9

# Java Persistence API (JPA)

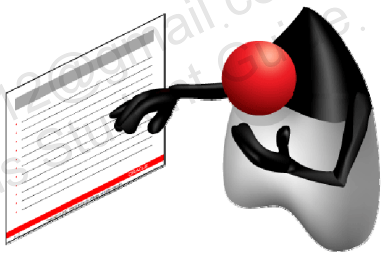
## Objectives

After completing this lesson, you should be able to:

- Describe the Java Persistence API (JPA)
- Define Object-Relational Mapping (ORM) and how JPA provides a framework to support ORM
- Use JPA to create, read, update, and delete database entities
- Create and use queries in JPA

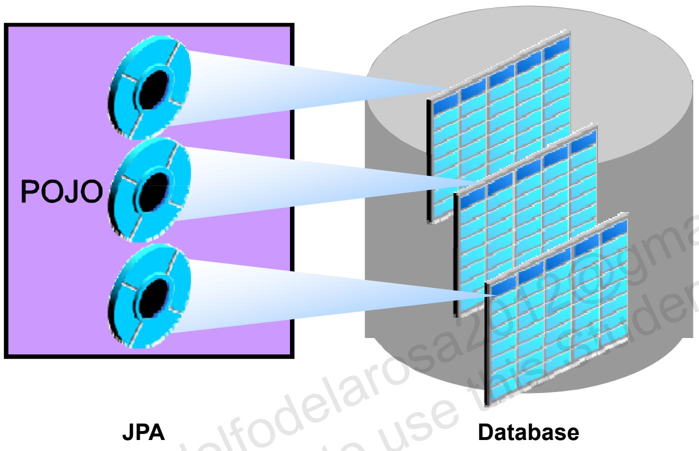
## Topics

- What is JPA?
- Components of JPA architecture
- Transactions
- Entity operations and queries



## Java Persistence API: Overview

The Java Persistence API (JPA) is a lightweight framework that leverages Plain Old Java Objects (POJOs) for persisting Java objects that represent relational data (typically in a database).



The Java Persistence API (1.0) began as a part of the Enterprise JavaBeans 3.0 specification (JSR-220) to standardize a model from Object-Relational Mapping.

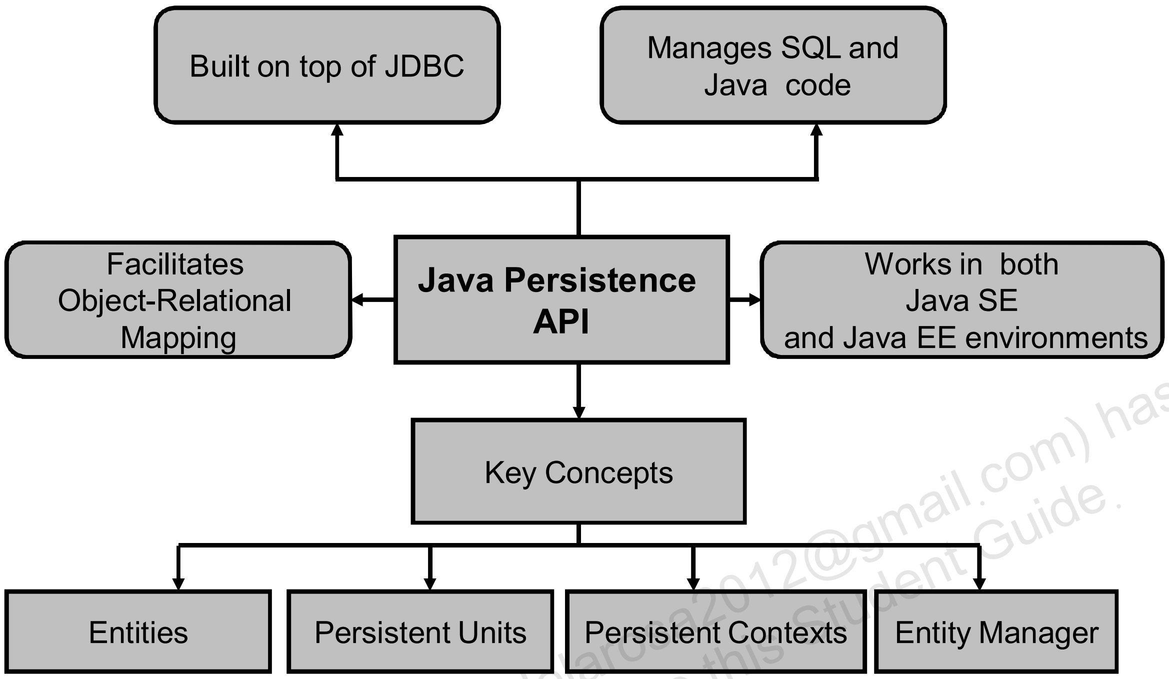
JPA 2.0 (JSR-317) set out to improve on the original JPA specification and was defined in its own JSR.

**POJO:** This term was added in the EJB 3.0 specification. It is a Java object that does not extend specific classes or implement specific interfaces as required by the EJB framework. Therefore, all normal Java objects are POJOs. POJO is another way of describing normal non-enterprise Java objects.

## Benefits of Using JPA

- You do not need to create complex data access objects (DAO).
- The API helps you manage transactions.
- You write standards-based code that interacts with any relational database, freeing you from vendor-specific code.
- You can avoid SQL and instead use a query language that uses your class names and properties.
- You can use and manage POJOs.
- You can use JPA for desktop application persistence.

## Features of JPA



JDBC was the first mechanism that Java developers used for persistent storage of data. However, working with JDBC requires that you understand how to map a Java object to a database table and maintain the set of SQL queries used to transform relational data into Java objects. JPA provides a framework for this Object-Relational Mapping while preserving the ability to manipulate databases directly.

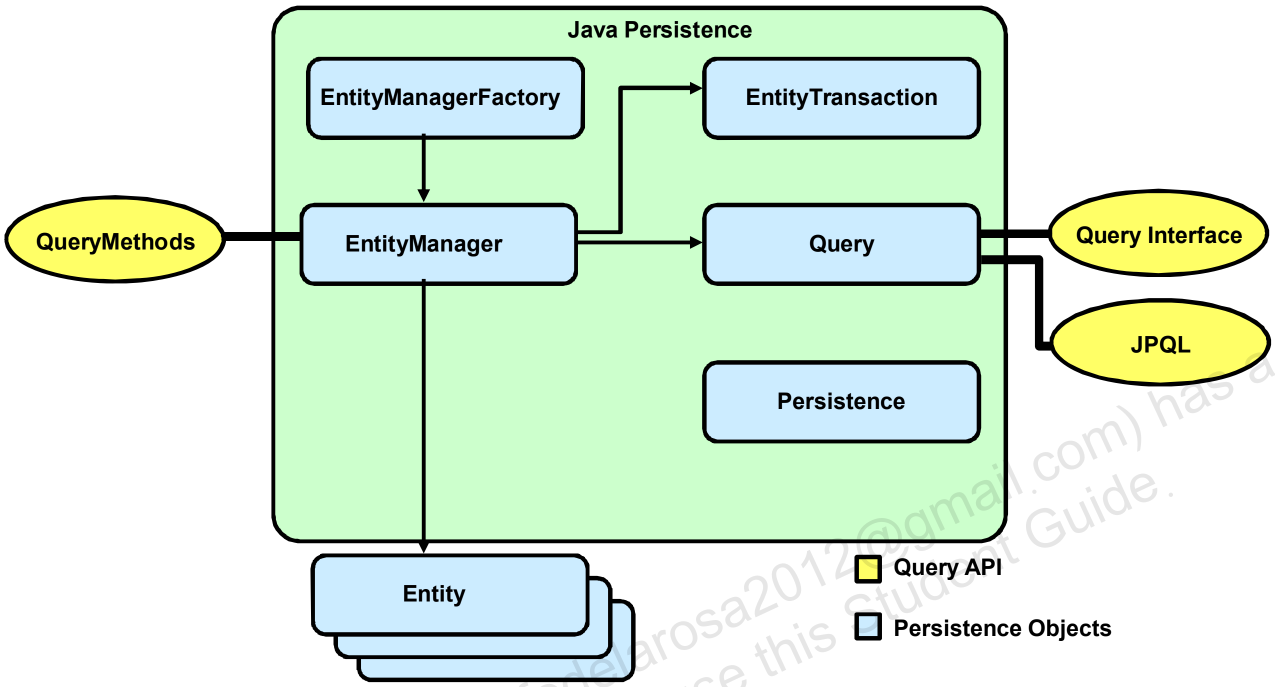
The key JPA concepts in this lesson are the starting points for writing JPA applications:

- An entity can be used to represent a relational table by a Java object. (This is a one-to-one mapping.)
- A persistence unit defines the set of all classes that are related or grouped by the application, and which must be co-located in their mapping to a single database.
- A persistence context is a set of entity instances in which there is a unique entity instance for any persistent entity identity.
- An entity manager does the work of creating, reading, and writing entities.

## Topics

- What is JPA?
- Components of JPA architecture
- Transactions
- Entity operations and queries

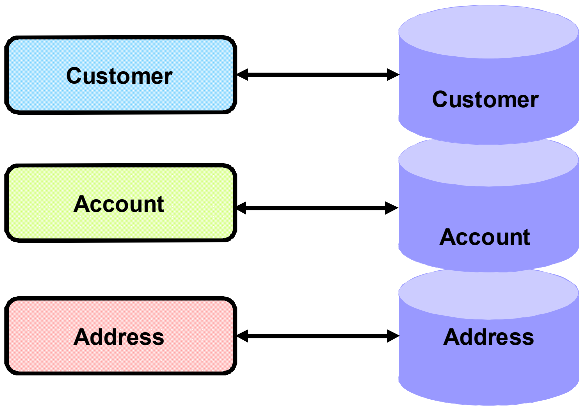
## JPA Components



The figure in the slide shows the components of the JPA architecture.

- **Entity:** The persistence object that represents one record in the database table. It is a (Plain Old Java Object) POJO class with annotations.
- **EntityManager:** The `EntityManager` interface provides the API for interacting with the entity. There are methods to persist the entity in the database, update the state of the entity, or remove the entity instance.
- **EntityManagerFactory:** The EntityManagerFactory is used to create an instance of EntityManager. When an application no longer uses the EntityManagerFactory, it is necessary to close the instance of EntityManagerFactory. After the EntityManagerFactory is closed, all its EntityManagers are also closed.

## Object-Relational Mapping (ORM)



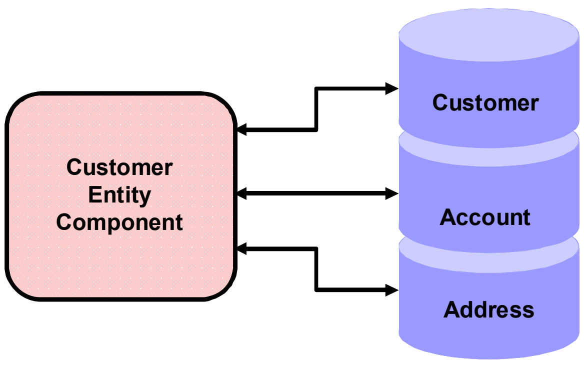
The storage mechanism that is most often used by applications is a relational database. A relational database is typically configured to store data in tables that have a relationship to one another. However, an object-oriented application design may not have the same organization in the same structure. A Java domain object can encompass partial data from a single database table, or it can include data from multiple tables depending on the normalization of the relational database.

Persistence mechanisms are the code that enable programmers to persist data (store) in a relational database. JDBC and JPA are two examples of persistence mechanisms.

Writing code to translate a relational schema to an object-oriented domain schema (or vice versa) can be time-consuming and error-prone. Object-Relational Mapping (ORM) software frameworks attempt to manage the mapping so that object-oriented programmers have to do very little coding. EclipseLink and Hibernate are examples of ORM software.

In the diagram in the slide, we see the most basic ORM mapping. This simple mapping of Java objects to database tables is a one-to-one mapping.

## Object-Relational Mapping (ORM)



The diagram in the slide shows a more complex ORM mapping, where a Java domain object is composed of data from multiple tables where there is a relationship across the tables. This is a one-to-many mapping.

This lesson focuses on the basics of using JPA in a simple one-to-one mapping. For more complex relationships, and a more in-depth coverage of JPA, enroll in the Oracle course titled *Building Database-Driven Applications with JPA*.

**Note:** EclipseLink is the continuation of Oracle's open-source version of TopLink, which Oracle donated to the Eclipse Foundation. EclipseLink is the reference implementation for the Java Persistence API 2.0 specification. For more information about EclipseLink, see http://www.eclipse.org/eclipselink/.

## JPA Entities

A JPA entity describes a concept or concrete thing that can be represented by attributes (data) that have possible relationships to other entities.

- Entities are meant to be persisted—to be stored in a relational database.
- An entity:
  - Is a POJO created by using the `new` keyword
  - Supports inheritance and polymorphism
  - Is serializable; can be used as detached objects
- Entities can be queried.
- Entities are managed at runtime through the Entity Manager API.

