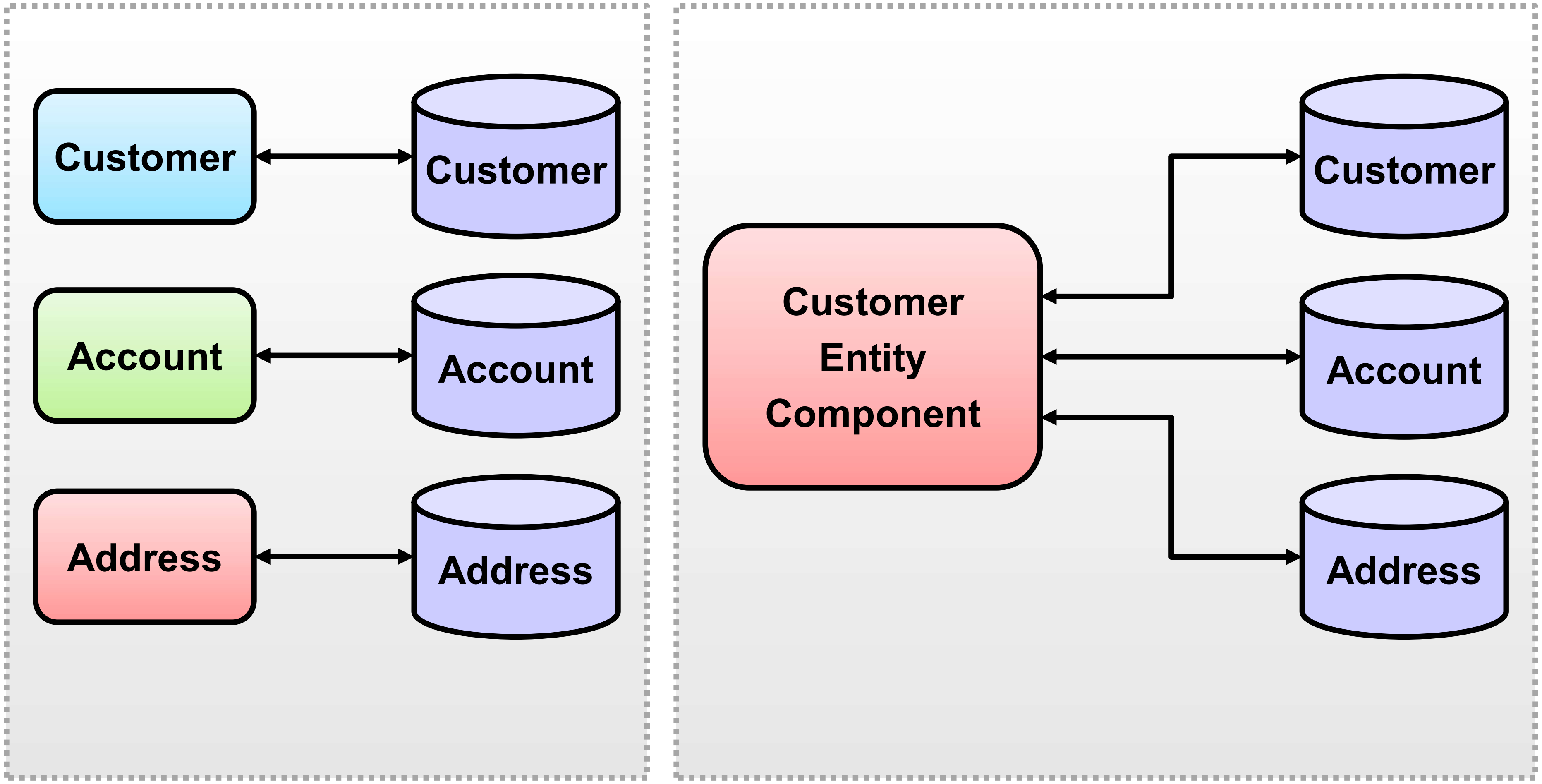
J P A

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).

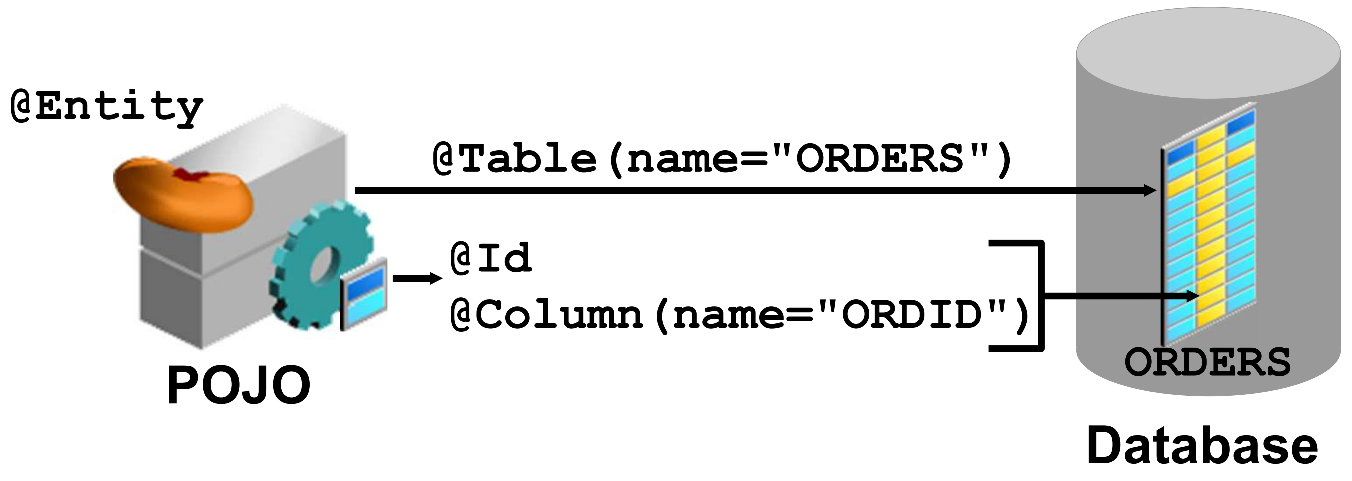
- JPA is built on top of JDBC and addresses the complexity of managing both SQL and Java code.
- JPA is designed to facilitate object-relational mapping.
- JPA works in both Java SE and Java EE environments.
- Key JPA concepts include:
  - Entities
  - Persistence units
  - Persistence contexts

Object-Relational Mapping



What Are JPA Entities?

