Component-based Software Development

Java Persistence APIs
&
Entity Manager

Dr. Vinod Dubey
SWE 645
George Mason University

Acknowledgement/Reference

https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

&

http://www.vogella.de/articles/JavaPersistenceAPI/article.html

Outline

- Data Persistence
- Java Persistence API
- JPA Entities
- Persistence Unit
- Persistent Context
- Entity Manager
- Examples using JPA

Data Persistence

- **Mechanism to store application data into a persistence storage such as database**

Java Object

Person

- id
- firstNme
- lastName

Table

PERSON_ID	FIRST_NAME	LAST_NAME
1	Greg	Martin
2	John	Doe
3	George	Smith

Data Persistence

- **Mechanism to store application data into a persistence storage such as database**
- **Approach so far: JDBC/SQL**

Java Object

Person

- id
- firstNme
- lastName

Table

PERSON_ID	FIRST_NAME	LAST_NAME
1	Greg	Martin
2	John	Doe
3	George	Smith

Data Persistence

- **Mechanism to store application data into a persistence storage such as database**
- **Approach so far: JDBC/SQL**
- **New Approach: O/R Mapping**
  - Object/Relational Mapping

Table

PERSON_ID	FIRST_NAME	LAST_NAME
1	Greg	Martin
2	John	Doe
3	George	Smith

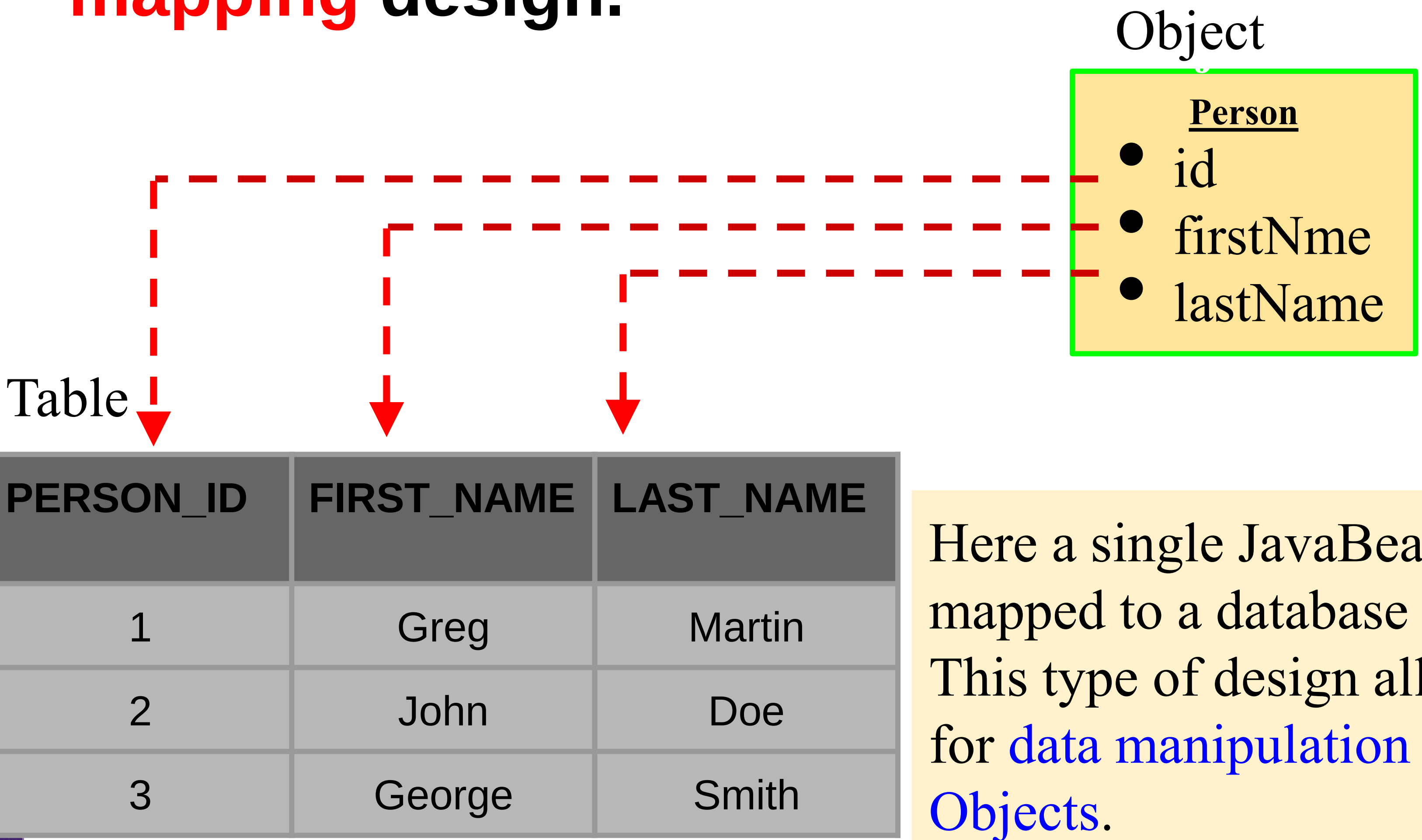
Java Object

<u>**Person**</u>

- id
- firstNme
- lastName

Object Relation Mapping (ORM)

The Java Persistence API utilizes the O/R mapping design.



PERSON_ID	FIRST_NAME	LAST_NAME
1	Greg	Martin
2	John	Doe
3	George	Smith

Here a single JavaBean is mapped to a database record. This type of design allows for data manipulation using Objects.

When to Use ORM

- **There is a natural mapping between objects and database tables**
- **Most create/update/delete operations work with individual objects**
- **You don’t want to worry about the mechanics of the database, and vendor specific SQL**

When NOT to Use ORM

- There is no easy way to map objects to database tables
- When not working with a Relational Database
- Data access is predominant through large data sets, batch operation

Goal

The Goal is to learn:

- **How to use Java Persistence APIs (JPA) annotations to configure Java classes (POJOs) into JPA entities to provide object to relational mapping**

- **How to use Persistence APIs (EntityManager) to implement CRUD operations**

JPA Programming Model

- **JPA Entities are plain Java classes (POJOs) that are mapped to relational database table**
  - Uses Java Persistence API (JPA) annotations
- **Entity Manager – is used to create, query, remove, and update JPA entities in your java application**
  - Which in turn updates records into the database table

JPA Programming Model

- **JPA is a Java specification that provides Object to Relational mapping**
  - It define syntax to configure a java class into a JPA entity, capture primary and foreign key constraints etc.
  - Provides capability to create, read, update, and delete database rows using JPA entities
  - Provides support for transactions and other similar capabilities required by an application accessing a database
  - Uses persistent metadata via annotations in the Java class

Java Persistence API (JPA)

- **Helps manage persistence of application data in Java Enterprise Edition (EE) and Java Standard Edition (SE) applications.**
  - Allows you to directly work with Java classes instead of database modules or SQL
- **Reference implementation of JPA: EclipseLink**
  - Other popular implementations: **Hibernate**, **Apache OpenJPA**

JPA Entity Classes

- Two Bare Minimum Requirements to specify a JPA entity
  - The class must be annotated with the @Entity
    - javax.persistence.Entity
  - Must define a primary key using @Id annotation
- JPA entity class (POJO) with a no-arg public constructor is used to define the mapping with one or more relational database tables
  - The instance variables follow JavaBeans-style properties and represent the persistent state of the entity

JPA Entity Classes: Standard POJOs with certain requirements

- **Other Requirements**
  - The class must have a public or protected, no-argument constructor
    - The class may have other constructors.
  - The class must not be declared final
    - No methods or persistent instance variables must be declared final
  - If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the Serializable interface
  - Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes
  - Persistent instance variables must be declared private, protected, or package-private
    - Can be accessed directly only by the entity class's methods
    - Clients must access the entity's state through accessor or business methods

JPA Entity Classes (Contd.)

- **By default, table name in the database correspond to the Entity class name**
    - Default table name can be changed using @Table(name="NEW_TABLE_NAME")
- **By default, entity fields names are mapped to columns with same name**
    - Default names can be changed using @Column annotation, e.g., @Column (name="newColumnName");

JPA Entity Classes (Contd.)

- **An Entity class must define a primary key**
  - @Id annotation
  - Can be auto generated using @GeneratedValue annotation
- **JPA persists all fields by default**
  - Fields that should not be persisted should be annotated with @Transient
- **Clients of entity beans make use of EntityManager to persist and manipulate data into database**
- **An instance of an Entity class represent a row in the table**

Persistent Field Types

- **The persistent fields or properties of an entity may be of the following types:**
  - Java primitive types,
  - java.lang.String,
  - java.math.BigInteger,
  - java.math.BigDecimal,
  - java.util.Date,
  - java.util.Calendar,
  - java.sql.Date,
  - java.sql.Time,
  - java.sql.Timestamp,
  - byte[], Byte[], char[],
  - Character[], enums and
  - other Java serializable types,
  - entity types, collections of entity types
  - embeddable classes, and collections of basic and embeddable classes.
  - The @Temporal annotation may be specified on fields of type java.util.Date and java.util.Calendar to specify the temporal type of the field, e.g., @Temporal(TemporalType.DATE) private Date startDate; *//puts date for startDate and not default timestamp*

Persistent Field Types

- **For String data, hibernate, by default, creates varchar(255)**
- **If a field need to store large amount of text, you can use @Lob (large object) annotation,**
  - which creates CLOB or BLOB data types and thus not restricted to 255 characters
- **For example,**

```
@Lob
private String comments;
```

JPA Entity – An example

- **Two bare minimum annotations to configure a POJO as a JPA entity: @Entity and @Id**

```
import javax.persistence.*;

@Entity
public class Customer implements java.io.Serializable{
    @Id
    private long id;
    private String firstName;
    private String lastName;

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getFirstName() {return firstName;}
    public void setFirstName(String fn) {this.firstName = fn;}

    public String getLastName() {return lastName;}
    public void setLastName(String ln) {this.lastName = ln;}
}
```

JPA Entity – An example

```
@Entity
@Table(name = "CUST_TBL")
public class Customer implements java.io.Serializable{
        private long id;
        private String firstName;
        private String lastName;

        @Id
        @Column(name="CUST_ID", nullable=false, columnDefinition="long")
        public long getId() { return id; }
        public void setId(long id) { this.id = id; }

        @Column(name="FIRST_NAME", length=255, nullable=false)
        public String getFirstName() {return firstName;}
        public void setFirstName(String fn) {this.firstName = fn;}

        @Column(name="LAST_NAME", length=255, nullable=false)
        public String getLastName() {return lastName;}
        public void setLastName(String ln) {this.lastName = ln;}

        }
```

JPA Entity – An example

```
@Entity
@Table(name = "CUST_TBL")
public class Customer implements java.io.Serializable{
        private long id;
        private String firstName;
        private String lastName;

        @Id
        @Column(name="CUST_ID", nullable=false, columnDefinition="long")
        public long getId() { return id; }
        public void setId(long id) { this.id = id; }

        @Column(name="FIRST_NAME", length=255, nullable=false)
        public String getFirstName() {return firstName;}
        public void setFirstName(String fn) {this.firstName = fn;}

        @Column(name="LAST_NAME", length=255, nullable=false)
        public String getLastName() {return lastName;}
        public void setLastName(String ln) {this.lastName = ln;}
```

When @column annotations are on top of getters instead of fieldnames, the values inserted into tables are picked up from what is returned from getters

Primary Key

- **A primary key is the identity of a given entity bean**
  - Every entity bean must have a primary key, and it must be unique
- **Primary keys can map to one (or more) properties and must map to one of the following types:**
  - Any Java primitive type
  - java.lang.String, or
  - a primary-key class composed of primitives and/ or strings

Primary Key

- **Natural Key**
  - Sometimes the primary **key** is made up of real data that has real business use, normally referred to as **natural keys**
  - For example, if every customer has to have a unique email id, which can be used as primary key
- **Surrogate Key**
  - No business use, but acts as a primary key
  - For example, if no column is unique, you can have another column that has unique number and saved as primary key
  - The surrogate **key** is generated at runtime when a new record is inserted into a table.
  - A **surrogate key** is typically a numeric value

Primary Key: @GeneratedValue annotation

- You can generate the primary key for your entity beans manually or have the persistence provider do it for you.
- Use @GeneratedValue annotation to tell hibernate to automatically generate a value for customerId
  - Each time you persist a Customer object – can use next value sequence
  - @javax.persistence.GeneratedValue annotation
- Don’t have to have setter method for customerId
- Example

```
@id @GeneratedValue
private int customerId;
```

Primary Key: @GeneratedValue annotation

- **Can use strategy attribute of @GeneratedValue to specify primary key generation type**
- **Four valid options are:**
  - public enum GenerationType { TABLE, SEQUENCE, IDENTITY, AUTO};
- **The GeneratorType.AUTO strategy is the most commonly used configuration, and it is the default**
- **Example**

```
@id @GeneratedValue
private int customerId;
```

Multiple Mapping with @SecondaryTable

One logical entity that is stored in two different tables

```
@Entity
@Table(name = "CUSTOMER_TABLE")
@SecondaryTable(name="ADDRESS_TABLE",
     pkJoinColumns={
      @PrimaryKeyJoinColumn(name="ADDRESS_ID")
     })
```

An Embedded Value object

- **A POJO (e.g., Address) class that does not have a persistent identity of its own and exclusively belongs to another entity (e.g., Student class.)**
  - This class is identified by @Embedded on the field in the entity class and annotated with @Embeddable in the class definition:

```
@Entity
public class Student implements Serializable {
   @Id
   private int id;
  private String name;
  private String grade;
  @Embedded
  private Address address;

  @ElementCollection
  @CollectionTable("StudentCourse")
  List<Course> courses;

  //. . .
}
```

Tells Address is embedded in Student and will be part of Student table, i.e., Address fields will be columns in Student table.

Tells this object is embedded someplace else, don't create separate table

```
@Embeddable
public class Address {
  private String street;
  private String city;
  private String zip;
  //. . .
}
```

Saving Collection

- **The @ElementCollection annotation signifies that a student's courses are listed in a different table**
  – By default, the table name is derived by combining the name of the owning class, the string "_," and the field name (e.g., Student_courses.
    - @CollectionTable can be used to override the default name of the table
    - @AtttributeOverrides can be used to override the default column names.

```
@Entity
public class Student implements Serializable {
  @Id
  private int id;
  private String name;
  private String grade;
  @Embedded
  private Address address;

  @ElementCollection
  @CollectionTable("StudentCourse")
  List<Course> courses;

  //. . .
}
```

Especially useful when you do not know how many Course objects you want to use in Student entity.

@ElementCollection can also be used for embedded class

EntityManager

- **The entities are managed by the EntityManager interface**
  - An object of this type manges data flow between the program and the database
- **The javax.persistence.EntityManager is the interface for all DB actions:**
  - Persist POJO into database
  - Create queries
  - Find objects
  - Synchronize objects
  - Cache
  - Transaction support

EntityManager Methods

Method Name	Description
persist	Makes an entity instance managed and persistent
find	Find an entity instance by executing a query by primary key on the database
contains	Returns true if the entity instance is in the PersistentContext
merge	Merges the state of a given entity into the current persistent context
remove	Removes the entity instance
flush	Forces the synchronization of the database with entities in the persistent context
refresh	Refreshes the entity instances in the persistent context from the database

Persistence Unit

- A **persistence unit** **defines a** **set of all entity classes** **that are** **managed** **by** **EntityManager** **instance in an application**
  - This set of entity classes represents the data contained within a single data store
- **Persistence units are** **defined** **by the** **persistence.xml** **configuration file**
  - An example **persistence.xml** file:

```
<persistence>
   <persistence-unit name="OrderManagement">
      <description>This unit manages orders and customers.  It does not rely on any vendor-specific       features
and can therefore be deployed to any persistence provider.
      </description>
      <jta-data-source>jdbc/MyOrderDB</jta-data-source>
      <jar-file>MyOrderApp.jar</jar-file>
      <class>com.widgets.Order</class>
      <class>com.widgets.Customer</class>
   </persistence-unit>
</persistence>
```

Other JPA interfaces/Classes

- **EntityManagerFactory interface**
  - An object of this type creates persistence unit’s EntityManager
- **Persistence class**
  - A static method of this class creates the specified persistence unit’s EntityManagerFactory

Persistent Context

- **A working copy of a persistent unit**
- **A set of managed entity instances that exist in a particular data store**
- **Defines the scope under which particular entity instances are created, persisted, and removed**
- **Each EntityManager instance is associated with a persistence context**

Container-Managed Entity Managers

- The Java EE container manages the lifecycle of container-managed entity managers
- To obtain an EntityManager instance, inject the entity manager into the application component:

```
@PersistenceContext  EntityManager  em;
```

Application-Managed Entity Managers

- **With an application-managed entity manager, the lifecycle of EntityManager instances is managed by the application**
  - Applications create EntityManager instances by using the **createEntityManager** method of **javax.persistence.EntityManagerFactory**

    ```
    @PersistenceUnit EntityManagerFactory emf;
    EntityManager em = emf.createEntityManager();
    ```

  - You can also use the following to get an instance of EntityManagerFactory

    ```
    javax.persistence.Persistence.createEntityManagerFactory(String persisitenUnitName)
    ```

Application-Managed Entity Managers

- **Such applications need to manually gain access to the JTA transaction manager and add transaction demarcation information when performing entity operations**

- **The javax.transaction.UserTransaction interface defines methods to begin, commit, and roll back transactions**

Application-Managed Entity Managers

- **Inject an instance of UserTransaction by creating an instance variable annotated with @Resource:**

  @Resource UserTransaction utx;

  - To begin a transaction, call the utx.begin() method.
  - When all the entity operations are complete, call the utx.commit() method to commit the transaction.
  - The utx.rollback() method is used to roll back the current transaction.
- **Alternatively, you can use EntityManager's getTransaction() method to call begin(), commit(), rollback() --** See an example later

Application-Managed Entity Managers

- The following example shows how to manage transactions in an application that uses an application-managed entity manager:

```
@PersistenceContext
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
        utx.begin();
        em.persist(SomeEntity);
        em.merge(AnotherEntity);
        em.remove(ThirdEntity);
        utx.commit();
} catch (Exception e) {
        utx.rollback();
}
```

EntityManager Services

Persisting Entities

```
Customer cust = new Customer();
cust.setFirstName("Greg");
entityManager.persist(cust);
```

Finding Entities

```
find(Class clazz, Object primkey)
getReference(Class clazz, Object primkey)
createQuery(String qry)
```

Update Entities

```
Customer cust = entityManager.find(....);
cust.setXXXX(...);
```

Merging Entities

Takes a detached object and merges changes back to persistent storage.

```
    entityManager.merge(obj);
```

EntityManager Services

Remove Entities

```
Customer cust = entityManager.find(....);
entityManager.remove(cust);
```

Refreshing Entities

Refreshes the state of an object from the persistent storage.

```
entityManager.refresh(obj)
```

contains() and clear()

contains(Object entity) – determine whether an object is managed by the entity manager.

clear() detach an object from the entity manager.

