6 : Object Relational Mapping with the Java Persistence API

IT 4206 – Enterprise Application Development

Level II - Semester 4

Overview

- Identify how to map objects and relations using Java persistence APIs.
- Identify how to create queries using Java Persistence Query Language.

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
  - Understand object relational mapping concepts.
  - Describe entity class and annotations.
  - Describe how to use EntityManager in an EJB.
  - Describe a persistence context XML descriptor.
  - Describe requirements for entity classes.
  - Describe entity fields and properties.
  - Describe the **EntityManager** interface and key methods.

List of sub topics

1.1 Object relational mapping concepts
1.2 Entity class and annotations
1.3 Use EntityManager in an EJB
1.4 Persisting Data
1.5 Creating Queries
1.6 Relationship between entities
1.7 Tuning the performance of loading relationship data

1.1 Object relational mapping concepts

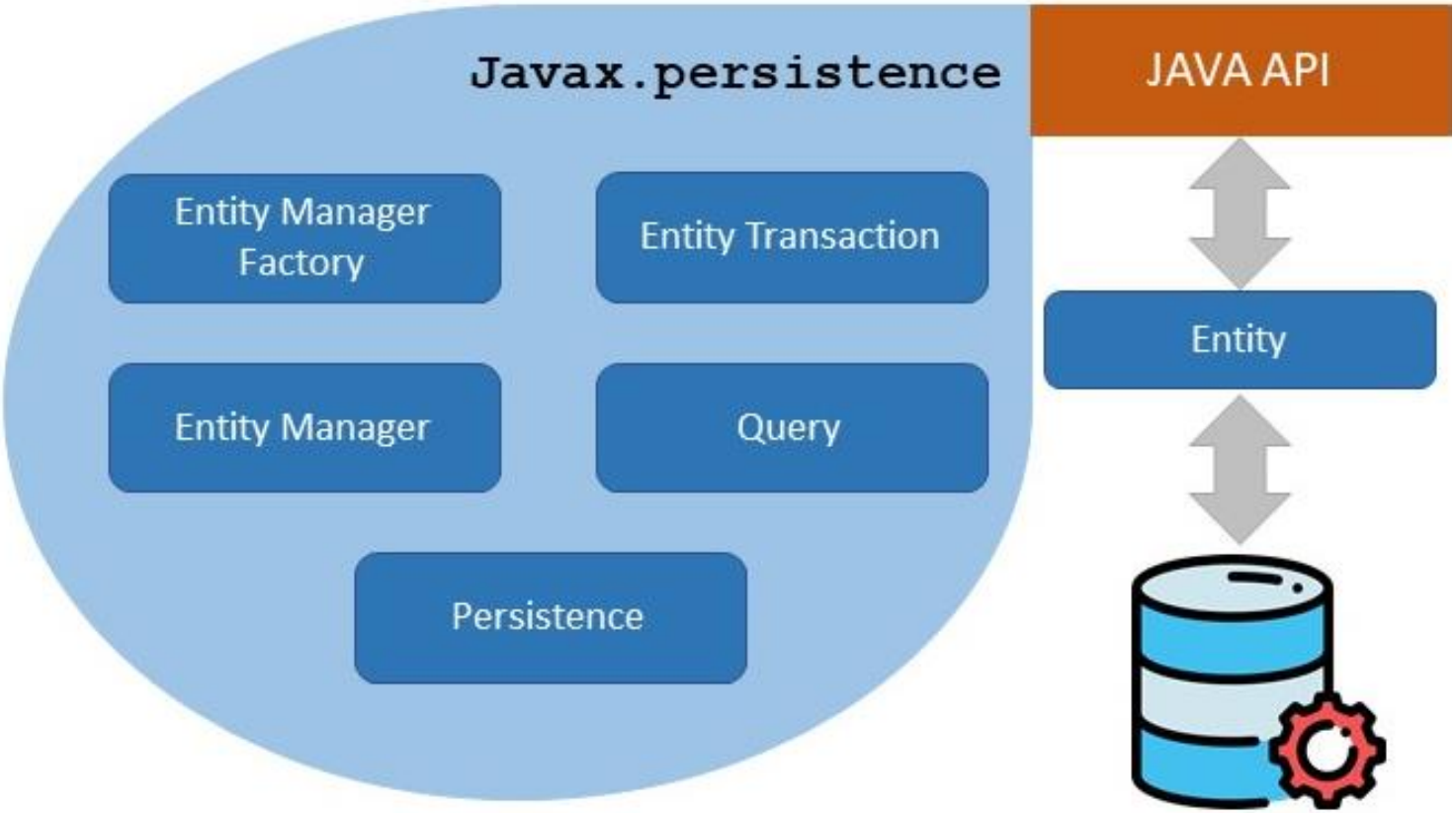
- Java objects and database tables use different data types.
- **String** in Java and **Varchar** in a database, to store business data.
- This can cause differences between the object model and the relational model(Impedance Mismatch).
- The technique to automate bridging the impedance mismatch is known as Object Relational Mapping (ORM).

