Getting Started with Persistence using EclipseLink JPA
(includes MySQL, Maven and EclipseLink)

Set up your MySQL database
Create your database and tables (possibly put data in it right away to test)

```
create database shopping;
use shopping;
create table items ( ID int NOT NULL AUTO_INCREMENT,
                     STORE varchar(20) NOT NULL,
                     ITEM varchar(35) NOT NULL,
                     PRIMARY KEY (ID));
```

What is JPA?
Mapping Java objects to database tables and vice versa is called Object-relational mapping (ORM). The Java Persistence API (JPA) is one possible approach to ORM. Via JPA the developer can map, store, update and retrieve data from relational databases to Java objects and vice versa.

JPA metadata is typically defined via annotations in the Java class. Alternatively, the metadata can be defined via XML or a combination of both. A XML configuration overwrites the annotations. We will use annotations in our project, as using XML is generally becoming deprecated in preference for annotations.

Set up your Eclipse Project
Start a new Java Project in Eclipse and name it **ConsoleShoppingList**
Configure → Convert to a Maven Project You'll be asked to provide a groupId, artifactId, and version.

>> Naming conventions on groupId, artifactId and version

- groupId will identify your project uniquely across all projects. This will typically match your project name, except all in lowercase. eg. consoleshoppinglist (for this project)

- artifactId is the name of the jar without version. This doesn't mean a lot to us right now, but you'll notice it when you search for dependencies. If you created it then you can choose whatever name you want with lowercase letters and no strange symbols. If it's a third party jar you have to take the name of the jar as it's distributed. eg. consoleshoppinglist (for this project)

- version - if you distribute it then you can choose any typical version with numbers and dots (1.0, 1.1, 1.0.1, ...). Don't use dates as they are usually associated with SNAPSHOT (nightly) builds. We will generally use the default. If it's a third party artifact, you have to use their version number (whatever it is, and as strange as it looks. eg. 2.0, 2.0.1, 1.3.1) eg. 0.0.1-SNAPSHOT (for this project)

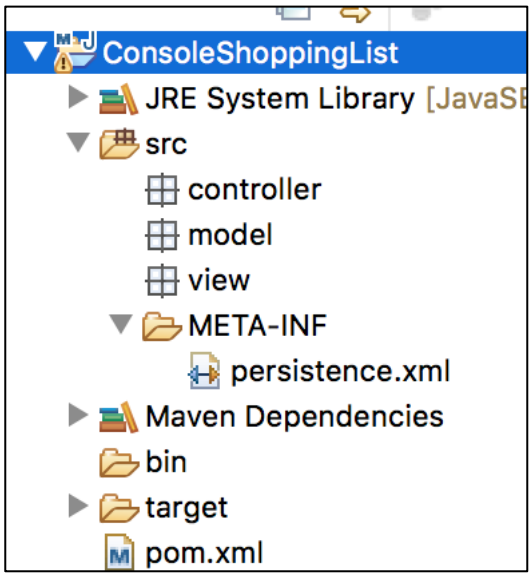
Add these dependencies to pom.xml. Remember to search for them. Don't mistype them.

```
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
	<version>8.0.13</version>
</dependency>
<dependency>
	<groupId>org.eclipse.persistence</groupId>
	<artifactId>eclipselink</artifactId>
	<version>2.5.0</version>
</dependency>
<dependency>
	<groupId>org.eclipse.persistence</groupId>
	<artifactId>javax.persistence</artifactId>
	<version>2.1.1</version>
</dependency>
```

These Maven dependencies will add in the support files we need to use the database connection and the .jar files to use Eclipselink Persistence. There are many providers for persistence. We'll start with Eclipselink because it has very low overhead. Save pom.xml before continuing and allow the jar files to download to your computer & apply to the project.

Now, let's add JPA support to our web project.
Configure → Convert to JPA Project – if necessary, add checkmark next to JPA
Next → Source Folders on Build Path (should have src in the box) → Press Next
Platform – EclipseLink 2.5x
JPA Implementation Type – Disable Library Configuration
✓Annotated classes must be listed in persistence.xml
Finish

Now in your src folder, you'll find a new META-INF folder with a persistence.xml file in it. If you don't see it immediately, right click on the project & refresh.



Create your project/package structure

Go to Java Resources (src) folder in your project, right click and select New Package to create one of the package structure: (my example uses the first option). We will be using these to separate out the use of our different files.

