext.xml

```
<beans ….>
  <bean id="employee"
class="com.seed.Employee">
  </bean>
</beans>
```

```
bean :  The Element which is most basic
configuration unit in Spring. It tells Spring
Container to create an Object.
id :      The Attribute which gives the bean a
name by which it
           will be referred to in the Spring
Container.
class : The Attribute which tells Spring the
type of a Bean.
```

Configuring Spring Container :

Spring container can be configured by using following ways

- XML configuration file (legency but most of legency application still use this)
- Java Annotation
- Java Source Code.

Step-3 & 4

```
ApplicationContext applicationContext=new
ClassPathXmlApplicationContext("applicationCont
ext.xml");
System.out.println("After application context
");
Employee employee =
(Employee)applicationContext.getBean("emp");
```

Example of configuring spring container using Annotation

Spring Configuration using Annotation

- Annotation minimizes xml configuration.
- Spring will scan all java classes for special annotation.
- These annotation automatically register the bean in the container.
- The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context.
- To use this annotation, apply it over class as below:

```
@Component
public class Employee {
    //data member
//getters and setters
}
```

To enable this scanning, you will need to use "context:component-scan" tag in your applicationContext.xml file. e.g

```
<context:component-scan base-
package="com.seed.service" />
```

File Appmain.java

```
//Spring using Autoscan
        ApplicationContext context =
                    new
ClassPathXmlApplicationContext(new String[]
{"applicationContext.xml"});
        Employee obj=(Employee)
context.getBean("employeeBean");
        System.out.println(obj);
```

Spring Configuration using Java class

Step 1 – Create a bean class

```
@Component
public class Employee {
    //data member
//getters and setters
}
```

Step 2 - Create a configuration class

@Configuration Indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

```
package com.spring.configuration;

import
org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Componen
tScan;
import
org.springframework.context.annotation.Configur
ation;
import
org.springframework.context.annotation.Descript
ion;
import com.spring.model.Employee;

@Configuration
public class EmployeeConfig {
  @Bean(name="employeeBean")
    @Description("This is a sample Employee
Bean")
  public Employee EmployeeConfig() {
      return new Employee();
  }
}
```

@Component

If we mark a class with @Component or one of the other Stereotype annotations these classes will be auto-detected using classpath scanning. As long as these classes are in under our base package or Spring is aware of another package to scan, a new bean will be created for each of these classes.

@Bean is used to explicitly declare a single bean.

@Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions.

Step 3 - Create main class to run application

```
//spring using Java configuarion using
annoation
       AbstractApplicationContext context = new
AnnotationConfigApplicationContext(EmployeeConf
ig.class);
          Employee bean = (Employee)
context.getBean("employeeBean");
          bean.setEmployeeId(444);
          bean.setName("jayshree");
          System.out.println(bean);
          context.close();
```

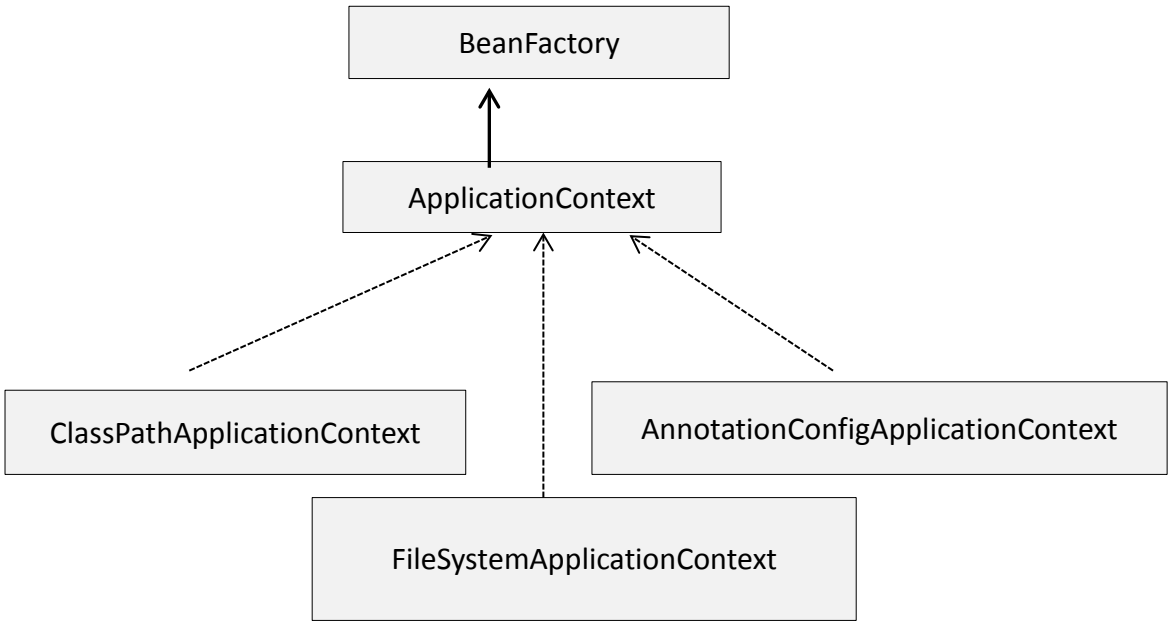
Chapter 2 - Spring Core

Spring Ioc Containers

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application. These objects are called Spring Beans which we will discuss in next chapter.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram is a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

Spring provides following two distinct types of containers.



Spring Container API

- BeanFactory is the root interface of Spring IoC container. ApplicationContext is the child interface of BeanFactory interface that provide Spring AOP features, i18n etc.
- Spring IoC container classes are part of org.springframework.beans and org.springframework.context packages.
- Spring IoC container provides us different ways to decouple the object dependencies.

ApplicationContext implementation classes

- **AnnotationConfigApplicationContext**: It is used in Spring standalone java application using annotation for configuration.
- **ClassPathXmlApplicationContext**: It is used in Spring standalone java application for spring bean configuration in xml file .
- **FileSystemXmlApplicationContext:** This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.
- **AnnotationConfigWebApplicationContext** and **XmlWebApplicationContext** for web applications.

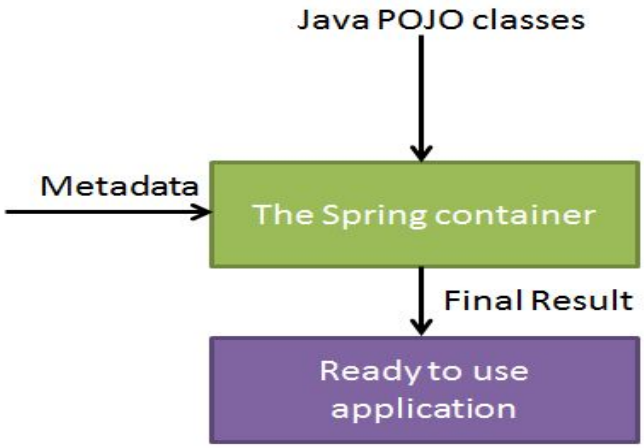
1. Spring BeanFactory Container

This is the simplest container providing basic support for DI and defined by the org.springframework.beans.factory.BeanFactory interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring.

2. Spring ApplicationContext Container

This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the org.springframework.context.ApplicationContext interface.

The ApplicationContext container includes all functionality of the BeanFactory container, so it is generally recommended over the BeanFactory. BeanFactory can still be used for light weight applications like mobile devices or applet based applications where data volume and speed is significant.

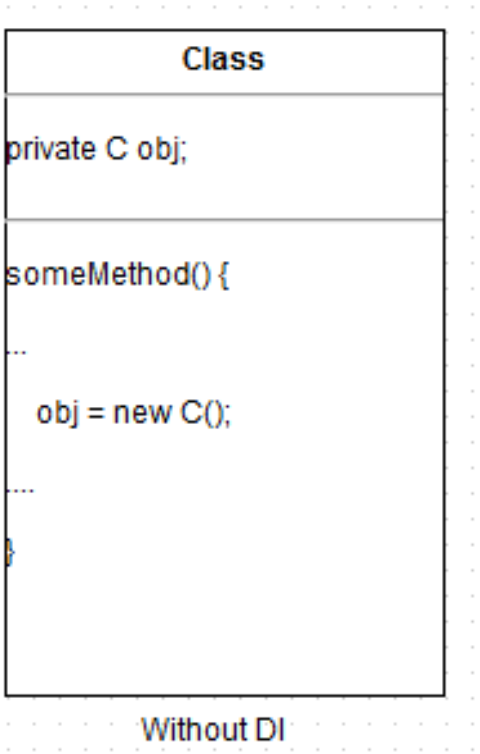


Dependency Injection

Dependency injection is a jargon created by Martin Fowler and it is also called as inversion of control. In object oriented design, objects have relationship with one another. A class (A) can have attributes (B) and methods. Those attributes are again instances of another class (C). If class (A) wants to work and perform its objective, attributes (B) should be instantiated.

There are different ways to instantiate an object and we have seen a lot in our design pattern tutorial series. A simple and direct way is to use the “new” operator and call the constructor of Class (C) where we need that instance in class (A). This is class A has obsolute control over creation of attribute (B). It decides which class (C) to call and how to call etc.

Now, if we outsource that 'instantiation and supplying an instance' job to some third party. Class (A) needs instance of class (C) to operate, but it outsources that responsibility to some third party. The designated third party, decides the moment of creation and the type to use to create the instance. The dependency between class (A) and class (C) is injected by a third party. Whole of this agreement involves some configuration information too. This whole process is called dependency injection.



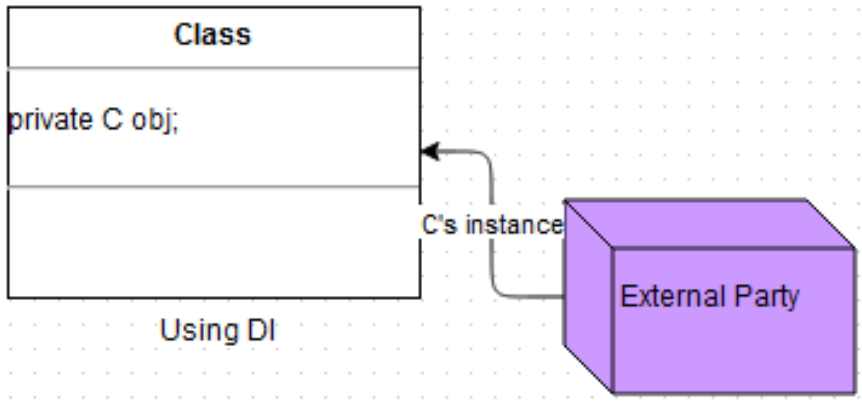
Difference between Dependency Injection and Factory

Factory design pattern and dependency injection may look related but we look at them with microscope then we can understand the are different. Even if we use a factory the dependent class has the responsibility of creating the instance but the core of dependency injection is separating that responsibility to external component.

Factory Example:

```
Class A {
private C obj;

public void someMethod() {
   ...
   this.obj = MyObjectFactory.getC();
   ...
}
}
```

With Dependency Injection:




```
Class A {
private C obj;

public void someMethod(C obj) {
   ...
   this.obj = obj;
   ...
}}
```

With DI the contract is different, pass C's instance to get the job done. So the responsibility is with an external person to decide.

Advantages of Dependency Injection

- Loosely couple architecture.
- Separation of responsibility
- Configuration and code is separate.
- Using configuration, a different implementation can be

supplied without changing the dependent code.

- Testing can be performed using mock objects.

Dependency Injection Types

Dependency injection is classified into three categories namely,

- Constructor Injection
- Setter Injection

- **Constructor Injection** - as the name suggests, in this form the dependencies are injected using a constructor and hence the constructor needs to have all the dependencies (either scalar values or object references) declared into it. Though, this form of DI is directly supported by the **Spring IoC Container**. But, Constructor Injection is mainly used by a highly embeddable, lightweight, and full-service IoC Container named PicoContainer.
- **Setter/Mutator Injection** - this form of DI uses the setter methods (also known as mutators) to injectthe dependencies into the dependent components. Evidently all the dependent objects will have setter methods in their respective classes which would eventually be used by the Spring IoC container. This form of DI is also directly supported by Spring IoC Container. Though Spring framework supports both forms of DI namely the Constructor Injection and the Setter/Mutator Injection directly, but the IoC Container prefers the latter over the first.

Example :

DI through XML configuration

File :Employee.java

```
package com.spring.model;

public class Employee {
    int id;
    String name;
    EmployeeAddress address;

  public Employee() {
```

```
        super();
        // TODO Auto-generated constructor stub
    }
    public Employee(int id, String name,
EmployeeAddress address) {
        super();
        this.id = id;
        this.name = name;
        this.address = address;
    }
    //getters and setter }
```

File :EmployeeAddress.java

```
package com.spring.model;

public class EmployeeAddress {

    String city,state,landmark;


    //default constructor

    //parametric constructor

    //setter and getters
```

```
<beans…>
<!--  property Injection  -->
<bean id="addrp"
class="com.spring.model.EmployeeAddress">
    <property name="city" value="Nasik"/>
    <property name="state"
value="Maharashtra"></property>
    <property name="landmark" value="Gangapur
Road"></property>
</bean>
<bean id="ep" class="com.spring.model.Employee">
```

```
   <property name="id" value="444"></property>
   <property name="name"
value="kavita"></property>
   <property name ="address" ref="addrp"
></property>
</bean>
<!--  Constructor  Injection  -->
<bean id="addrc"
class="com.spring.model.EmployeeAddress">
   <constructor-arg name="city" value="Pune"
></constructor-arg>
   <constructor-arg name="state"
value="Maharashtra" ></constructor-arg>
   <constructor-arg name="landmark"
value="Senapati Bapat road" ></constructor-arg>
</bean>
<bean id="ec" class="com.spring.model.Employee">
   <constructor-arg name="id"
value="231"></constructor-arg>
   <constructor-arg name="name" value="Kapil
K"></constructor-arg>
   <constructor-arg name="address"
ref="addrc"></constructor-arg>
</bean>
</beans>
```

File : App.java

```
package com.spring.main;
import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXml
ApplicationContext;
import com.spring.model.Employee;
/**
 * Dependancy Injection  Using XML configuration
 *
 */
```

```
public class App
{
   public static void main(String[] args) {
       ApplicationContext applicationContext=new
ClassPathXmlApplicationContext("applicationConte
xt.xml");
       System.out.println("After application
context ");
       Employee epobj =
(Employee)applicationContext.getBean("ep");
       System.out.println("By using Proerty
injection Employee Object is :-");
       System.out.println(epobj);

       Employee ecobj =
(Employee)applicationContext.getBean("ec");
       System.out.println("By using Proerty
injection Employee Object is :-");
       System.out.println(ecobj);
   }
}
```

DI using Annotation

Bean Auto wiring

You have learnt how to declare beans using the <bean> element and inject <bean> with using <constructor-arg> and <property> elements in XML configuration file.

The Spring container can autowire relationships between collaborating beans without using <constructor-arg> and <property> elements which helps to cut down on the amount of XML configuration you write for a big Spring based application.

Autowiring Modes

Mode	Description
no	This is default setting which means no autowiring and you should use explicit bean reference for wiring.
byName	Autowiring by property name. Spring container looks at the properties of the beans on which autowire attribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
byType	Autowiring by property datatype. Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exists, a fatal exception is thrown.
constructor	Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
autodetect	Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType.

There are following autowiring modes which can be used to instruct Spring container to use autowiring for dependency injection. You use the autowire attribute of the <bean/> element to specify autowire mode for a bean definition.

You can use byType or constructor autowiring mode to wire arrays and other typed-collections

Autowiring Example for Employee and EmployeeAdress

Bean Configuration file

Auto-Wiring 'no'

This is the default mode, you need to wire your bean via 'ref' attribute.

```
<bean id="addrp"
class="com.spring.model.EmployeeAddress">
  <property name="city" value="Nasik"/>
  <property name="state"
value="Maharashtra"></property>
  <property name="landmark" value="Gangapur
Road"></property>
</bean>
<bean id="ep"
class="com.spring.model.Employee">
  <property name="id" value="444"></property>
  <property name="name"
value="kavita"></property>
  <property name ="address" ref="addrp"
></property>
</bean>
```

Auto-Wiring 'byName'

Auto-wire a bean by property name. In this case, since the name of "address" bean is same with the name of the "employee" bean's property ("address"), so, Spring will auto wired it via setter method – "setAddress(EmployeeAddress address)".

```
   <bean id="employee"
class="com.spring.model.Employee"
autowire="byName" />
   <bean id="address" class="
com.spring.model.EmployeeAddress " />
```

Auto-Wiring 'byType'

Auto-wire a bean by property data type. In this case, since the data type of "address" bean is same as the data type of the "employee" bean's property (address object), so, Spring will auto wired it via setter method – "setPerson(Person person)".

```
   <bean id="employee"
class="com.spring.model.Employee"
autowire="byType" />
   <bean id="address" class="
com.spring.model.EmployeeAddress " />
```

Auto-Wiring 'constructor'

Auto-wire a bean by property data type in constructor argument. In this case, since the data type of "address" bean is same as the constructor argument data type in "employee" bean's property (address object), so, Spring auto wired it via constructor method – "public Employee(EmployeeAddress address)".

```
<bean id="employee"
class="com.spring.model.Employee"
autowire="constructor" />
<bean id="address" class="
com.spring.model.EmployeeAddress " />
```

Auto-Wiring 'autodetect'

If a default constructor is found, uses "employee"; Otherwise, uses "byType". In this case, since there is a default constructor in "Employee" class, so, Spring auto wired it via constructor method – "public Employee(EmployeeAddress address) ".

```
<bean id="employee"
class="com.spring.model.Employee"
autowire="autodetect" />
<bean id="address" class="
com.spring.model.EmployeeAddress " />
```

Note : Autodetect functionality is applied when used with the 2.5 and 2.0 schemas. It has been deprecate from 3.0+

DI using Annotation

@Resource	Annotation used to inject an object that is already in the Application Context. It searches the instance by name. It also works on setter methods.
@Autowired	Annotation used to inject objects in many possible ways, such as: instance variable, constructor and methods. It does not rely on name as @Resource, so, for multiple concrete implementations, the @Qualifier annotation must be used with it
@Qualifier	Annotation used to distinguish between mult-ipleconcrete implementations. Used alongside with @Autowired annotation that does not rely on name.

@Resource

Spring supports injection using the @Resource annotation, when it is applied on a field or setter method of a bean.

The @Resource annotation follows by-name semantics i.e. it takes a *name* attribute for injection, which is analogous to '*autowiring by Name'* in XML based configuration.

Example: Application.java

```
package com.spring.model;
import javax.annotation.Resource;
import
org.springframework.stereotype.Component;
@Component("application")
public class Application {
   @Resource(name="applicationUser")
      private ApplicationUser user;
      @Override
      public String toString() {
          return "Application [user=" + user +
"]";
      }
}
```

Standard `@Resource` annotation marks a resource that is needed by the application. It is analogous to @Autowired in that both injects beans by type when no attribute provided. But with name attribute, @Resource allows you to inject a bean by it's name.

File :ApplicationUser.java

```
package com.spring.model;
import org.springframework.stereotype.Component;
@Component("applicationUser")
public class ApplicationUser {
   private String name = "defaultName";
              //getters and setters
   // toString()
```

In above code, Application's user property is annotated with

@Resource(name="applicationUser"). In this case, a bean with name 'applicationUser' found in applicationContext will be injected here.

File : AppConfig.java

```
package com.spring.configuration;
import
org.springframework.context.annotation.Component
Scan;
import
org.springframework.context.annotation.Configura
tion;
@Configuration
@ComponentScan("com.spring")
public class AppConfig {
}
```

Notice `@ComponentScan` which will make Spring auto detect the annotated beans via scanning the specified package and wire them wherever needed (using @Resource or @Autowired).

Example

@Autowired

In last Spring auto-wiring in XML example, it will autowired the matched property of any bean in current Spring container. In most cases, you may need autowired property in a particular bean only.

In Spring, you can use **@Autowired** annotation to auto wire bean on the setter method, constructor or a field. Moreover, it can autowired property in a particular bean.

Note

package com.spring.model;

```
import org.springframework.stereotype.Component;
@Component
public class License {
    private String number="123456ABC";

    @Override
    public String toString() {
        return "License [number=" + number + "]";
    }
        //setters, getters
}
```

The @Autowired annotation is auto wire the bean by matching data type.

@Autowired on Setter method

```
package com.spring.model;
import
org.springframework.beans.factory.annotation.Aut
owired;
import org.springframework.stereotype.Component;
@Component("driver")
public class Driver {

    private License license;

    @Autowired
```

```
    public void setLicense(License license) {
        this.license = license;
    }
    public License getLicense() {
        return license;
    }
    @Override
    public String toString() {
        return "Driver [license=" + license +
"]";
    }
}
```

@Autowired on Constructor

```
package com.spring.model;
import
org.springframework.beans.factory.annotation.Aut
owired;
import org.springframework.stereotype.Component;
@Component("driver")
public class Driver {
    private License license;
    @Autowired
    public void setLicense(License license) {
        this.license = license;
    }
public License getLicense() {
        return license;
    }
    @Override
    public String toString() {
        return "Driver [license=" + license +
"]";
    }
}
```

@Qualifier

`@Qualifier` is useful for the situation where you have more than one bean matching the type of dependency and thus resulting in ambiguity.

