

JPA With Hibernate

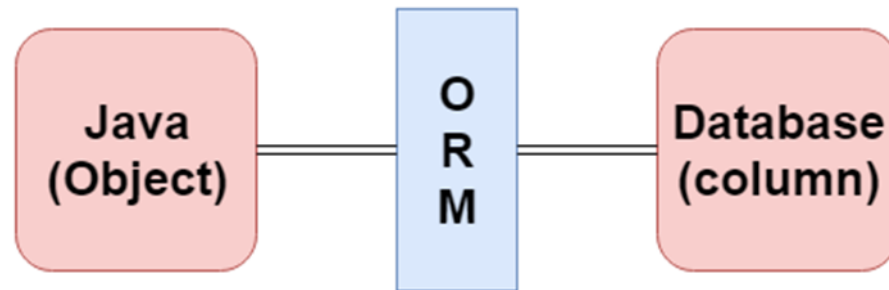
JPA

- ▶ A JPA (Java Persistence API) is a specification of Java which is used to access, manage, and persist data between Java object and relational database. It is considered as a standard approach for Object Relational Mapping.
- ▶ JPA acts as a bridge between object-oriented domain models and relational database systems.
- ▶ As JPA is just a specification, it doesn't perform any operation by itself.
- ▶ It requires an implementation. So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence.

•
Object Relational Mapping (ORM) is a functionality which is used to develop and maintain a relationship between an object and relational database by mapping an object state to database column.

- It is capable to handle various database operations easily such as inserting, updating, deleting etc.

Continue....



What is Hibernate?

- ▶ A Hibernate is a Java framework which is used to store the Java objects in the relational database system. It is an open-source, lightweight, ORM (Object Relational Mapping) tool.
- ▶ Hibernate is an implementation of JPA. So, it follows the common standards provided by the JPA.

JPA	Hibernate
Java Persistence API (JPA) defines the management of relational data in the Java applications.	Hibernate is an Object-Relational Mapping (ORM) tool which is used to save the state of Java object into the database.
It is just a specification. Various ORM tools implement it for data persistence.	It is one of the most frequently used JPA implementation.
It is defined in javax.persistence package.	It is defined in org.hibernate package.
The EntityManagerFactory interface is used to interact with the entity manager factory for the persistence unit. Thus, it provides an entity manager.	It uses SessionFactory interface to create Session instances.
It uses EntityManager interface to create, read, and delete operations for instances of mapped entity classes. This interface interacts with the persistence context.	It uses Session interface to create, read, and delete operations for instances of mapped entity classes. It behaves as a runtime interface between a Java application and Hibernate.
It uses Java Persistence Query Language (JPQL) as an object-oriented query language to perform database operations.	It uses Hibernate Query Language (HQL) as an object-oriented query language to perform database operations.

ORM Frameworks

- ▶ Following are the various frameworks that function on ORM mechanism: -
 - Hibernate
 - TopLink
 - ORMLite
 - iBATIS
 - JPOX

