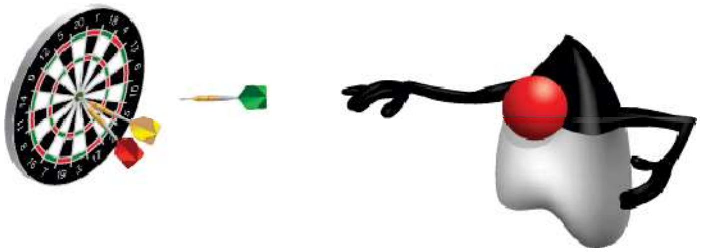
7

Developing Persistence Layer with JPA Entities

Objectives

After completing this lesson, you should be able to do the following:

- What are JPA Entities?
- Domain Modeling with JPA
- Creating an Entity (a POJO with annotations)
- Specifying Object Relational (OR) Mapping
- Mapping Relationships between Entities
- Inheritance Mapping Strategy (Singe Table, Joined Subclass)



What Is Persistence?

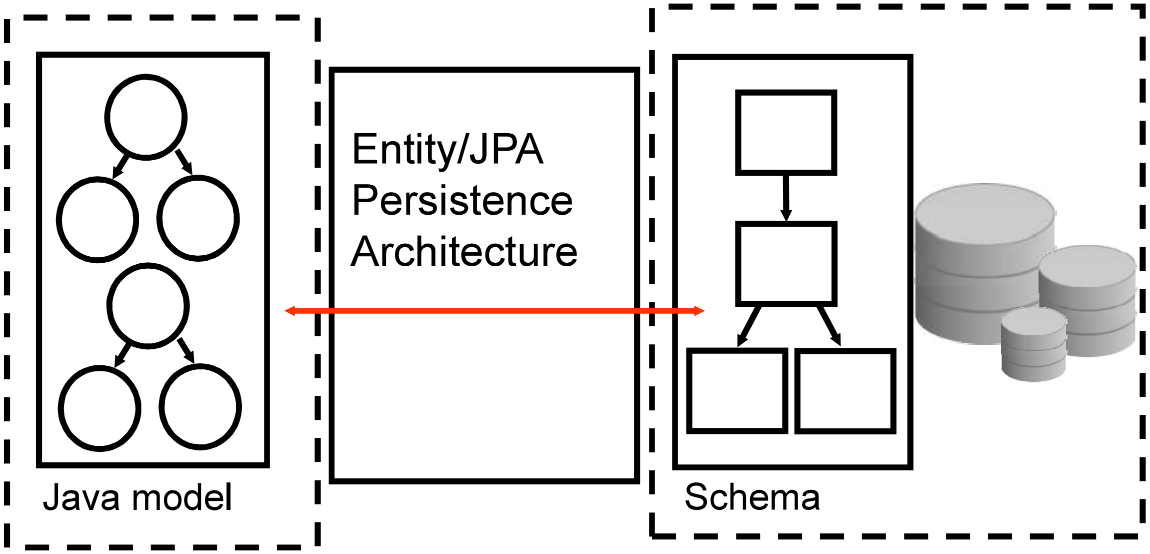
The persistence layer provides mapping between objects and database tables.

•This layer:

- Enables portability across databases and schemas
- Supports read, write, and caching capabilities
- Protects developers from database issues
- Facilitates change and maintenance
- Should be used in any application that has an object model

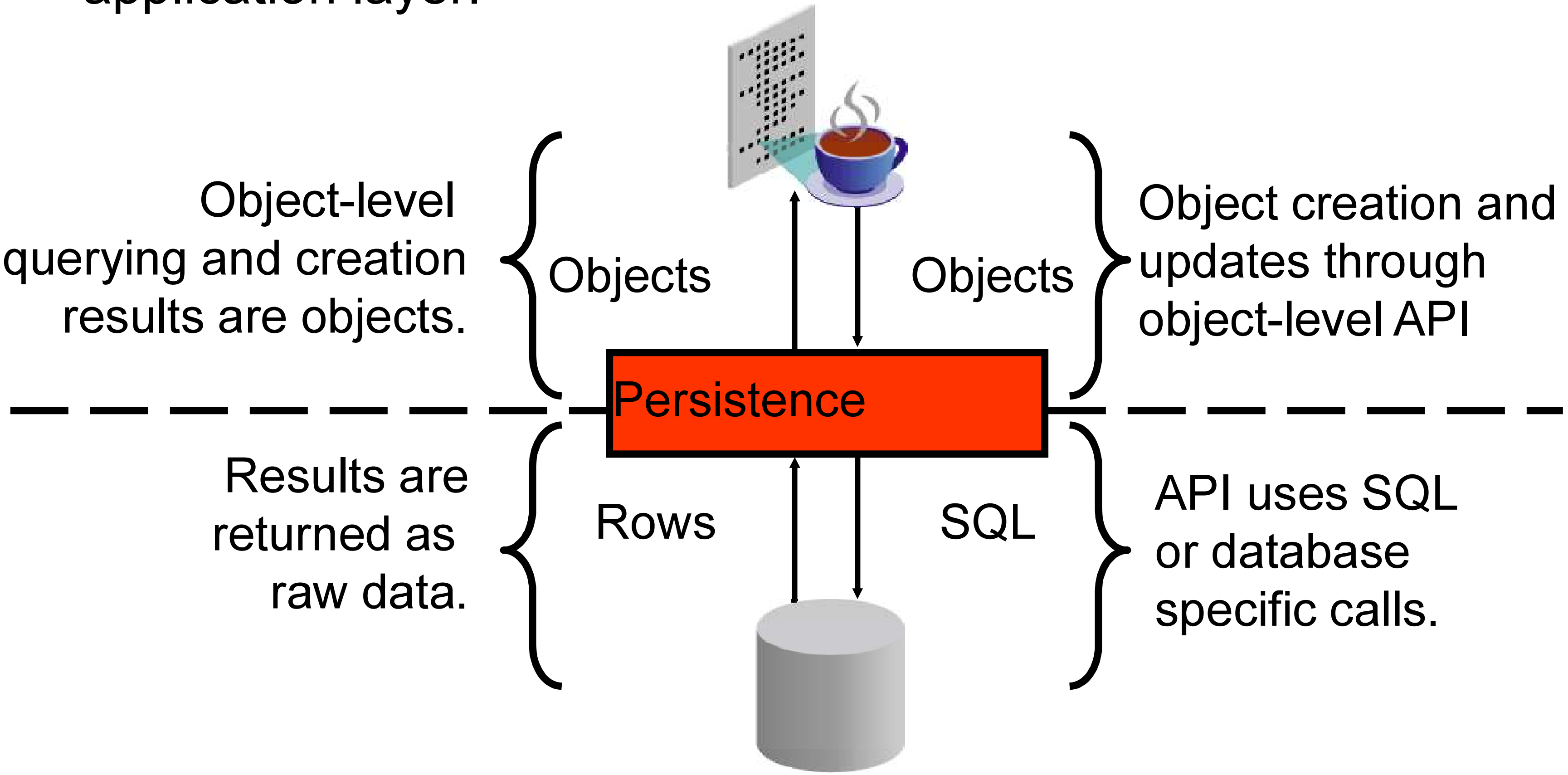
Persistence: Overview

- Mapping relational database objects to Java objects enables easy Java EE application development.
- Frameworks such as EJB 3.0 JPA provide this object-relational mapping.



Persistence Layer

A persistence layer abstracts persistence details from the application layer.



Object-level querying and creation results are objects.

Objects

Objects

Object creation and updates through object-level API

Persistence

Results are returned as raw data.