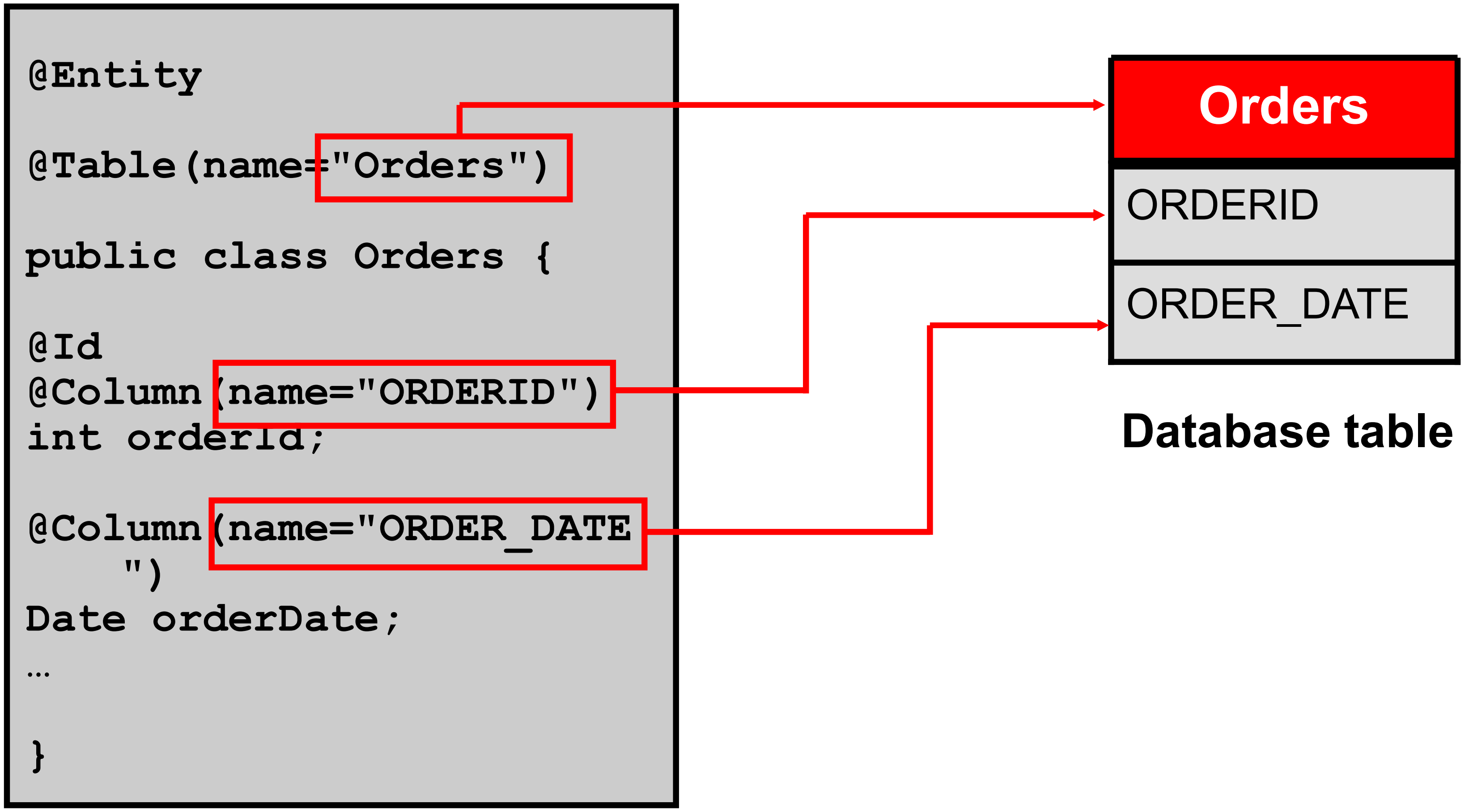
- A Java Persistence API (JPA) entity is:
  - A lightweight object that manages persistent data
  - Defined as a Plain Old Java Object (POJO) marked with the `@Entity` annotation



JPA Entity annotations

- The following annotations are used to map a POJO to a relational data construct:
    - `@Table` maps the object to a table.
    - `@Column` maps a field to a column (required if the field and column names are different)
    - `@Id` identifies primary key fields

JPA Entity mapping



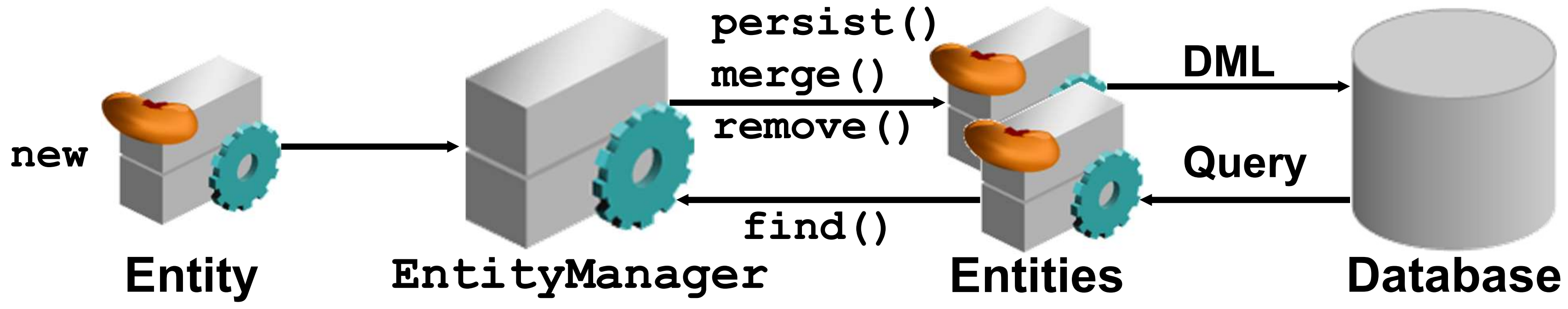
```
@Entity

@Table(name="Orders")

public class Orders {

@Id
@Column(name="ORDERID")
int orderId;

@Column(name="ORDER_DATE
    ")
Date orderDate;
…

}
```

Database table

JPA entity

Managing Persistence of Entities

- The life cycle of an entity is managed by using the `EntityManager` interface, which is part of the JPA.
- An entity can be created by using:
  - The `new` operator (creates detached instance)
  - The `EntityManager` Query API (synchronized with the database)
- An entity is inserted, updated, or deleted from a database through the `EntityManager` API.



