Sathya Technologies, Ameerpet

HIBERNATE

By: Raghu

ECLIPSE

Overview

Eclipse is an Integrated Development Environment (IDE) for Java and other programming languages like C, C++, PHP, and Ruby etc.

Today it is the leading development environment for Java with a market share of approximately 65%.

The Eclipse IDE can be extended with additional software components. Eclipse calls these software components as plug-ins.

Several Open Source projects and companies have extended the Eclipse IDE or created standalone applications on top of the Eclipse framework.

Eclipse Licensing

Eclipse platform and other plug-ins from the Eclipse foundation is released under the Eclipse Public License (EPL). EPL ensures that Eclipse is free to download and install. It also allows Eclipse to be modified and distributed.

Downloading Eclipse

You can download eclipse from http://www.eclipse.org/downloads/. The download page lists a number of flavours of eclipse.

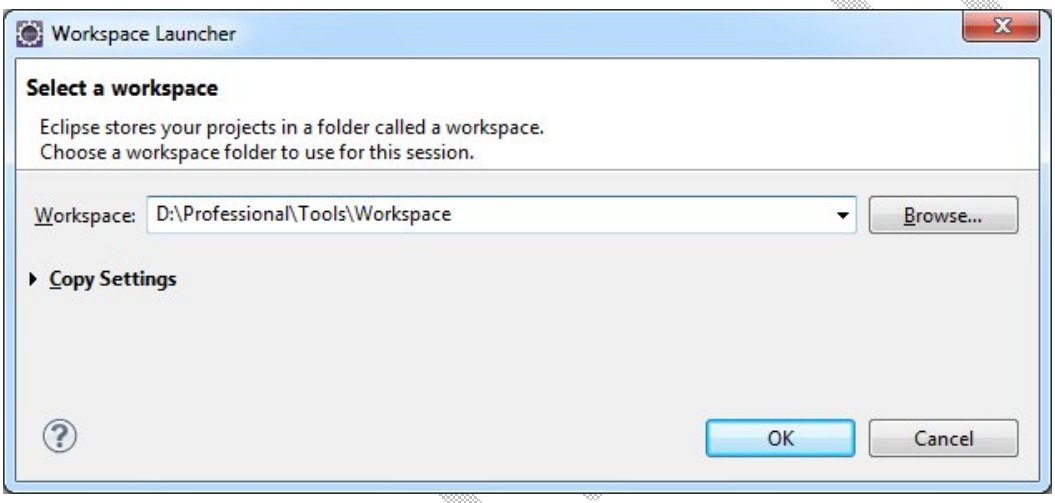
Prerequisites for Running Eclipse

Eclipse is written in Java and will thus need an installed JRE or JDK in which to execute.

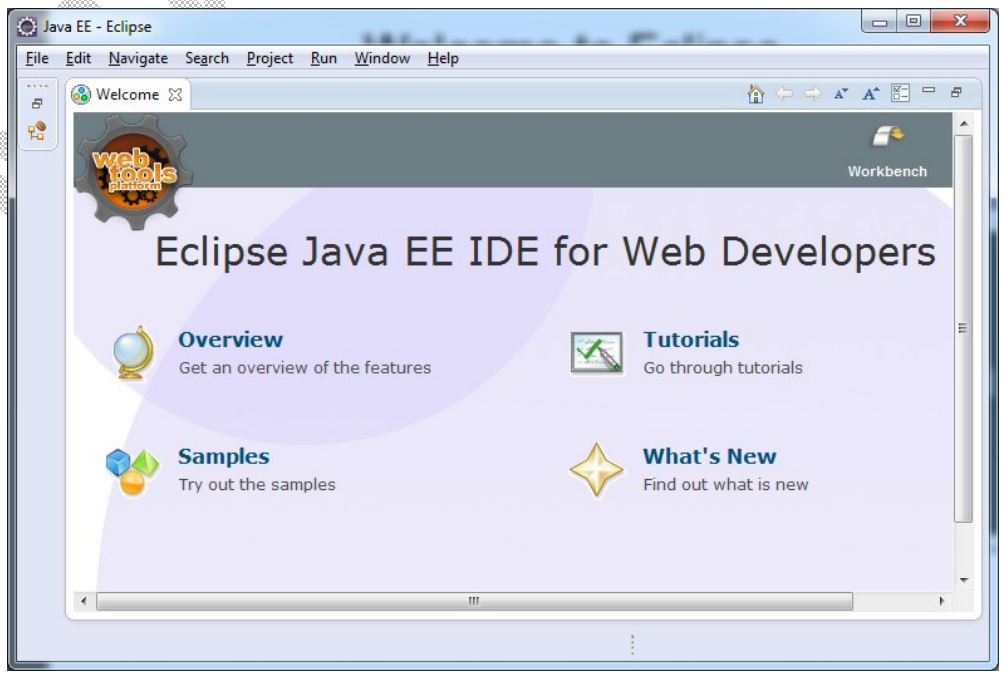
Launching Eclipse

On the windows platform, if you extracted the contents of the zip file to c:\, then you can start eclipse by using c:\eclipse\eclipse.exe

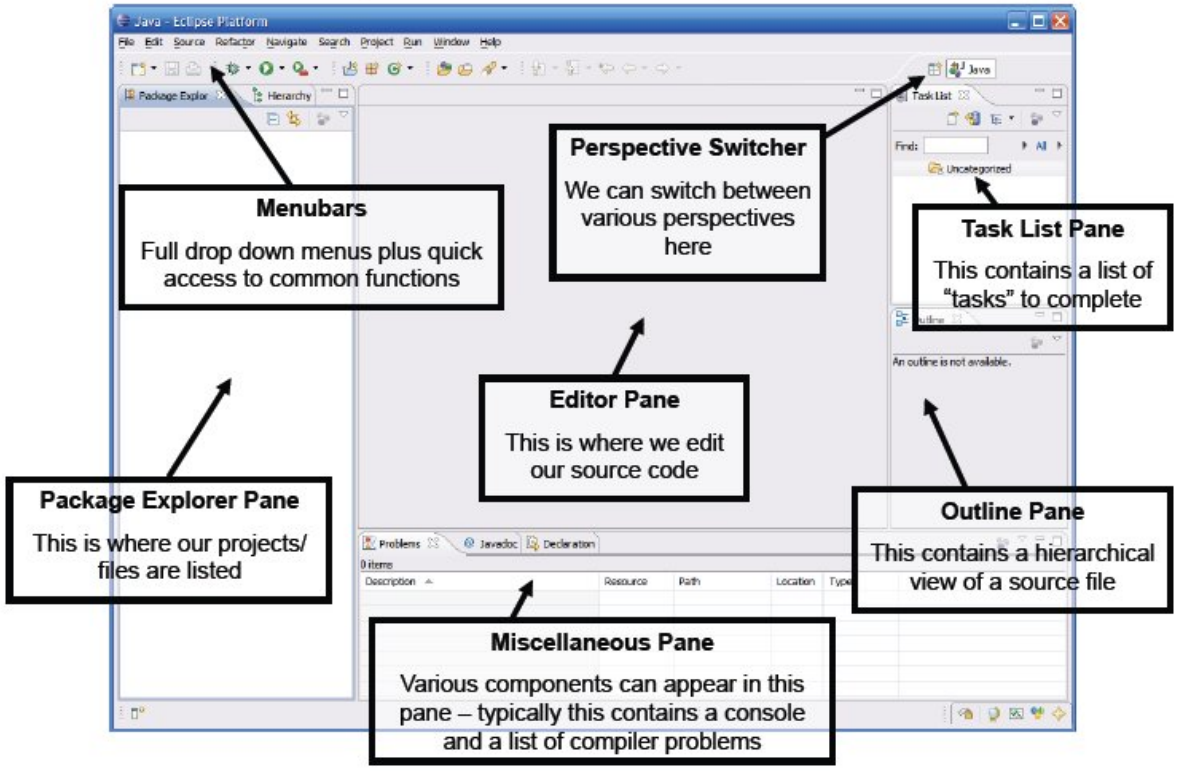
When eclipse starts up for the first time it prompts you for the location of the workspace folder. All your data will be stored in the workspace folder. You can accept the default or choose a new location.



The first time you launch Eclipse, you will be presented with a welcome screen. From here you can access an overview to the platform, tutorials, sample code, etc...



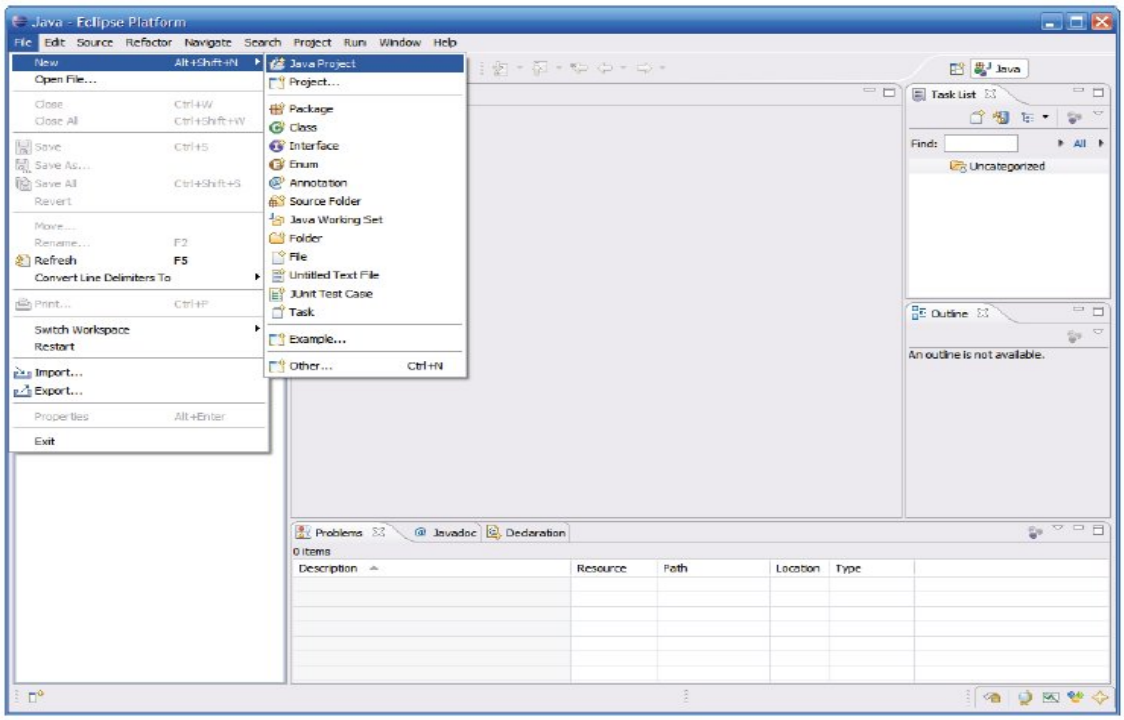
Eclipse IDE Components



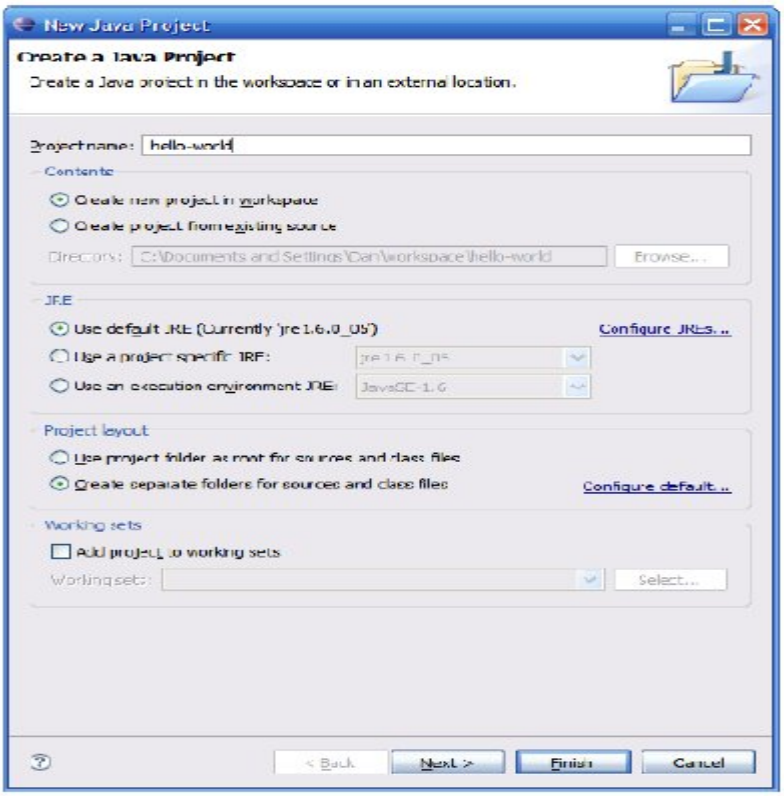
Creating a New Project

- All code in Eclipse needs to live under a project

- To create a project: File →New → Java Project



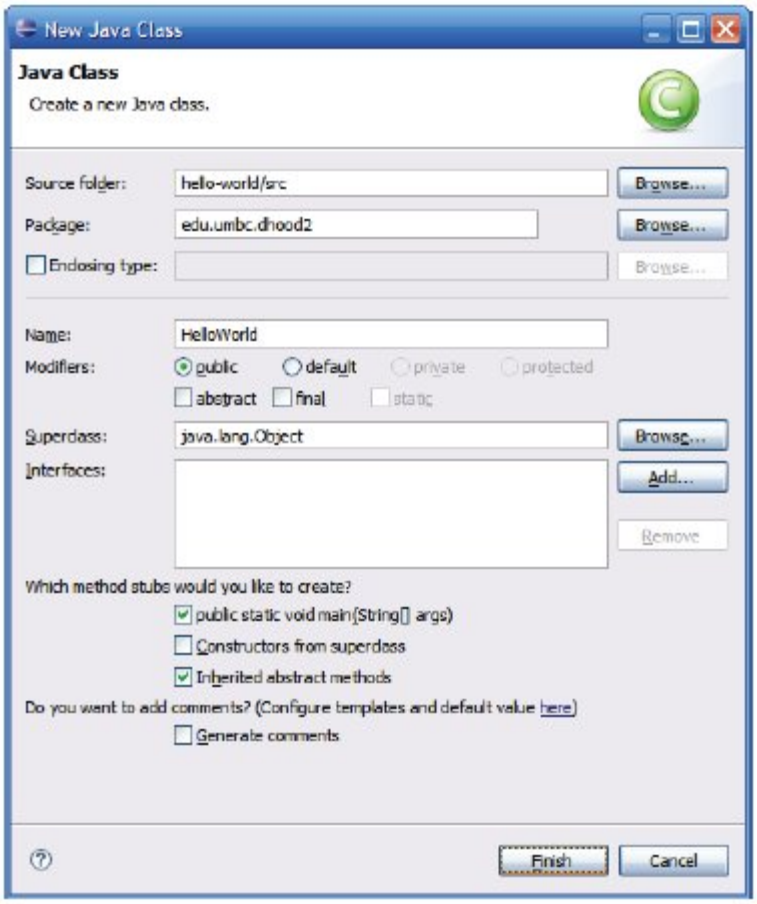
- Enter a name for the project, then click Finish



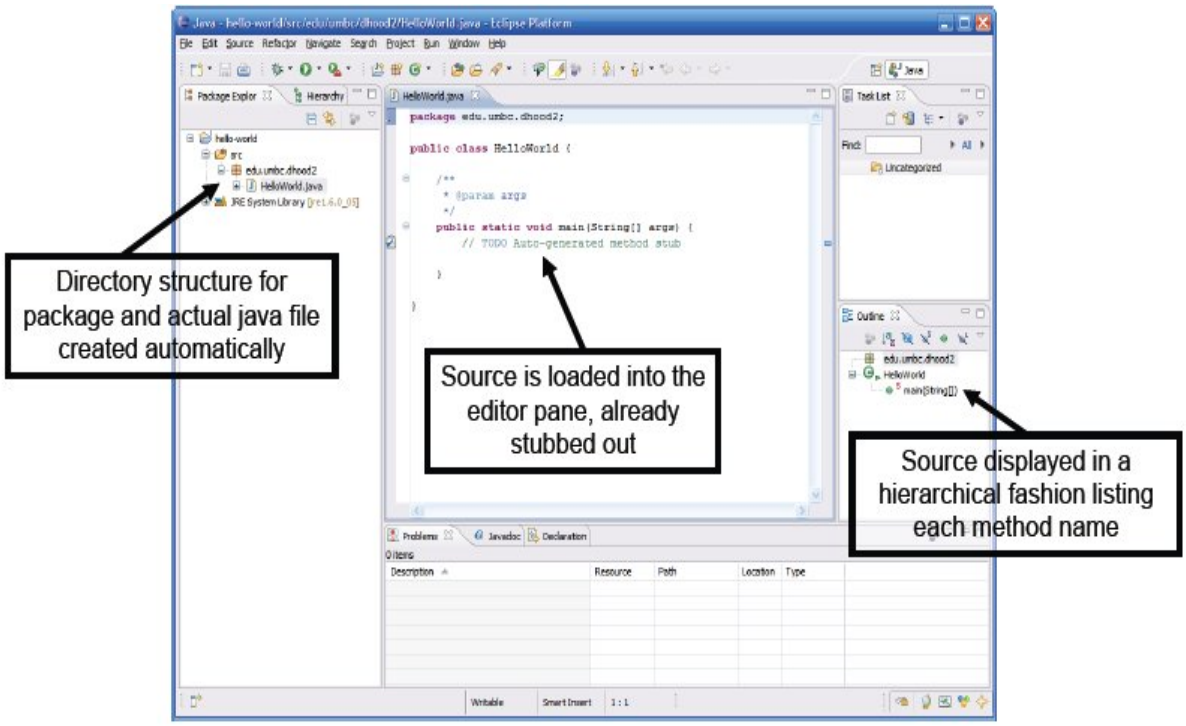
- The newly created project should then appear under the Package Explorer.
- Eclipse automatically creates a folder to store your source code in called “src”.

Creating a Class

- To create a class, simply click on the New button, then select Class.
- This brings up the new class wizard, from here you can specify the Package, Class name, Super class, whether or not to include a main() method etc.
- Fill in necessary information then click Finish to continue.

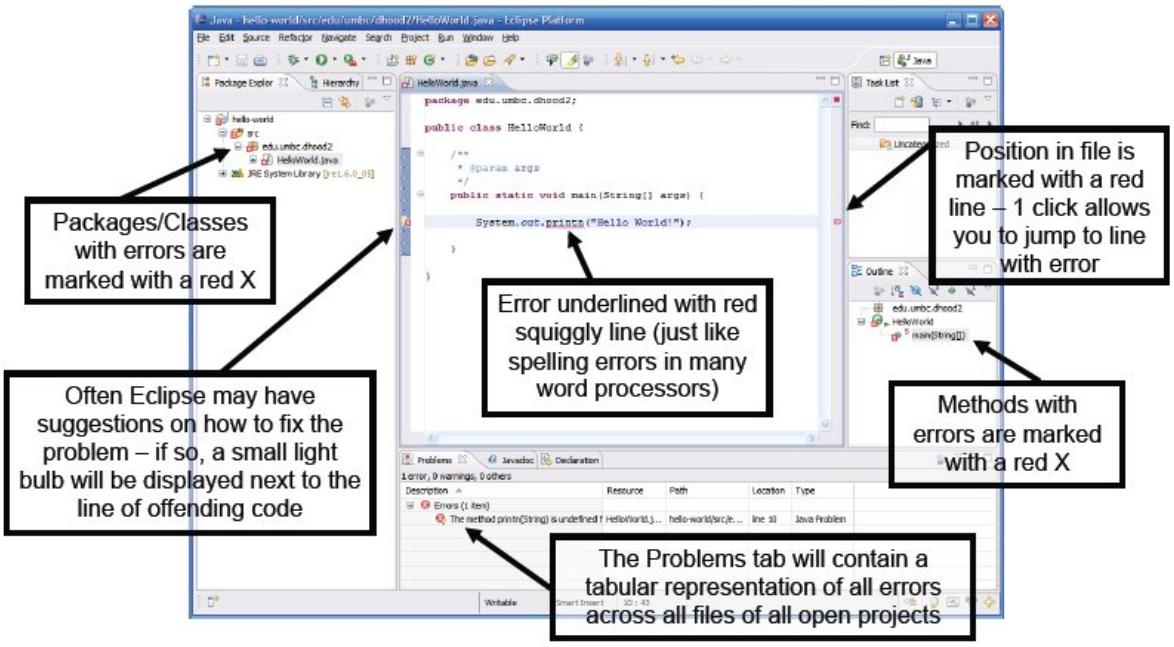


The class is created as shown below:



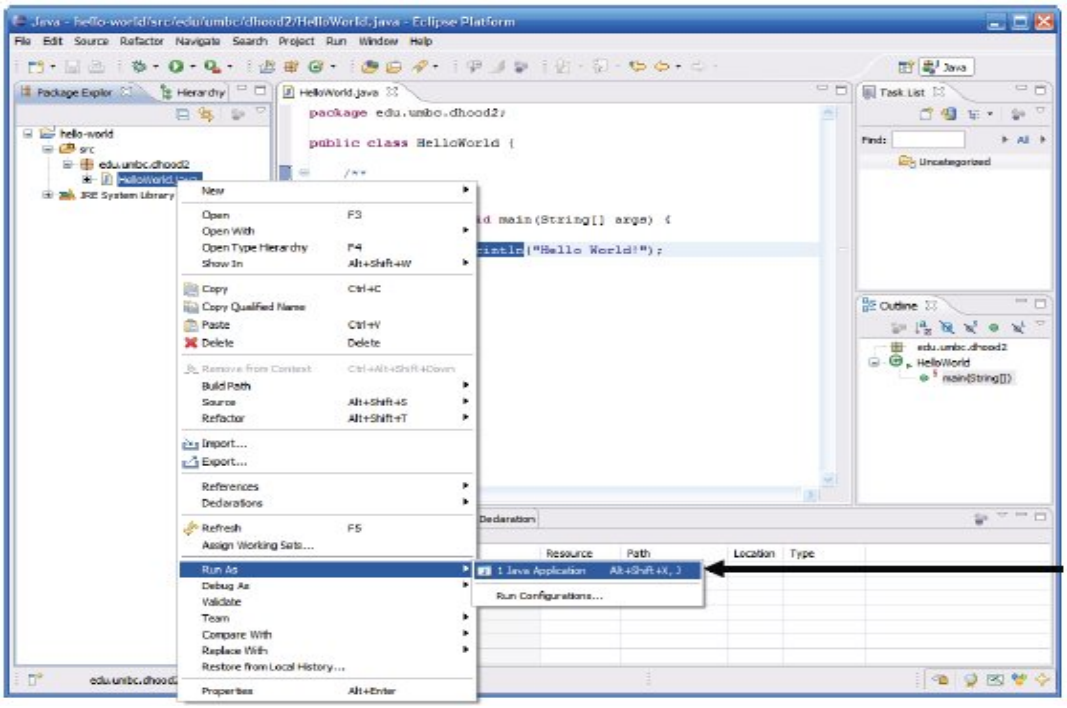
Compiling Source Code

- One big advantage/feature of Eclipse is that it automatically compiles your code in the background.
- No longer need to go to the command prompt and compile code directly.
- Iterative development is the best approach to developing code, but going to command prompt to do a compile can interrupt the normal course of development.