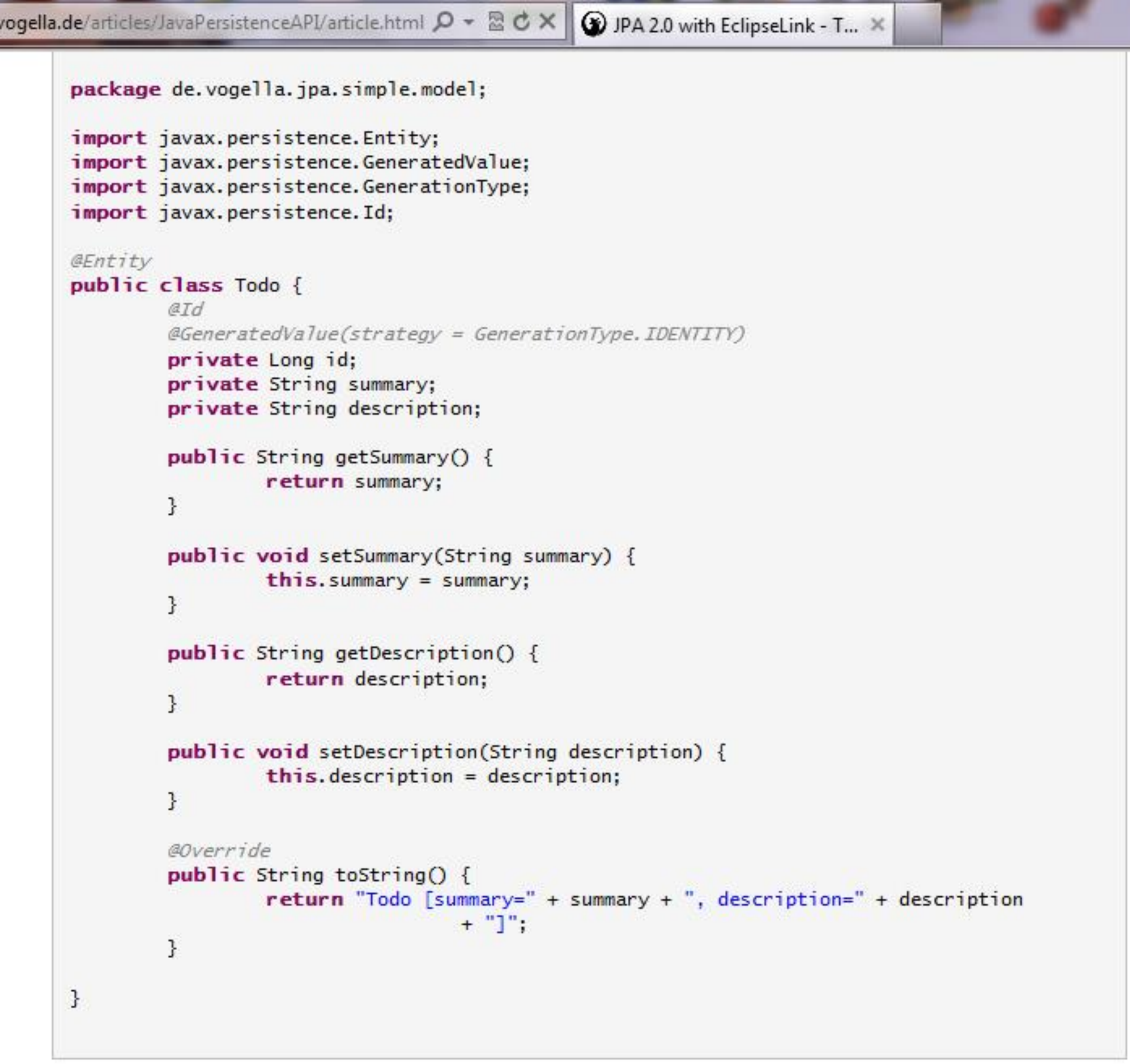
Flushing Data

flush() - force synchronization with persistent storage

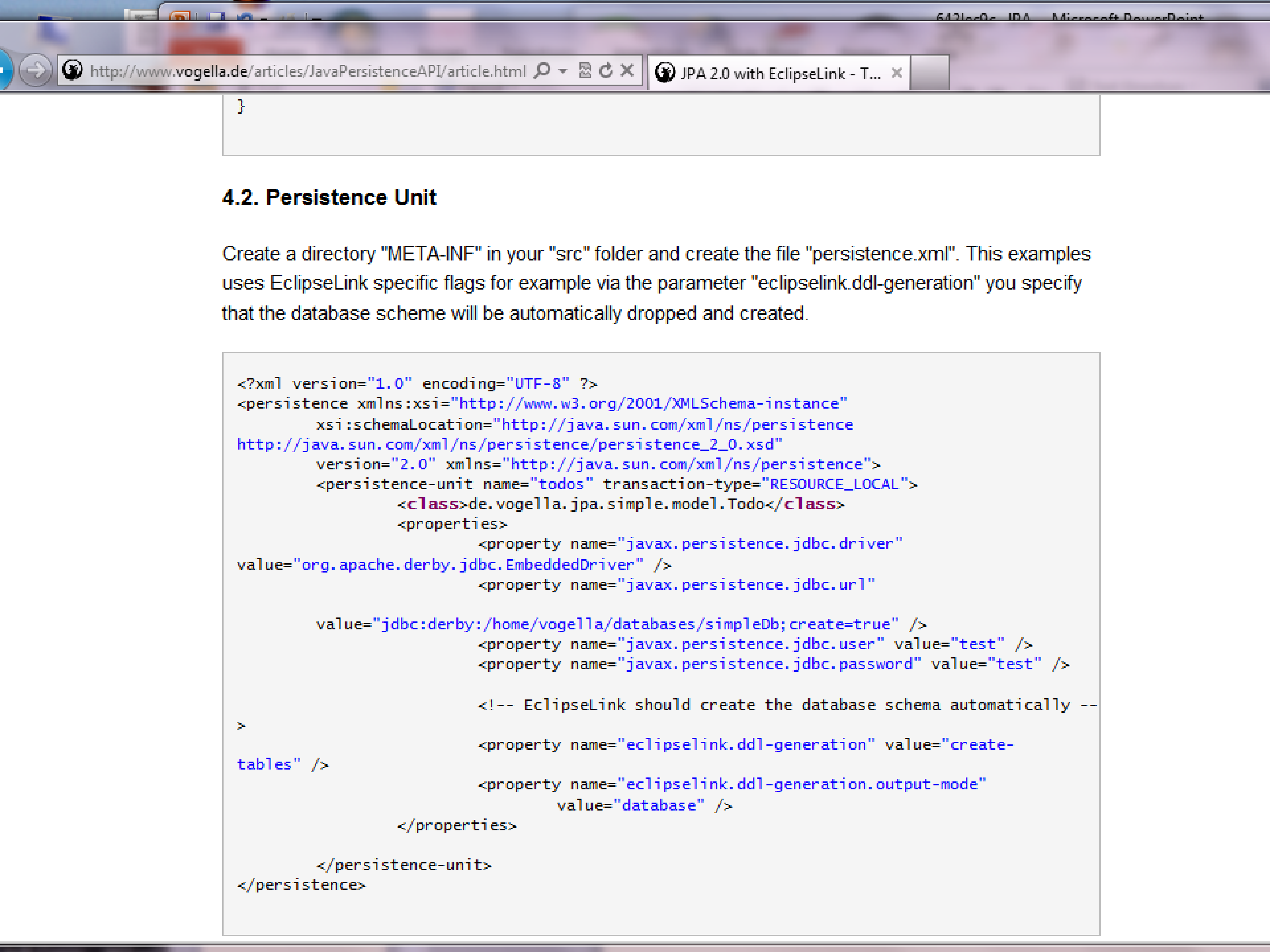
FlushModeType - AUTO, COMMIT

JPA Implementation Example using EclipseLink

JPA Entity: Todo


ogella.de/articles/JavaPersistenceAPI/article.html
JPA 2.0 with EclipseLink - T...

```java
package de.vogella.jpa.simple.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Todo {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	private String summary;
	private String description;

	public String getSummary() {
		return summary;
	}

	public void setSummary(String summary) {
		this.summary = summary;
	}

	public String getDescription() {
		return description;
	}

	public void setDescription(String description) {
		this.description = description;
	}

	@Override
	public String toString() {
		return "Todo [summary=" + summary + ", description=" + description
				+ "]";
	}

}
```

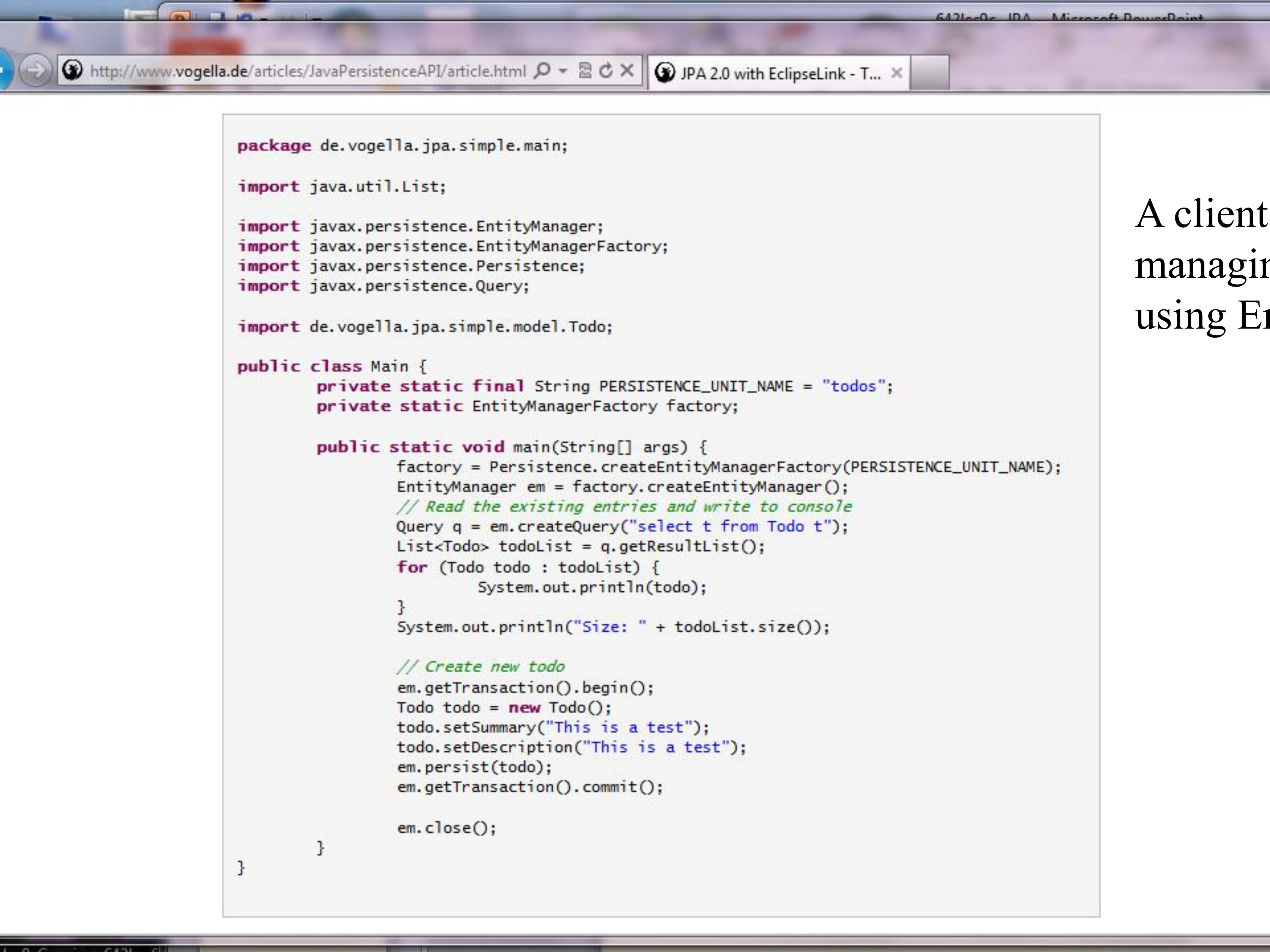

http://www.vogella.de/articles/JavaPersistenceAPI/article.html
JPA 2.0 with EclipseLink - T...

```
}
```

4.2. Persistence Unit

Create a directory "META-INF" in your "src" folder and create the file "persistence.xml". This examples uses EclipseLink specific flags for example via the parameter "eclipselink.ddl-generation" you specify that the database scheme will be automatically dropped and created.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
        version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
        <persistence-unit name="todos" transaction-type="RESOURCE_LOCAL">
                <class>de.vogella.jpa.simple.model.Todo</class>
                <properties>
                        <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.EmbeddedDriver" />
                        <property name="javax.persistence.jdbc.url"

        value="jdbc:derby:/home/vogella/databases/simpleDb;create=true" />
                        <property name="javax.persistence.jdbc.user" value="test" />
                        <property name="javax.persistence.jdbc.password" value="test" />

                        <!-- EclipseLink should create the database schema automatically --
>
                        <property name="eclipselink.ddl-generation" value="create-
tables" />
                        <property name="eclipselink.ddl-generation.output-mode"
                                value="database" />
                </properties>

        </persistence-unit>
</persistence>
```


http://www.vogella.de/articles/JavaPersistenceAPI/article.html
JPA 2.0 with EclipseLink - T...