EntityManager interface

- The EntityManager interface provides:
  - The find() method to retrieve a database row and instantiate an entity copy
  - Access to a Query API for creating and executing queries based on either of the following:
    - Java Persistence Query Language (JPQL)
    - Native SQL statements
  - Methods to perform persistent operations such as -
    - persist() to mark a new instance for insertion into the database
    - merge() to integrate (either insert or update) an instance into the database
    - remove() to remove an instance from the database

Declaring an Entity

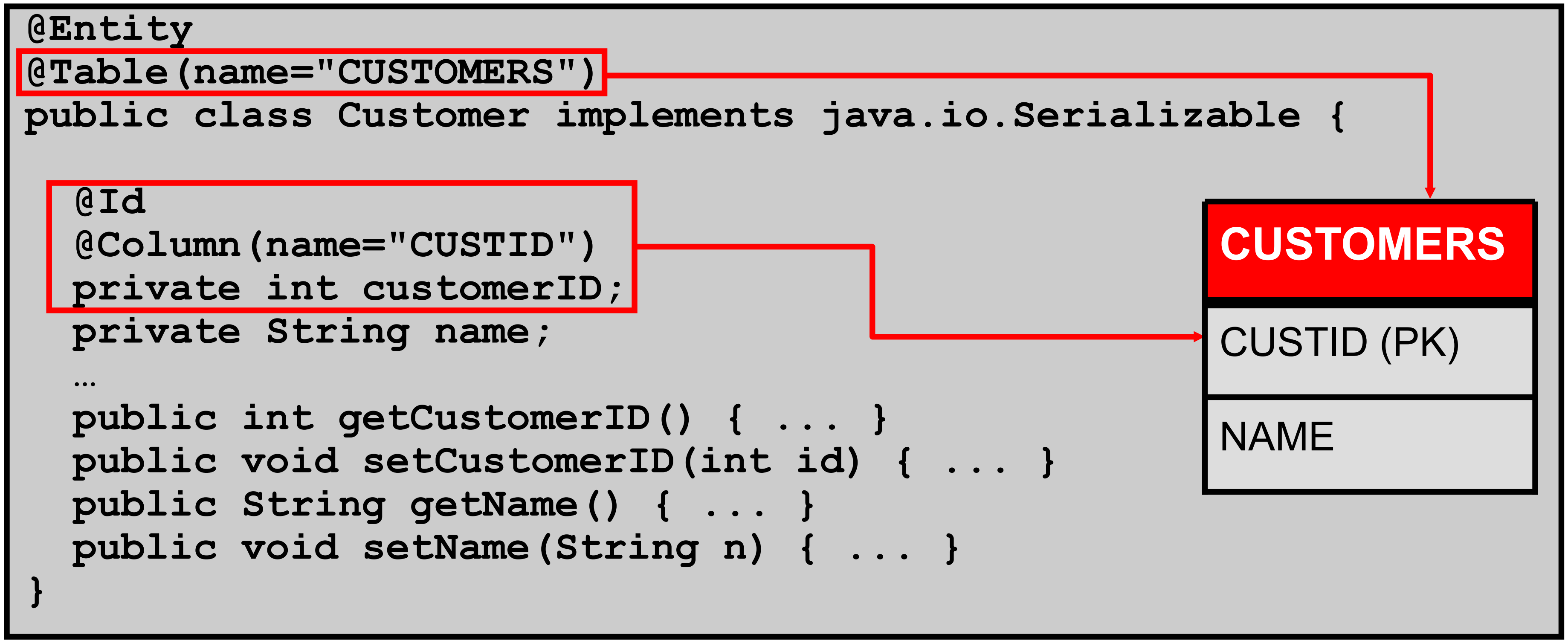
- Declare a new Java class with a no-arg constructor.
- Annotate it with `@Entity`.
- Add fields corresponding to each database column:
  - Add setter and getter methods.
  - Use the `@Id` annotation on the primary key getter method
- If `@Table` annotation is omitted, the class name is mapped to table name
- If `@Column` annotation is omitted, the field names are mapped to the column names

Declaring an Entity - example

```
@Entity
public class Customer implements
   java.io.Serializable {

   @Column (name = “CUSTID”)
   private int customerID;
   private String name;

   public Customer() { ... }  // no-arg
   constructor
   @Id                        // annotation
   public int getCustomerID() { ... }
   public void setCustomerID(int id) { ... }
   public String getName() { ... }
   public void setName(String n) { ... }
}
```

Mapping Entities

- Mapping of an entity to a database table is performed:
  - By default
  - Explicitly using annotations or in an XML deployment descriptor



```
@Entity
@Table(name="CUSTOMERS")
public class Customer implements java.io.Serializable {

  @Id
  @Column(name="CUSTID")
  private int customerID;
  private String name;
  …
  public int getCustomerID() { ... }
  public void setCustomerID(int id) { ... }
  public String getName() { ... }
  public void setName(String n) { ... }
}
```

Persistence Unit

- A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application
- This set of entity classes represents the data contained within a single data store.
- Persistence units are defined by the persistence.xml which will be placed in :
  - EJB JAR’s META-INF directory.
  - WAR file’s WEB-INF/classes/META-INFdirectory.

Persistence Unit

Example :

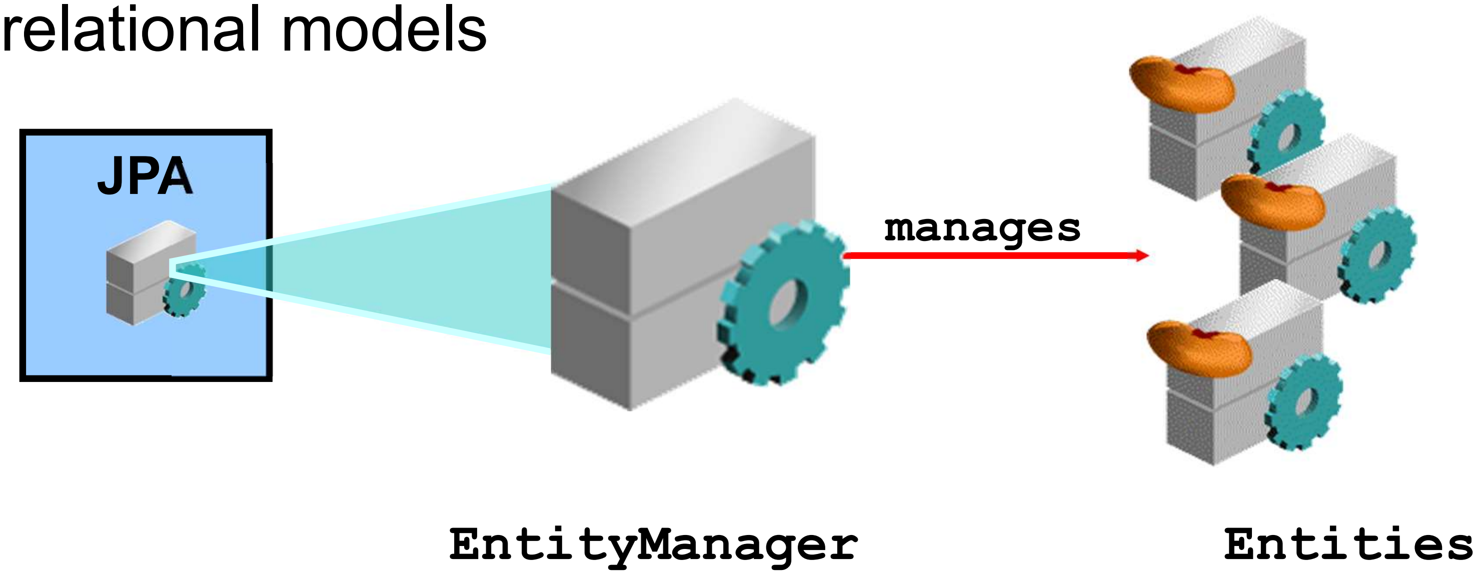
```
<persistence>
    <persistence-unit name="OrderManagement">
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <class>com.widgets.Order</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
```

