B.Sc. In Software Development. Year 4.
Semester I. Enterprise Development.
Java Persistence Architecture (JPA).

LIMERICK INSTITUTE OF TECHNOLOGY
SCHOOL OF SCIENCE, ENGINEERING & I.T.
Department of Information Technology

JPA – An Introduction

- A relatively new way of working with databases in Java that has many advantages over the older JDBC method.
  - Can automatically create database tables based on the relationship between business objects.
  - Can convert between objects and rows in a relational database.
  - Can perform *joins* to satisfy relationships between objects.

- JPA makes it easier to work with OO data and relationships between objects.
  - It makes it easier to convert between the business objects and the relational database of an application.
  - Known as O/R mapping or ORM.

- JPA can be used for any type of Java application (desktop, mobile or web).
  - Runs on top of JDBC.

JPA – An Introduction

- There are several implementations of JPA and all of them follow the same specification.
  - All skills are transferrable

- Full Java EE servers typically provide their own implementation of JPA.
  - Glassfish uses TopLink. WildFly uses Hibernate. When using Tomcat you can choose the JPA.

JPA – An Introduction

- When working with JPA, (business) objects are known as entities and are managed by an entity manager.

- In a full Java EE server such as Glassfish, the server provides a built-in entity manager that includes advanced features such as automatic transaction rollback.
- We are going to look at using entity managers outside of a full EE server – meaning you can used them in a web or desktop application.

- To turn a normal business class into an entity, you need to code annotations in the class.
- These annotations specify how the class should be stored in a database and they specify how one class relates with another.

Configuring Netbeans with JPA

1. Add a JDBC driver to your project.
2. Add the JPA library you want to use to your project.
3. Add a persistence unit to your project.

A persistence unit (an XML file) tells JPA how to connect to your database.

Coding JPA Entities

- A JPA entity is essentially a business class with annotations added to it.
  - JPA uses these annotations to determine how to use the data in the class within the application.
- The following code examples assume a database exists with tables called User, Invoice and LineItem within them.
- Here, one Invoice can have **many** line items.
- However, each line item can only belong to one invoice.
  - Therefore there is a **one to many** relationship between these two tables.

Coding JPA Entities

User table

#	Name	Type	Null	Default	Extra	Action
1	**UserID**	int(11)	No	*None*	AUTO_INCREMENT	Change Drop Primary
2	**FirstName**	varchar(50)	Yes	*NULL*		Change Drop Primary
3	**LastName**	varchar(50)	Yes	*NULL*		Change Drop Primary
4	**EmailAddress**	varchar(50)	Yes	*NULL*		Change Drop Primary

Coding JPA Entities

```
10  @Entity
    public class User implements Serializable {
12
13      @Id
14      @GeneratedValue(strategy = GenerationType.AUTO)
15      private Long UserID;
16      private String FirstName;
17      private String LastName;
18      private String EmailAddress;
19
20      /**
21       * @return the UserID
22       */
23      public Long getUserID() {
24          return UserID;
25      }
26
27      /**
28       * @param UserID the UserID to set
29       */
30      public void setUserID(Long UserID) {
31          this.UserID = UserID;
32      }
```

Remaining getters/setters are not included

Getter Annotations

- In my code listing I placed the @Id annotation directly above the UserID field.
  - This is *field annotation*.

- Another option is to place the annotation above the **getUserId** method instead.
  - This is *getter annotation*.

```java
10  @Entity
    public class User implements Serializable {
12
13      private Long UserID;
14      private String FirstName;
15      private String LastName;
16      private String EmailAddress;
17
18      /**
19       * @return the UserID
20       */
21      @Id
22      @GeneratedValue(strategy = GenerationType.AUTO)
23      public Long getUserID() {
24          return UserID;
25      }
26
27      /**
28       * @param UserID the UserID to set
29       */
30      public void setUserID(Long UserID) {
31          this.UserID = UserID;
32      }
```

Getter Annotations

- When you use getter annotations, JPA uses the get and set methods of your class to get and set the fields.

- When you use field annotations, JPA uses reflection to get and set the field.
  - Reflection allows JPA to access the fields directly, even if they are private.
  - That means that even if you have get and set methods in your class, JPA doesn't call them when you use field annotation.

Coding Relationships Between Entities

- When you define an entity, it can have a relationship to other entities. For example, an invoice may contain one or more line item entities that are dependent on the invoice entity.
- When you code a query in JPA, it automatically performs any joins necessary to satisfy the relationships between entities.
- Also when you have an entity, JPA automatically saves any dependent entities.
- To make these automatic relationships work, you code annotations in your entity classes that define the relationships.