File : Car.java

```
package com.spring.model;
public interface Car {
  public void getCarName();

}
```

File :Bond.java

```
package com.spring.model;
import
org.springframework.beans.factory.annotation.Au
towired;
import
org.springframework.beans.factory.annotation.Qu
alifier;
import
org.springframework.stereotype.Component;
@Component
public class Bond {
  @Autowired
  @Qualifier("Ferari")
  private Car car;
  public void showCar(){
      car.getCarName();
  }
}
```

File : Ferari.java

```
package com.spring.model;
import
org.springframework.stereotype.Component;

@Component("Ferari")
public class Ferari implements Car{

  public void getCarName() {
      System.out.println("This is Ferari");
  }
}
```

File :Appmain.java

```
AbstractApplicationContext context = new
AnnotationConfigApplicationContext(
              AppConfig.class);

      // Byname Autowiring
      Application application = (Application)
context.getBean("application");
      System.out.println("Application Details :
" + application);

      // ByType Autowiring
      Employee employee = (Employee)
context.getBean("employee");
      System.out.println("Employee Details : "
+ employee);

      // By Constructor Autowiring
      Performer performer = (Performer)
context.getBean("performer");
      System.out.println("Performer Details : "
+ performer);
      // Setter Autowiring
```

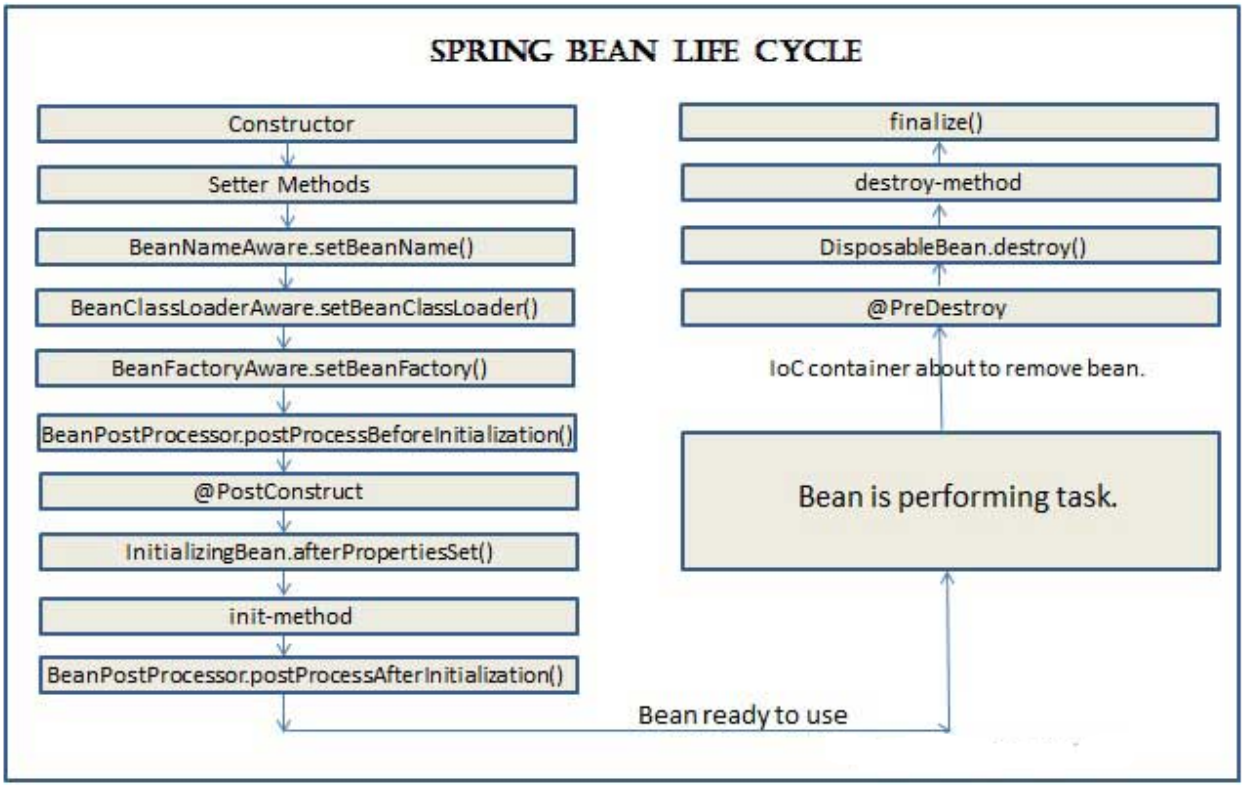
```
      Driver driver = (Driver)
context.getBean("driver");
      System.out.println("Driver Details : " +
driver);
      Bond bond = (Bond)
context.getBean("bond");
      bond.showCar();
```

Bean Life cycle

In spring bean life cycle, initialization and destruction callbacks are involved. Different spring bean aware classes are also called during bean life cycle. Once dependency injection is completed, initialization callback methods execute. Their purposes are to check the values that have been set in bean properties, perform any custom initialization or provide a wrapper on original bean etc. Once the initialization callbacks are completed, bean is ready to be used. When IoC container is about to remove bean, destruction callback methods execute. Their purposes are to release the resources held by bean or to perform any other finalization tasks. When more than one initialization and destructions callback methods have been implemented by bean, then those methods execute in certain order. Here on this page we will discuss spring bean life cycle step by step. We will discuss the order of execution of initialization and destruction callbacks as well as spring bean aware classes.

Spring Bean Life Cycle Diagram and Steps

Find the spring bean life cycle diagram. Here are showing the steps involved in spring bean life cycle.



A bean life cycle includes the following steps.

1) Within IoC container, a spring bean is created using class constructor.
2) Now the dependency injection is performed using setter method.
3) Once the dependency injection is completed, BeanNameAware.setBeanName() is called. It sets he name of bean in the bean factory that created this bean.
4) ow < code>BeanClassLoaderAware.setBeanClassLoader() is called that supplies the bean class loader to a bean instance.
5) Now < code>BeanFactoryAware.setBeanFactory() is called that provides the owning factory to a bean instance.
6) Now the IoC container calls BeanPostProcessor.postProcessBeforeInitialization on the bean. Using this method a wrapper can be applied on original bean.
7) Now the method annotated with @PostConstruct is called.
8) After @PostConstruct, the method InitializingBean.afterPropertiesSet() is called.
9) Now the method specified by init-method attribute of bean in XML configuration is called.

10) And then BeanPostProcessor.postProcessAfterInitialization() is called. It can also be used to apply wrapper on original bean.

11) Now the bean instance is ready to be used. Perform the task using the bean.

12) Now when the ApplicationContext shuts down such as by using registerShutdownHook() then the method annotated with @PreDestroy is called.

13) After that DisposableBean.destroy() method is called on the bean.

14) Now the method specified by destroy-method attribute of bean in XML configuration is called.

15) Before garbage collection, finalize() method of Object is called.

Spring Bean Life Cycle Order

Here we will provide a demo in which we will use all initialization and destructions callbacks and bean aware to check their order of execution in bean life cycle. We will use XML configuration.

File : Book.java

Spring Bean Life Cycle Order

Here we will provide a demo in which we will use all initialization and destructions callbacks and bean aware to check their order of execution in bean life cycle. We will use XML configuration.

File : Book.java

```
package com.spring.model;
import org.springframework.beans.BeansException;
import
org.springframework.beans.factory.BeanClassLoade
rAware;
import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.BeanFactoryAwa
re;
```

```
import
org.springframework.beans.factory.BeanNameAware;
import
org.springframework.beans.factory.DisposableBean
;
import
org.springframework.beans.factory.InitializingBe
an;

public class Book implements  InitializingBean,
DisposableBean, BeanFactoryAware, BeanNameAware,
BeanClassLoaderAware{
   private String bookName;
   private Book() {
       System.out.println("---inside
constructor---");
   }
```

```
@Override
   public void setBeanClassLoader(ClassLoader
classLoader) {
          System.out.println("---
BeanClassLoaderAware.setBeanClassLoader---");
   }
   @Override
   public void setBeanName(String name) {
          System.out.println("---
BeanNameAware.setBeanName---");
   }
   public void myPostConstruct() {
            System.out.println("---init-method--
-");
   }

   public void springPostConstruct() {
            System.out.println("---
@PostConstruct---");
```

```
    }
    @Override
    public void setBeanFactory(BeanFactory
beanFactory) throws BeansException {
        System.out.println("---
BeanFactoryAware.setBeanFactory---");
    }
    @Override
    public void afterPropertiesSet() throws
Exception {
        System.out.println("---
InitializingBean.afterPropertiesSet---");
    }
    public String getBookName() {
        return bookName;
    }
    public void setBookName(String bookName) {
        this.bookName = bookName;
        System.out.println("setBookName: Book
name has set.");
    }
    public void myPreDestroy() {
        System.out.println("---destroy-method---
");
    }

    public void springPreDestroy() {
        System.out.println("---@PreDestroy---");
    }
    @Override
            pblic void destroy() throws Exception
{
        System.out.println("---
DisposableBean.destroy---");
    }
    @Override
    protected void finalize() {
        System.out.println("---inside finalize---
");
```

```
    }
}
```

File :applicationContext.xml

```
   <beans ..>

<bean id="book" class="com.spring.model.Book">
<property name="bookName" value="Ramayan"/>

</bean>

</beans>
```

File :Appmain.java

```
  ApplicationContext applicationContext=new
ClassPathXmlApplicationContext("applicationConte
xt.xml");

        System.out.println("After application
context ");
        Book book =
(Book)applicationContext.getBean("book");

        System.out.println(book.getBookName());
```

Spring Bean Management

The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container.

When defining a <bean> in Spring, you have the option of declaring a scope for that bean. For example, To force Spring to

produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be prototype. Similar way if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be singleton.

The Spring Framework supports following five scopes, three of which are available only if you use a web-aware ApplicationContext.

Scope	Description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default).
prototype	This scopes a single bean definition to have any number of object instances.
request	This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
session	This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
global-session	This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

Note : In most cases, you may only deal with the Spring's core scope – singleton and prototype, and the default scope is singleton.

Example

using xml configuration

```
   <!-- A bean definition with singleton scope -
->
<bean id="..." class="..." scope="prototype">
```

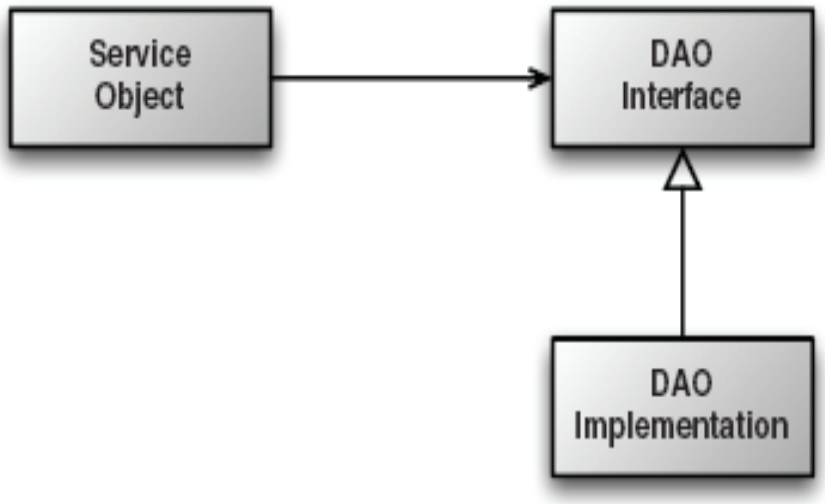
```
    <!-- collaborators and configuration for
this bean go here -->
</bean>
```

Using scope annotation

```
@Component("application")
@Scope("prototype")
public class Application {
-----
}
```

Chapter 3 DataBase Support in Spring Framework

Spring's Data Access Design

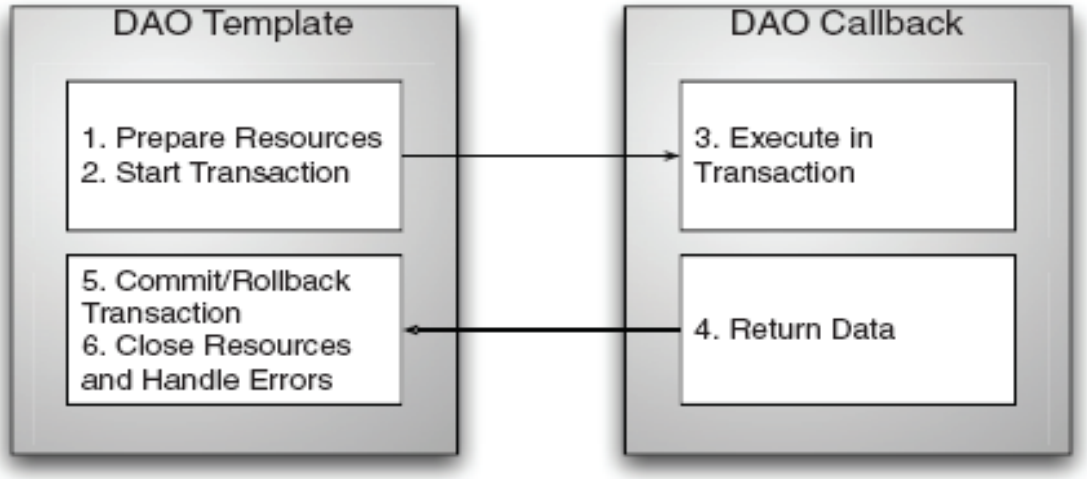


- Spring's data access support allows to develop applications following principle of coding to Interfaces.
- Data Access Object (DAO) exposes this functionality through an interface by which the rest of the application will access them.
- Service objects accessing DAO's through interfaces is better design approach because
  - DAO implementation can be swapped with minimum efforts.
  - No tight coupling to specific implementation of DAO.
  - Can be tested with any mock implementation of DAO.

Data Access Templates

- Spring separates the fixed and variable parts of the data access process into two classes
  - Templates
  - manage the fixed part of data access process such as opening connection and closing it at the end.
  - Callbacks

- handle custom data access code.



DAO Support Classes

- Spring provides DAO support classes on top of template-callback design
  - To be subclassed by developer's own DAO.
- DAO support classes provide convenient access to the appropriate template class.

Data Access Object (DAO)

The Data Access Object (DAO) support in Spring is aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a consistent way. This allows one to switch between the aforementioned persistence technologies fairly easily and it also allows one to code without worrying about catching exceptions that are specific to each technology.

- JdbcDaoSupport –

  It is superclass for JDBC data access objects. Requires a DataSource to be provided; in turn, this class provides a JdbcTemplate instance initialized from the supplied DataSource to subclasses.

- HibernateDaoSupport –

  It is superclass for Hibernate data access objects. Requires a SessionFactory to be provided; in turn, this class provides a HibernateTemplate instance initialized from the supplied SessionFactory to subclasses. Can alternatively be initialized directly via a HibernateTemplate, to reuse the latters settings like SessionFactory, flush mode, exception translator, and so forth.

- JdoDaoSupport –

  It is super class for JDO data access objects. Requires a PersistenceManagerFactory to be provided; in turn, this class provides a JdoTemplate instance initialized from the supplied PersistenceManagerFactory to subclasses.

- JpaDaoSupport –

  It is super class for JPA data access objects. Requires a EntityManagerFactory to be provided; in turn, this class provides a JpaTemplate

In Spring JDBC Framework there are many DAO support classes which help to reduce the configuration of JdbcTemplate, SimpleJdbcTemplate and NamedParamJdbcTemplate with dataSource object.

For JdbcTemplate:

org.springframework.jdbc.core.support.JdbcDaoSupport
For SimpleJdbcTemplate:

org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport
For NamedParamJdbcTemplate:

org.springframework.jdbc.core.namedparam.NamedParameterJdbcDaoSupport

Spring provides a convenient translation from technology-specific exceptions like SQLException to its own exception class hierarchy with the DataAccessException as the root exception. These

exceptions wrap the original exception so there is never any risk that one might lose any information as to what might have gone wrong.

In addition to JDBC exceptions, Spring can also wrap Hibernate-specific exceptions, converting them to a set of focused runtime exceptions (the same is true for JDO and JPA exceptions). This allows one to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without having annoying boilerplate catch-and-throw blocks and exception declarations in one's DAOs. (One can still trap and handle exceptions anywhere one needs to though.) As mentioned above, JDBC exceptions (including database-specific dialects) are also converted to the same hierarchy, meaning that one can perform some operations with JDBC within a consistent programming model.

Introduction to Spring Framework JDBC

Spring JdbcTemplate solve two main problems in the application. It solves redundant code problem from the application and another it solves poor exception handling of the data access code in the application. Without Spring JdbcTemplate in your application 20% of the code is only required for query a row, but 80% is boilerplate to handling exceptions and managing resources. If you use Spring JdbcTemplate then no need to worry about 80% boilerplate code to handling exceptions and managing resources. Spring JdbcTemplate in a Nutshell is responsible for following responsibilities.

Spring JdbcTemplate

- Spring abstracts away the boilerplate data access code behind the Template classes.
  - JdbcTemplate
  - provides simple access to a database through JDBC and simple indexed-parameter queries.
  - NamedParameterJdbcTemplate
  - performs queries where values are bound to named parameters in SQL, rather than indexed parameters.
  - SimpleJdbcTemplate
  - takes advantage of Java 5 features such as autoboxing,

generics and varargs

Spring JdbcTemplate implements the interface JdbcOperations. Spring template classes are based on Template method pattern.

Most of the common repetitive logic, like opening the database connection, preparing the Jdbc statements, handling and processing exception, handling transactions, closing the connection are done by JdbcTemplate object for us. During this process, JdbcTemplate calls the client code asking for application specific things like the sql statement along with parameters and for processing the result set (typically a call back on RowMapper).

Examples of Spring JdbcTemplate class usage:

Let us see how we can perform CRUD (Create, Read, Update and Delete) operation on database tables using SQL and jdbcTemplate object.

Querying for an integer:

```
String SQL = "select count(*) from Employee";
int rowCount = jdbcTemplateObject.queryForInt(
SQL );
```

Querying for a long:

```
String SQL = "select count(*) from Employee";
long rowCount = jdbcTemplateObject.queryForLong(
SQL );
```

A simple query using a bind variable:

```
String SQL = "select salary from Employee where
empid = ?";
long salary =
jdbcTemplateObject.queryForLong(SQL, new
Object[]{10000});
```

Querying for a String:

```
String SQL = "select name from Employee where
empid = ?";
```

```
String name =
jdbcTemplateObject.queryForObject(SQL, new
Object[]{10000}, String.class);
```

Querying and returning an object:

```
String SQL = "select * from Employee where empid
= ?";
Employee employee =
jdbcTemplateObject.queryForObject(SQL,
                  new Object[]{10000}, new
EmployeeMapper());

public class EmployeeMapper implements
RowMapper<Employee> {
   public Employee mapRow(ResultSet rs, int
rowNum) throws SQLException {
      Employee employee = new Employee();
      employee.setEmpid(rs.getInt("empid"));
      employee.setName(rs.getString("name"));
      employee.setAge(rs.getInt("age"));
      employee.setSalary(rs.getLong("salary"));
      return employee;
   }
}
```

Querying and returning multiple objects:

```
String SQL = "select * from Employee";
List<Employee> employee =
jdbcTemplateObject.query(SQL,
                         new EmployeeMapper());

public class EmployeeMapper implements
RowMapper<Employee> {
   public Employee mapRow(ResultSet rs, int
rowNum) throws SQLException {
      Employee employee = new Employee();
      employee.setEmpid(rs.getInt("empid"));
```

```
        employee.setName(rs.getString("name"));
        employee.setAge(rs.getInt("age"));
        employee.setSalary(rs.getLong("salary"));
        return employee;
    }
}
```

Inserting a row into the table:

```
String SQL = "insert into Employee (name, age)
values (?, ?)";
jdbcTemplateObject.update( SQL, new
Object[]{"Dinesh", 25} );
```

Updating a row into the table:

```
String SQL = "update Employee set name = ? where
empid = ?";
jdbcTemplateObject.update( SQL, new
Object[]{"Dinesh", 10000} );
```

Deletng a row from the table:

```
String SQL = "delete Employee where empid = ?";
jdbcTemplateObject.update( SQL, new
Object[]{10000} );
```

Spring JDBC using Annotation based configuration

- Step 1: (Create Employee table)
- Step 2: (Create Data Transfer Object)

Create an entity class as follows :Employee.java

```
package com.seed.app;
public class Employee {
   private int id;
   private String name;
   private int sal;
   //setter and getters
   //toString()
```

- Step 3: (Create configuration class) SpringJDBCConfiguration.java

```
package com.seed.app;
import javax.sql.DataSource;
import
org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configura
tion;
import
org.springframework.jdbc.core.JdbcTemplate;
import
org.springframework.jdbc.datasource.DriverManage
rDataSource;
@Configuration
public class SpringJDBCConfiguration {
   @Bean
   public DataSource dataSource() {
       DriverManagerDataSource dataSource = new
DriverManagerDataSource();
dataSource.setDriverClassName("oracle.jdbc.drive
r.OracleDriver");

   dataSource.setUrl("jdbc:oracle:thin:@Oracle12
.GURUKUL.COM:1521:oracle12");
       dataSource.setUsername("java");
       dataSource.setPassword("java");
```

```
        return dataSource;
    }
    @Bean
    public JdbcTemplate jdbcTemplate() {
        JdbcTemplate jdbcTemplate = new
JdbcTemplate();
        jdbcTemplate.setDataSource(dataSource());
        return jdbcTemplate;
    }
    @Bean
    public EmployeeDAO employeeDAO(){
        EmployeeDAOImpl empDao = new
EmployeeDAOImpl();
        empDao.setJdbcTemplate(jdbcTemplate());
        return empDao;
    }
}
```

The configuration class for obtaining the DataSource details and JDBCTemplate.