JPA Entity Introduction

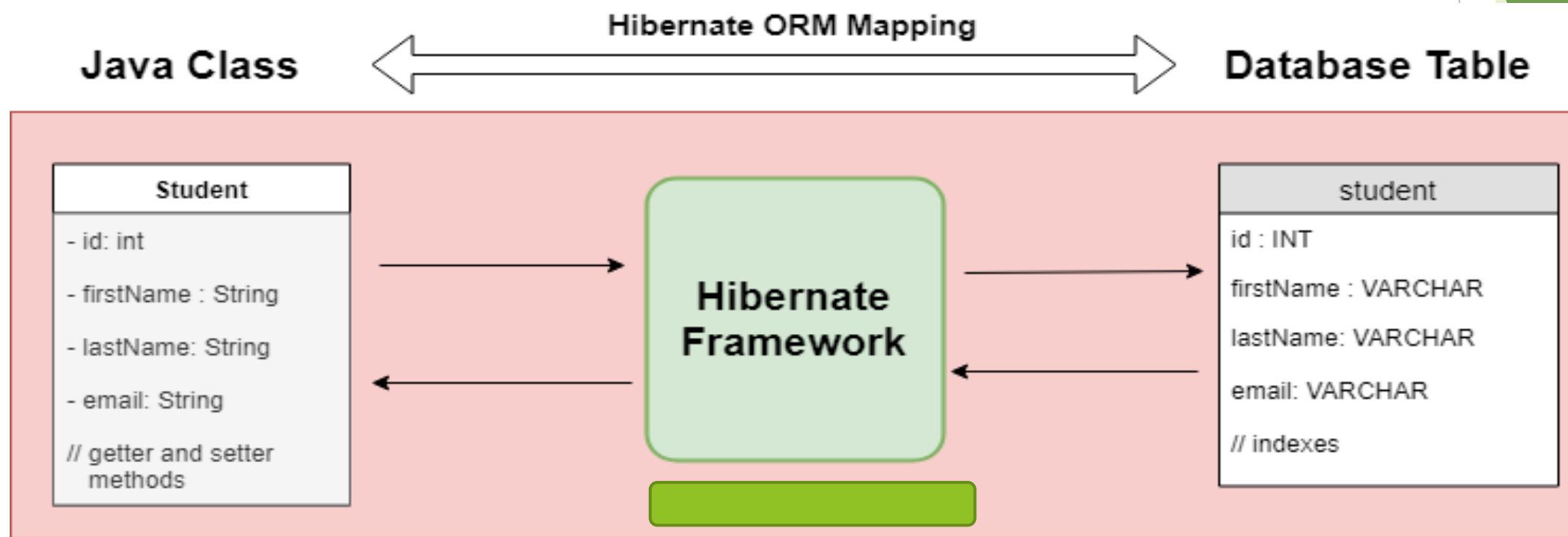
► What is JPA Entity?

- Entities in JPA are nothing but POJOs representing data that can be persisted in the database. An entity represents a table stored in a database. Every instance of an entity represents a row in the table.

► Entity Properties

- These are the properties of an entity that an object must have: -
 - **Persistability** - An object is called persistent if it is stored in the database and can be accessed anytime.
 - **Persistent Identity** - In Java, each entity is unique and represents as an object identity. Similarly, when the object identity is stored in a database then it is represented as persistence identity. This object identity is equivalent to primary key in database.
 - **Transactionality** - Entity can perform various operations such as create, delete, update. Each operation makes some changes in the database. It ensures that whatever changes made in the database either be succeed or failed atomically.
 - **Granularity** - Entities should not be primitives, primitive wrappers or built-in objects with single dimensional state.

JPA Creating an Entity



Annotation

1) This annotation specifies that the class is an entity:

@Entity

```
public class Student {  
}
```

2) The entity name defaults to the name of the class. We can change its name using the name element.

@Entity(name="student")

```
public class Student {  
}
```

3) This annotation specifies the table in the database with which this entity is mapped.

@Entity

@Table(name = "student")

```
public class Student {  
}
```

- 4) @Id: This annotation specifies the primary key of the entity:

```
@Entity
```

```
@Table(name = "stu")
```

```
public class Student {
```

```
    @Id
```

```
    private int eid;}
```

- 5) @GeneratedValue: This annotation specifies the generation strategies for the values of primary keys:

```
@Entity
```

```
@Table(name = "student")
```

```
public class Student {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    private int eid; }
```

Note: We can choose from four id generation strategies with the strategy element. The value can be AUTO, TABLE, SEQUENCE, or IDENTITY.

Continue...