Rows

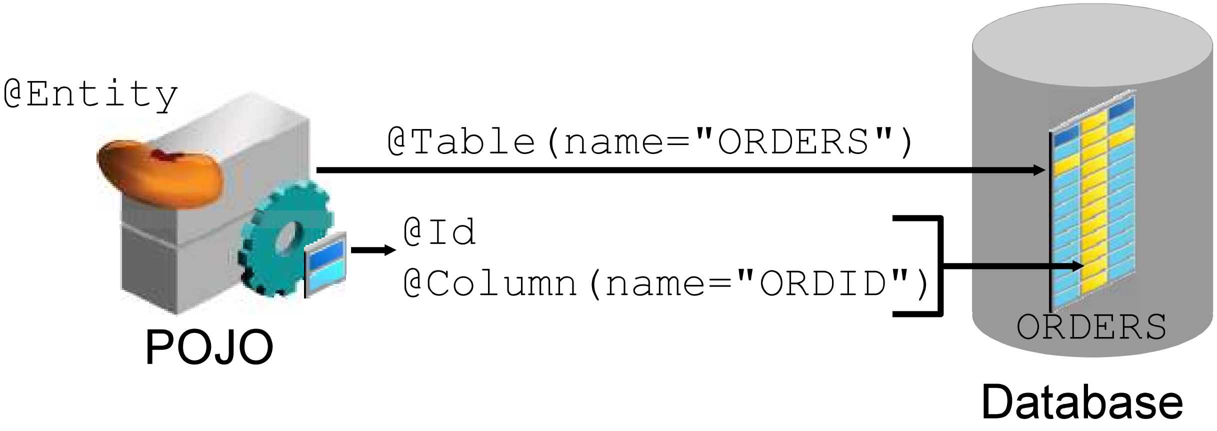
SQL

API uses SQL or database specific calls.

What Are JPA Entities?

A Java Persistence API (JPA) entity is:

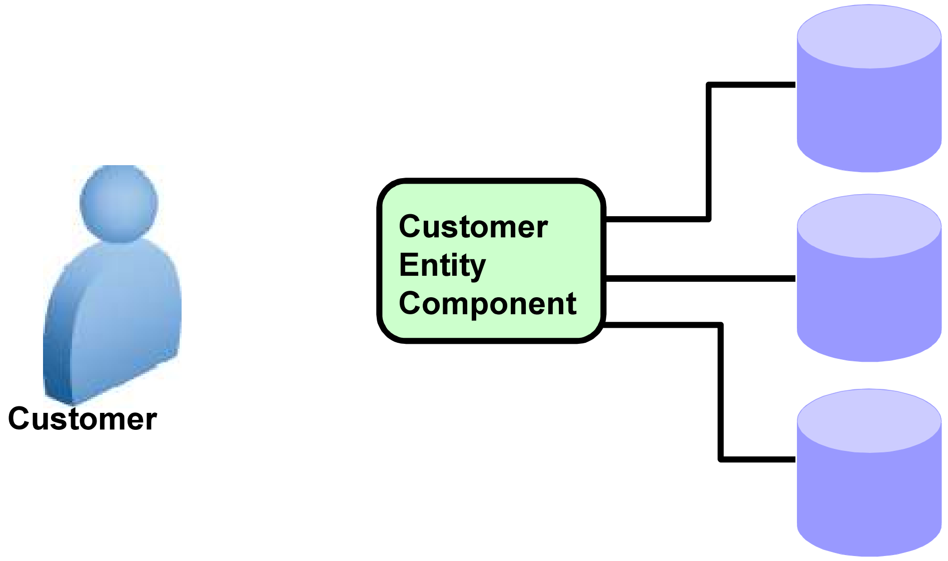
- A lightweight object that manages persistent data
- Defined as a Plain Old Java Object (POJO) marked with the `@Entity` annotation (no interfaces required)
- Not required to implement interfaces
- Mapped to a database by using annotations



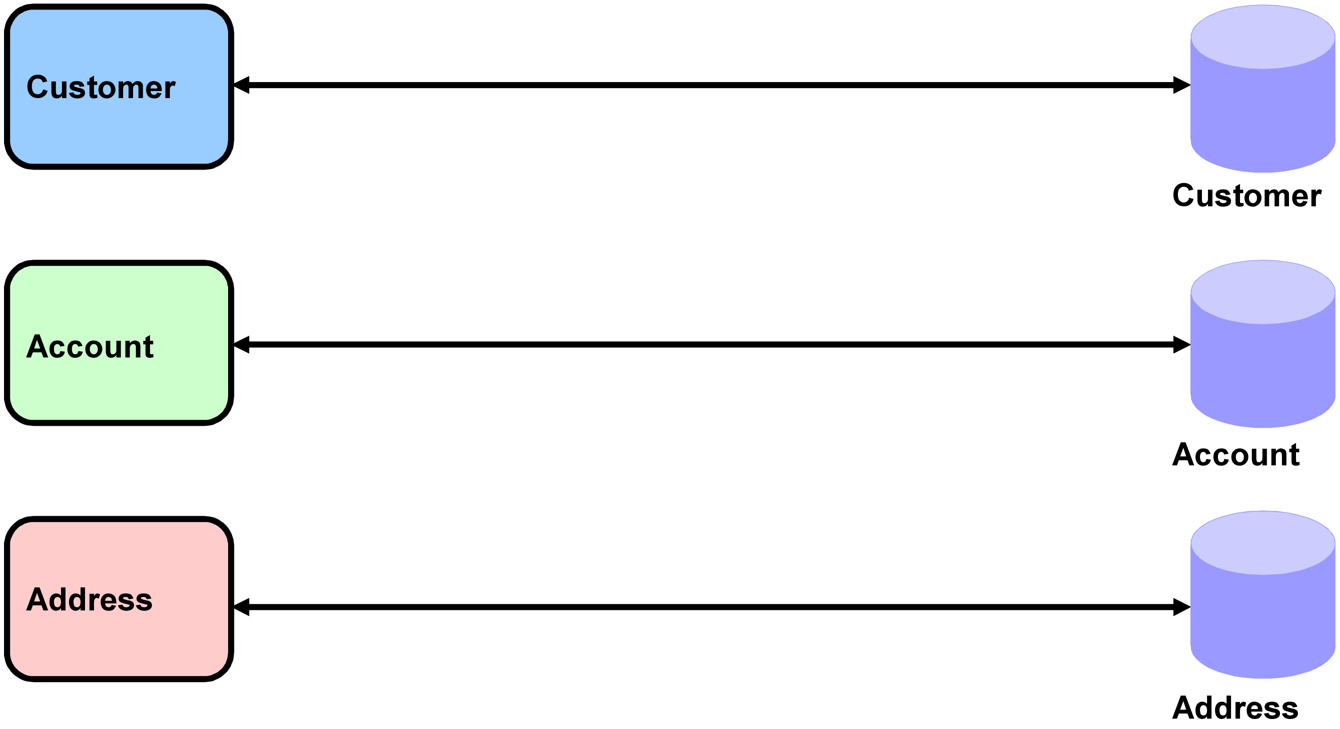
Object Relational Mapping

Object-relational mapping (ORM) software:

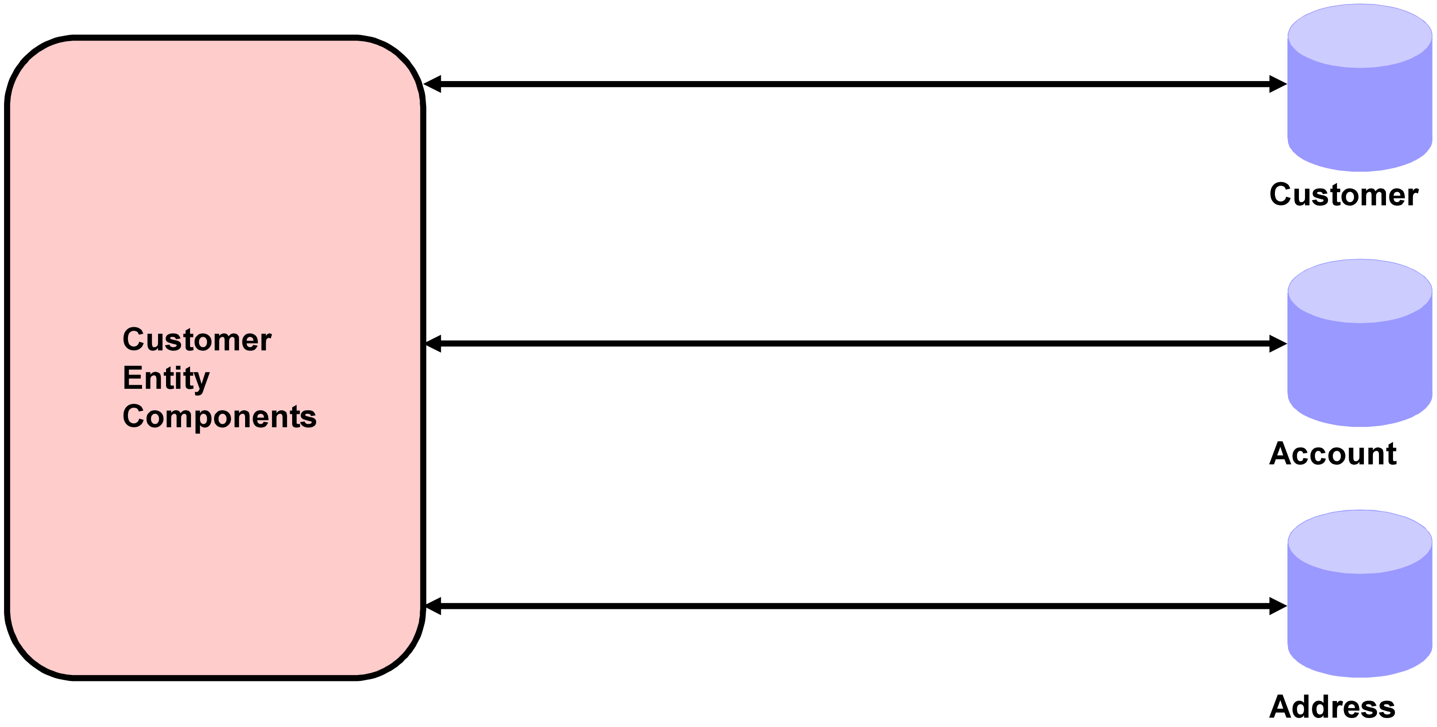
- Provides an object-oriented view of the database
- Examples include EclipseLink and Hibernate