Coding Relationships Between Entities

Invoice table

Name	Type	Null	Default	Extra	Action
InvoiceID	int(11)	No	*None*	AUTO_INCREMENT	Change Drop Primary
UserID	int(11)	No	*None*		Change Drop Primary
InvoiceDate	datetime	No	0000-00-00 00:00:00		Change Drop Primary
TotalAmount	float	No	0		Change Drop Primary
IsProcessed	enum('y', 'n')	Yes	*NULL*		Change Drop Primary

LineItem table

Name	Type	Null	Default	Extra	Action
LineItemID	int(11)	No	*None*	AUTO_INCREMENT	Change Drop Primary
InvoiceID	int(11)	No	0		Change Drop Primary
ProductID	int(11)	No	0		Change Drop Primary
Quantity	int(11)	No	0		Change Drop Primary

Coding Relationships Between Entities

```
23  public class Invoice {
24
25      @ManyToOne
26      private User user;
27
28      @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
29      private List<LineItem> items;
30
31      @Temporal(javax.persistence.TemporalType.DATE)
32      private Date IvoiceDate;
33
34      @Id
35      @GeneratedValue(strategy = GenerationType.AUTO)
36      private Long invoiceNumber;
37
38      private boolean isProcessed;
39
40      private double TotalAmount;
```

getter/setters are not included

Coding Relationships Between Entities

- One customer can have many invoices.
- However, each invoice can only belong to one customer.
- To show this relationship in JPA, you code a **@ManyToOne** annotation directly before the field for the user/getUser method.

- One invoice can have many line items.
- However, each line item can only belong to one invoice.
- To show this relationship, you code a **@OneToMany** annotation above the line items field/getLineItems method.
- The **@OneToMany** annotation typically includes some additional elements.

- The fetch element determines when JPA loads the line items for the invoice.

Coding Relationships Between Entities

- By default JPA uses lazy loading.
- This means that when JPA first gets the invoice from the DB its line items are empty.
- Then, JPA fetches these items the first time you attempt to access the line items in this invoice.

- If you don't want to use lazy loading, you can set the fetch element to a value of **FetchType.EAGER.**

- If you want to use lazy loading you can set the fetch attribute to a value of **FetchType.LAZY.**
  - This is however, only a request to JPA.

Coding Relationships Between Entities

- Two optional attributes of @OneToMany

1. **fetch**

   - **FetchType.EAGER** specifies that all of the line items for the invoice should be loaded when the invoice is loaded from the database.

   - **FetchType.LAZY** specifies that all of the line items for the invoice should be loaded only when the application needs them.

2. **Cascade**

   - **CascadeType.All** specifies that all operations that change the invoice should also update all of the line items.

Coding Relationships Between Entities

2. Cascade

- **CascadeType.PERSIST** specifies that any time a new Invoice is inserted into the database that a LineItem should also be added.

- **CascadeType.MERGE** specifies that any time a new Invoice is updated in the database that any changes to its LineItem should also be made.

- **CascadeType.REMOVE** specifies that any time a new Invoice is removed from the database all of its LineItems should also be removed.

Coding Relationships Between Entities

- As can be seen from the previous slide, other values can also be used with the cascade element.

- If you want to combine values, you can use the standard way of combining an elements values.

  - For example, if you want to combine the PERSIST and the MERGE values of the cascade element you can code them like this:

cascade={CascadeType.PERSIST, CascadeType.MERGE}

Coding Relationships Between Entities

- In the java.util package, the Date and Calendar types can map to multiple possible SQL data types.

- As a result, when you use either a Date or Calendar type in an entity, you need to use the **@Temporal** annotation to specify its SQL data type.

- Within the @Temporal annotation you can code one of the three values.
  - **TemporalType.DATE** specifies that JPA should only store the date, not the time.
  - **TemporalType.TIME** specifies that JPA should only store the time, not the date.
  - **TemporalType.TIMESTAMP** specifies that JPA should store both the date and time.

References

Murach, J., (2014) *Murachs JavaServlets JSP*, 3rd edn. Mike Murach and Associates, Inc.

Keith, M., Schincariol, M. (2013) *Pro JPA 2: Second Edition (Expert's Voice in Java)*, 2nd edn. Apress.

http://www.objectdb.com/api/java/jpa

https://www.tutorialspoint.com/jpa/

http://wiki.eclipse.org/EclipseLink/Examples/JPA/JSF_Tutorial