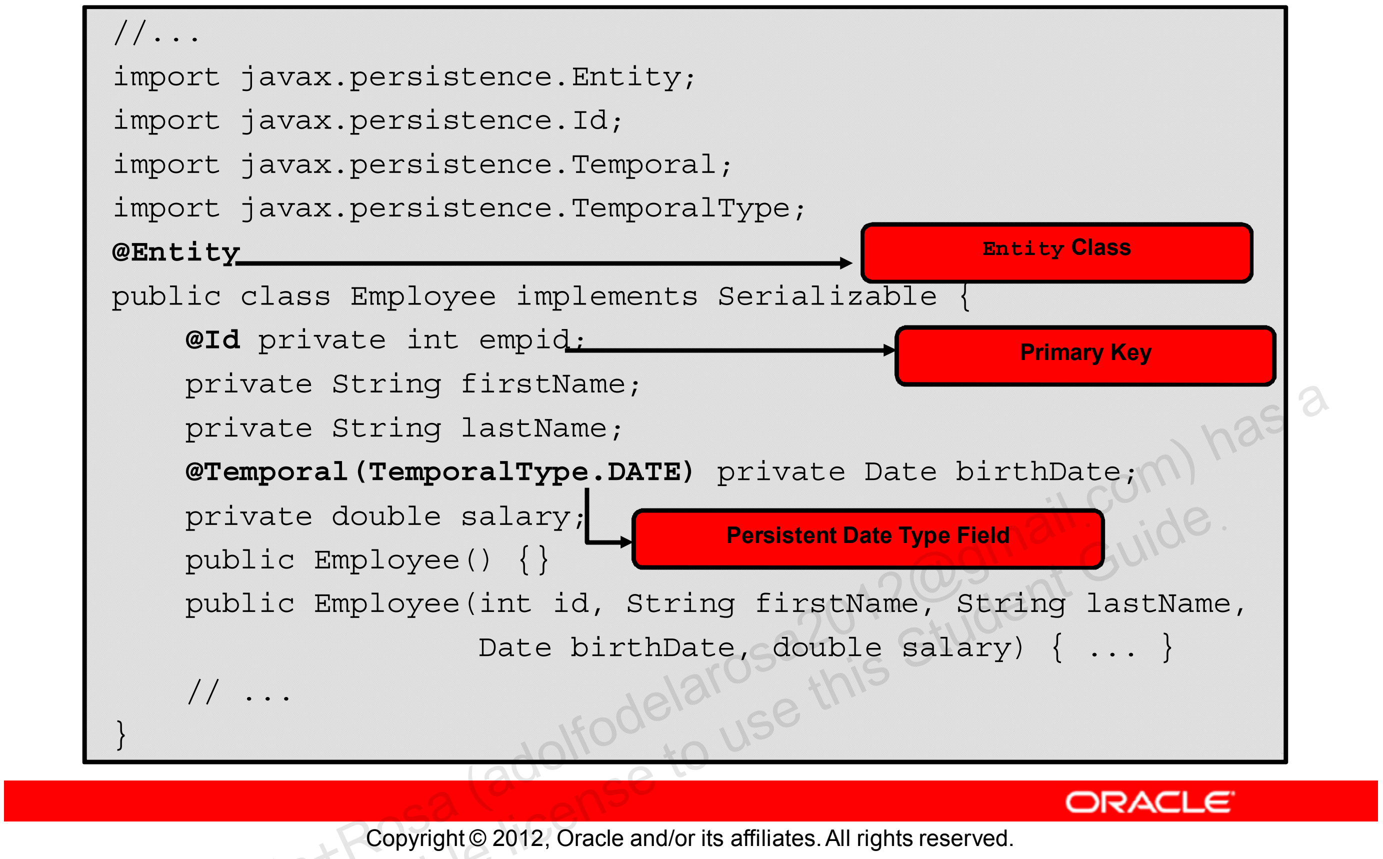
The JPA entity is the Java object that represents something to be persisted—a customer, an employee, a book, and so on. Basically, anything that is typically represented in a relational database can be represented by a POJO that defines the elements of the entity, and includes methods to set and get the values of each element.

The JPA characteristics of entities include:

- **Persistence:** Entities that are created can be stored in a database (persisted), but they are treated as objects.
- **Identity:** Each entity has a unique object identity that is usually associated with a key in the persistent store (database). Typically, the key is the database primary key.
- **Transactions:** JPA requires a transactional context for operations that commit changes to the database.
- **Granularity:** JPA entities can be as fine-grained or coarse-grained as a developer wants, but are intended to represent a single row in a table.

Entities are managed by the `EntityManager` API, which is covered later in this lesson.

## Creating an Entity



```
//...
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
@Entity
public class Employee implements Serializable {
    @Id private int empid;
    private String firstName;
    private String lastName;
    @Temporal(TemporalType.DATE) private Date birthDate;
    private double salary;
    public Employee() {}
    public Employee(int id, String firstName, String lastName,
                    Date birthDate, double salary) { ... }
    // ...
}
```

An entity can be created from a POJO by using annotations. In the example in the slide, you can see the `@Entity` annotation to declare the Employee class as an entity class to manage.

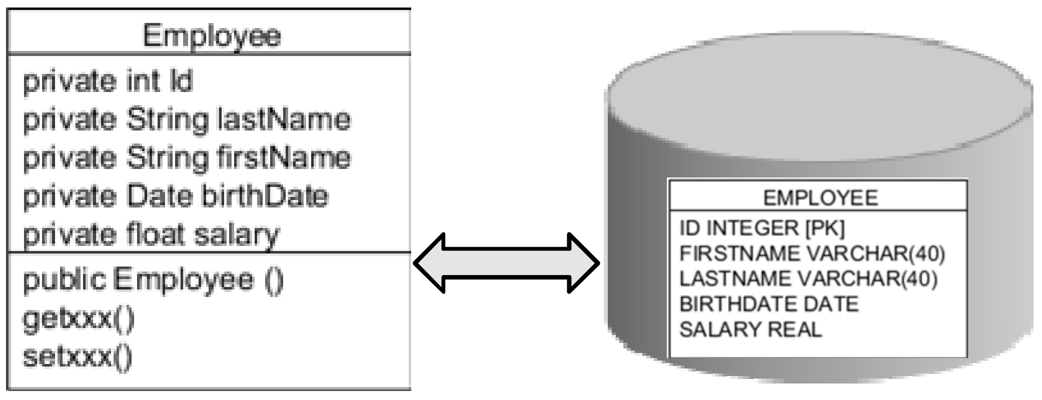
The `@Id` annotation declares the primary key. (Primary keys are covered in detail later in this lesson.)

The `@Temporal` annotation is required for any persisted field that is of `java.util.Date` or `java.util.Calendar` type. Because databases store dates in different ways (some as TimeStamp types and some as Date types), the `@Temporal` annotation indicates that you want Java to manage the conversion to and from the Java Date or Calendar type to the appropriate type in the database.

The requirements for an entity class include the following:

- The entity class must be annotated with the `@Entity` annotation.
- The entity class must have a no-arg constructor. The entity class may have other constructors as well.
- The no-arg constructor must be `public` or `protected`.
- The entity class must be a top-level class. An `enum` or `interface` must not be designated as an entity.
- The entity class must not be `final`. No methods or persistent instance variables of the entity class may be `final`.
- If an entity instance is to be passed by value as a detached object (for example, through a remote interface), the entity class must implement the `java.io.Serializable` interface.

## Entity Mapping



| ID (PK) | FIRSTNAME | LASTNAME | BIRTHDATE | SALARY |
|---|---|---|---|---|
| 110 | Troy | Hammer | 1965-03-31 | 102109.15 |
| 123 | Michael | Walton | 1986-08-25 | 93400.2 |
| 201 | Thomas | Fitzpatrick | 1961-09-22 | 75123.45 |
| 101 | Abhijit | Gopali | 1956-06-01 | 89345.0 |
| 120 | Rajiv | Sudahari | 1969-12-22 | 68400.22 |

In a simple entity mapping, the default name of the table used to store an entity is the name of the entity class.

Each row in the table is represented by an instance of an object that maps each of the elements of the table to a field in the class by name. So the table element FIRSTNAME is mapped to the `firstName` field in the Employee class.

You can modify the default table and field names by using annotations. You will learn this later in the lesson.

## Primary Keys

Every entity must have a primary key. The primary key is used to distinguish one entity instance from another.