```
package de.vogella.jpa.simple.main;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

import de.vogella.jpa.simple.model.Todo;

public class Main {
        private static final String PERSISTENCE_UNIT_NAME = "todos";
        private static EntityManagerFactory factory;

        public static void main(String[] args) {
                factory = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
                EntityManager em = factory.createEntityManager();
                // Read the existing entries and write to console
                Query q = em.createQuery("select t from Todo t");
                List<Todo> todoList = q.getResultList();
                for (Todo todo : todoList) {
                        System.out.println(todo);
                }
                System.out.println("Size: " + todoList.size());

                // Create new todo
                em.getTransaction().begin();
                Todo todo = new Todo();
                todo.setSummary("This is a test");
                todo.setDescription("This is a test");
                em.persist(todo);
                em.getTransaction().commit();

                em.close();
        }
}
```

A client
managin
using E

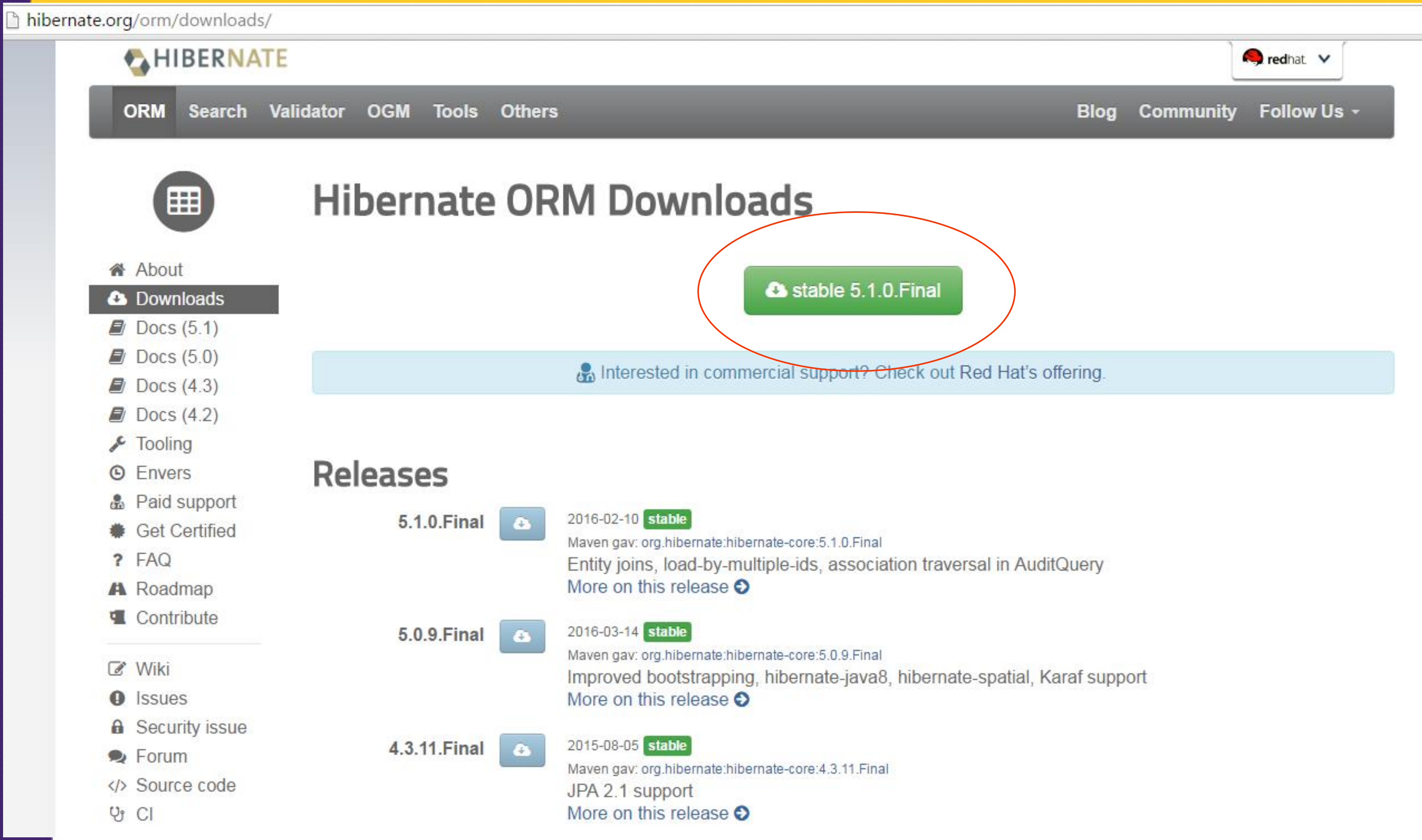
JPA Providers

- Hibernate
- EclipseLink -- reference implementation
- Apache OpenJPA

JPA Providers

- **Hibernate**
  - To use hibernate, please Download hibernate (zip file) www.hibernate.org/downloads and jar files to project build path/classpath and add them to lib folder of your application
  - On download page, click on release bundle, e.g., 3.6.4…..zip and, which has
    - has lib folder which has jars needed by hibernate
    - You will need all jars in lib/jpa and lib/required folders
  - Also, download JDBC driver (of database choice) and add it to your project's java build path
    - This configures hibernate to use JDBC driver (jar) to connect to database

JPA Providers


hibernate.org/orm/downloads/

Hibernate ORM Downloads

Interested in commercial support? Check out Red Hat's offering.

Releases

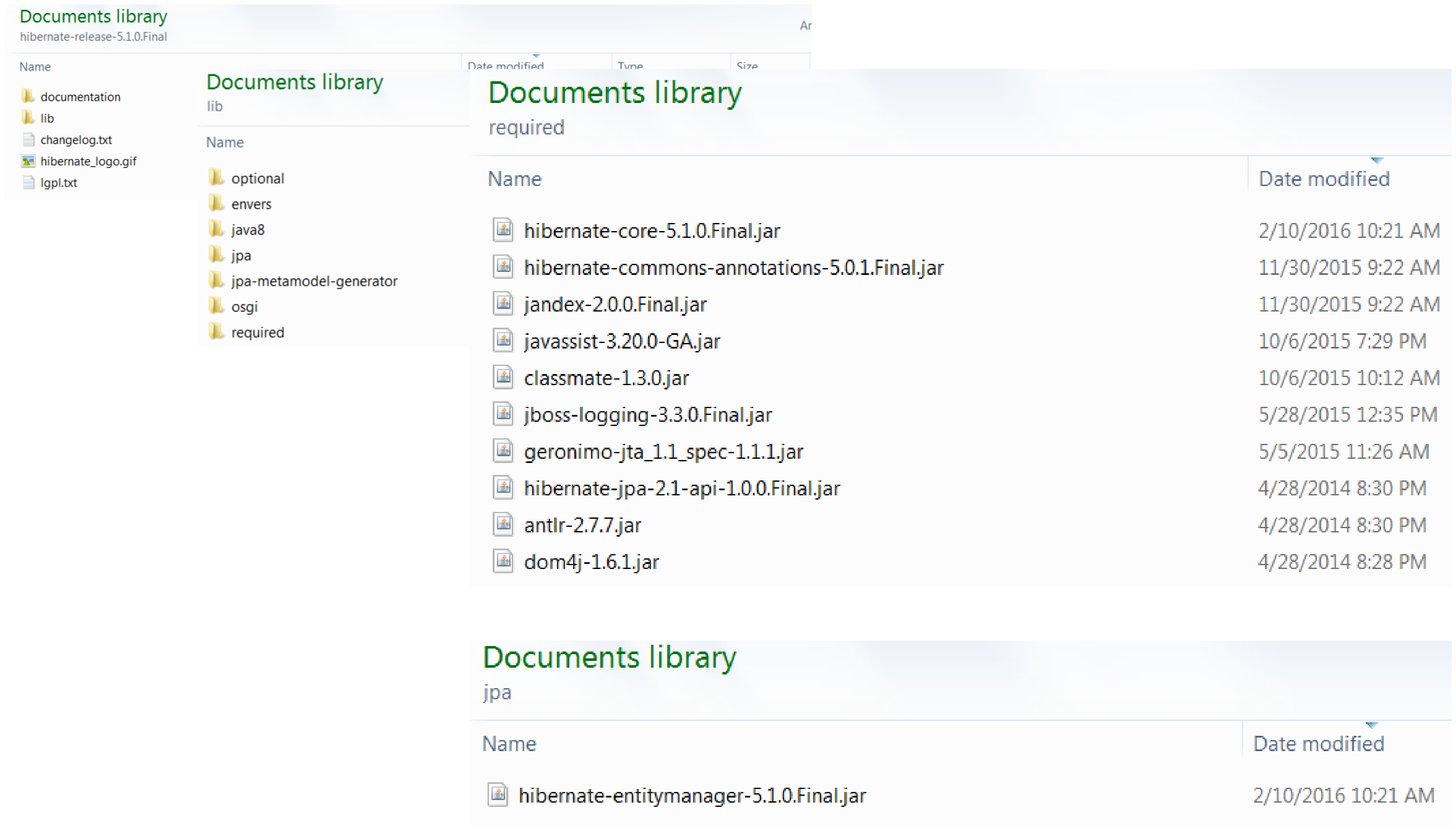
5.1.0.Final — 2016-02-10 stable
Maven gav: org.hibernate:hibernate-core:5.1.0.Final
Entity joins, load-by-multiple-ids, association traversal in AuditQuery
More on this release

5.0.9.Final — 2016-03-14 stable
Maven gav: org.hibernate:hibernate-core:5.0.9.Final
Improved bootstrapping, hibernate-java8, hibernate-spatial, Karaf support
More on this release

4.3.11.Final — 2015-08-05 stable
Maven gav: org.hibernate:hibernate-core:4.3.11.Final
JPA 2.1 support
More on this release

Adding Hibernate/JPA jars to your classpath in eclipse

- Unzip the downloaded file. Your jars are inside lib folder, specifically inside lib/required and lib/jpa



Documents library
hibernate-release-5.1.0.Final

Name	Date modified	Type	Size
documentation			
lib			
changelog.txt			
hibernate_logo.gif			
lgpl.txt			

Documents library
lib

Name
optional
envers
java8
jpa
jpa-metamodel-generator
osgi
required

Documents library
required

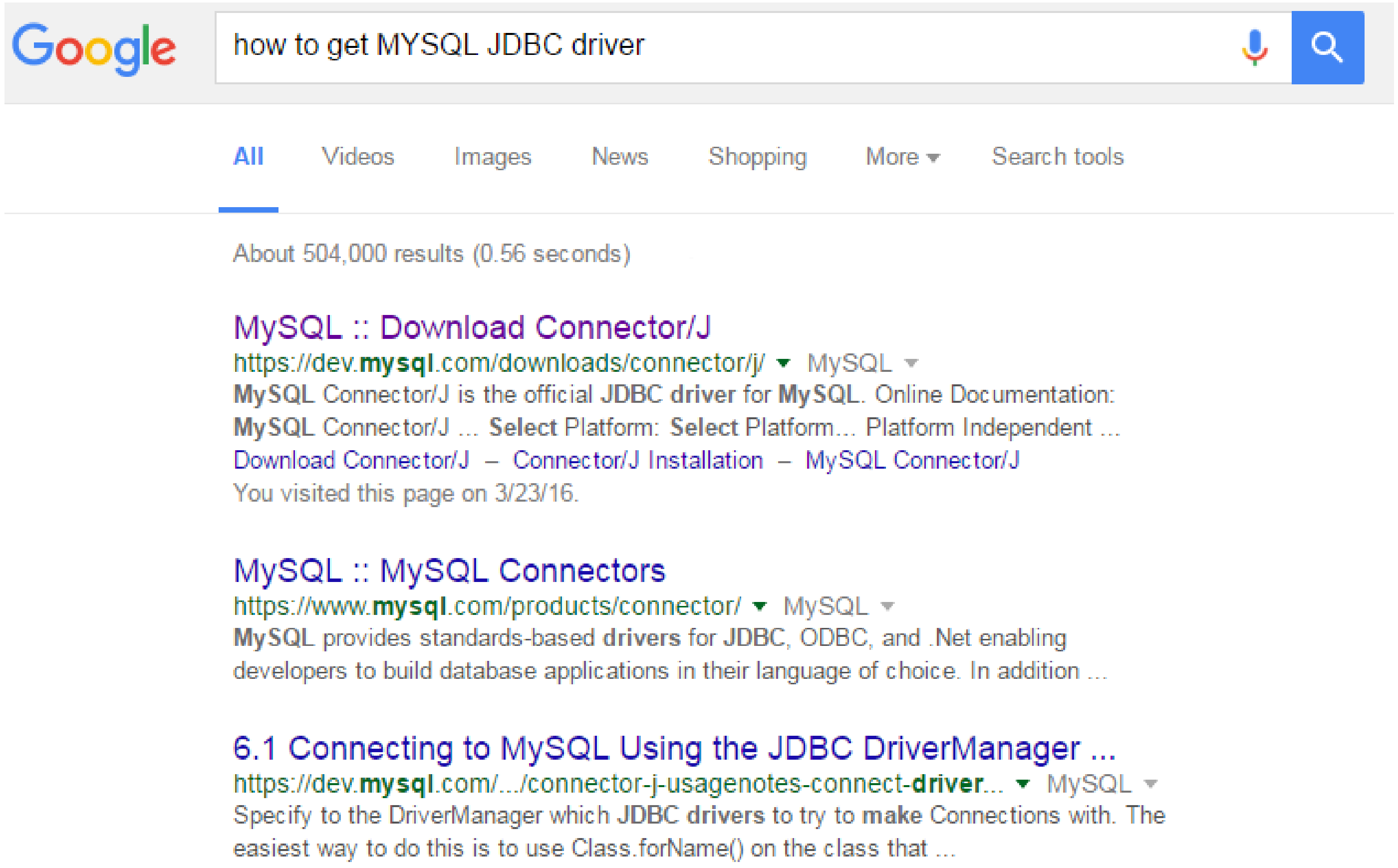
Name	Date modified
hibernate-core-5.1.0.Final.jar	2/10/2016 10:21 AM
hibernate-commons-annotations-5.0.1.Final.jar	11/30/2015 9:22 AM
jandex-2.0.0.Final.jar	11/30/2015 9:22 AM
javassist-3.20.0-GA.jar	10/6/2015 7:29 PM
classmate-1.3.0.jar	10/6/2015 10:12 AM
jboss-logging-3.3.0.Final.jar	5/28/2015 12:35 PM
geronimo-jta_1.1_spec-1.1.1.jar	5/5/2015 11:26 AM
hibernate-jpa-2.1-api-1.0.0.Final.jar	4/28/2014 8:30 PM
antlr-2.7.7.jar	4/28/2014 8:30 PM
dom4j-1.6.1.jar	4/28/2014 8:28 PM

Documents library
jpa

Name	Date modified
hibernate-entitymanager-5.1.0.Final.jar	2/10/2016 10:21 AM

Download MySQL JDBC driver and add it to Java Build Path of your project in Eclipse

- **Here is what I got in Google search – click the first link “MySQL::Download Connector/J”**



About 504,000 results (0.56 seconds)

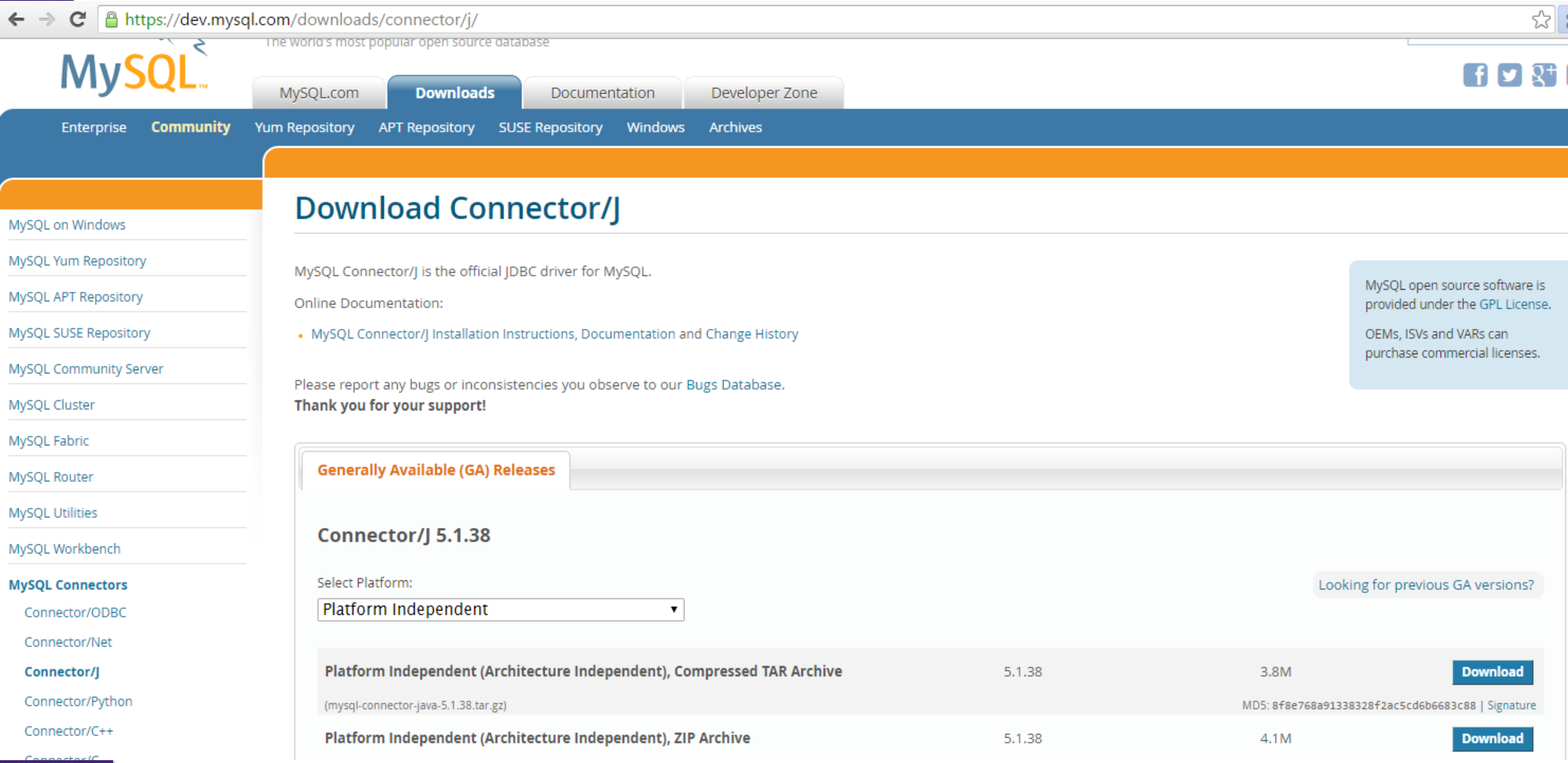
MySQL :: Download Connector/J
https://dev.mysql.com/downloads/connector/j/ ▾ MySQL ▾
MySQL Connector/J is the official JDBC driver for MySQL. Online Documentation: MySQL Connector/J ... Select Platform: Select Platform... Platform Independent ...
Download Connector/J – Connector/J Installation – MySQL Connector/J
You visited this page on 3/23/16.

MySQL :: MySQL Connectors
https://www.mysql.com/products/connector/ ▾ MySQL ▾
MySQL provides standards-based drivers for JDBC, ODBC, and .Net enabling developers to build database applications in their language of choice. In addition ...

6.1 Connecting to MySQL Using the JDBC DriverManager ...
https://dev.mysql.com/.../connector-j-usagenotes-connect-driver... ▾ MySQL ▾