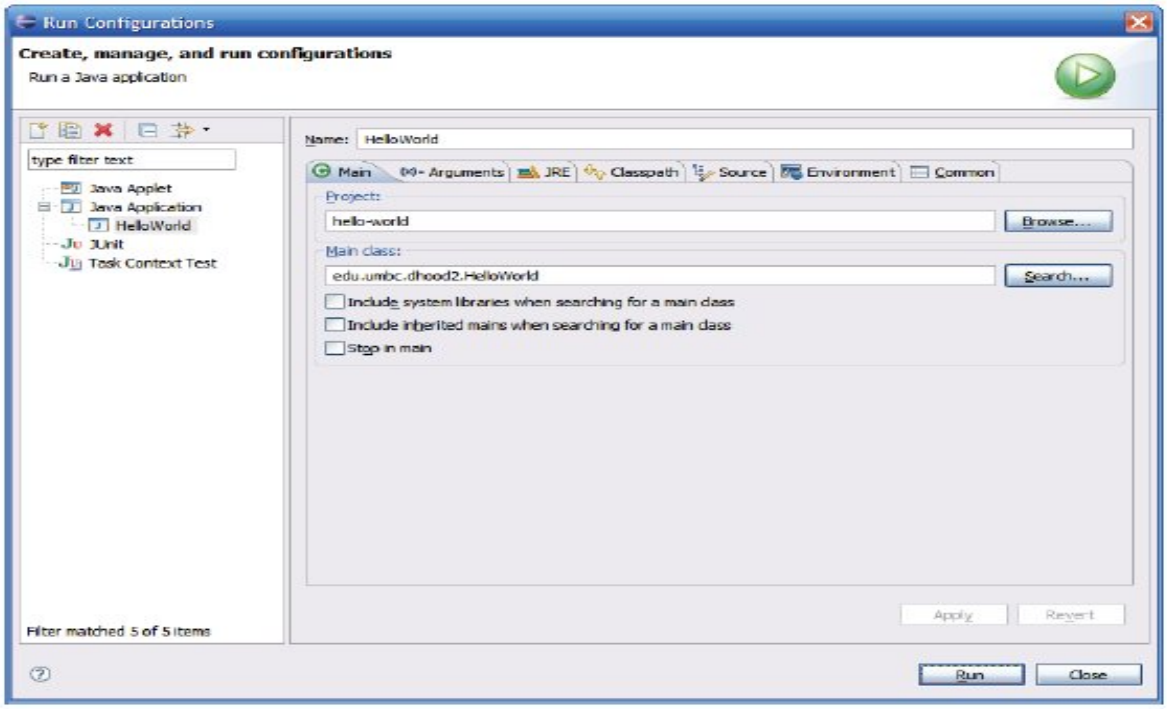
Running Code

- An easy way to run code is to right click on the class and select Run As → Java Application.
- The output of running the code can be seen in the Console tab in the bottom pane.



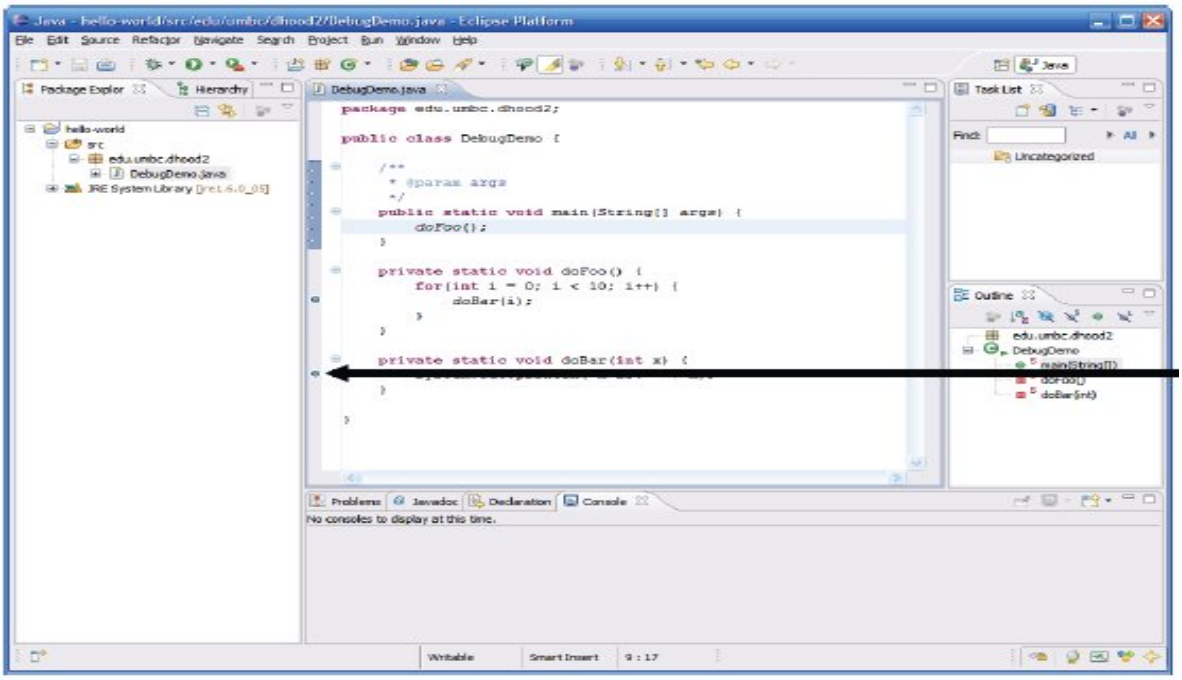
- Advanced options for executing a program can be found by right clicking the class then clicking Run As → Run Configurations

- Here you can change/add any of JVM arguments, Command line arguments, Class path settings, Environment, variables etc.

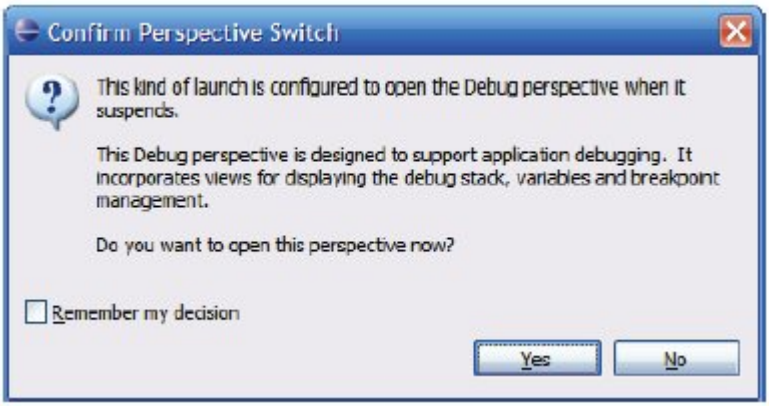


Debugging Code

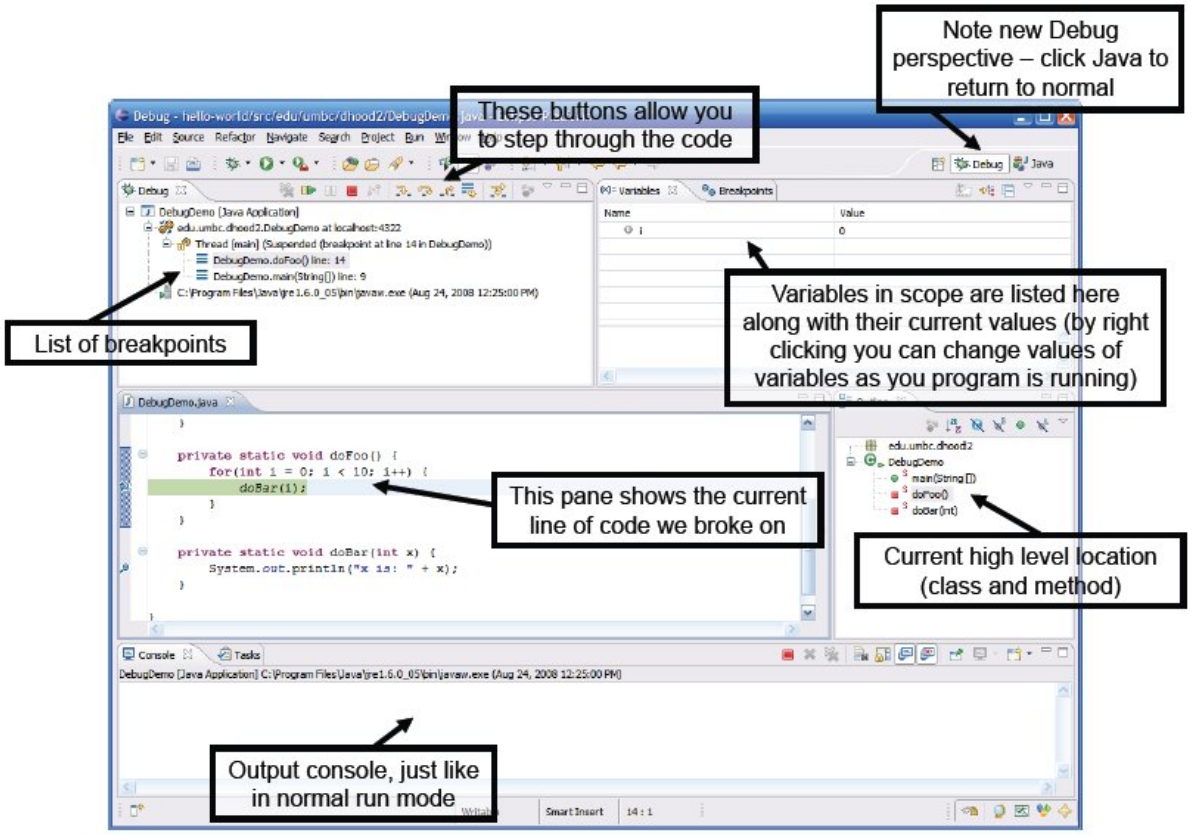
- Eclipse comes with a pretty good built-in debugger.
- You can set break points in your code by double clicking in the left hand margin break points are represented by these blue bubbles.



- An easy way to enter debug mode is to right click on the class and select Debug As → Java Application.
- The first time you try to debug code you will be presented with the following dialog.
- Eclipse is asking if you want to switch to a perspective that is more suited for debugging, click Yes.
- Eclipse has many perspectives based on what you are doing (by default we get the Java perspective).



Debug Perspective



Below are some of the important features of Eclipse:

Import Organization

- Eclipse can automatically include import statements for any classes you are using, just press Control + Shift + o (letter o).
- If the class is ambiguous (more than one in the API) then it will ask you to select the correct one.
- Import statements automatically included and organized, you can organize imports to clean them up at any time.

Context Assist

- If you are typing and press a "." character and pause a second, eclipse will show you a list of all available methods for the class.
- Get context assist at any time by pressing Control + Space.

Javadoc Assist

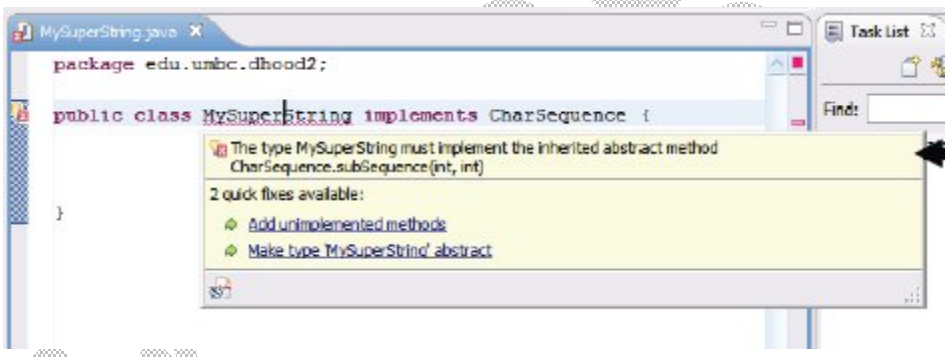
- Eclipse can also help generate javadoc comments for you, simply place the cursor before the method and then type "/**" then Enter.
- Eclipse will automatically generate a javadoc header for the method all stubbed out with the parameters, return type and exceptions.

Getter/Setter Generation

- Eclipse can automatically generate getters and setters for member of a class.
- To generate getters and setters, right click in the main pane, and then select Source →Generate Getters and Setters.

Add Unimplemented Methods

- Eclipse can also stub out methods that need to be present as a result of implementing an interface.
- You can use the quick fix light bulb to add the interfaces unimplemented methods to the class.



Exception Handling

- The eclipse will also pickup on unhandled exceptions.
- By clicking on the quick fix light bulb, Eclipse can suggest what to do to handle the exception.
- Eclipse can automatically add a "throwsdeclaration" to the method signature.
- Alternately, Eclipse can also wrap the code inside a try/catch block.

Local History

- The eclipse maintains a local history of file revisions which can be accessed by right clicking on the class, then selecting Compare With→ Local History.
- Previous saved revisions are displayed in the History pane, double click a revision to view in the built-in diff viewer.

Create Jar Files

The Jar File wizard can be used to export the content of a project into a jar file.

To bring up the Jar File wizard:

- In the Package Explorer select the items that you want to export. If you want to export all the classes and resources in the project just select the project.
- Click on the File menu and select Export.
- In the filter text box of the first page of the export wizard type in "JAR".
- Under the Java category select JAR file
- Click on Next

In the JAR File Specification page:

- Enter the JAR file name and folder
- The default is to export only the classes. To also export the source code Click on the "Export Java source files and resources" check box.
- Click on Next to change the JAR packaging options
- Click on Next to change the JAR Manifest specification
- Click on Finish

Building a Java Project

A project can have zero or more builders associated with it. A java project is associated with a java builder. To see the builders associated with a project:

- In the Package Explorer view right click on the project and select Properties
- On the tree in the left hand side click Builders

The java builder is responsible for compiling the java source code and generating classes. The java builder is notified of changes to the resources in a workspace and can automatically compile java code.

Locating and Installing Plug-ins

The Eclipse platform which provides the foundation for the Eclipse IDE is composed of plug-ins and is designed to be extensible using additional plug-ins.

Several hundreds of plug-ins are available. Each plug-in adds more functionality to Eclipse. You can locate a plug-in that provides certain functionality by searching the Eclipse Market place (http://marketplace.eclipse.org/).

- From within the Eclipse IDE you can search the market place by using the Eclipse Marketplace dialog which can be invoked by clicking on the Help menu and selecting Eclipse Marketplace.
- You can also install a plug-in by using the Install New Software menu item accessible from the Help menu (Help > Install New Software).

Tips & Tricks

- Tip #1 - Discovering Shortcut Keys
  Press Ctrl + Shift + L to open a widget that shows all the shortcut keys.

- Tip #2 - Content Assist
  In the Java editor press Ctrl + Space to see a list of suggested completions. Typing one or more characters before pressing Ctrl + Space will shorten the list.

- Tip #3 - Parameter Hint
  When the cursor is in a method argument, press Ctrl + Shift + Space to see a list of parameter hints.

- Tip #4 - Camel Case Support in Code Completion
  Code completion supports camel case patterns. Entering NPE and pressing Ctrl + Space will propose NullPointerException and NoPermissionException

- Tip #5 - Creating Getters and Setters
  Click on Source > Generate Getter and Setter to open the wizard that allows you to generate getter and setter methods.

- Tip #6 - Generating hashCode() and equals() methods
  Click on Source > Generate hashCode () and equals () to generate this methods for a Java class

- Tip #7 - Adding code around a block of code
  Select a block of code and press Alt+Shift+Z to see a menu of items like if statement, for loop, try/catch etc that can enclose the selected block of code.

- Tip #8 - Locating a matching bracket
  Select an opening or closing bracket and press Ctrl+Shift+P to find its matching bracket.

- Tip #9 - Smart Javadoc
  Type '/**' and press Enter to automatically adds a Javadoc comment stub.

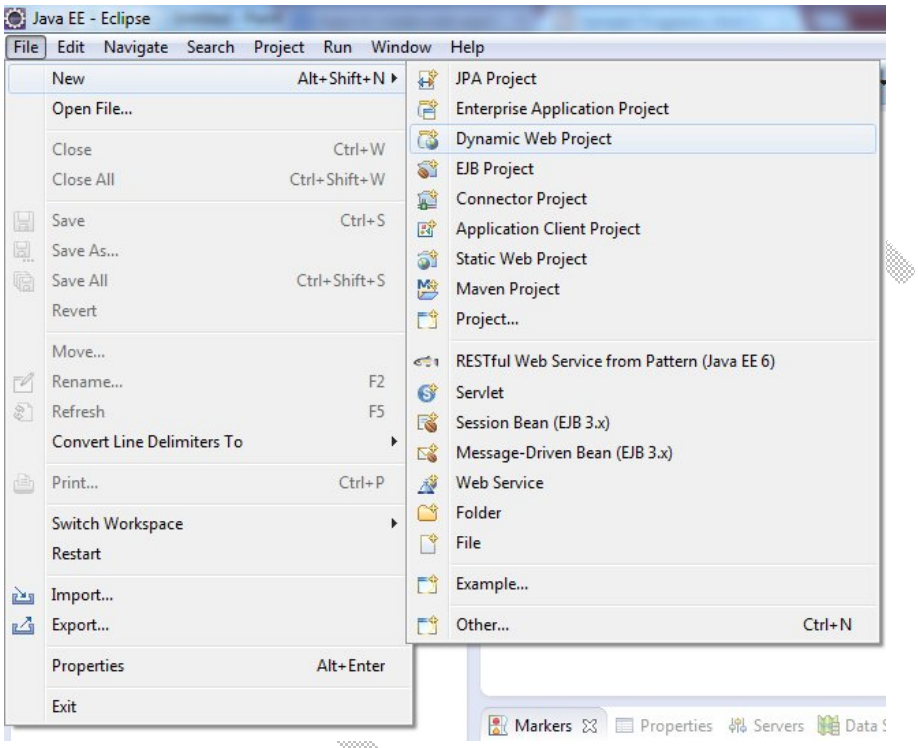
- Tip #10 - Organizing Imports
  Press Ctrl+Shift+O to organize all the imports.

- Tip #11 - Activating the Menu bar
  Press F10 to activate the Menu bar

- Tip #12 - Making a view/editor active
  Press Ctrl+F7 to see a list of open views and editor area and switch to one of them.

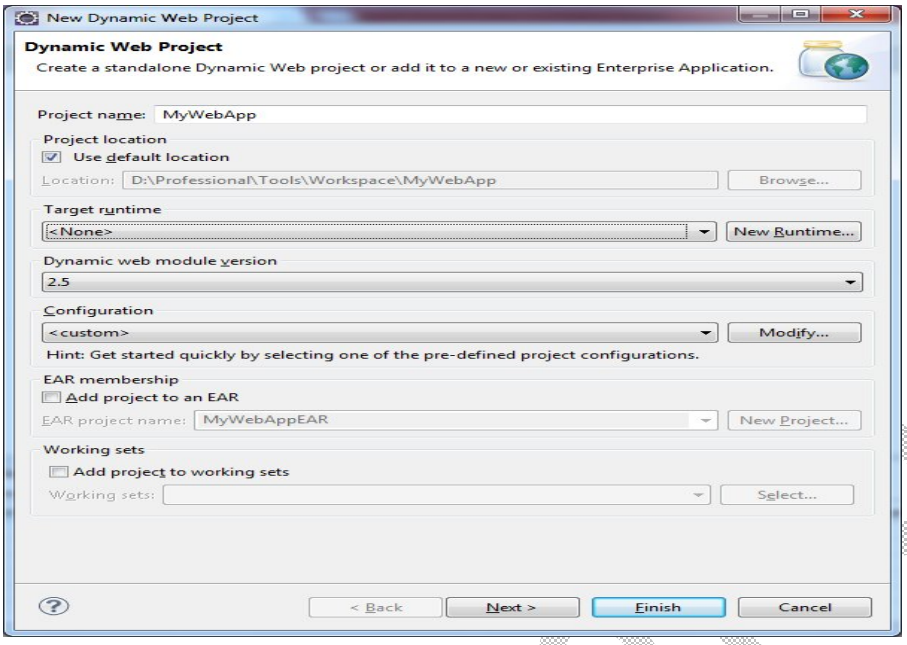
Creating Web Application

1. Launch Dynamic Web Project Wizard.
2. If you are in the Java EE perspective then you can simply do it form File menu by going to File -> New -> Dynamic Web Project.

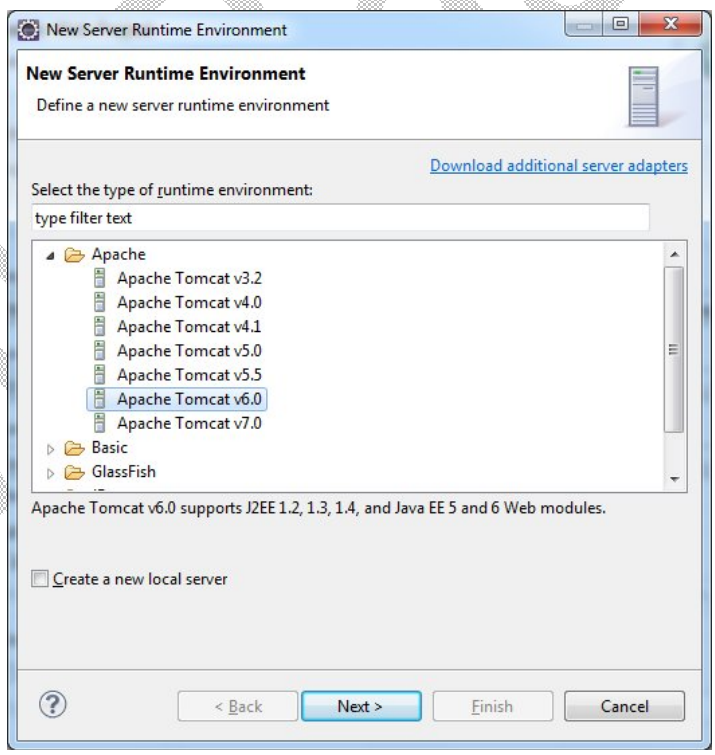


3. That will show the Dynamic Web Project wizard.

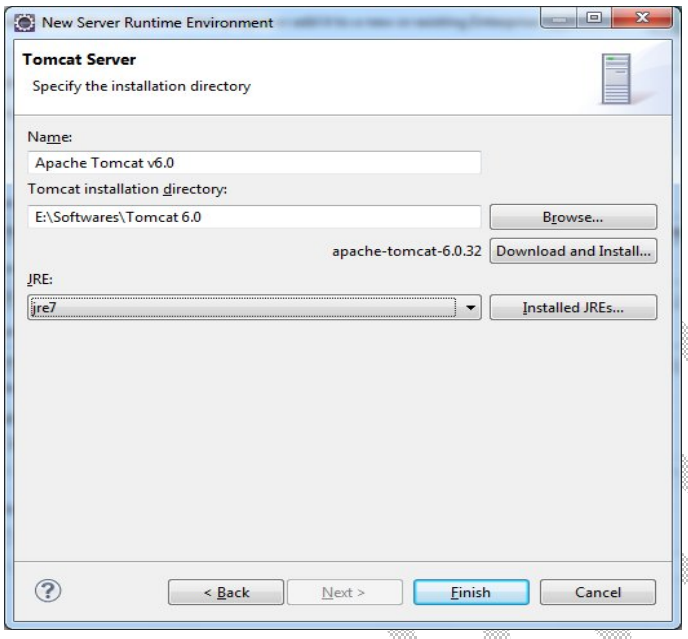
4. Enter the Project Name



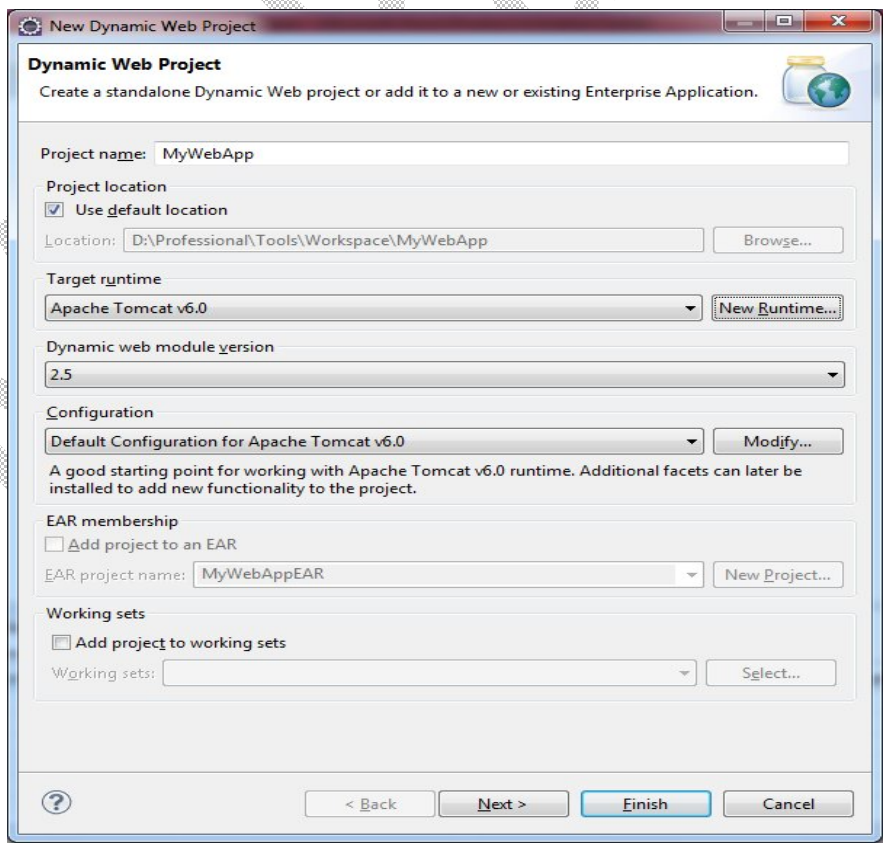
5. Select the Target Runtime from the list if does not exist then add new target runtime by clicking New Runtime button. Here from the list of the list of Runtime select your server's runtime e.g. I selected Apache Tomcat v6.0.