- The primary key provides the identity of an entity and is used by the entity manager.
- The `@Id` annotation is used to identify the primary key.
- The most common primary key is an integer field or a string field, but it can be a custom class that corresponds to several database columns (compound key).

```
@Id private int id;
```

- Primary keys can be generated automatically.

```
@Id @GeneratedValue (strategy=GenerationType.AUTO)
@Column(name="ID")
private int id;
```

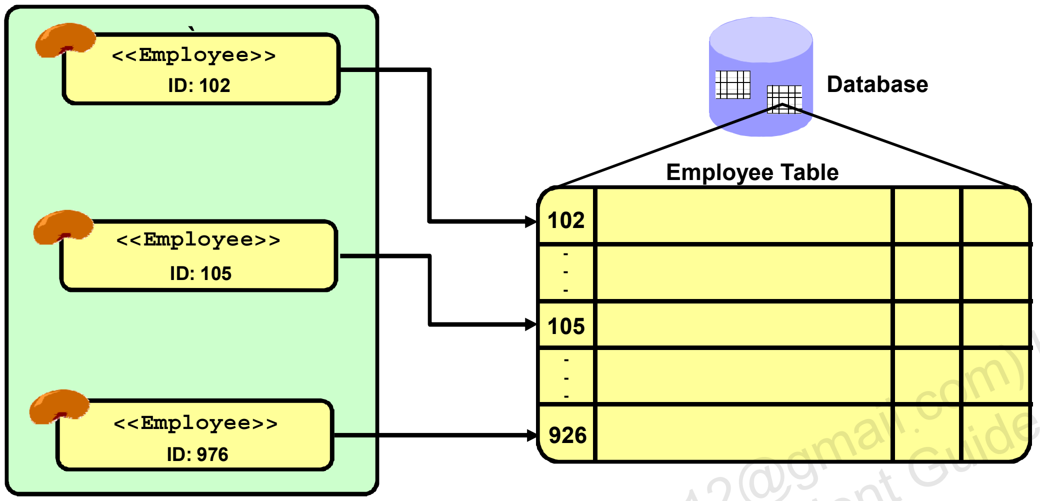
As shown in the previous slide, the primary key uniquely identifies each row in the Employee table.

Note that when you generate keys automatically, the primary key must be a simple key type, and the persistence provider is responsible for determining how to generate the key:

- **AUTO:** The persistence provider chooses the best strategy for key generation.
- **TABLE:** The persistence provider uses an underlying database table for key generation.
- **SEQUENCE, IDENTITY:** The persistence provider uses a sequence or identity column.

The entity's primary key is the `id` property and is correctly marked with the `@Id` annotation. Primary keys can be auto-generated values. The reference implementation will generate a key automatically if you add the `GeneratedValue` annotation to the primary key.

## Entity Component Primary Key Association



The figure in the slide shows the relationship between entity instances and table rows.

## Overriding Mapping

- `@Table`: Is used to override the default mapping between a class and table

```
@Entity
@Table(name="EMPLOYEE") //EMPLOYEE is the database table name
public class Emp {
    //...
}
```

- `@Column`: Is used to override column name mapping

```
public class Emp {
    @Column(name="FIRSTNAME")
    private String fName;
    //...
}
```

### JPA Annotations

Without overriding a table or column name, JPA uses introspection to determine the table and column names.

In the example shown in the slide, the name of the column in the EMPLOYEE table is FIRSTNAME and is mapped to a field named `fName`. So, while you write code that gets and sets or somehow changes the value of `fName`, JPA always maps this field data to FIRSTNAME in the database.

## Transient Fields

`@Transient`: Like the `transient` keyword, this annotation indicates that a field is not persistent.

```
public class Cart {
   @Id int customerId;
   //...
   @Transient float cartTotal;
//...
}
```

**`@Transient` Versus `transient`**

The JPA annotation `@Transient` provides an indication to the persistence provider not to persist a field. However, `@Transient` does not prevent a field from being serialized.

The code example in the slide shows a Cart class with a field cartTotal that is not required to be persisted.

## Persistent Fields Versus Persistent Properties

Entity classes have their state synchronized with a database. The state of an entity is obtained either from its fields or from its accessor methods (properties).

- Field-based or property-based access is determined by the placement of annotations.
- An annotation placed on one field (variable) means that the persistence state of the entity class is field-based.

```
@Entity
public class Employee implements Serializable {
    @Id private int id;
    @Temporal(TemporalType.DATE) private Date birthDate;
    //...
}
```

### Field-based and Property-based Persistent State

JPA provides two types of access to the data of a persistent class:

- **Field access:** Maps the instance variables (fields) to columns in the database table
- **Property access:** Uses the getters to determine the property names that will be mapped to the table

The access type depends on where you put the `@Id` annotation. For field-based access, the annotation is placed in the `id` field. For property-based access, the annotation is placed in the `getId()` method. You cannot use both access types in the same class.

**Persistent Field Access**

- Persistent fields cannot be `public`.
- Persistent fields should not be read by clients directly.
- Unless annotated with the `@Transient` annotation or `transient` keyword, all fields are persisted. (The use of the `@Column` annotation defines only the column name to use.)

**Note:** Accessor methods can be present in an entity class that uses field-based access and may be useful for other types of operations, but they are ignored by the persistence provider.

## Persistent Fields Versus Persistent Properties

An annotation placed on a getter method means that the persistence state is property-based.

```
@Entity
public class Employee implements Serializable {
    private int id;
    private String firstName;

    @Id @Column(name="ID") public int getId() {
        return id;
    }
    @Temporal(TemporalType.DATE) public Date getBirthDate() {
        return birthDate;
    }
    //...
}
```

**Persistent Properties**

When you use persistent properties, the persistence provider will retrieve an object's state by calling the accessor methods of the entity class.

- Methods must be declared `public` or `protected`.
- Methods must follow the JavaBeans naming convention.
- `@Column` may also be used to define the column name in the database.
- Persistence annotations can be placed only on getter methods.

Developers prefer filed access over property access for several reasons:

- With field access, your annotated persistent fields are all neatly organized near the top of your class.
- With field access, the state is well encapsulated.
- With property access, you must define getter methods for every field just for use by JPA even if your code will never call them.

However, as the application design grows, property access might be preferable for performance or security reasons.

## Persistence Data Types

Persistence fields or properties can be of the following data types:

- Java primitive types
- Java wrappers, such as `java.lang.Integer`
- `java.lang.String`
- `byte[]` and `Byte[]`
- `char[]` and `Character[]`
- Any serializable types including but not limited to:
  - `java.util.Date`
  - `java.sql.Date`
  - `java.sql.TimeStamp`
  - User-defined types

**Note**

Complex data types that do not have simple mapping (such as a `java.lang.String` to a `VARCHAR` or `CHAR` column) might require additional annotations. These annotations are described in the course titled *Building Database-Driven Applications with Java Persistence API.*