Specify to the DriverManager which JDBC drivers to try to make Connections with. The easiest way to do this is to use Class.forName() on the class that ...

Download MySQL JDBC driver and add it to Java Build Path of your project in Eclipse

- Download the zip file (the 2nd Download link)



https://dev.mysql.com/downloads/connector/j/

The world's most popular open source database

MySQL.com | Downloads | Documentation | Developer Zone

Enterprise | Community | Yum Repository | APT Repository | SUSE Repository | Windows | Archives

MySQL on Windows
MySQL Yum Repository
MySQL APT Repository
MySQL SUSE Repository
MySQL Community Server
MySQL Cluster
MySQL Fabric
MySQL Router
MySQL Utilities
MySQL Workbench
MySQL Connectors
Connector/ODBC
Connector/Net
Connector/J
Connector/Python
Connector/C++

Download Connector/J

MySQL Connector/J is the official JDBC driver for MySQL.

Online Documentation:

- MySQL Connector/J Installation Instructions, Documentation and Change History

Please report any bugs or inconsistencies you observe to our Bugs Database.
Thank you for your support!

MySQL open source software is provided under the GPL License.

OEMs, ISVs and VARs can purchase commercial licenses.

Generally Available (GA) Releases

Connector/J 5.1.38

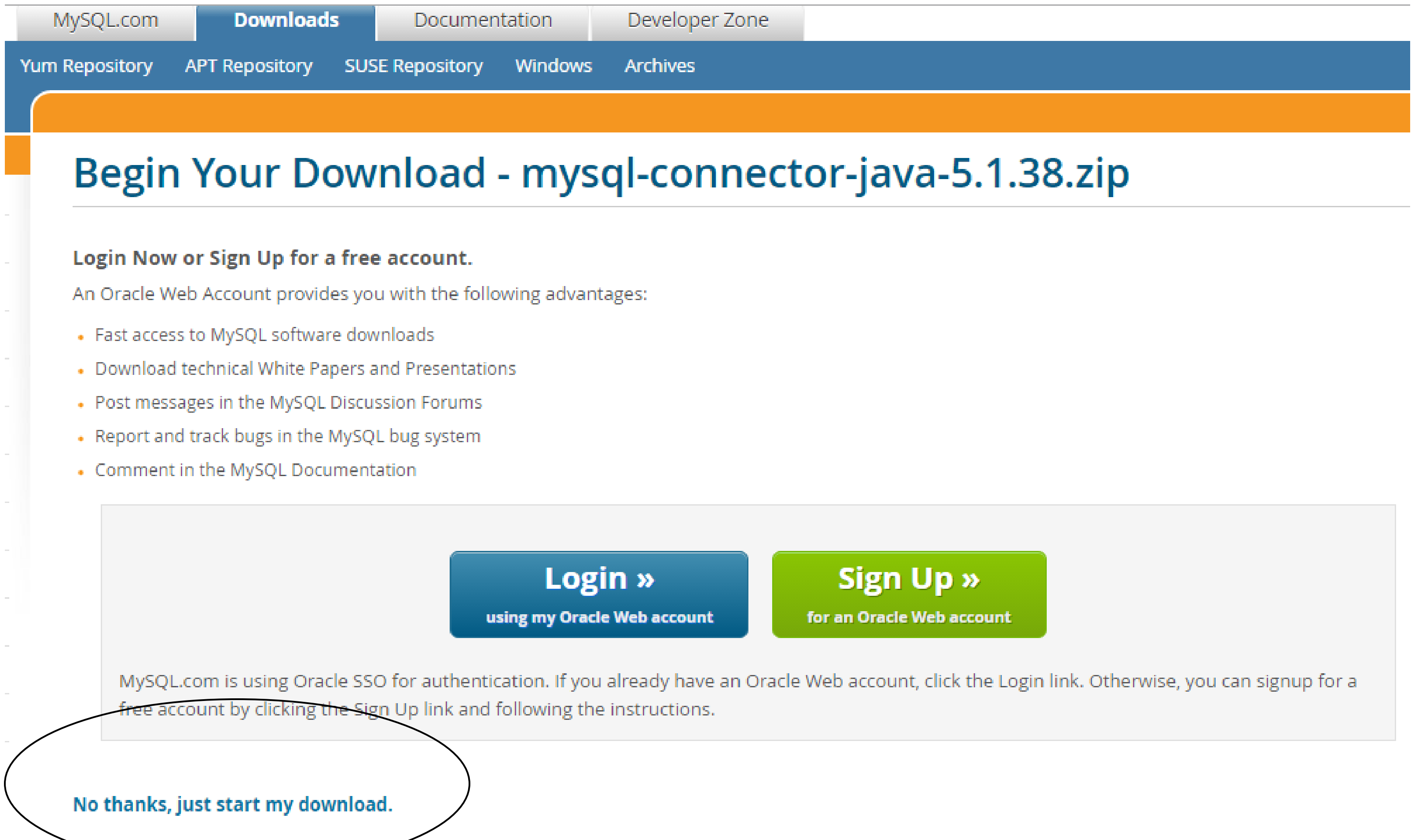
Select Platform: Platform Independent

Looking for previous GA versions?

Platform Independent (Architecture Independent), Compressed TAR Archive (mysql-connector-java-5.1.38.tar.gz)	5.1.38	3.8M	Download MD5: 8f8e768a91338328f2ac5cd6b6683c88 | Signature
Platform Independent (Architecture Independent), ZIP Archive	5.1.38	4.1M	Download

Download MySQL JDBC driver and add it to Java Build Path of your project in Eclipse

- On the page that follows, click on "No thanks, just start my download"



Begin Your Download - mysql-connector-java-5.1.38.zip

Login Now or Sign Up for a free account.

An Oracle Web Account provides you with the following advantages:

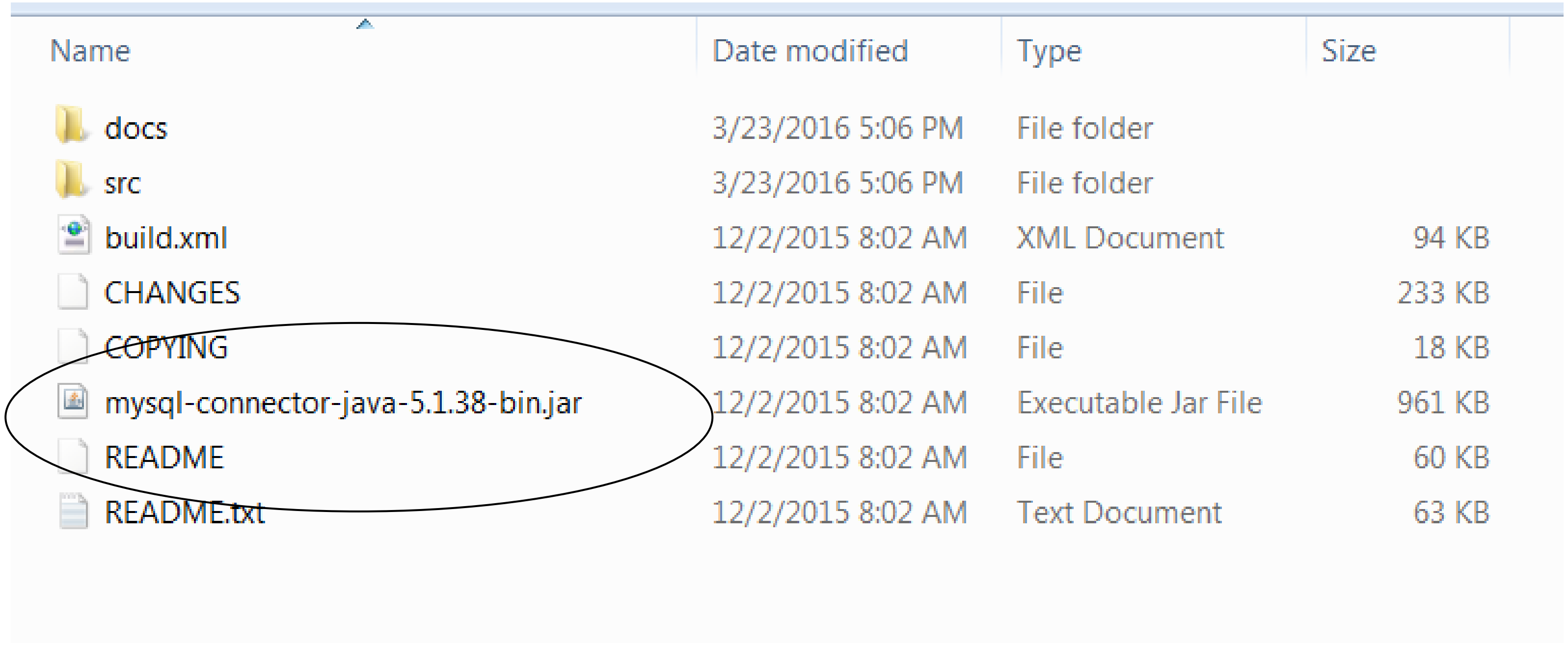
- Fast access to MySQL software downloads
- Download technical White Papers and Presentations
- Post messages in the MySQL Discussion Forums
- Report and track bugs in the MySQL bug system
- Comment in the MySQL Documentation

MySQL.com is using Oracle SSO for authentication. If you already have an Oracle Web account, click the Login link. Otherwise, you can signup for a free account by clicking the Sign Up link and following the instructions.

No thanks, just start my download.

Download MySQL JDBC driver and add it to Java Build Path of your project in Eclipse

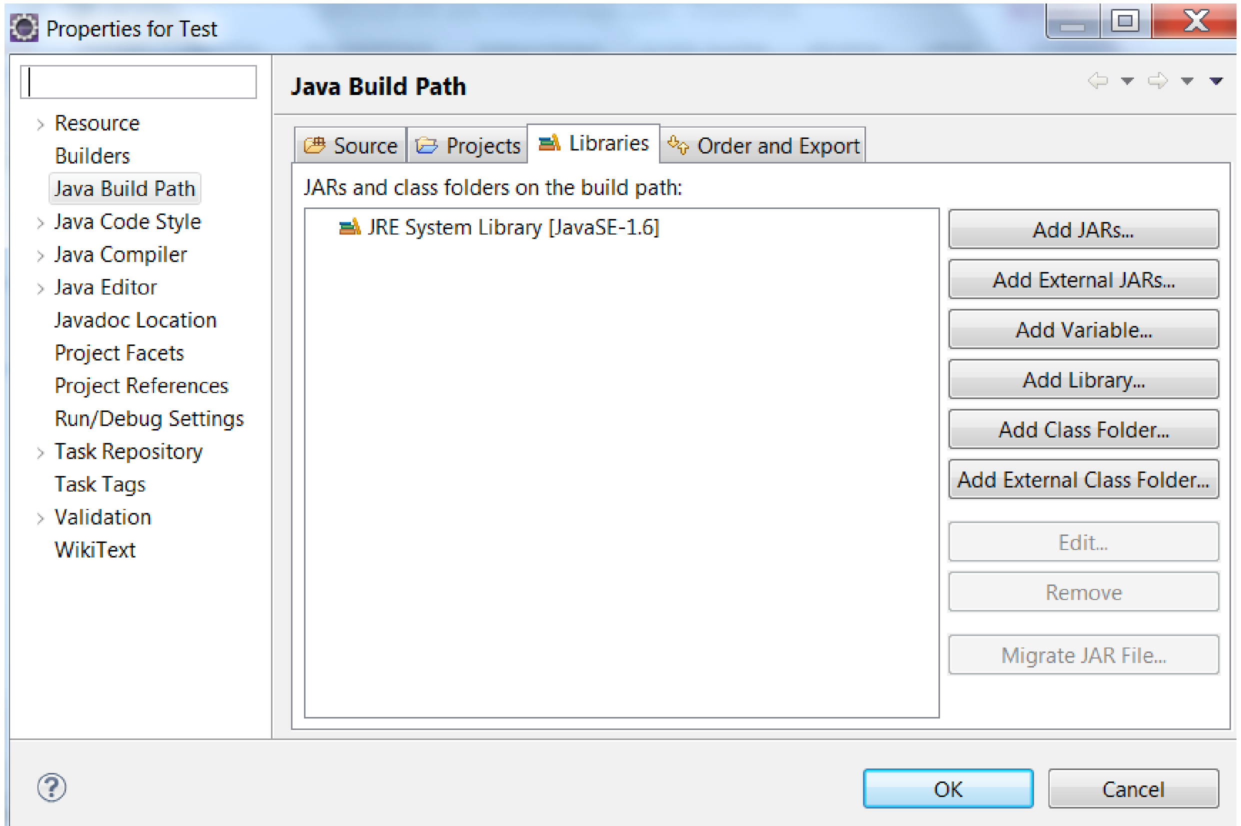
- **Unzipped downloaded zip file contains one jar file, which is your jdbc driver**
- **You must add this jar file to Java Build Path of your project in Eclipse**



Name	Date modified	Type	Size
docs	3/23/2016 5:06 PM	File folder	
src	3/23/2016 5:06 PM	File folder	
build.xml	12/2/2015 8:02 AM	XML Document	94 KB
CHANGES	12/2/2015 8:02 AM	File	233 KB
COPYING	12/2/2015 8:02 AM	File	18 KB
mysql-connector-java-5.1.38-bin.jar	12/2/2015 8:02 AM	Executable Jar File	961 KB
README	12/2/2015 8:02 AM	File	60 KB
README.txt	12/2/2015 8:02 AM	Text Document	63 KB

Adding Hibernate/JPA jars to your classpath in eclipse

- **Select or Click on your Java Project or Dynamic Web Project name in Eclipse, and the select Project->Properties for your project**
  - **You will see something like this**


Properties for Test

Java Build Path

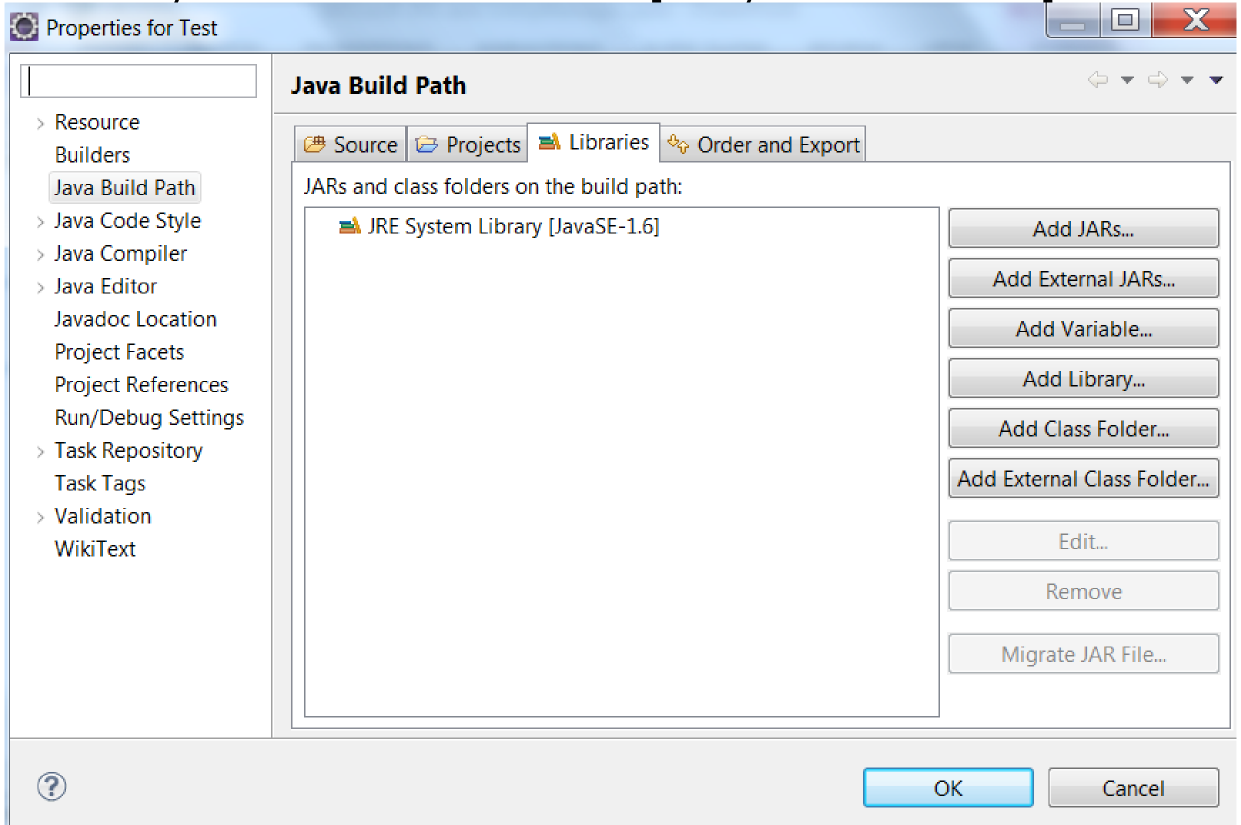
Source | Projects | Libraries | Order and Export

JARs and class folders on the build path:

JRE System Library [JavaSE-1.6]

Adding Hibernate/JPA jars to your classpath in eclipse

- **Now select Java Build Path (in the left panel) and the Library tab (as shown below)**
- **Then use Add External Jars link to add all jars in lib/required and lib/jpa folder of your hibernate deployment and press OK**



Properties for Test

Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project Facets
Project References
Run/Debug Settings
Task Repository
Task Tags
Validation
WikiText

Java Build Path

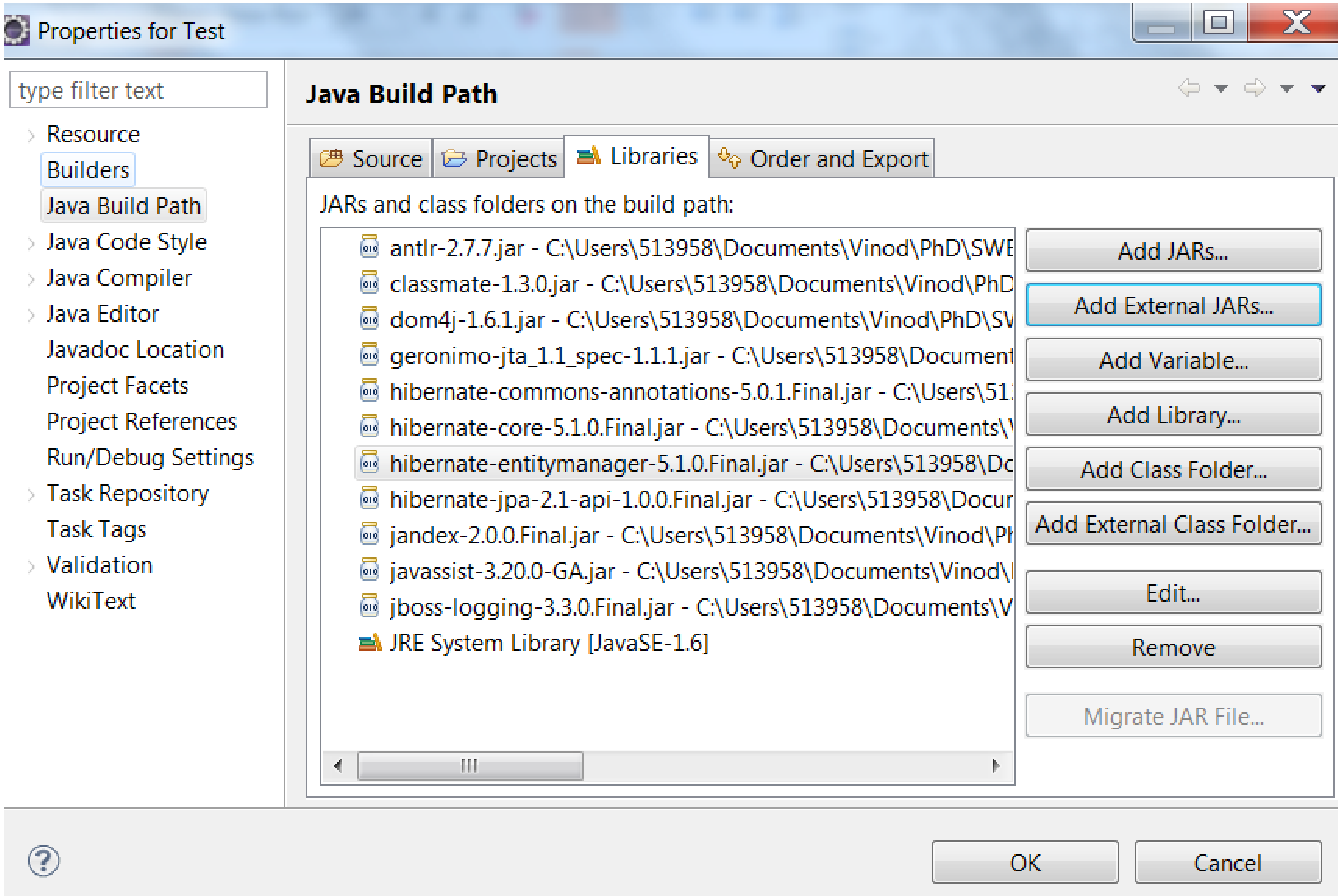
Source | Projects | Libraries | Order and Export

JARs and class folders on the build path:

JRE System Library [JavaSE-1.6]

Adding Hibernate/JPA jars to your classpath in eclipse

- **After adding the jar files, you should see something like as shown below – now your project's class path has hibernate-core jar and all its dependencies jars as well as jpa jars!**


Properties for Test

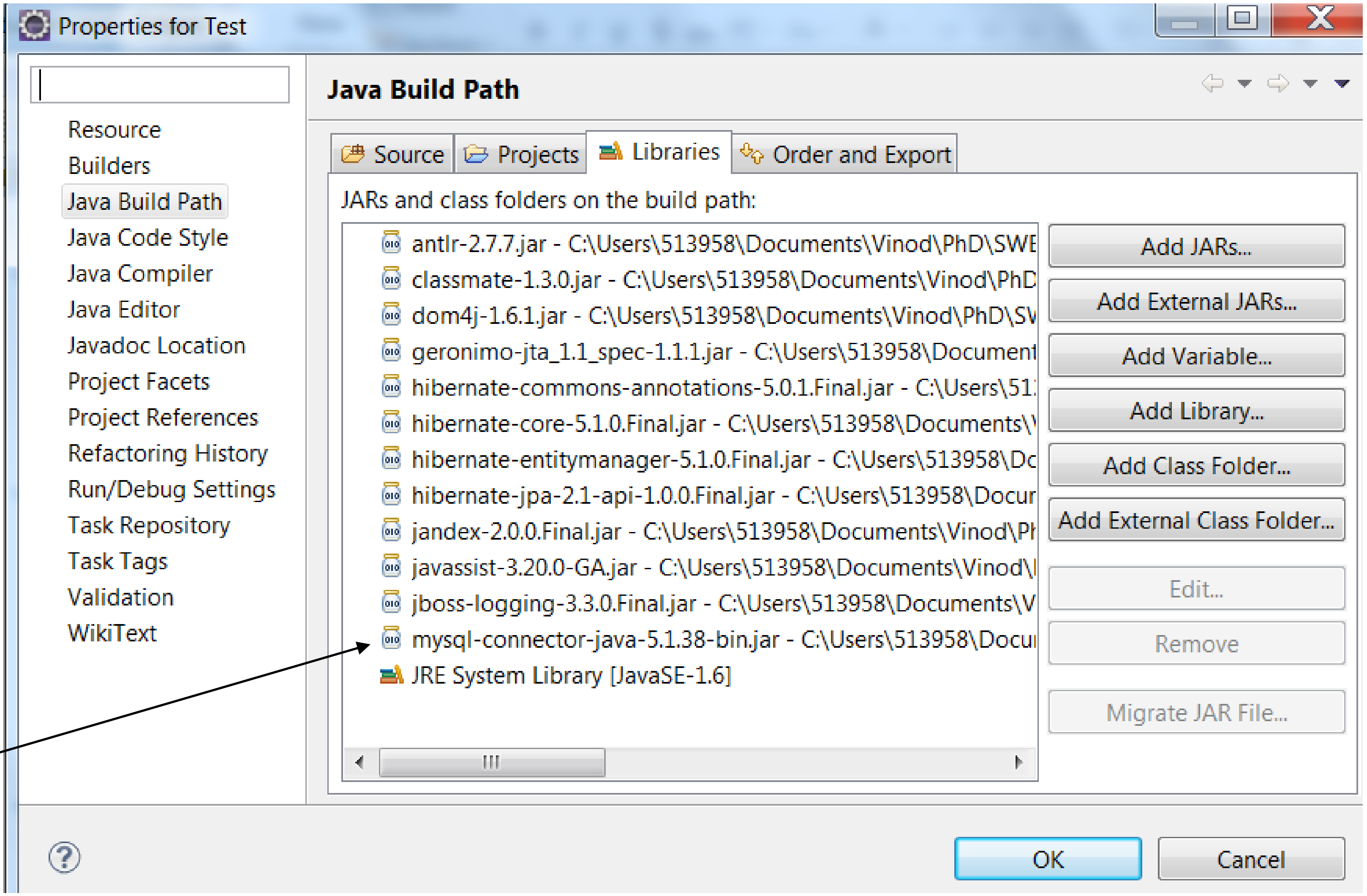
Java Build Path

JARs and class folders on the build path:

```
antlr-2.7.7.jar - C:\Users\513958\Documents\Vinod\PhD\SWE
classmate-1.3.0.jar - C:\Users\513958\Documents\Vinod\PhD
dom4j-1.6.1.jar - C:\Users\513958\Documents\Vinod\PhD\SV
geronimo-jta_1.1_spec-1.1.1.jar - C:\Users\513958\Document
hibernate-commons-annotations-5.0.1.Final.jar - C:\Users\51
hibernate-core-5.1.0.Final.jar - C:\Users\513958\Documents\
hibernate-entitymanager-5.1.0.Final.jar - C:\Users\513958\Dc
hibernate-jpa-2.1-api-1.0.0.Final.jar - C:\Users\513958\Docur
jandex-2.0.0.Final.jar - C:\Users\513958\Documents\Vinod\Pł
javassist-3.20.0-GA.jar - C:\Users\513958\Documents\Vinod\
jboss-logging-3.3.0.Final.jar - C:\Users\513958\Documents\V
JRE System Library [JavaSE-1.6]
```

Adding Hibernate/JPA jars to your classpath in eclipse

- **Repeat the same steps to add MySQL JDBC Driver jar**



Properties for Test

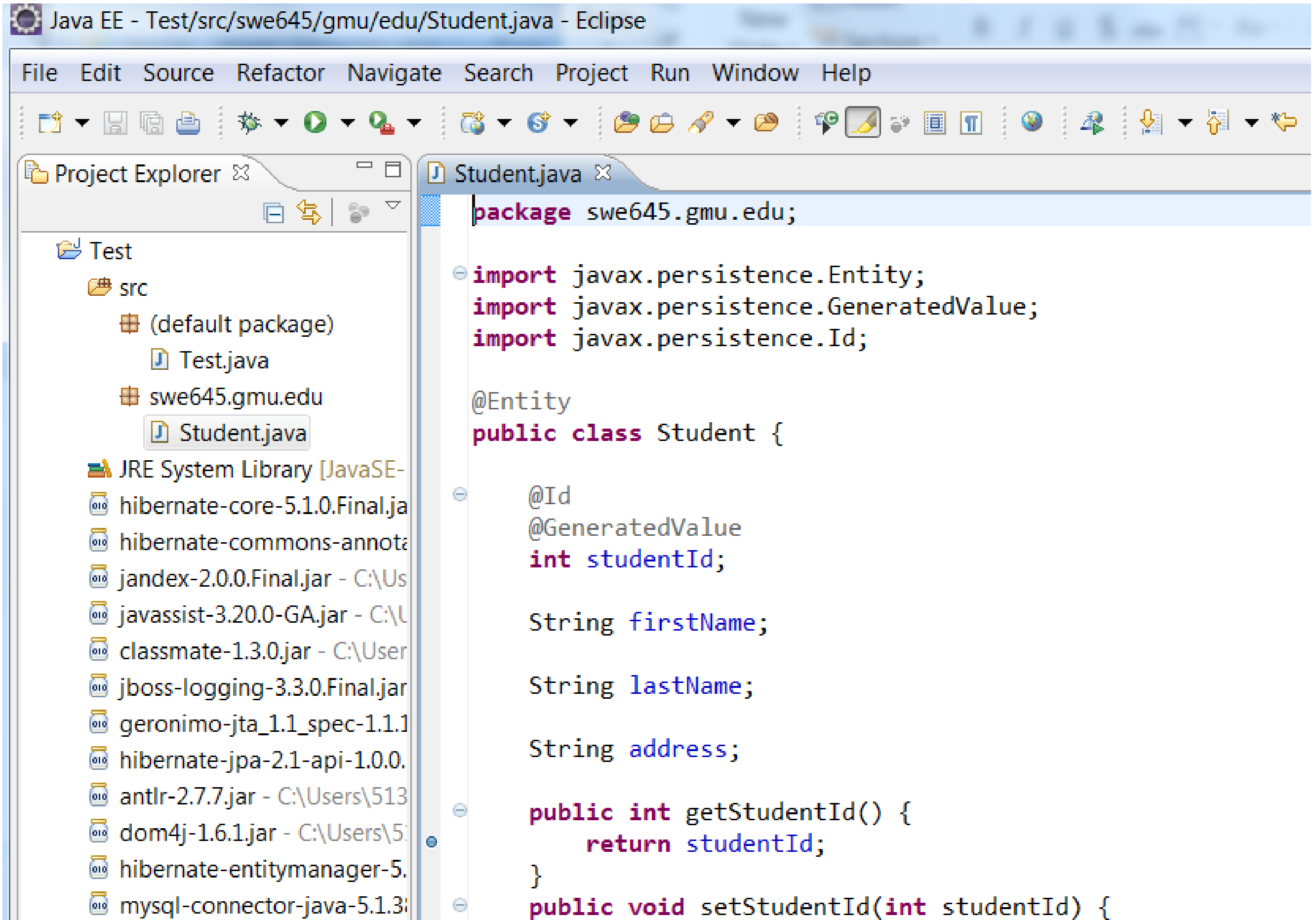
Java Build Path

JARs and class folders on the build path:

- antlr-2.7.7.jar - C:\Users\513958\Documents\Vinod\PhD\SWE
- classmate-1.3.0.jar - C:\Users\513958\Documents\Vinod\PhD
- dom4j-1.6.1.jar - C:\Users\513958\Documents\Vinod\PhD\SV
- geronimo-jta_1.1_spec-1.1.1.jar - C:\Users\513958\Document
- hibernate-commons-annotations-5.0.1.Final.jar - C:\Users\51
- hibernate-core-5.1.0.Final.jar - C:\Users\513958\Documents\
- hibernate-entitymanager-5.1.0.Final.jar - C:\Users\513958\Dc
- hibernate-jpa-2.1-api-1.0.0.Final.jar - C:\Users\513958\Docur
- jandex-2.0.0.Final.jar - C:\Users\513958\Documents\Vinod\Pl
- javassist-3.20.0-GA.jar - C:\Users\513958\Documents\Vinod\
- jboss-logging-3.3.0.Final.jar - C:\Users\513958\Documents\V
- mysql-connector-java-5.1.38-bin.jar - C:\Users\513958\Docu
- JRE System Library [JavaSE-1.6]

Your project is now ready to create JPA/Hibernate entity

- You can now create POJO and use JPA/hibernate annotations to create your JPA or hibernate entity
  - Make sure to use import javax.persistence.*



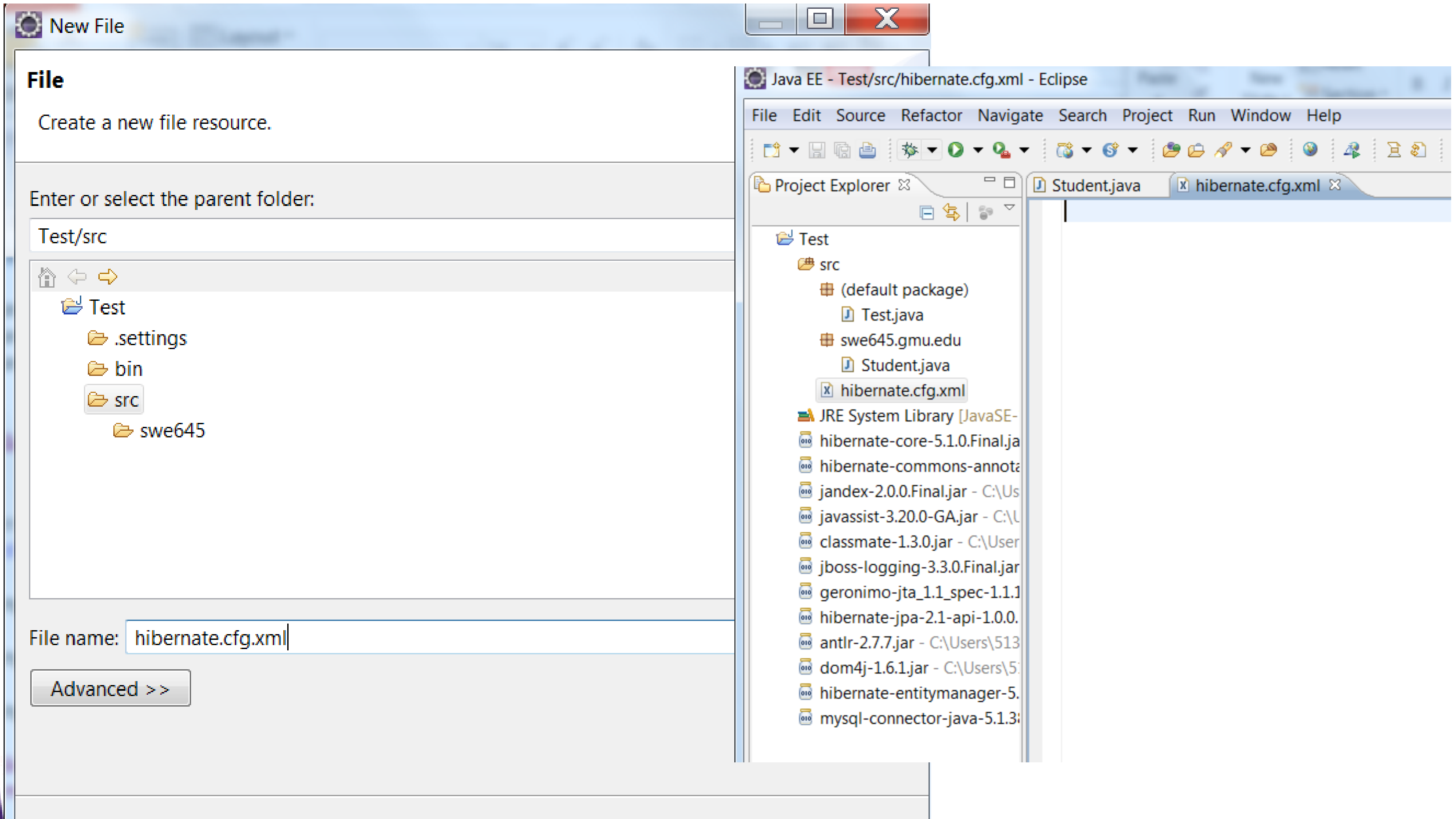
Java EE - Test/src/swe645/gmu/edu/Student.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
Student.java

Test
src
(default package)
Test.java
swe645.gmu.edu
Student.java
JRE System Library [JavaSE-
hibernate-core-5.1.0.Final.ja
hibernate-commons-annota
jandex-2.0.0.Final.jar - C:\Us
javassist-3.20.0-GA.jar - C:\U
classmate-1.3.0.jar - C:\User
jboss-logging-3.3.0.Final.jar
geronimo-jta_1.1_spec-1.1.1
hibernate-jpa-2.1-api-1.0.0.
antlr-2.7.7.jar - C:\Users\513
dom4j-1.6.1.jar - C:\Users\5
hibernate-entitymanager-5.
mysql-connector-java-5.1.3