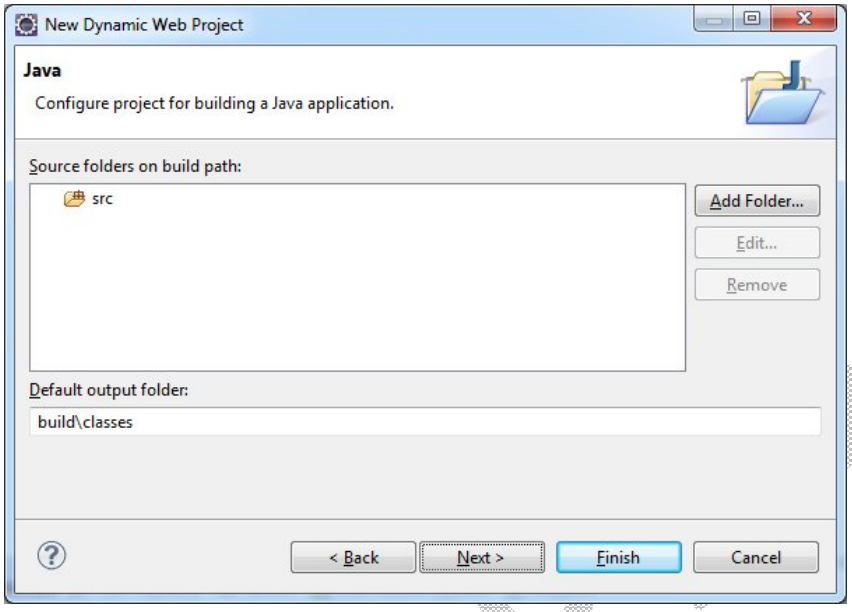
6. Click next to specify the server configuration, where you can specify the name, location and JRE for the server runtime and click on Finish button



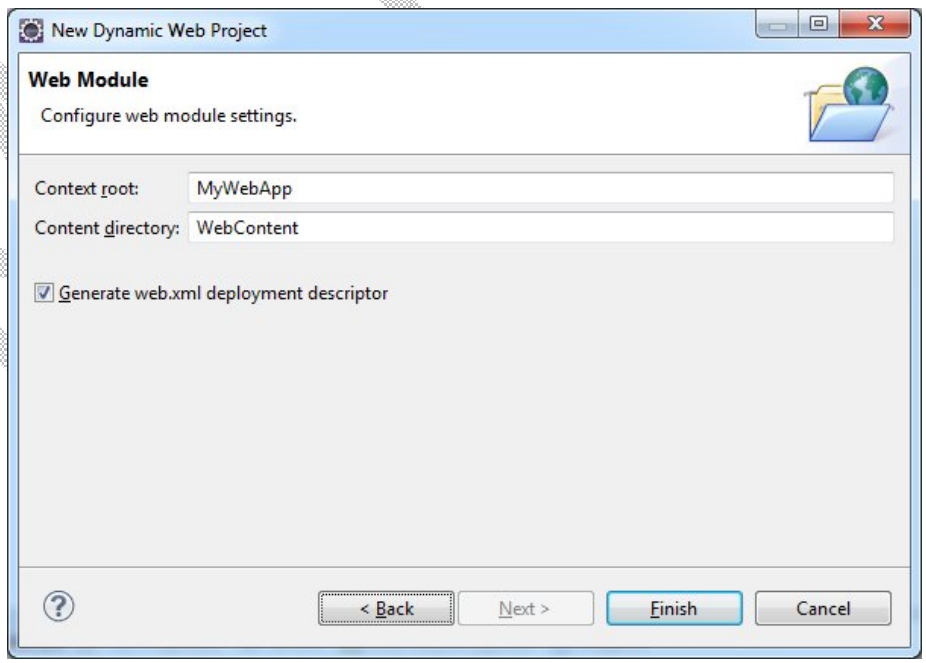
7. Now you will come back to first page of the Dynamic Web Project wizard. Here select Dynamic Web module version to 2.5 or higher and leave all the other parameters as default values though you can change them as per your need.



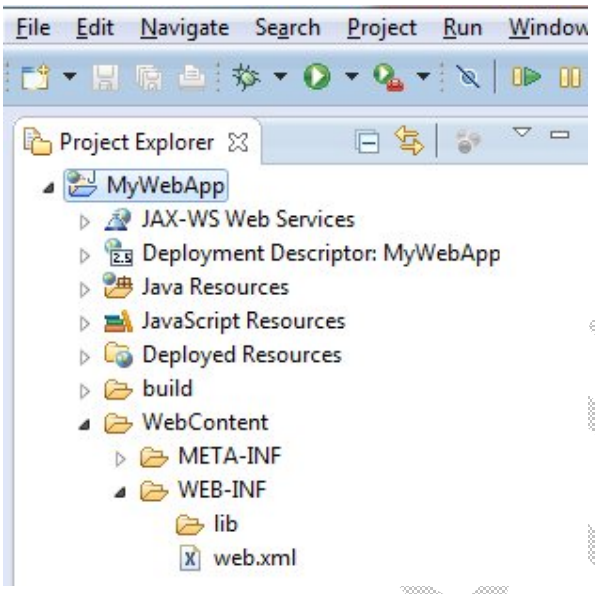
8. Click on the Next button to configure source location and Default output folder similar to Java Project.



9. By default source location would be src change that to WebContent/WEB-INF/src and by default the Default output folder would be build/classes change that to WebContent/WEB-INF/classes.
10. Then click on Next button to configure Web Module for Dynamic Web Project. Here you can configure Context root, Context Directory and select check box to Generate web.xml deployment descriptor

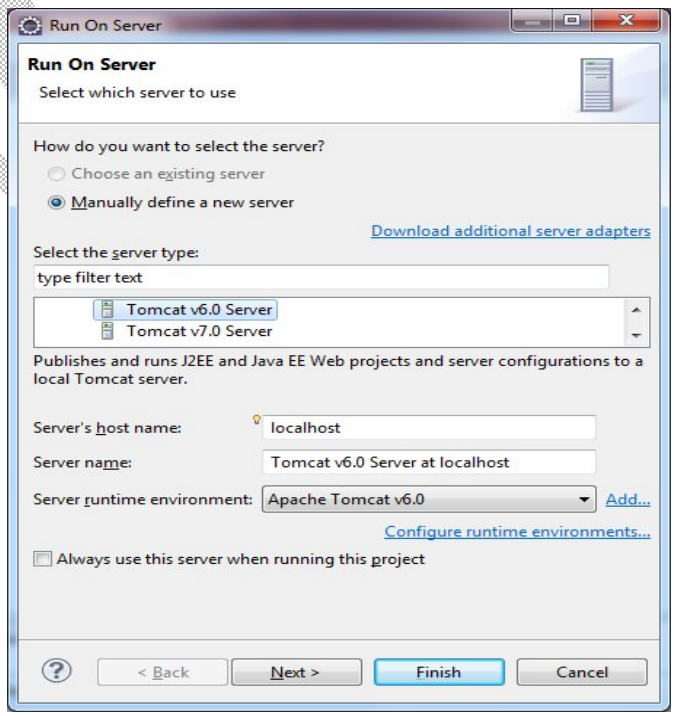


11. Now hit the finish button and Dynamic Web Project gets created with following structure.

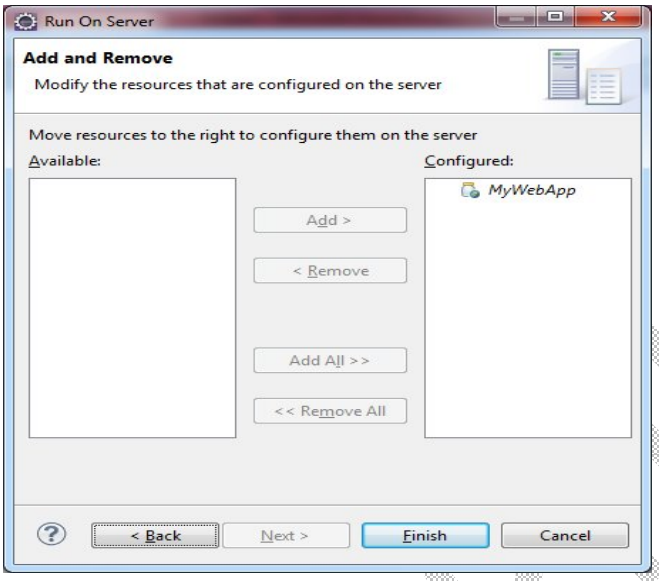


12. Create "index.jsp" in WebContent folder.
13. Run the application by right clicking on project and select "Run on Server" option.

14. In "Run on Server" window click next button.



15. Click finish button.



The Eclipse will create below folders/files in dynamic web project:

Web Deployment Descriptor

The standard web application deployment descriptor (the web.xml file).

JavaSource

Contains the project's Java source code for classes, beans, and servlets. When these resources are added to a Web project, they are automatically compiled and the generated files are added to the WEB-INF/classes directory. The contents of the source directory are not packaged in WAR files unless an option is specified when a WAR file is created.

WebContent folder

The mandatory location of all Web resources, including HTML, JSP, graphic files, and so on.

META-INF

This directory contains the MANIFEST.MF file, which is used to map class paths for dependent JAR files that exist in other projects in the same Enterprise Application project..

WEB-INF

Based on the Sun Microsystems (now Oracle) Java Servlet 2.3 Specification, this directory contains the supporting Web resources for a Web application, including the web.xml file and the classes and lib directories.

/classes

This directory is for servlets, utility classes, and the Java compiler output directory. The classes in this directory are used by the application class loader to load the classes. Folders in this directory will map package and class names, as in: /WEB-INF/ classes/ com/ pack/ servlets/ MyServlet.class.

/lib

The supporting JAR files that your Web application references. Any classes in .jar files placed in this directory will be available for your Web application.

INTRODUCTION

A Java program always produces results after execution (after running) of a ".class "file. But this result is sorted in a temporary location called RAM(Random Access Memory). Here RAM is temporary memory location.

To store Data permanently, to access some other time, In Java Sun Micro System introduced an API called java.io in JDK 1.0(January 23, 1996). This is used to communicate with any file system and independent of OS.

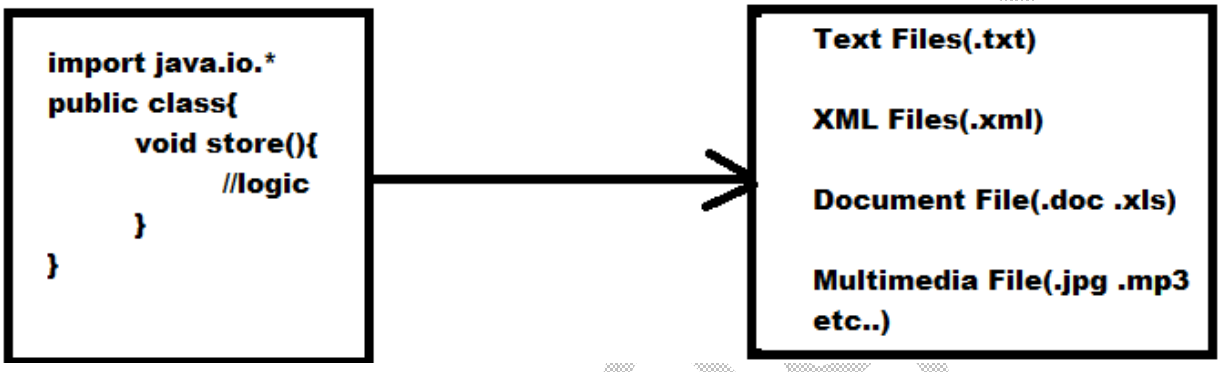


Fig: 1.0 Java program interacting File System

Draw Backs of File System:

- Data Redundancy: We cannot handle redundant data(duplicate records) in file system.
- Security: File System can be accessed directly by every person easily. Providing high level security is not possible.
- Limited Size: Storing data in a file always limited (Maximum file size depends on OS).
- Difficulty in Accessing Data:
  It is not easy to retrieve information using a conventional file processing system. Convenient and efficient information retrieval is almost impossible using conventional file processing system.

And so on.....

So, to overcome the problems of File System DBMS and RDBMS Concepts are introduced.

Advantages of DBMS:

- Controlling Data Redundancy
- Sharing of Data over the network
- Integration of one database with other.
- Data Security
- Backup and Recovery Procedures
  And so on...

Evaluation of JDBC:

In java (JDK 1.1 February 19, 1997) Sun Micro System introduced a Concept Java Database Connectivity (JDBC) that supports Java program to interact with database.

Java database connectivity (JDBC) is the JavaSoft specification of a standard application programming interface (API) that allows Java programs to access database management systems.

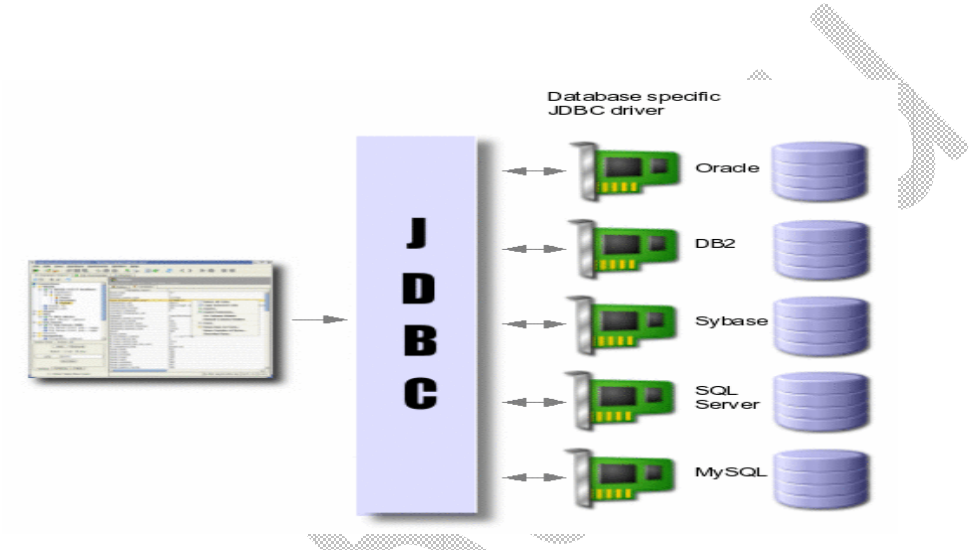


Fig: 1.1 Java Program interacting with database.

Advantages of JDBC:

- It is designed to interact with any kind of database.
- It supports processing SQL/PL-SQL Concepts.
- It supports Transaction Management.
- It supports working with multimedia data types.
- It supports all data types of all databases (Ex: time, date, timestamp etc).
- It supports 4 Different kind of Implementations(4 Types of Drivers)
- It supports Batch updates.
- It supports Annotation based programming (JDBC4.0 onwards).

Limitations of JDBC:

- Programmer also required of learning SQL/PL-SQL concepts. (Some time writing of SQL/PL-SQL program is database dependent).
- All the Exceptions are checked exceptions. So developer must handle these exceptions while writing JDBC program (EX: SQLException).

- JDBC follows traditional RDBMS concept. It does not support OOP based database operations.
- Creating and closing connections must be taken care of JDBC Developer.

Evaluation of ORM:

Object-relational mapping (ORM, O/RM, and O/R mapping) is a Object Oriented concept. It is used to convert RDBMS Data to Object Oriented Data. So that Java programmer can easily work with database to perform persistency operations.

This concept supports of complete OOPs concepts of wiring Persistency programs (ex: encapsulation, inheritance etc..).

Implementations of ORM:

Hibernate
Ibatis
OJB
Torque
Castor
Java Ultra-Lite Persistence
Speedo

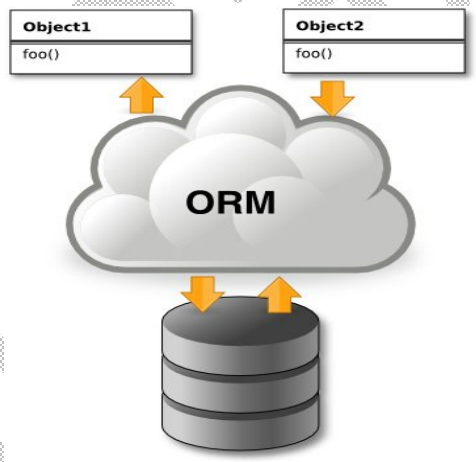


Fig: 1.3 ORM (Low level Architecture)

Advantages of ORM:

- A high-level API for creating, reading, updating, and deleting the persistence objects.
- An object-oriented query language for querying the persistence objects.
- A caching mechanism with proper locking modes.
- Supports Relationships like one-to-one, one-to-many or many-to-one, many-to-many.

HIBERNATE

Hibernate was started in 2001 by Gavin King with colleagues from Cirrus Technologies as an alternative to using EJB2-style entity beans. Its original goal was to offer better persistence capabilities than offered by EJB2 by simplifying the complexities and supplementing missing features.

In early 2003, the Hibernate development team began Hibernate2 releases, which offered many significant improvements over the first release.

JBoss, Inc. (now part of Red Hat) later hired the lead Hibernate developers in order to further its development.

HIBERNATE

Developer(s)	Red Hat
Stable release	4.3.5[1] / April 2, 2014[1]
Development status	Active
Written in	Java
Operating system	Cross-platform (JVM)
Platform	Java Virtual Machine
Type	Object-relational mapping
License	GNU Lesser General Public License
Website	hibernate.org

Gavin King, is the founder of the Hibernate project, a popular object/relational persistence solution for Java, and the creator of Seam, an application framework for Java EE 5. Furthermore, he contributed heavily to the design of EJB 3.0 and JPA

Blog: http://relation.to/Bloggers/Gavin

1. Hibernate is an Object/Relational Mapping framework for Java environments.
2. Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities.
3. It can also significantly reduce development time otherwise spent with manual data handling in SQL and JDBC.
4. Hibernate can certainly help you to remove or encapsulate vendor-specific SQL code.
5. In JDBC all exceptions are checked exceptions, so we must write code in try, catch and throws, but in hibernate we only have Un-checked exceptions, so no need to write try, catch, or no need to write throws.
6. Hibernate has capability to generate primary keys automatically while we are storing the records into database.

7. Hibernate supports caching mechanism by this, the number of round trips between an application and the database will be reduced, by using this caching technique an application performance will be increased automatically.
8. Hibernate supports annotations based programming.
9. Hibernate provided Dialect classes which create Queries Dynamically.
10. Hibernate has its own query language, i.e hibernate query language which is database independent.
11. Hibernate supports collections like List, Set, Map (Only new collections).
12. Hibernate supports Connection pooling.
13. Hibernate Supports below databases.

DB2	Informix
MySQL	Ingres
Sybase	Mckoi SQL
SAP DB	Pointbase
HypersonicSQL	Firebird
Progress	PostgreSQL
Interbase	Oracle
FrontBase	Microsoft SQL Server

Hibernate Architecture:

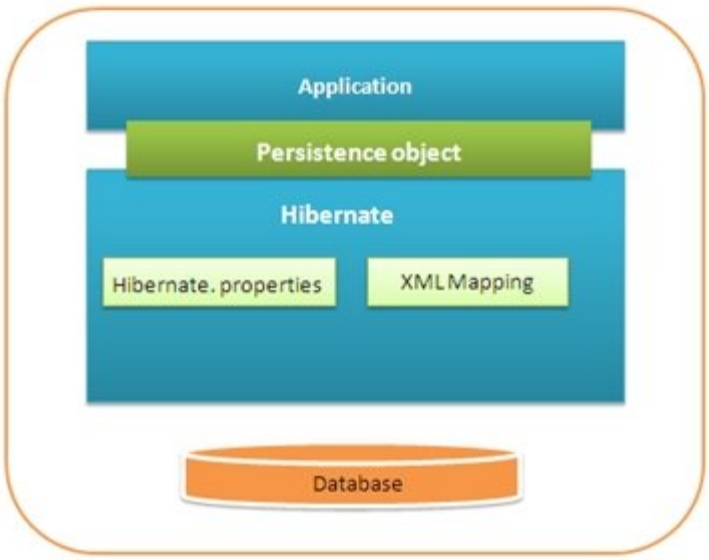


Fig: 2.0 Hibernate Framework Architecture.(Low level)

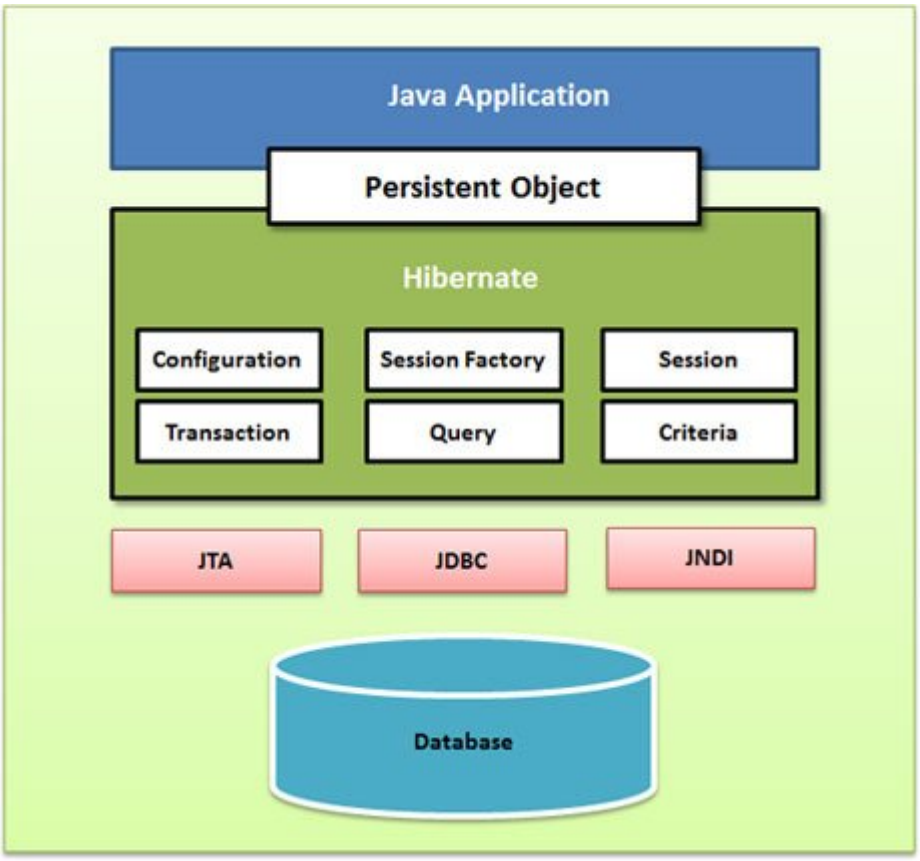


Fig: 2.1 Hibernate Framework Architecture (High Level).