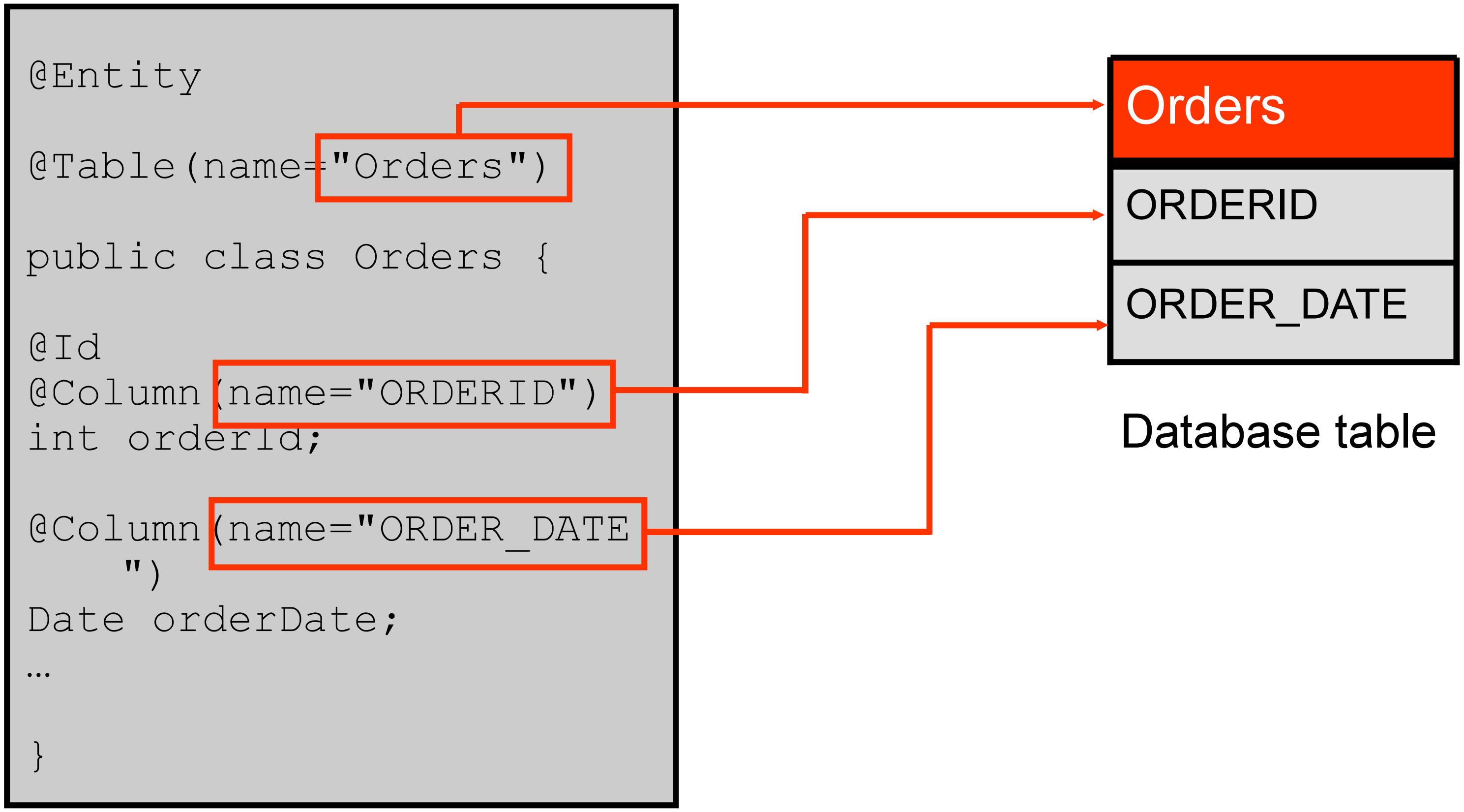
Normalized Data Mapping



Use of an Entity Component Across a Set of Database Tables



JPA Entities



```
@Entity

@Table(name="Orders")

public class Orders {

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

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

}
```

JPA entity

Orders

ORDERID

ORDER_DATE

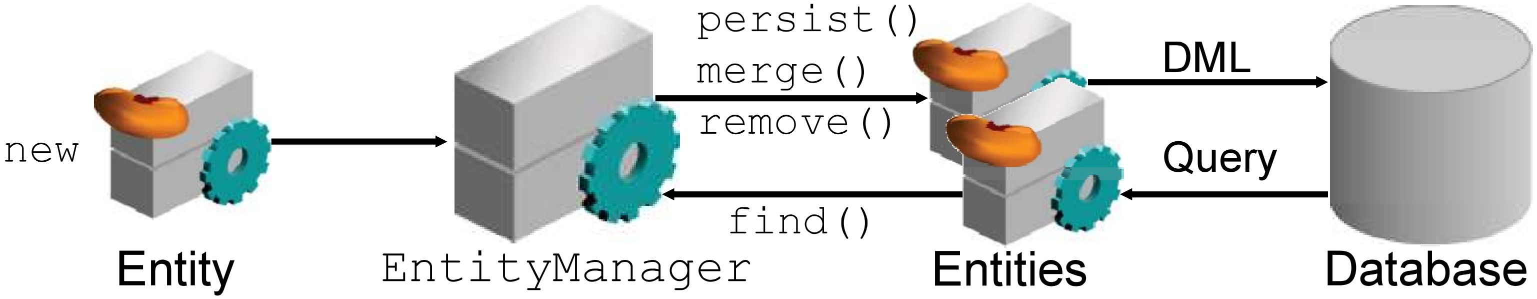
Database table

Domain Modeling with Entities

- Entities support standard object-oriented domain modeling techniques:
  - Inheritance
  - Encapsulation
  - Polymorphic relationships
- Entities can be created with the `new` operator.

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.