- model/view/controller
- entities/beans/program

Create your POJO in your model or entities package

- default, no arg constructor
- private attributes/instance variables
- getters & setters
- any helper methods or additional constructors
- Our POJO details
    - public class ListItem
        - Attributes/instance variables
            - private int id;
            - private String store;
            - private String item;
        - Default, no arg constructor

          public ListItem{

          }
        - Getters and setters (Source → Generate Getters and Setters)
        - Helper methods – create one more constructor & a print method

          - public ListItem(String store, String item){
                this.store = store;
                this.item = item;
            }
          - public String returnItemDetails( ) {
                return store + ": " + item;
            }

Annotate your POJO and add in the import statements

An annotation is a simple, expressive means of decorating Java source code with metadata that is compiled into the corresponding Java class files for interpretation at run time by a JPA persistence provider to manage persistent behavior. This POJO is called an Entity class when used with persistence.

An entity class must follow these requirements.

- The class must be annotated with the javax.persistence.Entity annotation.
- 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. ← We will discuss and work with this later when we start working with beans.
- 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 and can be accessed directly only by the entity class's methods. All access to the entity's state (instance variables) are through accessor methods (getters & setters).

In order to use persistence, our Java classes must have several annotations to be recognized.

@Entity: A class which should be persisted in a database it must be annotated with javax.persistence.Entity. Such a class is called an Entity. JPA uses a database table for every entity. Persisted instances of the class will be represented as one row in the table.

All entity classes must define a primary key, must have a non-arg constructor and not allowed to be final. Keys can be a single field or a combination of fields.

By default, the table name corresponds to the class name. You can change this with the addition to the annotation @Table(name="NEWTABLENAME").

JPA allows to auto-generate the primary key in the database via the @GeneratedValue annotation.

- @Entity
- @Table(name = "items")
- @Id
- @GeneratedValue(strategy = GenerationType.IDENTITY)

IDENTITY – The table will automatically provide the primary key
AUTO – You will provide the primary key

See the final annotated ListItem POJO for the how and where of the annotations (coming up in this document shortly).

Persistence of Fields: The fields of the Entity will be saved in the database. JPA can use either your instance variables (fields) or the corresponding getters and setters to access the fields. You are not allowed to mix both methods. If you want to use the setter and getter methods the Java class must follow the Java Bean naming conventions. JPA persists by default all fields of an Entity. If you have fields that should not be saved (like calculated instance variables based on other instance variables), they must be marked with @Transient.

By default each field is mapped to a column with the name of the field. You can change the default name via @Column (name="newColumnName").

- @Column(name = "ID")

Again, see the final annotated ListItem POJO for the how and where of the column name annotations.

Relationship Mapping: JPA allows us to define relationships between classes. It can be defined that a class is part of another class (containment). For example, an Employee object could contain an Address object. Classes can have @OneToOne, @OneToMany, @ManyToOne or @ManyToMany relationships with other classes.

A relationship can be bidirectional or unidirectional. There is a lot of reading you can do on this subject so that you can make sure you are selecting the correct relationship annotation. This document, since we only have one class, will not get into it.

```
//final annotated ListItem POJO
package model;

//All these import statements came in by clicking the red X and using
Import option from the javax.persistence package
//If you are asked to create a class, interface, constant, etc, you
made a spelling or capitalization error!!
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="items")
public class ListItem  {
```

```
	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	@Column(name="ID")
	private int id;
	@Column(name="STORE")
	private String store;
	@Column(name="ITEM")
	private String item;

	public ListItem(){
		super();
	}

	public ListItem(String store, String item){
		super();
		this.store = store;
		this.item = item;
	}

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getStore() {
		return store;
	}
	public void setStore(String store) {
		this.store = store;
	}
	public String getItem() {
		return item;
	}
	public void setItem(String item) {
		this.item = item;
	}
	public String returnItemDetails( ) {
		return store + ": " + item;
	}
}
```

After completing this (and saving), you should have an error on the public class ListItem stating that it's managed but not listed in the persistence.xml file. We'll fix it now.