@Column

The @Column annotation is used to specify the mapping between a basic entity attribute and the database table column.:

@Entity

@Table(name = "student")

```
public class Student {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "id")
```

```
    private int id;
```

```
    @Column(name = "first_name", length=50, nullable=false, unique=false)
```

```
    private String firstName;
```

```
    // getters and setters
```

```
}
```

JPA Entity Manager

- ▶ Following are some of the important roles of an entity manager: -
- ▶ The entity manager implements the API and encapsulates all of them within a single interface.
- ▶ Entity manager is used to read, delete and write an entity.
- ▶ An object referenced by an entity is managed by entity manager.
- ▶ Steps to persist an entity object.
- ▶ 1) Creating an entity manager factory object
- ▶ The EntityManagerFactory interface present in java.persistence package is used to provide an entity manager.
- ▶ EntityManagerFactory
`emf=Persistence.createEntityManagerFactory("Student_details");`

- Persistence - The Persistence is a bootstrap class which is used to obtain an EntityManagerFactory interface.
- createEntityManagerFactory() method - The role of this method is to create and return an EntityManagerFactory for the named persistence unit. Thus, this method contains the name of persistence unit passed in the Persistence.xml file.

- 2) Obtaining an entity manager from factory.

EntityManager em=emf.createEntityManager();

EntityManager - An EntityManager is an interface

createEntityManager() method - It creates new application-managed EntityManager

- 3) Initializing an entity manager.

em.getTransaction().begin();

getTransaction() method - This method returns the resource-level EntityTransaction object.

begin() method - This method is used to start the transaction.

4) Persisting a data into relational database.

```
em.persist(s1);
```

`persist()` - This method is used to make an instance managed and persistent. An entity instance is passed within this method.

5) Closing the transaction

```
em.getTransaction().commit();
```

6) Releasing the factory resources.

```
emf.close();
```

```
em.close();
```

Entity Operations

- ▶ Inserting an Entity: In JPA, we can easily insert data into database through entities. The EntityManager provides persist() method to insert records. Ex: `em.persist(s1);`
- ▶ Finding an Entity: To find an entity, EntityManager interface provides find() method that searches an element on the basis of primary key. Ex: `StudentEntity s=em.find(StudentEntity.class,101);`
- ▶ Updating an Entity: JPA allows us to change the records in database by updating an entity.
- ▶ Deleting an Entity: To delete a record from database, EntityManager interface provides remove() method. The remove() method uses primary key to delete the particular record.

Ex: `em.getTransaction().begin();`
`StudentEntity s=em.find(StudentEntity.class,102);`
`em.remove(s);`
`em.getTransaction().commit();`

Collection Mapping

- ▶ A Collection is a java framework that groups multiple objects into a single unit. It is used to store, retrieve and manipulate the aggregate data.
- ▶ Collection Types
- ▶ On the basis of requirement, we can use different type of collections to persist the objects.
- ▶ List
- ▶ Set
- ▶ Map
- ▶ A List is an interface which is used to insert and delete elements on the basis of index. It can be used when there is a requirement of retrieving elements in a user-defined order.

```
private List<Address> address=new ArrayList<Address>();
```

- ▶ This example contains the following steps: -
 - Create an entity class Employee.java under com.javatpoint.jpa package that contains employee id, name and embedded object (employee Address). The annotation @ElementCollection represents the embedded object.


```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int eid;
    private String ename;
    @ElementCollection
        private List<Address> address=new
    ArrayList<Address>();
    getter-setter methods.....
}
```

@Embeddable //The annotation
@Embeddable represents the
embeddable object.

```
public class Address {
    private int epincode;
    private String ecity;
    private String estate;

    getter -setter methods

}
```

JPA Type of Mapping

One-To-One Mapping

One-To-Many Mapping

Many-To-One Mapping

Many-To-Many Mapping

- 1) **The One-To-One mapping** represents a single-valued association where an instance of one entity is associated with an instance of another entity. @OneToOne
- 2) **One-To-Many Mapping** comes into the category of collection-valued association where an entity is associated with a collection of other entities.

@OneToMany

- ▶ The Many-To-One mapping represents a single-valued association where a collection of entities can be associated with the similar entity. @ManyToOne
- ▶ **The Many-To-Many mapping** represents a collection-valued association where any number of entities can be associated with a collection of other entities.
- ▶ Note: Programs are in eclipse folder .

