

Day 17-18-19-20-21-22-23-24/90

Day 25/90

Done

Day 26/90

Java Persistence API (JPA) (continue)

Keying Persistable Maps by *Basic Type*

Keying Persistable Maps by *Entities*

🔗 📌 🔗 Summary

EJB - Enterprise Java Beans

Features of EJB

Architecture of EJB

1 Session Beans :

2 Singleton Beans :

3 Message-Driven Beans :

Lifecycle of EJBs

Done

Day 27/90

EJB - Enterprise Java Beans (continue)

Transaction

Properties Of Transaction

Transaction Management Attributes

Persistence Unit vs Persistence Context - Intro

Entity Manager - How To Get Access

Entity Manager - Operations

Cascade Operations

Entity Detachment

Elements Of Persistence Unit

JPQL - Java Persistence Query Language

@NamedQuery

Combined Path Expressions

Done

Day 28/90

JPQL - Java Persistence Query Language (continue)

Constructor Expression

From Clause - Join

From Clause - Join Maps

From Clause - Fetch Join

Where Clause

Where Clause - Between Operator

Where Clause - Like Operator

Done

Day 17-18-19-20-21-22-23-24/90

Date : 26-April-2023 *till* 3-May-2023

No progress since I had 2 exams one after another. ("Linear Algebra" & "The theory of Languages and machines")

Day 25/90

Date : 04-May-2023

REVISION

Done


☒ Day 25/90

Day 26/90

Date : 05-May-2023

Java Persistence API (JPA) (continue)

Keying Persistable Maps by *Basic Type*

 **Note:** Keying Persistable Maps by Basic Type in JPA JavaEE with `@OneToMany` and `@MapKey`

For example, let's say we have an `Employee` entity and a `Department` entity. We want to create a map that associates each employee with their department. To do this, we can annotate the Employee entity with the `@OneToMany` annotation and set the target entity to the Department entity using the `mappedBy` attribute.

```
@Entity
public class Employee {
    @Id
    private Long id;
    // other attributes

    @ManyToOne
    private Department department;

    // getters and setters
}

@Entity
public class Department {
    @Id
    private Long id;
    // other attributes

    @OneToMany(mappedBy = "department")
    @MapKey(name = "id")
```

```

private Map<Long, Employee> employeesByDepartment;

// getters and setters
}

```

Here, we're using the `@MapKey` annotation to specify that we want the key of the map to be the id of the Employee entity. This means that we can access the employees for a given department using its id.

When we run this code, JPA will generate two tables in the database: one for `Employee` and one for `Department`. The Department table will have a foreign key column referencing the primary key of the Employee table.

=> when we persist data using this mapping, it will create three tables in the database:

1. `Employee` table: This table will have columns for all attributes in the `Employee` class, including a foreign key column referencing the primary key of the associated `Department`.
2. `Department` table: This table will have columns for all attributes in the `Department` class, as well as a primary key column.
3. `Department_employee` table: This table will be generated automatically by Hibernate to manage the one-to-many relationship between `Department` and `Employee`. It will have two columns: a foreign key column referencing the primary key of the `Department` table, and a foreign key column referencing the primary key of the `Employee` table.

Here's some example data :

```

-- Department table
| id | name      |
|----|-----|
| 1  | Sales     |
| 2  | Marketing |
| 3  | IT        |

-- Employee table
| id | name      | department_id |
|----|-----|-----|
| 1  | Alice     | 1            |
| 2  | Bob       | 1            |
| 3  | Charlie   | 2            |
| 4  | Dave      | 3            |

-- Department_employee table
| department_id | employee_id |
|-----|-----|
| 1           | 1           |
| 1           | 2           |
| 2           | 3           |
| 3           | 4           |

```

In this example, there are three departments (Sales, Marketing, and IT) and four employees (Alice, Bob, Charlie, and Dave). The `department_id` column in the `Employee` table is used to link each employee to a specific department, and the `Department_employee` table is used to manage the relationship between departments and employees.

Keying Persistable Maps by *Entities*

Let me explain how to key persistable maps by entity in JPA JavaEE with an example 🤖

To start, let's say we have two entities `Employee` and `Department` in our database. We want to specify the ranks of employees by integer value. The higher the value, the lower the rank of employees. To do this, we can create a map called "`employeeRanks`" in the `Department` entity using the `@OneToMany` annotation.

🔑 Keying Persistable Maps by Entity:

```
@Entity
public class Department {
    @Id
    private Long id;

    @OneToMany(mappedBy = "department")
    @MapKeyJoinColumn(name="employee_id")
    private Map<Employee, Integer> employeeRanks;

    // getters and setters
}

@Entity
public class Employee {
    @Id
    private Long id;

    @ManyToOne
    private Department department;

    // other fields, getters and setters
}
```

In the `Department` entity, we use the `@OneToMany` annotation with the "`mappedBy`" attribute to indicate that this relationship is bidirectional. This means that changes made to the "`employeeRanks`" map on the `Department` side will be updated in the `Employee` entity as well.

We also use the `@MapKeyJoinColumn` annotation to specify that the key of the map should be the `Employee` entity, with the name of the join column being "`employee_id`".


For the `Employee` entity, we use the `@ManyToOne` annotation to establish the relationship between the `Employee` and `Department` entities.

🗄 Database Tables: This setup will create three tables in your database:

- `Department` (with columns: `id`)

- Employee (with columns: id, department_id)
- Department_Employee (with columns: department_id, employee_id, rank)

The Department_Employee table serves as the join table between the Department and Employee entities, with an additional "rank" column to store the employee rank.

 Example: Let's say we have a Department entity with id=1, and two Employee entities with ids 2 and 3 respectively. We want to set their ranks as follows:

- Employee with id 2 has rank 2
- Employee with id 3 has rank 1

Department--table

id
1

Employee--table

id	department_id
2	1
3	1

Department_Employee--table

department_id	employee_id	rank
1	2	2
1	3	1

As you can see, the Department table has one row with id=1. The Employee table has two rows with ids 2 and 3 respectively, both of which have department_id=1 to indicate that they belong to the same department.

The Department_Employee table serves as the join table between the Department and Employee entities. In this case, it has two rows with department_id=1, indicating that both employees belong to the same department. The "employee_id" column identifies which Employee entity each row corresponds to (2 or 3), and the "rank" column stores their respective ranks (2 for Employee id 2, and 1 for Employee id 3).

 **Note** : In a persistable map, When

- the "Key" is a *basicType* (Integer, String, etc), and the "Value" is an *entity*, then we are bound to put `Any_Relationship_Annotation` i.e. (`@OneToMany` etc).
- the "Value" is a *basicType*, then we are bound to put `@ElementCollection`. It doesn't matter what is the "Key" in this case.