```
package swe645.gmu.edu;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Student {

    @Id
    @GeneratedValue
    int studentId;

    String firstName;

    String lastName;

    String address;

    public int getStudentId() {
        return studentId;
    }
    public void setStudentId(int studentId) {
```

Create Hibernate JPA Configuration file: persistence.xml

- R-Click on your Java Project in Eclipse, select New-File and give it a name, persistence.xml, save it in src/META folder



New File

File

Create a new file resource.

Enter or select the parent folder:

Test/src

- Test
  - .settings
  - bin
  - src
    - swe645

File name: hibernate.cfg.xml

Advanced >>

Java EE - Test/src/hibernate.cfg.xml - Eclipse

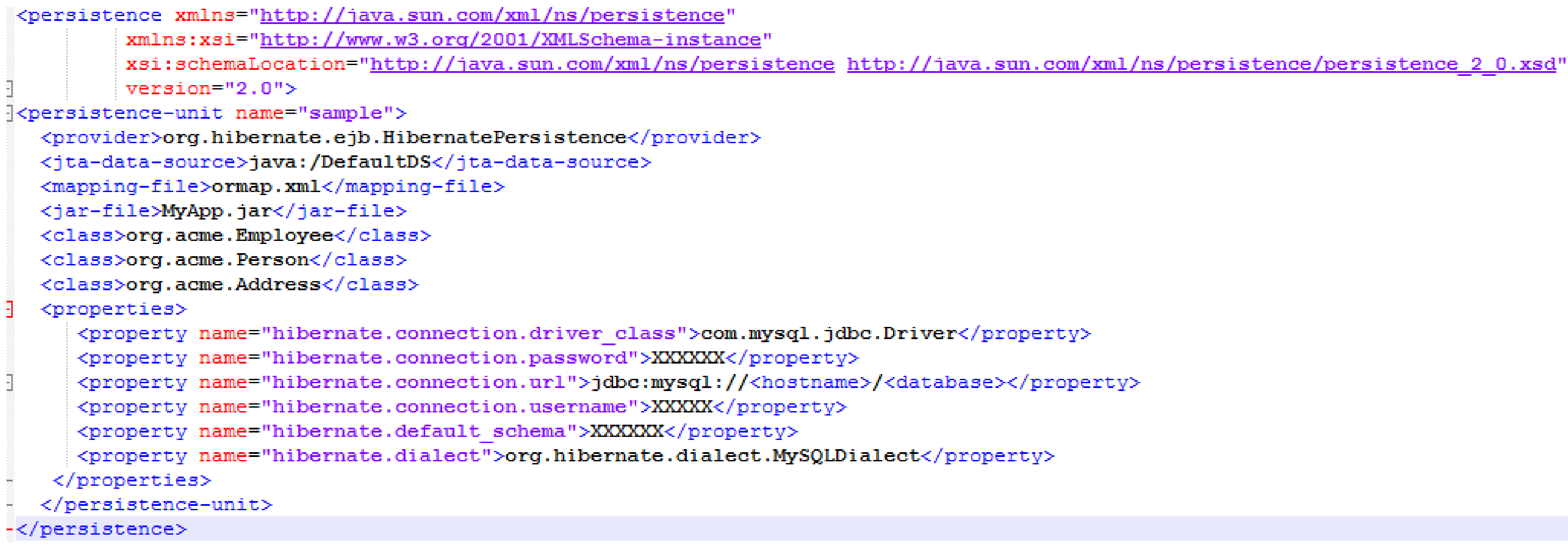
File Edit Source Refactor Navigate Search Project Run Window Help

Project Explorer | Student.java | hibernate.cfg.xml

- Test
  - src
    - (default package)
      - Test.java
    - swe645.gmu.edu
      - Student.java
    - hibernate.cfg.xml
  - JRE System Library [JavaSE-
  - hibernate-core-5.1.0.Final.ja
  - hibernate-commons-annota
  - jandex-2.0.0.Final.jar - C:\Us
  - javassist-3.20.0-GA.jar - C:\U
  - classmate-1.3.0.jar - C:\User
  - jboss-logging-3.3.0.Final.jar
  - geronimo-jta_1.1_spec-1.1.1
  - hibernate-jpa-2.1-api-1.0.0.
  - antlr-2.7.7.jar - C:\Users\513
  - dom4j-1.6.1.jar - C:\Users\5
  - hibernate-entitymanager-5.
  - mysql-connector-java-5.1.3

Create Hibernate JPA Configuration file: persistence.xml

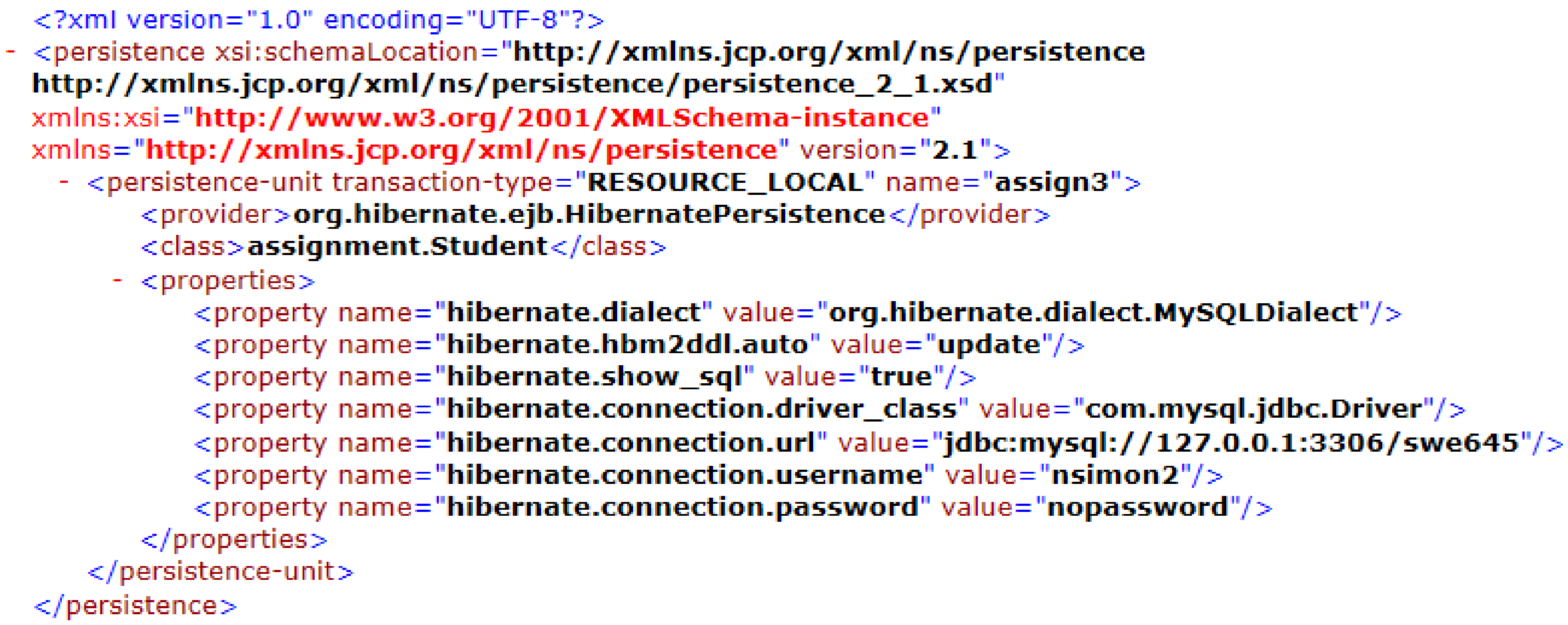
- Below is a template example of persistence.xml, which resides in the META-INF folder.



```xml
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
         version="2.0">
<persistence-unit name="sample">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>java:/DefaultDS</jta-data-source>
  <mapping-file>ormap.xml</mapping-file>
  <jar-file>MyApp.jar</jar-file>
  <class>org.acme.Employee</class>
  <class>org.acme.Person</class>
  <class>org.acme.Address</class>
  <properties>
     <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
     <property name="hibernate.connection.password">XXXXXX</property>
     <property name="hibernate.connection.url">jdbc:mysql://<hostname>/<database></property>
     <property name="hibernate.connection.username">XXXXX</property>
     <property name="hibernate.default_schema">XXXXXX</property>
     <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
   </properties>
  </persistence-unit>
</persistence>
```

Create Hibernate JPA Configuration file: persistence.xml

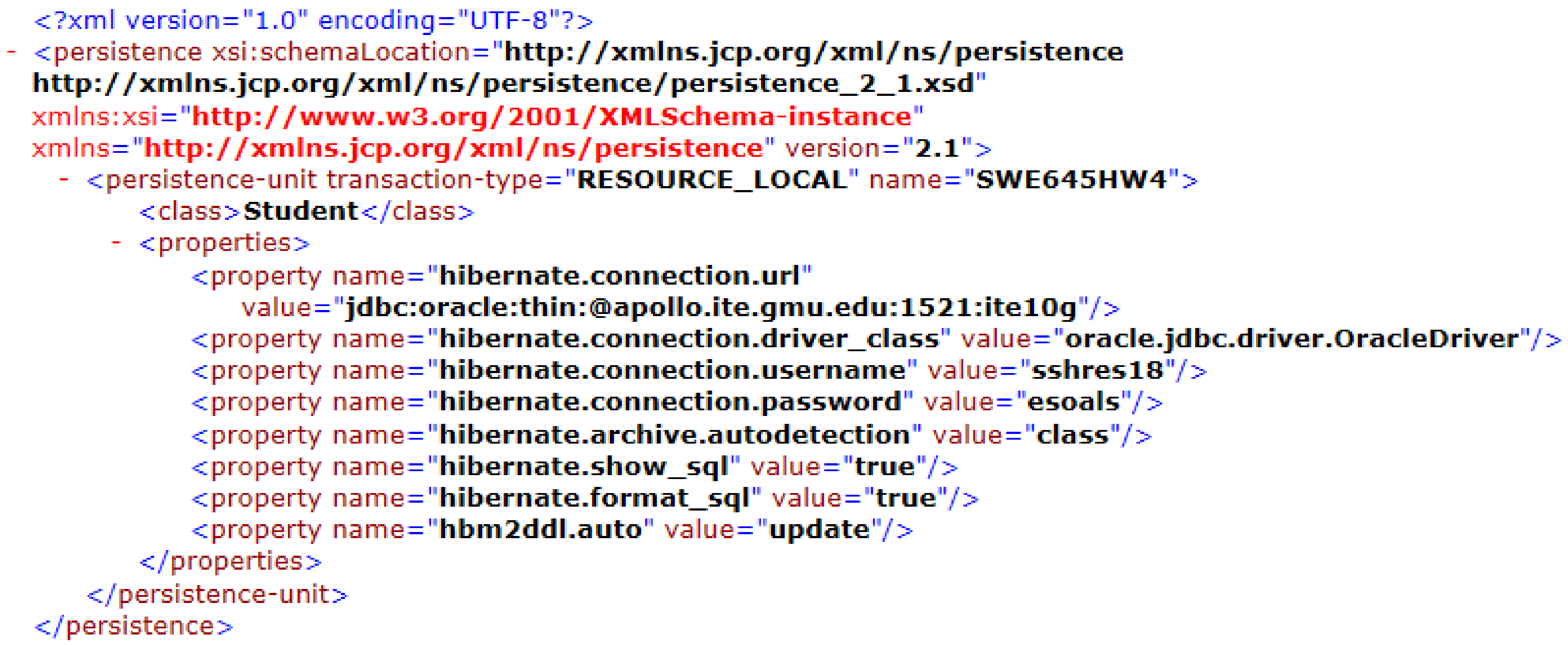
- **Below is an example of persistence.xml, which resides in the src/META-INF folder. Here swe645 is the MySQL database name created via Amazon RDS.**



```
  <?xml version="1.0" encoding="UTF-8"?>
- <persistence xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">
  - <persistence-unit transaction-type="RESOURCE_LOCAL" name="assign3">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>assignment.Student</class>
      - <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver"/>
            <property name="hibernate.connection.url" value="jdbc:mysql://127.0.0.1:3306/swe645"/>
            <property name="hibernate.connection.username" value="nsimon2"/>
            <property name="hibernate.connection.password" value="nopassword"/>
        </properties>
    </persistence-unit>
  </persistence>
```

Create Hibernate JPA Configuration file: persistence.xml

- Below is an example of persistence.xml using GMU’s Oracle database.