The JPQL (Java Persistence Query Language)

- ▶ The **JPQL (Java Persistence Query Language)** is an object-oriented query language which is used to perform database operations on persistent entities. Instead of database table, JPQL uses entity object model to operate the SQL queries.
- ▶ Here, the role of JPA is to transform JPQL into SQL. Thus, it provides an easy platform for developers to handle SQL tasks.
- ▶ **JPQL Features**
- ▶ It is a platform-independent query language.
- ▶ It is simple and robust.
- ▶ It can be used with any type of database such as MySQL, Oracle.
- ▶ JPQL queries can be declared statically into metadata or can also be dynamically built in code.

Continuous...

- ▶ Creating Queries in JPQL
- ▶ JPQL provides two methods that can be used to access database records. These methods are: -
- ▶ Query `createQuery(String name)` - The `createQuery()` method of `EntityManager` interface is used to create an instance of `Query` interface for executing JPQL statement.
- ▶ Query `createNamedQuery(String name)` - The `createNamedQuery()` method of `EntityManager` interface is used to create an instance of `Query` interface for executing named queries.
- ▶ `@NamedQuery(name = "find name" , query = "Select s from StudentEntity s")`

Continuous....

- ▶ This method is used to create static queries that can be defined in entity class.
- ▶ Now, we can control the execution of query by the following Query interface methods:
 -
- ▶ `int executeUpdate()` - This method executes the update and delete operation.
- ▶ `int getFirstResult()` - This method returns the first positioned result the query object was set to retrieve.
- ▶ `int getMaxResults()` - This method returns the maximum number of results the query object was set to retrieve.
- ▶ `java.util.List getResultList()` - This method returns the list of results as an untyped list.
- ▶ `Query setFirstResult(int startPosition)` - This method assigns the position of first result to retrieve.
- ▶ `Query setMaxResults(int maxResult)` - This method assigns the maximum numbers of result to retrieve.

JPA JPQL Basic Operations

- Here we see in the example how to use createQuery and NamedQuery.

```
@Entity
```

```
@Table(name="student")
```

```
public class StudentEntity {
```

```
    @Id
```

```
    private int s_id;
```

```
    private String s_name;
```

```
    private int s_age;
```

```
}
```

```
public class FetchColumn {
```

```
    public static void main( String args[]) {
```

```
        EntityManagerFactory emf =  
        Persistence.createEntityManagerFactory( "Student_details" );
```

```
        EntityManager em = emf.createEntityManager();
```

```
        em.getTransaction().begin( );
```

```
        Query query = em.createQuery("Select s.s_name from StudentEntity  
s");
```

```
@SuppressWarnings("unchecked")
```

```
        List<String> list =query.getResultList();
```

```
        System.out.println("Student Name :");
```

```
        for(String s:list) {            System.out.println(s); } }
```

Named Query

@Entity

@Table(name="student")

@NamedQuery(name =
"find name" , query = "Select
s from StudentEntity s")

public class
StudentEntity {

@Id

private int s_id;

private String s_name;

```
public class FetchColumn {  
    public static void main( String args[]) {  
        EntityManagerFactory emf =  
        Persistence.createEntityManagerFactory( "Student_details" );  
        EntityManager em = emf.createEntityManager();  
        em.getTransaction().begin( );  
        Query query = em.createNamedQuery("find name");  
        @SuppressWarnings("unchecked")  
        List<StudentEntity> list =query.getResultList();  
        System.out.println("Student Name :");  
        for(StudentEntity s:list) {  
            System.out.println(s.getS_name());  
        }  
    }  
}
```

Jpa bulk data operation

Query

```
query=em.createQuery("select  
e.ename from Employee e");
```

```
@SuppressWarnings("unchecke  
d")
```

```
List<String>  
l=query.getResultList();
```

```
System.out.println("Name  
of Employee");
```

```
for(String s:l)
```