Summary

- Java Persistence API (JPA) 🏗️: A framework for managing relational data in Java applications.
- Setting up Payara Server 🖥️: Steps to install and configure the Payara Server, which is a popular application server that supports JPA.
- JPA Entity 🏠: An annotated Java class that represents a persistent object in a database.
- Customizing Table Mapping 📄: Techniques for mapping an entity to a specific table in the database.
- Using Super Classes 🐶: Inheriting properties from a parent class when defining JPA entities.
- Overriding Super Class Field ↩️: Changing the behavior of inherited fields in a subclass.
- Mapping Simple Java Types 📄: Translating basic Java data types (such as int and String) to their equivalent database types.
- Transient Fields 🗑️: Fields that are not persisted to the database.
- Field Access Type 🗑️: Defining whether JPA should access fields directly or through getter/setter methods.
- Mapping Enumerator Type 📄: Persisting Java enums as database values.
- Mapping Large Objects (e.g. images) 🖼️: Techniques for storing large binary data (such as images) in a database.
- Lazy & Eager Fetching Of Entity State 🏠🐶: Specifying when JPA should load related objects eagerly (right away) or lazily (on demand).
- Mapping Java 8 DateTime Types 🕒: Storing date/time data using the new Java 8 Date/Time API.
- Mapping Embeddable classes 📄: Including non-entity classes inside an entity class.
- Mapping Primary Keys 🔑: Defining how JPA should generate primary keys for entities.
- Auto Primary Key Generation Strategy 🏠: Configuring JPA to automatically generate primary keys for entities.
- Entity Relationship Mapping 🐶: Defining how different JPA entities are related to each other.
- Roles 🐶: Identifying the role that an entity plays in a relationship (such as "owner" or "child").
- Directionality ↔: Specifying whether a relationship is unidirectional or bidirectional.
- Cardinality 1, * 📄: Defining the number of entities that can be associated with another entity in a relationship.
- Ordinality 1, 1 📄: Defining the order of entities in a relationship.
- Ownership of Relationships 🏠: Identifying which entity "owns" the relationship and controls cascade operations (such as deletes).
- Unidirectional ➡️: A relationship where one entity has a reference to another entity, but the reverse is not true.
- Bidirectional ↔: A relationship where two entities have references to each other.
- @ManyToOne 🐶: An annotation used to define a many-to-one relationship between two entities.

- Fetch Mode 🐾 🌈: Specifying how JPA should load related objects (eagerly or lazily) for collection-valued relationships.
- Collection Mapping Of Embeddable Objects and Collection Table 🏠 👥: Techniques for mapping embedded objects and collections to database tables.
- Ordering The Contents Of a Persistable Collection 📄 123: Defining the order in which objects should be retrieved from a collection-valued relationship.
- Mapping Persistable Maps 📖 🗑️: Techniques for persisting maps (key-value pairs) to a database.
- Using Enums As Persistable Map Keys 📖 🔑: Storing Java enums as keys in a map that is persisted to the database.
- Keying Persistable Maps by Basic Type 🔑 📖: Using basic Java data types (such as String or int) as keys in a map that is persisted to the database.
- Keying Persistable Maps by Entities 🔑 📖 🏠: Using other JPA entities as keys in a map that is persisted to the database.

EJB - Enterprise Java Beans

🤖 Imagine you run a restaurant and your customers expect efficient service. You can hire a lot of staff to handle different tasks, but it's hard to manage them all on your own. That's where Enterprise JavaBeans (EJBs) come in!

🍳 An EJB is like a specialized employee at your restaurant who handles specific tasks, such as taking orders or preparing food. Just like how your employees have specific roles, EJBs have predetermined roles and responsibilities within a JavaEE application.



🤖 EJB stands for Enterprise JavaBeans, which is a technology used in JavaEE to develop distributed and scalable applications.


🏢 In an enterprise application, there are many different components that need to work together, like databases, web servers, and client applications. EJBs provide a standardized way to manage these components by defining roles and responsibilities for each component.

🍷 For example, imagine you want to add a new dish to your menu. With EJBs, you can easily create a "Recipe EJB" that manages the data around your new dish, like its ingredients, cooking instructions, and nutritional information. This makes it easy for other parts of your application, like the ordering system, to access and use this information.

👤 Some examples of EJBs in JavaEE include:

- Entity Beans: Used for representing persistent data in a database.
- Message-Driven Beans: Used for processing messages asynchronously.
- 🔍 For example, Session Beans are EJBs that handle business logic and are responsible for processing client requests. Entity Beans represent persistent data stored in a database, while Message-Driven Beans handle asynchronous messaging.
- 💻 EJBs also provide built-in services like transaction management, security, and persistence, which makes it easier for developers to focus on writing business logic rather than worrying about low-level details.

-  One of the big advantages of using EJBs is that they make it easy to scale your application horizontally (i.e., across multiple machines) as demand grows. Because each EJB has a well-defined role and can communicate with other EJBs, you can add more instances of an EJB to handle increased load without disrupting the rest of your application.
-  Another advantage of EJBs is that they make it easier to secure your application. By defining roles and permissions for each EJB, you can control access to sensitive data and functionality.

 EJBs are a technology in JavaEE that allow developers to build scalable, maintainable, and distributable applications. They provide a set of services, such as transaction management, security, and persistence, that help simplify the development process.

Features of EJB

1. Declarative Metadata

Declarative metadata is information about your code that is specified outside of the code itself. In the context of Java EE EJB, this refers to annotations or XML files that provide additional information about your Enterprise JavaBeans. This metadata helps the application server understand how to manage and run your EJB.

2. Configuration by Exception

Configuration by exception is a design pattern used in Java EE EJB where default settings are used unless otherwise specified. This means that instead of having to explicitly configure every aspect of your EJB, you only need to specify any exceptions or deviations from the default behavior.

3. Dependency Management

Dependency management is the process of identifying and resolving dependencies between different components in your application. In Java EE EJB, this involves managing dependencies between your Enterprise JavaBeans and other resources like databases, message queues, and web services.

4. Lifecycle Management

Lifecycle management refers to the various stages an EJB goes through during its lifetime, including creation, activation, passivation, and removal. The application server manages the lifecycle of EJBs and ensures they are instantiated and destroyed at the appropriate times.

5. Scalability

Scalability refers to the ability of an application to handle increasing amounts of traffic or workload. In Java EE EJB, this involves horizontal scaling, where multiple instances of an EJB are created and distributed across multiple servers to handle increased demand.

6. Transactionality

Transactionality refers to the ability of an EJB to participate in transactions - groups of operations that are either all completed successfully or rolled back if any one operation fails. Transactions ensure data consistency and integrity in complex systems.

7. Security

Security refers to the measures taken to protect an application from unauthorized access and ensure data confidentiality, integrity, and availability. In Java EE EJB, this involves authentication (verifying user identities) and authorization (granting permissions based on those identities).

8. Portability 🚢

Portability refers to the ability to move an application between different environments or platforms without modification. In Java EE EJB, this means that your Enterprise JavaBeans should be able to run on any Java EE-compliant application server without needing to be modified.

Architecture of EJB

📄 Note: Different Kinds of EJB Architecture ☀️

Enterprise JavaBeans (EJB) is a technology used to develop modular components that can be distributed across different systems and platforms. There are three types of EJB architecture:

1 Session Beans :

Session beans are used to represent a transient conversation between a client and an application server. They can be stateless or stateful, depending on whether or not they maintain state information between method invocations.