Some Important terminology to know start the Hibernate:

1. POJO
2. Factory Design Pattern
3. XML and DTD
4. Database Configuration Properties

POJOs:

Plain Old Java Object (POJO) is a common java class that follows some rules like

- ✓ POJO class must be declared with public access modifier.
- ✓ It must have a default constructor.
- ✓ All Fields (variable) must be declared as private.
- ✓ For each Filed getXXX and setXXX methods defined.
- ✓ This classes that do not extend or implement some specialized classes and interfaces required by the EJB framework.

***But It can extend Core API classes and interfaces.

Ex: A Sample POJO Class.

```
1  public class Person
2  {
3       private int id;
4       private String name;
5       private int salary;
6
7       public Perrson(){}
8       public int getId(){
9            return id;
10      }
11
12      public String getName(){
13           return name;
14      }
15
16      public int getSalary(){
17           return salary;
18      }
19
20      public void setId(){
21       this.id = id;
22      }
23      public void setName(){
24       this.name = name;
25      }
26      public void setSalary(){
27       this.salary = salary;
28      }
29 }
```

Factory Pattern:

- Factory design pattern in Java one of the core design pattern which is used heavily not only in JDK but also in various Open Source framework such as Spring, Struts and Apache along with decorator design pattern in Java.
- Factory method is used to create different object from factory often refereed as Item and it encapsulate the creation code. So instead of having object creation code on client side we encapsulate inside Factory method in Java.
- One of the best examples of factory pattern in Java is BorderFactory Class of Swing API

This Design pattern is basically used to derive an object creation at runtime.

Example: Factory Pattern:

```
 interface Currency {
        String getSymbol();
}
// Concrete Rupee Class code
class Rupee implements Currency {
        @Override
        public String getSymbol() {
                return "Rs";
        }
}

// Concrete SGD class Code
class SGDDollar implements Currency {
        @Override
        public String getSymbol() {
                return "SGD";
        }
}

// Concrete US Dollar code
class USDollar implements Currency {
        @Override
        public String getSymbol() {
                return "USD";
        }
}
```

```
 // Factroy Class code
class CurrencyFactory {

       public static Currency createCurrency (String country) {
       if (country. equalsIgnoreCase ("India")){
              return new Rupee();
       }else if(country. equalsIgnoreCase ("Singapore")){
              return new SGDDollar();
       }else if(country. equalsIgnoreCase ("US")){
              return new USDollar();
        }
       throw new IllegalArgumentException("No such currency");
       }
}
```

```java
// Factory client code
public class Factory {
	public static void main(String args[]) {
		String country = args[0];
		Currency rupee = CurrencyFactory.createCurrency(country);
		System.out.println(rupee.getSymbol());
	}
}
```

XML and DTD:

XML stands for eXtensible Markup Language.

XML is designed to transport and store data.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
</note>
```

DTD: Document Type Definition is used to specify tags, their attribute details to write an XML file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

DTD Specification

Example DTD:

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

Database Configuration Properties:

To uniquely identify any data base we commonly use properties like username and password. Apart from that few more properties are used while configuration of database in hibernates.

S.No.	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.

Important classes and interfaces of Hibernate

1. Configuration (org.hibernate.cfg)
2. SessionFactory(org.hibernate)
3. Session (org.hibernate)
4. Query (org.hibernate)
5. Criteria (org.hibernate)

Configuration

- The org.hibernate.cfg.Configuration class is the basic element of the Hibernate API that allows us to build SessionFactory.
- That means we can refer to Configuration as a factory class that produces SessionFactory.
- The Configuration object encapsulates the Hibernate configuration details such as connection properties, dialect and mapping details described in the Hibernate mapping documents which are used to build the SessionFactory.
- The Configuration object takes the responsibility to read the Hibernate configuration and mapping XML documents, resolves them and loads the details into built-in Hibernate objects.

SessionFactory

- The org.hibernate.SessionFactory interface provides an abstraction for the application to obtain Hibernate Session objects.
- Creating a SessionFactory object involves a huge process that includes the following operations:
  - Start the cache (second level)
  - Initialize the identifier generators
  - Pre-compile and cache the named queries (HQL and SQL)
  - Obtain the JTA TransactionManager

Note: It is generally recommended to use single SessionFactory per JVM instance. There is no problem in using a single SessionFactory for an application with multiple threads also.

Session

- The org.hibernate.Session is a central interface between the application and Hibernate framework. The Hibernate Session operates using a single JDBC Connection.
- The Session interface provides an abstraction for Java application to perform CRUD (Create, Read,Update&Delete) operations on the instances of mapped persistence classes.

Query

- The org.hibernare.Query interface provides an abstraction for executing the Hibernate query and retrieving the results.

Criteria

- The org.hibernate.Criteria is the central interface for using Criterion API and it provides a simplified API for using a programmatic approach of searching the persistent objects, alternative to the HQL and SQL.

XML Files In Hibernate:

Hibernate Configuration

- Hibernate allows us to add configuration parameters and mapping files location to the configuration object.

Hibernate Mappings

- The ORM implementations basically perform various services like creating, reading, updating, and deleting the persistence objects.

- To perform this, ORM frameworks requires metadata information for the persistent types (i.e., metadata for each persistent class).
- This metadata is described by using the mapping XML documents or annotations.

Perform Basic Operations Using Hibernate:

- To use Hibernate in an application to perform various persistent operations we need to implement some basic steps.
- The basic steps involved in using the Hibernate in a Java application are:
    1. Prepare Configuration object
    2. Build Session Factory
    3. Obtain a Session
    4. Perform persistence operations
    5. Close the Session

1. Prepare Configuration Object

- The org.hibernate.cfg.Configuration object encapsulates the hibernate configuration details such as connection properties, dialect and mapping details described in the Hibernate mapping documents, which are used to build the Session Factory.

- To do this we can use configuration mapping documents in a programmatic approach or use an XML configuration file in a declarative approach or annotations.

- We can create a Hibernate configuration object using its no-argument constructor, as shown in the following code snippet:

Configuration cfg=new Configuration ();
cfg.configure ();

- The configure() method locates the hibernate.cfg.xml file in order to load the configuration parameters and the mapping files location.
- We can also use a XML configuration file with a different file name as shown in the code snippet below:

Configuration cfg=new Configuration ();

cfg.configure("myconfigs/adminmodule.cfg.xml");

2. Build Session Factory

- After successfully preparing the Configuration object by setting all the configuration parameters and mapping documents location, the configuration object is used to create the SessionFactory.
- To do this we use the buildSessionFactory() method of configuration, which creates a new SessionFactory using the configuration parameters and mappings in this configuration.

Configuration cfg=new Configuration();

cfg.configure();

SessionFactory factory=cfg.buildSessionFactory();

3. Obtain a Session

- The Hibernate Session works on top of a SessionFactory, meaning a Session is bound to a SessionFactory.
- The SessionFactory object is used to open a Hibernate Session.

Configuration cfg=new Configuration();

cfg.confogure();

SessionFactory factory=cfg.buildSessionFactory();

Session session=factory.openSession();

- The openSession() method is used to open a new Session. However, in some cases, especially in the managed environments, where multiple components are involved in processing the request in a transaction, we want to work with a single Session to execute the persistent operations.
- The getCurrentSession() method can be used in such an environments. The getCurrentSession() method obtains the Session associated with the current JTA transaction; if a Session is not already associated with the current JTA transaction, a new Session will be opened and it will be associated with the JTA transaction.
- If Hibernate has no access to the TransactionManager or if the application is not associated with any transaction, the getCurrentSession() method throws an HibernateException.
- The following code snippet shows the code for location the associated with the current transaction.

Configuration cfg=new Configuration();

cfg.configure();

SessionFactory factory=cfg.buildSessionFactory();

Session session=factory.getCurrentSession();

4. Perform Persistence Operations

- The Session interface provides an abstraction for Java application to perform the basic CRUD (Create Read Update Delete) operations on the instances of mapped persistence classes.

Different convenient methods of Session

Method	Description
Object load(Class entityClass, Serializable id)	Locates the persistent instance of the given entity class with the given identifier, if the persistent instance is not available in the cache then it loads the instance from the database and returns it to caller.
Object load(String entityName, Serializable id)	Locates the persistent instance of the given entity class with the given name and identifier, if the persistent instance is not available in the cache then it loads the instance from the database and returns it to caller.
void load(Class entityClass, Serializable id)	Reads the persistent state located with the given identifier into the given transient instance.
Serializable save(Object obj)	Generates an identifier, associates it to the given transient instance and persist.
void update(Object obj)	Update the persistence instance with the identifier of the given object.
void saveOrUpdate(Object obj)	Either save or update the given instance, depending upon resolution of the unsaved-value checks.

- The methods of Session interface described allows the Java application to use Hibernate for performing basic CRUD operations but most of the time we have a requirement to query for persistent objects.

To support this requirement Hibernate includes two interfaces—Query and Criteria

5. Close the Session

- After we complete the use of Session, it has to be closed to release all the resources such as cached entity objects, collections, proxies, and a JDBC connection.
- That means the Hibernate Session encapsulates two important types of state, one is the cache of the entity objects, and second a JDBC connection.
- To close a Hibernate Session we can use close() method.

```
//get the SessionFactory object reference

Session session=factory.openSession();

//Use session for performing persistence operations

//after the use of session, now closing the session

session.close();
```

Hibernate Libraries:

Hibernate can be downloaded from:
http://www.hibernate.org/downloads
From the download page that appears, choose to download the current latest release of Hibernate Core. I have used Hibernate 3.3.1.GA in all the examples in this tutorial.

The Hibernate Library includes below files:

- From the hibernate-distribution-3.3.1.GA directory
  - hibernate3.jar
- From the hibernate-distribution-3.3.1.GA →lib →required directory
  - antlr-2.7.6.jar
  - jta-1.1.jar
  - javaassist-3.4.GA.jar
  - commons-collections-3.1.jar
  - dom4j-1.6.1.jar
- From the hibernate-distribution-3.3.1.GA →lib →bytecode →cglib directory
  - hibernate-cglib-repack-2.1_3.jar

- From the hibernate-annotations-3.4.0.GA → lib directory
  - slf4j-api.jar
  - slf4j-log4j12.jar
- From the hibernate-annotations-3.4.0.GA →lib → test directory
  - log4j.jar

These files are available in the directory of the Hibernate Core and Hibernate Annotations download.

Project 1: SaveDataToDB

1. Student.java:

```
1  public class Student{
2
3      private int sid;
4      private String sname;
5      private double fee;
6
7      public void setSid(int sid)
8      {
9          this.sid = sid;
10     }
11     public int getSid()
12     {
13         return sid;
14     }
15     public void setSname(String sname)
16     {
17         this.sname = sname;
18     }
19     public String getSname()
20     {
21         return sname;
22     }
23     public void setFee(double fee)
24     {
25         this.fee = fee;
26     }
27     public float getFee()
28     {
29         return fee;
30     }
31 }
```

2. Student.hbm.xml:

```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="Student" table="student">
        <id name="sid" column = "id"/>

        <property name="sname"
                column="name"/>

        <property name="fee" column = "totfee"/>

    </class>

</hibernate-mapping>
```

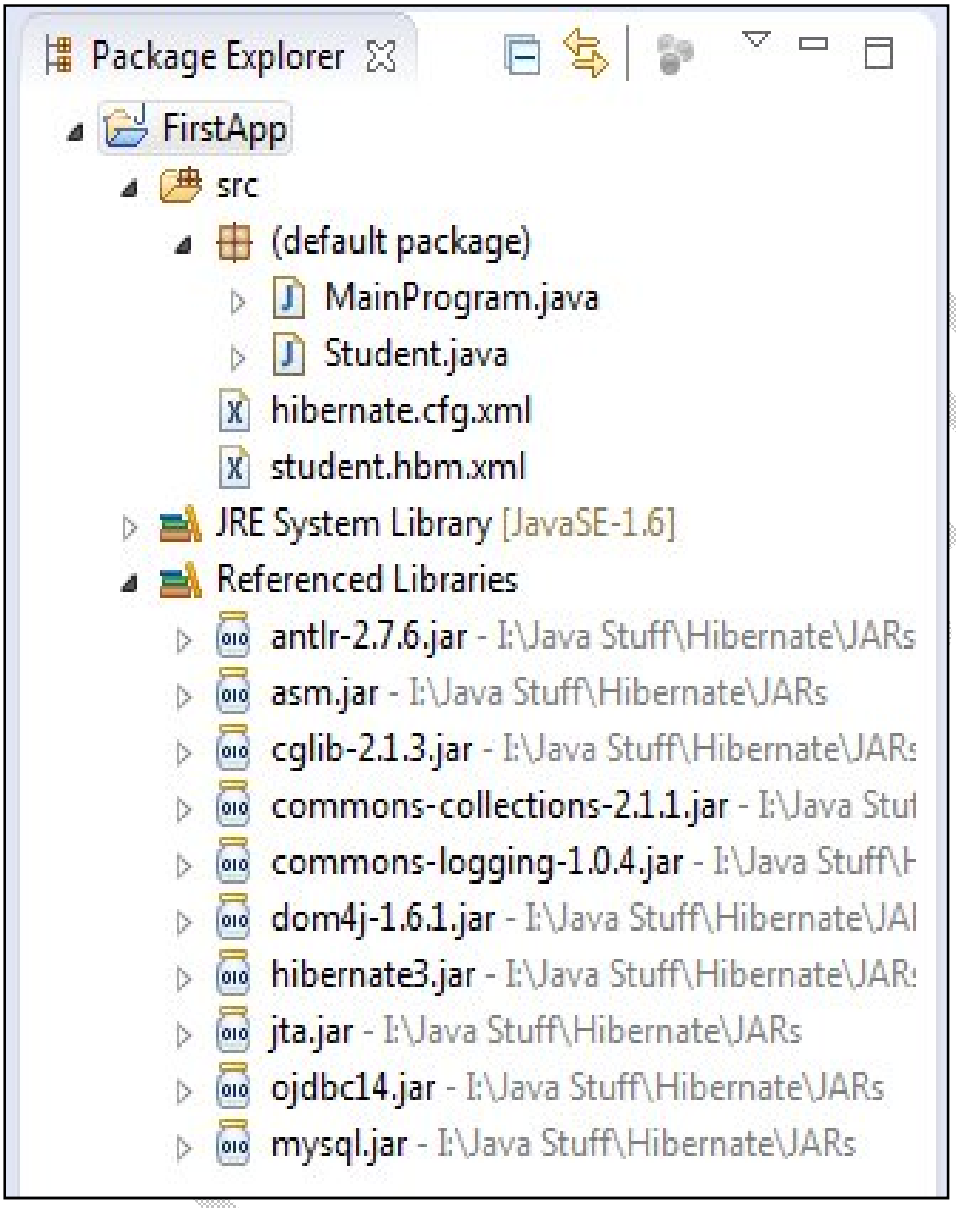
3. hibernate.cfg.xml:

```xml
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/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:@localhost:1521:xe</property>

        <property name = "hibernate.connection.username">system</property>


        <property name = "hibernate.connection.password">manager</property>

        <property name = "hibernate.dialect">org.hibernate.dialect.OracleDialect</property>

        <property name="show_sql">true</property>

        <mapping resource="student.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```java
// insertion of a record into student table

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

class MainProgram
{
    public static void main(String[] args)
    {
        //activate hibernate software
        Configuration cfg = new Configuration();

        //read configuration file
        cfg  = cfg.configure(); //reads "hibernate.cfg.xml"

        // create SessionFactory

        SessionFactory factory = cfg.buildSessionFactory();

        //getting session from session factory

        Session ses = factory.openSession();

        //creation of StudentBean class object

        Student st1 = new Student();

        st1.setSid(101);
        st1.setSname("Venkatesh");
        st1.setFee(5000.55);

        //creation of Transaction object as we
        //are modifying database table

        Transaction tx = ses.beginTransaction();

        ses.save(st1);
        System.out.println("Record Inserted Successfull!");
        tx.commit();
        ses.close();
        factory.close();

    }
}
```

Folder Structure (With Jar Files):



Project 2 : UpdateDBApp

Student.java

```java
public class Student
{
    private int sid;
    private String sname;
    private double fee;
    public void setSid(int sid)
    {
        this.sid = sid;
    }
    public int getSid()
    {
        return sid;
    }
    public void setSname(String sname)
    {
        this.sname = sname;
    }
    public String getSname()
    {
        return sname;
    }
    public double getFee() {
        return fee;
    }
    public void setFee(double fee) {
        this.fee = fee;
    }
```

Student.hbm.xml

```xml
<hibernate-mapping>

    <class name="Student" table="student">

        <id name="sid"/>

        <property name="sname"/>

        <property name="fee" column="sfee"/>

    </class>

</hibernate-mapping>
```

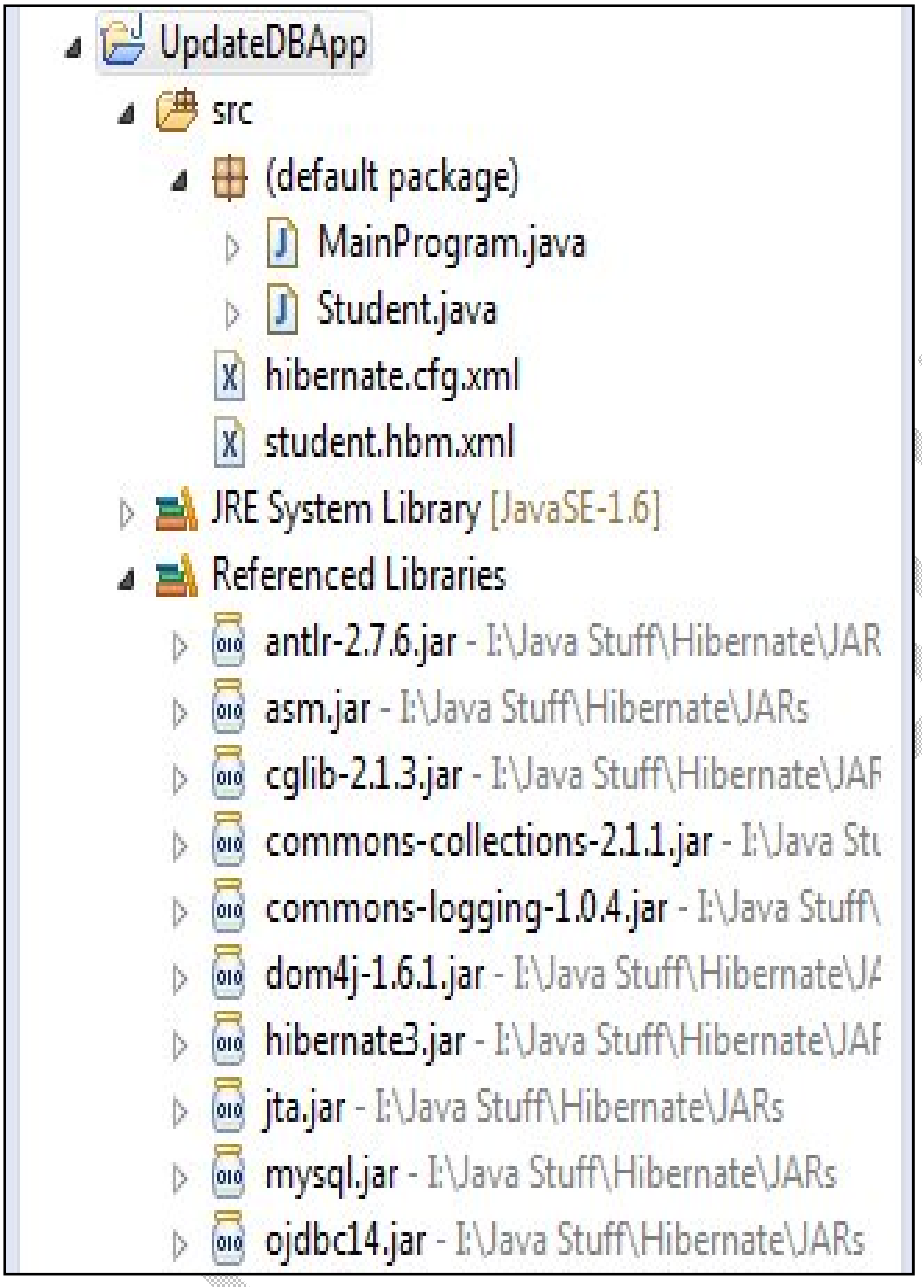
Hibernate.cfg.xml:

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">root</property>
        <property name = "hibernate.connection.password">root</property>
        <property name = "hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <mapping resource="student.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
public class MainProgram
{
    public static void main(String[] args)
    {
        Configuration cfg = new Configuration();
        cfg  = cfg.configure();

        SessionFactory factory = cfg.buildSessionFactory();
        Session ses = factory.openSession();

        Student st1 = new Student();
        st1.setSid(200);//use already existed ID
        st1.setSname("BA");
        st1.setFee(300);

        Transaction tx = ses.beginTransaction();

        ses.update(st1);

        tx.commit();
        ses.close();
        System.out.println("Record Updated Successfully!");
        factory.close();

    }
}
```

Folder Structure:



Project 3 :SaveorUpdate

Student.java

```java
public class Student
{
    private int sid;
    private String sname;
    private double fee;
    public void setSid(int sid)
    {
        this.sid = sid;
    }
    public int getSid()
    {
        return sid;
    }
    public void setSname(String sname)
    {
        this.sname = sname;
    }
    public String getSname()
    {
        return sname;
    }
    public double getFee() {
        return fee;
    }
    public void setFee(double fee) {
        this.fee = fee;
    }
```

Student.hdb.xml

```xml
<hibernate-mapping>

    <class name="Student" table="student">

        <id name="sid"/>

        <property name="sname"/>

        <property name="fee" column="sfee"/>

    </class>

</hibernate-mapping>
```