- **Step 4: (Create DAO classes)**

  Let's create the EmployeeDAO interface and implementation class for the db operations. We are using JDBCTemplate for the db operations here.

```
package com.seed.app;
import java.util.List;
public interface EmployeeDAO {
    public String getEmployeeName(int id);
    public int addEmployee(Employee emp);
    public int updateEmployee(Employee emp);
    public int deleteEmployee(Employee emp);
    public List<Employee> getAllEmployees();
}
```

```
package com.seed.app;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import java.util.List;
import
org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import
org.springframework.stereotype.Repository;
@Repository
public class EmployeeDAOImpl implements
EmployeeDAO {
   private JdbcTemplate jdbcTemplate;
   public void setJdbcTemplate(JdbcTemplate
jdbcTemplate){
        this.jdbcTemplate = jdbcTemplate;
   }
   public String getEmployeeName(int id) {
        String sql = "select empname from myemp
where empid = ?";
        String name =
jdbcTemplate.queryForObject(sql,new
Object[]{id},String.class);
        return name;
   }

@Override
   public int addEmployee(Employee emp) {
        // TODO Auto-generated method stub
       String sql="insert into myemp
values(?,?,?)";
     int i= jdbcTemplate.update(sql, new
Object[]{emp.getId(),emp.getName(),emp.getSal()}
);
       return i;
   }
@Override
   public int updateEmployee(Employee emp) {

        // TODO Auto-generated method stub
        String sql="update myemp set sal=? where
empid=?";
```

```
        int i=jdbcTemplate.update(sql,new
Object[]{emp.getSal(),emp.getId()});
        return i;
    }
    @Override
    public int  deleteEmployee(Employee emp) {

        // TODO Auto-generated method stub
        String sql="delete from myemp where
empid=?";
        int i=jdbcTemplate.update(sql,new
Object[]{emp.getId()});
        return i;
    }
    @Override
    public List<Employee> getAllEmployees() {
        // TODO Auto-generated method stub
        String sql="select * from myemp";
                return emp;
            }
        });
        return empList;
    }
}
```

```
List<Employee>
empList=jdbcTemplate.query(sql,new
RowMapper<Employee>(){
            @Override
            public Employee mapRow(ResultSet rs,
int rownum)
                    throws SQLException {
                // TODO Auto-generated method
stub
                Employee emp=new Employee();
                emp.setId(rs.getInt("empid"));

    emp.setName(rs.getString("empname"));
```

```
                emp.setSal(rs.getInt("sal"));

                return emp;
            }
        });
        return empList;
    }
}
```

An alternative to explicit configuration is to use component-scanning and annotation support for dependency injection. In this case you annotate the class with `@Repository` (which makes it a candidate for component-scanning) and annotate the `DataSource` setter method with `@Autowired`.

- **Step 5: (Main App class)**

```
package com.seed.app;
import java.sql.SQLException;
import java.util.Iterator;
import java.util.List;
import
org.springframework.context.annotation.Annotatio
nConfigApplicationContext;
public class App {
   public static void main(String[] args)
throws SQLException   {
AnnotationConfigApplicationContext
applicationContext = new
AnnotationConfigApplicationContext(
                SpringJDBCConfiguration.class);
        EmployeeDAO empDAO =
applicationContext.getBean(EmployeeDAO.class);
        String empName =
empDAO.getEmployeeName(400);
        System.out.println("Employee name is " +
empName);
         Employee e900=new Employee();
         e900.setId(900);
```

```
        e900.setName("Tushar");
        e900.setSal(6000);
        int i=empDAO.addEmployee(e900);
        System.out.println("No of rec inserted
are "+i);
        System.out.println(e900);
         Employee e231=new Employee();
        e231.setId(231);
        e231.setSal(45000);
        i=empDAO.updateEmployee(e231);
        System.out.println("No of rec updated
are "+i);
        Employee e3003=new Employee();
        e3003.setId(3003);
        i=empDAO.deleteEmployee(e3003);
        System.out.println("No of rec deleted
are "+i);
        System.out.println("Here is the list of
All employees");
        List<Employee>
eList=empDAO.getAllEmployees();
        Iterator<Employee>iterator =
eList.iterator();
        while(iterator.hasNext())
            System.out.println(iterator.next());
       applicationContext.close();
   }
}
```

Hibernate is a powerful technology for persisting data in any kind of Application. *Spring*, on the other hand is a dependency injection framework that supports **IOC**. The beauty of Spring is that it can integrates well with most of the prevailing popular technologies.

One of the problem with using **Hibernate** is that the client Application that accesses the database using *Hibernate Framework* has to depend on the **Hibernate** APIs like `Configuration`, `SessionFactory` and `Session`. These objects will continue to get scattered across the code throughout the Application. Moreover, the Application code has to manually maintain and manage these objects. In the case of **Spring**, the business objects can be highly configurable with the help of **IOC Container**. In simple words, the state of an object can be externalized from the Application code. It means that now it is possible to use the Hibernate objects as **Spring Beans** and they can enjoy all the facilities that Spring provides.

HibernateTransactionManager in Spring

HibernateTransactionManager handles transaction manager in Spring.

HibernateTransactionManager belongs to the package org.springframework.orm.hibernate3. The application that uses single hibernate session factory for database transaction, has good choice to use HibernateTransactionManager. HibernateTransactionManager can work with plain JDBC too. HibernateTransactionManager allows bulk update and bulk insert and ensures data integrity.

Example:

- **Step-1 Configure Hibernate (HibernateConfiguration.java)**

```
package com.seed.spring.configuration;
import java.util.Properties;
import javax.sql.DataSource;
import org.hibernate.SessionFactory;
import
org.springframework.beans.factory.annotation.Aut
owired;
```

```
import
org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Component
Scan;
import
org.springframework.context.annotation.Configura
tion;
import
org.springframework.context.annotation.PropertyS
ource;
import org.springframework.core.env.Environment;
import
org.springframework.jdbc.datasource.DriverManage
rDataSource;
import
org.springframework.orm.hibernate4.HibernateTran
sactionManager;
import
org.springframework.orm.hibernate4.LocalSessionF
actoryBean;
import
org.springframework.transaction.annotation.Enabl
eTransactionManagement;
@Configuration
@EnableTransactionManagement
@ComponentScan({ "com.seed.spring.configuration"
})
@PropertySource(value = {
"classpath:application.properties" })
public class HibernateConfiguration {
    @Autowired
    private Environment environment;
    @Bean
    public LocalSessionFactoryBean
sessionFactory() {
        LocalSessionFactoryBean sessionFactory =
new LocalSessionFactoryBean();
```

```
sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan(new
String[] { "com.seed.spring.model" });

sessionFactory.setHibernateProperties(hibernateP
roperties());
        return sessionFactory;     }
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new
DriverManagerDataSource();

dataSource.setDriverClassName(environment.getReq
uiredProperty("jdbc.driverClassName"));

dataSource.setUrl(environment.getRequiredPropert
y("jdbc.url"));

dataSource.setUsername(environment.getRequiredPr
operty("jdbc.username"));

dataSource.setPassword(environment.getRequiredPr
operty("jdbc.password"));
        return dataSource;
    }
       private Properties hibernateProperties()
{
        Properties properties = new
Properties();
        properties.put("hibernate.dialect",
environment.getRequiredProperty("hibernate.diale
ct"));
        properties.put("hibernate.show_sql",
environment.getRequiredProperty("hibernate.show_
sql"));
        properties.put("hibernate.format_sql",
environment.getRequiredProperty("hibernate.forma
t_sql"));
```

```
properties.put("hibernate.hbm2ddl.auto",environm
ent.getRequiredProperty("hibernate.hbm2ddl.auto"
));
         return properties;
    }

   @Bean
    @Autowired
    public HibernateTransactionManager
transactionManager(SessionFactory s) {
       HibernateTransactionManager txManager =
new HibernateTransactionManager();
       txManager.setSessionFactory(s);
       return txManager;
    }
}
```

@Configuration indicates that this class contains one or more bean methods annotated with @Bean producing beans manageable by spring container. In our case, this class represent hibernate configuration.

@ComponentScan is equivalent to context:component-scan base-package="..." in xml, providing with where to look for spring managed beans/classes.

@EnableTransactionManagement is equivalent to Spring's tx:* XML namespace, enabling Spring's annotation-driven transaction management capability.

@PropertySource is used to declare a set of properties(defined in a properties file in application classpath) in Spring run-time Environment, providing flexibility to have different values in different application environments.

Method sessionFactory() is creating a LocalSessionFactoryBean, which exactly mirrors the XML based configuration : We need a

dataSource and hibernate properties (same as hibernate.properties). Thanks to @PropertySource, we can externalize the real values in a .properties file, and use Spring's Environment to fetch the value corresponding to an item. Once the SessionFactory is created, it will be injected into Bean method transactionManager which may eventually provide transaction support for the sessions created by this sessionFactory.

/src/application.properties

- jdbc.driverClassName = oracle.jdbc.driver.OracleDriver
- jdbc.url = jdbc:oracle:thin:@Oracle12.GURUKUL.COM:1521:oracle12
- jdbc.username = java
- jdbc.password = java
- hibernate.dialect = org.hibernate.dialect.Oracle10gDialect
- hibernate.show_sql = true
- hibernate.format_sql = false
- hibernate.hbm2ddl.auto=create

- **Step 2: Configure Spring**

```
package com.seed.spring.configuration;
import
org.springframework.context.annotation.Component
Scan;
import
org.springframework.context.annotation.Configura
tion;

@Configuration
@ComponentScan(basePackages = "com.seed.spring")
public class AppConfig {

}
@ComponentScan which provides beans auto-
detection facility.
```

- **Step 2: Add DAO Layer**

```
package com.seed.spring.dao;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import
org.springframework.beans.factory.annotation.Aut
owired;

public abstract class AbstractDao {

    @Autowired
    private SessionFactory sessionFactory;

    protected Session getSession() {
        return
sessionFactory.getCurrentSession();
    }

    public void persist(Object entity) {
        getSession().persist(entity);
    }

    public void delete(Object entity) {
        getSession().delete(entity);
    }
}
```

Notice above, that SessionFactory we have created earlier in step 2, will be auto-wired here.This class serve as base class for database related operations.

```
File : com.seed.spring.dao. EmployeeDao
package com.seed.spring.dao;
import java.util.List;
import com.seed.spring.model.Employee;

public interface EmployeeDao {
```

```
    void saveEmployee(Employee employee);

    List<Employee> findAllEmployees();

    void deleteEmployeeById(int id);

    Employee findById(int id);

    void updateEmployee(Employee employee);
}
```

File : com.seed.spring.dao. EmployeeDaoImpl

package com.seed.spring.dao;

```
import java.util.List;

import org.hibernate.Criteria;
import org.hibernate.Query;
import org.hibernate.criterion.Restrictions;
import
org.springframework.stereotype.Repository;

import com.seed.spring.model.Employee;
```

```
@Repository("employeeDao")
public class EmployeeDaoImpl extends AbstractDao
implements EmployeeDao{

    public void saveEmployee(Employee employee) {
        persist(employee);
    }

    @SuppressWarnings("unchecked")
    public List<Employee> findAllEmployees() {
        Criteria criteria =
getSession().createCriteria(Employee.class);
        return (List<Employee>) criteria.list();
    }
```

```
    public void updateEmployee(Employee
employee){
        getSession().update(employee);
    }

    @Override
    public void deleteEmployeeById(int id) {
        // TODO Auto-generated method stub
        Query query =
getSession().createSQLQuery("delete from
Employee where id = :id");
        query.setInteger("id", id);
        query.executeUpdate();
    }

    public Employee findById(int id) {
        // TODO Auto-generated method stub
        Criteria criteria =
getSession().createCriteria(Employee.class);
        criteria.add(Restrictions.eq("id",id));
        return (Employee)
criteria.uniqueResult();
    }
}
```

- **Step4: Add Service Layer**

File : com.seed.spring.service. EmployeeService.java

```
package com.seed.spring.service;
import java.util.List;
import com.seed.spring.model.Employee;
public interface EmployeeService {

    void saveEmployee(Employee employee);

    List<Employee> findAllEmployees();
```

```
    void deleteEmployeeById(int id);

    Employee findById(int id);
    void updateEmployee(Employee employee);
}
File : com.seed.spring.service.
EmployeeServiceImpl.java
package com.seed.spring.service;
import java.util.List;
import
org.springframework.beans.factory.annotation.Aut
owired;
import org.springframework.stereotype.Service;
import
org.springframework.transaction.annotation.Trans
actional;

import com.seed.spring.dao.EmployeeDao;
import com.seed.spring.model.Employee;

@Service("employeeService")
@Transactional
public class EmployeeServiceImpl implements
EmployeeService{
```

```
    @Autowired
private EmployeeDao dao;

    public void saveEmployee(Employee employee) {
        dao.saveEmployee(employee);
    }

    public List<Employee> findAllEmployees() {
        return dao.findAllEmployees();
    }
```

```
    public void updateEmployee(Employee
employee){
        dao.updateEmployee(employee);
    }
```

```
    @Override
public void deleteEmployeeById(int id) {
        // TODO Auto-generated method stub
        dao.deleteEmployeeById(id);
    }
```

```
    @Override
public Employee findById(int id) {
        // TODO Auto-generated method stub
        return dao.findById(id);
    }
}
@Transactional which starts a transaction on
each method start, and commits it on each method
exit ( or rollback if method was failed due to
an error). Note that since the transaction are
on method scope, and inside method we are using
DAO, DAO method will be executed within same
transaction.
```

- **Step 5: Create Domain Entity Class(POJO)**

```
package com.seed.spring.model;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;
@Entity
@Table(name="HibSpringEmployee",

uniqueConstraints={@UniqueConstraint(columnNames
={"ID"})})
public class Employee {

   @Id

   @Column(name="ID", nullable=false,
unique=true, length=11)
   private int id;

   @Column(name="NAME", length=20,
nullable=true)
   private String name;

   @Override
   public String toString() {
       return "Employee [id=" + id + ", name=" +
name + ", role=" + role + ", insertTime=" +
insertTime + "]";
   }
   @Column(name="ROLE", length=20,
nullable=true)
   private String role;

   @Column(name="insert_time", nullable=true)
   private Date insertTime;
```

```
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getRole() {
        return role;
    }
    public void setRole(String role) {
        this.role = role;
    }
    public Date getInsertTime() {
        return insertTime;
    }
    public void setInsertTime(Date insertTime) {
        this.insertTime = insertTime;
    }
}
```

This is a standard Entity class annotated with JPA annotations `@Entity`, `@Table`, `@Column` along with hibernate specific annotation.

- **Step 6: Create Main to run as Java Application**

```
package com.seed.spring;
import java.math.BigDecimal;
import java.util.Date;
import java.util.List;
import
org.springframework.context.annotation.Annotatio
nConfigApplicationContext;
```

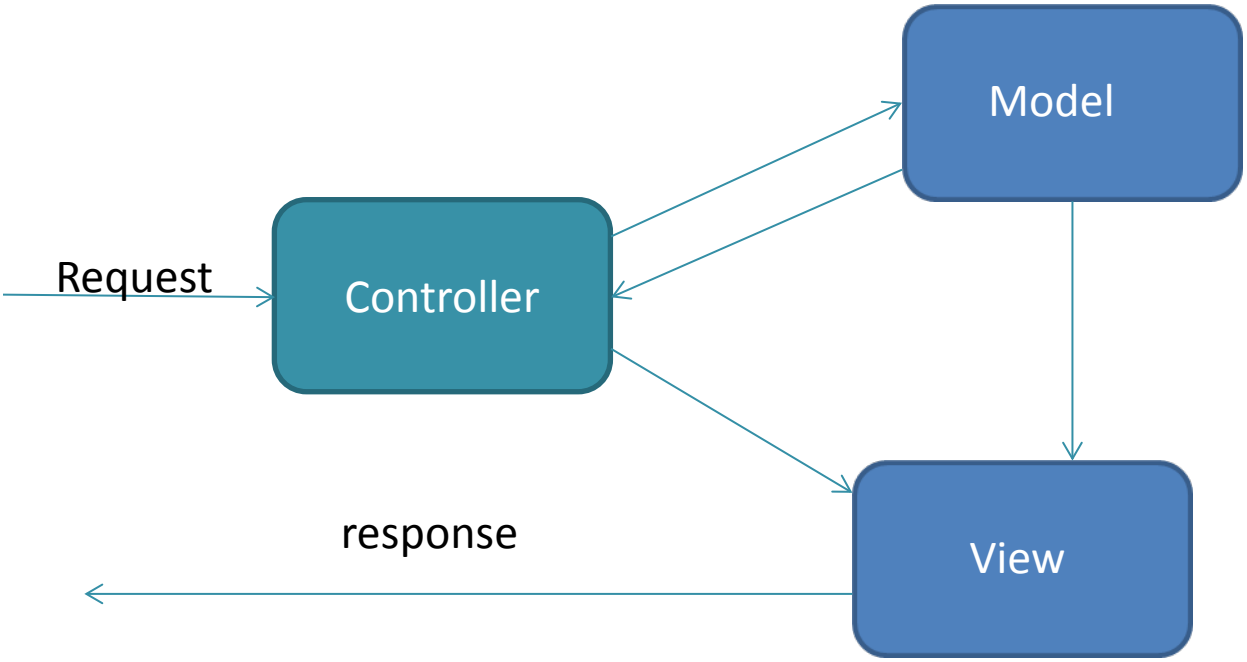
```
import
org.springframework.context.support.AbstractAppl
icationContext;

import com.seed.spring.configuration.AppConfig;
import com.seed.spring.model.Employee;
import com.seed.spring.service.EmployeeService;

public class AppMain {

   public static void main(String args[]) {
       AbstractApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.cla
ss);

       EmployeeService service =
(EmployeeService)
context.getBean("employeeService");

       /*
        * Create Employee1
        */
       Employee e2 = new Employee();
       e2.setId(124);
       e2.setName("Aniket");
       e2.setRole("Clerk");

       Employee e1 = new Employee();
       e1.setId(121);
       e1.setName("Manasi");
       e1.setRole("MD");

       Employee e = new Employee();
       e.setId(100);
       e.setName("Sam");
       e.setRole("Manager");
       e.setInsertTime(new Date());
       /*
        * Persist both Employees
```

```
         */
        service.saveEmployee(e1);
        service.saveEmployee(e2);
        service.saveEmployee(e);

        /*
         * Get all employees list from database
         */
        List<Employee> employees =
service.findAllEmployees();
        for (Employee emp : employees) {
            System.out.println(emp);
        }

        /*
         * delete an employee
         */
        service.deleteEmployeeById(124);

        /*
         * update an employee
         */

        Employee employee =
service.findById(100);
        employee.setName("jayshree");
        service.updateEmployee(employee);

        /*
         * Get all employees list from database
         */
        List<Employee> employeeList =
service.findAllEmployees();
        for (Employee emp : employeeList) {
            System.out.println(emp);
        }

        context.close();
```

```
        }
}
```

Chapter 4 Spring MVC

Intoduction to MVC



MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

- Model - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
- View - View represents the visualization of the data that model contains.
- Controller - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

Front Controller MVC

- The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler.
- This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to

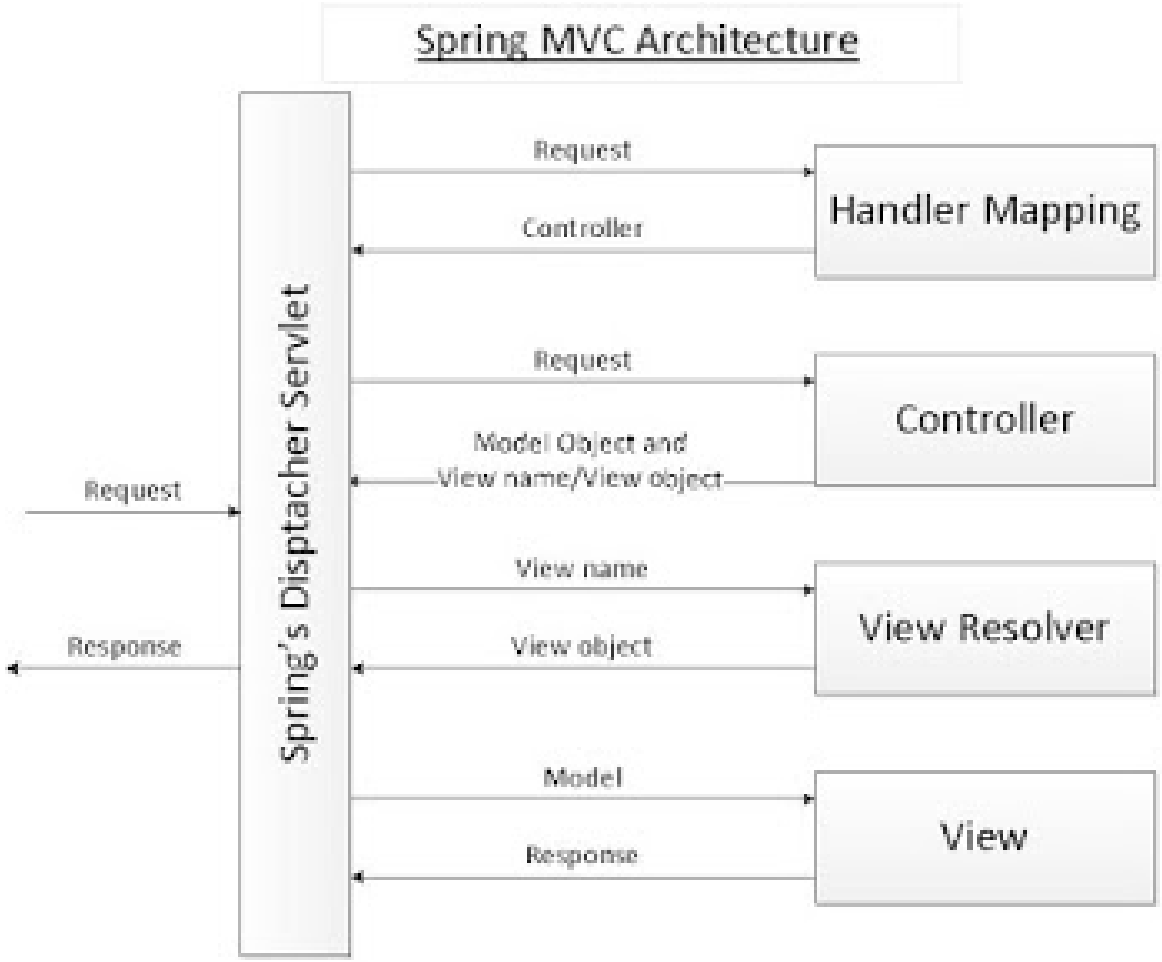
corresponding handlers

- Following are the entities of this type of design pattern.
  - **Front Controller** - Single handler for all kinds of requests coming to the application (either web based/ desktop based).
  - **Dispatcher** - Front Controller may use a dispatcher object which can dispatch the request to corresponding specific handler.
  - **View** - Views are the object for which the requests are made.