Stateless session beans do not maintain conversational state, while *stateful* session beans maintain state information. The main advantage of session beans is that they provide a way to encapsulate business logic in a module that can be accessed by multiple clients. These are the most common type of EJB. They represent individual client sessions and perform business logic. For example, a session bean might handle user authentication or manage a shopping cart for an online store.

====> *Stateful session* :

Here's an example: Let's say you have a shopping cart application and you want to keep track of the items that a particular user has added to their cart. You can use a stateful EJB to maintain the state of the user's shopping cart throughout their session. 🛒🛒

When the user adds an item to their cart, the stateful EJB updates its state accordingly. When the user removes an item from their cart, the EJB updates its state again. This way, the EJB can keep track of the user's shopping cart throughout their session. 🖥️🛒

```
@Stateful
public class ShoppingCart implements ShoppingCartRemote {

    private List<String> items = new ArrayList<>();

    public void addItem(String item) {
        items.add(item);
    }

    public void removeItem(String item) {
        items.remove(item);
    }

    public List<String> getItems() {
```

```

        return items;
    }

    @Remove
    public void checkout() {
        // Perform checkout logic here
    }
}

```

In this example, we define a stateful EJB called "ShoppingCart" that implements the remote interface "ShoppingCartRemote". The stateful nature of this EJB is evident by the fact that it maintains a list of items added to the shopping cart.

The `addItem` and `removeItem` methods allow the client to add or remove items from their shopping cart, while the `getItems` method allows the client to retrieve the current contents of their cart.

Finally, the `checkout` method performs any necessary actions when the client checks out their shopping cart, such as calculating the total cost and updating the inventory. The `@Remove` annotation indicates that the EJB should be removed once the client has checked out.

====> *Stateless session* :

Here's an example: Let's say you have a banking application that exposes a service to transfer money from one account to another. You can use a stateless EJB to implement this service. The EJB would handle the transfer operation, but it wouldn't maintain any state about the transfer itself. 🗉 💰

When a client submits a transfer request, the stateless EJB performs the necessary operations to transfer the money and returns a result. The EJB doesn't need to keep track of the transfer beyond that point, as it has no further impact on the operation of the system. 📈 🏦

One advantage of using stateless EJBs is that they can be more scalable than stateful EJBs, as they don't require resources to maintain conversational state with clients. However, it's important to carefully consider whether a stateless EJB is appropriate for your application, as they may not be suitable for all use cases. 🤖

Here's an example of a stateless EJB in Java code:

```

@Stateless
public class BankTransfer implements BankTransferRemote {

    public boolean transfer(String accountFrom, String accountTo, double amount) {
        // Perform the necessary operations to transfer the money
        return true;
    }
}

```

In this example, we define a stateless EJB called "BankTransfer" that implements the remote interface "BankTransferRemote".

The `transfer` method takes three parameters: the account to transfer from, the account to transfer to, and the amount to transfer. When invoked, the EJB performs the necessary operations to transfer the money between the accounts and returns a result indicating whether the transfer was successful or not.

2 Singleton Beans :

A singleton bean in EJB is a type of EJB that only allows one instance to be created and shared across multiple clients. The singleton bean is instantiated when the application server starts up and remains in memory until the application server shuts down. 🧑🏽 🏠 💬

Here's an example:

```
@Singleton
public class MySingletonBean {

    private int count = 0;

    public int getCount() {
        return count;
    }

    public void incrementCount() {
        count++;
    }
}
```

In this example, we have defined a singleton bean called `MySingletonBean`, which has a single instance that is shared by all clients. The bean has a method `getCount()` that returns the current value of a private variable `count`, and a method `incrementCount()` that increments the value of `count` by one. 🙌 💻 🔥

To use this singleton bean in your EJB application, you would simply inject it into your client code like this:

```
@Stateless
public class MyStatelessBean {

    @EJB
    private MySingletonBean singletonBean;

    public void doSomething() {
        int count = singletonBean.getCount();
        singletonBean.incrementCount();
        // do something with count
    }
}
```

In this example, we have a stateless bean called `MyStatelessBean` that injects an instance of `MySingletonBean` using the `@EJB` annotation. The stateless bean then calls the `getCount()` method to get the current value of `count`, and the `incrementCount()` method to increment it.

3 Message-Driven Beans :

Message-driven beans (MDBs) are used to process messages asynchronously. They are triggered by messages sent to a message queue or topic, and can perform processing tasks based on the contents of the message. MDBs are typically used in enterprise integration scenarios, where disparate systems need to exchange data asynchronously. One of the key advantages of MDBs is that they allow for loosely coupled integrations between systems. These beans receive and process messages asynchronously. They are commonly used in messaging systems and event-driven architectures. For example, a message-driven bean might process incoming orders for an online store or update a user's account based on a system event.

Lifecycle of EJBs

1. Stateful Session Beans 🌱❤️

Stateful Session Beans (SFSB) maintain state information between client invocations, meaning that each client has a unique bean instance. The lifecycle of an SFSB is as follows:

- **Creation:** The container creates a new instance of the SFSB when a client makes a request.
- **Method Invocation:** The client invokes methods on the bean, which modifies its state.
- **Passivation:** If the bean is not accessed for a specified amount of time, the container can choose to serialize its state and remove it from memory to free up resources.
- **Activation:** When a client requests a passivated SFSB, the container restores its state from the serialized data and activates the bean instance.
- **Removal:** The container removes the SFSB instance when the client session ends or when the bean is explicitly removed by the client.

Example: A shopping cart in an online store would be a good example of an SFSB since it needs to maintain state information (items in the cart) for a specific client session.

2. Stateless Session Beans 🌱🔄

Stateless Session Beans (SLSB) do not maintain any state information between client invocations, meaning that each client request is processed independently. The lifecycle of an SLSB is as follows:

- **Creation:** The container creates a pool of bean instances when the application starts up.
- **Method Invocation:** The client invokes methods on the bean, which performs some processing and returns a result.
- **Destruction:** The container returns the bean instance to the pool after the method invocation, making it available for another client request.

Example: A calculator that performs some computation based on the input values would be a good example of an SLSB since it does not need to maintain any state information between client invocations.

3. Singleton Session Beans 🌱👑

Singleton Session Beans (SSB) maintain a single instance of the bean for the entire application, meaning that all clients share the same instance. The lifecycle of an SSB is as follows:

- Creation: The container creates a single instance of the SSB when the application starts up.
- Method Invocation: All clients invoke methods on the same instance, which modifies its state.
- Destruction: The container destroys the SSB instance when the application shuts down or when it is explicitly removed by the client.

Example: A configuration manager that maintains some global settings for the entire application would be a good example of an SSB since it needs to maintain a single instance with the same state information across all client requests.