## Entity Manager

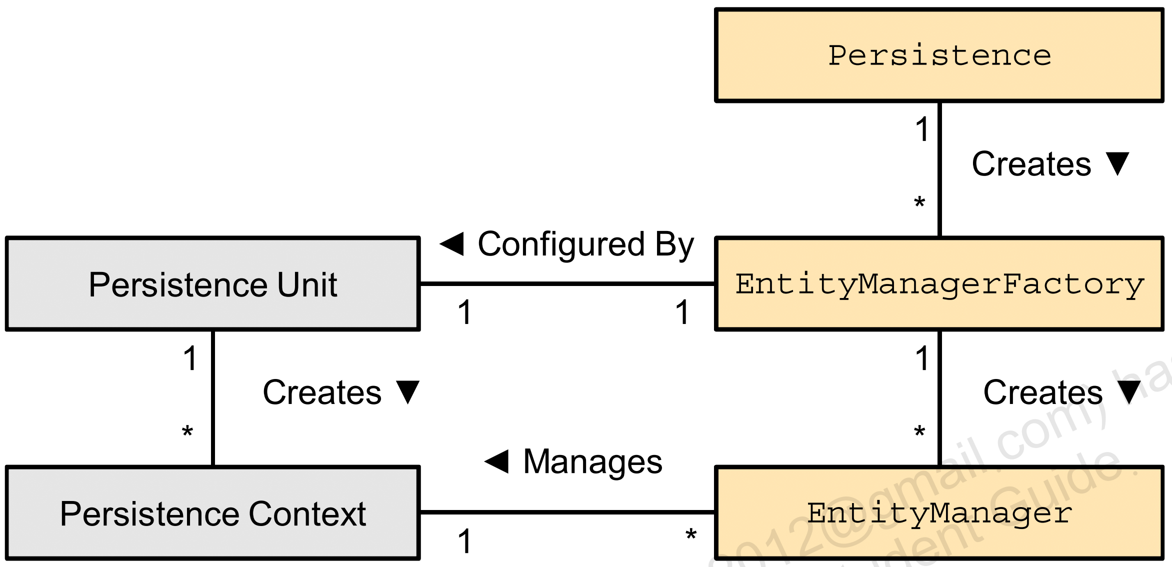
The `EntityManager` interface performs the work of persisting entities.

- Entities that an entity manager instance holds references to are *managed* by the entity manager.
- A set of entities in an entity manager at any given time is called its *persistence context*.
- Entities in the persistence context are unique. For example, there is only one Java instance with an ID of 110.
- A *persistence provider* creates the implementation details to read and write to a given database.
- An instance of an entity manager is obtained from a factory of the type `EntityManagerFactory`.

Entities do not add or remove themselves from the database when they are created or deleted. It is the logic of the application that must manipulate entities to manage their persistent life cycle. JPA provides the EntityManager interface for this purpose enable applications to manage and search for entities in the relational database.

A persistence unit is a named configuration of entity classes. A persistence context is a managed set of entity instances. Every persistence context is associated with a persistence unit, restricting the classes of the managed instances to the set defined by the persistence unit. Saying that an entity instance is managed means that it is contained in a persistence context and it can be acted on by an entity manager. This is why we say that an entity manager manages a persistence context.

# Persistence Unit



JPA is a specification for the API and life-cycle behavior. It is not ORM software by itself. To use the JPA to perform ORM operations, you must first obtain an instance of a JPA provider implementation.

A persistence unit provides the concrete classes and object-relational mapping information. There is a one-to-one relationship between the persistence unit and its concrete `EntityManagerFactory`. A persistence unit:

- Maps to a single database
- Defines the scope for queries and relationships
- Defines the configuration information for the persistence provider
- Is specified by using a `persitence.xml` file

Although there can be multiple `EntityManager` instances, they will all point to a single persistence context.

The `Persistence` class is a bootstrap class for obtaining an `EntityManagerFactory` instance in Java SE environments. In Java EE environments, an `EntityManager` can be obtained using dependency injection.

## Persistence Unit: Definition

The `persistence.xml` file is used to describe a persistence unit for an application.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" ...>
  <persistence-unit name="EmployeePU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>examples.model.Employee</class>
    <validation-mode>NONE</validation-mode>
    <properties>
      <property name="javax.persistence.jdbc.url"
value="jdbc:derby://localhost:1527/EmployeeDB"/>
      <property name="javax.persistence.jdbc.password" value="tiger"/>
      <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.user" value="public"/>
    </properties>
  </persistence-unit>
</persistence>
```

A persistence context is a grouping of unique entity instances that are managed by the persistence provider at runtime. A similar term is *persistence unit*, which is the set of all entity classes that an application might use. A persistence unit defines a group of entities mapped to a single database.

## Obtaining an Entity Manager

An entity manager instance is obtained from an `EntityManagerFactory`.

In Java SE environments, this instance is obtained through the `Persistence` bootstrap class:

```
    EntityMangerFactory emf =
    Persistence.createEntityManagerFactory
("EmployeePU");
```

- The string passed to the method identifies the name of the persistence unit configuration.
- From the factory, you can obtain an entity manager instance supported by the provider:

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

### EntityManager Instances

The persistence unit configuration is defined in the `persistence.xml` file. You will see this file later in this lesson.

## Using JPA in a Java SE Application

1. Create the database in JavaDB or any database server.
2. Add the required libraries to your Java application.
   – EclipseLink (JPA 2.0)
   – `DerbyClient.jar`
3. Define Entities in your application.
4. Create `persistence.xml` and configure it.
   a. Define the persistent unit and transaction type,
   b. Provide the names of the Entity classes.
   c. Define the JDBC connection properties.
5. Create instances of `EntityManagerFactory` and `EntityManager`.
6. Write code to perform persistent entity operations using the entity manager instance.

You have now learned about all the JPA components. This slide lists the steps to start developing a JPA application.

The Reference Implementation for the Java Persistence API is called EclipseLink and is an open-source and freely available Eclipse project derived from the Oracle TopLink product code base. EclipseLink can be downloaded from java.net and is part of the NetBeans IDE.

Steps 3 through 5 are easily generated if we use NetBeans.

Subsequent slides provide details about step 6.

## Quiz

To create an entity from a POJO, you must:

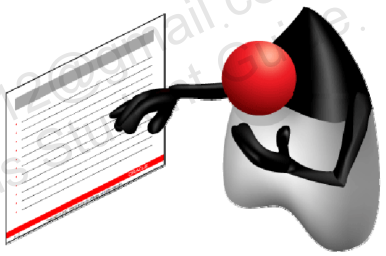
a. Implement the `javax.persistence.Entity` interface
b. Create a `persistence.xml` file
c. Annotate the class with `@Entity`
d. Declare the class as `final`

**Answer: c**

To define an POJO as an Entity class, you must annotate the class with `@Entity` (`javax.persistence.Entity`).

## Topics

- What is JPA?
- Components of JPA architecture
- Transactions
- Entity operations and queries



## What Is a Transaction?

- A transaction is a mechanism to handle groups of operations as though they were one operation.
- Either all operations in a transaction occur or none occur at all.
- The operations involved in a transaction might rely on multiple databases.

A typical example of using a transaction is as follows. Suppose that a client application needs to make a service request that might involve multiple read and write operations to a database. If any one invocation is unsuccessful, any state that is written (either in memory or, more typically, to a database) must be rolled back.

Consider an interbank fund transfer application in which money is transferred from one bank to another.

The transfer operation requires the server to make the following invocations:

1. Invoking the debit method on one account at the first bank
2. Invoking the credit method on another account at the second bank

If the credit invocation on the second bank fails, the banking application must roll back the previous debit invocation on the first bank.

## ACID Properties of a Transaction

A transaction is formally defined by the set of properties that is known by the acronym **ACID**.

- **A**tomicity: A transaction is done or undone completely. In the event of a failure, all operations and procedures are undone, and all data rolls back to its previous state.
- **C**onsistency: A transaction transforms a system from one consistent state to another consistent state.
- **I**solation: Each transaction occurs independently of other transactions that occur at the same time.
- **D**urability: Completed transactions remain permanent, even during system failure.

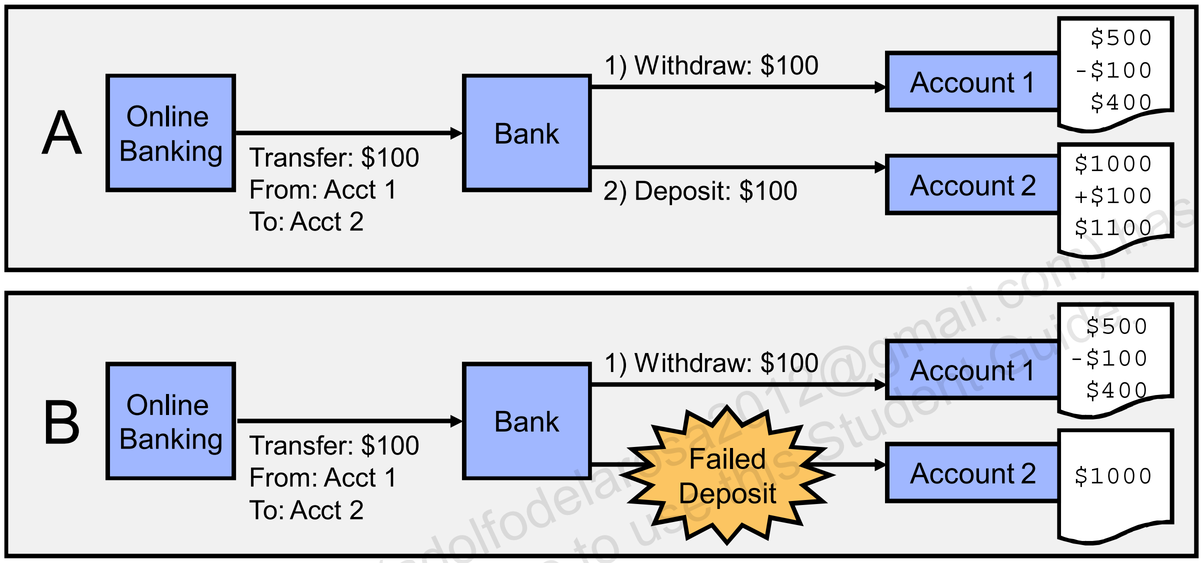
Transactions should have the following ACID properties:

- **Atomicity:** All or nothing; all operations involved in the transaction are implemented or none are.
- **Consistency:** The database must be modified from one consistent state to another. If the system or database fails during the transaction, the original state is restored (rolled back).
- **Isolation:** An executing transaction is isolated from other executing transactions in terms of the database records it is accessing.
- **Durability:** After a transaction is committed, it can be restored to this state if a system or database failure occurs.

## Transferring Without Transactions

A. Successful transfer

B. Unsuccessful transfer (Accounts remain in an inconsistent state.)



Consider an online banking application. Users can make online money transfers from one account to another account. In such a transfer scenario, one account should get debited and another account should get credited with the amount. The application must ensure that both operations happen successfully. If one operation fails, the other operation should also be nullified. This is easily managed in a transaction.

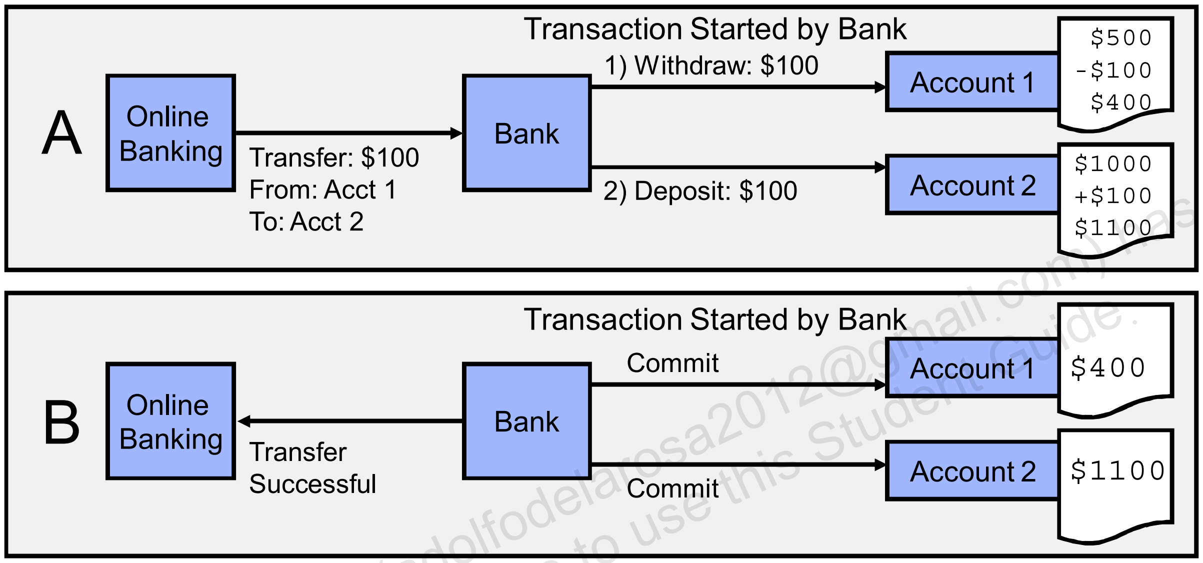
Transactions are appropriate in this online banking scenario.

A client application must converse with an object that is managed, and it must make multiple invocations on a specific object instance. The conversation can be characterized by one or more of the following steps:

1. Data is cached in memory or written to a database during or after each successive invocation.
2. Data is written to a database at the end of the conversation.
3. The client application requires that the object maintain an in-memory context between each invocation; each successive invocation uses the data that is maintained in memory.
4. At the end of the conversation, the client application requires the capability to cancel all the database write operations that may have occurred during or at the end of the conversation.

## Successful Transfer with Transactions

A. Changes in a transaction are buffered.
B. If a transfer is successful, changes are committed (made permanent).



If the transaction is successful, the buffered changes are committed (that is, made permanent).

Within the scope of one client invocation on an object, the object performs multiple changes to the data in a database. If one change fails, the object must roll back all the changes.

Consider a banking application. The client invokes the transfer operation on a teller object. The operation requires the teller object to make the following invocations on the bank database:

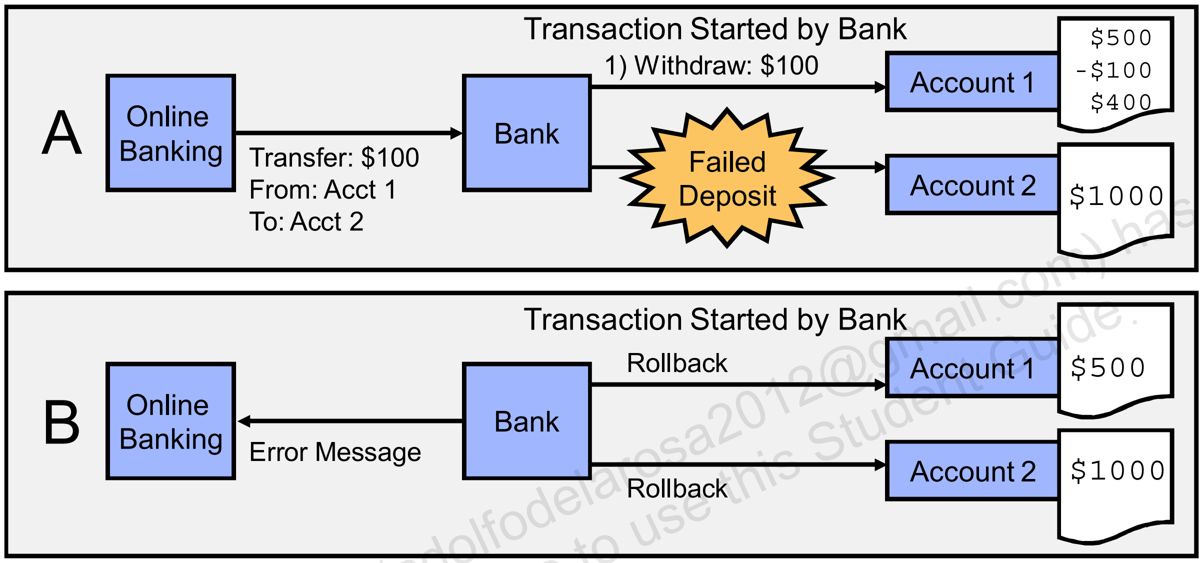
1. Invoking the debit method on one account
2. Invoking the credit method on another account

If the credit invocation on the bank database fails, the banking application must roll back the previous debit invocation.

## Unsuccessful Transfer with Transactions

A. Changes in a transaction are buffered.

B. If a problem occurs, the transaction is rolled back to the previous consistent state.



If the transaction is unsuccessful, the buffered changes are thrown out and the database is rolled back to its previous consistent state.

## Using JPA for Transactions

Entity operations are transactional. That is, they are required to be part of a transaction.

- Java SE applications can use resource-local transactions.
- The `EntityTransaction` interface supports resource local transactions.
- An `EntityTransaction` instance is obtained from `EntityManager`:

```
EntityTransaction et = em.getTransaction();
```

- Typical transaction methods include:
  - `begin()`: Starts a new resource transaction context
  - `commit()`: Completes the current transaction context and writes any unflushed changes to the database
  - `rollback()`: Rolls back the current transaction

There are two transactional models supported by JPA: resource-local transactions and JTA transactions.

- Resource local transactions are native transactions supported by the JDBC drivers referenced in the persistence unit.
- JTA transactions are part of a Java EE server.

There are three other methods in the `EntityTransaction` interface:

- **`setRollbackOnly`**: Sets the current transaction so that the only possible outcome is a rollback. This is especially appropriate for deleting a current long-running (atomic) transaction.
- **`getRollbackOnly`**: Returns a `boolean` indicating whether the current transaction is marked for rollback
- **`isActive`**: Returns a `boolean` indicating whether the resource transaction is in process

## Quiz

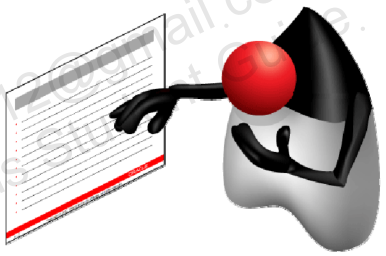
The transaction type in Java SE applications that use JPA is:

a. RESOURCE_LOCAL
b. JTA transactions
c. NULL
d. LOCAL

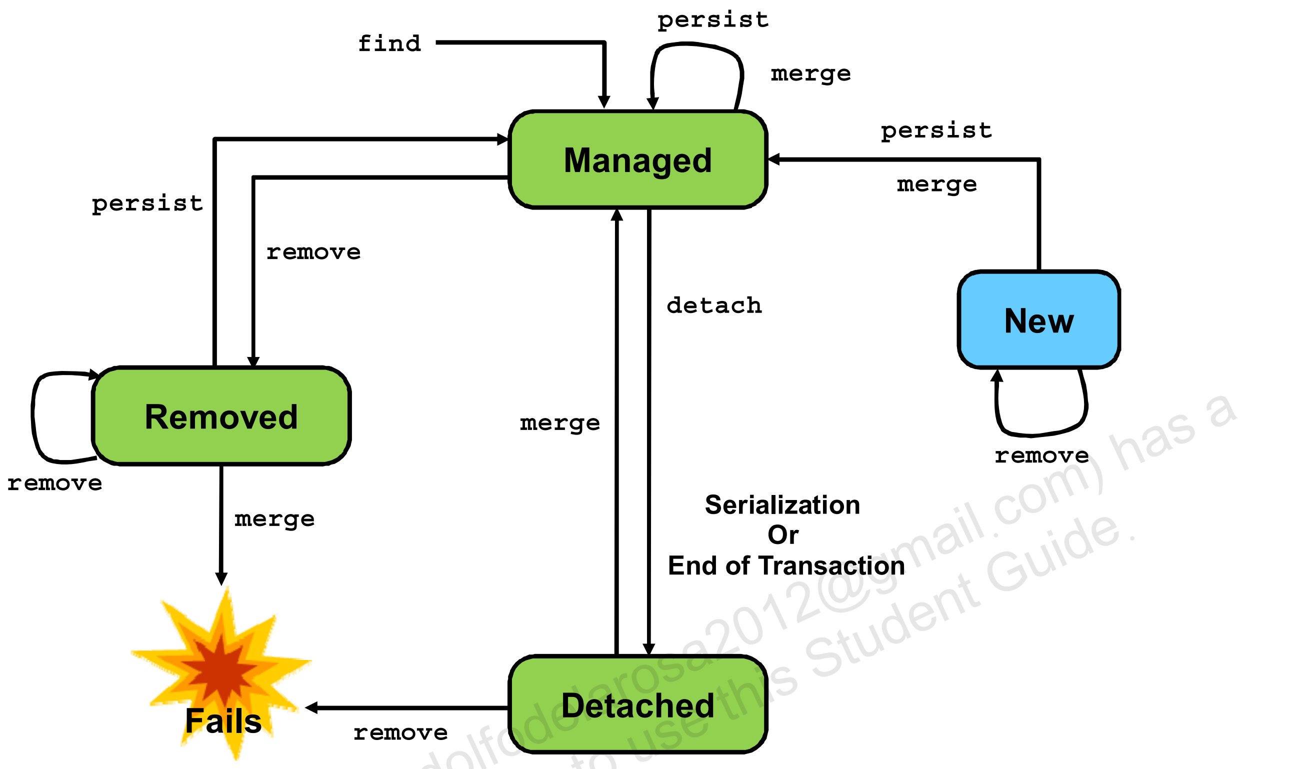
**Answer: a**

## Topics

- What is JPA?
- Components of JPA architecture
- Transactions
- Entity operations and queries



## Entity Instance Life Cycle



The life cycle of an entity is depicted in the slide. Entity instances are in one of four states: new, managed, detached, or removed.

- **New:** An entity object instance that has no persistent identity and is not yet associated with a persistence context (for example, when an Employee object is created using the new keyword)
- **Managed:** An instance with a persistent identity that is currently associated with a persistence context
- **Detached:** An instance with a persistent identity that is not (or is no longer) associated with a persistence context
- **Removed:** An instance with a persistent identity, associated with a persistence context, that will be removed from the database on transaction commit

You manage entity instances by invoking operations on the entity by means of an EntityManager instance.

The `EntityManager` interface defines the methods to manage entities, including:

- **Persisting an entity:** Taking a transient or one that has no persistent representation in the database and storing its state. After it is persisted, an entity is also in the managed state. Any further changes to this entity (before the transaction commit) are also persisted in the database.
- **Finding an entity:** Locating an entity in the persistent store. If an entity is successfully located in the persistent store, the entity is returned by the find method and managed by the entity manager.
- **Removing an entity:** Removing an entity from the persistent store (using a `DELETE` statement)
- **Updating an entity:** Locating an existing entity in the persistent store and making changes that are then persisted

These operations are the basic CRUD (Create, Read, Update, and Delete) operations performed on databases.

## Persisting an Entity

Persisting an entity is the operation of writing an entity to the database. It is the equivalent of a SQL `INSERT` statement.