Front controller Responsibilities

- Initialize the framework to cater the request.
- Load the map of all URLs and the component responsible to handle the request.
- Prepare the map for views.

Spring4 –MVC



Spring MVC components

- FrontController –DispatcherServlet
- Controllers
- RequestMapping
- ViewResolver

Spring4 MVC-FrontController DispatcherServlet

- The DispatcherServlet of Spring Web MVC framework is an implementation of FrontController and is a Java Servlet component.
- DispathcherServlet is the FrontController that receives all incoming HTTP client request for the spring Web mvc application.
- It is responsible for initializing spring web mvc framework for our application.

- DispatcherServlet also required to be configured in our web-application like any other Servlet i.e., web application deployment descriptor(web.xml) or through Java configuration.

```
package com.jayshree.springmvc.configuration;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.apache.catalina.Container;
import
org.springframework.web.WebApplicationInitialize
r;
import
org.springframework.web.context.support.Annotati
onConfigWebApplicationContext;
import
org.springframework.web.servlet.DispatcherServle
t;

public class HelloWorldInitalizer implements
WebApplicationInitializer {

   public void onStartup(ServletContext
container) throws ServletException {
       AnnotationConfigWebApplicationContext
ctx=new AnnotationConfigWebApplicationContext();

   ctx.register(HelloWorldConfiguration.class);
       ctx.setServletContext(container);
       ServletRegistration.Dynamic servlet=
container.addServlet("dispatcher",new
DispatcherServlet(ctx));
       servlet.setLoadOnStartup(1);
       servlet.addMapping("/");
   }

}
```

HelloWorldInitalizer is a class that implements WebApplicationInitializer interface. We hook up to onStartup() method, to add DispatcherServlet to ServletContext.

Several things happen here:

- AnnotationConfigWebApplicationContext is created.
  It's WebApplicationContext implementation that looks for Spring configuration in classes annotated with @Configuration annotation. setConfigLocation() method gets hint in which package(s) to look for them.
- DispatcherServlet is created and initialized with WebApplicationContext we have created, and it's mapped to "/*" URLs and set to eagerly load on application startup.

Configuration Class

The main HelloWorldConfiguration class doesn't do anything but hits Spring on where to look for its components through @ComponentScan annotation.

```
package com.jayshree.springmvc.configuration;
import
org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Component
Scan;
import
org.springframework.context.annotation.Configura
tion;
import
org.springframework.web.servlet.ViewResolver;
import
org.springframework.web.servlet.config.annotatio
n.EnableWebMvc;
import
org.springframework.web.servlet.view.InternalRes
ourceViewResolver;
```

```
import
org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages="com.jayshree.spring
mvc")
public class HelloWorldConfiguration {
    }
```

Controller Class

Controllers are annotated with @Controller. It is found by Spring because of @ComponentScan annotation in HelloWorldConfiguration.

```
package com.jayshree.springmvc.controller;
import
org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import
org.springframework.web.bind.annotation.RequestM
apping;
import
org.springframework.web.bind.annotation.RequestM
ethod;

@Controller
@RequestMapping("/")
public class HelloWorldController {

    @RequestMapping(method=RequestMethod.GET)
    public String sayHello(ModelMap m)
    {
        m.addAttribute("greeting", "HelloWorld
from Spring MVC 4");
        return "welcome";
    }
    @RequestMapping(value="/helloAgain",method=Re
questMethod.GET)
    public String sayHelloAgain(ModelMap m)
```

```
    {

        m.addAttribute("greeting", "Hello world
again ,from Spring MVC 4");
        return "welcome";
    }
}
```

Controller Class Annotation

ANNOTAtion	USEUsdf	DESCRIPTION
@Controller	Type	Stereotypes a component as a Spring MVC controller.
@InitBinder	Method	Annotates a method that customizes data binding.
@ModelAttribute	Parameter, Method	When applied to a method, used to preload the model with the value returned from the method. When applied to a parameter, binds a model attribute to the parameter. table
@RequestMapping	Method, Type	Maps a URL pattern and/or HTTP method to a method or controller type.

@RequestParam	Parameter	Binds a request parameter to a method parameter.
@SessionAttributes	Type	Specifies that a model attribute should be stored in the session.

View Resolver

- All MVC frameworks provides a way to working with the view.
- Spring does that via the view resolvers, which enable you to render models in the browser without tying the implementation to a specific view technology.
- The *ViewResolver* maps view names to actual views.
- Spring framework comes with quite a few view resolvers
  e.g. *InternalResourceViewResolver*, *XmlViewResolver*, *ResourceBundleViewResolver* and a few others.

Add an InternalResourceViewResolver

This *ViewResolver* allows us to set properties such as prefix or suffix to the view name to generate the final view page URL package com.jayshree.springmvc.configuration;

```
import
org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Component
Scan;
import
org.springframework.context.annotation.Configura
tion;
import
org.springframework.web.servlet.ViewResolver;
import
org.springframework.web.servlet.config.annotatio
n.EnableWebMvc;
```

```
import
org.springframework.web.servlet.view.InternalRes
ourceViewResolver;
import
org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages="com.jayshree.spring
mvc")
public class HelloWorldConfiguration {


    @Bean
    public ViewResolver viewResolver()
    {
        InternalResourceViewResolver
viewResolver=new InternalResourceViewResolver();

    viewResolver.setViewClass(JstlView.class);
        viewResolver.setPrefix("/WEB-
INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

In this case only need a simple *jsp* page, placed in the */WEB-INF/view* folder as defined in the configuration.

❑❑❑

Hibernate

Compiled by SEED Infotech Ltd.

Index

1. Introduction to ORM (Object relation mapping)

Persistence, in computer science, is an adjective describing data that outlives the process that created it. Java persistence could be defined as storing anything to any level of persistence using the Java programming language. There are many ways to make data persist in Java, including (JDBC, serialization, file IO, relational databases. However, the majority of data is persisted in databases, specifically relational databases.

1) **Java Database Connectivity** (**JDBC**) is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is Java based data access technology and used for Java database connectivity. It is part of the Java Standard Edition platform, from Oracle Corporation. It provides methods to query and update data in a database, and is oriented towards relational databases. The API provides a mechanism for dynamically loading the correct Java packages and registering them with the JDBC Driver Manager. The Driver Manager is used as a connection factory for creating JDBC connections.

   JDBC connections support creating and executing statements. These may be update statements such as SQL's CREATE, INSERT, UPDATE and DELETE, or they may be query statements such as SELECT. Additionally, stored procedures may be invoked through a JDBC connection. JDBC represents statements using one of the following classes:

   - Statement – the statement is sent to the database server each and every time.
   - PreparedStatement – the statement is cached and then the execution path is pre-determined on the database server allowing it to be executed multiple times in an efficient manner.
   - CallableStatement – used for executing stored procedures on the database.

   Update statements such as INSERT, UPDATE and DELETE

return an update count that indicates how many rows were affected in the database. These statements do not return any other information.

Query statements return a JDBC row result set. The row result set is used to walk over the result set. Individual columns in a row are retrieved either by name or by column number. There may be any number of rows in the result set. The row result set has metadata that describes the names of the columns and their types.

There is an extension to the basic JDBC API in the `javax.sql`.

JDBC connections are often managed via a connection pool rather than obtained directly from the driver.

2) **serialization** is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection link) and reconstructed later (possibly in a different computer environment). When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects does not include any of their associated methods with which they were previously linked.

   This process of serializing an object is also called marshalling an object.[2] The opposite operation, extracting a data structure from a series of bytes, is **deserialization** (which is also called **unmarshalling**).

   Java provides automatic serialization which requires that the object be marked by implementing the `java.io.Serializable` interface. Implementing the interface marks the class as "okay to serialize", and Java then handles serialization internally. There are no serialization methods defined on the `Serializable` interface, but a serializable class can optionally define methods with certain special names and signatures that if defined, will be called as part of the serialization/deserialization process. The language also allows

the developer to override the serialization process more thoroughly by implementing another interface, the Externalizable interface, which includes two special methods that are used to save and restore the object's state. There are three primary reasons why objects are not serializable by default and must implement the Serializable interface to access Java's serialization mechanism. Firstly, not all objects capture useful semantics in a serialized state.

3) A **relational database management system** (**RDBMS**) is a database management system (DBMS) that is based on the relational model. This model organizes data into one or more tables (or "relations") of columns and rows, with a unique key identifying each row. Rows are also called records or tuples. Columns are also called attributes. Generally, each table/relation represents one "entity type" (such as customer or product). The rows represent instances of that type of entity (such as "Lee" or "chair") and the columns representing values attributed to that instance (such as address or price).

   RDBMSs have been a common choice for the storage of information in new databases used for financial records, manufacturing and logistical information, personnel data, and other applications since the 1980s. Relational databases have often replaced legacy hierarchical databases and network databases because they are easier to understand and use. However, relational databases have received unsuccessful challenge attempts by object database management systems in the 1980s.

   What is the need of the ORM Object realational mapping)?

   The Object Oriented (Domain) model use classes whereas the relational database use tables. This creates a gap (The Impedance Mismatch). Due to the difference between the two different models, getting the data and associations from objects into relational table structure and vice versa requires a lot of tedious programming.

   Loading and storing objects using a tabular relational database exposes us to 5 mismatch problems.

1) granularity
2) Inheritance
3) identity
4) associations
5) data navigation

1) Granularity

Granularity is the extent to which a system could be broken down into small parts. Lets take an example of Person details, we could broke down person details into two classes one is Person and another is Address for code reusability and code maintainability purpose. But assume that to store Person details in database there is only one table called Person.

Sometimes you will have an object model which has more classes than the number of corresponding tables in the database (we says the object model is more granular than the relational model). This is Granularity Mismatch between Object Model and Relational Model.

2) Inheritance or Subtypes

Inheritance is a natural concept in object-oriented programming languages. However, RDBMSs do not define anything similar on the whole (yes some databases do have subtype support but it is completely non-standardized). This is Inheritance Mismatch between Object Model and Relational Model.

3) Identity

A RDBMS defines exactly one notion of ‘sameness’: the primary key.

Java objects define two different notions of sameness:

- Object identity (roughly equivalent to memory location, checked with a==b)
- Equality as determined by the implementation of the equals() method (also called equality by value)

On the other hand, the identity of a database row is expressed as the primary key value. It's common for several non identical objects to simultaneously represent the same row of the database, for example, in concurrently running application threads. Furthermore, some subtle difficulties are involved in implementing equals() correctly for a persistent class.

4) Associations

Object-oriented languages represent associations using object references; but in the relational world, an association is represented as a foreign key column, with copies of key values (and a constraint to guarantee integrity). There are substantial differences between the two representations.

Object references are inherently directional; the association is from one object to the other. They're pointers. If an association between objects should be navigable in both directions, you must define the association twice, once in each of the associated classes. You've already seen this in the domain model classes:

```
Public class User{
Private Set billingDetails;
….
}
```

5) Data Navigation

There is a fundamental difference in the way you access data in Java and in a relational database. In Java, when you access a user's billing information, you call aUser.getBillingDetails ().getAccountNumber() or something similar. This is the most natural way to access object-oriented data, and it's often described as walking the object network.

You navigate from one object to another, following pointers between instances. Unfortunately, this isn't an efficient way to retrieve data from an SQL database.This is not an efficient way of retrieving data from a relational database.

You typically want to minimize the number of SQL queries and thus load several entities via JOINs and select the targeted entities before you start walking the object network.

The solution for above problems is use ORM tool

An **Object Relational Mapping** Tool provides a simple API for storing and retrieving Java objects directly to and from the relational database. ORM is technique that allows an application written in an object oriented language to deal with information as objects, rather than using database specific concepts such as Rows, Columns and Tables which facilitated by Object/Relational Mapper.

Advantages of ORM tools

- Better System Architecture
- Reduce Coding Time
- Caching And Transactions

- **Better System Architecture**

  A good ORM tool designed by very experienced software architects will implement effective design patterns that almost force you to use good programming practices in an application. This can help support a clean separation of concerns and independent development that allows parallel, simultaneous development of application layers. If you create a class library ORM-generated data access code, you can easily reuse the data objects in a variety of applications. This way, code reusability will increase. If use in this way there will be a cleaner separation of data access layer.

- **Reduce Coding Time**

  Most of the time database access code spec is simple insert, update or delete. These are SQL statements sometimes are quite tedious to code, if you use JDBC it does not allow you to store objects directly to the database, you must convert the objects to a relational format. ORM tool helps here, by generating them on fly and there by saves a lot of time.

- **Caching and Transactions**

  Most ORM tools such as hibernate come with features such as Caching and Transactions. These features, if chosen to hand code are not so easy to implement.

Introduction to Hibernate Framework

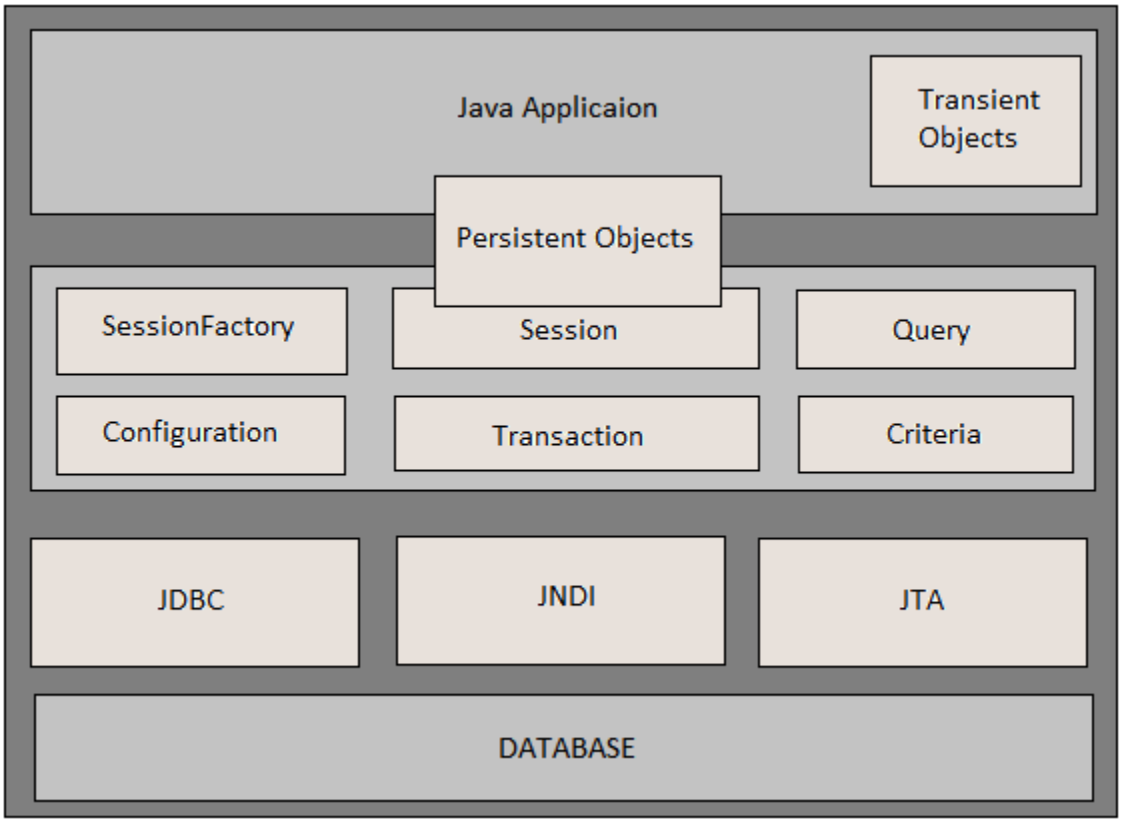
Hibernate is an Object-relational mapping (ORM) tool. Object-relational mapping or ORM is a programming method for mapping the objects to the relational model where entities/classes are mapped to tables, instances are mapped to rows and attributes of instances are mapped to columns of table.

Hibernate is a persistence framework which is used to persist data from Java environment to database. Persistence is a process of storing the data to some permanent medium and retrieving it back at any point of time even after the application that had created the data ended.

Hibernate is a non-invasive framework, means it won't forces the programmers to extend/implement any class/interface, and in hibernate we have all POJO classes so its light weight.

Hibernate can runs with in or without server, it will suitable for all types of java applications (stand alone or desktop or any servlets)

Hibernate is purely for persistence (to store/retrieve data from Database).



The above diagram shows a comprehensive architecture of Hibernate. In order to persist data to a database, Hibernate create an instance of entity class (Java class mapped with database table). This object is called Transient object as they are not yet associated with the session or not yet persisted to a database. To persist the object to database, the instance of SessionFactory interface is created. SessionFactory is a singleton instance which implements Factory design pattern. SessionFactory loads hibernate.cfg.xml file (Hibernate configuration file. More details in following section) and with the help of TransactionFactory and ConnectionProvider implements all the configuration settings on a database.

Each database connection in Hibernate is created by creating an instance of Session interface. Session represents a single connection with database. Session objects are created from SessionFactory object.

Hibernate also provides built-in Transaction APIs which abstracts away the application from underlying JDBC or JTA transaction. Each transaction represents a single atomic unit of work. One Session can span through multiple transactions.

Hibernate functionality/flow/usage can be described as follows:

- On Application startup, hibernate reads it configuration file (hibernate.cfg.xml or hibernate.properties) which contains information required to make the connection with underlying database and mapping information. Based on this information, hibernate creates Configuration Object , which in turns creates SessionFactory which acts as singleton for the whole application.
- Hibernate creates instances of entity classes.Entity classes are java classes which are mapped to the database table using metadata(XML/Annotaitons).These instances are called transient objects as they are not yet persisted in database.
- To persist an object, application ask for a Session from SessionFactory which is a factory for Session.Session represent a physical database connection.
- Application then starts the transaction to make the unit of work atomic, & uses Session API's to finally persist the entity instance in database.Once the entity instance persisted in database, it's known as persistent object as it represent a row in database table.Application then closes/commits the transaction followed by session close.
- Once the session gets closed , the entity instance becomes detatched which means it still contains data but no more attached to the database table & no more under the management of Hibernate. Detatched objects can again become persistent when associated with a new Session, or can be garbage collected once no more used.

Below is the brief description of commonly used core API's in a typical application persistence with Hibernate.

Configuration (org.hibernate.cfg.Configuration)

The Configuration class is the initial entry point of a Hibernate application. An instance of Configuration class represents application specific configuration that provides persistence services. This class provides convinient ways to import the Hibernate Configuration File and the necessary Mapping Files into an application which can be later used by SessionFactory to create Session objects.Usually only one instance of a

Configuration class will be maintained by an application.

SessionFactory (org.hibernate.SessionFactory)

A thread-safe, immutable cache of compiled mappings for a single database. A factory for org.hibernate.Session instances. A client of org.hibernate.connection.ConnectionProvider. Optionally maintains a second level cache of data that is reusable between transactions at a process or cluster level.

Session (org.hibernate.Session)

A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC java.sql.Connection. Factory for org.hibernate.Transaction. Maintains a first level cache of persistent the application's persistent objects and collections; this cache is used when navigating the object graph or looking up objects by identifier.

Transaction (org.hibernate.Transaction)

It's a single-thread object used by the application to define units of work. A transaction is associated with a Session. Transactions abstract application code from underlying transaction implementations(JTA/JDBC), allowing the application to control transaction boundaries via a consistent API. It's an Optional API and application may choose not to use it.

ConnectionProvider (org.hibernate.connection.ConnectionProvider)

A factory for, and pool of, JDBC connections. It abstracts the application from underlying javax.sql.DataSource or java.sql.DriverManager. It is not exposed to application, but it can be extended and/or implemented by the developer.

TransactionFactory (org.hibernate.TransactionFactory)

(Optional) A factory for org.hibernate.Transaction instances. It is not exposed to the application, but it can be extended and/or

implemented by the developer.

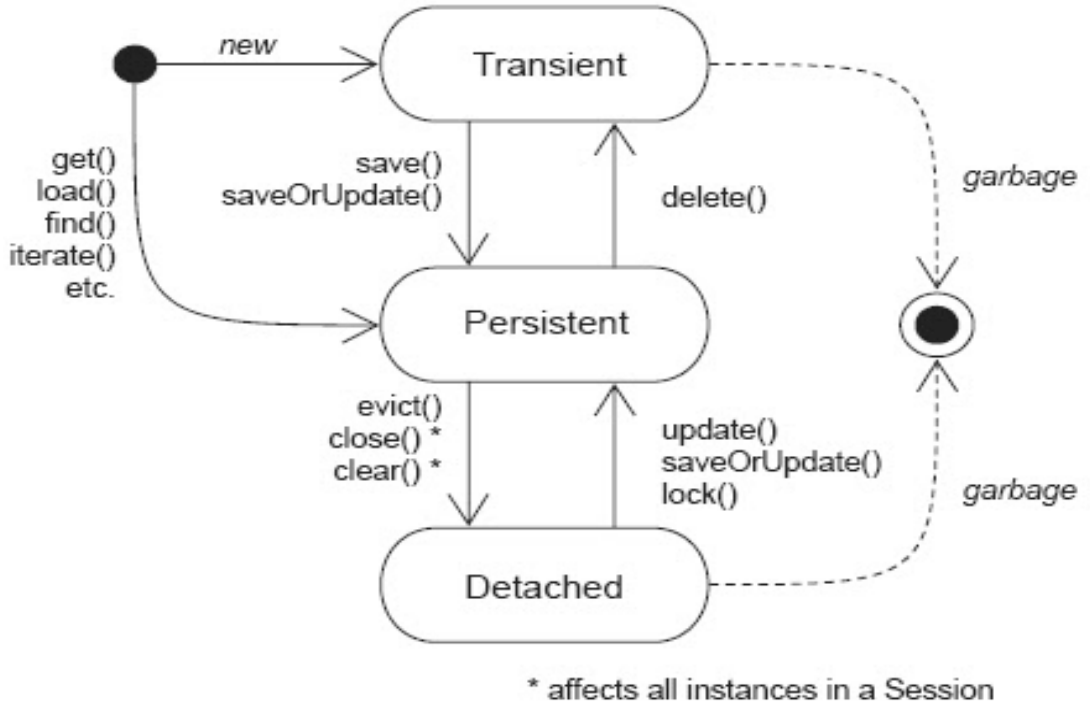
Query (org.hibernate.Query)

A single-thread object used to perform query on underlying database. A Session is a factory for Query. Both HQL(Hibernate Query Language) & SQL can be used with Query object.

Criteria (org.hibernate.Criteria)

It is an alternative to HQL , very useful for the search query involving multiple conditions.

Object Classification based on States



Short-lived, single threaded objects containing persistent state and business function. These can be ordinary JavaBeans/POJOs.Such objects are categorized in Hibernate technology based on the state of the object. They are,

- Persistent Object
- Transient Object
- Detached Object

Persistent Object:

A persistent object is one which is associated with the current Session and has its state values synchronized with the database.

Transient Object:

A transient object doesn't have any association with the current Session, but can be made persistent at a later time.

Detached Object:

A detached object was having an association at a previous point of time with the Session interface, but now is made to detach (taken-off) from the current Session.

Advantages of using Hibernate Framework:

1) Open Source
2) Vendor Independent
3) Less lines of code

Hiberrnate Mappings

Hibernate Configuration

Hibernate configuration is managed by an instance of org.hibernate.cfg.Configuration. An instance of org.hibernate.cfg.Configuration represents an entire set of mappings of an application's Java types to an SQL database. The org.hibernate.cfg.Configuration is used to build an immutable org.hibernate.SessionFactory.

Hibernate requires to know in advance where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called hibernate.properties, or as an XML file named hibernate.cfg.xml or Annotations . There are several approaches to provide this information to Hibernate. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application's classpath.

1) Using only hibernate.cfg.xml

Following is the list of important properties you would require to configure for a databases in a standalone situation:

S.N.	Properties and Description
1	hibernate.dialect This property makes Hibernate generate the appropriate SQL for the chosen database.
2	hibernate.connection.driver_class The JDBC driver class.
3	hibernate.connection.url The JDBC URL to the database instance.
4	hibernate.connection.username

	The database username.
5	hibernate.connection.password The database password.
6	hibernate.connection.pool_size Limits the number of connections waiting in the Hibernate database connection pool.
7	hibernate.connection.autocommit Allows autocommit mode to be used for the JDBC connection.

examples

hibernate.cfg.xml is a standard XML file containing database connection information along with mapping information.This file needs to be found on root of application classpath. Shown below is a sample hibernate.cfg.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
       "-//Hibernate/Hibernate Configuration DTD
3.0//EN"
       "http://hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
       <property
name="hibernate.connection.driver_class">oracle.
jdbc.driver.OracleDriver</property>
       <property
name="hibernate.connection.url">jdbc:oracle:thin
:@172.16.10.2:1521:orcl</property>
       <property
name="hibernate.connection.username">java</prope
rty>
```

```
        <property
name="hibernate.connection.password">java</prope
rty>
        <property
name="hibernate.connection.pool_size">10</proper
ty>
        <property
name="hibernate.current_session_context_class">t
hread</property>
        <property
name="hibernate.show_sql">true</property>
        <property
name="hibernate.hbm2ddl.auto">create</property>
        <property
name="hibernate.dialect">org.hibernate.dialect.O
racle10gDialect</property>
            <mapping resource="employee.hbm.xml"
/>
    </session-factory>
</hibernate-configuration>
```