Done

✅ Day 26/90 ==> [JavaEE: Day 26/90 - Java Persistence API \(JPA\)](#)

Day 27/90

Date : 06-May-2023

EJB - Enterprise Java Beans (continue)

Transaction

😬 So what is a transaction? A transaction is a logical unit of work that groups together several database operations. These operations are executed as a single unit, which either succeeds or fails completely.

💰 Let's say you want to transfer money from one bank account to another. A transaction would ensure that both accounts are updated correctly, and that no money is lost along the way. If something goes wrong during the transfer (e.g. a network error), the transaction can be rolled back, and both accounts will remain unchanged.

📄 In JavaEE, transactions are managed by the container (e.g. Tomcat, Glassfish, JBoss). You start a transaction by marking a method with the `@Transactional` annotation. The container then takes care of managing the transaction for you.

🚀 Here's an example:

```
@Transactional
public void transferMoney(Account sourceAccount, Account destinationAccount, double
amount) {
    try {
        // Deduct the amount from the source account
        sourceAccount.setBalance(sourceAccount.getBalance() - amount);
        accountDao.update(sourceAccount);
    }
}
```

```

        // Add the amount to the destination account
        destinationAccount.setBalance(destinationAccount.getBalance() + amount);
        accountDao.update(destinationAccount);
    } catch (Exception e) {
        // Something went wrong, so roll back the transaction
        throw new RuntimeException(e);
    }
}

```

👁 In this example, we're transferring money from one account to another. We've marked the method with the `@Transactional` annotation, which tells the container to manage the transaction for us.

🏠 First, we deduct the amount from the source account and update its balance in the database. Then, we add the same amount to the destination account and update its balance as well. If anything goes wrong during the transfer (e.g. an exception is thrown), the container will automatically roll back the transaction, and both accounts will remain unchanged.

👉 And that's transactions in a nutshell! They're a powerful tool for ensuring data integrity and consistency in your applications.

Properties Of Transaction

1. 🔥 **Atomicity** : Atomicity is the property of a transaction that ensures that all the database operations within the transaction are treated as a single, indivisible unit. This means that either all the operations within the transaction succeed or none of them do. If any operation fails, the entire transaction is rolled back to the previous state.


💎 For example, let's say you're buying a pair of shoes online. The transaction would involve deducting the amount from your bank account and updating the inventory system to reduce the number of shoes in stock. If either operation fails, the entire transaction must fail, and the bank account and inventory should remain unchanged.

2. 🧑 **Consistency** : Consistency is the property of a transaction that ensures that the database remains in a valid state throughout the entire transaction. This means that any changes made to the database within the transaction must satisfy all the integrity constraints and business rules defined for the database.

🛒 For example, if you're buying a product online, the transaction must ensure that the total cost of the purchase is correctly calculated and that any discounts or taxes are applied correctly according to the business rules.

3. 💛 **Isolation** : Isolation is the property of a transaction that ensures that each transaction is independent of other concurrent transactions executing on the same database. This means that the results of one transaction should not affect the results of other transactions executing concurrently.

👥 For example, if two people are booking tickets for the same movie at the same time, each transaction should be isolated from the other. This ensures that one person's booking does not affect the other person's booking, and both bookings can proceed independently.

4.  **Durability** : Durability is the property of a transaction that ensures that once the transaction is committed, its effects are permanent and persistent, even in the face of system failure (such as power outage or hardware failure). This means that the changes made to the database within a committed transaction must survive any subsequent system failures.

💡 For example, if you're transferring money from one account to another, the transaction must ensure that the transfer is recorded permanently, even if there's a power outage or hardware failure during the transaction. Once the transaction is committed, it should be durable, and the money transfer should be reflected in both accounts, even in the event of a system failure.

Transaction Management Attributes

In JavaEE, transaction management is a way of ensuring that changes made to a database or other data store are handled consistently and reliably. It's like making sure that each step in a process is completed before moving on to the next one.

To understand this better, imagine you're trying to transfer money from one bank account to another. You wouldn't want the money to be deducted from one account without being added to the other account, right? That's where transaction management comes in.

JavaEE has a few different attributes for managing transactions, but let's focus on two of the most basic ones: `REQUIRED` and `REQUIRES_NEW`.

- The `REQUIRED` attribute means that the method being called must participate in an existing transaction if one exists. If there is no current transaction, a new one will be started.

For example, imagine you have a method that updates the balance of a bank account:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void updateAccountBalance(String accountNumber, double amount) {
    // code to update the balance goes here
}
```

If this method is called while a transaction is already in progress (e.g. because it's part of a larger banking transaction), it will participate in that transaction. If not, a new transaction will be started specifically for this method.

- The `REQUIRES_NEW` attribute means that a new independent transaction will always be started, even if there is already an existing transaction.

Using the same example as above, here's how you could modify the code to use `REQUIRES_NEW`:

```
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void updateAccountBalance(String accountNumber, double amount) {
    // code to update the balance goes here
}
```

In this case, a new transaction will always be started whenever this method is called, regardless of whether there is already a transaction in progress.

- **MANDATORY**: This attribute specifies that the method being called must be executed within an active transaction context. If there is no active transaction, a `TransactionRequiredException` will be thrown.

For example:

```
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public void updateUserData(String username, String data) {
    // code to update user data goes here
}
```

This method can be called only if there is already an active transaction in progress. If there is no active transaction, then a `TransactionRequiredException` will be thrown.

- **SUPPORTS**: This attribute specifies that the method being called supports transactions, but does not require one. If there is an active transaction in progress, the method will participate in it. If there is no active transaction, the method will execute without a transaction.

For example:

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public int getUserCount() {
    // code to count users in the database goes here
}
```

This method can be called with or without an active transaction in progress. If there is an active transaction, it will participate in that transaction. If not, it will execute without a transaction.

- **NOT_SUPPORTED**: This attribute specifies that the method being called should not be executed within a transaction context. If there is an active transaction in progress, it will be suspended while this method executes.

For example:

```
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public List<String> getPublicData() {
    // code to fetch public data goes here
}
```

This method should never be called within an active transaction. If there is an active transaction in progress, it will be suspended while this method executes. When the method completes, the transaction will resume.

Persistence Unit vs Persistence Context - Intro



A Persistence Unit in Java EE is like a container for all the information needed to connect to a database. It contains the configuration and mapping files, as well as the database connection details.



A Persistence Context in Java EE is like a workspace where data is manipulated before it is saved to the database. It acts as a cache of entity instances that are managed by the `EntityManager`.

In simpler terms, a Persistence Unit provides the information needed to connect to a database, while a Persistence Context manages the interaction between the application and the database by caching and manipulating data.

Entity Manager - How To Get Access

👉 What is an Entity Manager? An Entity Manager is a Java EE component that manages the lifecycle of entities within a Java Persistence API (JPA) context. In simpler terms, it's what allows you to interact with your database using JPA.

👉 How do you get access to an Entity Manager? You can obtain an instance of the Entity Manager by using Dependency Injection or JNDI lookup. Here's an example of using DI:

```
@PersistenceContext
private EntityManager entityManager;
```

In this example, we're injecting an instance of the Entity Manager into a managed bean using the `@PersistenceContext` annotation.