Set up your persistence.xml to talk to your database & register the entity

Next, we need to tell the persistence data which database we want to persist & where it is

- Located inside src/META-INF is the persistence.xml file
- Navigate to it and click on the General Tab:
  - Persistence Provider: org.eclipse.persistence.jpa.PersistenceProvider
  - Managed classes: model.ListItem (the package and name of your class) (should find/predict if you've got the path correct)
- Connection Tab
  - Transaction Type – Local Resource
  - Driver – com.mysql.cj.jdbc.Driver
  - URL – jdbc:mysql://localhost:3306/shopping (where shopping is the name of your database)
  - User – root
  - Password – Your MySQL password
- Save persistence.xml

Create a way to work with your entity

- Java Application (a simple command line)
- Servlet/JSP pages
- Spring Beans & MVC

We will separate our view (getting the item from the user) from our controller/helper (actually persisting or merging to the database). Our helper will do all the persistence to the database (sometimes you'll hear this referred to as a DAO or Data Access Object). Our view will grab the object and pass it over to the helper to run the appropriate method. Grab the skeleton StartProgram.java and add it to your root src folder (default package). Create a ListItemHelper.java class inside your controller package. You will create the methods in the ListItemHelper and utilize the StartProgram.java for the interface to gather the objects to pass over to the ListItemHelper methods for persistence.

The entity manager (javax.persistence.EntityManager) provides the operations from and to the database, e.g. find objects, persists them, remove objects from the database, etc. Entities, which are managed by an Entity Manager, will automatically propagate these changes to the database (if this happens within a commit statement). If the Entity Manager is closed (via close()) then the managed entities are in a detached state and not committed to the database.

The EntityManager is created by the EntityManagerFactory which is configured by the persistence unit. The persistence unit is described via the persistence.xml file in the META-INF directory of the source folder. A set of entities which are logically connected will be grouped via a persistence unit. The persistence.xml file defines the connection data to the database, e.g. the driver, the user and the password.

Luckily, the EntityManagerFactory and EntityManager does a lot of the behind the scenes work so we can just work with the object. Try this easy example:

- Inside the ***ListItemHelper class***, create a global instance of the EntityManagerFactory
  - static EntityManagerFactory emfactory = Persistence.createEntityManagerFactory("**ConsoleShoppingList** "); (where **ConsoleShoppingList** is the name of your Eclipse Project – if not, check the General Tab of the persistence.xml and put the name from the name field between the double quotes) Make sure you add the two imports.
- Now, let's create a method to accept a ListItem to add. Create a method: public void insertItem(ListItem li)
  - You'll have to tell the ListItemHelper where to find the ListItem class. Add the import statement to the model.ListItem when prompted by the red X.
- Inside that method, add the following code (add the ListItem & EntityManager imports):
  - EntityManager em = emfactory.createEntityManager();
  - em.getTransaction().begin();
  - em.persist(li);
  - em.getTransaction().commit();
  - em.close();
- Now, we need to provide a object to persistence. Flip over to your StartProgram.java and locate the addAnItem( ) method. It already has the code to get a store and an item. We need to wrap that into a ListItem object and send it to that method to persist.
  - ListItem toAdd = new ListItem(store, item);
- To put the item in the database
  - lih.insertItem(toAdd);

Run the StartProgram.java file. You should be able to select 1 from the list, type in a store and item and it will persist. After the menu shows up again, go to your MySQL instance (console window or whatever you're using) to check to see that the new items have been entered. You can use `select * from items;` to see the entries in the table through MySQL.

From this point forward, we'll implement viewing all items, deleting an item and editing and item. The key to understanding is that the StartProgram file will find/create the item, then it will pass the item over to the ListItemHelper to persist or merge to the database.

Java Persistence Query Language (JPQL)

The simplest JPQL query selects all the instances of a single entity type that match the criteria passed in. Consider the following query**: SELECT i FROM ListItem i**
This JPA query is going to select every ListItem from the database and turn them into ListItem objects for us. SQL queries select from a table in MySQL. JPQL selects from an entity from the application domain model.