```
 <?xml version="1.0" encoding="UTF-8"?>
- <persistence xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">
  - <persistence-unit transaction-type="RESOURCE_LOCAL" name="SWE645HW4">
        <class>Student</class>
      - <properties>
            <property name="hibernate.connection.url"
                value="jdbc:oracle:thin:@apollo.ite.gmu.edu:1521:ite10g"/>
            <property name="hibernate.connection.driver_class" value="oracle.jdbc.driver.OracleDriver"/>
            <property name="hibernate.connection.username" value="sshres18"/>
            <property name="hibernate.connection.password" value="esoals"/>
            <property name="hibernate.archive.autodetection" value="class"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hbm2ddl.auto" value="update"/>
        </properties>
    </persistence-unit>
 </persistence>
```

JPA Providers

- **EclipseLink -- reference implementation**
  - Download the "EclipseLink Installer Zip" implementation from http://www.eclipse.org/eclipselink/downloads/
  - The download contains several jars. You may need the following jars:
    - eclipselink.jar
    - javax.persistence_*.jar

Summary

- **Java Persistence API**
  - **JPA Entities --** How to Use Annotations to configure POJOs into JPA entities
- **Entity Manager**
  - Entity Manager Methods to interact with persistence storage
  - Persistence Unit/Persistence Context
- **Examples**

JPA Entity Relationships

Topics in this section

- **Intro to JPA Entity Relationships**
  - Cardinality
  - Unidirectional vs. bidirectional
- **Seven Entity Relationships**
- **Ordered List-based Relationships**
- **Cascading**
- **Lazy vs. Eager Loading**

Entity Relationships

- **Entity Relationships are used to model real-world business concepts**
- **Entity beans must be capable of forming relationships**
- **For instance:**
  - An employee has an address or
  - An employee has multiple phone numbers

Entity Relationships

- **Seven types of relationships can exist between entity beans**
- **There are 4 types of cardinality:**
  - one-to-one,
  - one-to-many,
  - many-to-one, and
  - many-to-many
- **Each relationship can be either unidirectional or bidirectional**

The Seven Relationship Types

- One-to-one unidirectional
- One-to-one bidirectional
- One-to-many unidirectional
- One-to-many bidirectional
- Many-to-one unidirectional
- Many-to-many unidirectional
- Many-to-many bidirectional

JPA Support for Entity Relationship

- **The relationships between different entities are defined using @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany annotations**
  - A unidirectional relationship requires the **owning side** to **specify** the annotation.
  - A bidirectional relationship also requires the non-owning side to refer to its owning side by use of the **mappedBy** element of the OneToOne, OneToMany, or ManyToMany annotation.

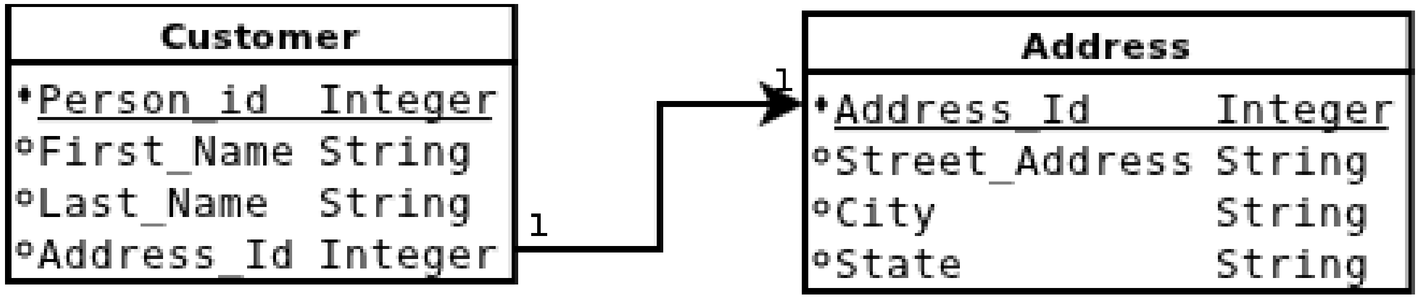
One-to-One Unidirectional Relationship

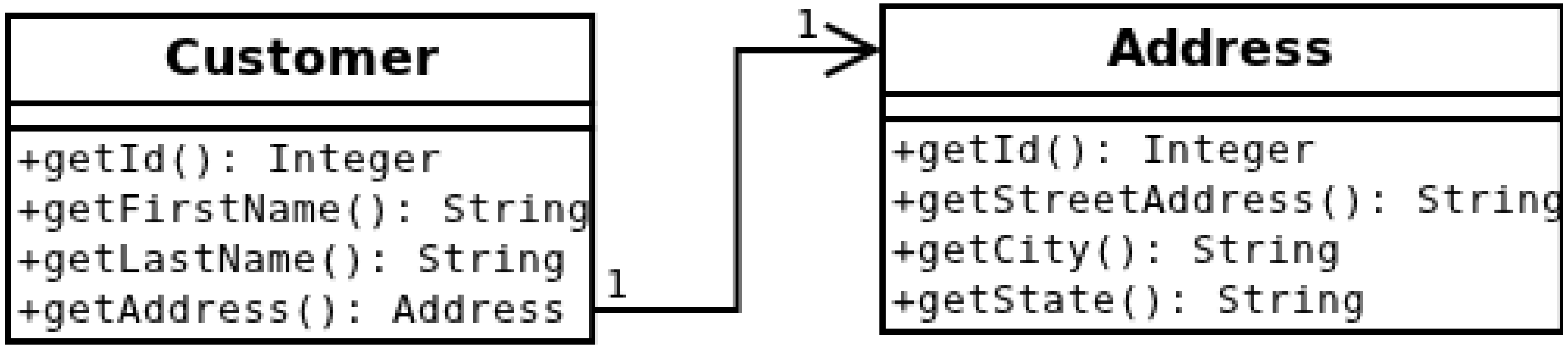
- **One-to-One relationship refers to a relationship where one entity is related to exactly one other entity**
  - For example, relationship between Customer and Address entities
  - **Each customer has exactly one address**, and each address belongs to one customer
- **Unidirectional means only one entity has access method to get the other object**

One-to-One Unidirectional Relationship

- **In the next example, Address_ID is a join column in both tables**
- **getAddress() in Customer table provides access to Address entity**

One-to-One Unidirectional Relationship





Bean Class

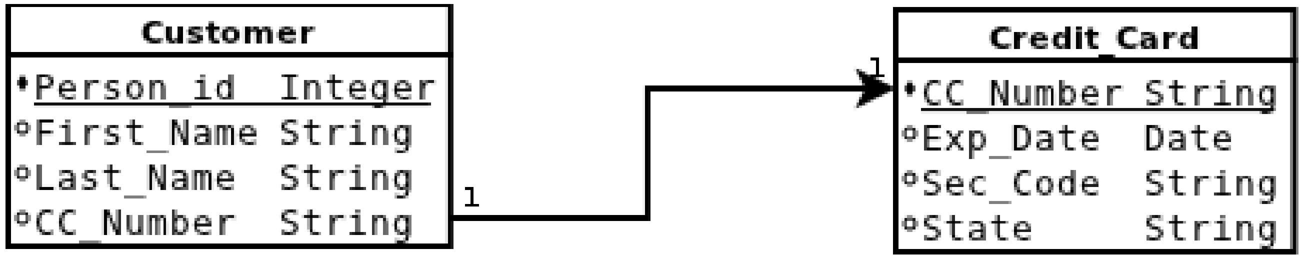
- A **unidirectional relationship** requires the **owning side** to **specify** the **annotation**

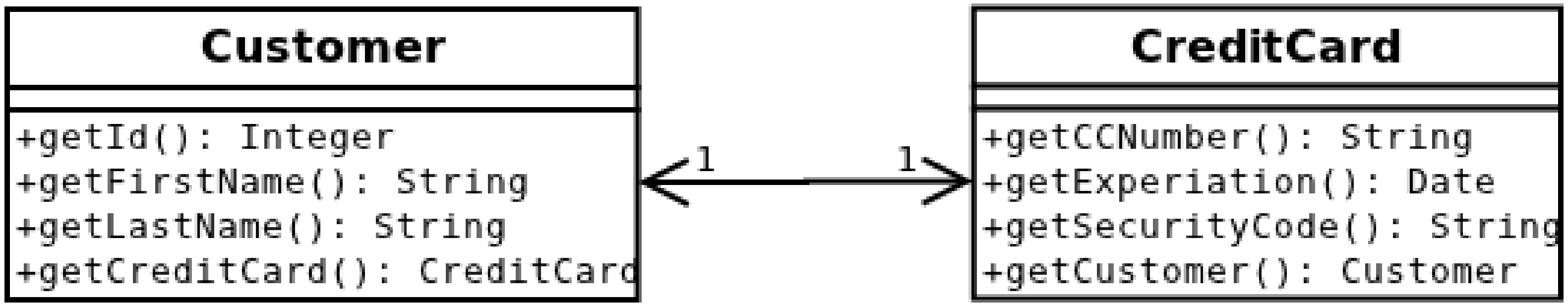
```
@Entity
public class Customer implements Serializable{
  private Address address;

   @OneToOne
     @JoinColumn(name="Address_ID")
   public Address getAddress() {
        return address;
   }
   public void setAddress(Address address) {
      this.address = address;
   }
```

One-to-One Bi-directional Relationship

- Both entities have methods to access the other object
- The following example, creates a join column CC_Number in both tables

One-to-One Bi-directional Relationship





One-to-One Bi-directional Relationship

- **In bi-directional relation, need to specify that the mapping is done by one entity**
- **A bidirectional relationship requires the non-owning side to refer to its owning side by use of the mappedBy element of the OneToOne, OneToMany, or ManyToMany annotation.**

Bean Classes

```
@Entity
public class Customer implements Serializable{
    private CreditCard creditCard;

    @OneToOne
    @JoinColumn(name="CC_Number")
        public CreditCard getCreditCard() {return creditCard;}
    public void setCreditCard(CreditCard cc)
        { creditCard = cc; }
```

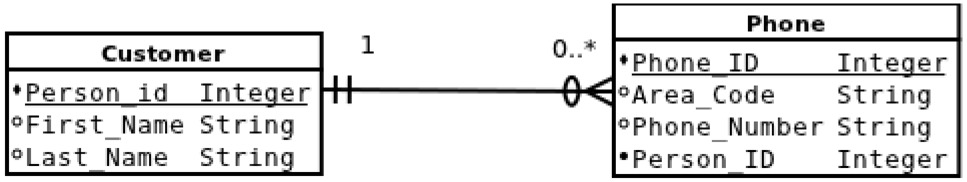
```
@Entity
public class CreditCard implements Serializable {
       private Customer cust;

    @OneToOne(mappedBy="creditCard")
    Public Customer getCustomer() { returncust; }
```

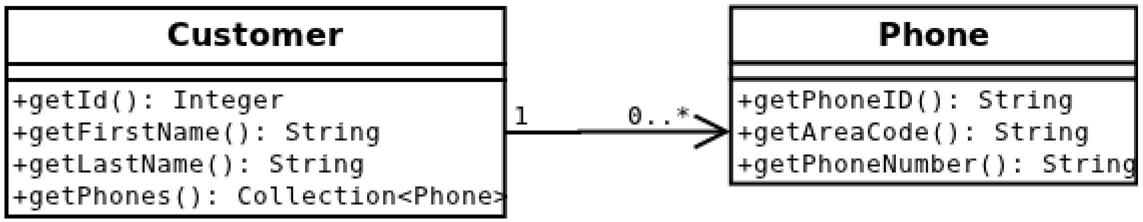
- In bi-directional relation, need to specify, in the non owning side, that the mapping is done by the owning side entity,.

One-to-Many Unidirectional Relationship

- **One-to-Many relationship refers to a relationship where one entity is related to more than one other entity**
  - For example, relationship between Customer and Phone entities
  - **A customer has many phones**
- **Relationship is unidirectional means only customer entity has access method to get the phone objects**

One-to-Many Unidirectional Relationship





- A Customer has many Phones

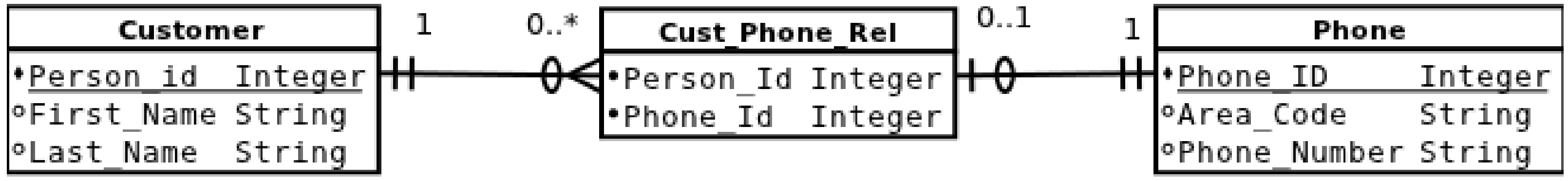
Bean Class

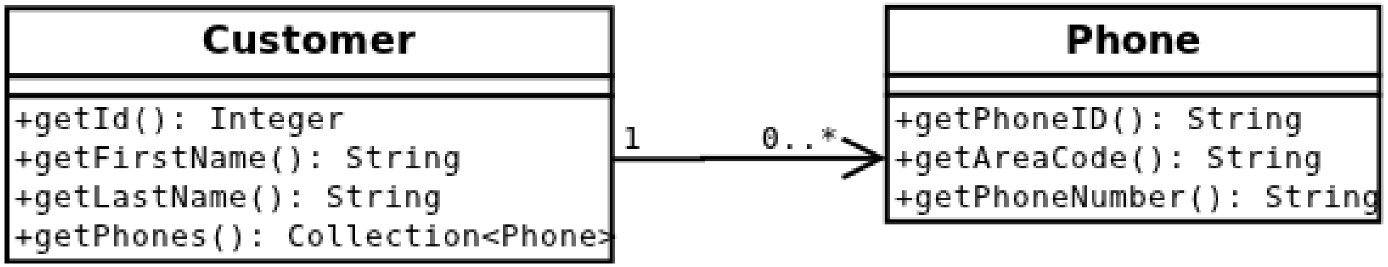
```
@Entity
public class Customer implements Serializable{
      private Collection<Phone> phoneNumbers = ... ...

    @OneToMany
      @JoinColumn(name="Person_ID")
      public Collection<Phone> getPhones() {
       return phones;
      }

      public void setPhone(Collection<Phone> phones) {
       this.phones = phones;
      }
```

One-to-Many Relationship With Join Table

- **The Hibernate, by default, creates a separate mapping table (or join table) for one to many relationships**
  - Default join table name is concatenation of two entity names
  - Default join column names are tablename+columnname for both tables
- **You can override default names for join columns and join tables created by hibernate**

One-to-Many Relationship With Join Table





- The Hibernate, by default, creates a separate mapping table (or join table) for one to many relationships.

Bean Class

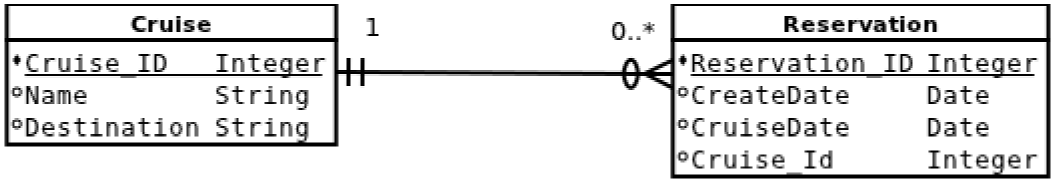
```
@Entity
public class Customer implements Serializable{
      private Collection<Phone> phoneNumbers = ... ...

    @OneToMany
       @JoinTable(name="Cust_Phone_Rel",
        joinColumns={@JoinColumn(name="Person_ID")},
        inverseJoinColumns={@JoinColumn(name="Phone_ID")})
    public Collection<Phone> getPhones() {
        return phones;
       }

      public void setPhone(Collection<Phone> phones) {
        this.phones = phones;
       }
```

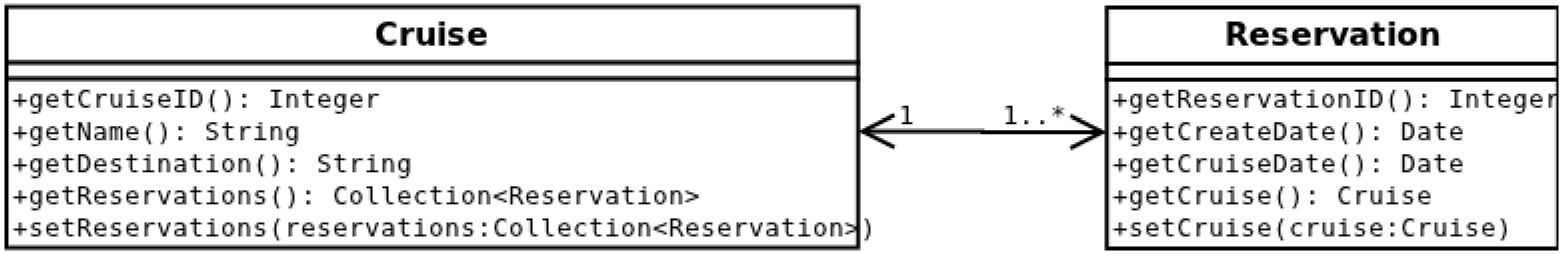
- **You can override default names for join columns and join tables created by hibernate**
- **Default join table name is concatenation of two entity names**
- **Default join column names are tablename+columnname for both tables**

One-to-Many Bi-directional Relationship without a separate Join table

- **Another way to represent one-to-many relationship is to have the many side of the relationship (e.g., Reservation) a reference to one-side of the relationship (e.g., Cruise).**
  - That is Reservation has CruiseID as column instead of creating a separate mapping table.
- **To accomplish this, use mappedBy of OneToMany relationship**
  - to say that Cruise instance of Reservation class has to do the mapping, as shown on the next page.
  - This creates only two tables Cruise and Reservation.

One-to-Many Bi-directional Relationship





Cruise
+getCruiseID(): Integer
+getName(): String
+getDestination(): String
+getReservations(): Collection<Reservation>
+setReservations(reservations:Collection<Reservation>)
1
1..*
Reservation
+getReservationID(): Integer
+getCreateDate(): Date
+getCruiseDate(): Date
+getCruise(): Cruise
+setCruise(cruise:Cruise)

- **Another way to represent one-to-many relationship is to have the many side of the relationship (e.g., Reservation) a reference to one-side of the relationship (e.g., Course). That is Reservation has CruiseID as column instead of creating a separate mapping table.**
- **To accomplish this, use mappedBy of OneToMany relationship to say that Cruise instance of Reservation class has to do the mapping, as shown on the next page. This creates only two tables Cruise and Reservation.**

Bean Classes

```
@Entity
public class Reservation implements Serializable{
      private Cruise cruise;

   @ManyToOne
   @JoinColumn(name="Cruise_Id")
   public Cruise getCruise()
   public void setCruise(...)
```

```
@Entity
public class Cruise implements Serializable{
      private Collection<Reservation> reservations;

   @OneToMany(mappedBy="cruise")
   public Collection<Reservation> getReservations()
{.....}
```

mappedBy used to tell that mapping is done by Cruise instance in Reservation and this creates a Cruise_Id column in Reservation table instead of creating a separate join table.

Ordered List-Based Relationships

```
@Entity
public class CruiseShip implements Serializable{
  private Collection<Cabin> cabins;