Generating Primary key values

- A primary key can be given a value manually, or you can use the underlying persistence framework to automatically generate the primary key values using `@GeneratedValue` annotation
- `@GeneratedValue` strategies :
  - `TABLE` indicates that the container assigns values by using an underlying database table.
  - `SEQUENCE` and `IDENTITY` specify the use of a database sequence or identity column, respectively.
  - `AUTO` indicates that the JPA persistence provider should pick an appropriate strategy for the particular database (default)

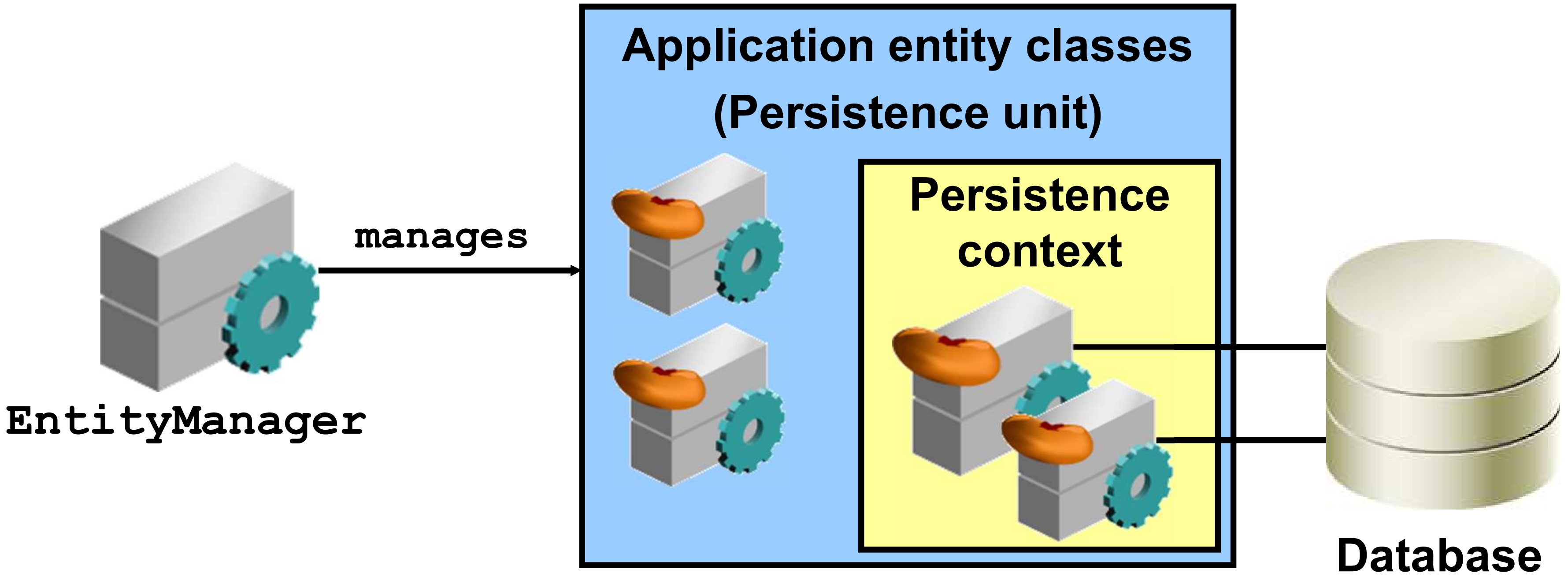
What is `EntityManager`?

- `EntityManager`:
  - Is an interface defined in JPA
  - Is a standard API for performing CRUD operations for entities
  - Acts as a bridge between the object-oriented and the relational models

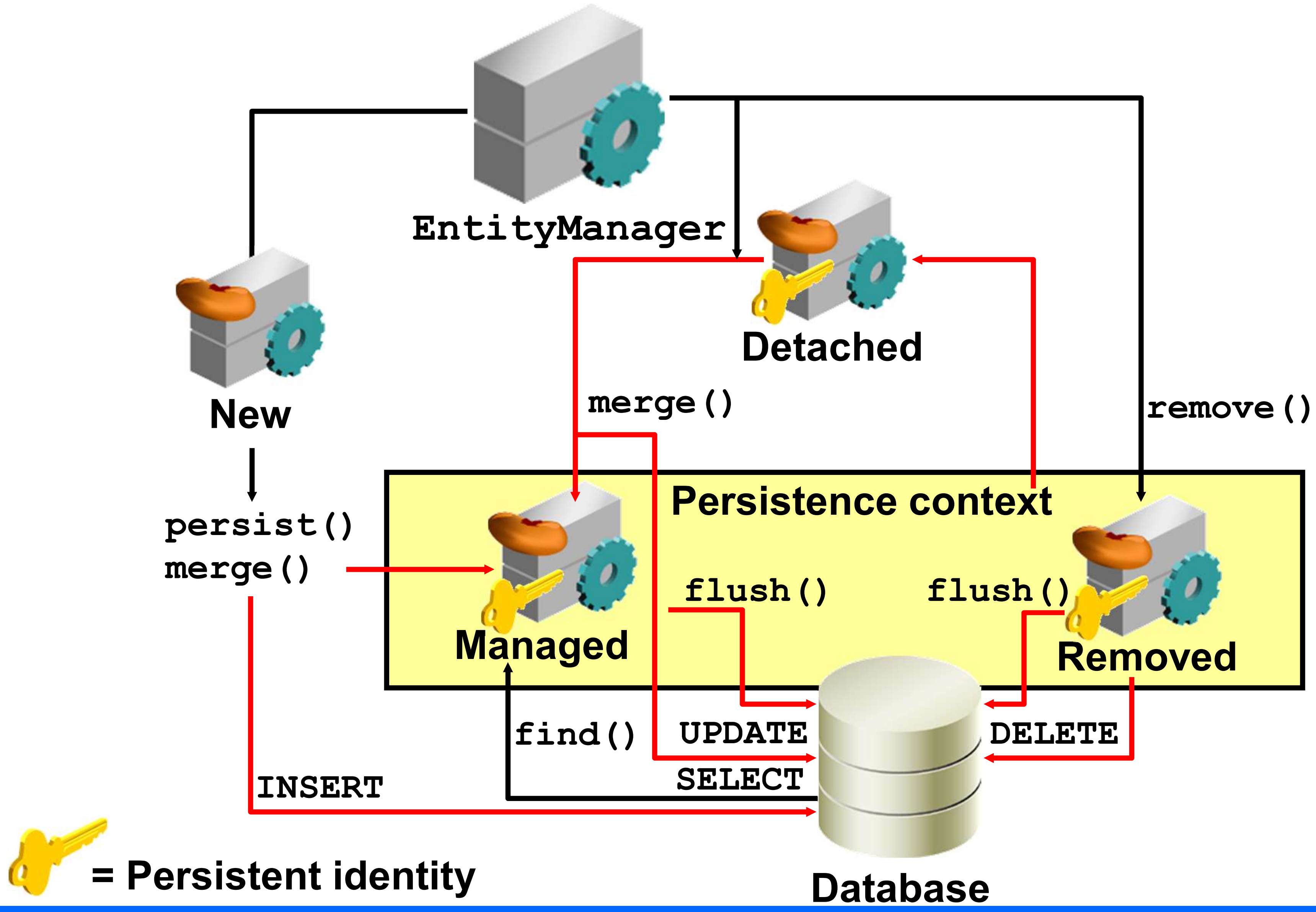


What Is `EntityManager`?