Declaring an Entity

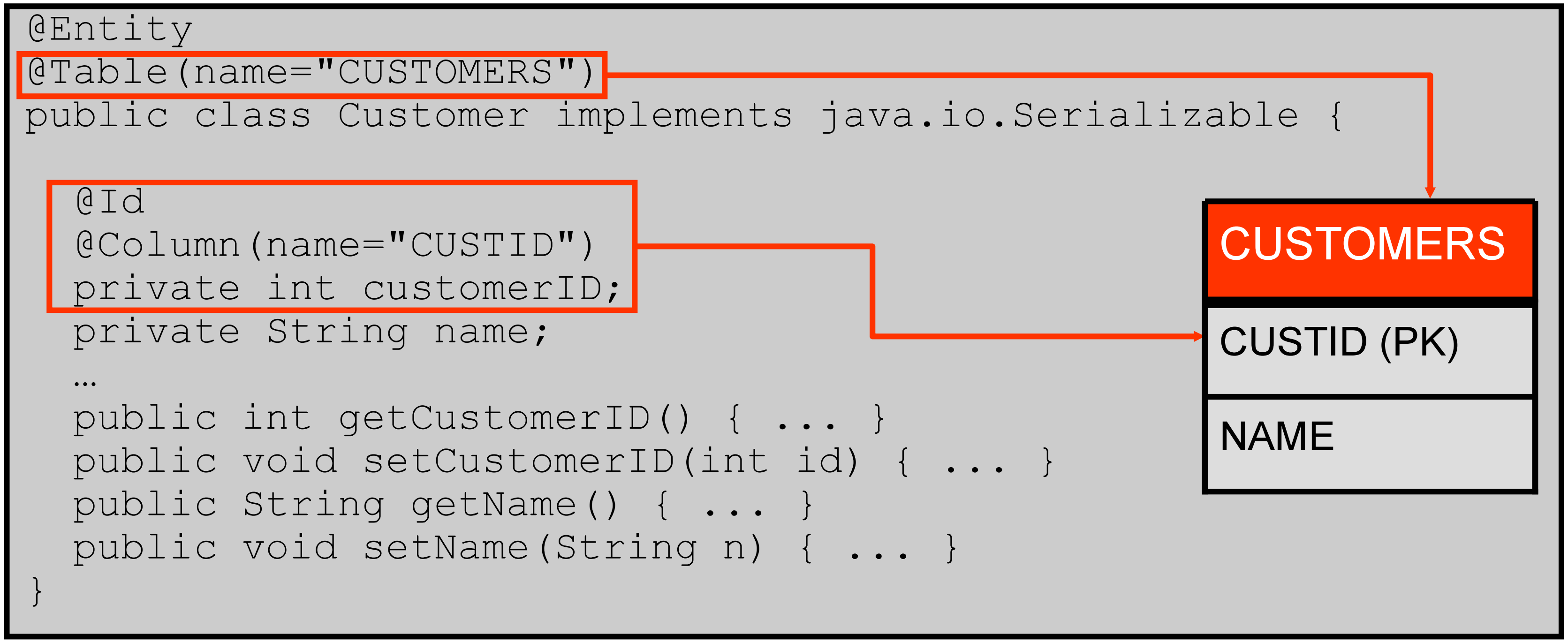
- Declare a new Java class with a no-arg constructor.
- Annotate it with `@Entity`.
- Add fields corresponding to each database column:
  - Add setter and getter methods.
  - Use the `@Id` annotation on the primary key getter method.

```
@Entity                             // annotation
public class Customer implements java.io.Serializable {
  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) { ... }
}
```

Quiz

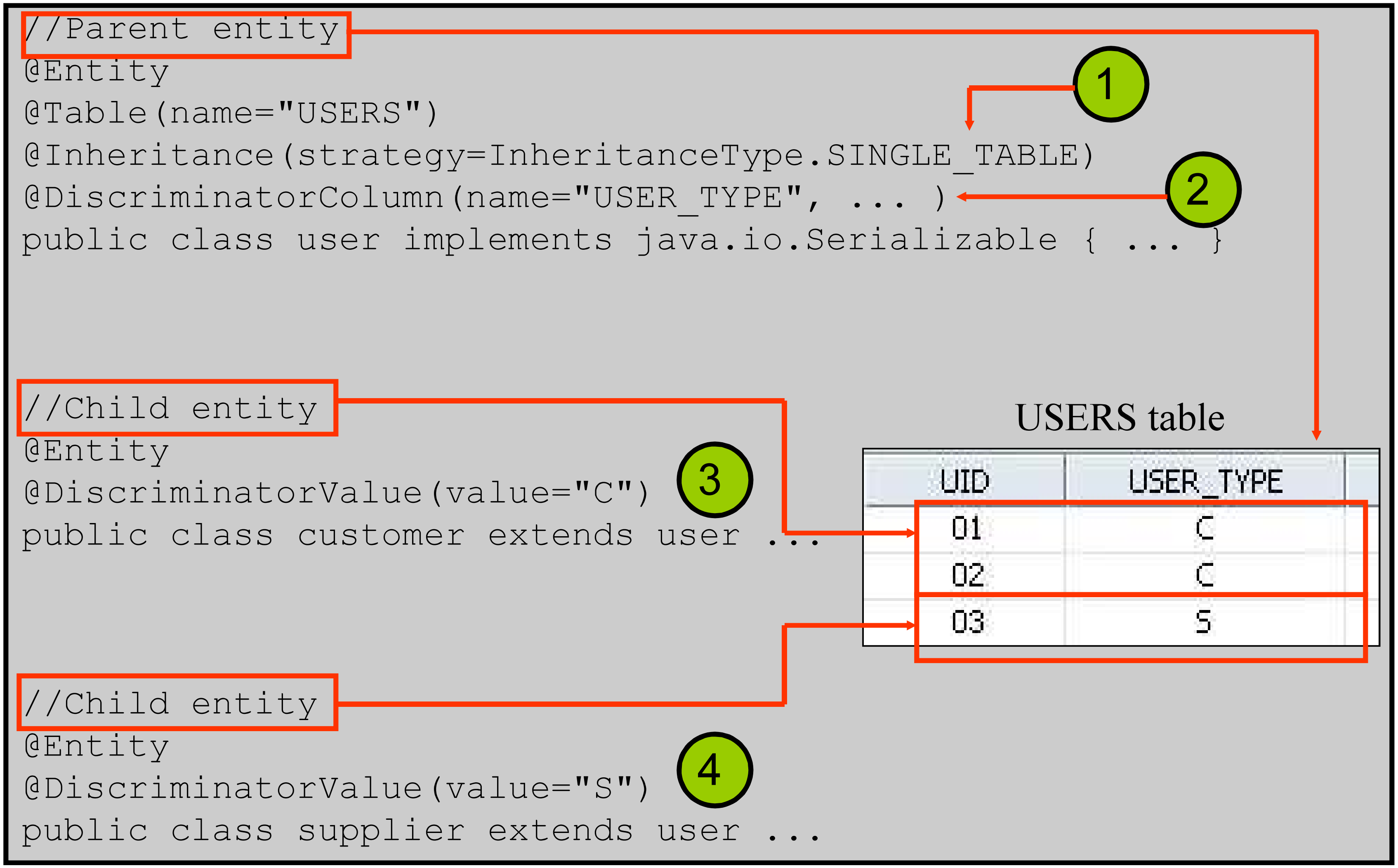
An entity is a lightweight persistence domain object that represents:

1. A relational database
2. A table in a relational database
3. Entity beans in EJB 2.x specification
4. Persistence data in a file

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.

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

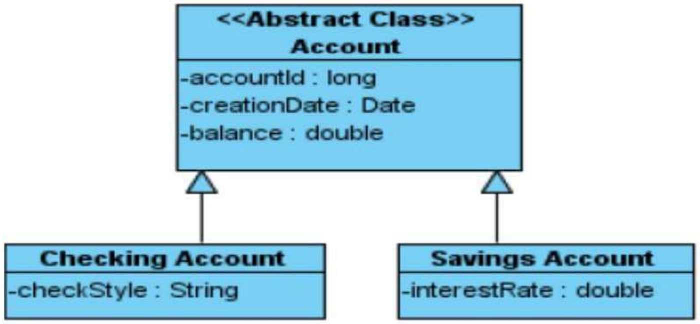
UID	USER_TYPE
01	C
02	C
03	S

Table-per-class-hierarchy

- **Database**
  - *One* database table for *all* subclasses
  - Denormalized table has columns for all attributes
- **Hibernate Mapping**
  - *Single mapping file* still based on superclass
  - Includes ‘subclass’ definitions for inherited classes
  - Use ‘discriminator’ column/field to identity concrete type

Table-per-class-hierarchy

- One table for all inherited classes

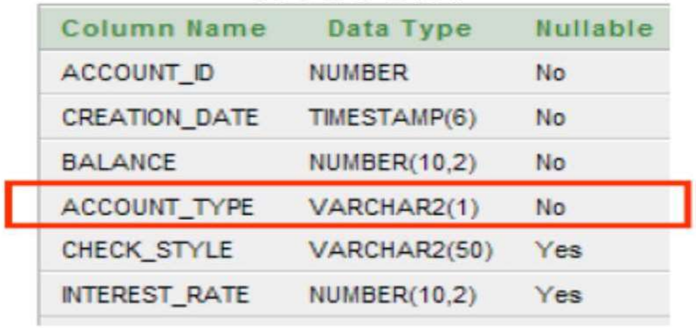


<<Abstract Class>>
Account
-accountId : long
-creationDate : Date
-balance : double