- ORM software uses metadata to describe mapping between the classes defined in an application and the schema of a database table.
- A fully implemented ORM provides optimization techniques, caching, database portability, query language in addition to object persistence.
- The three key concepts related to the Java Persistence API;
  - entities
  - persistence units
  - persistence context

What is JPA

- Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.
- To reduce the burden of writing codes for relational object management, a programmer follows the ‘JPA Provider’ framework, which allows easy interaction with database instance.

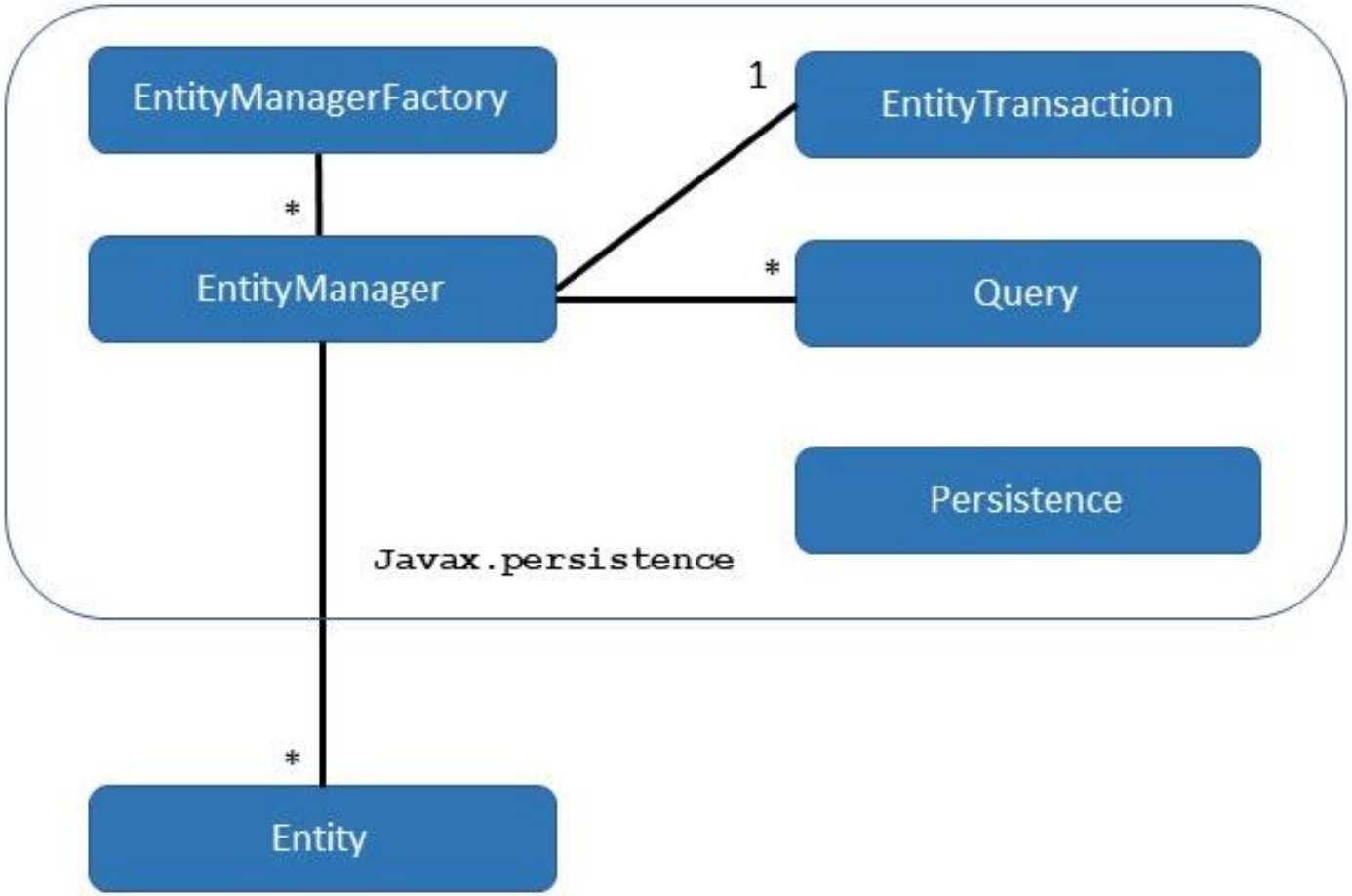
Architecture of Java Persistent API



Units of the Architecture