- `EntityManager` is:
  - Associated with a persistence context
  - An object that manages a set of entities defined by a persistence unit



Managing an Entity Life Cycle with `EntityManager`

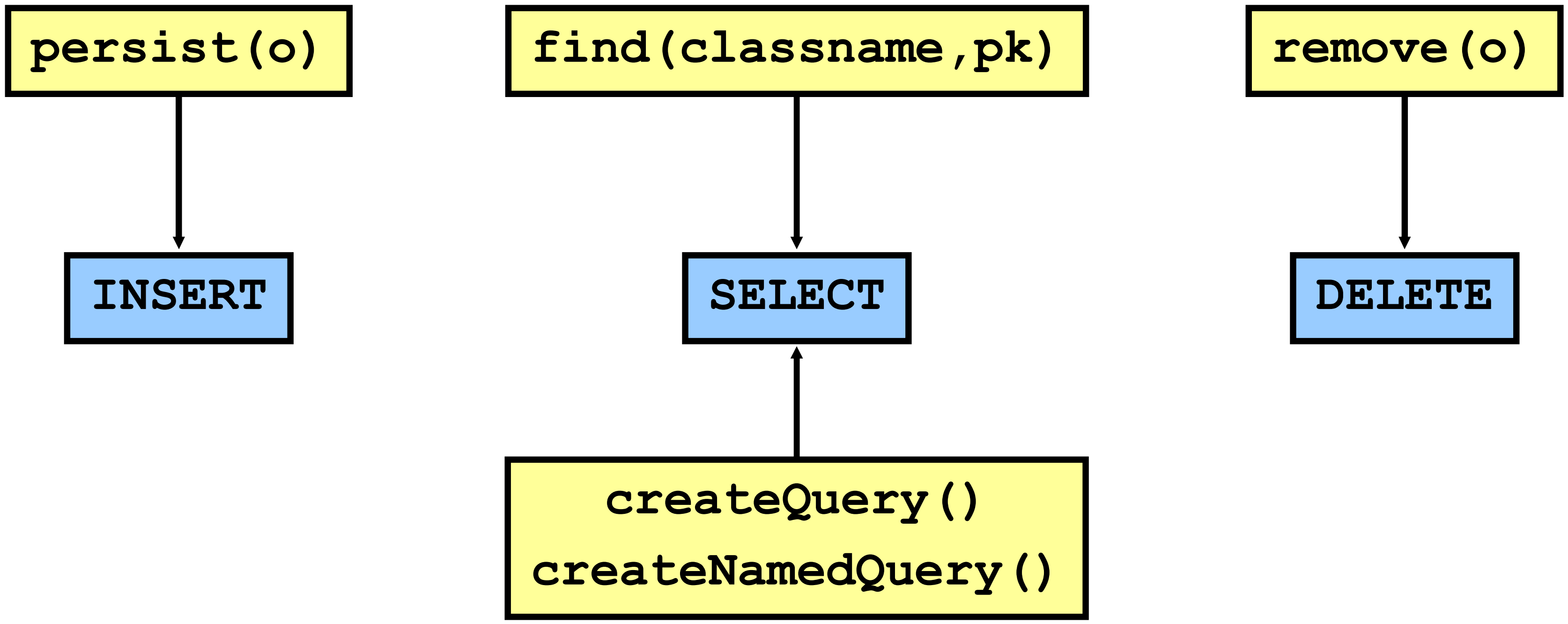


Accessing an `EntityManager` Instance

- Container-managed `EntityManager` instances:
  - Are implemented inside a Java EE container
  - Use the `@PersistenceContext` annotation
  - Are obtained in an application through dependency injection or JNDI lookup
- Application-managed `EntityManager` instances:
  - Are implemented outside a Java EE container
  - Are obtainable by using the `EntityManagerFactory` interface

Database Operations with `EntityManager` API

- The `EntityManager` API provides the following methods that map to CRUD database operations:



Inserting New Data

- To insert new data, perform the following steps:

1. Create a new entity object.
2. Call the `EntityManager.persist()` method.

```
@PersistenceContext(unitName="Model")
private EntityManager em; // inject the EntityManager
...                       // object
public void persistUser() {
  Users user = new Users();
  user.setFirstName("Steve");
  user.setLastName("King");

  em.persist(user);
  // On return the user object contains persisted state
  // including fields populated with generated id values
}
```

Deleting Data

- To delete data, perform the following steps:

1. Find, set, or refresh the state of the entity to be deleted.
2. Call the `EntityManager.remove()` method.

```
@PersistenceContext(unitName="Model")
private EntityManager em;
...
// Remove a Product by primary key Id value
 public void removeProducts(Products products) {
   products = em.find(Products.class,
                              products.getProdId());
   em.remove(products);
 }
...
```

Updating Data

- To update data, perform the following steps:

1. Find, set, or refresh the state of the entity to be updated.
2. Call the `EntityManager.merge()` method. `merge()` will insert the object if it does not exist

```
@PersistenceContext(unitName="Model")
private EntityManager em;
...
// Update a Product by primary key Id value
 public void updateProducts(Products products) {
   products = em.find(Products.class,
                                products.getProdId());
   products.setListPrice("1000");
...
    products = em.merge(products);
 }
...
```