👉 How do you use the Entity Manager? Once you have access to the Entity Manager, you can use it to perform CRUD (Create, Read, Update, Delete) operations on your entities. Here's an example of persisting an entity to the database:

```
Customer customer = new Customer();
customer.setName("John Doe");
customer.setEmail("johndoe@example.com");

entityManager.persist(customer);
```

In this example, we're creating a new `Customer` entity, setting some properties on it, and then using the `persist()` method of the Entity Manager to save it to the database.

👉 How does this compare to Spring Framework? In Spring Framework, you can also use JPA for database interaction, but the way you obtain an instance of the Entity Manager is slightly different. Instead of using Dependency Injection or JNDI lookup, you would typically use the `LocalContainerEntityManagerFactoryBean` to create an instance of the Entity Manager.

Here's an example of configuring the Entity Manager in Spring Framework:

```

@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean em =
        new LocalContainerEntityManagerFactoryBean();
    em.setDataSource(dataSource());
    em.setPackagesToScan(new String[] { "com.example.entities" });

    JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    em.setJpaVendorAdapter(vendorAdapter);

    return em;
}

```

In this example, we're creating a `LocalContainerEntityManagerFactoryBean` and configuring it with a `datasource`, packages to scan for entities, and a JPA vendor adapter (in this case, Hibernate).

Once you have configured the Entity Manager in Spring Framework, you can use it in a similar way to Java EE:

```

@Autowired
private EntityManager entityManager;

@Transactional
public void saveCustomer(Customer customer) {
    entityManager.persist(customer);
}

```

In this example, we're injecting an instance of the Entity Manager using `@Autowired` and then using it to persist a `Customer` entity to the database.

Entity Manager - Operations

Let's consider two entities `Student` and `Course` with a many-to-many relationship between them. One student can enroll in multiple courses, and one course can have multiple students enrolled.

Here are the entity classes with their respective relationships:

```

@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany(mappedBy = "students")
    private List<Course> courses = new ArrayList<>();

    // constructors, getters, setters, and other methods
}

```

```

@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany
    @JoinTable(
        name = "course_student",
        joinColumns = @JoinColumn(name = "course_id"),
        inverseJoinColumns = @JoinColumn(name = "student_id"))
    private List<Student> students = new ArrayList<>();

    // constructors, getters, setters, and other methods
}

```

Now, let's perform some entity manager operations on these entities:

1. `persist`: This operation is used to save a new entity to the database.

👉 Example: Let's create a new `Student` entity and enroll them in an existing `Course` entity using the `persist` operation:

```

EntityManager entityManager = getEntityManager(); // assume this method returns an
instance of EntityManager

// Retrieve an existing course from the database
Course course = entityManager.find(Course.class, courseId);

// Create a new student
Student student = new Student();
student.setName("John");

// Enroll the student in the course
course.getStudents().add(student);

// Save the new student to the database
entityManager.persist(student);

```

2. `find`: This operation is used to retrieve an entity from the database based on its primary key.

👉 Example: Let's retrieve a specific `Course` entity from the database and print its enrolled students using the `find` operation:

```

EntityManager entityManager = getEntityManager(); // assume this method returns an
instance of EntityManager

// Retrieve the course with ID 1
Course course = entityManager.find(Course.class, 1L);

// Print the names of all students enrolled in the course
for (Student student : course.getStudents()) {
    System.out.println(student.getName());
}

```

3. `remove`: This operation is used to delete an entity from the database.

👉 Example: Let's remove a specific `Student` entity from the database along with their enrollment in all courses using the `remove` operation:

```

EntityManager entityManager = getEntityManager(); // assume this method returns an
instance of EntityManager

// Retrieve the student we want to delete
Student student = entityManager.find(Student.class, studentId);

// Remove the student from all courses they are enrolled in
for (Course course : student.getCourses()) {
    course.getStudents().remove(student);
}

// Remove the student itself
entityManager.remove(student);

```

4. `merge`: This operation is used to update an existing entity in the database.

👉 Example: Let's update a specific `Course` entity in the database by adding a new `Student` entity to it using the `merge` operation:

```

EntityManager entityManager = getEntityManager(); // assume this method returns an
instance of EntityManager

// Retrieve the course we want to update
Course course = entityManager.find(Course.class, courseId);

// Create a new student
Student student = new Student();
student.setName("Jane");

// Enroll the student in the course
course.getStudents().add(student);

// Update the course in the database
entityManager.merge(course);

```

Cascade Operations

Cascade operations refer to a set of actions that are automatically applied to related entities when an operation is performed on a parent entity. These operations include persisting (📄), removing (✖), refreshing (↺), merging (🔄), detaching (👤), or applying the operation to all related entities (💡).

For example, let's say we have a database with two tables: `Order` and `Item`. An order can have multiple items associated with it. We can use cascade operations to automatically perform actions on the associated items when an action is performed on the order.

If we set the cascade type to `PERSIST`, then when we save a new order to the database, any new items associated with that order will also be saved automatically. Similarly, if we set the cascade type to `REMOVE`, then when we delete an order from the database, any associated items will also be deleted automatically.

Here's some example code to demonstrate :

1. Persist (📄): This operation is used to save new entities into the database.

```
@Entity
public class Order {
    @Id
    private Long id;

    @OneToMany(mappedBy="order", cascade=CascadeType.PERSIST)
    private List<Item> items;

    // getters and setters
}

@Entity
public class Item {
    @Id
    private Long id;

    @ManyToOne
    private Order order;

    // getters and setters
}

// create a new order with two items
Order order = new Order();
Item item1 = new Item();
Item item2 = new Item();

// associate the items with the order
item1.setOrder(order);
item2.setOrder(order);

// add the items to the order's list of items
order.getItems().add(item1);
```

```
order.getItems().add(item2);

// persist the order (and its associated items)
entityManager.persist(order);
```

In this example, we create a new `Order` object and associate two new `Item` objects with it. We then set the cascade type to `PERSIST` on the `items` field of the `Order` entity, which means that when we persist the `Order` to the database, any new `Item` objects associated with it will also be persisted automatically.

2. Remove (✖): This operation is used to delete existing entities from the database.

```
// remove an order (and its associated items)
Order order = entityManager.find(Order.class, orderId);
entityManager.remove(order);
```

In this example, we retrieve an existing `Order` object from the database using its `id` field, and then call `entityManager.remove(order)` to delete it from the database. Because we've set the cascade type to `REMOVE` on the `items` field of the `Order` entity, any associated `Item` objects will also be deleted automatically.

3. Refresh (↶): This operation is used to reload the state of an entity from the database.

```
// refresh an order (and its associated items)
Order order = entityManager.find(Order.class, orderId);
entityManager.refresh(order);
```

In this example, we retrieve an existing `Order` object from the database using its `id` field, and then call `entityManager.refresh(order)` to reload its state from the database. Because we've set the cascade type to `REFRESH` on the `items` field of the `Order` entity, any associated `Item` objects will also be refreshed automatically.