The syntax is very close to SQL except we return to the objects with an alias (the i in the example above).
SELECT <select_expression>
FROM <from_clause>
[WHERE <conditional_expression>]
[ORDER BY <order_by_clause>]

Show All Items
Let's create a query to show all the items in our database to our console program. Inside the ListItemHelper, create a method called showAllItems( ). This will return a generic list of all the items in the database. Make sure you import the java.util list (and not the java.awt list).

```
public List<ListItem> showAllItems(){
        EntityManager em = emfactory.createEntityManager();
        List<ListItem> allItems = em.createQuery("SELECT i FROM ListItem
i").getResultList();
        return allItems;
        }
```

This queries the database, grabs all the items and turns them into Java objects for our use. Notice the .getResultList( ) at the end of the createQuery tells JPA that we want the whole list. The object the we save the resultList into is a List – they match. If we wanted just one item, we would use:

```
        ListItem oneItem = em.createQuery("SELECT i FROM ListItem i").getSingleResult();
```

This would be a mess to use right now because we have more than one item in the database and we'd need to use a WHERE cause to limit it to one item. We'll use this in a little bit to find a specific item to edit. But you can see the getSingleResult( ) just returns one ListItem so our type that we use would just be one ListItem.

Now, locate the viewTheList( ) method in the StartProgram.java file. Let's capture the results from this method into a list and iterate through it to print them all out.

```
        List<ListItem> allItems = lih.showAllItems();
        for(ListItem singleItem : allItems){
                System.out.println(singleItem.returnItemDetails());
        }
```

Run the StartProgram. java and select 4 from the list. You should see your list display.

TypedQuery
A TypedQuery is a JPA query that returns an object of a certain parameter. We will use it to return a single ListItem object from our database that meets the requirements we pass in.

Parameterizing Your Queries
One item that we will discuss later in our course (during the Developing Secure Java Applications unit) is parameterizing queries so that someone can't inject SQL into your queries to drop your tables.

JPQL supports two types of parameter binding syntax.

The first is positional binding, where parameters are indicated in the query string by a question mark followed by the parameter number. When the query is executed, the developer specifies the parameter number that should be replaced. You are welcome to research this more if desired.

The second type of binding is by using named parameters. For ease of understand of what goes where, we will be using named parameters. Named parameters are indicated in the query string by a colon followed by the parameter name.

When the query is executed, the developer specifies the parameter name that should be replaced. This type of parameter allows for more descriptive parameter specifiers and it easier for developers to understand what is being searched.

Removing an item

First, you have to find the item that you want to remove. Then you actually remove it. In the ListItemHelper, create the following method:

```
public void deleteItem(ListItem toDelete) {
        EntityManager em = emfactory.createEntityManager();
        em.getTransaction().begin();
        TypedQuery<ListItem> typedQuery = em.createQuery("select li from
ListItem li where li.store = :selectedStore and li.item = :selectedItem", ListItem.class);

        //Substitute parameter with actual data from the toDelete item
        typedQuery.setParameter("selectedStore", toDelete.getStore());
        typedQuery.setParameter("selectedItem", toDelete.getItem());

        //we only want one result
        typedQuery.setMaxResults(1);

        //get the result and save it into a new list item
        ListItem result = typedQuery.getSingleResult();

        //remove it
        em.remove(result);
        em.getTransaction().commit();
        em.close();

    }
```

This method takes the ListItem that is passed into it and searches for it by looking for the listitem by the store and item that is passed in. If there are more than one items with the

same store and item, it only selects the first one. It grabs that result and stores it into the result ListItem. Then, we can finally remove the item by using the remove( ) method.

In the StartProgram.java, locate the deleteAnItem( ) method. After getting the store and item, we need to call the deleteItem( ) from the ListItemHelper. Add the following code to deleteAnItem( ) method:

```
ListItem toDelete = new ListItem(store, item);
lih.deleteItem(toDelete);
```

How it works: The user types in a store and item. We create a new item with that information to pass over to the deleteItem method in the List item helper.
Once inside the deleteItem method, we search for the item with that store and item the user entered in. Once we find one result, we ask the list item helper to remove it.

In theory – we need some protection built into this method in case the user enters a value that isn't found – for example, they type 'rarget' instead of 'target'. This lab doesn't implement this but it should!
Note: this is just one approach to deleting an item. You could also search for the item if you know the ID and then delete it. The entitymanager has a method called find( ) that you could use if you know the id:

- ListItem found = em.find(ListItem.class, 3);
  //where 3 coincides with the id of the item

Try to delete one of the items you entered in and then check your list to make sure it's gone.

Updating an item

This is a little more complicated because first we need to find the item before we update it. We have three ways we could find an item:

- Searching by store
- Searching by item
- Searching by id

For all these, the first thing we need to get it the store or item from the user. Inside the StartProgram.java, I have written out the logic for this. You just need to create the methods.