Configuring the Persistence class

Usually <persistence-object>.hbm.xml file is used to configure the Persistence object. Hibernate XML mapping file contains the mapping relationship between Java class and database table.This file is declared in the Hibernate configuration file “hibernate.cfg.xml”.

For example “Employee.hbm.xml”

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-
mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.seed.Employee"
table="employee">
<meta attribute="class-description">
```

```
          This class contains the employee
detail.
</meta>
<id name="empId" type="int" column="Emp_id">
<generator class="native"/>
</id>
<property name="empName" type="java.lang.String"
   column="Emp_name" not-null="true" length="50"
/>
<property name="empSal" type="java.lang.Double"
  column="Emp_sal" not-null="true" />
</class>
</hibernate-mapping>
```

- The mapping document is an XML document having **<hibernate-mapping>** as the root element which contains all the <class> elements.
- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.
- The **<meta>** element is optional element and can be used to create the class description.
- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.
- The **<generator>** element within the id element is used to automatically generate the primary key values. Set the **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity ,sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

The **<property>** element is used to map a Java class property to

a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate

Test Project:

Following is the code listing of a Java class called 'User' that is to be persisted.

Employee.java:

Employee.java:

```
package com.hibernate.model;
import java.util.Date;
public class Employee {

   private int id;
   private String name;
   private String role;
   //private Date insertTime;

   public int getId() {
       return id;
   }
   public void setId(int id) {
       this.id = id;
   }
   public String getName() {
       return name;
   }
   public void setName(String name) {
       this.name = name;
   }
   public String getRole() {
       return role;
   }
   public void setRole(String role) {
       this.role = role;
   }
   @Override
```

```
    public String toString() {
        return "Employee [id=" + id + ", name=" +
name + ", role=" + role + "]";
    }

}
```

Client Application:

Following is the listing for the Client Application.

HibernateUtil.java

```
package com.hibernate.util;
import java.util.Properties;
import org.hibernate.SessionFactory;
import
org.hibernate.boot.registry.StandardServiceRegis
tryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;
import com.hibernate.model.Employee;
public class HibernateUtil {

    //XML based configuration
    private static SessionFactory sessionFactory;

     private static SessionFactory
buildSessionFactory() {
         try {
             // Create the SessionFactory from
hibernate.cfg.xml
            Configuration configuration = new
Configuration();

    configuration.configure("hibernate.cfg.xml");
            System.out.println("Hibernate
Configuration loaded");
```

```
            ServiceRegistry serviceRegistry = new
StandardServiceRegistryBuilder().applySettings(c
onfiguration.getProperties()).build();
            System.out.println("Hibernate
serviceRegistry created");

            SessionFactory sessionFactory =
configuration.buildSessionFactory(serviceRegistr
y);
                        return sessionFactory;
        }
        catch (Throwable ex) {
            // Make sure you log the exception,
as it might be swallowed
            System.err.println("Initial
SessionFactory creation failed." + ex);
            throw new
ExceptionInInitializerError(ex);
        }
    }
    public static SessionFactory
getSessionFactory() {
       if(sessionFactory == null) sessionFactory
= buildSessionFactory();
        return sessionFactory;
    }
}
```

Service and Registries are the newly introduced as part of Hibernate 4.0, here as part the purpose of Service and Registries.

What is Service?

Services (Components) are the way through which we can add various different functionalities in pluggable manner. Hibernate requires various different resources like Connections, ClassLoaders.. to perform an persistence operation, such components can be plugged-in into the hibernate through the help of Service. There are several Service Role interfaces defined for certain functionalities and the Service

Implementations for those Service Contract Interfaces. As the Service Implementations are being consumed through Service Interfaces these are said to be pluggable components;

To better understand a Service lets look at the below While performing a persistence operation Hibernate requires Jdbc Connections to database. The way it obtains and releases the connections is through ConnectionProvider service. The service is defined by the interface (Service Role) org.hibernate.engine.jdbc.connections.spi.ConnectionProvider. There are multiple implementations are there for the ConnectionProvider interface "org.hibernate.engine.jdbc.connections.internal.DatasourceConnectionProviderImpl" for DataSource Connections, "org.hibernate.c3p0.internal.C3P0ConnectionProvider" for using C3P0 connections.

Internally Hibernate always referes to ConnectionProvider Interface rather than a specific implementations in consuming the Service.

What is ServiceRegistry?

ServiceRegistry holds and manages (lifecycle) the Services. The Services has several charateristics like Services has lifecycle, scope and dependencies with other Services. ServiceRegistry fulfils all such charateristics by acting as an ioc container for managing all Services.

ServiceRegistries are hierarchial, so a ServiceRegistry has parent. Services in one registry can depend on and utilize Services in the same or its parent registry.

Types of ServiceRegistries

In Hibernate there are 3 different ServiceRegistries are implementing ServiceREgistry interface exists.

BootstrapServiceRegistry

The root registry is the BootstrapServiceRegistry and holds 3 Services in it.

1) ClassLoaderService
2) IntegratorService

3) StrategySelectorService

StandardServiceRegistry

It is the main Hibernate Registry and has its parent as BootstrapServiceRegistry. This holds most of the Services used by Hibernate.

1) ConnectionProvider
2) JdbcServices
3) TransactionFactory
4) JtaPlatform
5) RegionFactory

SessionFactoryServiceRegistry

It is the another Service registry and has its parent as StandardServiceRegistry. This has been designed to hold Services that are required to access SessionFactory.

1) EventListenerRegistry
2) StatisticsImplementor

For all the above ServiceRegistries there are respective Builders to create the Objects of them.

Building ServiceRegistry

In Hibernate 4 we build ServiceRegistries and pass as input while creating the SessionFactory. Usually we create StandardServiceRegistry and pass it as an input while creating the SessionFactory leaving other registries to their default.

Following are the ways we can create StandardServiceRegistry

```
StandardServiceRegistry registry = new
StandardServiceRegistryBuilder().applySetting("h
ibernate.connection.driver_class", "
racle.jdbc.driver.OracleDriver").applySetting("h
ibernate.connection.url", "
jdbc:oracle:thin:@172.16.10.2:1521:orcl").build(
);
```

Even we can create StandardServiceRegistry through hibernate.cfg.xml or properties as shown below.

```
StandardServiceRegistry registry = new
StandardServiceRegistryBuilder().configure().loa
dProperties("hibernate.properties").build();
```

Once the StandardServiceRegistry has been created pass it as an input in creating the SessionFactory as shown below.

```
StandardServiceRegistry registry = new
StandardServiceRegistryBuilder().configure().bui
ld();
SessionFactory sessionFactory = new
Configuration().buildSessionFactory(registry);
```

HibernateMain.java

package com.hibernate.main;

```
import java.util.Iterator;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import com.hibernate.model.Employee;
import com.hibernate.util.HibernateUtil;
public class HibernateMain {
   public static void main(String[] args) {
```

```
Employee e2 = new Employee();
        e2.setId(124);

 Employee e1 = new Employee();
        e1.setId(121);
        e1.setName("Andrina");
        e1.setRole("MD");
        Employee e = new Employee();
        e.setId(100);
        e.setName("Sam");
        e.setRole("Manager");
        //Get Session
        Session session =
HibernateUtil.getSessionFactory().getCurrentSess
ion();
               //start transaction
        session.beginTransaction();
        //Save the Model object means insert the
record in the datadbase.
        session.save(e2);
        session.save(e1);
        session.save(e);
              //To fetch the all the records.
        //using the Query object we are
generating the query which will fetch the all
the records
        Query query = session.createQuery("From
Employee e");
        List<Employee> list = query.list();
        Iterator<Employee> itr = list.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
```

```
//Use of the load method to get the particular
record
        e = (Employee)
session.load(Employee.class,121);
               //Now the same record is we can
update
       e.setName("Sam");
       e.setRole("Manager");
       session.update(e);
       //To delete the records
      e1 = (Employee)
session.get(Employee.class,124);
       session.delete(e1);
       //Commit transaction
       session.getTransaction().commit();

       //terminate session factory, otherwise
program won't end

   HibernateUtil.getSessionFactory().close();
   }
}
```

Hibernate using Annotation

Annotations to provide metadata configuration along with the Java code. That way the code is easy to understand. Annotations are less powerful than XML configuration. XML also gives you the ability to change the configuration without building the project. So use annotations only for table and column mappings, not for frequently changing stuff like database connection and other properties. Annotations are preconfigured with sensible default values, which reduce the amount of coding required, e.g. class name defaults to table name and field names defaults to column names.

Annotation	Modifier	Description
`@Entity`		Marks a class as a Hibernate Entity (Mapped class)
`@Table`	`Name`	Maps this class with a database table specified by `name` modifier. If `name` is not supplied it maps the class with a table having same name as the class
`@Id`		Marks this class field as a primary key column
`@GeneratedValue`		Instructs database to generate a value for this field automatically
`@Column`	`Name`	Maps this field with table column specified by

		`name` and uses the field name if `name` modifier is absent
`@ManyToMany`	`Cascade`	Marks this field as the owning side of the many-to-many relationship and `cascade` modifier specifies which operations should cascade to the inverse side of relationship
	`mappedBy`	This modifier holds the field which specifies the inverse side of the relationship
`@JoinTable`	`Name`	For holding this many-to-many relationship, maps this field with an intermediary database

		join table specified by `name` modifier
	`joinColumns`	Identifies the owning side of columns which are necessary to identify a unique owning object
	`inverseJoinColumns`	Identifies the inverse (target) side of columns which are necessary to identify a unique target object
`@JoinColumn`	`Name`	Maps a join column specified by the `name` identifier to the relationship table specified by `@JoinTable`

Setup Database Configuration

Create `hibernate.cfg.xml` file as shown in below and configure the database connection and mapping classes. Here is

the XML configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
	"-//Hibernate/Hibernate Configuration DTD
3.0//EN"
	"http://hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>

<session-factory>
	<property
name="hibernate.connection.driver_class">oracle.
jdbc.driver.OracleDriver</property>
<property
name="hibernate.connection.url">jdbc:oracle:thin
:@172.16.10.2:1521:orcl</property>
	<property
name="hibernate.connection.username">java</prope
rty>
	<property
name="hibernate.connection.password">java</prope
rty>
	<property
name="hibernate.connection.pool_size">10</proper
ty>
	<!-- org.hibernate.HibernateException: No
CurrentSessionContext configured! -->
	<property
name="hibernate.current_session_context_class">t
hread</property>
	<!-- Outputs the SQL queries, should be
disabled in Production -->
	<property
name="hibernate.show_sql">true</property>
	<property
name="hibernate.hbm2ddl.auto">create</property>
```

```
        <!-- Dialect is required to let Hibernate
know the Database Type, MySQL, Oracle etc
             Hibernate 4 automatically figure out
Dialect from Database Connection Metadata -->
        <property
name="hibernate.dialect">org.hibernate.dialect.O
racle10gDialect</property>

        <!-- mapping file, we can use Bean
annotations too -->
        <mapping
class="com.hibernate.model.Employee" />
    </session-factory>
</hibernate-configuration>
```

```
package com.hibernate.model;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;
@Entity
@Table(name="Employee",

uniqueConstraints={@UniqueConstraint(columnNames
={"ID"})})
public class Employee {
    @Id
    //@GeneratedValue(strategy=GenerationType.IDE
NTITY)
    @Column(name="ID", nullable=false,
unique=true, length=11)
    private int id;
```

```
    @Column(name="NAME", length=20,
nullable=true)
    private String name;
                    @Column(name="ROLE",
length=20, nullable=true)
    private String role;
             @Column(name="insert_time",
nullable=true)
    private Date insertTime;
```

Create annotated model classes

Create model classes `Employee.java` and map it to the database using annotations as follows:

```
public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getRole() {
        return role;
    }
    public void setRole(String role) {
        this.role = role;
    }
    public Date getInsertTime() {
        return insertTime;
    }
```

```
    public void setInsertTime(Date insertTime) {
        this.insertTime = insertTime;
    }
}
```

Writing Test Program

```
package com.hibernate.util;
import java.util.Properties;
import org.hibernate.SessionFactory;
import
org.hibernate.boot.registry.StandardServiceRegis
tryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

import com.hibernate.model.Employee;

public class HibernateUtil {

    //XML based configuration
    private static SessionFactory sessionFactory;

    //Annotation based configuration
    private static SessionFactory
sessionAnnotationFactory;
    //Property based configuration
    private static SessionFactory
sessionJavaConfigFactory;
    private static SessionFactory
buildSessionAnnotationFactory() {
        try {
             // Create the SessionFactory from
hibernate.cfg.xml
            Configuration configuration = new
Configuration();
            configuration.configure("hibernate-
annotation.cfg.xml");
```

```
            System.out.println("Hibernate
Annotation Configuration loaded");
```

Create a HibernateUtil class to read XML database configuration we did earlier and create a Configuration object to obtain a SessionFactory object. Please make sure that hibernate.cfg.xml file is on the classpath. Following is code of the HibernateUtil.java class:

```
ServiceRegistry serviceRegistry = new
StandardServiceRegistryBuilder().applySettings(c
onfiguration.getProperties()).build();
            System.out.println("Hibernate
Annotation serviceRegistry created");

            SessionFactory sessionFactory =
configuration.buildSessionFactory(serviceRegistr
y);

                return sessionFactory;
            }
            catch (Throwable ex) {
                // Make sure you log the exception,
as it might be swallowed
                System.err.println("Initial
SessionFactory creation failed." + ex);
                throw new
ExceptionInInitializerError(ex);
            }
}
public static SessionFactory
getSessionAnnotationFactory() {
        if(sessionAnnotationFactory == null)
sessionAnnotationFactory =
buildSessionAnnotationFactory();
            return sessionAnnotationFactory;
    }
}
package com.hibernate.main;
import java.util.Iterator;
```

```
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import com.hibernate.model.Employee;
import com.hibernate.util.HibernateUtil;
public class HibernateMain {
   public static void main(String[] args) {
        Employee e2 = new Employee();
        e2.setId(124);
        e2.setName("Aniket");
        e2.setRole("Clerk");
        Employee e1 = new Employee();
        e1.setId(121);
        e1.setName("Manasi");
        e1.setRole("MD");
          Employee e = new Employee();
         e.setId(100);
            e.setName("Sam");
        e.setRole("Manager");
        //Get Session
        Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
        //start transaction
        session.beginTransaction();
        //Save the Model object means insert the
record in the datadbase.
        session.save(e2);
        session.save(e1);
        session.save(e);
```

Here is code of the HibernateMain class:

```
//Use of the load method to get the particular
record
         e = (Employee)
session.load(Employee.class,121);
         //Now the same record is we can update
        e.setName("Sam");
        e.setRole("Manager");
```

```
          session.update(e);
          //To delete the records
        e1 = (Employee)
session.get(Employee.class,124);
          session.delete(e1);
        //Commit transaction
        session.getTransaction().commit();
        //terminate session factory, otherwise
program won't end

   HibernateUtil.getSessionAnnotationFactory().c
lose();
    }
}
```

you can configure session factory connection details using XML and entities using annotation configuration respectively in Hibernate and access the database. By using XML, database connection properties can be easily changed without changing the Java source files which is an added advantage. By using annotations, Java entity classes are more expressive and you don’t have to refer to another XML file for figuring out the Hibernate-Database mapping.

Creating Persistent Classes

The entire concept of Hibernate is to take the values from Java class attributes and persist them to a database table. A mapping document helps Hibernate in determining how to pull the values from the classes and map them with table and associated fields.

Java classes whose objects or instances will be stored in database tables are called persistent classes in Hibernate. Hibernate works best if these classes follow some simple rules, also known as the **Plain Old Java Object** (POJO) programming model.

There are following main rules of persistent classes, however, none of these rules are hard requirements –

- All Java classes that will be persisted need a default constructor.
- All classes should contain an ID in order to allow easy identification of your objects within Hibernate and the database. This property maps to the primary key column of a database table.
- All attributes that will be persisted should be declared private and have **getXXX** and **setXXX** methods defined in the JavaBean style.
- A central feature of Hibernate, proxies, depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods.
- All classes that do not extend or implement some specialized classes and interfaces required by framework.

Simple POJO Example

Based on the few rules mentioned above, we can define a POJO class as follows

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname,
int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
      public int getId() {
      return id;
   }
public void setId( int id ) {
      this.id = id;
   }

   public String getFirstName() {
      return firstName;
   }

   public void setFirstName( String first_name )
{
      this.firstName = first_name;
   }

   public String getLastName() {
      return lastName;
   }

   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
```

```
   public int getSalary() {
      return salary;
   }

   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

So far, we have seen very basic O/R mapping using hibernate, but there are three most important mapping topics, which we have to learn in detail.

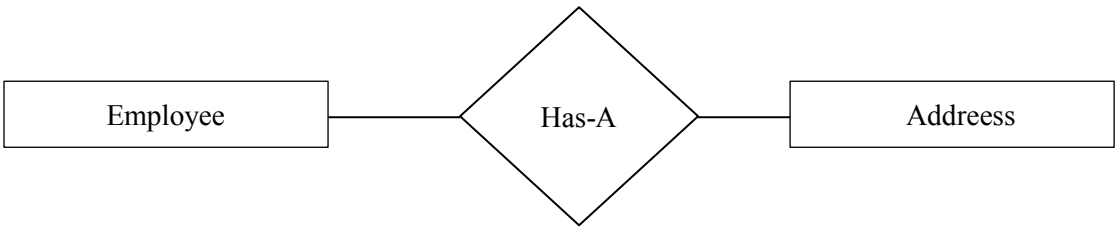
These are −

- Component Mappings.
- Inheritance Mappings
- Mapping of associations between entity classes.

Component Mappings

It is very much possible that an Entity class can have a reference to another class as a member variable. If the referred class does not have its own life cycle and completely depends on the life cycle of the owning entity class, then the referred class hence therefore is called as the Component class.

In this example you will learn how to map components using Hibernate Annotations. Consider the following relationship between *Employee* and *Address* entity.