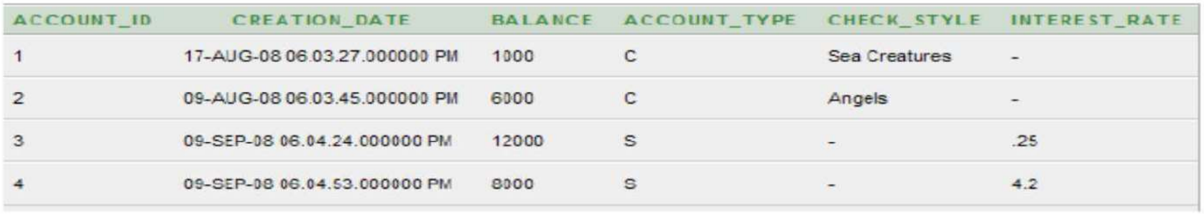
Checking Account
-checkStyle : String

Savings Account
-interestRate : double

ACCOUNT

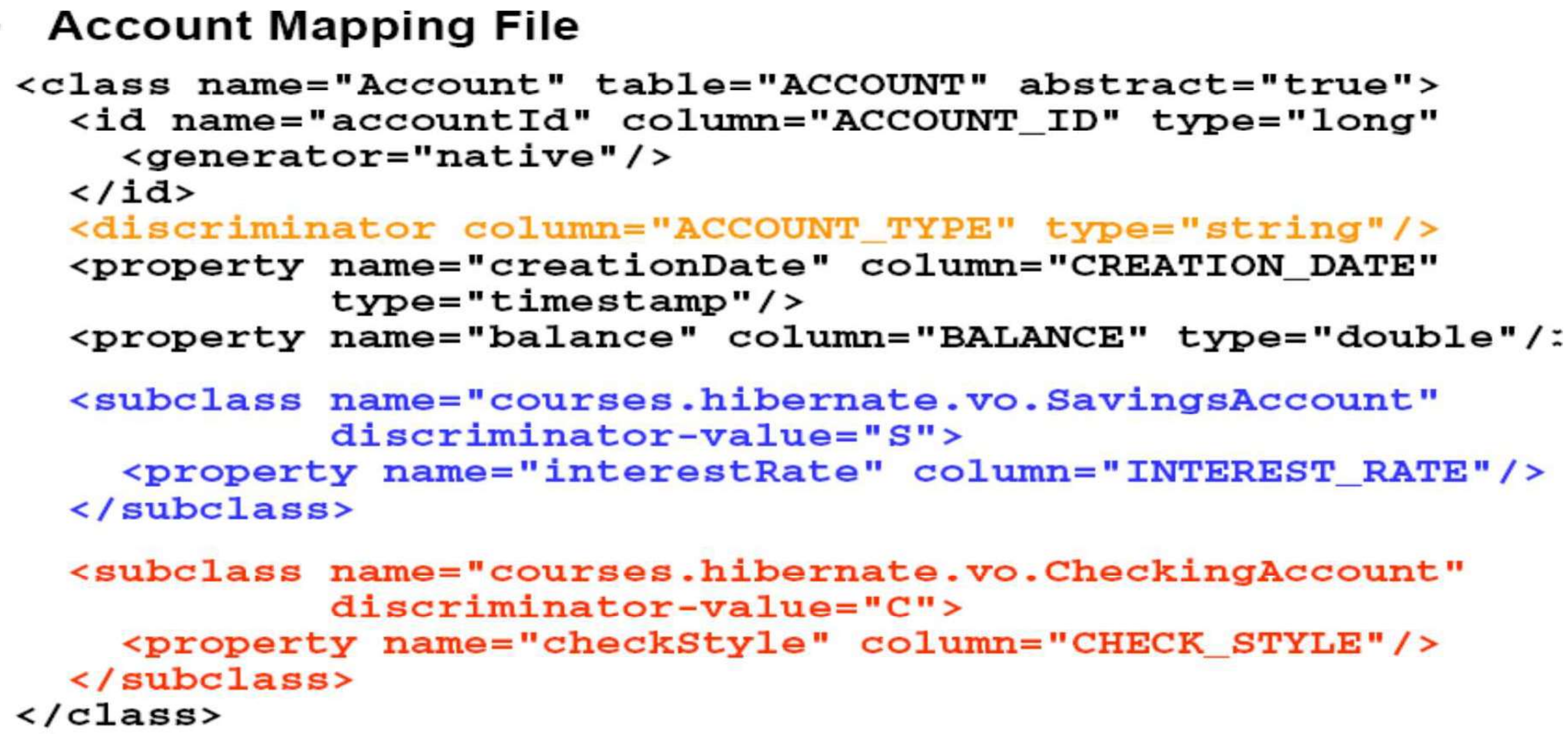


Column Name	Data Type	Nullable
ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No
ACCOUNT_TYPE	VARCHAR2(1)	No
CHECK_STYLE	VARCHAR2(50)	Yes
INTEREST_RATE	NUMBER(10,2)	Yes



ACCOUNT_ID	CREATION_DATE	BALANCE	ACCOUNT_TYPE	CHECK_STYLE	INTEREST_RATE
1	17-AUG-08 06.03.27.000000 PM	1000	C	Sea Creatures	-
2	09-AUG-08 06.03.45.000000 PM	6000	C	Angels	-
3	09-SEP-08 06.04.24.000000 PM	12000	S	-	.25
4	09-SEP-08 06.04.53.000000 PM	8000	S	-	4.2

Table-per-class-hierarchy



Account Mapping File

```
<class name="Account" table="ACCOUNT" abstract="true">
  <id name="accountId" column="ACCOUNT_ID" type="long"
    <generator="native"/>
  </id>
  <discriminator column="ACCOUNT_TYPE" type="string"/>
  <property name="creationDate" column="CREATION_DATE"
            type="timestamp"/>
  <property name="balance" column="BALANCE" type="double"/:

  <subclass name="courses.hibernate.vo.SavingsAccount"
            discriminator-value="S">
    <property name="interestRate" column="INTEREST_RATE"/>
  </subclass>

  <subclass name="courses.hibernate.vo.CheckingAccount"
            discriminator-value="C">
    <property name="checkStyle" column="CHECK_STYLE"/>
  </subclass>
</class>
```

Table-per-class-hierarchy

- **Advantages**
  - Simple
  - Fast reads/writes, even across types
- **Disadvantages**
  - Lots of nullable columns
    - Possible data integrity concern
  - Denormalized table generally considered bad database design

Table-per-subclass

- **Database**
  - *One* database table for the superclass ***AND*** one *per* subclass
    - Shared columns in superclass table
    - Subclass tables have their object-specific columns
- **Hibernate Mapping File**
  - *Single mapping file* based on the superclass
  - Includes ‘joined-subclass’ definitions for inherited classes

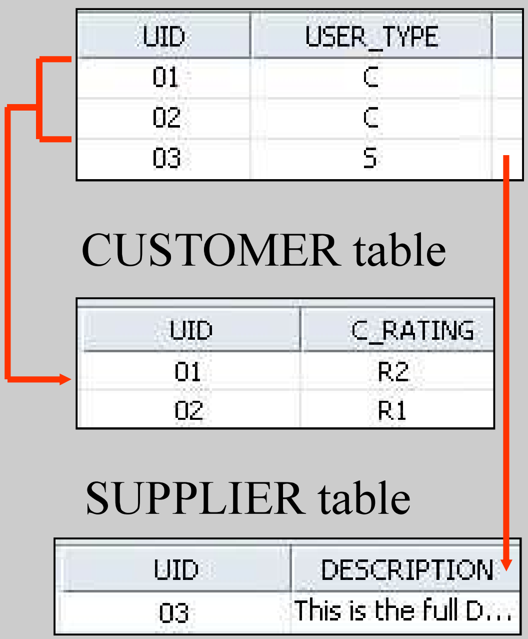
Joined-Tables Strategy

```
//Parent entity
@Entity
@Table(name="USERS")
@Inheritance(strategy=InheritanceType.JOINED)   1
@DiscriminatorColumn(name="USER_TYPE", ... )
public abstract class user ...

//Child entity
@Entity
@Table(name="CUSTOMER")
@DiscriminatorValue(value="C")
@PrimaryKeyJoinColumn(name="UID")   2
public class customer extends user ...

//Child entity
@Entity
@Table(name="SUPPLIER")
@DiscriminatorValue(value="S")
@PrimaryKeyJoinColumn(name="UID")   3
public class supplier extends user ...
```

USERS table

CUSTOMER table

SUPPLIER table



USERS table

UID	USER_TYPE
01	C
02	C
03	S

CUSTOMER table

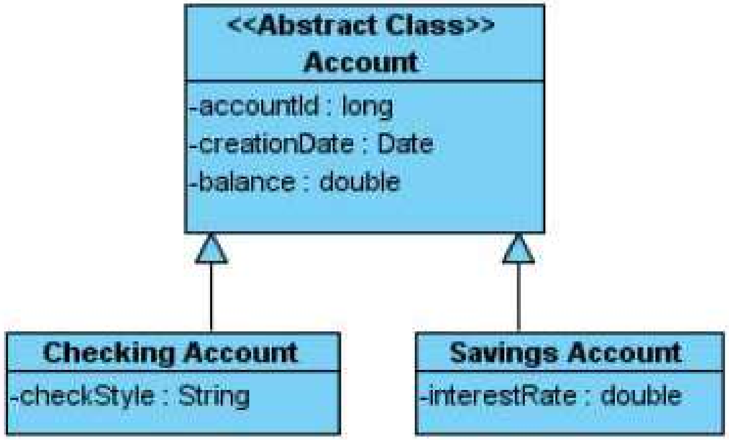
UID	C_RATING
01	R2
02	R1

SUPPLIER table

UID	DESCRIPTION
03	This is the full D...