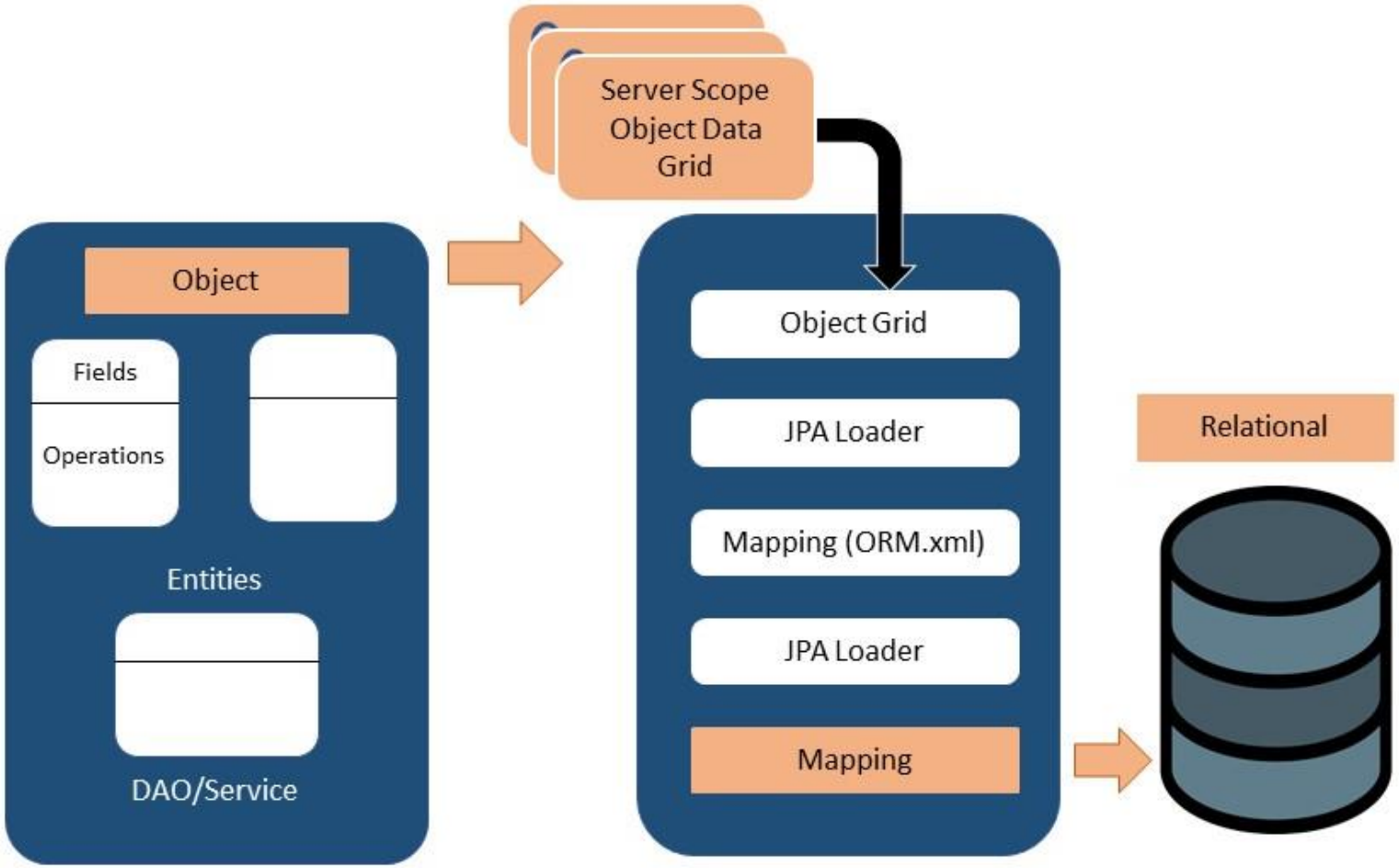
- EntityManagerFactory - This is a factory class of EntityManager. It creates and manages multiple EntityManager instances.
- EntityManager - It is an Interface, it manages the persistence operations on objects. It works like factory for Query instance.
- Entity - Entities are the persistence objects, stores as records in the database.
- EntityTransaction - It has one-to-one relationship with EntityManager. For each EntityManager, operations are maintained by EntityTransaction class.
- Persistence - This class contain static methods to obtain EntityManagerFactory instance.
- Query - This interface is implemented by each JPA vendor to obtain relational objects that meet the criteria.