According to the relationship each Employee should have a unique address.

Since the Employee and Address entities are strongly related (composition relation), it is better to store them in a single table.

Employee class is used to create the Employee table.

```
package com.hibernate.model;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Embedded;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;
@Entity
@Table(name="Employee",

uniqueConstraints={@UniqueConstraint(columnNames
={"ID"})})
public class Employee {
   @Id
   //@GeneratedValue(strategy=GenerationType.IDE
NTITY)
   @Column(name="ID", nullable=false,
unique=true, length=11)
   private int id;
   @Column(name="NAME", length=20,
nullable=true)
   private String name;
   @Column(name="ROLE", length=20,
nullable=true)

   @Column(name="insert_time", nullable=true)
   private Date insertTime;
    @Embedded
   private Address add;

   public int getId() {
       return id;
   }
private String role;
   @Column(name="insert_time", nullable=true)
   private Date insertTime;
```

```
        @Embedded
    private Address add;
    public int getId() {
        return id;
    }
    public Address getAdd() {
        return add;
    }
    public void setAdd(Address add) {
        this.add = add;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getRole() {
        return role;
    }
public void setRole(String role) {
        this.role = role;
    }
    public Date getInsertTime() {
        return insertTime;
    }
    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" +
name + ", role=" + role + ", insertTime=" +
insertTime + ", add="
                + add + "]";
    }
    public void setInsertTime(Date insertTime) {
        this.insertTime = insertTime;
```

```
    }
}
```

The @Embedded annotation is used to specify the Address entity should be stored in the Employee table as a component.

@Embeddable annotation is used to specify the Address class will be used as a component. The Address class cannot have a primary key of its own, it uses the enclosing class primary key.

```
package com.hibernate.model;
import javax.persistence.Column;
import javax.persistence.Embeddable;
@Embeddable
public class Address {
   @Column(name="Street", length=20,
nullable=true)
   private String street;
   @Column(name="Province", length=20,
nullable=true)
               private String province;
   @Column(name="Country", length=20,
nullable=true)
             private String country;
   @Column(name="Zip", length=10, nullable=true)
              private String zip;
   public String getStreet() {
       return street;
   }
   @Override
   public String toString() {
       return "Address [street=" + street + ",
province=" + province + ", country=" + country +
", zip=" + zip + "]";
   }
public void setStreet(String street) {
       this.street = street;
   }
   public String getProvince() {
```

```
        return province;    }
         public void setProvince(String province)
{
        this.province = province;
    }
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD
3.0//EN"
        "http://hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Database connection properties -
Driver, URL, user, password -->
        <property
name="hibernate.connection.driver_class">oracle.
jdbc.driver.OracleDriver</property>
        <property
name="hibernate.connection.url">jdbc:oracle:thin
:@172.16.10.2:1521:orcl</property>
        <property
name="hibernate.connection.username">java</prope
rty>

public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
    public String getZip() {
        return zip;
    }
    public void setZip(String zip) {
        this.zip = zip;
    }
}
```

```
<property
name="hibernate.connection.password">java</prope
rty>
        <!-- Connection Pool Size -->
        <property
name="hibernate.connection.pool_size">10</proper
ty>

        <!-- org.hibernate.HibernateException: No
CurrentSessionContext configured! -->
        <property
name="hibernate.current_session_context_class">t
hread</property>
        <!-- Outputs the SQL queries, should be
disabled in Production -->
        <property
name="hibernate.show_sql">true</property>
        <property
name="hibernate.hbm2ddl.auto">create</property>

        <!-- Dialect is required to let Hibernate
know the Database Type, MySQL, Oracle etc
             Hibernate 4 automatically figure out
Dialect from Database Connection Metadata -->
        <property
name="hibernate.dialect">org.hibernate.dialect.O
racle10gDialect</property>
        <!-- mapping file, we can use Bean
annotations too -->
        <mapping
class="com.hibernate.model.Employee" />
    </session-factory>
</hibernate-configuration>
```

Create the Main class to run the example

```
package com.hibernate.main;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;

import com.hibernate.model.Address;
import com.hibernate.model.Employee;
import com.hibernate.util.HibernateUtil;
public class HibernateMain {
   public static void main(String[] args) {
       Address ad = new Address();
       ad.setStreet("Raj Patha");
       ad.setProvince("Maharashtra");
       ad.setCountry("India");
       ad.setZip("411038");
       Employee e2 = new Employee();
       e2.setId(124);
       e2.setName("Soham");
       e2.setRole("Clerk");
       e2.setAdd(ad);
       e2.setInsertTime(new Date());
       //Get Session
       Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
       //start transaction
       session.beginTransaction();
       //Save the Model object means insert the
record in the datadbase.
       session.save(e2);
       //To fetch the all the records.
       //using the Query object we are
generating the query which will fetch the all
the records
```

```
        Query query = session.createQuery("From
Employee e");
        List<Employee> list = query.list();
        Iterator<Employee> itr = list.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next()
Employee e1 = (Employee)
session.get(Employee.class,124);
        System.out.println(e1);
        session.getTransaction().commit();

        //terminate session factory, otherwise
program won't end

    HibernateUtil.getSessionAnnotationFactory().c
lose();
    }
}
```

Hibernate Inheritance Mapping

Relational databases don't have a straightforward way to map class hierarchies onto database tables.

To address this, the JPA specification provides several strategies:

- Table Per Hierarchy
- Table Per Concrete class
- Table Per Subclass

Each strategy results in a different database structure.

Entity inheritance means that we can use polymorphic queries for retrieving all the sub-class entities when querying for a super-class.

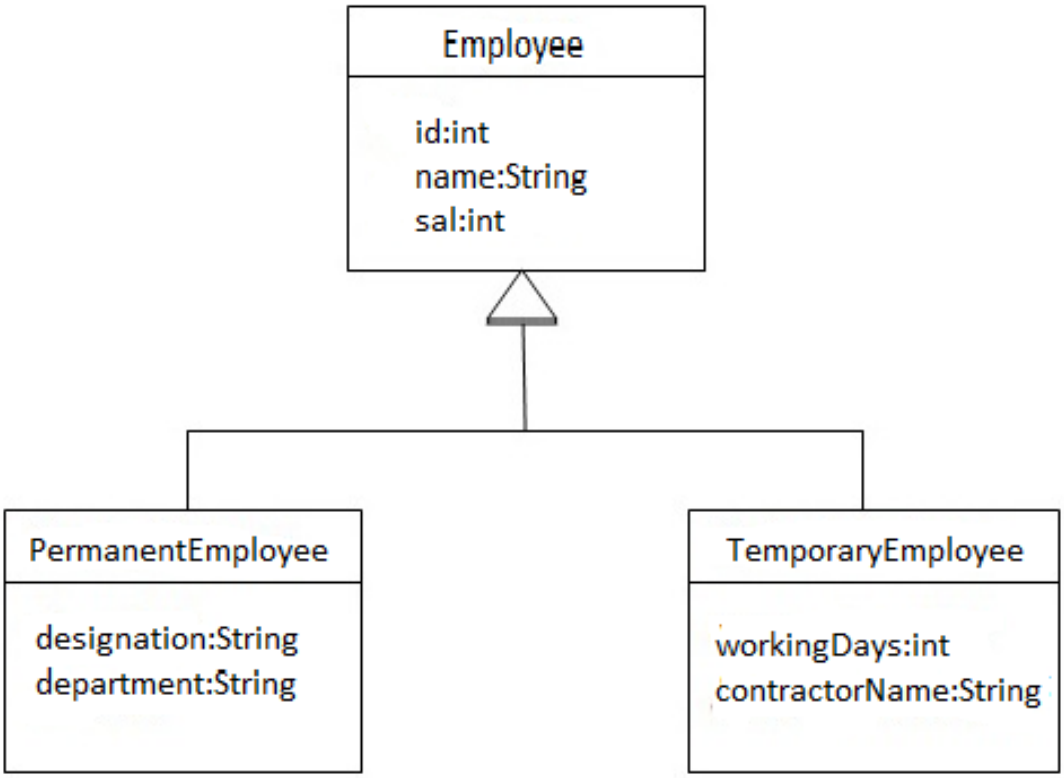
Since Hibernate is a JPA implementation, it contains all of the above as well as a few Hibernate-specific features related to inheritance.

In the next sections, we'll go over available strategies in more detail.

Table Per Hierarchy

In table per hierarchy mapping, single table is required to map the whole hierarchy, an extra column (known as discriminator column) is added to identify the class. But nullable values are stored in the table .we have mapped the inheritance hierarchy with one table only using xml file. Here, we are going to perform this task using annotation. You need to use @Inheritance(strategy=InheritanceType.SINGLE_TABLE), @DiscriminatorColumn and @DiscriminatorValue annotations for mapping table per hierarchy strategy.

In case of table per hierarchy, only one table is required to map the inheritance hierarchy. Here, an extra column (also known as discriminator column) is created in the table to identify the class.



There are three classes in this hierarchy. Employee is the super class for Permanent_Employee and Temporary_Employee classes.

The table structure for this hierarchy is as shown below:

Column Name	Data Type	Nullable	Default	Primary Key
ID	Number(10,0)	No		1
EmployeeType	Varchar2(255)	No		
NAME	Varchar2(255)	Yes		
Salary	Number(10,0)	Yes		
Designation	Varchar2(255)	Yes		
Department	Varchar2(255)	Yes		
WorkingDays	Number(10,0)	Yes		
ContractorName	Varchar2(255)	Yes		

Example of Hibernate Table Per Hierarchy using Annotation

You need to follow following steps to create simple example:

- Create the persistent classes
- Create the configuration file
- Create the class to store the fetch the data

1) Create the Persistent classes Employee.java

```
package com.hibernate.model;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;
import javax.persistence.InheritanceType;
import javax.persistence.DiscriminatorValue;
import javax.persistence.DiscriminatorType;
@Entity
@Table(name="Employee",

uniqueConstraints={@UniqueConstraint(columnNa
mes={"ID"})})
//Here we are using One Table Per Class
Hierarchy
@Inheritance(strategy=InheritanceType.SINGLE_
TABLE)
@DiscriminatorColumn(
        name="EmployeeType",

discriminatorType=DiscriminatorType.STRING,
        length=10   )
@DiscriminatorValue(value="Emp")
public class Employee {
    @Id
    //@GeneratedValue(strategy=GenerationType
.IDENTITY)
    @Column(name="ID", nullable=false,
unique=true, length=11)
    private int id;
    @Column(name="NAME", length=20,
nullable=true )
```

```
    private String name;
    @Column(name="Salary", nullable=true)
    private  int sal;
    //getters and setters
    }
```

File: PermanentEmployee.java

```
package com.hibernate.model;
import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.Table;
@Entity
@Table(name="Employee")
@DiscriminatorValue("PEmp")
public class PermanentEmployee extends Employee
{
@Column(name="Designation", length=20,
nullable=true )
  private String designation;
   @Column(name="Department", length=20,
nullable=true )
   private String department;
//getters and setters
}
```

File: TemporaryEmployee.java

```
package com.hibernate.model;
import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.Table;
@Entity
@Table(name="Employee")
@DiscriminatorValue("CEmp")
public class TemporaryEmployee extends Employee
{
```

```
    @Column(name="WorkingDays", nullable=true )
    private short workingDays;
    @Column(name="ContractorName", nullable=true
,length=20 )
    private  String contractorName;
    //getters and setters
}
```

Hibernate-annotation.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD
3.0//EN"
        "http://hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
                <property
name="hibernate.connection.driver_class">oracle.
jdbc.driver.OracleDriver</property>
    <property
name="hibernate.connection.url">jdbc:oracle:thin
:@172.16.10.2:1521:orcl</property>
```

The client class HibernateMainWithTablePerClass.java

```
package com.hibernate.main;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.hibernate.model.PermanentEmployee;
import com.hibernate.model.TemporaryEmployee;
import com.hibernate.util.HibernateUtil;
public class HibernateMainWithTablePerClass {
    public static void main(String[] args) {
```

```
   Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
   //start transaction
        Transaction tx
=session.beginTransaction();
<property
name="hibernate.connection.username">java</prope
rty>
   <property
name="hibernate.connection.password">java</prope
rty>
   <property
name="hibernate.connection.pool_size">10</proper
ty>
   <property
name="hibernate.current_session_context_class">t
hread</property>
   <property
name="hibernate.show_sql">true</property>
   <property
name="hibernate.hbm2ddl.auto">create</property>
   <property
name="hibernate.dialect">org.hibernate.dialect.O
racle10gDialect</property>
   <!-- mapping file, we can use Bean
annotations too -->
       <mapping
class="com.hibernate.model.Employee" />
       <mapping
class="com.hibernate.model.PermanentEmployee" />
       <mapping
class="com.hibernate.model.TemporaryEmployee" />
   </session-factory>
</hibernate-configuration>
//Creating the object of two Permanent Employees
        PermanentEmployee p1 = new
PermanentEmployee();
        p1.setId(1011);
```

```
         p1.setName("S N Rao");
         p1.setSal(25000);
         p1.setDesignation("Chemist");
         p1.setDepartment("Drugs");
        PermanentEmployee p2 = new
PermanentEmployee();
          p2.setId(1014);
          p2.setName("Sridhar");
          p2.setDesignation("Foreman");
          p2.setDepartment("Chemicals");
        //Creating the object of two Temporary
Employees
            TemporaryEmployee t1 = new
TemporaryEmployee();
            t1.setId(102);
            t1.setName("Jyostna");
            t1.setWorkingDays((short)28);
            t1.setContractorName("Raju");
          TemporaryEmployee t2 = new
TemporaryEmployee();
            t2.setId(103);
            t2.setName("Jyothi");
            t2.setWorkingDays((short)22);
            t2.setContractorName("Pratap");
            session.save(p1);
            session.save(p2);
            session.save(t1);
            session.save(t2);
```

Table Per Concrete class using Annotation

In case of Table Per Concrete class, tables are created per class. So there are no nullable values in the table. Disadvantage of this approach is that duplicate columns are created in the subclass tables.

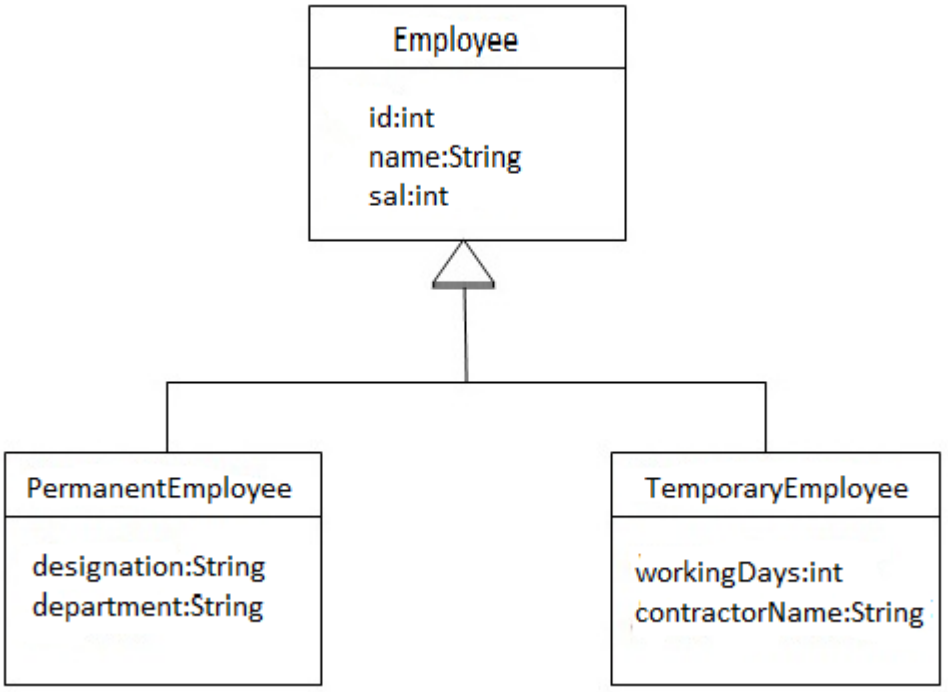
Here, we need to use @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) annotation in the parent class and @AttributeOverrides annotation in the subclasses.

@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) specifies that we are using table per concrete class strategy. It should be specified in the parent class only.

@AttributeOverrides defines that parent class attributes will be overriden in this class. In table structure, parent class table columns will be added in the subclass table.

The class hierarchy is given below:

```
Query q = session.createQuery("FROM
PermanentEmployee");
            List<PermanentEmployee> myList =
q.list();
            for(PermanentEmployee p :myList)
            {
              System.out.println(p1.getId()+ ", "
+ p1.getName() + ", " + p1.getDesignation() + ",
" + p1.getDepartment());
            }
         Query q1 = session.createQuery("FROM
TemporaryEmployee");
            List<TemporaryEmployee> yourList =
q1.list();
            for(TemporaryEmployee t : yourList)
            {
              System.out.println(t1.getId()+ ", "
+ t1.getName() + ", " + t1.getWorkingDays() + ",
" + t1.getContractorName());
            }
        tx.commit();
          //terminate session factory, otherwise
program won't end

    HibernateUtil.getSessionAnnotationFactory().c
lose();
    }
}
```



The Structure of Employee class is shown below

Column Name	Data Type	Nullable	Default	Primary Key
ID	Number(10,0)	No		1
NAME	Varchar2(255)	Yes		
Salary	Number(10,0)	Yes		

ParmanentEmployee :

Column Name	Data Type	Nullable	Default	Primary Key
ID	Number(10,0)	No		1
NAME	Varchar2(255)	Yes		
Salary	Number(10,0)	Yes		
Designation	Varchar2(255)	Yes		

Department	Varchar2(255)	Yes		

Example of Table per concrete class

1) Create the Persistent classes

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

File: Employee.java

```
package com.hibernate.model;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;
import javax.persistence.InheritanceType;
import javax.persistence.DiscriminatorValue;
import javax.persistence.DiscriminatorType;
@Entity
@Table(name="Employee",

uniqueConstraints={@UniqueConstraint(columnNames
={"ID"})})
//Here we are using One Table Per Class
Hierarchy
@Inheritance(strategy=InheritanceType.TABLE_PER_
CLASS)
public class Employee {
    @Id
    //@GeneratedValue(strategy=GenerationType.IDE
NTITY)
```

```
   @Column(name="ID", nullable=false,
unique=true, length=11
             private int id;
@Column(name="NAME", length=20, nullable=true)
   private String name;

   @Column(name="Salary", nullable=true)
   private  int sal;
  //getters and setters
```

File: PermanentEmployee

```
package com.hibernate.model;
import javax.persistence.AttributeOverride;
import javax.persistence.Column;
import javax.persistence.AttributeOverrides;
import javax.persistence.Entity;
import javax.persistence.Table;
@Entity
@Table(name="PermanentEmployee")
@AttributeOverrides({
    @AttributeOverride(name="name",
column=@Column(name="NAME")),
    @AttributeOverride(name ="Sal",
column=@Column(name="Salary"))
})
public class PermanentEmployee extends Employee
{
@Column(name="Designation", length=20,
nullable=true )
   private String designation;
    @Column(name="Department", length=20,
nullable=true )
    private String department;
   //getters and setters
```

File: TemporaryEmployee

```
package com.hibernate.model;
import javax.persistence.AttributeOverride;
import javax.persistence.AttributeOverrides;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;
@Entity
@Table(name="TemporaryEmployee")
@AttributeOverrides({
    @AttributeOverride(name="name",
column=@Column(name="NAME")),
    @AttributeOverride(name ="Sal",
column=@Column(name="Salary"))
})
public class TemporaryEmployee extends Employee
{
   @Column(name="WorkingDays", nullable=true )
   private short workingDays;
   @Column(name="ContractorName", nullable=true
,length=20 )
   private  String contractorName;
       //getters  and setters
```

2) Add mapping of hbm file in configuration file

File: hibernate-annoatation.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
       "-//Hibernate/Hibernate Configuration DTD
3.0//EN"
       "http://hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
```

```
        <!--  Mappling of hibernate properties
like driver_class,url,username,password
,dialet,show_sql -- >
     <!-- mapping file, we can use Bean
annotations too -->
       <mapping
class="com.hibernate.model.Employee" />
       <mapping
class="com.hibernate.model.PermanentEmployee" />
       <mapping
class="com.hibernate.model.TemporaryEmployee" />
   </session-factory>
</hibernate-configuration>
```

3) Create the class that stores the persistent object

In this class, we are simply storing the employee objects in the database.