Finding an Entity by Primary Key

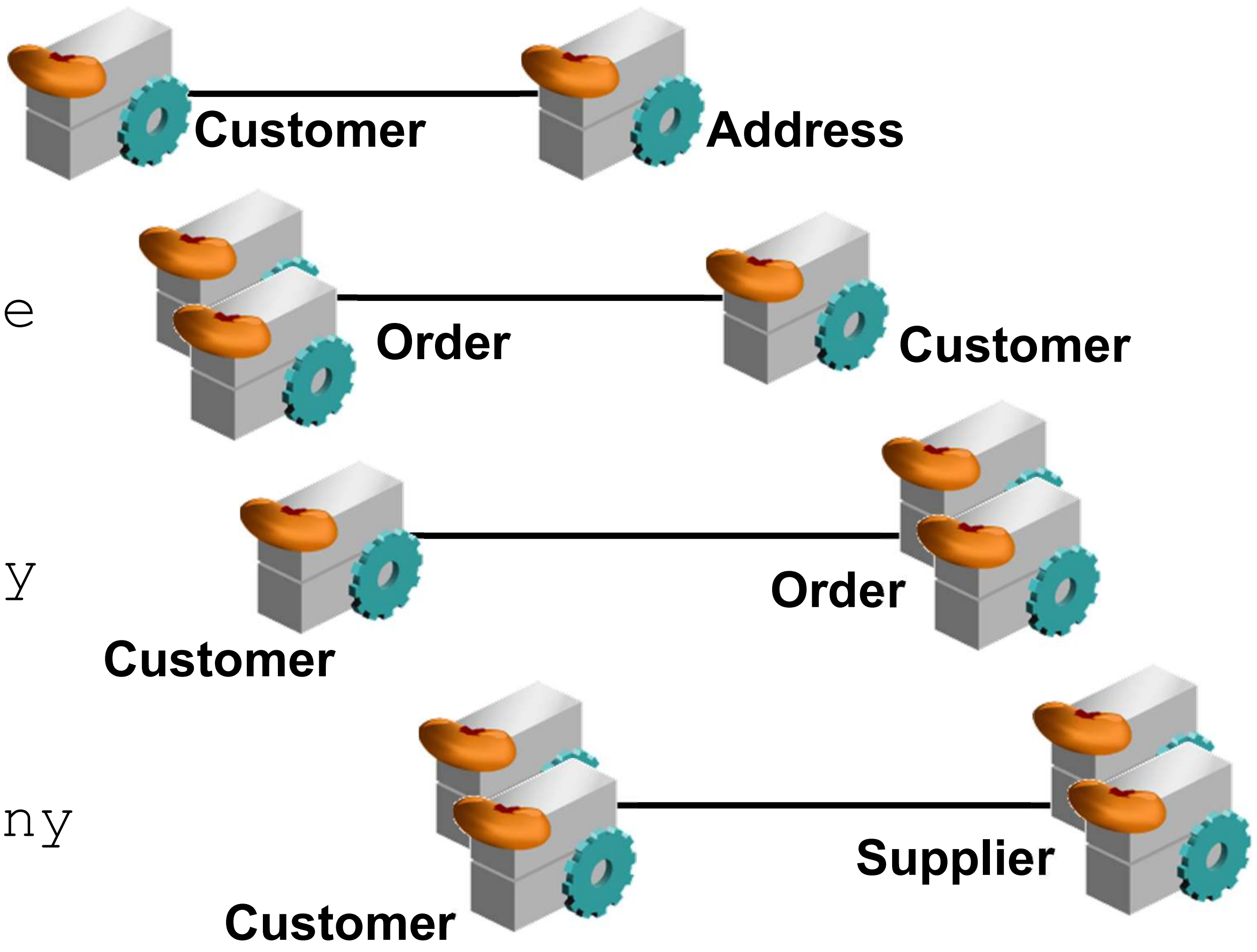
- To locate an entity by primary key, perform the following steps:

1. Create and set the primary key object and value.
2. Call the `EntityManager find()` method with the following parameters:
   - Entity class
   - Primary-key object

```
import org.srdemo.persistence.Users;
@PersistenceContext(unitName="Model")
private EntityManager em;
...
public Users findUserByPrimaryKey(Long id) {
  Users user = null;
  user = em.find(Users.class, id);
  return user;
}
```

Mapping Relationships Between Entities

- Annotations for entity relationships:
  - `@OneToOne`
  - `@ManyToOne`
  - `@OneToMany`
  - `@ManyToMany`



Relationships

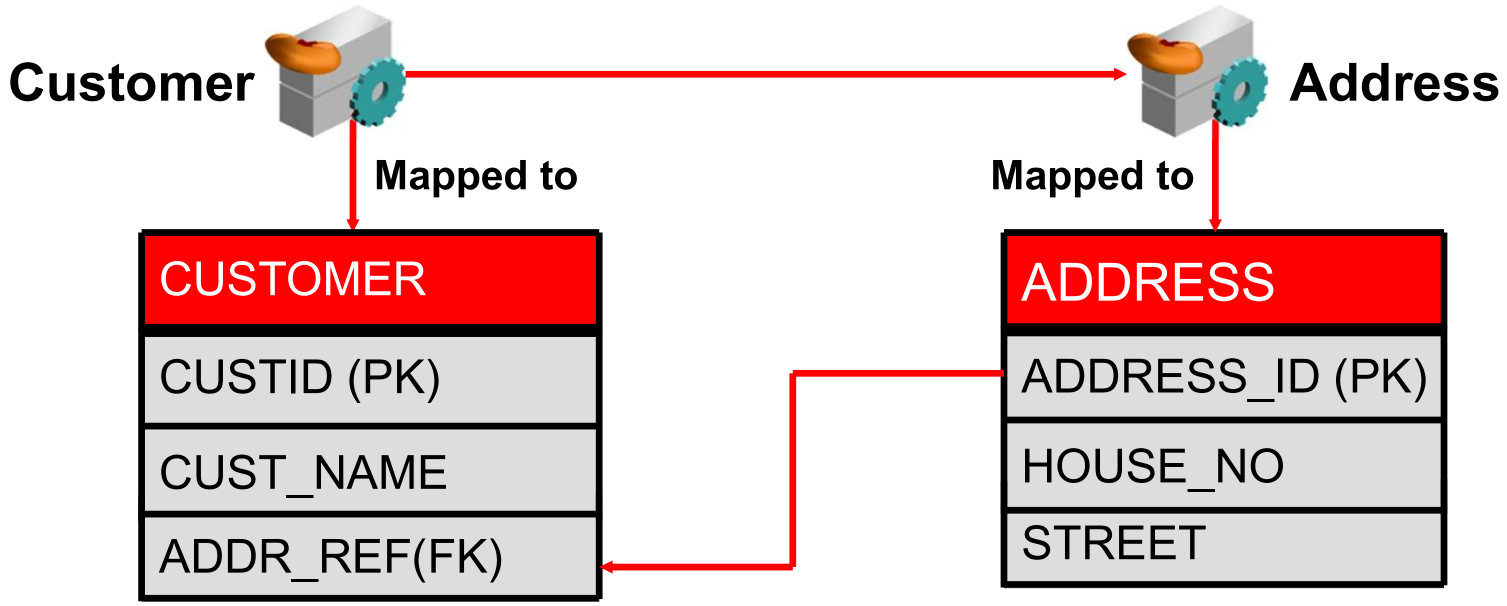
- Relationships can be unidirectional or bidirectional
- Bidirectional relationships, which have an owning side and an inverse side, are persisted based on references held by the owning side of the relationship
- Unidirectional relationships have only an owning side

Relationships

- In bi-directional relationship, the inverse side refers to its owning side by using the `mappedBy` element of the `OneToOne`, `OneToMany`, or `ManyToMany` annotation
- The `mappedBy` element designates the relationship's owning field.
  - The `mappedBy` element cannot be specified on the `ManyToOne` annotation and must be on the inverse side of a `OneToMany` or `OneToOne` relationship.
  - In `OneToOne` relationships, the owning side contains the foreign key.
  - In `ManyToMany` relationships, either side can be the owning side.