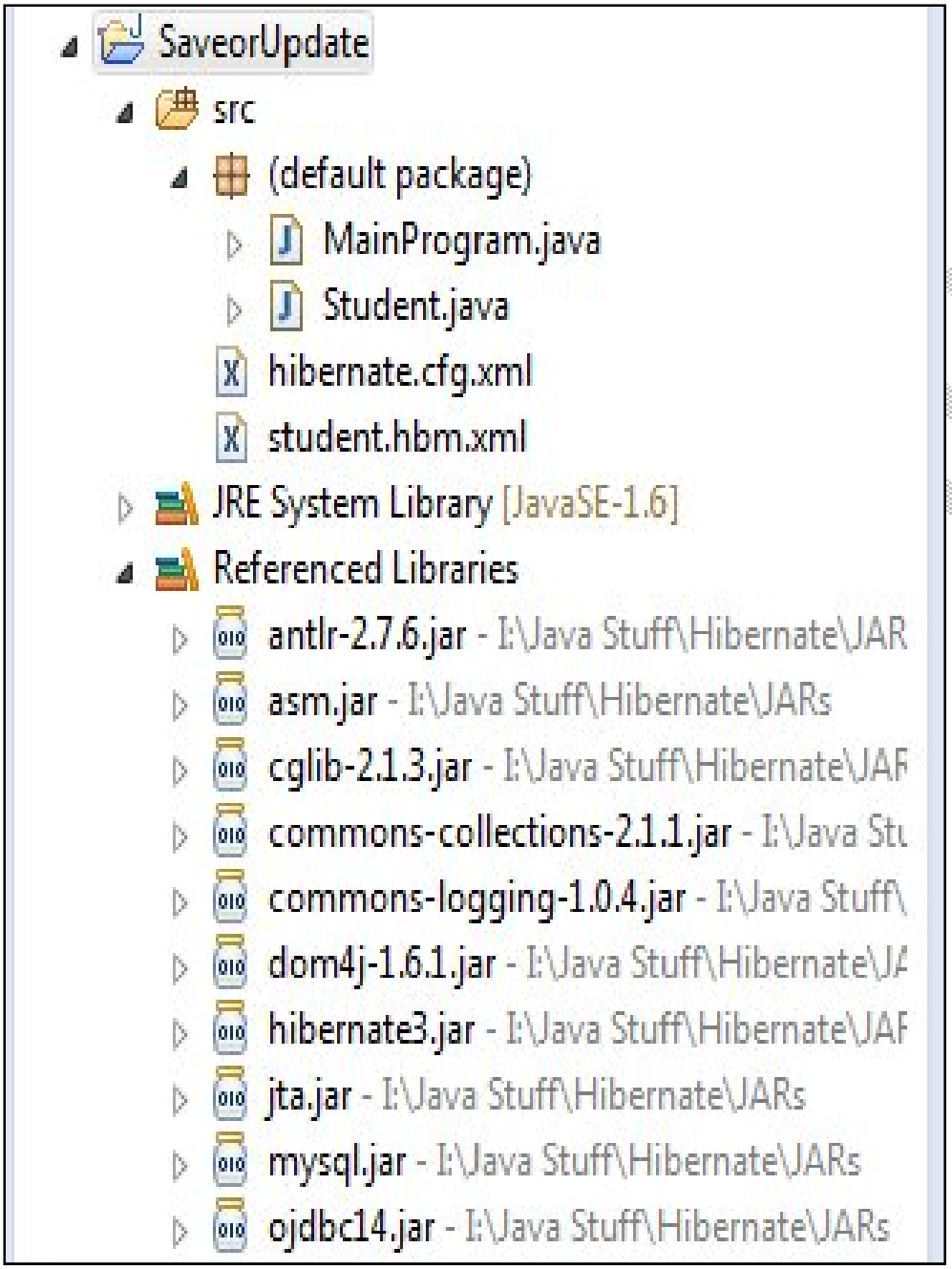
hibernate.hbm.xml

```
<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">root</property>
        <property name = "hibernate.connection.password">root</property>
        <property name = "hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <mapping resource="student.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```
import org.hibernate.Session;
public class MainProgram
{
    public static void main(String[] args)
    {
        Configuration cfg = new Configuration();
        cfg  = cfg.configure();

        SessionFactory factory = cfg.buildSessionFactory();
        Session ses = factory.openSession();

        Student st1 = new Student();
        st1.setSid(200);//use already existed ID
        st1.setSname("Venkatesh");
        st1.setFee(45300);

        Transaction tx = ses.beginTransaction();
        ses.saveOrUpdate(st1);

        tx.commit();
        ses.close();
        System.out.println("Record Updated Successfully!");
        factory.close();

    }
}
```

Folder Structure:



Project 4: GetDatApp

Student.java

```java
public class Student
{
    private int sid;
    private String sname;
    private double fee;
    public void setSid(int sid)
    {
        this.sid = sid;
    }
    public int getSid()
    {
        return sid;
    }
    public void setSname(String sname)
    {
        this.sname = sname;
    }
    public String getSname()
    {
        return sname;
    }
    public double getFee() {
        return fee;
    }
    public void setFee(double fee) {
        this.fee = fee;
    }
```

Student.hbm.xml

```xml
<hibernate-mapping>

    <class name="Student" table="student">

        <id name="sid"/>

        <property name="sname"/>

        <property name="fee" column="sfee"/>

    </class>

</hibernate-mapping>
```

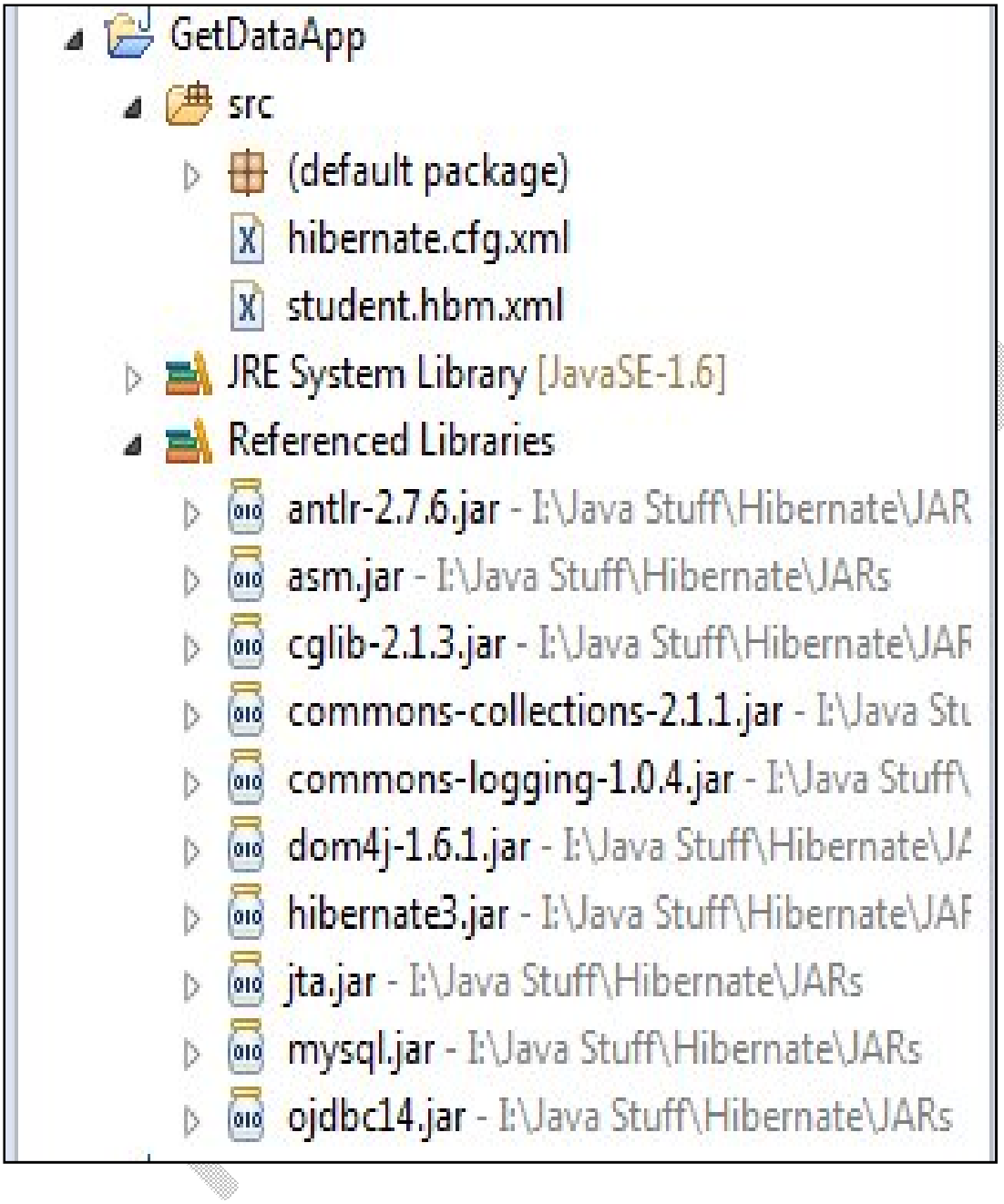
hibernate.hbm.xml

```
<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">root</property>
        <property name = "hibernate.connection.password">root</property>
        <property name = "hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <mapping resource="student.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```
import org.hibernate.Session;
public class MainProgram
{
    public static void main(String[] args)
    {
        Configuration cfg = new Configuration();
        cfg  = cfg.configure();

        SessionFactory factory = cfg.buildSessionFactory();
        Session ses = factory.openSession();

        Student st1;// = new StudentBean();
        st1 = (Student) ses.get(Student.class,200);
        ses.close();

        if(st1 ==  null)
                System.out.println("Record not found");
        else{
            System.out.println("Record Values r: ");
            System.out.println(st1.getSid()+" "+st1.getSname()+
                    " "+st1.getFee());
        }

        factory.close();

    }
}
```

Folder Structure:



Defference between get() and load() :

1. session.load()

It will always return a “proxy” (Hibernate term) without hitting the database. In Hibernate, proxy is an object with the given identifier value, its properties are not initialized yet, it just look like a temporary fake object. If no row found , it will throws an ObjectNotFoundException.

2. session.get()

It always hit the database and return the real object, an object that represent the database row, not proxy. If no row found , it return null.

SQL Dialects

Always set the hibernate.dialect property to the correct org.hibernate.dialect.Dialect subclass for your database. If you specify a dialect, Hibernate will use sensible defaults for some of the other properties listed above. This means that you will not have to specify them manually.

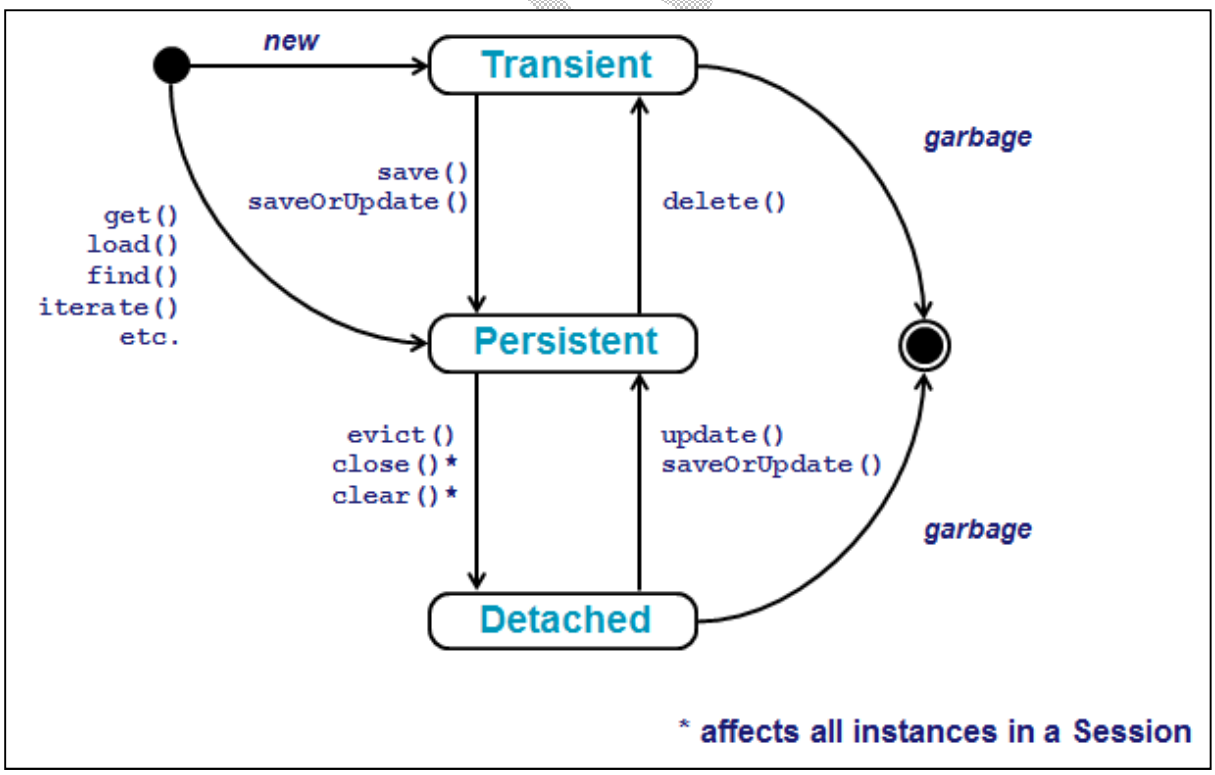
Table :Hibernate SQL Dialects (hibernate.dialect)

RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect

RDBMS	Dialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
Firebird	org.hibernate.dialect.FirebirdDialect

Object State Transition and Session Methods:

- Hibernate works with normal Java objects (POJO) that the application creates using the new operator. These objects are mapped to actual database tables using Hibernate Mapping mechanism.
- An instance of an object that is mapped to Hibernate can be in any one of three different states i.e., Transient, Persistent or Detached.



Transient Object

- Transient Objects are those that are instantiated by the application but not yet made persistent by calling the save() method of the session object.
- This kind of object is instantiated using the new operator but is not immediately persistent. Hence such an object is not associated with any database table row.
- To convert a transient object to persistent state, the save() method of the session object is invoked.

Persistent Object

- Persistent Objects are those that are instantiated by the application and then made persistent by calling the save() method of the session object.
- Persistent objects are always associated with a session and are transactional.
- Persistent objects participate in transactions and their state is synchronized with the database at the end of the transaction. This means when a transaction is committed, the state held in memory is propagated to the database by the execution of the appropriate SQL.

Detached Object

- After a transaction completes, all the associated persistent objects still exists in memory but they lose their association with the session on encountering a session.close().
- Such objects are called Detached Objects. A detached object indicates that its state will no longer be synchronized with database state.
- These objects can be reused in a new transaction by re-associating them with a new session object.
- A detached object can be re-associated with a new Hibernate session by invoking one of these methods such as load(), refresh(), merge(), update() or save() on the new session with a reference to the detached object.

<generator> Element:

- When inserting a new record/row in a database table for the instantiated java object, the ID column must be populated with a unique value in order to uniquely identify that persisted object.
- The <generator> element helps define how to generate a new primary key for a new instance of the class.
- The <generator> element accepts a java class name that will be used to generate unique identifiers for instances of the persistent class.
- The <generator> element is the child element of <id> element.

Syntax:

```
<id name="<PropertyName>" type="<typeName>" column="<ColumnName>">
                        <generator class="<GeneratorClass>">
            <param name="<ParameterName>">ParameterName</param>
                        </generator>
</id>
```

Example:

```
<id name="id" type="long" column="CustomerNo">
                        <generator class="hilo">
<param name="table">hi_value</param>
<param name="column">next_value</param>
                        <param name="max_lo">10</param>
                        </generator>
</id>
```

The increment Class

- The increment class generates the primary key value by adding 1 to the current highest primary key value.
- Technically, this is achieved as follows:
- The generator fires an SQL select query and retrieves the highest value of the primary key column.
- The generator then increments the primary key column value by 1.
- This class generates identifiers of type long, short or int.

The identity Class

- The identity class supports the identity in database such as DB2, MYSQL, My SQL Server and Sybase.
- This class is not a fully portable option.
- ✓ This class generates identifiers of type long, short or int.

Example:

```
<id name="id" type="long" column="CustomerNo" unsaved-value="0">
                        <generator class="identity"/>
</id>
```

The sequence Class

- The sequence class uses a sequence in databases such as DB2, PostgreSQL, Oracle, SAP DB.
- This class is not a fully portable option.
- This class generates identifiers of type long, short or int.

Example:

```
<id name="id" type="long" column="CustomerNo" unsaved-value="0">
```

```
<generator class="sequence">
<param name="sequence"> CustomerNo_Sequence </param>
</generator>
</id>
```

The hilo Class

- The hilo class uses a hi/lo algorithm to efficiently and portably maintain and generate identifiers that are unique to that database.
- It uses a database table and columns as source of hi values to generate unique identifiers.
- This class generates identifiers of type long, short or int.

Example:

```
<id name="id" type="long" column="CustomerNo">
<generator class="hilo">
<param name="table">hi_value</param>
<param name="column">next_value</param>
<param name="max_lo">10</param>
</generator>
</id>
```

The uuid Class

- The uuid class uses a 128-bit UUID algorithm to generate identifiers, unique with in a network.
- The uuid class attempts to portably generate a unique primary key value, which is composed of the following:
  - The local IP address
  - The startup time of JVM

- The system time
- A counter value

- This class, however, cannot guarantee that a given key is unique, but it is a good enough for most clustering purposes.
- The UUID is encoded as a string of hexadecimal digits of length 32.

The native Class

- The native class selects one of the following depending upon the capabilities of the underlying database:
- Identity
- Sequence
- Hilo

The assigned Class

- The assigned class allows the application to assign an identifier to the object before invoking the save() method. The assigned class is invoked, by default, if no <generator> element is specified.

<composite-id> Element:

- Composite Identifiers (key) are primary key identifiers for classes that consist of more than one primary key column.
- For a table with a composite key, the <composite-id> element allows mapping multiple properties of the class as identifier properties.
- It accepts <key-property> property mappings and <key-many-to-one> mappings as child elements.

Project 5: CompositApp

Book.java

```
1  import java.io.Serializable;
2
3  public class Book implements  Serializable{
4
5      private static final long serialVersionUID = 1L;
6
7      private int bid;
8      private String bname;
9      private String author;
10     public int getBid() {
11         return bid;
12     }
13     public void setBid(int bid) {
14         this.bid = bid;
15     }
16     public String getBname() {
17         return bname;
18     }
19     public void setBname(String bname) {
20         this.bname = bname;
21     }
22     public String getAuthor() {
23         return author;
24     }
25     public void setAuthor(String author) {
26         this.author = author;
27     }
28
29 }
```

book.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="Book" table="book">
        <composite-id>
            <key-property name="bid" column="id"/>
            <key-property name="bname" column="name"/>
            <key-property name="author" column="author"/>
        </composite-id>
    </class>
</hibernate-mapping>
```

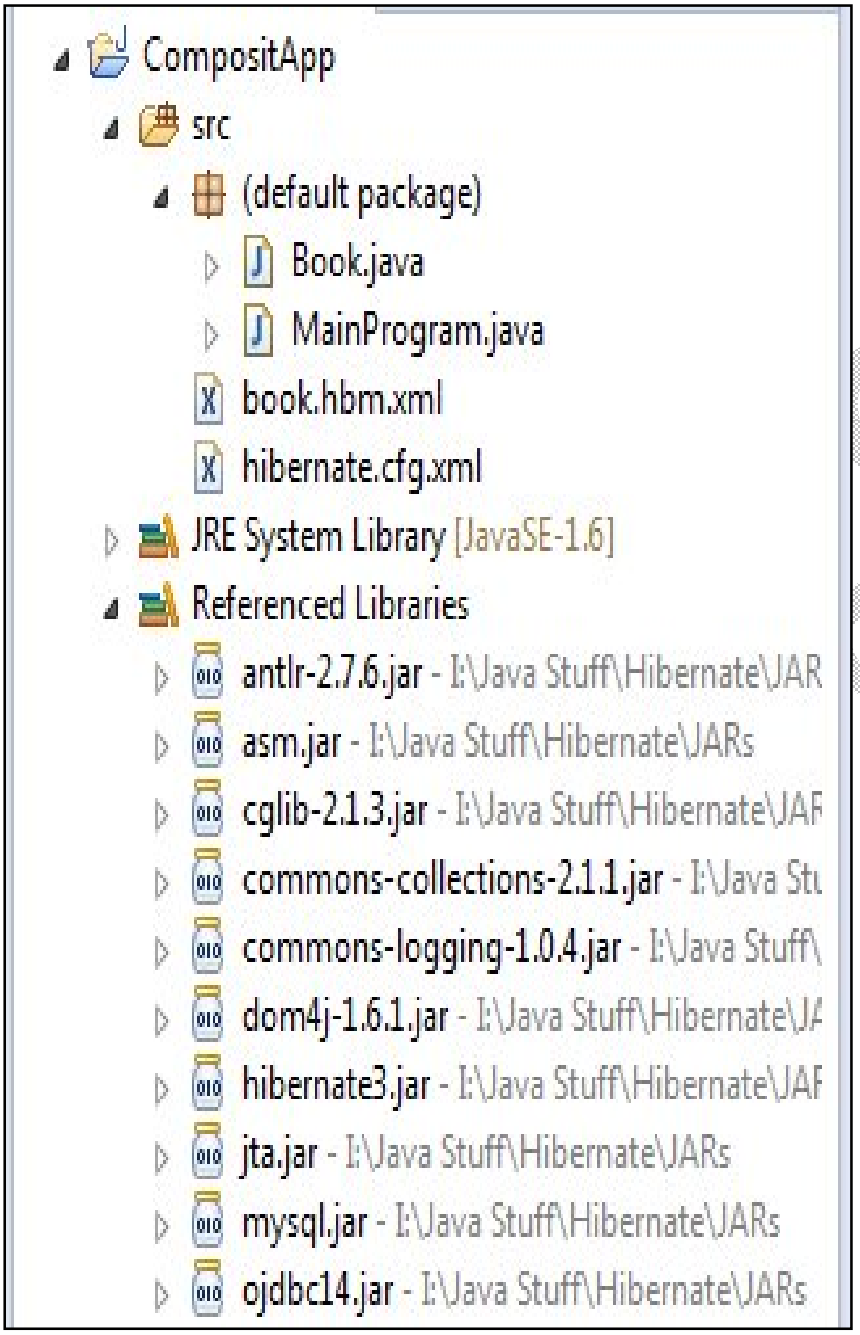
hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

    <hibernate-configuration>
        <session-factory>
            <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver</property>
            <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/test</property>
            <property name="hibernate.connection.username">
            root</property>
            <property name="hibernate.connection.password">
            root</property>

            <property name="hibernate.dialect">
            org.hibernate.dialect.OracleDialect</property>
            <mapping resource="book.hbm.xml"/>
        </session-factory>
    </hibernate-configuration>
```

MainProgram.java

```
1  import org.hibernate.Session;
5  public class MainProgram {
6      public static void main(String[] args) {
7
8              Configuration config=new Configuration();
9              config.configure();
10
11             SessionFactory factory=config.buildSessionFactory();
12             Session s=factory.openSession();
13
14             Transaction tx=s.beginTransaction();
15             Book student=new Book();
16             student.setBid(200);
17             student.setBname("JAVA");
18             student.setAuthor("Venkatesh");
19             s.save(student);
20
21             System.out.println("Record Inserted!");
22             //Executing Same project gives exception
23             //org.hibernate.exception.ConstraintViolationException
24             //if it contains same values
25             tx.commit();
26             s.close();
27             factory.close();
28     }
29 }
```

Folder Structure:



HIBERNATE QUERY LANGUAGE(HQL)

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries which in turns perform action on database.

Although you can use SQL statements directly with Hibernate using Native SQL but I would recommend to use HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's SQL generation and caching strategies.

Keywords like SELECT , FROM and WHERE etc. are not case sensitive but properties like table and column names are case sensitive in HQL.