Uncomment out the editAnItem( ) method. There are two methods already referenced in this section. Autostub the searchForItemById(int itToEdit) method into the ListItemHelper.java. Do the same for the updateItem(ListItem toEdit) method.

Before we go away from this method, we need to add some logic to call methods to search by the store or item. Add the following lines to this if statement:

```
if (searchBy == 1) {
        System.out.print("Enter the store name: ");
```

```
                String storeName = in.nextLine();
                foundItems = lih.searchForItemByStore(storeName);
            } else {
                System.out.print("Enter the item: ");
                String itemName = in.nextLine();
                foundItems = lih.searchForItemByItem(itemName);
                }
```

Go ahead and add stubs for those methods as well.

Now, we need to create the methods to search for the items by store or item name. They are actually going to be the same except for the parameter that is searched. Here are those two methods:

>>> Search for Item By Store <<<

```
public List<ListItem> searchForItemByStore(String storeName) {
        // TODO Auto-generated method stub
        EntityManager em = emfactory.createEntityManager();
        em.getTransaction().begin();
        TypedQuery<ListItem> typedQuery = em.createQuery("select li from
ListItem li where li.store = :selectedStore", ListItem.class);
        typedQuery.setParameter("selectedStore", storeName);

        List<ListItem> foundItems = typedQuery.getResultList();
        em.close();
        return foundItems;
    }
```

>>> Search for Item By Item <<<

```
public List<ListItem> searchForItemByItem(String itemName) {
        // TODO Auto-generated method stub
        EntityManager em = emfactory.createEntityManager();
        em.getTransaction().begin();
        TypedQuery<ListItem> typedQuery = em.createQuery("select li from
ListItem li where li.item = :selectedItem", ListItem.class);
        typedQuery.setParameter("selectedItem", itemName);

        List<ListItem> foundItems = typedQuery.getResultList();
        em.close();
        return foundItems;
}
```

Now we have a list of the items that meet those criteria. The next thing our program does in the StartProgram.java is print out those objects that were found that meet the criteria with the id. The user then picks the ID that we want to delete. Now, we need to search for the item by that id. Let's implement the searchForItemById(idToEdit) method.

```
public ListItem searchForItemById(int idToEdit) {
		// TODO Auto-generated method stub
		EntityManager em = emfactory.createEntityManager();
		em.getTransaction().begin();
		ListItem found = em.find(ListItem.class, idToEdit);
		em.close();
		return found;
	}
```

After find the item, the StartProgram.java will echo the found item back to them and ask them if they want to change the store or the item. Depending on what the user picks will determine what is updated. If they select to update the store, the program will use the setter for the setStore to set the new store. After the changes have been made to the item, we need to make the update to the database. This is where the updateItem(toEdit) method will come into play. It will take the new item details and update the item in the database, keeping the ID the same from when it found the item. Implement the updateItem method in the ListItemHelper.

```
public void updateItem(ListItem toEdit) {
		// TODO Auto-generated method stub
		EntityManager em = emfactory.createEntityManager();
		em.getTransaction().begin();

		em.merge(toEdit);
		em.getTransaction().commit();
		em.close();
	}
```

Notice the use of the merge( ) method. After merging in an entity we can change its property and the EntityManager would update the database automatically.

Closing the EntityManagerFactory

One thing that you'll want to make sure you do is close the EntityManagerFactory connection after you are finished making changes to the database. In a real-world application, you would probably do this more frequently but it's still something we need to remember to do so we don't have unused connections open to the database. Inside the ListItemHelper, add this method:

```
	public void cleanUp(){
		emfactory.close();
	}
```

Now, in our runMenu( ) method in the StartProgram.java, run your clean-up method to close those connections after you are finished with the application. It's already there – just need to uncomment it.

```
} else {
        lih.cleanUp();
        System.out.println("   Goodbye!   ");
        goAgain = false;
        }
```

Pulling the Console Based Shopping List Together

We can easily modify this project to make it a web application (make some simple HTML forms and JSP to display the output.) using the ListItemHelper and the ListItem object with the entity annotations and we will later. I've included code on my GitHub account that you're welcome to reference to in case you're not sure where something goes.

https://github.com/kjkleindorfer/ConsoleShoppingList