JPA Class Relationship



- The relationship between EntityManagerFactory and EntityManager is one-to-many. It is a factory class to EntityManager instances.
- The relationship between EntityManager and EntityTransaction is one-to-one. For each EntityManager operation, there is an EntityTransaction instance.
- The relationship between EntityManager and Query is one-to-many. Many number of queries can execute using one EntityManager instance.
- The relationship between EntityManager and Entity is one-to-many. One EntityManager instance can manage multiple Entities.

Object Relational Mapping Architecture



- The above architecture explains how object data is stored into relational database in three phases.

Phase 1

- The first phase, named as the **Object data** phase contains POJO classes, service interfaces and classes.
- It is the main business component layer, which has business logic operations and attributes.

Phase 2

- The second phase named as **mapping** or **persistence** phase which contains JPA provider, mapping file(ORM.xml), JPA Loader, and Object Grid.

Phase 3

- The third phase is the Relational data phase.
- It contains the relational data which is logically connected to the business component.

Mapping.xml

- The mapping.xml file is to instruct the JPA vendor for mapping the Entity classes with database tables.

Example

- Consider the example of Employee entity which contains four attributes as eid, ename, salary, and deg.
- The POJO class of Employee entity named **Employee.java** is as follows:

```
public class Employee {

   private int eid;
   private String ename;
   private double salary;
   private String deg;

   public Employee(int eid, String ename, double salary, String deg) {
      super( );
      this.eid = eid;
      this.ename = ename;
      this.salary = salary;
      this.deg = deg;
   }

   public Employee( ) {
      super();
   }

   public int getEid( ) {
      return eid;
   }

   public void setEid(int eid) {
      this.eid = eid;
   }

   public String getEname( ) {
      return ename;
   }

   public void setEname(String ename) {
      this.ename = ename;
   }
```

```
    public void setEname(String ename) {
        this.ename = ename;
    }

    public double getSalary( ) {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public String getDeg( ) {
        return deg;
    }

    public void setDeg(String deg) {
        this.deg = deg;
    }
}
```

- The above code is the Employee entity POJO class. It contain four attributes.
- Consider these attributes are the table fields in the database and eid is the primary key of this table.
- The mapping file named **mapping.xml** is as follows:

mapping.xml

```
<? xml version="1.0" encoding="UTF-8" ?>

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
   http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
   version="1.0">

   <description> XML Mapping file</description>

   <entity class="Employee">
      <table name="EMPLOYEETABLE"/>
      <attributes>

         <id name="eid">
            <generated-value strategy="TABLE"/>
         </id>

         <basic name="ename">
            <column name="EMP_NAME" length="100"/>
         </basic>

         <basic name="salary">
         </basic>

         <basic name="deg">
         </basic>

      </attributes>
   </entity>

</entity-mappings>
```

- The above script for mapping the entity class with database table. In this file
- <entity-mappings> : tag defines the schema definition to allow entity tags into xml file.
- <description> : tag defines description about application.
- <entity> : tag defines the entity class which you want to convert into table in a database. Attribute class defines the POJO entity class name.
- <table> : tag defines the table name. If you want to keep class name as table name then this tag is not necessary.
- <attributes> : tag defines the attributes (fields in a table).
- <id> : tag defines the primary key of the table. The <generated-value> tag defines how to assign the primary key value such as Automatic, Manual, or taken from Sequence.
- <basic> : tag is used for defining remaining attributes for table.
- <column-name> : tag is used to define user defined table field name.