FROM Clause:

You will use FROM clause if you want to load a complete persistent objects into memory. Following is the simple syntax of using FROM clause:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

If you need to fully qualify a class name in HQL, just specify the package and class name as follows:

```
String hql = "FROM com.sathya.criteria.Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

AS Clause:

The AS clause can be used to assign aliases to the classes in your HQL queries, specially when you have long queries. For instance, our previous simple example would be the following:

```
String hql = "FROM Employee AS E";
Query query = session.createQuery(hql);
List results = query.list();
```

The AS keyword is optional and you can also specify the alias directly after the class name, as follows:

```
String hql = "FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

SELECT Clause:

The SELECT clause provides more control over the result set than the from clause. If you want to obtain few properties of objects instead of the complete object, use the SELECT clause. Following is the simple syntax of using SELECT clause to get just first_name field of the Employee object:

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

It is notable here that Employee.firstName is a property of Employee object rather than a field of the EMPLOYEE table.

WHERE Clause:

If you want to narrow the specific objects that are returned from storage, you use the WHERE clause. Following is the simple syntax of using WHERE clause:

```
String hql = "FROM Employee E WHERE E.id = 10";
Query query = session.createQuery(hql);
List results = query.list();
```

ORDER BY Clause:

To sort your HQL query's results, you will need to use the ORDER BY clause. You can order the results by any property on the objects in the result set either ascending (ASC) or descending (DESC). Following is the simple syntax of using ORDER BY clause:

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas as follows:

```
String hql = "FROM Employee E WHERE E.id > 10 " +
        "ORDER BY E.firstName DESC, E.salary DESC ";
Query query = session.createQuery(hql);
List results = query.list();
```

GROUP BY Clause:

This clause lets Hibernate pull information from the database and group it based on a value of an attribute and, typically, use the result to include an aggregate value. Following is the simple syntax of using GROUP BY clause:

```
String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " +
        "GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

Using Named Paramters:

Hibernate supports named parameters in its HQL queries. This makes writing HQL queries that accept input from the user easy and you do not have to defend against SQL injection attacks. Following is the simple syntax of using named parameters:

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

UPDATE Clause:

Bulk updates are new to HQL with Hibernate 3, and deletes work differently in Hibernate 3 than they did in Hibernate 2. The Query interface now contains a method called executeUpdate() for executing HQL UPDATE or DELETE statements.

The UPDATE clause can be used to update one or more properties of an one or more objects. Following is the simple syntax of using UPDATE clause:

```
String hql = "UPDATE Employee set salary = :salary "  +
        "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

DELETE Clause:

The DELETE clause can be used to delete one or more objects. Following is the simple syntax of using DELETE clause:

```
String hql = "DELETE FROM Employee "  +
        "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 10);
```

```
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

INSERT Clause:

HQL supports INSERT INTO clause only where records can be inserted from one object to another object. Following is the simple syntax of using INSERT INTO clause:

```
String hql = "INSERT INTO Employee(firstName, lastName, salary)"  +
             "SELECT firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

Aggregate Methods:

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL and following is the list of the available functions:

S.N.	Functions	Description
1	avg(property name)	The average of a property's value
2	count(property name or *)	The number of times a property occurs in the results
3	max(property name)	The maximum value of the property values
4	min(property name)	The minimum value of the property values
5	sum(property name)	The sum total of the property values

The distinct keyword only counts the unique values in the row set. The following query will return only unique count:

```
String hql = "SELECT count(distinct E.firstName) FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

Project : HQLApp

Student.java

```java
public class Student
{
    private int sid;
    private String sname;
    private double fee;
    public void setSid(int sid)
    {
        this.sid = sid;
    }
    public int getSid()
    {
        return sid;
    }
    public void setSname(String sname)
    {
        this.sname = sname;
    }
    public String getSname()
    {
        return sname;
    }
    public double getFee() {
        return fee;
    }
    public void setFee(double fee) {
        this.fee = fee;
    }

}
```

Student.hbm.xml

```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="Student" table="student">
        <id name="sid"/>
        <property name="sname"/>
        <property name="fee" column="sfee"/>
    </class>
</hibernate-mapping>
```

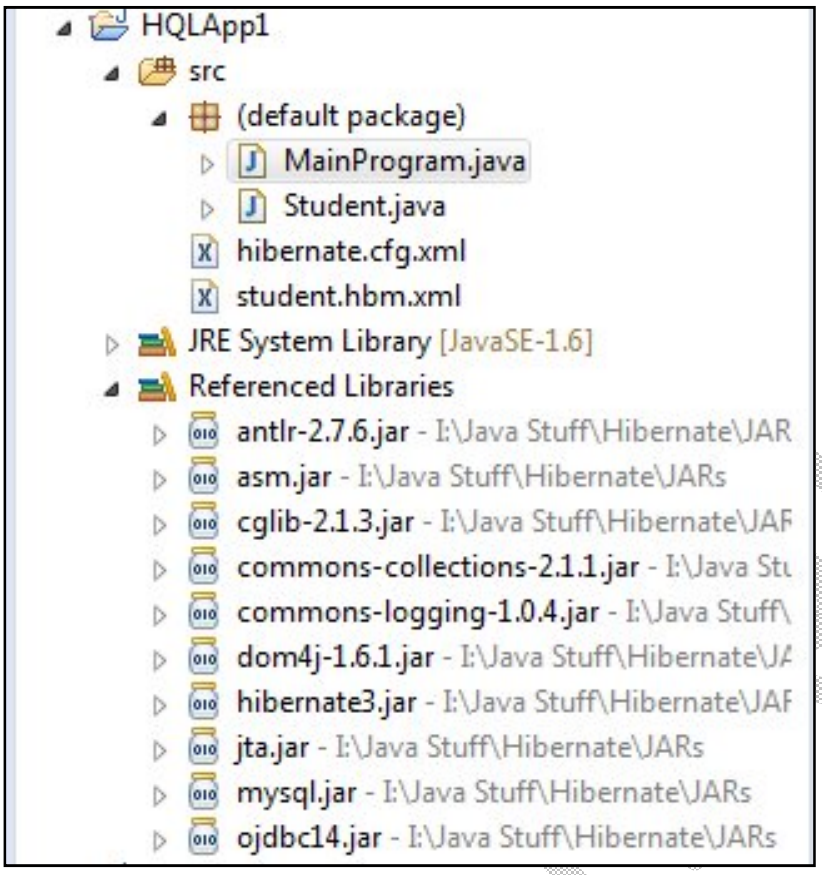
hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">root</property>
        <property name = "hibernate.connection.password">root</property>
        <property name = "hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <mapping resource="student.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```
import java.util.Iterator;
public class MainProgram
{
    public static void main(String[] args)
    {
        Configuration cfg = new Configuration();
        cfg  = cfg.configure();

        SessionFactory factory = cfg.buildSessionFactory();
        Session ses = factory.openSession();

        String qry="From Student st";
        Query query=ses.createQuery(qry);
        List<Student> list=query.list();
        Iterator<Student> i=list.iterator();

        while (i.hasNext()) {
            Student s = (Student) i.next();

            System.out.println("ID :"+s.getSid());
            System.out.println("Name :"+s.getSname());
            System.out.println("Fee:"+s.getFee());
        }
        System.out.println("============");
        ses.close();
        factory.close();
    }
}
```

Folder Structure:



CRITERIA

Hibernate provides alternate ways of manipulating objects and in turn data available in RDBMS tables. One of the methods is Criteria API which allows you to build up a criteria query object programmatically where you can apply filtration rules and logical conditions.

The Hibernate Session interface provides createCriteria() method which can be used to create a Criteria object that returns instances of the persistence object's class when your application executes a criteria query.

Following is the simplest example of a criteria query is one which will simply return every object that corresponds to the Employee class.

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

Restrictions with Criteria:

You can use add() method available for Criteria object to add restriction for a criteria query. Following is the example to add a restriction to return the records with salary is equal to 2000:

```
Criteria cr = session.createCriteria(Employee.class);
cr.add(Restrictions.eq("salary", 2000));
List results = cr.list();
```

Following are the few more examples covering different scenarios and can be used as per requirement:

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

// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));

// To get records having salary less than 2000
cr.add(Restrictions.lt("salary", 2000));

// To get records having fistName starting with zara
cr.add(Restrictions.like("firstName", "zara%"));

// Case sensitive form of the above restriction.
cr.add(Restrictions.ilike("firstName", "zara%"));

// To get records having salary in between 1000 and 2000
cr.add(Restrictions.between("salary", 1000, 2000));

// To check if the given property is null
```

```
cr.add(Restrictions.isNull("salary"));

// To check if the given property is not null
cr.add(Restrictions.isNotNull("salary"));

// To check if the given property is empty
cr.add(Restrictions.isEmpty("salary"));

// To check if the given property is not empty
cr.add(Restrictions.isNotEmpty("salary"));
```

You can create AND or OR conditions using LogicalExpression restrictions as follows:

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

Criterion salary = Restrictions.gt("salary", 2000);
Criterion name = Restrictions.ilike("firstNname","zara%");

// To get records matching with OR condistions
LogicalExpression orExp = Restrictions.or(salary, name);
cr.add( orExp );


// To get records matching with AND condistions
LogicalExpression andExp = Restrictions.and(salary, name);
cr.add( andExp );

List results = cr.list();
```

Project : CriteriaApp

Student.java

```java
public class Student
{
    private int sid;
    private String sname;
    private double fee;
    public void setSid(int sid)
    {
        this.sid = sid;
    }
    public int getSid()
    {
        return sid;
    }
    public void setSname(String sname)
    {
        this.sname = sname;
    }
    public String getSname()
    {
        return sname;
    }
    public double getFee() {
        return fee;
    }
    public void setFee(double fee) {
        this.fee = fee;
    }

}
```

Student.hbm.xml

```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="Student" table="student">
        <id name="sid"/>
        <property name="sname"/>
        <property name="fee" column="sfee"/>
    </class>
</hibernate-mapping>
```

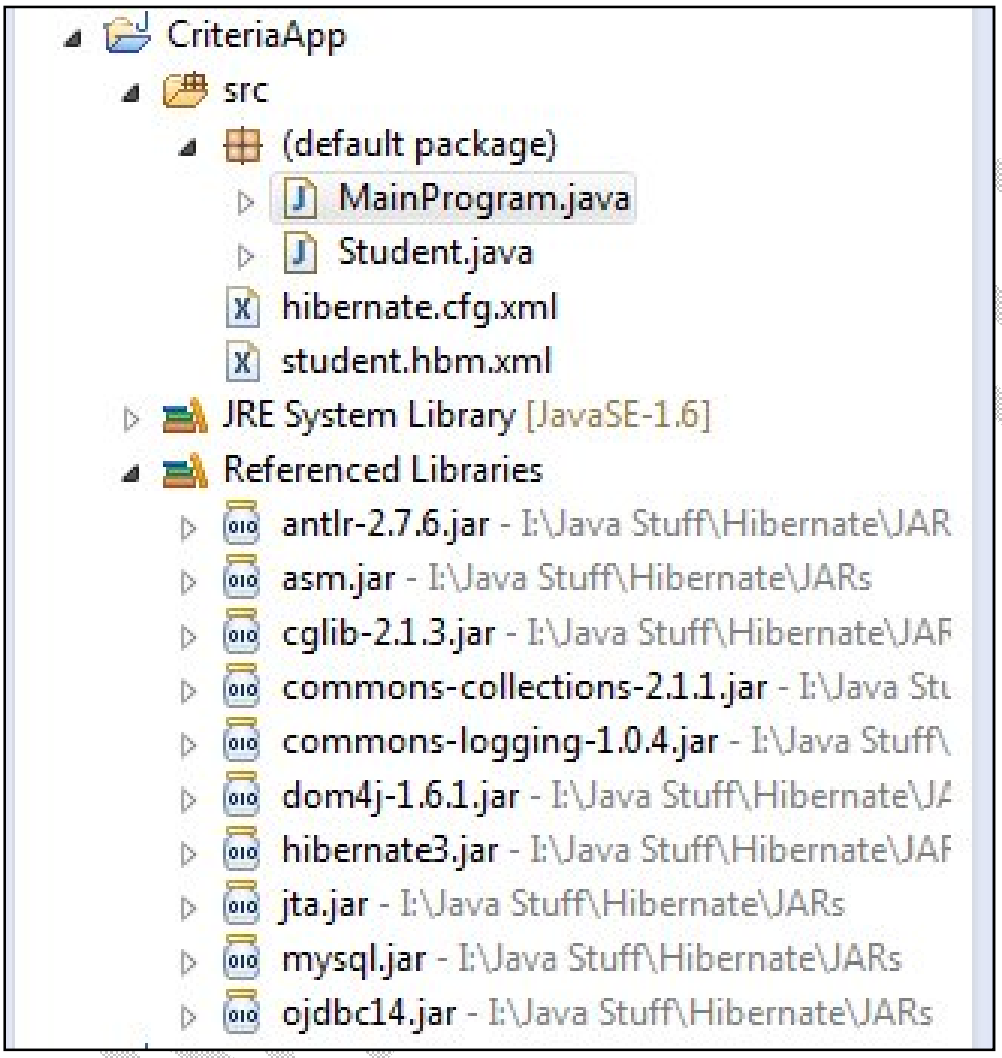
hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">root</property>
        <property name = "hibernate.connection.password">root</property>
        <property name = "hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <mapping resource="student.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```
import java.util.Iterator;
public class MainProgram
{
    public static void main(String[] args)
    {
        Configuration cfg = new Configuration();
        cfg  = cfg.configure();

        SessionFactory factory = cfg.buildSessionFactory();
        Session ses = factory.openSession();

        Criteria c=ses.createCriteria(Student.class);
        Criterion c1=Restrictions.gt("fee",120.0);
        c.add(c1);
        List<Student> list=c.list();
        Iterator<Student> i=list.iterator();

        while (i.hasNext()) {
            Student s = (Student) i.next();

            System.out.println("ID :"+s.getSid());
            System.out.println("Name :"+s.getSname());
            System.out.println("Fee:"+s.getFee());
        }
        System.out.println("============");
        ses.close();
        factory.close();
    }
}
```

Folder Structure:



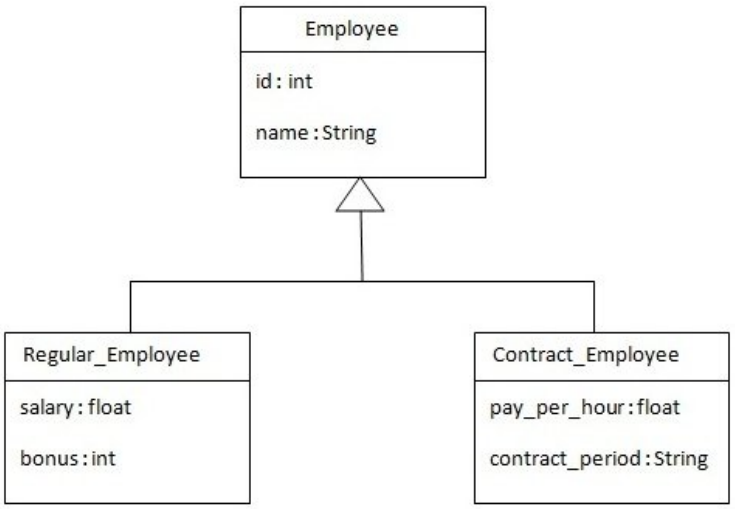
HIBERNATE INHERITANCE MAPPING

The Hibernate supports three different approaches of inheritance hierarchy mapping:

1. Table perClass hierarchy
2. Table per Subclass
3. Table per Concrete class

1.Table per Class Hierarchy

- In this approach we map the entire class hierarchy to a single table, i.e., a single table is used to represent the entire class hierarchy.
- This table includes columns for the properties for all classes in the hierarchy.
- This approach of inheritance mapping is basic, easier to use, and efficient compared to the other two approaches.
- This way of configuring the subclasses is best to use when we are designing our application top-down approach; that means we design the domain model and then design a SQL schema accordingly.
- However, this approach suffers from one major problem—the columns for properties declared by subclasses must be declared to accept null values, i.e., these columns cannot be declared with NOT NULL constraint.
- This limitation may cause serious problems for data integrity.



For above hierarchy it generates a table with all attributes as a single table as shown below.

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	(NULL)
type	varchar(255)	NO		(NULL)
name	varchar(255)	YES		(NULL)
salary	float	YES		(NULL)
bonus	int(11)	YES		(NULL)
pay_per_hour	float	YES		(NULL)
contract_duration	varchar(255)	YES		(NULL)

Project : PerHibrarchyApp

Employee.java

```
1  public class Employee {
2      private int id;
3      private String name;
4
5      public int getId() {
6          return id;
7      }
8      public void setId(int id) {
9          this.id = id;
10     }
11     public String getName() {
12         return name;
13     }
14     public void setName(String name) {
15         this.name = name;
16     }
17 }
```

Regular_Employee.java

```java
public class Regular_Employee extends Employee {

    private float salary;
    private int bonus;

    public float getSalary() {
        return salary;
    }
    public void setSalary(float salary) {
        this.salary = salary;
    }
    public int getBonus() {
        return bonus;
    }
    public void setBonus(int bonus) {
        this.bonus = bonus;
    }
}
```

Contract_Employee.java

```java
public class Contract_Employee extends Employee {

    private float pay_per_hour;
    private String contract_duration;

    public float getPay_per_hour() {
        return pay_per_hour;
    }
    public void setPay_per_hour(float pay_per_hour) {
        this.pay_per_hour = pay_per_hour;
    }
    public String getContract_duration() {
        return contract_duration;
    }
    public void setContract_duration(String contract_duration) {
        this.contract_duration = contract_duration;
    }
}
```

employee.hbm.xml

```
<hibernate-mapping>

    <class name="Employee" table="emp">
        <id name="id">
                <generator class="increment"/>
        </id>

        <discriminator column="type" type="string"/>
        <property name="name"></property>

        <subclass name="Regular_Employee"
        discriminator-value="reg_emp">
                <property name="salary"></property>
                <property name="bonus"></property>
        </subclass>

        <subclass name="Contract_Employee"
        discriminator-value="contract_emp">
                <property name="pay_per_hour"></property>
                <property name="contract_duration"></property>
        </subclass>
    </class>

</hibernate-mapping>
```

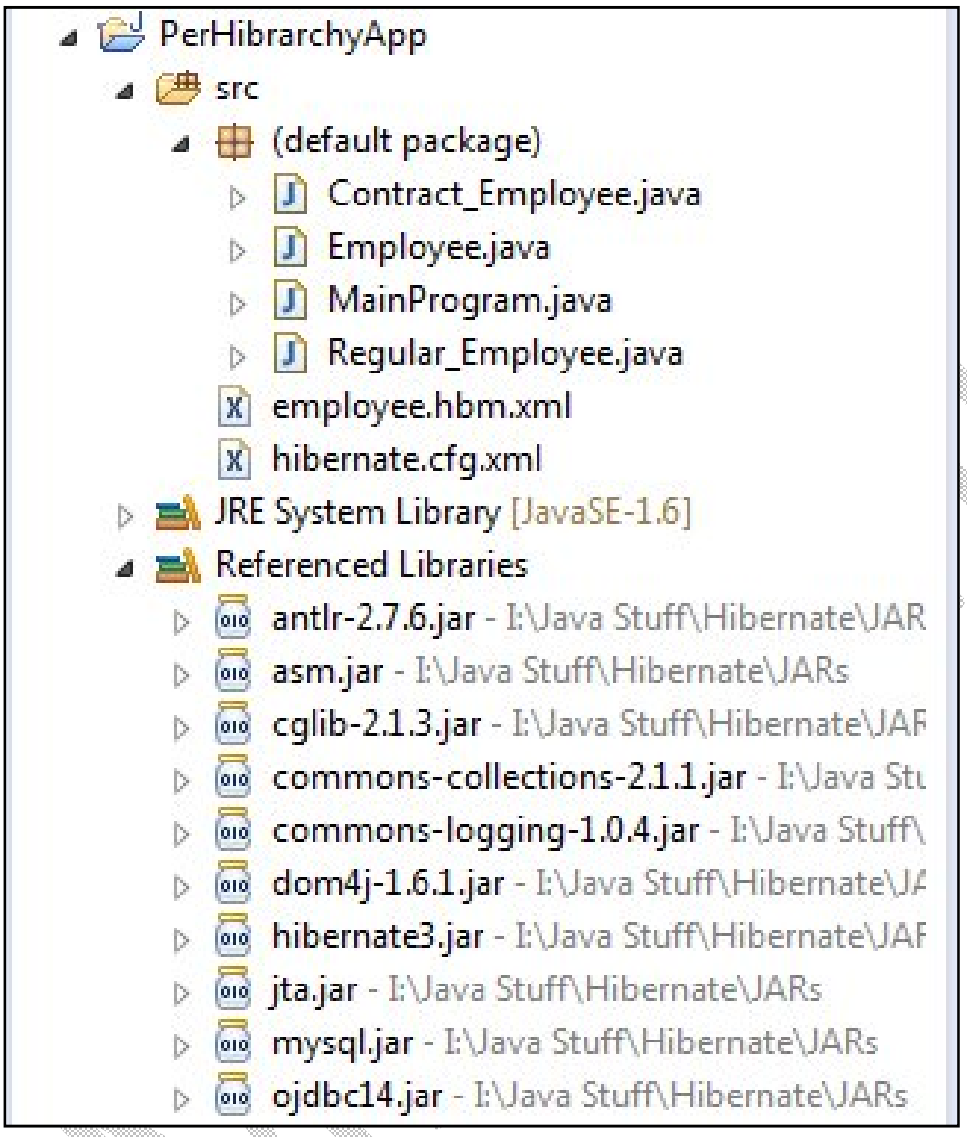
Hibernate.cfg.xml

```
<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">
        com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">
        jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">
        root</property>
        <property name = "hibernate.connection.password">
        root</property>
        <property name = "hibernate.dialect">
        org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">
        create</property>
        <mapping resource="employee.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```
1  import org.hibernate.Session;
5  public class MainProgram
6  {
7      public static void main(String[] args)
8      {
9          Configuration cfg = new Configuration();
10         cfg  = cfg.configure();
11
12         SessionFactory factory = cfg.buildSessionFactory();
13         Session ses = factory.openSession();
14
14
15         Transaction tx = ses.beginTransaction();
16         tx.begin();
17
18         Employee e1=new Employee();
19
20         e1.setName("SAM");
21
22         Regular_Employee e2=new Regular_Employee();
23         e2.setName("S Kumar");
24         e2.setSalary(50000);
25         e2.setBonus(5);
26
27         Contract_Employee e3=new Contract_Employee();
28         e3.setName("R Kumar");
29         e3.setPay_per_hour(1000);
30         e3.setContract_duration("15 hours");
31
32         ses.save(e1);
33         ses.save(e2);
34         ses.save(e3);
35
36         tx.commit();
37         ses.close();
38         System.out.println("Record Updated Successfully!");
39         factory.close();
40
41     }
42 }
```

Folder structure:



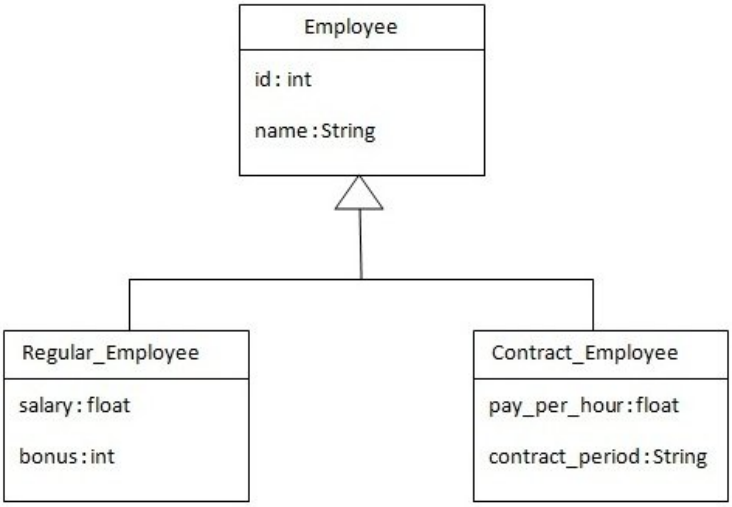
Note:

- Here, an extra column (also known as discriminator column) is created in the table to identify the class.
- In case of table per class hierarchy an discriminator column is added by the hibernate framework that specifies the type of the record. It is mainly used to distinguish the record. To specify this, discriminator subelement of class must be specified.

2.Table per Subclass

- In this approach we map the class hierarchy to multiple tables associated with the relational foreign key.

All the classes in the hierarchy that declare the persistent properties (including abstract classes) are mapped to a separate table.



For above hierarchy it generates a table for each class with given attributes as shown below.

Table : emp

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	(NULL)
name	varchar(255)	YES		(NULL)

Table: reg_emp

Field	Type	Null	Key	Default
eid	int(11)	NO	PRI	(NULL)
salary	float	YES		(NULL)
bonus	int(11)	YES		(NULL)

Table :contract_emp

Field	Type	Null	Key	Default
eid	int(11)	NO	PRI	(NULL)
pay_per_hour	float	YES		(NULL)
contract_duration	varchar(255)	YES		(NULL)

- In Hibernate, we use <joined-subclass> element to indicate the table per subclass mapping.

- The <joined-subclass> includes two important attributes to specify the subclass and table name.
- The <joined-subclass> encloses the mapping definition for all the properties declared in the joined subclass.

Project : PerSubClassApp

Employee.java