File: HibernateMainWithTablePerConvcreteClass.java

```
package com.hibernate.main;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.hibernate.model.PermanentEmployee;
import com.hibernate.model.TemporaryEmployee;
import com.hibernate.util.HibernateUtil;
public class
HibernateMainWithTablePerConvcreteClass {

public static void main(String[] args) {
       Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
       //start transaction
        Transaction tx
=session.beginTransaction();
```

```
                                //Creating the
object of two Permanent Employees
        PermanentEmployee p1 = new
PermanentEmployee();
        p1.setId(1011);
        p1.setName("S N Rao");
        p1.setSal(25000);
        p1.setDesignation("Chemist");
        p1.setDepartment("Drugs");
       PermanentEmployee p2 = new
PermanentEmployee();
           p2.setId(1014);
           p2.setName("Sridhar");
           p2.setDesignation("Foreman");
           p2.setDepartment("Chemicals");
                              //Creating the
object of two Temporary Employees
           TemporaryEmployee t1 = new
TemporaryEmployee();
           t1.setId(102);
           t1.setName("Jyostna");
           t1.setWorkingDays((short)28);
           t1.setContractorName("Raju");
        TemporaryEmployee t2 = new
TemporaryEmployee();
           t2.setId(103);
           t2.setName("Jyothi");
           t2.setWorkingDays((short)22);
           t2.setContractorName("Pratap");
session.save(p1);
           session.save(p2);
           session.save(t1);
           session.save(t2);
             tx.commit();
         //terminate session factory, otherwise
program won't end

   HibernateUtil.getSessionAnnotationFactory().c
lose();
```

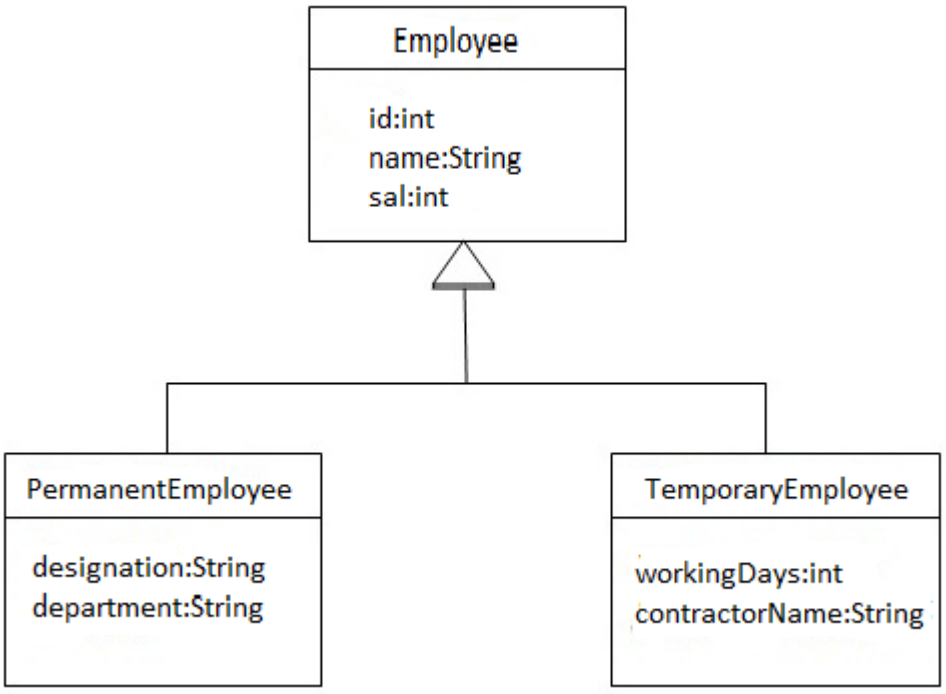
```
    }
}
```

Table Per Subclass using Annotation

In case of table per subclass strategy, tables are created as per persistent classes but they are related using primary and foreign key. So there will not be duplicate columns in the relation.

We need to specify **@Inheritance(strategy=InheritanceType.JOINED)** in the parent class and **@PrimaryKeyJoinColumn** annotation in the subclasses.

Let's see the hierarchy of classes that we are going to map.



The table structure for each table will be as follows:

Table structure for Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	Number(10,0)	No		1
NAME	Varchar2(255)	Yes		
Salary	Number(10,0)	Yes		

Table structure for ParmanentEmployee

Column Name	Data Type	Nullable	Default	Primary Key
ID	Number(10,0)	No		1
Designation	Varchar2(255)	Yes		
Department	Varchar2(255)	Yes		

Table structure for TemparoryEmployee

Column Name	Data Type	Nullable	Default	Primary Key
ID	Number(10, 0)	No		1
WorkingDays	Number(10, 0)	Yes		
ContractorName	Varchar2(255)	Yes		

Example of Table per subclass class using Annotation

1) Create the Persistent classes

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

File: Employee.java

```
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;
import javax.persistence.InheritanceType;
import javax.persistence.DiscriminatorValue;
import javax.persistence.DiscriminatorType;
@Entity
@Table(name="Employee",

uniqueConstraints={@UniqueConstraint(columnNames
={"ID"})})
//Here we are using One Table Per Class
Hierarchy
@Inheritance(strategy=InheritanceType.JOINED)
public class Employee {
    @Id
    //@GeneratedValue(strategy=GenerationType.IDE
NTITY)
```

```
   @Column(name="ID", nullable=false,
unique=true, length=11)
   private int id;
@Column(name="NAME", length=20, nullable=true)
   private String name;
   @Column(name="Salary", nullable=true)
   private  int sal;
        //getters and setters methods
```

File: PermanentEmployee.java

```
package com.hibernate.model;
import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;
@Entity
@Table(name="PermanentEmployee")
@PrimaryKeyJoinColumn(name="P_ID")
public class PermanentEmployee extends Employee
{
   @Column(name="Designation", length=20,
nullable=true )
   private String designation;
    @Column(name="Department", length=20,
nullable=true )
    private String department;
   //getters and setters method
```

File: TemporaryEmployee

```
package com.hibernate.model;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;
@Entity
@Table(name="TemporaryEmployee")
```

```
@PrimaryKeyJoinColumn(name="T_ID")
public class TemporaryEmployee extends Employee
{
   @Column(name="WorkingDays", nullable=true )
   private short workingDays;
   @Column(name="ContractorName", nullable=true
,length=20 )
   private  String contractorName;
```

1) create configuration file

```
     <!-- mapping file, we can use Bean
annotations too -->
       <mapping
class="com.hibernate.model.Employee" />
       <mapping
class="com.hibernate.model.PermanentEmployee" />
       <mapping
class="com.hibernate.model.TemporaryEmployee" />
   </session-factory>
</hibernate-configuration>
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
       "-//Hibernate/Hibernate Configuration DTD
3.0//EN"
       "http://hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
        <!--  Mappling of hibernate properties
like driver_class,url,username,password
,dialet,show_sql -- >
```

2) Create the class **that stores the persistent object**

In this class, we are simply storing the employee objects in the database.

File: HibernateMainWithTablePerSubclass.java

```
package com.hibernate.main;
```

```
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.hibernate.model.PermanentEmployee;
import com.hibernate.model.TemporaryEmployee;
import com.hibernate.util.HibernateUtil;
public class HibernateMainWithTablePerSubclass {

   public static void main(String[] args) {

       Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
       //start transaction
        Transaction tx
=session.beginTransaction();
               //Creating the object of two
Permanent Employees
        PermanentEmployee p1 = new
PermanentEmployee();
        p1.setId(1011);
        p1.setName("S N Rao");
        p1.setSal(25000);
```

3) Create the class that stores the persistent object

In this class, we are simply storing the employee objects in the database.

File: HibernateMainWithTablePerConvcreteClass.java

```
       p1.setDesignation("Chemist");
                    p1.setDepartment("Drugs");
PermanentEmployee p2 = new PermanentEmployee();
           p2.setId(1014);
           p2.setName("Sridhar");
           p2.setDesignation("Foreman");
           p2.setDepartment("Chemicals");
```

```
        //Creating the object of two Temporary
Employees
            TemporaryEmployee t1 = new
TemporaryEmployee();
            t1.setId(102);
            t1.setName("Jyostna");
            t1.setWorkingDays((short)28);
            t1.setContractorName("Raju");
         TemporaryEmployee t2 = new
TemporaryEmployee();
            t2.setId(103);
            t2.setName("Jyothi");
            t2.setWorkingDays((short)22);
            t2.setContractorName("Pratap");
            session.save(p1);
            session.save(p2);
            session.save(t1);
            session.save(t2);
tx.commit();
          //terminate session factory, otherwise
program won't end

   HibernateUtil.getSessionAnnotationFactory().c
lose();

   }
}
```

Mapping of associations between entity classes.

Using hibernate, if we want to put relationship between two entities [objects of two pojo classes], then in the database tables, there must exist foreign key relationship, we call it as Referential integrity.

The main advantage of putting relationship between objects is, we can do operation on one object, and the same operation can transfer onto the other object in the database [remember, object means one row in hibernate terminology]

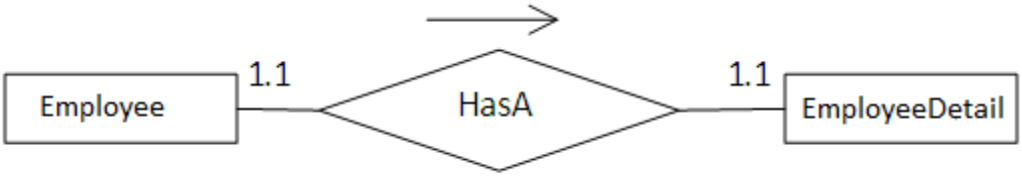
While selecting, it is possible to get data from multiple tables at a time if there exits relationship between the tables, nothing but in hibernate relationships between the objects

Using hibernate we can put the following 4 types of relationships

- One-To-One
- Many-To-One
- Many-To-Many
- One-To-Many

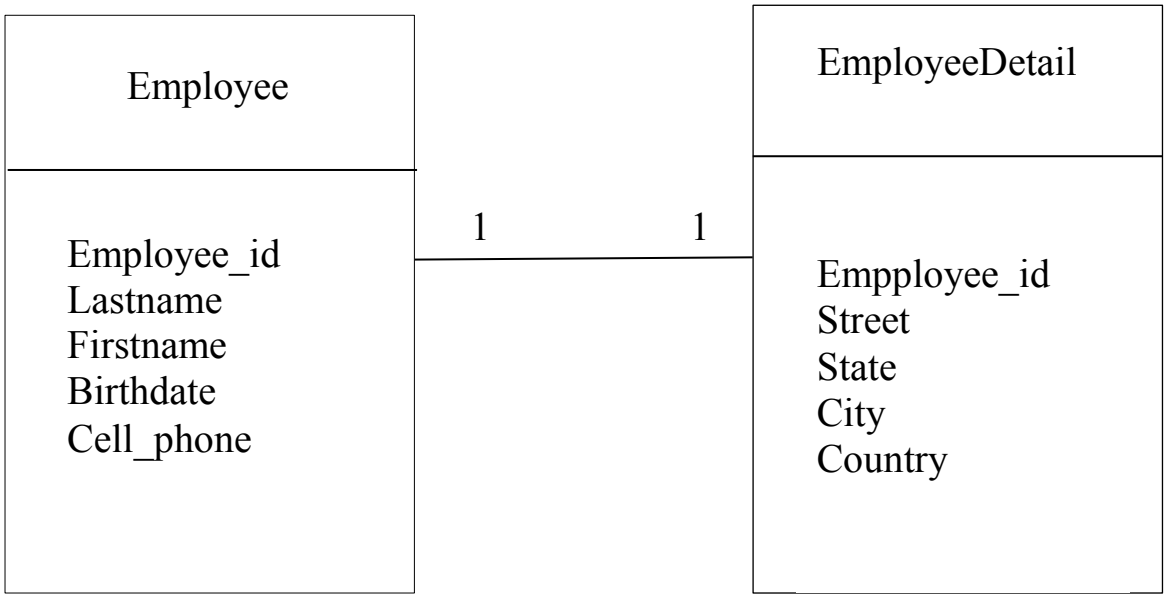
Hibernate One To One mappling using Annotation

Consider the following relationship between *Employee* and *Employee*Detail entity.



According to the relationship each Employee should have a unique EmployeeDetail.

To create this relationship you need to have a Employee and EmployeeDetail table. The relational model is shown below.



1>Database with One-to-one relationship tables.

2>Model class to Entity class

We had define two model classes Employee.java and EmployeeDetail.java in our example. These classes will be converted to Entity classes and we will add Annotations to it.

```
package com.hibernate.model;
import java.sql.Date;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;
@Entity
@Table(name="EMPLOYEE")
public class Employee {
    @Id
    @GeneratedValue
    @Column(name="employee_id")
```

File :Employee.java

```
private Long employeeId;
    @Column(name="firstname",length=15)
    private String firstname;
    @Column(name="lastname",length=15)
    private String lastname;
    @Column(name="birth_date")
    private Date birthDate;
    @Column(name="cell_phone",length=20)
    private String cellphone;
              //getters and setters
```

File :EmployeeDetail.java

```
package com.hibernate.model;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;
import
org.hibernate.annotations.GenericGenerator;
import org.hibernate.annotations.Parameter;
@Entity
@Table(name="EMPLOYEEDETAIL")
public class EmployeeDetail {
```

```
@Id
    @Column(name="employee_id", unique=true,
nullable=false)
    @GeneratedValue(generator="gen")
    @GenericGenerator(name="gen",
strategy="foreign",
parameters=@Parameter(name="property",
value="employee"))
    private Long employeeId;
    @Column(name="street",length=20)
    private String street;
    @Column(name="city",length=20)
    private String city;
    @Column(name="state",length=20)
    private String state;
    //getters and setters
    @Column(name="country",length=20)
    private String country;
    @OneToOne
    @PrimaryKeyJoinColumn
    private Employee employee;
```

```
//setters and getters
```

Note that in EmployeeDetail class we have used @GenericGenerator to specify primary key. This will ensure that the primary key from Employee table is used instead of generating a new one.

3> Update Hibernate Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD
3.0//EN"
        "http://hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
        <!--  Mappling of hibernate properties
like driver_class,url,username,password
,dialet,show_sql -- >
```

```
   <!-- mapping file, we can use Bean
annotations too -->
        <mapping
class="com.hibernate.model.Employee" />
        <mapping
class="com.hibernate.model.EmployeeDetail" />
   </session-factory>
</hibernate-configuration>
```

4>Main class to test One-to-one mapping HibernateWithMainOneToOne.java

```
package com.hibernate.main;
import java.util.List;
import java.sql.Date;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.hibernate.model.Employee;
import com.hibernate.model.EmployeeDetail;
```

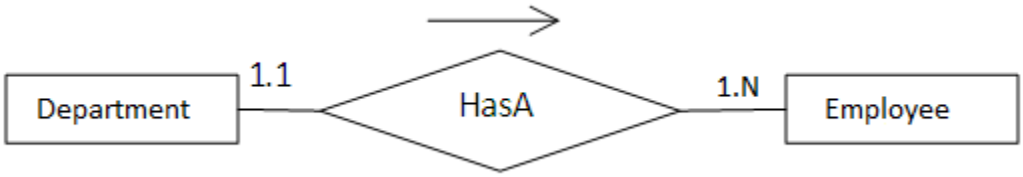
```
import com.hibernate.util.HibernateUtil;
public class HibernateWithMainOneToOne {
   public static void main(String[] args) {
            Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
       //start transaction
        Transaction tx
=session.beginTransaction();
       EmployeeDetail employeeDetail = new
EmployeeDetail("10th Street", "LA", "San
Francisco", "U.S.");
       Employee employee = new Employee("Nina",
"Mayers", new Date(121212),
                "114-857-965");

employee.setEmployeeDetail(employeeDetail);
           employeeDetail.setEmployee(employee);

           session.save(employee);
       }
```

```
List<Employee> employees =
session.createQuery("from Employee").list();
           for (Employee employee1 : employees)
{

   System.out.println(employee1.getFirstname() +
" , "
                        + employee1.getLastname()
+ ", "
                        +
employee1.getEmployeeDetail().getState());
           }
           tx.commit();
           //terminate session factory,
otherwise program won't end
```

```
    HibernateUtil.getSessionAnnotationFactory().c
lose();
    }
}
```

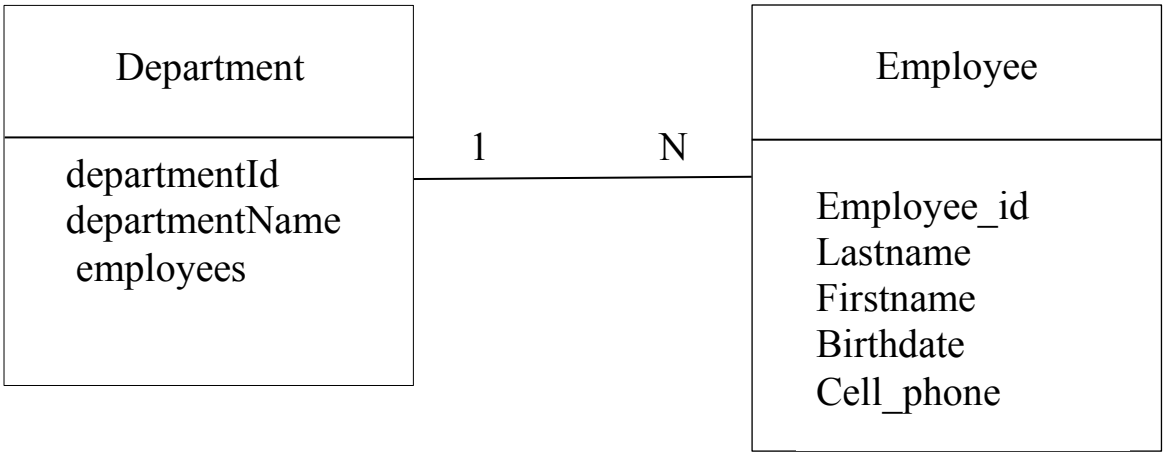
Hibernate One To Many mappling using Annotation

Consider the following relationship between *Department* and *Employee* entity.



According to the relationship each Department should have a multiple Employee associated .

To create this relationship you need to have a Departement and Employee table. The relational model is shown below.



1>Database with One-to-one relationship tables.

2>Model class to Entity class

We had define two model classes Employee.java and Department.java in our example. These classes will be converted to Entity classes and we will add Annotations to it.

File : Employee.java

```
package com.hibernate.model;
import java.sql.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
@Entity
@Table(name="EMPLOYEE")
public class Employee {
```

```
@Id
   @GeneratedValue
   @Column(name="employee_id")
   private Long employeeId;
   @Column(name="firstname",length=20)
   private String firstname;
   @Column(name="lastname" ,length=20)
   private String lastname;
   @Column(name="birth_date")
   private Date birthDate;
   @Column(name="cell_phone", length=10)
   private String cellphone;
   @ManyToOne
   @JoinColumn(name="department_id")
   private Department department;
            //getters and setters
```

@ManyToOne annotation defines a single-valued association to another entity class that has many-to-one multiplicity. It is not normally necessary to specify the target entity explicitly since it can usually be inferred from the type of the object being referenced.

@JoinColumn is used to specify a mapped column for joining an entity association.

File :Department.java

```
package com.hibernate.model;
import java.util.Set;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
@Entity
@Table(name="DEPARTMENT")
public class Department {
@Id
   @GeneratedValue
   @Column(name="DEPARTMENT_ID")
   private Long departmentId;

   @Column(name="DEPT_NAME",length=10)
   private String departmentName;

   @OneToMany(mappedBy="department")
   private Set<Employee> employees;
            //setters and getters
```

@OneToMany annotation defines a many-valued association with one-to-many multiplicity. If the collection is defined using generics to specify the element type, the associated target entity

@OneToMany annotation defines a many-valued association with one-to-many multiplicity.

If the collection is defined using generics to specify the element type, the associated target entity type need not be specified; otherwise the target entity class must be specified.

The association may be bidirectional. In a bidirectional relationship, one of the sides (and only one) has to be the owner: the owner is responsible for the association column(s) update. To declare a side as not responsible for the relationship, the attribute mappedBy is used. mappedBy refers to the property name of the association on the owner side. In our case, this is passport. As you can see, you don't have to (must not) declare the join column since it has already been declared on the owners side.

3> Update Hibernate Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD
3.0//EN"
        "http://hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
        <!--  Mappling of hibernate properties
like driver_class,url,username,password
,dialet,show_sql -- >
<!-- mapping file, we can use Bean annotations
too -->
        <mapping
class="com.hibernate.model.Employee" />
        <mapping
class="com.hibernate.model.EmployeeDetail" />
   </session-factory>
</hibernate-configuration>
```

```
package com.hibernate.main;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.hibernate.model.Department;
import com.hibernate.model.Employee;
import com.hibernate.util.HibernateUtil;
public class HibernateMainWithOneToMany {
   public static void main(String[] args) {
       Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
       //start transaction
        Transaction tx
=session.beginTransaction();
        Department department = new
Department();

   department.setDepartmentName("Sales");
           session.save(department);
           Employee emp1 = new Employee("Vinay",
"Mayers", "111");
           Employee emp2 = new Employee("Anuja",
"Almeida", "222");
           emp1.setDepartment(department);
           emp2.setDepartment(department);

           session.save(emp1);
           session.save(emp2);
           tx.commit();
            //terminate session factory,
otherwise program won't end

   HibernateUtil.getSessionAnnotationFactory().c
lose();
   }
}
```

File : HibernateMainWithOneToMany

Hibernate Query Language

HQL is mainly used to perform the bulk operations in hibernate. In our previous sessions so far, we have executed CURD operations on a single object at a time. If we want to execute a set of operations at a time, we can go with bulk operations.

- Hibernate Query Language (HQL)
- Hibernate Criteria API
- Native SQL

Lets discuss HQL in detail .

Hibernate Query Language (HQL)

- HQL is a Hibernate's own query language.
- HQL is a Database independent query language, that is., using this query language, we can develop data base independent queries.
- HQL queries are easy to learn, because HQL looks like SQL only.
- We can easily translate the SQL commands into HQL commands, by replacing the table columns with pojo class variable names, and table name with pojo class name with reference variable.
- HQL queries are also called as Object Oriented form of SQL queries.
- At runtime Hibernate will convert the HQL query into SQL query according to the database. So we no need to write a query according to the database.

Hibernate Query for Select Operations :

In hibernate, we can divide the select operations into 2 types :

- Reading Complete Entity
- Reading Partial Entity

Reading Complete Entity :

In Hibernate, selecting the all columns is called reading a complete entity.

Example :

- **SQL :** select * from emp;
- **HQL :** from Employee e;

HQL commands to read a complete entity or entities are directly begins with **"from"** keyword.

Reading Partial Entity :

In Hibernate, reading a specific columns is called reading partial entity or entities.

Example :

- SQL : select empno,sal from emp;
- HQL : select e.employeeId, e.employeeSalary from Employee e;

In HQL, reading a partial entity is begins with "select" keyword.

HQL select with conditions :

In HQL we can pass the query parameters either in index parameters or named parameters.

Example :

- SQL : select * from emp where deptno = ?;
- HQL : from Employee e where e.deptNumber = ?

OR

- HQL : from Employee e where e.deptNumber =:deptNo;

Executing Select Query in HQL :

To perform the select operation, Hibernate given us Query object.