Implementing One-to-One Relationships

- You can map one-to-one relationships by using the `@OneToOne` annotation.
- Depending on the foreign key location, the relationship can be implemented by using:
  - The `@JoinColumn` annotation
  - The `@PrimaryKeyJoinColumn` annotation



Implementing One-to-One Relationships

- Example: Mapping a one-to one relationship between the `Customer` class and the `Address` class by using the `@JoinColumn` annotation

```
// In the Customer class:
@Table(name="CUSTOMER")
...
@OneToOne
@JoinColumn(name=“ADDR_REF",
                    referencedColumnName="ADDRESS_PK")
private Address address;
...

// In the Address class:
@Table(name="ADDRESS")
...
@column(name="ADDRESS_PK")
...
```

Bi-directional One-to-One Relationships

```
@Table(name="CUSTOMER")
class Customer {

    ...
    @OneToOne
    @JoinColumn(name="ADDR_REF",
                     referencedColumnName="ADDRESS_PK")
    private Address address;
...
}

@Table(name="ADDRESS")
class Address {
    @column(name="ADDRESS_PK")
    private  int addressId;

    @OneToOne (mappedBy="address")
    private Customer customer;
    ...
}
```

Implementing Many-to-One Relationships

- Mapping a many-to-one relationship:
  - Using the `@ManyToOne` annotation
  - Defines a single-valued association
- Example: Mapping an `Orders` class to a `Customer`

```
// In the Order class
@Entity
@Table(name="ORDER")
public class Order ... {
...
@ManyToOne
@JoinColumn(name="ORDERS_CUSTID_REF",
             referenceColumnName="CUSTID_PK”)
protected Customer customer;
...
}
```

Implementing One-to-Many Relationships

- Mapping a one-to-many relationship by using the `@OneToMany` annotation.

```
//In the Customer class:
@Table(name="CUSTOMER")
...
@OneToMany(mappedBy="customer")
protected Set<Order> order;
...

// In the Order class:
@Table(name="ORDER")
...
@ManyToOne
@JoinColumn(name="ORDERS_CUSTID_REF",
     referenceColumnName="CUSTID_PK", updatable=false)
protected Customer customer;
...
```

Implementing Many-to-Many Relationships

- Mapping a many-to-many relationship by using the `@ManyToMany` annotation.

```
// In the Customer class:
...
@ManyToMany(cascade=PERSIST)
@JoinTable(name="CUST_SUP",
             joinColumns=
           @JoinColumn(name="CUST_ID",referencedColumnName="CID"),
            inverseJoinColumns=
           @JoinColumn(name="SUP_ID", referencedColumnName="SID"))
protected Set<Supplier> suppliers;
...

// In the Supplier class:
...
@ManyToMany(cascade=PERSIST, mappedBy="suppliers")
protected Set<Customer> customers;
...
```

Cascade operations

- Entities that use relationships often have dependencies on the existence of the other entity in the relationship
- The javax.persistence.CascadeType enumerated type defines the cascade operations that are applied in the cascade element of the relationship annotations
- Example :

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
    public Set<Order> getOrders() {
            return orders;
    }
```

Cascade operations

Cascade Operation	Description
ALL	All cascade operations will be applied to the parent entity's related entity. All is equivalent to specifying cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}
DETACH	If the parent entity is detached from the persistence context, the related entity will also be detached.
MERGE	If the parent entity is merged into the persistence context, the related entity will also be merged.
PERSIST	If the parent entity is persisted into the persistence context, the related entity will also be persisted.
REFRESH	If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.
REMOVE	If the parent entity is removed from the current persistence context, the related entity will also be removed.

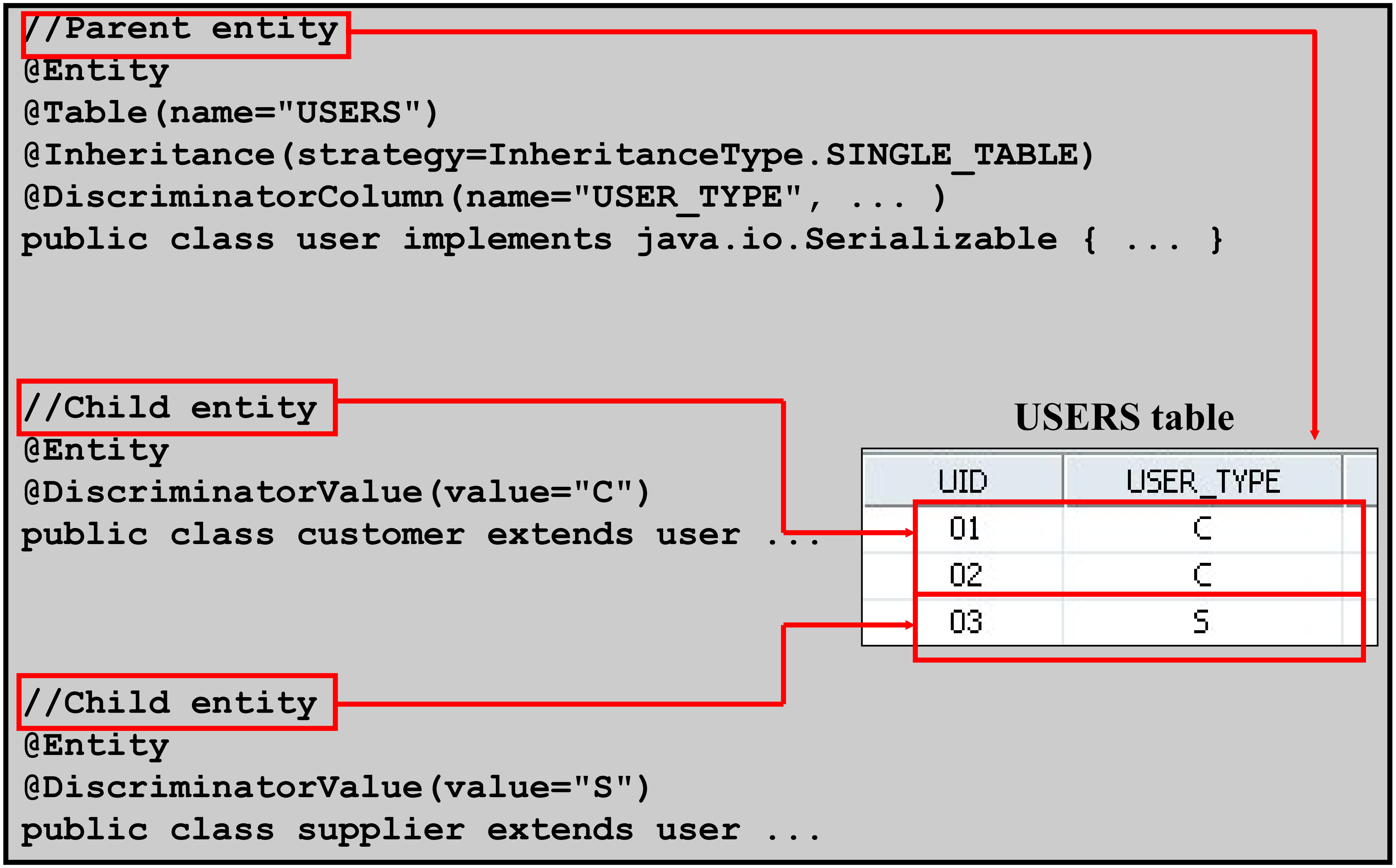
Mapping Inheritance

- Entities can implement inheritance relationships.
- You can use three inheritance mapping strategies to map entity inheritance to database tables:
  - Single-table strategy
  - Joined-tables strategy
  - Table-per-class strategy
- Use the `@Inheritance` annotation.

Mapping Inheritance

- ***SINGLE_TABLE:*** This is the default strategy. The entity hierarchy is essentially flattened into the sum of its fields, and these fields are mapped down to a single table.
- ***JOINED:*** Common base table, with joined subclass tables. In this approach, each entity in the hierarchy maps to its own dedicated table that maps only the fields declared on that entity. The root entity in the hierarchy is known as the base table, and the tables for all other entities in the hierarchy join with the base table.
- ***TABLE_PER_CLASS:*** Single-table-per-outermost concrete entity class. This strategy maps each leaf (i.e., outermost, concrete) entity to its own dedicated table. Each such leaf entity branch is flattened, combining its declared fields with the declared fields on all of its super-entities, and the sum of these fields is mapped onto its table.

Single-Table Strategy