```
1  public class Employee {
2      private int id;
3      private String name;
4
5      public int getId() {
6          return id;
7      }
8      public void setId(int id) {
9          this.id = id;
10     }
11     public String getName() {
12         return name;
13     }
14     public void setName(String name) {
15         this.name = name;
16     }
17 }
```

Regular_Employee.java

```
1  public class Regular_Employee extends Employee {
2
3      private float salary;
4      private int bonus;
5
6      public float getSalary() {
7          return salary;
8      }
9      public void setSalary(float salary) {
10         this.salary = salary;
11     }
12     public int getBonus() {
13         return bonus;
14     }
15     public void setBonus(int bonus) {
16         this.bonus = bonus;
17     }
18 }
```

Contract_Employee.java

```java
public class Contract_Employee extends Employee {

    private float pay_per_hour;
    private String contract_duration;

    public float getPay_per_hour() {
        return pay_per_hour;
    }
    public void setPay_per_hour(float pay_per_hour) {
        this.pay_per_hour = pay_per_hour;
    }
    public String getContract_duration() {
        return contract_duration;
    }
    public void setContract_duration(String contract_duration) {
        this.contract_duration = contract_duration;
    }
}
```

employee.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

    <class name="Employee" table="emp">
        <id name="id">
                <generator class="increment"/>
        </id>
        <property name="name"></property>

        <joined-subclass name="Regular_Employee" table="reg_emp">
                <key column="eid"></key>
                <property name="salary"></property>
                <property name="bonus"></property>
        </joined-subclass>

        <joined-subclass name="Contract_Employee" table="contract_emp">
                <key column="eid"></key>
                <property name="pay_per_hour"></property>
                <property name="contract_duration"></property>
        </joined-subclass>
    </class>

</hibernate-mapping>
```

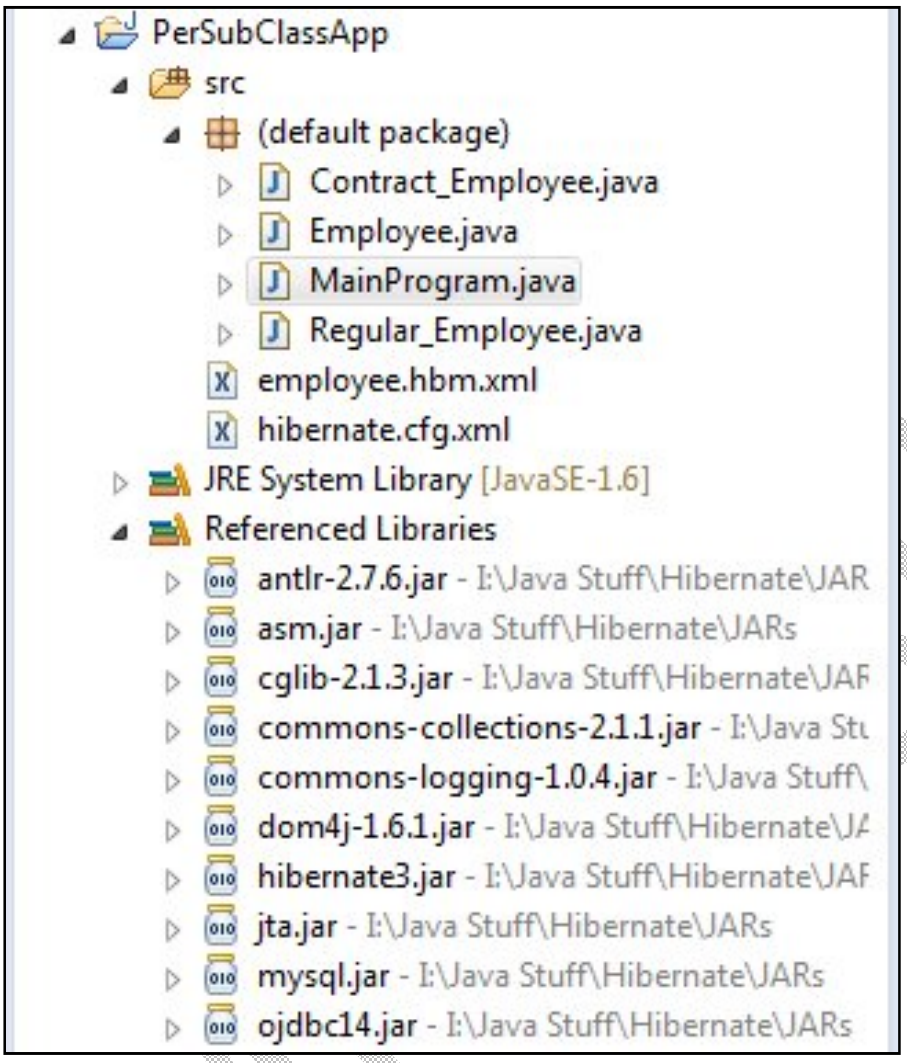
Hibernate.cfg.xml

```
<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">
        com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">
        jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">
        root</property>
        <property name = "hibernate.connection.password">
        root</property>
        <property name = "hibernate.dialect">
        org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">
        create</property>
        <mapping resource="employee.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```
1  import org.hibernate.Session;
5  public class MainProgram
6  {
7      public static void main(String[] args)
8      {
9          Configuration cfg = new Configuration();
10         cfg  = cfg.configure();
11
12         SessionFactory factory = cfg.buildSessionFactory();
13         Session ses = factory.openSession();
14
14
15         Transaction tx = ses.beginTransaction();
16         tx.begin();
17
18         Employee e1=new Employee();
19
20         e1.setName("SAM");
21
22         Regular_Employee e2=new Regular_Employee();
23         e2.setName("S Kumar");
24         e2.setSalary(50000);
25         e2.setBonus(5);
26
27         Contract_Employee e3=new Contract_Employee();
28         e3.setName("R Kumar");
29         e3.setPay_per_hour(1000);
30         e3.setContract_duration("15 hours");
31
32         ses.save(e1);
33         ses.save(e2);
34         ses.save(e3);
35
36         tx.commit();
37         ses.close();
38         System.out.println("Record Updated Successfully!");
39         factory.close();
40
41     }
42 }
```

Folder Structure:



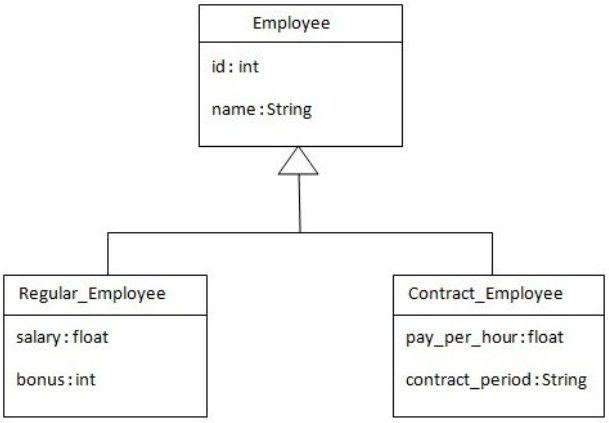
Note:

- The joined-subclass subelement of class, specifies the subclass. The key subelement of joined-subclass is used to generate the foreign key in the subclass mapped table. This foreign key will be associated with the primary key of parent class mapped table

3.Table per Concrete Class

- In this approach we map the class hierarchy to a multiple tables with no relation.

We use one table for each concrete (non-abstract) class in the hierarchy, declaring the columns for all the persistent properties declared by the class and inherited from its superclass.



For above hierarchy it generates a table for each non abstract class with given attributes as shown below.

Table : emp

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	(NULL)
name	varchar(255)	YES		(NULL)

Table: reg_emp

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	(NULL)
name	varchar(255)	YES		(NULL)
salary	float	YES		(NULL)
bonus	int(11)	YES		(NULL)

Table: contract_emp

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	(NULL)
name	varchar(255)	YES		(NULL)
pay_per_hour	float	YES		(NULL)
contract_duration	varchar(255)	YES		(NULL)

Project : PerConcreetClassApp

Employee.java

```java
public class Employee {
    private int id;
    private String name;

    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;
    }
}
```

Regular_Employee.java

```java
public class Regular_Employee extends Employee {

    private float salary;
    private int bonus;

    public float getSalary() {
        return salary;
    }
    public void setSalary(float salary) {
        this.salary = salary;
    }
    public int getBonus() {
        return bonus;
    }
    public void setBonus(int bonus) {
        this.bonus = bonus;
    }
}
```

Contract_Employee.java

```java
public class Contract_Employee extends Employee {

    private float pay_per_hour;
    private String contract_duration;

    public float getPay_per_hour() {
        return pay_per_hour;
    }
    public void setPay_per_hour(float pay_per_hour) {
        this.pay_per_hour = pay_per_hour;
    }
    public String getContract_duration() {
        return contract_duration;
    }
    public void setContract_duration(String contract_duration) {
        this.contract_duration = contract_duration;
    }
}
```

employee.hbm.xml

```xml
<hibernate-mapping>

    <class name="Employee" table="emp">
        <id name="id">
                <generator class="increment"/>
        </id>
        <property name="name"></property>

        <union-subclass name="Regular_Employee" table="reg_emp">
                <property name="salary"></property>
                <property name="bonus"></property>
        </union-subclass>

        <union-subclass name="Contract_Employee" table="contract_emp">
                <property name="pay_per_hour"></property>
                <property name="contract_duration"></property>
        </union-subclass>
    </class>

</hibernate-mapping>
```

Hibernate.cfg.xml

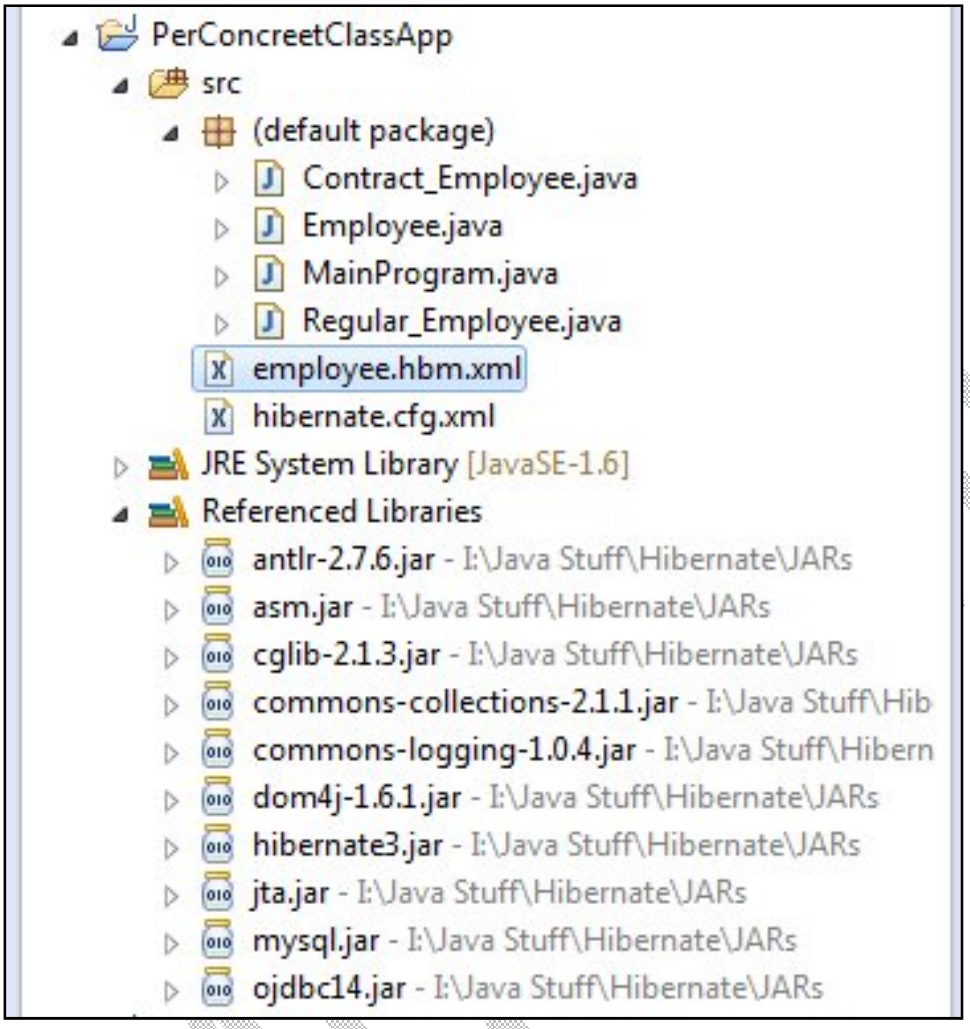
```
<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">
        com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">
        jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">
        root</property>
        <property name = "hibernate.connection.password">
        root</property>
        <property name = "hibernate.dialect">
        org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">
        create</property>
        <mapping resource="employee.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```
import org.hibernate.Session;..
public class MainProgram
{
    public static void main(String[] args)
    {
        Configuration cfg = new Configuration();
        cfg  = cfg.configure();

        SessionFactory factory = cfg.buildSessionFactory();
        Session ses = factory.openSession();

```

```
14
15          Transaction tx = ses.beginTransaction();
16          tx.begin();
17
18          Employee e1=new Employee();
19
20          e1.setName("SAM");
21
22          Regular_Employee e2=new Regular_Employee();
23          e2.setName("S Kumar");
24          e2.setSalary(50000);
25          e2.setBonus(5);
26
27          Contract_Employee e3=new Contract_Employee();
28          e3.setName("R Kumar");
29          e3.setPay_per_hour(1000);
30          e3.setContract_duration("15 hours");
31
32          ses.save(e1);
33          ses.save(e2);
34          ses.save(e3);
35
36          tx.commit();
37          ses.close();
38          System.out.println("Record Updated Successfully!");
39          factory.close();
40
41      }
42  }
```

Folder Structure:



Note:

The union-subclass subelement of class, specifies the subclass. It adds the columns of parent table into this table. In other words, it is working as a union.

ASSOCIATION MAPPING

Association Mapping is a concept of handling relationship between the objects of different types.

Example: We might design Faculty object to represent the Faculty details. Each Faculty object might be associated with one Address object; similarly, we might even design Dept object to represent department details where a Dept object might be associated with many number of Faculty objects.

The following are the possible relationships based on the multiplicity:

- One-to-One: In this case one entity object is exactly related to one object of another entity. An example of this is the relationship between a Faculty and his PersonalDetails.
- One-to-Many: In this type of relation one entity object is related to many objects of another entity. An example of this is the relationship between a department and the Faculty working in that department.
- Many-to-One: In this type relation many objects of an entity are related to one object of another entity. An example of this is the relationship between employee and their project. There may be many employees working for a single project.
- Many-to-Many: In this type of relation many objects of any entity are related to many objects of another entity. An example of this is the relationship between Student and courses. There may be many Student registered for one course and likewise a Student can register for multiple courses.

1.One-to-One:

Considering that One Faculty must have one record of personal details. Here Faculty and PersonalDetails are two classes, for them I took two tables faculty,personal.

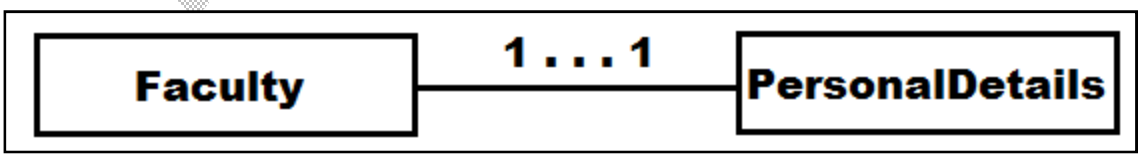


Fig: one-to-one

Project: One2OneApp

Faculty.java

```java
public class Faculty {

	private int fid;
	private String dept;
	private PersonalDetails details;
	public int getFid() {
		return fid;
	}
	public void setFid(int fid) {
		this.fid = fid;
	}
	public String getDept() {
		return dept;
	}
	public void setDept(String dept) {
		this.dept = dept;
	}
	public PersonalDetails getDetails() {
		return details;
	}
	public void setDetails(PersonalDetails details) {
		this.details = details;
	}
}
```

faculty.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="Faculty" table="faculty">
        <id name="fid" column="id">
            <generator class="increment"/>
        </id>

        <property name="dept" column="depname"/>
        <one-to-one name="details" class="PersonalDetails"/>
    </class>
</hibernate-mapping>
```

PersonalDetails.java

```java
public class PersonalDetails {
    private int pid;
    private String name;
    private int age;
    private String gender;
    public int getPid() {
        return pid;
    }
    public void setPid(int pid) {
        this.pid = pid;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
}
```

personaldetails.hbm.xml

```xml
<hibernate-mapping>
    <class name="PersonalDetails" table="personal">
        <id name="pid" column="id">
            <generator class="increment"/>
        </id>

        <property name="name" column="pname" />
        <property name="age" column="age"/>
        <property name="gender" column="gen"/>
    </class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">
        com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">
        jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">
        root</property>
        <property name = "hibernate.connection.password">
        root</property>
        <property name = "hibernate.dialect">
        org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">
        create</property>

        <mapping resource="faculty.hbm.xml"/>
        <mapping resource="personaldetails.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```
import org.hibernate.Session;..
public class MainProgram
{
    public static void main(String[] args)
    {
        Configuration cfg = new Configuration();
        cfg  = cfg.configure();

        SessionFactory factory = cfg.buildSessionFactory();
        Session ses = factory.openSession();
        Transaction tx=ses.beginTransaction();

```

```
16          PersonalDetails mydetails=new PersonalDetails();
17          mydetails.setName("RAM");
18          mydetails.setAge(36);
19          mydetails.setGender("Male");
20
21          Faculty f=new Faculty();
22          f.setDetails(mydetails);
23          f.setDept("CSE");
24
25          ses.save(mydetails);
26          ses.save(f);
27          tx.commit();
28          ses.close();
29          factory.close();
30
31      }
32  }
```

Folder Structure:

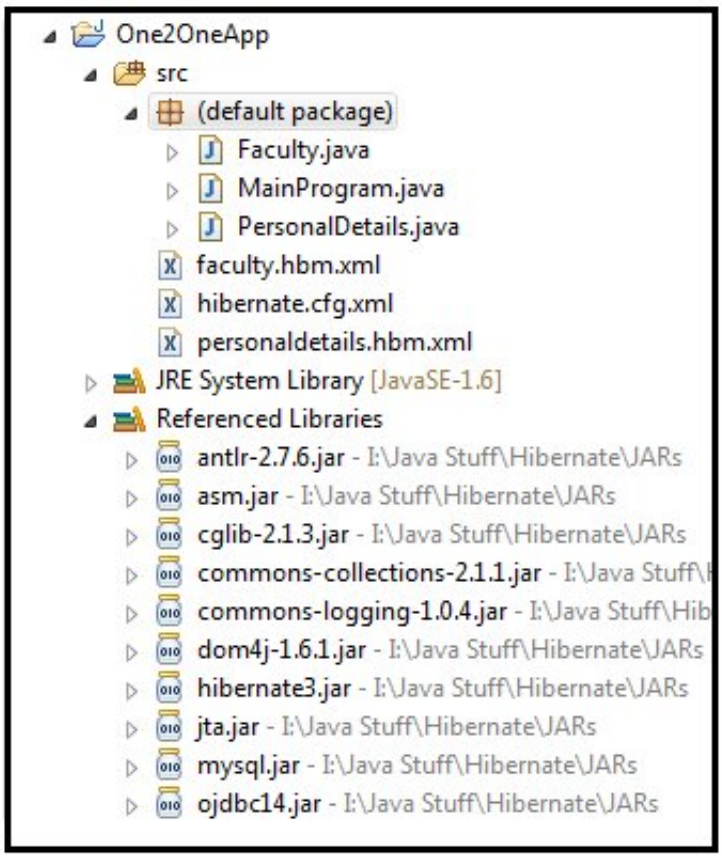


Table : Faculty

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	(NULL)
depname	varchar(255)	YES		(NULL)

Table: Personal

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	(NULL)
pname	varchar(255)	YES		(NULL)
age	int(11)	YES		(NULL)
gen	varchar(255)	YES		(NULL)

2.One-to-Many:

Considering that One Faculty is having more than one Address details like permanent, temporary, office etc.. Here Faculty and Address are two classes, for them I took two tables faculty,address.

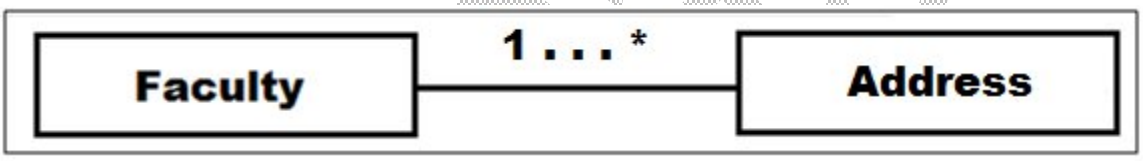


Fig: one-to-many

Project: One2ManyApp

Faculty.java

```
1  import java.util.Set;
2  public class Faculty {
3
4          private int fid;
5          private String dept;
6          private Set<Address> address;
7          public int getFid() {
8              return fid;
9          }
10         public void setFid(int fid) {
11             this.fid = fid;
12         }
13         public String getDept() {
14             return dept;
15         }
```

```
16          public void setDept(String dept) {
17              this.dept = dept;
18          }
19          public Set<Address> getAddress() {
20              return address;
21          }
22          public void setAddress(Set<Address> address) {
23              this.address = address;
24          }
25  }
```

Address.java

```
 1  public class Address {
 2      private int addressid;
 3      private String hno;
 4      private String street;
 5      public int getAddressid() {
 6          return addressid;
 7      }
 8      public void setAddressid(int addressid) {
 9          this.addressid = addressid;
10      }
11      public String getHno() {
12          return hno;
13      }
14      public void setHno(String hno) {
15          this.hno = hno;
16      }
17      public String getStreet() {
18          return street;
19      }
20      public void setStreet(String street) {
21          this.street = street;
22      }
23  }
```

faculty.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="Faculty" table="faculty">
        <id name="fid" column="id">
            <generator class="increment"/>
        </id>

        <property name="dept" column="depname"/>
        <set name="address" table="address">
            <key column="id"/>
            <one-to-many class="Address"/>
        </set>
    </class>
</hibernate-mapping>
```

address.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="Address" table="address">
        <id name="addressid" column="aid">
            <generator class="native"/>
        </id>