- ▶ //Aggregate function
- ▶ Query query1 =
em.createQuery("Select
MAX(e.salary) from Employee
e");
- ▶ int result =
(int)query1.getSingleResult();
- ▶ System.out.println("Max
Employee Salary :" + result);
- ▶
- ▶

Persisting Objects in JPA

- ▶ In JPA, every entity going from a transient to managed state is automatically handled by the EntityManager.
- ▶ The EntityManager checks whether a given entity already exists and then decides if it should be inserted or updated. Because of this automatic management, the only statements allowed by JPA are SELECT, UPDATE and DELETE.
- ▶ In the examples below, we'll look at different ways of managing and bypassing this limitation.
- ▶

```
@Transactional public void insertWithQuery(Person person) {  
    entityManager.createNativeQuery("INSERT INTO person (id, first_name,  
    last_name) VALUES (?, ?, ?)") .setParameter(1, person.getId())  
    .setParameter(2, person.getFirstName()) .setParameter(3,  
    person.getLastName()) .executeUpdate(); }
```

JPA Inheritance Overview

- ▶ Inheritance is a key feature of object-oriented programming language in which a child class can acquire the properties of its parent class. This feature enhances reusability of the code.
- ▶ The relational database doesn't support the mechanism of inheritance. So, Java Persistence API (JPA) is used to map the key features of inheritance in relational database model.
- ▶ JPA Inheritance Strategies:
 - 1) Single table strategy
 - 2) Joined strategy
 - 3) Table-per-class strategy

JPA Single Table Strategy

- ▶ The single table strategy is one of the most simplest and efficient way to define the implementation of inheritance. In this approach, instances of the multiple entity classes are stored as attributes in a single table only.
- ▶ The following syntax represents the single table strategy: -
- ▶ `@Inheritance(strategy=InheritanceType.SINGLE_TABLE)`

JPA Joined strategy

- ▶ In joined strategy, a separate table is generated for every entity class. The attribute of each table is joined with the primary key. It removes the possibility of duplicacy.
- ▶ The following syntax represents the joined strategy: -
 1. `@Inheritance(strategy=InheritanceType.JOINED)`

JPA Table-per-class Strategy

- ▶ In table-per-class strategy, for each sub entity class a separate table is generated. Unlike joined strategy, no separate table is generated for parent entity class in table-per-class strategy.
- ▶ The following syntax represents the table-per-class strategy: -
- ▶ `@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)`