```
//Parent entity
@Entity
@Table(name="USERS")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="USER_TYPE", ... )
public class user implements java.io.Serializable { ... }


//Child entity
@Entity
@DiscriminatorValue(value="C")
public class customer extends user ...


//Child entity
@Entity
@DiscriminatorValue(value="S")
public class supplier extends user ...
```

USERS table

UID	USER_TYPE
01	C
02	C
03	S

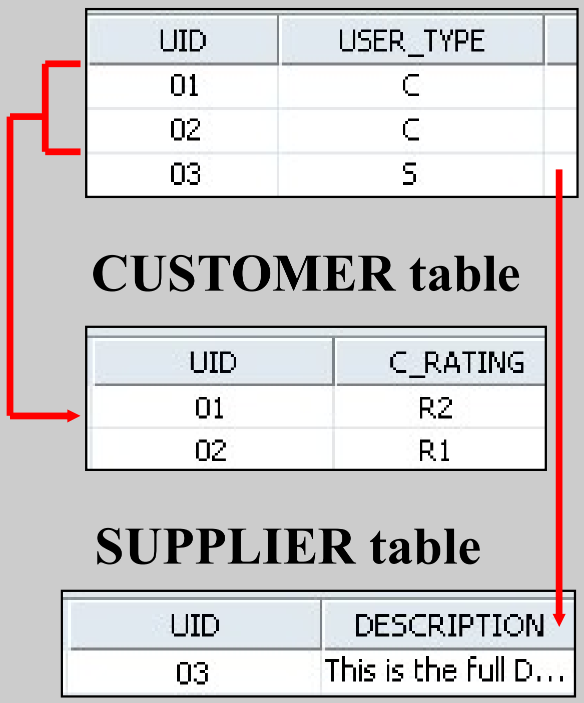
Single-Table Strategy

- The `@DiscriminatorColumn` annotation specifies the details of the discriminator column. The `name` element specifies the name of the discriminator, which is `USER_TYPE`.

- The `customer` class (subclass of `user`) specifies the discriminator value to be `"C"`, by using the `@DiscriminatorValue` annotation
- The `supplier` class (subclass of `user`) specifies the discriminator value to be `"S"`, by using the `@DiscriminatorValue` annotation.

Joined-Tables Strategy

```
//Parent entity
@Entity
@Table(name="USERS")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="USER_TYPE", ... )
public class user ...

//Child entity
@Entity
@Table(name="CUSTOMER")
@DiscriminatorValue(value="C")
@PrimaryKeyJoinColumn(name="UID")
public class customer extends user ...

//Child entity
@Entity
@Table(name="SUPPLIER")
@DiscriminatorValue(value="S")
@PrimaryKeyJoinColumn(name="UID")
public class supplier extends user ...
```



USERS table

UID	USER_TYPE
01	C
02	C
03	S

CUSTOMER table

UID	C_RATING
01	R2
02	R1

SUPPLIER table

UID	DESCRIPTION
03	This is the full D...

Joined-Tables Strategy

- In this type of strategy, the parent of the hierarchy contains only columns common to its children
- The `USERS` table contains columns (such as the `UID` column) that are common to all the user types
- The child tables in the hierarchy contain columns specific to the entity types. For example, the `C_RATING` column is specific to the `CUSTOMER` table
- The discriminator column is still used in the parent table to differentiate the user's type in the hierarchy

What Is JPA `Query` API?

- The JPA `Query` API:
  - Includes:
    - `EntityManager` methods to create queries
    - `Query` interface methods for executing queries
    - Java Persistence Query Language (JPQL)
  - Supports:
    - Named queries
    - Dynamic queries

Retrieving Entities by Using the `Query` API

- The `EntityManager` interface provides the `Query` API methods to execute JPQL statements:

`EntityManager`

```
EntityManager
 ├→createQuery(String jpql)
 └→ createNamedQuery(
      String name)
```

`Query` instance methods:

```
setParameter(String, Object)
Object getSingleResult()
List getResultList()
Query setMaxResults(int)
Query setFirstResult(int)
int executeUpdate()
```

Writing a Basic JPQL Statement

- Syntax for a simple JPQL statement:

```
SELECT object(o)
FROM abstract-schema-name o
[WHERE condition]
```

clauses
SELECT O and
SELECT OBJECT(O)
are synonymous

- Examples:
  - Find all Users entities:

```
SELECT object(o) FROM Users o
```

  - Find a Users entity with a specific email address:

```
SELECT object(o) FROM Users o
WHERE o.email = 'steve.king@srdemo.org'
```

  - Find a Users entity based on a parameter value:

```
SELECT object(o) FROM Users o
WHERE o.firstName = :givenName
```

Creating Named Queries

- To create a named query, perform the following steps:

1. Define the query with the `NamedQuery` annotation.

```
@NamedQuery(name="findUsersByCity",
   query="SELECT object(o) FROM Users o " +
          "where o.city = :city");
```

2. Create a `Query` object for the named query with the `createNamedQuery()` method, setting parameters and returning results.

```
public List<Users> findUsersinCity(String cityName) {
  Query query = em.createNamedQuery("findUsersByCity");
  query.setParameter("city", cityName);
  return query.getResultList();
}
```

Writing Dynamic Queries

- Example: Find service requests by primary key and a specified status.

```
public List findServiceRequests(Long id, String status){
  if (id != null && status != null ) {
    Query query = em.createQuery(
     "select object(sr) from ServiceRequests sr " +
     "where sr.svrId = :srvId and sr.status = :status");
    query.setParameter("srvId", id);
    query.setParameter("status", status);
    return query.getResultList();
  }
  return null;
}
```