        <property name="hno" column="hno"/>
        <property name="street" column="street"/>

    </class>
</hibernate-mapping>
```

MainProgram.java

```
import java.util.HashSet;
public class MainProgram
{
    public static void main(String[] args)
    {
        Configuration cfg = new Configuration();
        cfg  = cfg.configure();

        SessionFactory factory = cfg.buildSessionFactory();
        Session ses = factory.openSession();
        Transaction tx=ses.beginTransaction();

        Address addr1=new Address();
        addr1.setHno("3-345");
        addr1.setStreet("Dilsuknagar");

        Address addr2=new Address();
        addr2.setHno("4/A-245");
        addr2.setStreet("Ramanthpur");

        Set<Address> set1=new HashSet<Address>();
        set1.add(addr1);
        set1.add(addr2);

        Faculty f=new Faculty();
        f.setAddress(set1);
        f.setDept("CSE");

        ses.save(addr1);
        ses.save(addr2);
        ses.save(f);
        tx.commit();
        ses.close();
        factory.close();

    }
}
```

hibernate.cgf.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">
        com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">
        jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">
        root</property>
        <property name = "hibernate.connection.password">
        root</property>
        <property name = "hibernate.dialect">
        org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">
        create</property>

        <mapping resource="faculty.hbm.xml"/>
        <mapping resource="address.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Folder Structure:

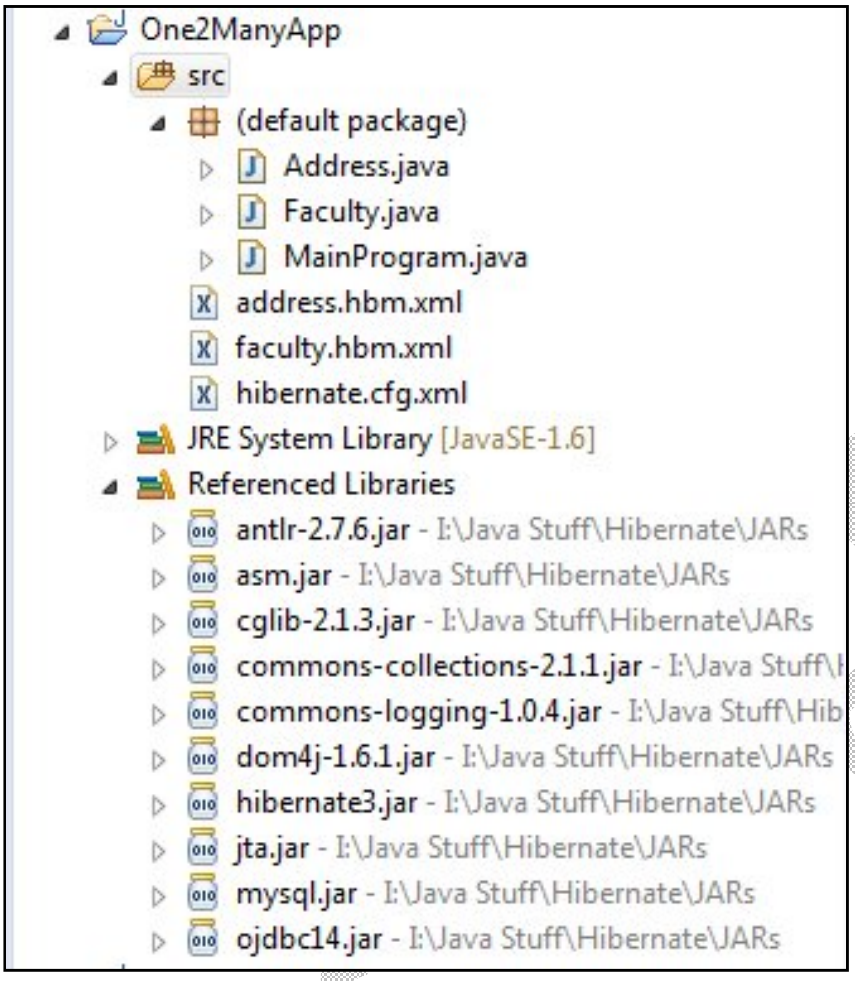


Table: faculty

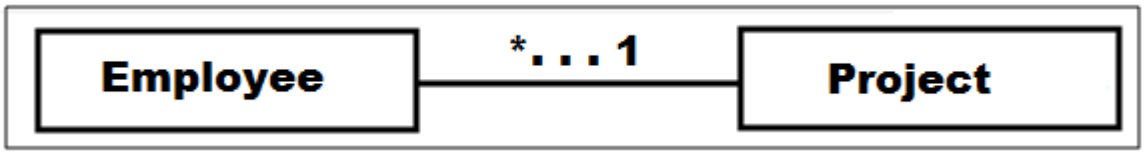
Field	Type	Null	Key	Default
id	int(11)	NO	PRI	(NULL)
depname	varchar(255)	YES		(NULL)

Table: Address

Field	Type	Null	Key	Default
aid	int(11)	NO	PRI	(NULL)
hno	varchar(255)	YES		(NULL)
street	varchar(255)	YES		(NULL)
id	int(11)	YES	MUL	(NULL)

3.Many-to-One:

Considering that more than one Employee is working one Project. Here Employee and Project are two classes, for them I took two tables employee, project.



Project: Many2OneApp

Employee.java

```java
public class Employee {
    private int eid;
    private String name;
    private Project project;
    public int getEid() {
        return eid;
    }
    public void setEid(int eid) {
        this.eid = eid;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Project getProject() {
        return project;
    }
    public void setProject(Project project) {
        this.project = project;
    }

}
```

employee.hbm.xml

```xml
<hibernate-mapping>
    <class name="Employee" table="employee">
        <id name="eid" column="eid">
            <generator class="increment"/>
        </id>

        <property name="name" column="ename"/>
        <many-to-one name="project" class="Project" column="project"/>
    </class>
</hibernate-mapping>
```

Project.java

```java
public class Project {
    private int projectId;
    private String pname;
    private String domain;
    public int getProjectId() {
        return projectId;
    }
    public void setProjectId(int projectId) {
        this.projectId = projectId;
    }
    public String getPname() {
        return pname;
    }
    public void setPname(String pname) {
        this.pname = pname;
    }
    public String getDomain() {
        return domain;
    }
    public void setDomain(String domain) {
        this.domain = domain;
    }

}
```

project.hbm.xml

```xml
<hibernate-mapping>
    <class name="Project" table="project">
        <id name="projectId" column="pid">
            <generator class="increment"/>
        </id>

        <property name="pname" column="pname"/>
        <property name="domain" column="type"/>
    </class>
</hibernate-mapping>
```

MainProgram.java

```
1  import org.hibernate.Session;..
5  public class MainProgram
6  {
7      public static void main(String[] args)
8      {
9          Configuration cfg = new Configuration();
10         cfg  = cfg.configure();
11
12         SessionFactory factory = cfg.buildSessionFactory();
13         Session ses = factory.openSession();
14         Transaction tx=ses.beginTransaction();
15
16         Project p=new Project();
17         p.setPname("SathyaTech");
18         p.setDomain("Java");
19
20         Employee e1=new Employee();
21         e1.setName("Venkatesh");
22         e1.setProject(p);
23
24         Employee e2=new Employee();
25         e2.setName("Naveen");
26         e2.setProject(p);
27
28         ses.save(p);
29         ses.save(e1);
30         ses.save(e2);
31         tx.commit();
32         ses.close();
33         factory.close();
34
35     }
36 }
```

hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">
        com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">
        jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">
        root</property>
        <property name = "hibernate.connection.password">
        root</property>
        <property name = "hibernate.dialect">
        org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">
        create</property>

        <mapping resource="employee.hbm.xml"/>
        <mapping resource="project.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Folder Structure:

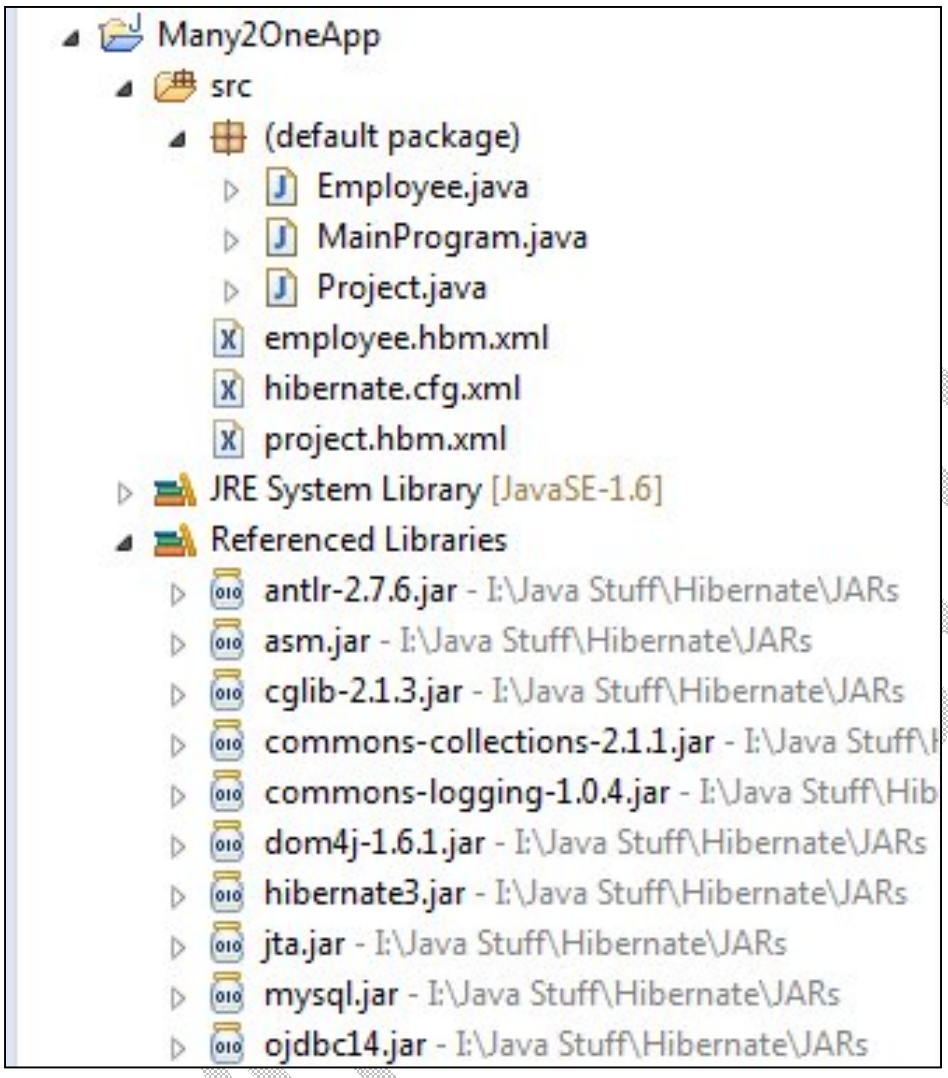


Table: Employee

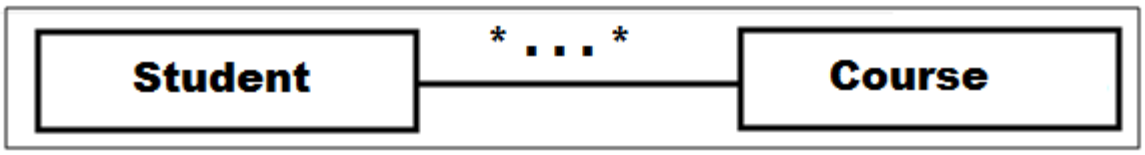
Field	Type	Null	Key	Default
eid	int(11)	NO	PRI	(NULL)
ename	varchar(255)	YES		(NULL)
project	int(11)	YES	MUL	(NULL)

Table : Project

Field	Type	Null	Key	Default
pid	int(11)	NO	PRI	(NULL)
pname	varchar(255)	YES		(NULL)
type	varchar(255)	YES		(NULL)

4.Many-to-Many:

Considering that more than one Student is pursuing more than one Course. Here Student and Course are two classes, for them I took two tables student, course.



Project: Many2ManyApp

Student.java

```
1  import java.util.Set;
2  public class Student {
3      private int sid;
4      private String sname;
5      private Set<Course> course;
6      public int getSid() {
7          return sid;
8      }
9      public void setSid(int sid) {
10         this.sid = sid;
11     }
12     public String getSname() {
13         return sname;
14     }
15     public void setSname(String sname) {
16         this.sname = sname;
17     }
18     public Set<Course> getCourse() {
19         return course;
20     }
21     public void setCourse(Set<Course> course) {
22         this.course = course;
23     }
24 }
```

student.hbm.xml

```xml
<hibernate-mapping>
    <class name="Student" table="student">
        <id name="sid" column="sid">
            <generator class="increment"/>
        </id>

        <property name="sname" column="name"/>
        <set name="course" table="stdcourse">
            <key column="sid"></key>
            <many-to-many class="Course" column="courseId"/>
        </set>
    </class>
</hibernate-mapping>
```

Course.java

```java
public class Course {
    private int courseId;
    private String cname;
    private double fee;
    public int getCourseId() {
        return courseId;
    }
    public void setCourseId(int courseId) {
        this.courseId = courseId;
    }
    public String getCname() {
        return cname;
    }
    public void setCname(String cname) {
        this.cname = cname;
    }
    public double getFee() {
        return fee;
    }
    public void setFee(double fee) {
        this.fee = fee;
    }

}
```

course.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="Course" table="course">
        <id name="courseId" column="cid">
            <generator class="increment"/>
        </id>

        <property name="cname" column="name"/>
        <property name="fee" column="amt"/>
    </class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.connection.driver_class">
        com.mysql.jdbc.Driver</property>
        <property name = "hibernate.connection.url">
        jdbc:mysql://localhost:3306/test</property>
        <property name = "hibernate.connection.username">
        root</property>
        <property name = "hibernate.connection.password">
        root</property>
        <property name = "hibernate.dialect">
        org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">
        create</property>

        <mapping resource="student.hbm.xml"/>
        <mapping resource="course.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

MainProgram.java

```
1  import java.util.HashSet;
8  public class MainProgram
9  {
10     public static void main(String[] args)
11     {
12         Configuration cfg = new Configuration();
13         cfg  = cfg.configure();
14
15         SessionFactory factory = cfg.buildSessionFactory();
16         Session ses = factory.openSession();
17         Transaction tx=ses.beginTransaction();
18
19         Course c1=new Course();
20         c1.setCname("Core Java");
21         c1.setFee(230.32);
22
23         Course c2=new Course();
24         c2.setCname("Hibernate");
25         c2.setFee(432.43);
26
27         Set<Course> set1=new HashSet<Course>();
28         set1.add(c1);
29         set1.add(c2);
30
31         Student s1=new Student();
32         s1.setSname("Venkatesh");
33         s1.setCourse(set1);
34
35         Student s2=new Student();
36         s2.setSname("Naveen");
37         s2.setCourse(set1);
38
39         ses.save(c1);
40         ses.save(c2);
41         ses.save(s1);
42         ses.save(s2);
43
44         tx.commit();
45         ses.close();
46         factory.close();
47
48     }
49 }
```

Folder Structure:

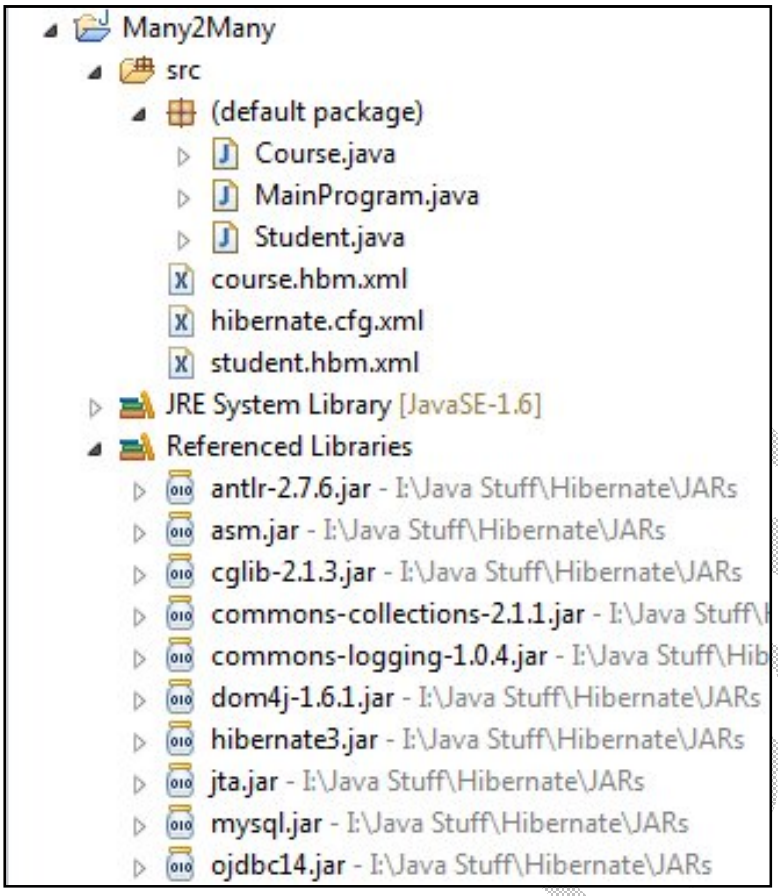


Table: Student

Field	Type	Null	Key	Default
sid	int(11)	NO	PRI	(NULL)
name	varchar(255)	YES		(NULL)

Table: Course

Field	Type	Null	Key	Default
cid	int(11)	NO	PRI	(NULL)
name	varchar(255)	YES		(NULL)
amt	double	YES		(NULL)

Table: stdcourse

Field	Type	Null	Key	Default
sid	int(11)	NO	PRI	(NULL)
courseId	int(11)	NO	PRI	(NULL)

**Note: In many-to-many association an additional table is generated to bind 2 tables.

HIBERNATE ANNOTATIONS

The hibernate application can be created with annotation. There are many annotations that can be used to create hibernate application such as @Entity, @Id, @Table etc.
Hibernate Annotations are based on the JPA 2 specification and supports all the features.

All the JPA annotations are defined in the javax.persistence.* package. Hibernate EntityManager implements the interfaces and life cycle defined by the JPA specification.

The core advantage of using hibernate annotation is that you don't need to create mapping (hbm) file. Here, hibernate annotations are used to provide the meta data.

Create the Persistent class:

Here, we are creating the same persistent class which we have created in the previous topic. But here, we are using annotation.

- @Entity annotation marks this class as an entity.
- @Table annotation specifies the table name where data of this entity is to be persisted. If you don't use @Table annotation, hibernate will use the class name as the table name bydefault.
- @Id annotation marks the identifier for this entity.
- @Column annotation specifies the details of the column for this property or field. If @Column annotation is not specified, property name will be used as the column name bydefault.

Example :

```
1  import javax.persistence.Entity;
2  import javax.persistence.Id;
3  import javax.persistence.Table;
4
5  @Entity
6  @Table(name= "emp500")
7  public class Employee {
8      @Id
9      private int id;
10     @Column(name="fname")
11     private String firstName;
12
13     //getters and setters
14 }
```

Project : AnnotationApp

Student.java

```
 1 import javax.persistence.Column;
 6
 7 @Entity
 8 @Table(name="student")
 9 public class Student {
10
11     @Id
12     @GeneratedValue
13     private int sid;
14     @Column(name="sname")
15     private String name;
16     @Column(name="sfee")
17     private double fee;
18
19     public int getSid() {
20         return sid;
21     }
22     public void setSid(int sid) {
23         this.sid = sid;
24     }
25     public String getName() {
26         return name;
27     }
28     public void setName(String name) {
29         this.name = name;
30     }
31     public double getFee() {
32         return fee;
33     }
34     public void setFee(double fee) {
35         this.fee = fee;
36     }
37 }
```

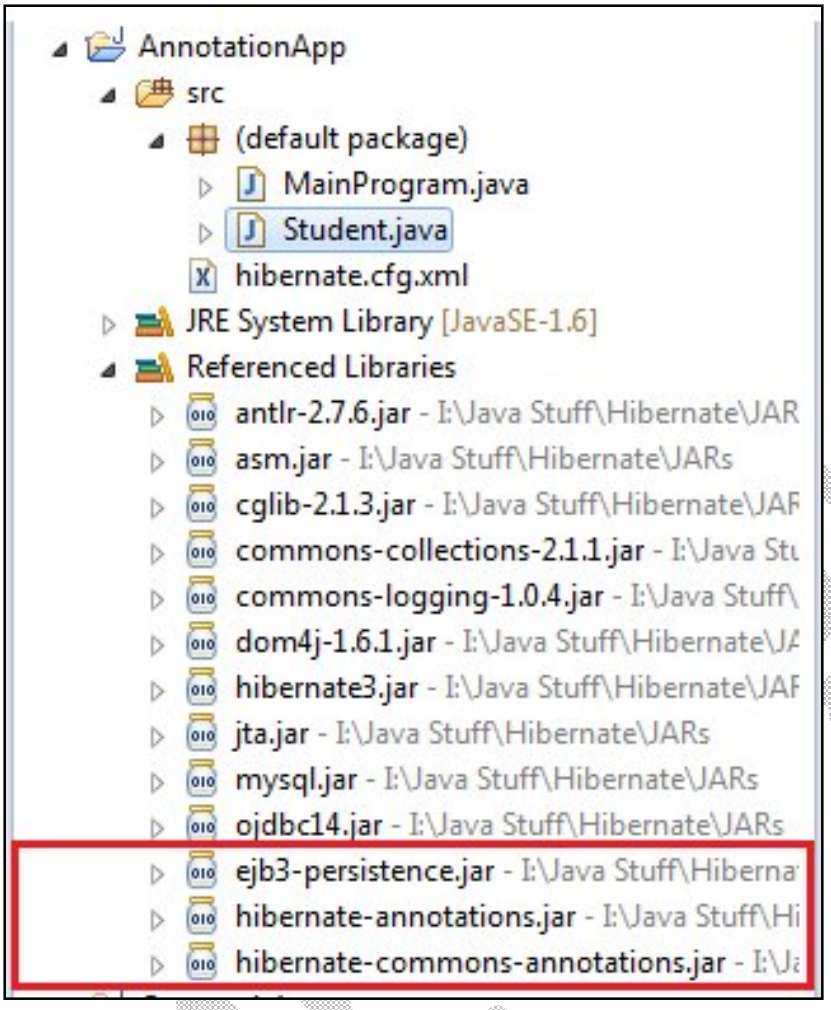
Hibernate.cfg.xml:

```
    <hibernate-configuration>
        <session-factory>
            <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver</property>
            <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/test</property>
            <property name="hibernate.connection.username">
            root</property>
            <property name="hibernate.connection.password">
            root</property>

            <property name="show_sql">true</property>
            <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect</property>
            <mapping class="Student"/>
        </session-factory>
    </hibernate-configuration>
```

MainProgram.java

```
import org.hibernate.Session;

public class MainProgram {
    public static void main(String[] args) {

            Configuration config=new AnnotationConfiguration();
            config.configure();

            SessionFactory factory=config.buildSessionFactory();
            Session s=factory.openSession();

            Transaction tx=s.beginTransaction();
            tx.begin();
            Student student=new Student();
            student.setSid(300);
            student.setName("Naveen");
            student.setFee(200.78);
            s.save(student);
            System.out.println("Record Inserted!");

            System.out.println(student.getSid());
            System.out.println(student.getName());
            System.out.println(student.getFee());

            tx.commit();
            s.close();
            factory.close();
    }
}
```

Folder Structure:



Note : ejb3-persistence.jar, hibernate-annotations.jar and hibernate-commons-annotations.jar are added to project libraries to work with annotation based mapping.