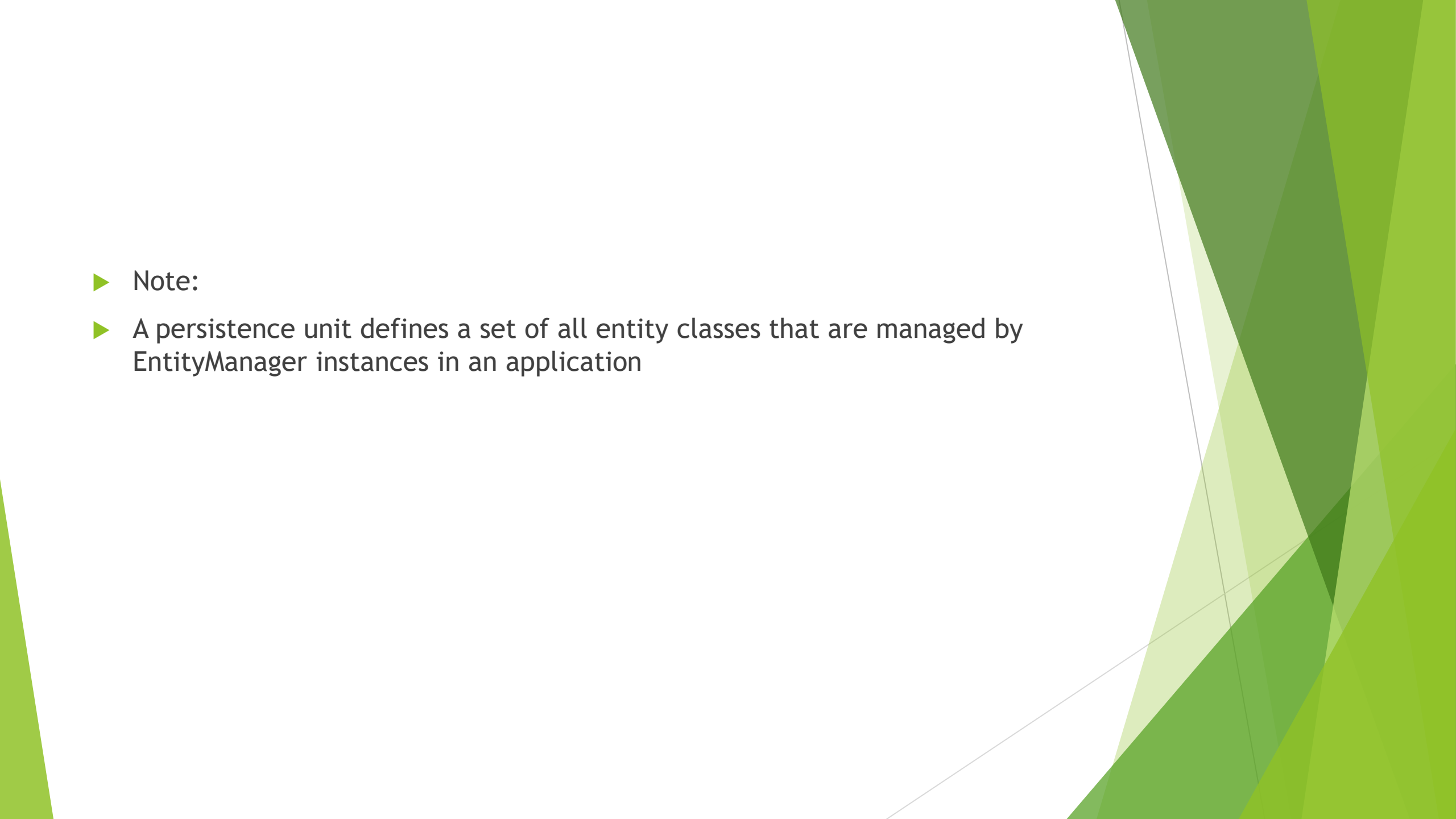
Caching

- ▶ Caching structurally implies a temporary store to keep data for quicker access later on
- ▶ Caching improves the performance of the application by pooling the object in the cache. It is useful when we have to fetch the same data multiple times.
- ▶ There are mainly two types of caching:
 - First Level Cache, and
 - Second Level Cache
- ▶ First Level Cache
- ▶ Session object holds the first level cache data. It is enabled by default. The first level cache data will not be available to entire application. An application can use many session object.

Second Level Cache

- ▶ SessionFactory object holds the second level cache data. The data stored in the second level cache will be available to entire application. But we need to enable it explicitly.
- ▶ Implementing Second Level Cache
- ▶ The `javax.persistence.Cache` interface of the persistence provider (`<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>`) can be used to interact with the second level cache.
- ▶ This interface provides functions such as, `contains`: to check whether the cache contains a given (as parameter) entity,
- ▶ variation of `evict`: to remove a particular entity from the cache,
- ▶ `evictAll`: to clear the cache and `unwrap`: to return specified cache implementation by the provider (ref. Javadoc JPA API).

- ▶ We can use @Cacheable annotation to make a POJO eligible to be cached. In this way the persistence provider knows which entity is to be cached. If no such annotation is supplied then entity and its state is not cached by the provider. @Cacheable takes a parameter of boolean value, default is true.
- ▶

- 
- The background of the slide features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic visual effect.
- ▶ Note:
 - ▶ A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application