1.2 Entity class and annotations

- An entity class is mapped to a table in a relational database.
- Each instance of an entity class has a primary key field.
- The primary key field is used to map an entity instance to a row in a database table.
- All non-transient attributes map to fields in a database table.

Entities

- An entity is a lightweight persistence domain object.
- Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table.
- The primary programming artifact of an entity is the entity class, although entities can use helper classes.
- The persistent state of an entity is represented through either persistent fields or persistent properties.
- These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

Default Entity to Table Mapping

ENTITY	TABLE
Entity Class	Table Name
Attributes of entity class	Columns in a database table
Entity instance	Record or row in a database table

Annotations

- Annotations are used to decorate the Java classes, fields, and methods with metadata for mapping, configuration, queries, validation, and so on.
- Some of the annotations can be listed as follows;
  - **@Entity**
  - **@Table**
  - **@Column**
  - **@Temporal**
  - **@Transient**
  - **@Id**

Requirements for Entity Classes

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.
- 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. Clients must access the entity's state through accessor or business methods.

Persistent Fields and Properties in Entity Class

The persistent state of an entity can be accessed through either the entity's instance variables or properties. The fields or properties must be of the following Java language types:

- Java primitive types
- java.lang.String
- Other serializable types, including:
  - Wrappers of Java primitive types
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - java.sql.Time
  - java.sql.TimeStamp
  - User-defined serializable types
  - byte[]
  - Byte[]
  - char[]
  - Character[]
- Enumerated types
- Other entities and/or collections of entities
- Embeddable classes

- Entities may use persistent fields, persistent properties, or a combination of both.
- If the mapping annotations are applied to the entity's instance variables, the entity uses persistent fields.
- If the mapping annotations are applied to the entity's getter methods for JavaBeans-style properties, the entity uses persistent properties.

1.3 Use EntityManager in an EJB

- The **EntityManager** API is defined to perform persistence operation.
- Obtains the reference to an entity and performs the actual CRUD operations on the database.
- Works within a set of managed entity instances.
- Entity instances are known as the entity manager's persistence context.

Manage Entity

- Entities are managed by the entity manager, which is represented by javax.persistence.
- EntityManager instances. Each EntityManager instance is associated with a persistence context: a set of managed entity instances that exist in a particular data store.
- A persistence context defines the scope under which particular entity instances are created, persisted, and removed.
- The EntityManager interface defines the methods that are used to interact with the persistence context.

The EntityManager Interface

- The EntityManager API creates and removes persistent entity instances, finds entities by the entity's primary key, and allows queries to be run on entities.

Container-Managed Entity Managers

- With a container-managed entity manager, an EntityManager instance's persistence context is automatically propagated by the container to all application components that use the EntityManager instance within a single Java Transaction API (JTA) transaction.

Application-Managed Entity Managers

- With an application-managed entity manager, on the other hand, the persistence context is not propagated to application components, and the lifecycle of EntityManager instances is managed by the application.

1.4 Persisting Data

- A Persistence unit describes configuration settings related to a data source, transactions, concrete classes, and object-relational mapping.
- Configured in a **persistence.xml** file.
- Every application that uses persistence has at least one persistence unit.
- A Persistence unit contains information about the persistence unit name, data source, and transactions type.

persistence.xml

- This module plays a crucial role in the concept of JPA. This xml file used to register the database and specify the entity class. This file will configure the database.

Example of a persistence.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="Eclipselink_JPA" transaction-type="RESOURCE_LOCAL">
    <class>com.my.eclipselink.entity.Employee</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/jpadb"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value="root"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="eclipselink.logging.level" value="FINE"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

- In the above xml, <persistence-unit> tag is defined with specific name for JPA persistence.
- The <class> tag defines entity class with package name.
- The <properties> tag defines all the properties, and <property> tag defines each property such as database registration, URL specification, username, and password.

- The Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications. Java Persistence consists of four areas:
  - The Java Persistence API
  - The query language
  - The Java Persistence Criteria API
  - Object/relational mapping metadata

Persistent Fields

- If the entity class uses persistent fields, the Persistence runtime accesses entity-class instance variables directly.
- All fields not annotated javax.persistence.Transient or not marked as Java transient will be persisted to the data store.
- The object/relational mapping annotations must be applied to the instance variables.