4. Merge (↻): This operation is used to update an entity in the database. It merges the state of an entity with the state of the corresponding managed entity in the persistence context.

```
// modify an order (and its associated items) and merge changes into the
database
Order order = entityManager.find(Order.class, orderId);
order.getItems().remove(0); // remove the first item from the order's list of
items

entityManager.merge(order); // merge the changes into the database
```

In this example, we retrieve an existing `Order` object from the database using its `id` field, modify it by removing the first `Item` object from its list of items, and then call `entityManager.merge(order)` to merge the changes back into the database. Because we've set the cascade type to `MERGE` on the `items` field of the `Order` entity, any associated `Item` objects will also be merged automatically.

5. Detach (👤): This operation is used to detach an entity from the persistence context, making it no longer managed.

```
// detach an order (and its associated items) from the persistence context
Order order = entityManager.find(Order.class, orderId);
entityManager.detach(order); // detach the order from the persistence context
```

In this example, we retrieve an existing `Order` object from the database using its `id` field, and then call `entityManager.detach(order)` to detach it from the persistence context. Because we've set the cascade type to `DETACH` on the `items` field of the `Order` entity, any associated `Item` objects will also be detached automatically.

6. All (💎): This operation applies the specified operation to all related entities.

```
@Entity
public class Order {
    @Id
    private Long id;

    @OneToMany(mappedBy="order", cascade=CascadeType.ALL)
    private List<Item> items;

    // getters and setters
}

// remove an order (and its associated items)
Order order = entityManager.find(Order.class, orderId);
entityManager.remove(order);
```

In this example, we retrieve an existing `Order` object from the database using its `id` field, and then call `entityManager.remove(order)` to delete it from.

Entity Detachment

In Java EE (Enterprise Edition), an entity refers to a specific object or data that is managed by a persistence framework like JPA. Entity detachment simply means that the framework stops managing that particular entity object (the entity gets out of the persistence context).

👉 Imagine you have a toy car (🚗) that your friend is playing with. As long as your friend is holding onto the car, they are in control of its movements and can make changes to it. This is similar to how a persistence framework controls an entity object.

👉 Now, if your friend puts the car down (detaches it), it's no longer under their control. They can't move it around anymore or make any changes to it. Similarly, when an entity is detached in Java EE, the persistence framework no longer manages changes to that object.








This can happen for a variety of reasons, such as when an entity is no longer needed in the context of a transaction, or when the persistence context (the set of managed entities) is closed.

Elements Of Persistence Unit

In Java EE, a persistence unit represents a set of entity classes that are managed together by the Java Persistence API (JPA). It provides configuration information to JPA, such as the database connection details and the entity classes to be managed.

Here's what each of the elements means :

```
<persistence-unit name="UserPersistenceUnit" transaction-type="JTA">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <jta-data-source>java:/comp/env/jdbc/userDataSource</jta-data-source>
  <class>com.example.User</class>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create"/>
  </properties>
</persistence-unit>
```

1. The `<persistence-unit>` element is used to define a persistence unit in Java EE. It has several attributes and child elements that are used to configure the persistence unit.
 - `name="UserPersistenceUnit"`: This attribute sets the name of the persistence unit to "UserPersistenceUnit". The name is used to uniquely identify the persistence unit within an application.
 - `transaction-type="JTA"`: This attribute specifies the type of transaction management that should be used for this persistence unit. In this case, "JTA" (Java Transaction API) is used, which means that transaction management will be handled by the application server.
2. Persistence provider:  The `<provider>` element is used to specify the JPA provider that should be used to manage the persistence unit. In this case, the Hibernate JPA provider is used (`org.hibernate.jpa.HibernatePersistenceProvider`).
3. JTA datasource:   The JTA datasource specifies the connection details required to connect to the database. It includes information such as the URL, username, and password. The `<jta-data-source>` element is used to specify the JNDI name of the DataSource that should be used for database connections. In this case, the JNDI name is set to `java:/comp/env/jdbc/userDataSource`.
4. Entity classes:  Entity classes are Java classes that represent tables in the database. They typically contain fields or properties that map to the columns in the table. The `<class>` element is used to specify the entity classes that should be managed by the persistence unit. In this case, the entity class `com.example.User` is specified.
5. Schema & script generation:   Schema and script generation refer to the process of creating database tables from the entity classes. This can be done automatically based on the entity mappings, or manually using SQL scripts.
6. Properties:  The `<properties>` element is used to specify additional properties that should be used when configuring the persistence unit. In this case, the `hibernate.hbm2ddl.auto` property is set to "create", which tells Hibernate to automatically generate the database schema based on the entity mappings.

For example, let's say we have an application that needs to manage a set of user data in a MySQL database. We might define a persistence unit like this in our `persistence.xml` file:

This defines a persistence unit named "UserPersistenceUnit" with a JTA transaction type. The provider is set to use Hibernate, and the JTA datasource is specified as "java:/comp/env/jdbc/userDataSource". We also specify that we want to manage a single entity class called "User". Finally, we set the `hibernate.hbm2ddl.auto` property to "create", which tells Hibernate to automatically generate the database schema based on our entity mappings.

To create this `persistence.xml` file, we can simply create a new XML file in our project and add the above code to it. We then need to make sure that the file is located in the `src/main/resources/META-INF/persistence.xml` directory of our application's classpath.

JPQL - Java Persistence Query Language

JPQL (Java Persistence Query Language) is a powerful tool in the JavaEE world for working with databases 🗄️. It's like a language that allows you to ask questions and retrieve data from your database 💬.

With JPQL, you can write queries to retrieve specific information from your entities, which are like tables in your database 📊. You can also join multiple entities together to get more complex results 🧡.

JPQL uses object-oriented terms rather than SQL's table and column terminology 🚀. For example, instead of selecting columns from a table, you select attributes from an entity.

@NamedQuery

In JavaEE, we often use the Java Persistence API (JPA) to work with databases 🗄️. One way to retrieve data from a database using JPA is by writing dynamic queries. These are queries that are built at runtime based on user input 📊.

However, there's another approach called "@NamedQuery" that allows you to create pre-defined named queries in your entity classes 📝.

Here's an example of how it works:

```
@Entity
@NamedQuery(name = "findEmployeeByName",
            query = "SELECT e FROM Employee e WHERE e.name = :name")
public class Employee {
    ...
}
```

In this example, we've added a named query to the Employee entity class. The query is named "findEmployeeByName" and selects all employees whose name matches the provided parameter ":name".

we can name the jpql this way too :

```

@Entity
@NamedQuery(name = Employee.FIND_EMPLOYEE_BY_NAME, //<-- this line is updated
           query = "SELECT e FROM Employee e WHERE e.name = :name")
public class Employee {
    private final String FIND_EMPLOYEE_BY_NAME = "Employee.findEmployeeByName";
    //<-- this line is new
    ...
}

```

With this named query, we can then easily execute the query from our code like this:

```

Query query = entityManager.createNamedQuery("findEmployeeByName");
query.setParameter("name", "John Doe");
List<Employee> employees = query.getResultList();

```

or :



```


TypedQuery<Employee> query =
entityManager.createNamedQuery(Employee.FIND_EMPLOYEE_BY_NAME, Employee.class);
List<Employee> employees = query.getResultList();

// we can make it simpler :
List<Employee> employees =
entityManager.createNamedQuery(Employee.FIND_EMPLOYEE_BY_NAME,
Employee.class).getResultList();

```

This will fetch all the employees with the name "John Doe" from the database.