      @OneToMany
      @OrderBy("roomClass asc, roomNumber desc")
   @JoinColumn(name="Ship_Id")
      public Collection<Cabin> getCabins() {
       return this.cabins;
      }
```

Cascading

- When you perform an EntityManager operation on an entity bean instance, you can automatically have the same operation performed on any relationship properties the entity may have. This is called cascading.
- Cascading can be applied to a variety of EntityManager operations, including
  - persist(), merge(), remove(), and refresh().
- This feature is enabled by setting the javax.persistence.CascadeType of the relationship annotation's *cascade* attribute.

Cascading

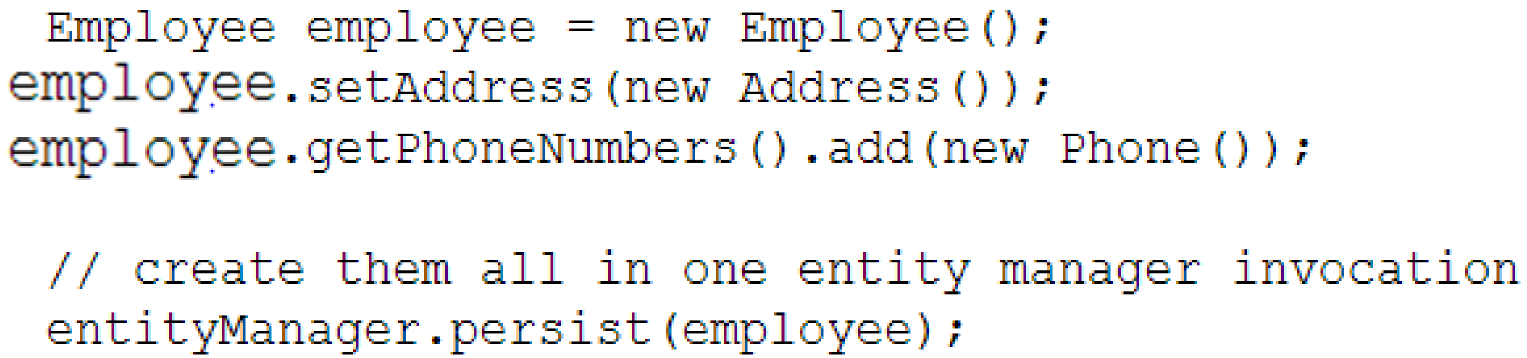
- The CascadeType is defined as a Java enumeration

```
public enum CascadeType
{
    ALL, PERSIST,
    MERGE, REMOVE,
    REFRESH
}
```

- The ALL value represents all of the cascade operations.
- The remaining values represent individual cascade operations.

Cascading

- **For example, persisting a new Employee entity with a new address and phone number**
- **All you have to do is wire the object, and the entity manager can automatically create the employee and its related entities, all in one persist() method call**



```
  Employee employee = new Employee();
employee.setAddress(new Address());
employee.getPhoneNumbers().add(new Phone());

  // create them all in one entity manager invocation
  entityManager.persist(employee);
```

Cascading

Used in relationship annotation

```
    @OneToMany(cascade={CascadeType.ALL})

public enum CascadeType {
        ALL, PERSIST, MERGE, REMOVE, REFRESH
}
```

Persist

```
Customer cust = new Customer();
Address address = new Address();
cust.setAddress(address);
entityManager.persist(cust);
```

Customer and Address are persisted.

Cascading

Merge

```
Cust.setName("William");
Cust.getAddress().setCity("Boston");
entityManager.merge(cust);
Phone phone = new Phone();
Phone.setNumber("123456789");
cust.getPhoneNumbers().add(phone);
entityManager.merge(cust);
```

Customer and Address are merged

Cascading

Remove

```
Customer cust =
entityManger.find(Cstomer.class, 1);
entityManager.remove(cust);
```

Refresh

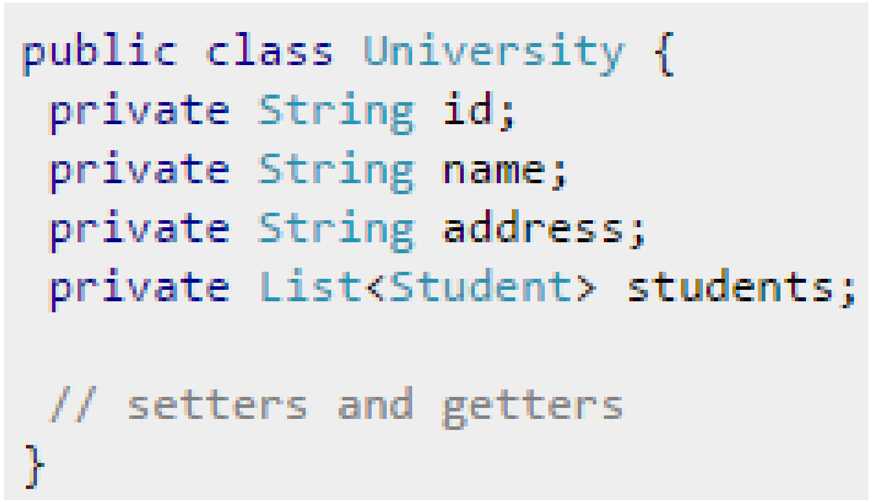
```
Customer cust =
entityManger.find(Cstomer.class, 1);
entityManager.refresh(cust);
```

Lazy vs. Eager Loading

- The FetchType.EAGER annotation may be specified on an entity to eagerly load the data from the database.
- The FetchType.LAZY annotation may be specified as a hint that the data should be fetched lazily when it is first accessed.

Lazy vs. Eager Loading

- **Consider a relationship between University and Student entities**
  - The University entity might have some basic properties such as id, name, address, etc. as well as a property called students:



```
public class University {
 private String id;
 private String name;
 private String address;
 private List<Student> students;

 // setters and getters
}
```

Lazy vs. Eager Loading

- **When a University entity is loaded from the database, JPA loads its id, name, and address fields for you.**
- **There are two options for students:**
  - to load it together with the rest of the fields (i.e. **eagerly**) or
  - to load it on-demand (i.e. **lazily**) when you call the university's getStudents() method.

Lazy vs. Eager Loading

- When a university has many students it is not efficient to load all of its students with it when they are not needed.
- So in such cases, you can declare that you want students to be loaded when they are actually needed. This is called lazy loading.

Lazy vs. Eager Loading

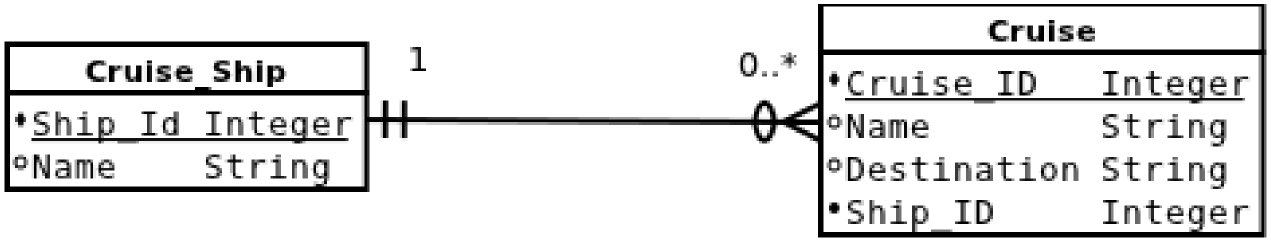
Relationship Type	Default Fetch Behavior	Number of Entity Retrieved
One-To-One	Eager	Single Entity
One-To-Many	Lazy	Collection of Entities
Many-To-One	Eager	Single Entity
Many-To-Many	Lazy	Collection of Entities

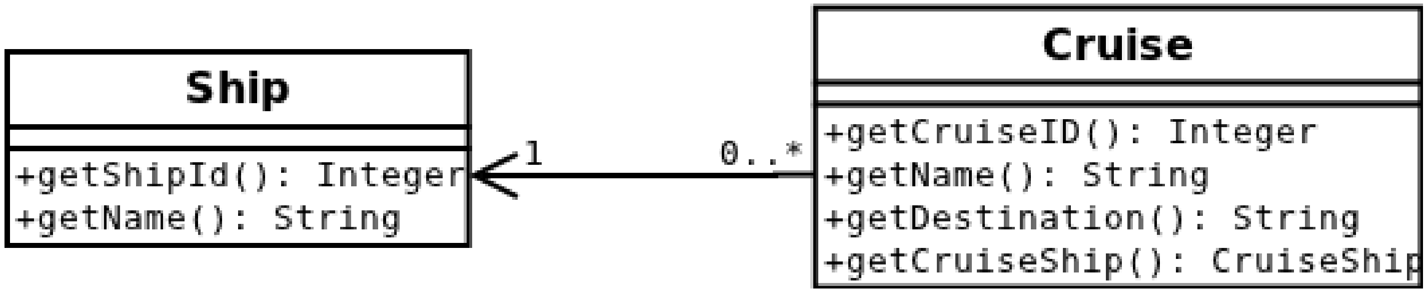
Lazy vs. Eager Loading

```
@Entity
public class Customer implements Serializable{

    private Collection<Phone> phones;

    @OneToMany(cascade={CascadeType.ALL},
               fetch=FetchType.EAGER)
    @JoinColumn(name="customer_id")
    public Collection<Phone> getPhones() {
          return phones;
    }
}
```

Optional Topics

Many-to-One Unidirectional





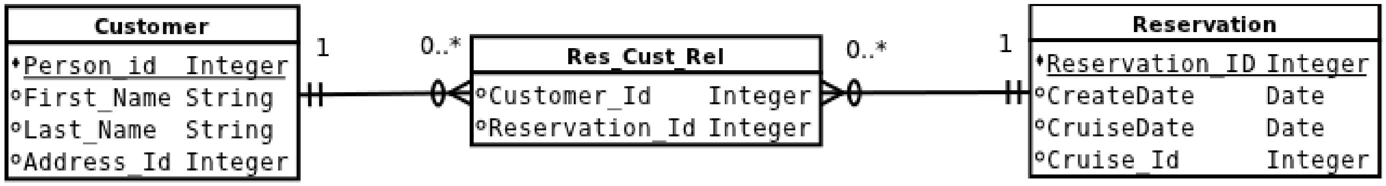
Bean Class

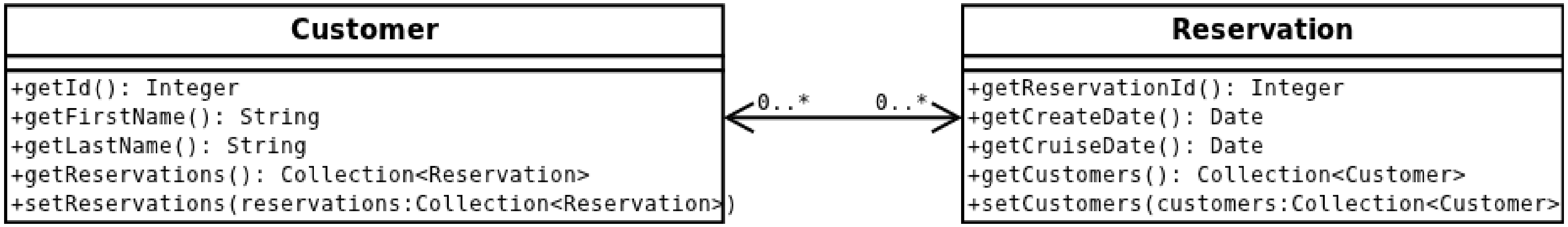
```
@Entity
public class Cruise implements Serializable{
      private CruiseShip ship;

      @ManyToOne
      @JoinColumn(name="Ship_Id")
      public CruiseShip getCruiseShip() {
       return ship;
      }

      public void setCruiseShip(CruiseShip ship) {
       this.ship = ship;
      }
```

Many-to-Many Bidirectional Relationship

- **Bidrectional means both entities have methods to access the other objects(s)**
- **Many-to-Many relationship has to have a separate mapping (i.e., join) table**
- **For example:**
  - Res_Cust_Rel table
  - where names of table and columns can be changed using joinColumns and inverseJoinColumns property as shown on the next page.

Many-to-Many Bidirectional Relationship





Customer
+getId(): Integer
+getFirstName(): String
+getLastName(): String
+getReservations(): Collection<Reservation>
+setReservations(reservations:Collection<Reservation>)
0..*
0..*
Reservation
+getReservationId(): Integer
+getCreateDate(): Date
+getCruiseDate(): Date
+getCustomers(): Collection<Customer>
+setCustomers(customers:Collection<Customer>

Many-to-Many Bidirectional Relationship

- **For @ManyToMany annotations in both tables, hibernate, by default, will create two mapping tables**
  - Mapping Reservation to Customer and
  - Mapping Customer to Reservation
- **This is redundant and can be overridden.**
- **That is, create only one mapping table (i.e., join table) by using mappedBy property in one method as shown on the next page.**

Bean Classes

```
@Entity
public class Reservation implements Serializable{
    @ManyToMany
    @JoinTable(name="Res_Cust_Rel",
        joinColumns{@JoinColumn(name="Reservation_ID"}),
        inverseJoinColumns={@JoinColumn(name="Customer_ID")})
    public Collection<Customer> getCustomers()
    public void setCustomer(...)
```

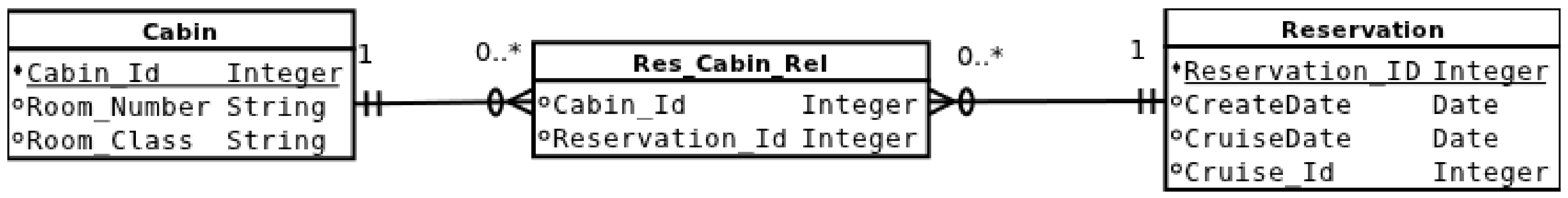
```
@Entity
public class Customer implements Serializable{
    @ManyToMany(mappedBy="customers")
    public Collection<Reservation> get Reservations()
    public void setReservations(...)
```

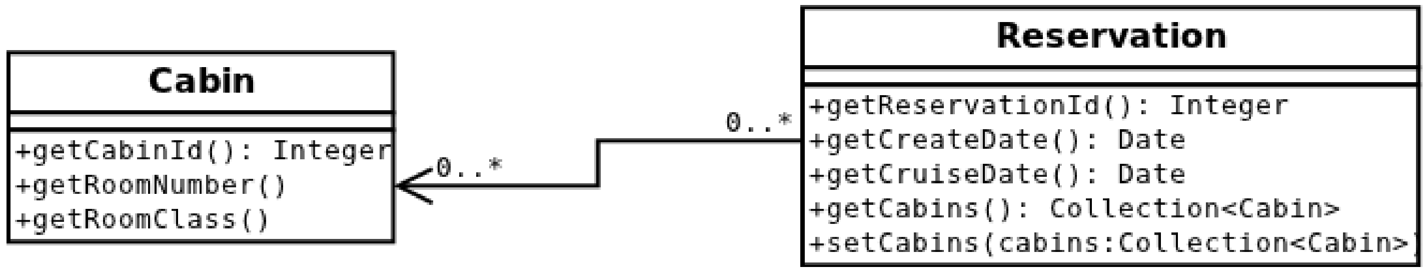
Here we say that mapping is done by the other entitiy which creates a join table, so don’t need to create another mapping.

Many-to-Many Unidirectional Relationship

- **Only one entity has access method to get the other objects**
  - Reservation can get all its related Cabins, but
  - Cabin does not know who made Reservation

Many-to-Many Unidirectional Relationship





No get method in Cabin to access Reservation

Bean Class

```
@Entity
public class Reservation implements Serializable{
   private Collection<Cabin> cabins;

      @ManyToMany
      @JoinTable(name="Res_Cabin_Rel",
       joinColumns={@JoinColumn(name="Reservation_Id)},
       inverseJoinColumns={@JoinColumn(name="Cabin_Id")})
      public Collection<Cabin> getCabins()

      public void setCabins(...)
```

JDBC Data Source

A basic DataSource implementation

Standard connection object that is **not pooled or used** in a distributed transaction.

A DataSource class that supports connection pooling

Connection objects that participate in connection pooling and will be recycled.

A DataSource class that supports distributed transaction

Connections objects that can be used in a distributed transaction.

Obtain Persistence Context inside EJB

A reference to the Persistence Context can be injected into the EntityManager.

```
@Stateless
public class StatelessEJB implements
StatelessRemote {

    @PersistenceContext(unitName="swe645Unit")
    private EntityManager entityManger;
```

Tips for Designing Entity Beans

Read an application description and underline the Nouns. These nouns can be representative of the Entities (tables).

Determine the relationship between the Entities.

- How do they relate?
- Are they one to many? Many to many? (Discussed later.)

System Description (example)

A new university is opening. There are 3 locations, VA, MD, and DC. The people who attend this university are Students and Employees. Employees and Students have contact information (name, phone number, address, email, and date of birth). Employees have a title, department, hire date, employment status, and a termination date. Students have a Major, start date, GPA, graduation date, and status. Employees can be Students. Students can register for multiple courses. Each course can have many students. Courses are assigned to a specific room, in a building, for a location. Each location has multiple buildings. Each building has multiple rooms.