Table-per-subclass

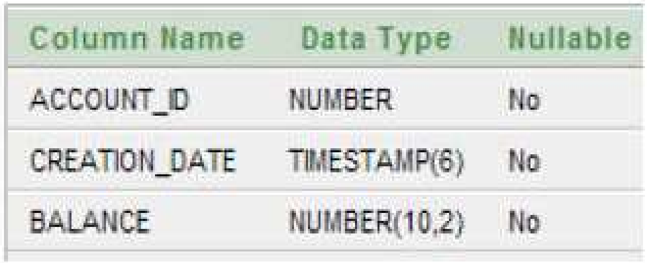
- **Every class that has persistent properties has its own table**
  - Each table contains a primary key, and non-inherited properties
  - Inheritance is realized through foreign keys



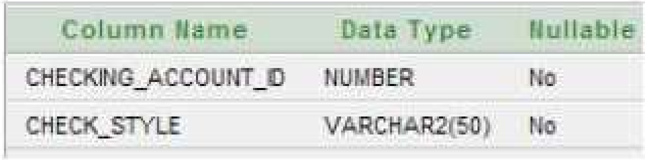
<<Abstract Class>>
Account
-accountId : long
-creationDate : Date
-balance : double

Checking Account
-checkStyle : String

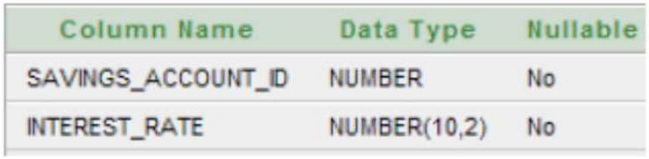
Savings Account
-interestRate : double



Column Name	Data Type	Nullable
ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No

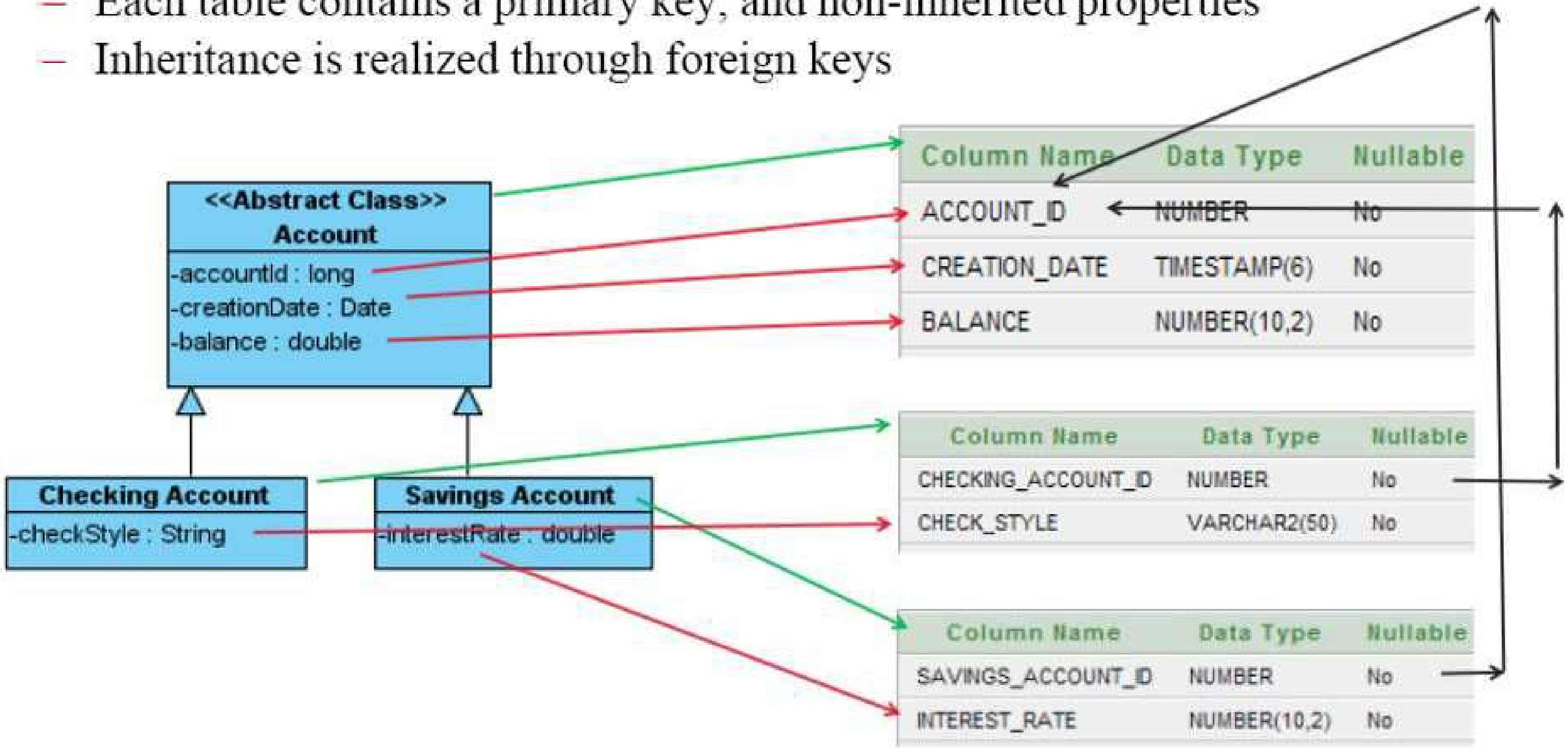


Column Name	Data Type	Nullable
CHECKING_ACCOUNT_ID	NUMBER	No
CHECK_STYLE	VARCHAR2(50)	No



Column Name	Data Type	Nullable
SAVINGS_ACCOUNT_ID	NUMBER	No
INTEREST_RATE	NUMBER(10,2)	No

- **Every class that has persistent properties has its own table**
  - Each table contains a primary key, and non-inherited properties
  - Inheritance is realized through foreign keys