The advantage of using named queries over dynamic queries is that they are pre-compiled and can be reused multiple times without having to rebuild the query each time the query is executed . This can lead to better performance and less overhead in your application .

When comparing Spring Framework to JavaEE, both frameworks support named queries and dynamic queries. However, Spring provides more advanced features, such as the ability to create specifications and criteria queries, which allows for even more flexibility and reusability in your codebase .

In terms of using named queries in Spring Boot, the process is as following : You would define your named query in the entity class :

```



@Entity
@NamedQuery(name = "findEmployeeByName",
           query = "SELECT e FROM Employee e WHERE e.name = :name")
public class Employee {
    ...
}

```

And then you would use the EntityManager or a JpaRepository to execute the named query :

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    List<Employee> findEmployeesByName(String name);
}
```

In this example, we're using the `JpaRepository` interface provided by Spring Data JPA to automatically create a repository implementation for us. We define the "findEmployeesByName" method just like before, and Spring Data JPA takes care of generating the query for us based on the method name and signature.

The advantage of using named queries in Spring Boot is the same as in Spring Framework and JavaEE: they are pre-compiled and can be reused multiple times without having to rebuild the query each time the query is executed . This can lead to better performance and less overhead in your application .

Combined Path Expressions

In JPQL, you can use combined path expressions to navigate through multiple related entities in a single query. This is useful when you need to retrieve data from associated entities that are linked by relationships like One-to-One, One-to-Many, or Many-to-Many.

For example, let's say we have two entities: `Order` and `Product`. An order can have many products, so we have a One-to-Many relationship between `Order` and `Product`. We want to retrieve the product name and price for all products belonging to a specific order.

Here's how we can do it using combined path expressions in JPQL :

```
// Define the JPQL query string with combined path expression
@NamedQuery(name = Price.GET_ORDER_NAME_AND_PRICE, query = "SELECT p.name, p.price
FROM Price p");
...

// Execute the query and retrieve the result as a list of Object[] arrays
...
public Collection<Object[]> getOrderNameAndPrice() {
    return entityManager.createQuery(Price.GET_ORDER_NAME_AND_PRICE,
    Object[].class).getResultList();
}
```

Done

☒ Day 27/90 ==> [JavaEE: Day 27/90 - EJB \(Enterprise Java Beans\)](#)

Day 28/90

Date : 08-May-2023

JPQL - Java Persistence Query Language (continue)

Constructor Expression

🤖 First, let's define what constructor expressions are. In JPQL, constructor expressions allow you to select specific fields from your entities and map them to a custom class or interface that you define. This can be useful when you want to retrieve only certain data from your database and store it in a custom object.

💡 So, when should you use constructor expressions? You might use them when you want to:

- Retrieve a subset of data from your database
- Map this data to a custom object or interface
- Use this custom object or interface in other parts of your application

👤 Let's use an example to illustrate this. Suppose we have two entities: Employee and Department. Each Employee works for one Department, and each Department has many Employees. We also have a Parking entity, which has a one-to-one relationship with Employee (each Employee has a parking spot).

📄 Here's an example of a named query that uses a constructor expression to select some fields from the Employee and Department entities and map them to a custom object called EmployeeDto:

```
@NamedQuery(  
    name = Employee.findEmployeeswithDeptAndParking,  
    query = "SELECT NEW com.example.EmployeeDto(e.name, d.name, p.location) FROM  
Employee e JOIN e.department d LEFT JOIN e.parking p WHERE e.salary >  
:salaryThreshold ORDER BY e.name"  
)
```

🤖 Let's break this down. The `SELECT` clause specifies that we want to create a new EmployeeDto object, which has three constructor arguments: `e.name`, `d.name`, and `p.location`. These correspond to the name of the Employee, the name of their Department, and the location of their Parking spot.

💡 The `FROM` clause specifies that we want to join the Employee entity with its associated Department, and optionally left join with Parking. We also add a `WHERE` clause to filter out Employees whose salary is below a certain threshold, and an `ORDER BY` clause to sort the results by name.

🎉 Now we can use this named query in our code to retrieve a list of EmployeeDto objects that contain only the data we need!

From Clause - Join

🔍 The `FROM` clause is used in JPQL (Java Persistence Query Language) to specify one or more entity types that will be included in the query.

💛 When you want to include multiple entities in your query, you can use the `JOIN` keyword to join them together based on a relationship between the entities.

👉 Here's an example of using the `FROM` clause with a join:

Let's say we have two entities, `Employee` and `Department`, with a relationship between them where each employee belongs to one department. We want to retrieve a list of all employees along with their department names.

```
SELECT e.name, d.name
FROM Employee e
JOIN e.department d
```

In this query, we are selecting the `name` property from both the `Employee` and `Department` entities. We are joining the `Employee` and `Department` entities together using the `JOIN` keyword, and specifying that we want to join on the `department` property of the `Employee` entity.

🧐 Here's how to interpret the query:

- Start by selecting all `Employee` objects (`e`) and their associated `Department` objects (`d`).
- Then, join the `Employee` and `Department` objects together using the `JOIN` keyword, and specify that you want to join on the `department` property of the `Employee` entity.
- Finally, select the `name` properties from both entities.

🚀 The result of running this query would be a list containing pairs of employee name and department name values.

you can use the `FROM` clause with joins in a JPA named query using the `@NamedQuery` annotation. Here's an example:

Let's say we want to define a named query to retrieve all employees along with their department names, sorted by department name in ascending order. We can define the named query like this:

```
@NamedQuery(
    name = Employee.findEmployeeswithDepartment,
    query = "SELECT e.name, d.name FROM Employee e JOIN e.department d ORDER BY
d.name ASC"
)
```

In this named query, we are selecting the `name` property from both the `Employee` and `Department` entities. We are joining the `Employee` and `Department` entities together using the `JOIN` keyword, and then ordering the results by the `name` property of the `Department` entity in ascending order.

To execute this named query, you can use the `createNamedQuery` method of the entity manager:

```
TypedQuery<Object[]> query =
em.createNamedQuery(Employee.findEmployeeswithDepartment, Object[].class);
List<Object[]> results = query.getResultList();
```

In this example, we are creating a typed query that will return an array of objects containing the employee name and department name for each result. The `getResultList` method is used to execute the query and retrieve the results.

From Clause - Join Maps

So, JPQL stands for Java Persistence Query Language and it's used to write queries against entities and their persistent state. The "from clause" is one of the most important clauses in a JPQL query, as it specifies the entity or entities to be queried.

Now, when we talk about "join maps," what we're really referring to is a join between two entities where one of the entities has a map attribute. This can be a bit tricky to understand at first, but let me give you an example.

Let's say we have two entities: Customer and Order. Each customer can have multiple orders, so we represent this relationship using a map in the Customer entity:

```
@Entity
public class Customer {
    @Id
    private Long id;

    @OneToMany(mappedBy = "customer")
    private Map<Long, Order> orders;

    // getters and setters
}
```

In this example, the "orders" attribute is a Map where the keys are order IDs and the values are Order objects.

Now, let's say we want to write a JPQL query that joins the Customer and Order entities on their respective IDs. We can do this using the following syntax:

```
SELECT c, o FROM Customer c JOIN c.orders o WHERE c.id = :customerId AND o.id = :orderId
```

In this example, we're selecting both the Customer and Order entities (hence the "c, o" in the SELECT clause), joining them on the "orders" attribute of the Customer entity, and filtering the results based on the IDs of both entities.

We can also use `@NamedQuery` annotation to define this query in our entity class like this:

```
@Entity
@NamedQuery(
    name = "Customer.findOrder",
    query = "SELECT c, o FROM Customer c JOIN c.orders o WHERE c.id = :customerId AND o.id = :orderId"
)
public class Customer {
```

```

@Id
private Long id;

@OneToMany(mappedBy = "customer")
private Map<Long, Order> orders;

// getters and setters
}

```

This way we can easily use this query in our code by calling `entityManager.createNamedQuery("Customer.findOrder")`.

From Clause - Fetch Join

let's say we have another entity called "Department" that is related to the "User" entity through a Many-to-One relationship. We can use a `FETCH JOIN` to retrieve both entities together in a single query:

```
SELECT u FROM User u JOIN FETCH u.department
```

In this query, we're using a `JOIN` clause to join the "User" entity with its related "Department" entity, and we're using the `FETCH` keyword to indicate that we want to retrieve the "Department" entity eagerly (i.e., load it in memory along with the "User" entity).

To use this JPQL query in a JPA application, we can define a named query using the `@NamedQuery` annotation on our entity class:

```

@Entity
@NamedQuery(
    name = "User.findAllWithDepartments",
    query = "SELECT u FROM User u JOIN FETCH u.department"
)
public class User {
    // ...
}

```

In this code snippet, we're defining a named query called `User.findAllWithDepartments` that corresponds to the JPQL query we wrote earlier. We can then use the `EntityManager` to execute this named query and retrieve the results:

```

TypedQuery<User> query = em.createNamedQuery("User.findAllWithDepartments",
User.class);
List<User> users = query.getResultList();

```

In this code snippet, we're creating a `TypedQuery` object using the named query we defined earlier, and we're specifying that the result type should be a list of `User` entities. We can then call the `getResultList()` method to execute the query and retrieve the results.

Where Clause

The `WHERE` clause is used to specify a condition that must be met for each record returned by the query.

For example, imagine you have a `Person` entity that has a `name` attribute. You could use JPQL and the `WHERE` clause to find all `Person` entities with the name "John":

```
SELECT p FROM Person p WHERE p.name = 'John'
```

Next, let's talk about passing parameters to the entity manager. There are two ways to pass parameters: positional parameters and named parameters. A positional parameter is represented by a question mark (?) in the query, and values are passed in the order they appear in the query. A named parameter is represented by a colon (:) followed by a name, and values are passed using the `setParameter()` method.

Here's an example of a JPQL query using a named parameter:

```
@NamedQuery(  
    name="findPersonByName",  
    query="SELECT p FROM Person p WHERE p.name = :name"  
)
```

In this example, the named parameter `name` is used in the `WHERE` clause to find a `Person` with a specific name. To pass a value to this query using the entity manager, you would use the `setParameter()` method like so:

```
String name = "John";  
TypedQuery<Person> query = em.createNamedQuery("findPersonByName", Person.class);  
query.setParameter("name", name);  
List<Person> people = query.getResultList();
```

In this code, we're setting the `name` parameter to "John" and then executing the query using the entity manager. The results are returned as a list of `Person` entities.

Where Clause - Between Operator

The "between operator" is used in a where clause to specify a range of values that data should fall within. For example, if we have a database of products with a "price" field, we could use the between operator to retrieve all products with prices between \$10 and \$20.

Here's an example of how you can use `@NamedQuery` and entity manager to write a JPQL query with a where clause and between operator:

Let's say we have an entity called "Product" with fields `id`, "name", and "price". We want to retrieve all products with prices between \$10 and \$20. Here's how we can do it:

1. First, we define our named query using the `@NamedQuery` annotation in the Product entity class:


```

@Entity
@NamedQuery(name = "Product.findInRange", query = "SELECT p FROM Product p WHERE
p.price BETWEEN :minPrice AND :maxPrice")
public class Product {
    // entity fields and methods go here
}

```

In this named query, we're selecting all instances of the Product class (denoted by "p") where the price field is between two parameters `:minPrice` and `:maxPrice`.

1. Next, we can use the EntityManager class to execute this named query in our code:

```

EntityManager em = // get entity manager instance
TypedQuery<Product> query = em.createNamedQuery("Product.findInRange",
Product.class);
query.setParameter("minPrice", 10);
query.setParameter("maxPrice", 20);
List<Product> results = query.getResultList();

```

In this code, we're creating a TypedQuery object that will execute our named query "Product.findInRange" and return a list of Product objects. We then set the values of the `:minPrice` and `:maxPrice` parameters to 10 and 20, respectively. Finally, we call `getResultList()` to retrieve all products that match our query.

Where Clause - Like Operator

In JPQL, the "like operator" is used in a where clause to match patterns in string values. This operator is useful when you want to retrieve data that matches specific text patterns.

Here's an example of how you can use `@NamedQuery` and entity manager to write a JPQL query with a where clause and like operator:

Let's say we have an entity called "Customer" with fields "id", "firstName", and "lastName". We want to retrieve all customers whose last name starts with "Smi". Here's how we can do it:

1. First, we define our named query using the `@NamedQuery` annotation in the Customer entity class:

```

@Entity
@NamedQuery(name = "Customer.findByLastNamePattern", query = "SELECT c FROM Customer
c WHERE c.lastName LIKE :pattern")
public class Customer {
    // entity fields and methods go here
}

```

In this named query, we're selecting all instances of the Customer class (denoted by "c") where the lastName field matches a pattern specified by the `:pattern` parameter.

1. Next, we can use the EntityManager class to execute this named query in our code:

```
EntityManager em = // get entity manager instance
TypedQuery<Customer> query = em.createNamedQuery("Customer.findByLastNamePattern",
Customer.class);
query.setParameter("pattern", "Smi%");
List<Customer> results = query.getResultList();
```

In this code, we're creating a TypedQuery object that will execute our named query "Customer.findByLastNamePattern" and return a list of Customer objects. We then set the value of the :pattern parameter to Smi%, which will match any last name that starts with "Smi". Finally, we call getResultList() to retrieve all customers that match our query.

Done

☒ Day 28/90 ==> [JavaEE: Day 28/90 - JPQL \(Java Persistence Query Language\)](#)