```
public static void main(String[] args) {
     EntityManagerFactory emf =
         Persistence.createEntityManagerFactory("EmployeePU");
     EntityManager em = emf.createEntityManager();

     Employee emp = new Employee (158, "John", "Doe", new
GregorianCalendar(1967, 11, 23).getTime(), 45000);
     em.getTransaction().begin();
     em.persist (emp);
     em.getTransaction().commit();

     em.close();
     emf.close();
}
```

In the code fragment in the slide, an `EntityManagerFactory` is obtained using the `Persistence` class and `EmployeePU` named configuration.

From the factory, an `EntityManager` is obtained.

Note that using the `EntityManager`, the resource-local transaction is also started and then committed immediately after persisting the new Employee entity.

## Finding an Entity

To locate an entity in the database, the EntityManager will locate a row in the database based on the primary key:

```
Employee emp = em.find (Employee.class, 110);
```

- The `find` method uses the entity class passed in the first argument to determine what type to use for the primary key and for the return type. (So an extra cast is not required.)
- When the call returns, the employee returned is now a managed entity.
- If no employee with the ID 158 is found, a null is returned.

ORACLE

**Using the `EntityManager.find` Method**

This method is equivalent to a SQL `SELECT` statement.

Note that the second argument of the `find` method is `java.lang.Object`. Java automatically converts the integer primitive to an `Integer` class.

## Updating an Entity

- After an entity is managed, you can make changes to the entity object without explicitly persisting it.
- This update is the equivalent of a SQL `UPDATE ... WHERE` statement.

```
emp = em.find(Employee.class, 158);
em.getTransaction().begin();
emp.setSalary(emp.getSalary() + 1000);
em.getTransaction().commit();
```

Perhaps the most interesting aspect of JPA is that after an entity is managed, changes made to the entity are automatically reflected in the persistent store (in a transaction context).

You may also want to be able to update an entity that is not managed. To do this, you can use the `merge` method. You will see this in later slides.

## Deleting an Entity

The removal of an entity from the database is the equivalent of a SQL DELETE statement. To remove an entity, the entity must first be managed:

```
emp = em.find(Employee.class, 158);
em.getTransaction().begin();
em.remove(emp);
em.getTransaction().commit();
```

**Note:** If the value returned by find is `null`, the remove method throws an `java.lang.IllegalArgumentException`.

After the transaction is committed, the entity is removed (DELETE) from the persistent store. However, the Java object is still in memory.

## Detach and Merge

A managed entity requires a transaction context. But there may be times when you want to “unmanage” an entity to make changes before returning the entity to managed status.

```
em.detach(emp);
emp.setLastName("Jones");
// ... in some other part of the code
em.getTransaction().begin();
emp = em.merge(emp);
emp.setSalary(emp.getSalary() + 20000);
em.getTransaction().commit();
```

In this example, the employee entity is detached from the entity manager. After it is detached, the entity can be operated on as if it were a Java object rather than a database record. A detached entity is not deleted from the persistent store.

After the changes are determined to be correct, the entity can then be merged back into the entity manager. As the example shows, the employee object returned from the merge method is managed, so additional changes can be made and the appropriate updates can be committed to the database.

## Queries with JPQL

The JPA framework:

- Makes it possible to operate on individual entities
- Provides Java Persistence Query Language (JPQL) to query over entities, as in the following example:

```
 TypedQuery<Employee> query =
     em.createQuery("SELECT e FROM Employee e",
Employee.class);
 List<Employee> emps = query.getResultList();
 for (Employee e : emps)
 System.out.println("Employee: " + e);
```

- Queries can be dynamic or static.
  - Static queries are defined with an annotation:
    - `@NamedQuery, @NamedNativeQuery`

**Note:** JPQL is a rich language with many features. A full discussion of JPQL is beyond the scope of this lesson.

In the code fragment in the slide, `TypedQuery<Employee>` indicates that you want to create a query that returns entities that are Employee objects.

Here are a few features that JPQL supports:

- The results can be of single type or multiple type.
- Sorting and grouping
- Aggregate functions, expressions with conditions, and subqueries
- Syntax with joins
- Queries that allow bulk delete or updates
- Capture results in classes that are nonpersistent

`Query` and `TypedQuery` interfaces can be used to write queries.

A query may either be dynamically specified at runtime or configured in persistence unit metadata (annotation or XML) and referenced by name. Dynamic queries are nothing more than strings, and therefore may be defined as needed. Named queries are static and unchangeable. They are more efficient to execute because the persistence provider can translate the JPQL string to SQL once when the application starts as opposed to every time the query is executed.

## Example: `@NamedQuery`

- Defining a named query:

```
@NamedQuery(name="Employee.findByName",
query="SELECT e FROM Employee e WHERE
e.name = :name")
```

- Executing a named query:

```
 public Employee findEmployeeByName(String name) { return
     em.createNamedQuery("Employee.findByName", Employee.class)
     .setParameter("name", name) .getSingleResult();
```

A named query is defined using the `@NamedQuery` annotation, which may be placed on the class definition for any entity.

The annotation defines the name of the query as well as the query text.

If more than one named query is to be defined on a class, they must be placed inside of a `@NamedQueries` annotation, which accepts an array of one or more `@NamedQuery` annotations.

In the code snippet, `em` is an `EntityManager` instance. The following code indicates how it was created from the `EntityManagerFactory` instance:

```
EntityMangerFactory emf =  Persistence.createEntityManagerFactory
("EmployeePU");

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

## Quiz

If an entity does not exist in the persistent store, using the `remove` method on the entity will throw an exception.

a. True

b. False

**Answer: a**

To remove an entity, it must first be managed using the `find` method. If the `find` method returns a null (the entity was not found with the primary key passed to the `find` method), the `remove` method throws a `java.lang.IllegalArgumentException`.

## Quiz

Suppose that you are creating a new Employee entity. Which of the following steps is a best practice for persisting this entity?

a. Use the `update` method to add the record.
b. Use `find` to locate the record, and then `update` it.
c. Use `find` to see if the entity exists and, if not, `persist` it.
d. Use `find` to see if the entity exists, and then `merge` to update it.

**Answer: c**

It is a good practice to use `find` to determine if the entity exists in the persistent store. Another option would be to attempt to persist the Employee entity and then catch `EntityExistsException` to handle situations where the entity that is to be added already exists.

## Summary

In this lesson, you should have learned how to:

- Describe the Java Persistence API (JPA)
- Define Object-Relational Mapping (ORM) and how JPA provides a framework to support ORM
- Use JPA to create, read, update, and delete database entities
- Create and use queries in JPA

## Practice 9: Overview

- Practice 9-1: Creating Entity Classes by Using JPA
- Practice 9-2: Implementing CRUD Operations by Using JPA