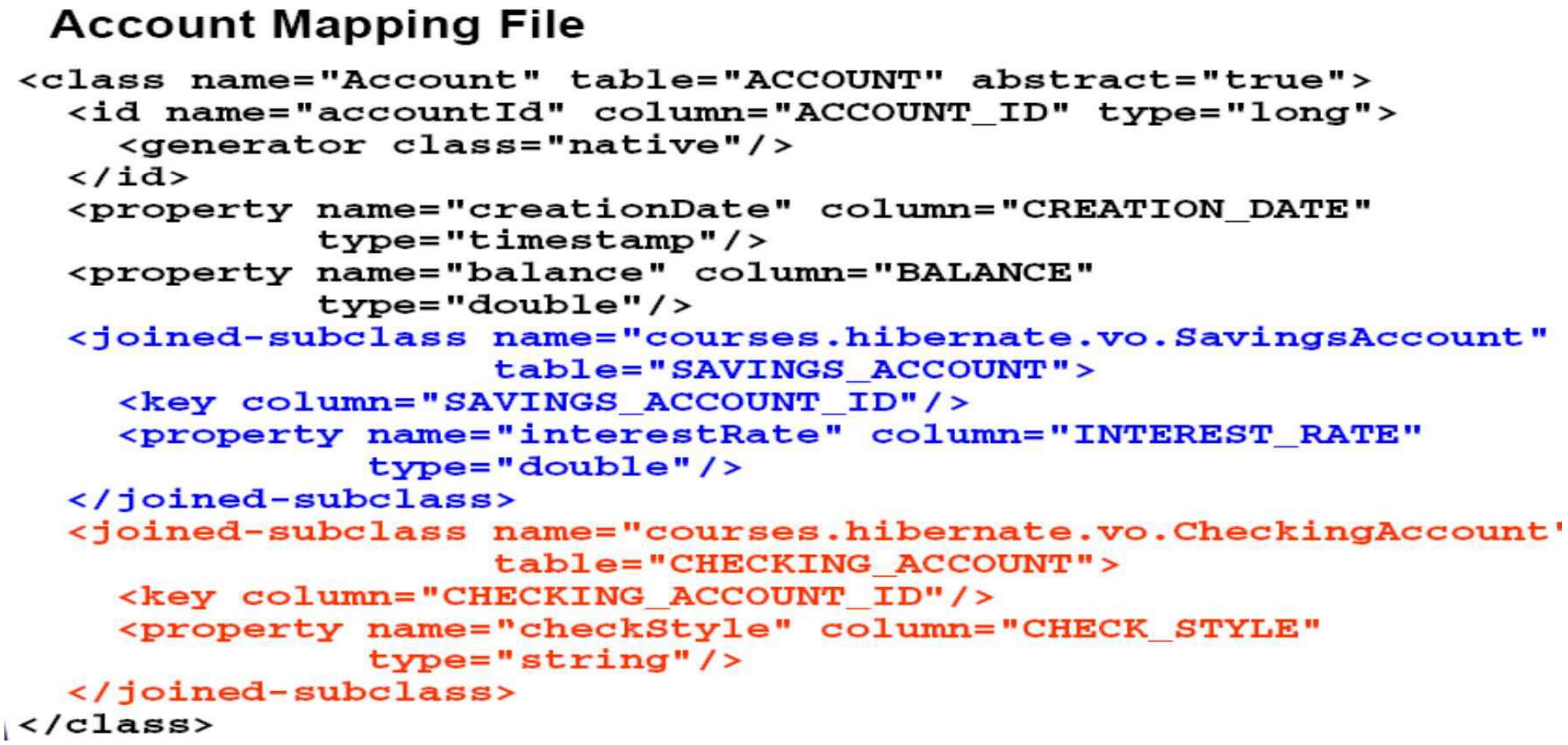


Column Name	Data Type	Nullable
ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No

Column Name	Data Type	Nullable
CHECKING_ACCOUNT_ID	NUMBER	No
CHECK_STYLE	VARCHAR2(50)	No

Column Name	Data Type	Nullable
SAVINGS_ACCOUNT_ID	NUMBER	No
INTEREST_RATE	NUMBER(10,2)	No

Table-per-subclass



Account Mapping File

```
<class name="Account" table="ACCOUNT" abstract="true">
  <id name="accountId" column="ACCOUNT_ID" type="long">
    <generator class="native"/>
  </id>
  <property name="creationDate" column="CREATION_DATE"
            type="timestamp"/>
  <property name="balance" column="BALANCE"
            type="double"/>
  <joined-subclass name="courses.hibernate.vo.SavingsAccount"
                   table="SAVINGS_ACCOUNT">
    <key column="SAVINGS_ACCOUNT_ID"/>
    <property name="interestRate" column="INTEREST_RATE"
              type="double"/>
  </joined-subclass>
  <joined-subclass name="courses.hibernate.vo.CheckingAccount'
                   table="CHECKING_ACCOUNT">
    <key column="CHECKING_ACCOUNT_ID"/>
    <property name="checkStyle" column="CHECK_STYLE"
              type="string"/>
  </joined-subclass>
</class>
```

Table-per-subclass

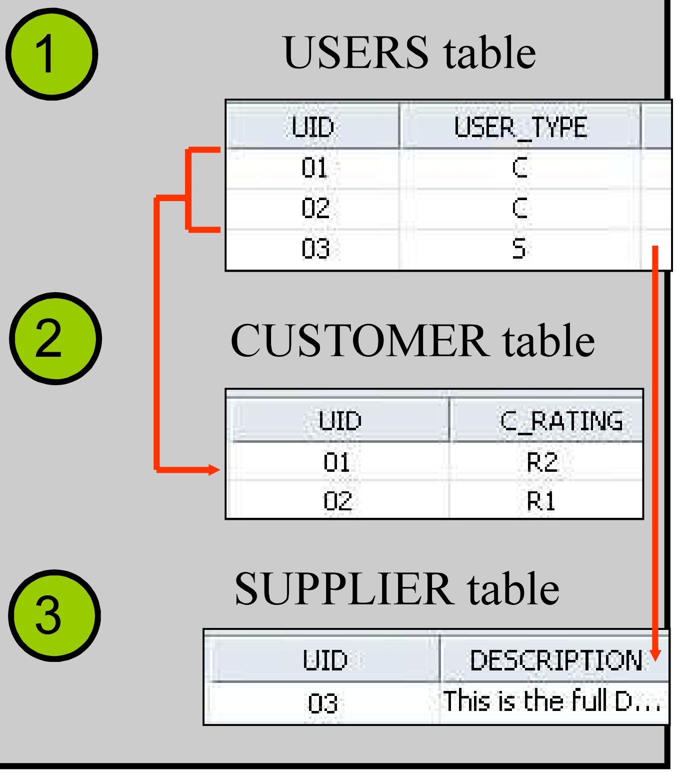
- **Advantages**
  - Normalized schema
    - Schema evolution and integrity are straight forward
  - Reduced number of SQL statements produced
    - Hibernate uses inner joins for subclass queries.
- **Disadvantages**
  - Can have poor performance for complex systems

When to use Which

- Leave implicit polymorphism for queries against interfaces *(based on behavior, not different attributes)*
- If you rarely require polymorphic queries, lean towards table-per-concrete-class.
- If polymorphic behavior is required, AND subclasses have only a few distinct properties, try table-perclass-hierarchy
- If polymorphic AND many distinct properties, look at table-per-subclass or table-per-concrete-class, weighing the cost of joins versus unions