Query : A Query is a Object Orientation representation of Hibernate query. We can get the Query instance, by calling the session.createQuery();

We can also pass the parameters to the Hibernate Query. To do that, we can use query.setParameter(int pos, Object o)

And then call the list() on query object : query.list();

query.list() returns the query results in the form of java.util.List.

Example :

A simple HQL Query :

```
Query query = session.createQuery("from Employee e");
 List qryResults = query.list();
```

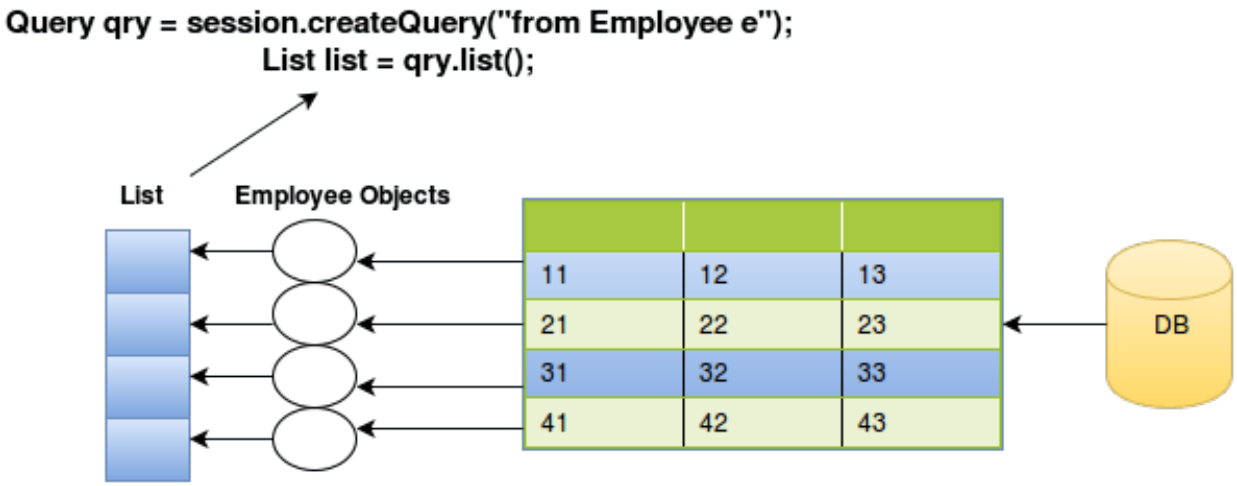
HQL Query with parameters :

```
Query query = session.createQuery("from Employee e where e.deptNumber=?");
 query.setParameter(0,300);
 List qryResults = query.list();
```

Reading the Complete Entity :

If the HQL command is to read the complete entity or entities then the list contains the complete objects. That means, Hibernate internally does like below:

- Hibernate gets the data from the database and it stores the records of table in ResultSet object.
- Each record of the ResultSet will be set to a pojo class object.
- Adds each object of the pojo class to the java.util.List and finally returns the List.

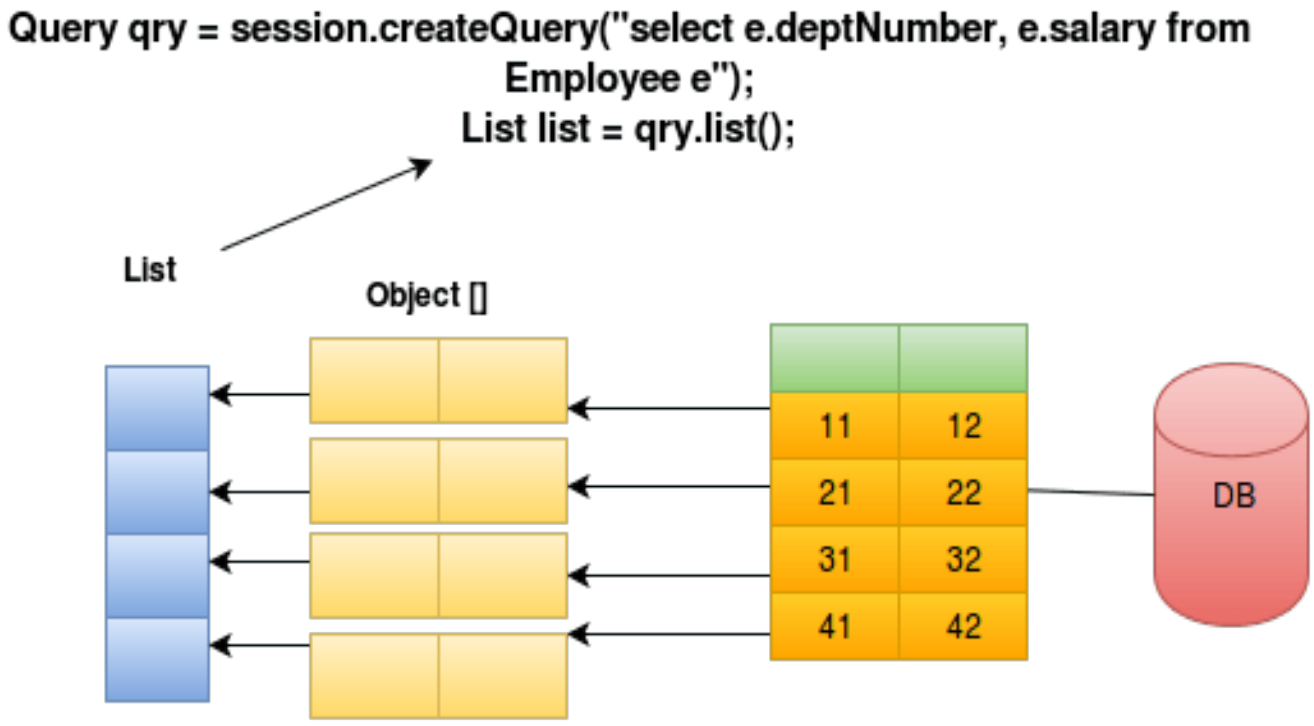


Reading the partial entity :

```
Iterator it =list.iterator();
while (it.hasNext()) {
            Employee emp = (Employee) it.next();


          System.out.println("Employee Name : "
+ emp.getEmployeeName()
                     + " , Salary : " +
emp.getSalary());
        }
```

If HQL command is to read partial entity or entities, then the list contains the Objects array (Object[]). That is the hibernate does internally like below:

- Hibernate reads partial entities from table and stores the result in ResultSet.
- Hibernate will set the data of each record in ResultSet to an Object array.
- Each object[] will be added to the list.
- Finally it returns the list.



```
Iterator it = list.iterator();

while (it.hasNext()) {
            Object[] object = (Object[])
it2.next();
            System.out.println("Department
Number : " + object[0]
                    + " Salary : " + object[1]);
        }
```

Reading partial entity with single column :

If the HQL command is to read partial entity with single column then the list contains Objects(Integer,String and etc.). That is., the Hibernate does internally like below:

- Hibernate reads a single column of each record and stores in ResultSet.
- Hibernate will set each record of ResultSet into an Object of that property type (depends on the property type)
- Adds objects to the list and finally returns the list.
- Example : Employee.java

```
package com.hibernate.model;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import
javax.persistence.UniqueConstraint;@Entity
@Table(name="Employee",
uniqueConstraints={@UniqueConstraint(columnNames
={"ID"})})
public class Employee {
   @Id
   //@GeneratedValue(strategy=GenerationType.IDE
NTITY)
```

```
   Column(name="ID", nullable=false,
unique=true, length=11)
   private int id;
```

```
@Column(name="NAME", length=20, nullable=true)
   private String name;
@Column(name="ROLE", length=20, nullable=true)
   private String role;
   @Column(name="Salary", nullable=true)
   private  int sal;
   //getters and setters
```

File : HibernateMainWithHQLClauses.java

```
package com.hibernate.main;
import java.util.Iterator;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import com.hibernate.model.Employee;
import com.hibernate.util.HibernateUtil;
public class HibernateMainWithHQLClauses {
   public static void main(String[] args) {
       Employee e2 = new Employee();
       e2.setId(124);
       e2.setName("Aniket");
       e2.setRole("Clerk");
       e2.setSal(7800);
       Employee e1 = new Employee();
       e1.setId(121);
       e1.setName("Manasi");
       e1.setRole("MD");
       e1.setSal(80222);
```

```
Employee e = new Employee();
        e.setId(100);
        e.setName("Sam");
        e.setRole("Manager");
        e.setSal(45000);
        //Get Session
        Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
        //start transaction
        session.beginTransaction();
        //Save the Model object means insert the
record in the datadbase.
        session.save(e2);
        session.save(e1);
        session.save(e);
        /To fetch the all the records.
        //using the Query object we are
generating the query which will fetch the all
the records
        //Different Clauses
        // From clause
        Query query = session.createQuery("From
Employee ");
        List<Employee> list = query.list();
        Iterator<Employee> itr = list.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
        System.out.println("Data Retrived");
        //As Clause(used to give alias
specifically when query is  long
         query = session.createQuery("From
Employee e ");
        /Select Clause
         System.out.println("Use of Select
Clause");
```

```
        query=session.createQuery("select
e.id,e.name from Employee e");
```

```
list=query.list();
            Iterator it=list.iterator();
            while(it.hasNext())
            {
                Object o[] = (Object[])it.next();
                System.out.println("Employee id :
"+o[0]+ "\tEmployee Name : "+o[1]);
            }
            System.out.println("----------------
------------------------------");
                        System.out.println("Use of
where clause");
            query=session.createQuery("from
Employee e where e.id<124");
            list=query.list();
            it=list.iterator();
            while(it.hasNext())
            {
                System.out.println(it.next());

            }
                            System.out.println("--
------------------------------------------
");
            //where Clause+Order By clause
            query=session.createQuery("from
Employee e where e.id>100 ORDER BY e.sal DESC");
             list=query.list();
             it=list.iterator();
            while(it.hasNext())
            {
                System.out.println(it.next());

            }
```

```
            System.out.println("----------------
------------------------------");
            //Group by clause
        query=session.createQuery("select
sum(e.sal),e.name from Employee e GROUP BY
e.name");
```

```
list=query.list();
             it=list.iterator();
            while(it.hasNext())
            {
                Object o[] = (Object[])it.next();

   System.out.println("Employee id : "+o[0]+
"\tEmployee Name : "+o[1]);
            }
            System.out.println("----------------
------------------------------");
            //terminate session factory,
otherwise program won't end

   HibernateUtil.getSessionAnnotationFactory().c
lose();
   }




}
```

HQL with Aggregate functions

Hibernate supports several aggregate functions **similar to SQL aggregate functions**.

Aggregate functions are used to get an **aggregate value** of any column like *SUM(salary) from Employee table* , *AVG(salary) from Employee table*

There are multiple such **aggregate methods** as below supported by Hibernate

avg(...)
sum(...)
min(...)
max(...)
count(*)
count(...)
count(distinct ...)
count(all...)

File : HibernateMainWithAggreegateFunctions.java

```
package com.hibernate.main;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import com.hibernate.model.Employee;
import com.hibernate.util.HibernateUtil;
public class
HibernateMainWithAggreegateFunctions {
   public static void main(String[] args) {


        Employee e2 = new Employee();
        e2.setId(124);
        e2.setName("Aniket");
        e2.setRole("Clerk");
        e2.setSal(7800);
        Employee e = new Employee();
```

```
                e.setId(100);
                e.setName("Sam");
        e.setRole("Manager");
        e.setSal(45000);
        //Get Session
        Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
```

```
//start transaction
        session.beginTransaction();
        session.save(e2);
        session.save(e1);
        session.save(e);
        //Use of Count aggregate function.
        Query q=session.createQuery("select
count(distinct e.name) from Employee e");
        List<Integer> list=q.list();
        System.out.println(list);
        System.out.println("----------------------
---------------------------");
        //Use Sum aggregate Function
         q=session.createQuery("select sum(e.sal)
from Employee e");
         list=q.list();
        System.out.println(list);
        System.out.println("----------------------
---------------------------");
        //Use of Avg aggregatefunction
         q=session.createQuery("select avg(e.sal)
from Employee e");
         list=q.list();
         System.out.println(list);
         System.out.println("----------------------
---------------------------");
        //Use of Min aggregate functions
        q=session.createQuery("select min(e.sal)
from Employee e");
```

```
        list=q.list();
        System.out.println(list);
         System.out.println("-------------------
--------------------------");
```

```
//Use of Max aggregate function
         q=session.createQuery("select max(e.sal)
from Employee e");
        list=q.list();
        System.out.println(list);
        //terminate session factory, otherwise
program won't end

   HibernateUtil.getSessionAnnotationFactory().c
lose();
   }

}
```

Parameter Binding in Hibernate

Parameter binding is the process of binding a Java variable with an HQL statement.Hibernate parameter binding is like prepared statements in any normal SQL language.

A simple select query Without parameter binding.

```
String name="Mukesh";
Query query = session.createQuery("from Employee
where employeeName = '"+name+"' ");
```

There is two types of query parameter binding in the Hibernate,positional parameter and named parameter.Hibernate recommend to use the named parameters since it is more flexible and powerful compare to the positional parameter.

1) Named parameters

In named parameters query string will use parameters in the variable name that can be replaced at runtime.The actual value

will be substituted at runtime using the setParameter() method.

You define a named place holder by writing the place holder name after a colon (:) as :holder name and use it within a setXXX method without the colon (:) as "holder name".

setParameter

The setParameter is smart enough to discover the parameter data type for you.

```
query query = session.createQuery("from Student
where studentId = :id ");
query.setParameter("id", 5);
```

setString

You can use setString to tell Hibernate this parameter date type is String.

```
String hql = "from Stock s where s.stockCode =
:stockCode";
List result = session.createQuery(hql)
.setString("stockCode", "1234")
.list();
```

The above code tells Hibernate to insert an object of type String in the query. You can also use setInteger,setBoolean etc for the corresponding types.

setProperties

By using setProperties you can pass an object into the parameter binding. Hibernate will automatic check the object's properties and match with the colon parameter.

```
Stock stock = new Stock();
stock.setStockCode("1234");
String hql = "from Stock s where s.stockCode =
:stockCode";
List result = session.createQuery(hql)
.setProperties(stock)
.list();
```

2. Positional parameters

This approach will use question mark (?) to define a named parameter, and you have to set your parameter according to the position sequence. Example:

```
String hql = "from Stock s where s.stockCode = ?
and s.stockName = ?";
List result = session.createQuery(hql)
.setString(0, "1234")
.setParameter(1, "HUL")
.list();
```

The basic disadvantage of this method is that if you change the orders of the parameters on the query you have to change the parameter binding code. For Example:

```
String hql = "from Stock s where s.stockName = ?
and s.stockCode = ?";
List result = session.createQuery(hql)
.setParameter(0, "HUL")
.setString(1, "1234")
.list();
```

Example File : HibernateMainUsingNamedParameter.java

```
package com.hibernate.main;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.hibernate.model.Employee;
import com.hibernate.util.HibernateUtil;
public class HibernateMainUsingNamedParameter {
   public static void main(String[] args) {

        Employee e2 = new Employee();
        e2.setId(124);
        e2.setName("Aniket");
        e2.setRole("Clerk");
```

```
        e2.setSal(7800);
        Employee e1 = new Employee();
        e1.setId(121);
        e1.setName("Manasi");
        e1.setRole("MD");
        e1.setSal(80222);
                Employee e = new Employee();
                e.setId(100);
                e.setName("Sam");
```

```
e.setRole("Manager");
e.setSal(45000);
        //Get Session
        Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
        //start transaction
        Transaction tx
=session.beginTransaction();
        session.save(e2);
        session.save(e1);
        session.save(e);
        //Using Label
        Query query=session.createQuery("update
Employee e set e.sal=:sal where e.id=:id");
        query.setParameter("sal", 9000);
        query.setParameter("id", 121);
        int i=query.executeUpdate();
        System.out.println("Data Updated
Successfully and "+i+" row changes");
        /Using question Mark Symbol
         query=session.createQuery("delete from
Employee e where e.id=?");
        query.setParameter(0,100);
         i=query.executeUpdate();
         System.out.println("Data Updated
Successfully and "+i+" row changes");
        tx.commit();
```

```
    HibernateUtil.getSessionAnnotationFactory().c
lose();
    }
}
```

Hibernate Criteria API

In Hibernate Query Language (HQL), we manually prepare the HQL queries for reading the data from database. For the better performance, it is always recommended to write a query as tuned query.

In case of HQL, we need to prepare the tuned queries. Hibernate will only translate HQL into SQL and then executes it, but it will not tune the queries. Here (HQL) the responsibility of tuning the queries on developer.

Instead of writing the HQL queries and tuning them explicitly, we can use **Hibernate Criteria API.**

In Hibernate Criteria API, there is no need to create a query. Instead, hibernate itself will prepare a tuned query. So that we can get better performance with criteria while reading the data from database.

Query tuning is required only while selecting the data, so hibernate Criteria API is for select operations only.

Hibernate Criteria is an interface, it is a simplified API for retrieving entities. We can obtain a reference of Criteria interface by calling the createCriteria() method on session by passing the object of a pojo class.

```
Criteria criteria =
session.createCriteria(Employee.class);
```

As we discussed in HQL example, we can read the data from database in three forms

- Reading complete entity
- Reading partial entity
- Reading partial entity with single column.

Example Reading Complete Entity

```
Criteria crit =
session.createCriteria(Employee.class);
        List list = crit.list();

        Iterator it = list.iterator();

        while (it.hasNext()) {
            Employee emp = (Employee) it.next();
            System.out.println("Employee : " +
emp.toString());

        }
```

Hibernate Criteria Reading the partial Entity :

To read the partial entity, hibernate criteria provides us, Projection interface.

By using the Projection interface, we can set the columns as parameters, which we want to read.

```
    Projection projection =
Projections.property("salary");
```

Finally, we need to add the projection to criteria.

```
    criteria.setProjection(projection);
```

If we want to read the multiple columns, we can create the multiple Projection instances like below :

```
Projection projection =
Projections.property("salary");
Projection projection2 =
Projections.property("departmentId");
Projection projection3 =
Projections.property("employeeName");
```

Get the ProjectionsList object and add all projections to projectionslist object like below :

```
ProjectionList pList =
Projections.projectionList();
pList.add(projection);
pList.add(projection2);
pList.add(projection3);
```

Adding condition to Hibernate Criteria :

To add a condition to criteria, hibernate provides us Criterion interface. We can obtain criterion object by calling the static methods of Restrictions class. To add the criterion object to criteria, we call add() method like below :

```
Criteria crit =
session.createCriteria(Employee.class);
Criterion criterion =
Restrictions.eq("departmentId", 101);
crit.add(criterion);
```

The above statements represents, get the employee records from database, whose department id is 101;

Example file : HibenateMainWithCriteria.java

```
package com.hibernate.main;
import java.util.Iterator;
import java.util.List;
```

```
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.criterion.Criterion;
import org.hibernate.criterion.Expression;
import
org.hibernate.criterion.LogicalExpression;
import org.hibernate.criterion.Order;
import org.hibernate.criterion.Projection;
import org.hibernate.criterion.Projections;
import org.hibernate.criterion.Restrictions;
import com.hibernate.model.Employee;
import com.hibernate.util.HibernateUtil;
public class HibenateMainWithCriteria {
   public static void main(String[] args) {
```

```
Employee e2 = new Employee();
        e2.setId(124);
        e2.setName("Aniket");
        e2.setRole("Clerk");
        e2.setSal(7800);
        Employee e1 = new Employee();
        e1.setId(121);
        e1.setName("Manasi");
        e1.setRole("MD");
        e1.setSal(80222);
        Employee e = new Employee();
            e.setId(100);
           e.setName("Sam");
        e.setRole("Manager");
        e.setSal(45000);
        Session session =
HibernateUtil.getSessionAnnotationFactory().getC
urrentSession();
        //start transaction
        Transaction tx
=session.beginTransaction();
        session.save(e2);
```

```
        session.save(e1);
        session.save(e);
        Criteria cr;
        cr =
session.createCriteria(Employee.class);
        List<Employee> list = cr.list();
        Iterator<Employee> iter =
list.iterator();
        while(iter.hasNext()) {
            System.out.println(iter.next());
        }
```

```
System.out.println("----------------------------
-------------");
        //Use of Criteria With Restrictions
        cr =
session.createCriteria(Employee.class);
        cr.add(Restrictions.like("name","A%"));
        list  = cr.list();
        iter = list.iterator();
        while(iter.hasNext()) {
            System.out.println(iter.next());
        }
        System.out.println("--------------------
--------------------");
        //Use of Criteria with and Expression
        cr =
session.createCriteria(Employee.class);
        cr.add(Expression.between("sal", 2000,
8000));
        list = cr.list();
        iter = list.iterator();
                while(iter.hasNext()) {
            System.out.println(iter.next());
        }
        System.out.println("--------------------
--------------------");
```

```
        //Criteria with the use of Criterion
concepts with the Restrictions with the use of
order
        Criteria
criteria=session.createCriteria(Employee.class);
        Criterion crt= Restrictions.idEq(new
Integer(124));
        crt=Restrictions.eq("sal", new
Integer(7800));
        criteria.add(crt);
        criteria.addOrder(Order.asc("sal"));
        list=criteria.list();
        Iterator iterator=list.iterator();
```

```
while(iterator.hasNext())
        {
        System.out.println(iterator.next());
}
System.out.println("-----------------------------
-------------");
//To put AND and OR in Criteria
Criterion cr1=Restrictions.between("sal", new
Integer(7000),new Integer(90000));
Criterion cr3=Restrictions.eq("name", "Manasi");
LogicalExpression
expression=Restrictions.or(cr3,cr1);
        criteria.add(expression);
List list1=criteria.list();
   Iterator iterator1=list1.iterator();
        while(iterator1.hasNext())
        {
        System.out.println(iterator1.next());
        }
}
}
```