Persistent Properties

- If the entity uses persistent properties, the entity must follow the method conventions of JavaBeans components.
- JavaBeans-style properties use getter and setter methods that are typically named after the entity class's instance variable names.

Collections in Entity Fields and Properties

- Collection-valued persistent fields and properties must use the supported Java collection interfaces regardless of whether the entity uses persistent fields or properties.
- The following collection interfaces may be used:
  - java.util.Collection
  - java.util.Set
  - java.util.List
  - java.util.Map

Validation fields and properties

- The Java API for JavaBeans Validation (Bean Validation) provides a mechanism for validating application data.
- Bean Validation is integrated into the Java EE containers, allowing the same validation logic to be used in any of the tiers of an enterprise application.

Primary keys in Entities

- Each entity has a unique object identifier. A customer entity, for example, might be identified by a customer number.
- The unique identifier, or **primary key**, enables clients to locate a particular entity instance.
- Every entity must have a primary key.
- An entity may have either a simple or a composite primary key.

Multiplicity in Entity Relationships

- One-to-one: Each entity instance is related to a single instance of another entity.
- One-to-many: An entity instance can be related to multiple instances of the other entities.
- Many-to-one: Multiple instances of an entity can be related to a single instance of the other entity. This multiplicity is the opposite of a one-to-many relationship.
- Many-to-many: The entity instances can be related to multiple instances of each other.

Cascade Operations and Relationships

- Entities that use relationships often have dependencies on the existence of the other entity in the relationship.

Entity Inheritance

- Entities support class inheritance, polymorphic associations, and polymorphic queries. Entity classes can extend non-entity classes, and non-entity classes can extend entity classes. Entity classes can be both abstract and concrete.

Abstract Entities

- An abstract class may be declared an entity by decorating the class with @Entity. Abstract entities are like concrete entities but cannot be instantiated.

- Abstract entities can be queried just like concrete entities. If an abstract entity is the target of a query, the query operates on all the concrete subclasses of the abstract entity.

1.5 Creating Queries

Creating Entity Class

- Similar to a standard POJO class
- Manage by the **EntityManager**
- Each instance variable should be accessed through the use of getter and setter methods.
- Class must at least have one constructor that has no arguments and can still have other constructors that take arguments.

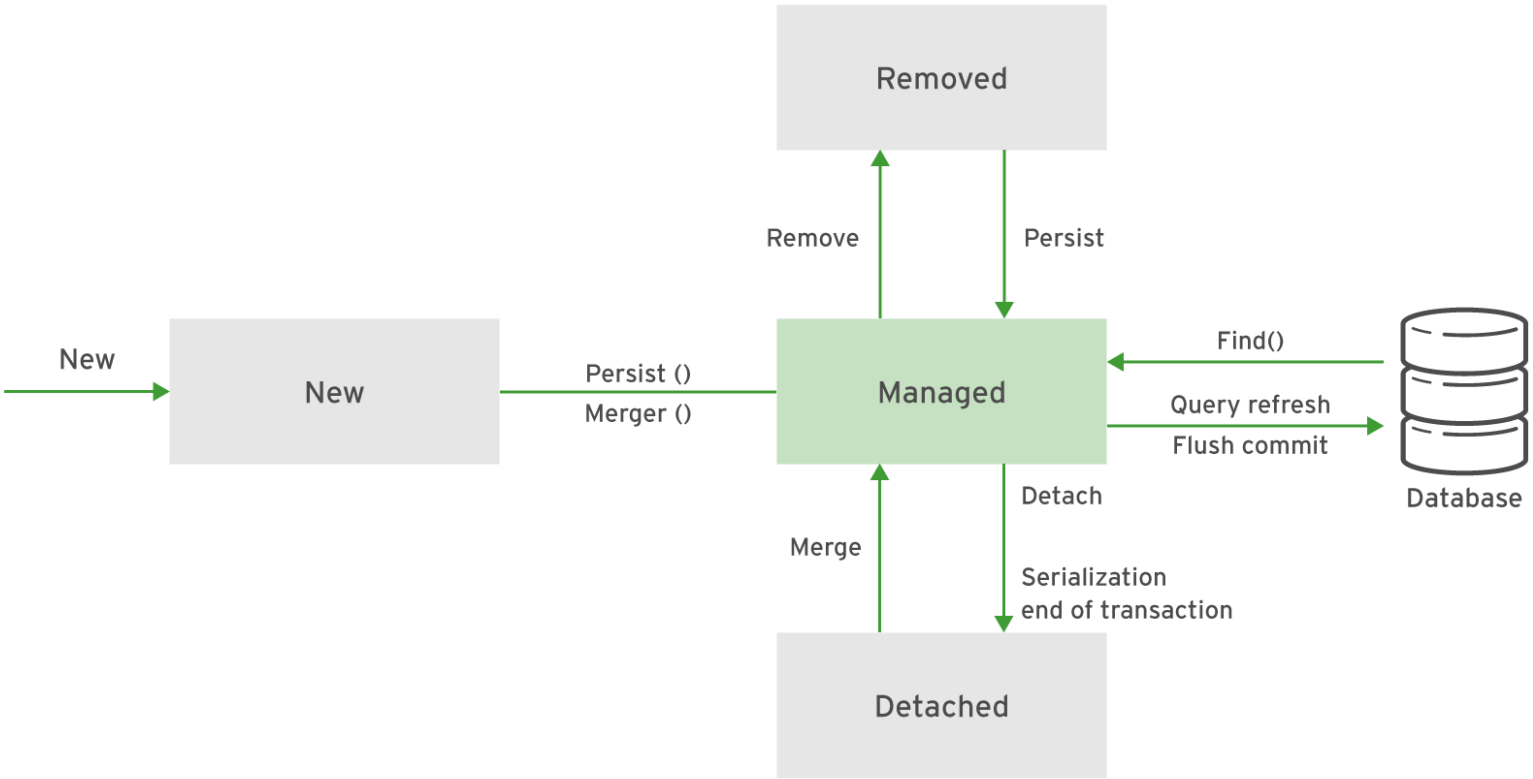
Entity field and properties

- The state is accessed by the provider is known as the access mode.
- There are two access modes: field-based access and property-based access.
  - **Field-based Access**
  - **Property-based Access**

Entity States

- An entity can exist in one of four states during its lifetime. These four states are:
  - **New State**: An entity instance does not have a persistent identity and is not yet associated with the persistence context.
  - **Managed State**: An entity instance with a persistent identity and that is associated with a persistence context is in a managed or persistent state.
  - **Removed State**: A persistent entity can be removed from the database table in many ways.
  - **Detached State**: An entity has a persistent entity identity but is not associated with the persistence context.

JPA Components Relationship



1.6 Relationship between entities

- To represent these relationships, databases use what is called a foreign key.
- When representing database tables in Java EE, developers use entity beans, one for each table.
- To create a relationship between two entities, class-level variables are used to represent an instance of one entity as an attribute of another entity.

EntityManager Key Methods

- persist()
- find()
- contains()
- merge()
- remove()
- clear()
- refresh()

Creating Queries

- Java Persistence Query Language (JPQL) is a platform-independent query language
- JPQL is similar to SQL in syntax
- JPQL queries are expressed in terms of Java entities rather than database tables and columns
- JPQL supports the **SELECT**, **UPDATE**, and **DELETE** statements
- Queries with the **WHERE** clause can be created with named parameters in JPQL

Standard JPA Relationship Annotations

ANNOTATION	DESCRIPTION
@OneToOne	Defines an entity relationship as a single value, where one row in table X is related to a single row in table Y, and vice versa.
@OneToMany	Defines an entity relationship as multi-valued, where one row in table X can be related to one or many rows in table Y.
@ManyToOne	Defines an entity relationship as multi-valued, where many rows in table X can be related to a single row in table Y.
@ManyToMany	Defines an entity relationship as multi-valued, where many rows in table X can be related to a one or may rows in table Y.
@JoinColumn	Defines the column that JPA uses as the foreign key.

1.7 Tuning the performance of loading relationship data

- JPA annotations requires extra processing to populate the relational data.
- Inhibits application performance with a large data set.
- To improve entity loading performance, JPA provides functionality called **lazy loading**, or **lazy fetching**.
- With lazy loading, entities can map relationships, but only load those relationships when needed.
- Behavior of whether or not JPA loads related entities is called the **fetch** type. There are two fetch types that can be used, **lazy** or **eager**.