Table Per Class Strategy

```
//Parent entity
@Entity
@Table(name="USERS")
@Inheritance(strategy=InheritanceType. TABLE_PER_CLASS)
@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...

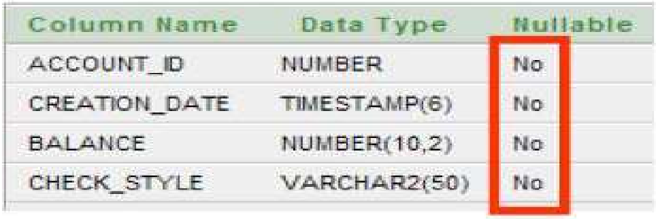
Table-per-concrete class

- **Database**
  - *One* database table *per* concrete class
- **Hibernate Mapping**
  - *Single mapping file*
    - Based on superclass
  - Includes '**union-subclass**' definitions for inherited classes

Table-per-concrete class

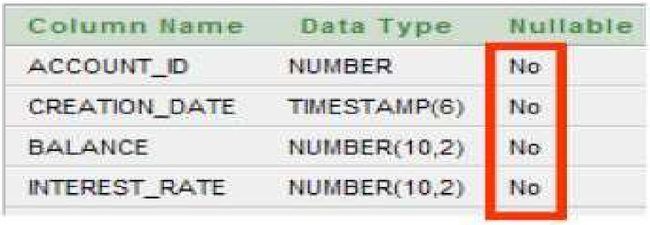
- One table per concrete class

CHECKING_ACCOUNT



Column Name	Data Type	Nullable
ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No
CHECK_STYLE	VARCHAR2(50)	No

SAVINGS_ACCOUNT



Column Name	Data Type	Nullable
ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No
INTEREST_RATE	NUMBER(10,2)	No

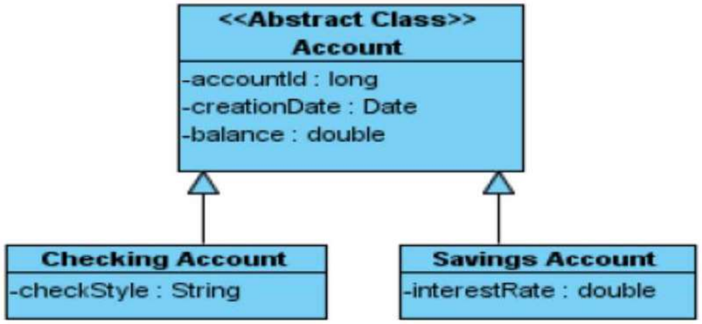
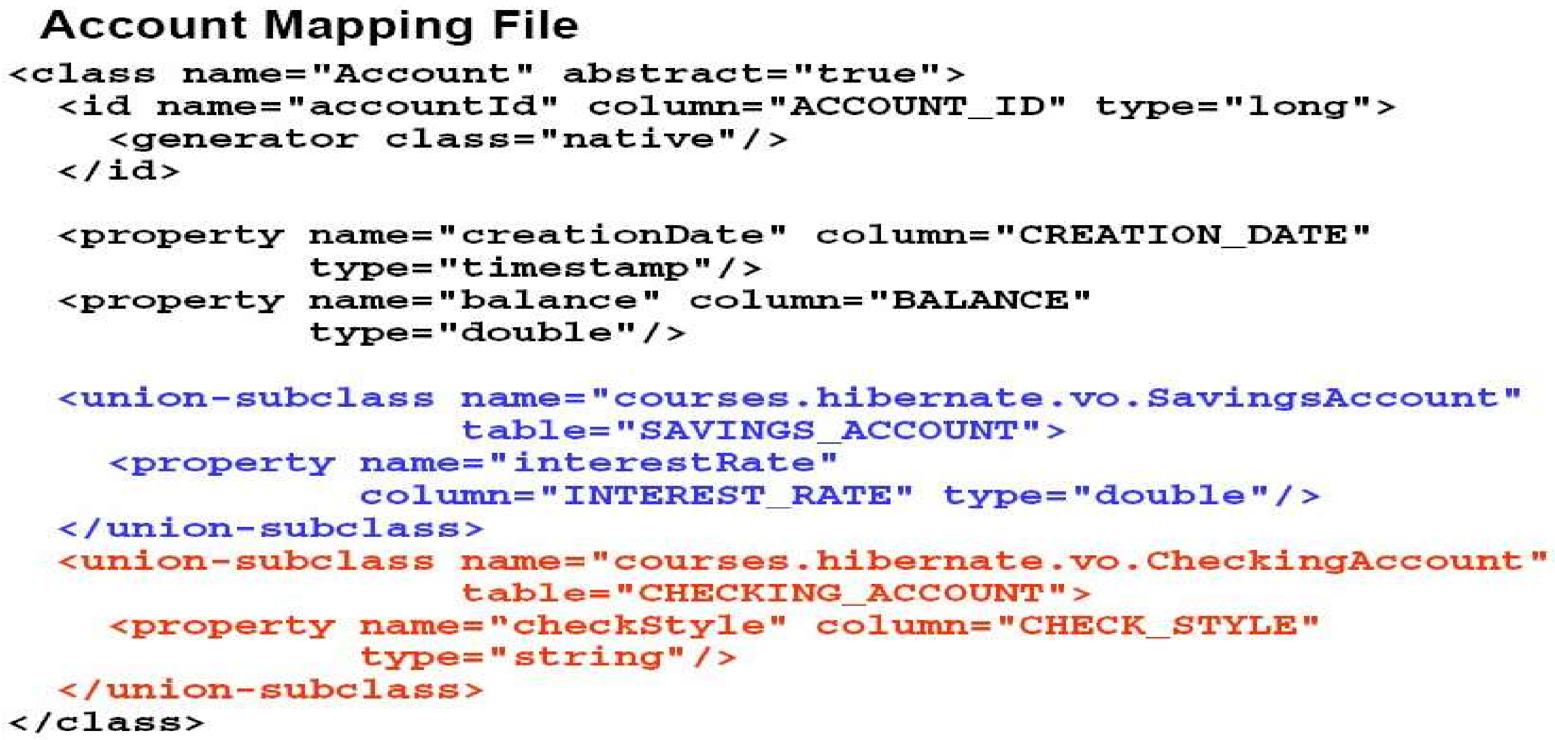


Table-per-concrete class



Account Mapping File

```
<class name="Account" abstract="true">
  <id name="accountId" column="ACCOUNT_ID" type="long">
    <generator class="native"/>
  </id>

  <property name="creationDate" column="CREATION_DATE"
            type="timestamp"/>
  <property name="balance" column="BALANCE"
            type="double"/>

  <union-subclass name="courses.hibernate.vo.SavingsAccount"
                  table="SAVINGS_ACCOUNT">
    <property name="interestRate"
              column="INTEREST_RATE" type="double"/>
  </union-subclass>
  <union-subclass name="courses.hibernate.vo.CheckingAccount"
                  table="CHECKING_ACCOUNT">
    <property name="checkStyle" column="CHECK_STYLE"
              type="string"/>
  </union-subclass>
</class>
```

Table-per-concrete class

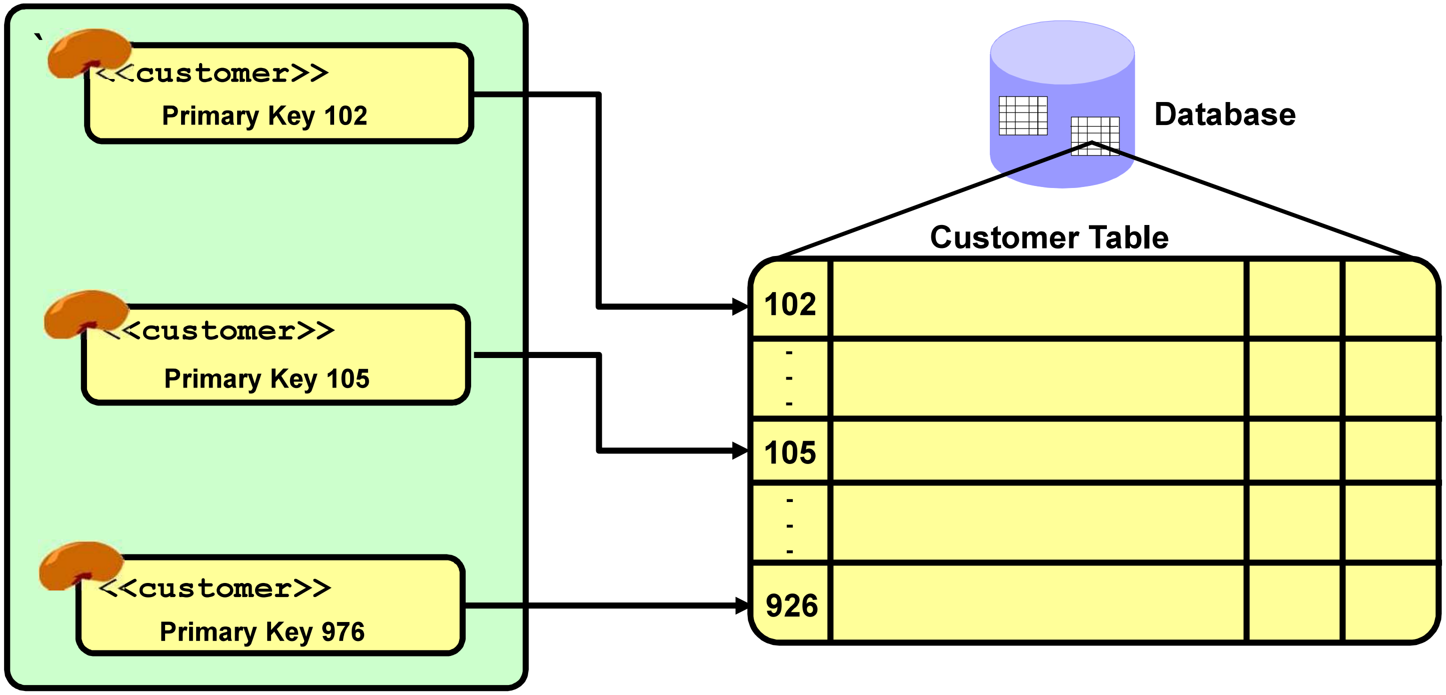
- **Advantages**
  - Shared mapping of common elements
    - Shared database id
  - Not a lot of nullable columns *(good for integrity)*
  - Queries against individual types are fast and simple
  - Less SQL statements generated with use of ‘Union’ for polymorphic queries
- **Disadvantages**
  - Still have difficulty with relationships
    - Foreign keying to two tables not possible

Specifying Entity Identity

•The identity of an entity can be specified by using:

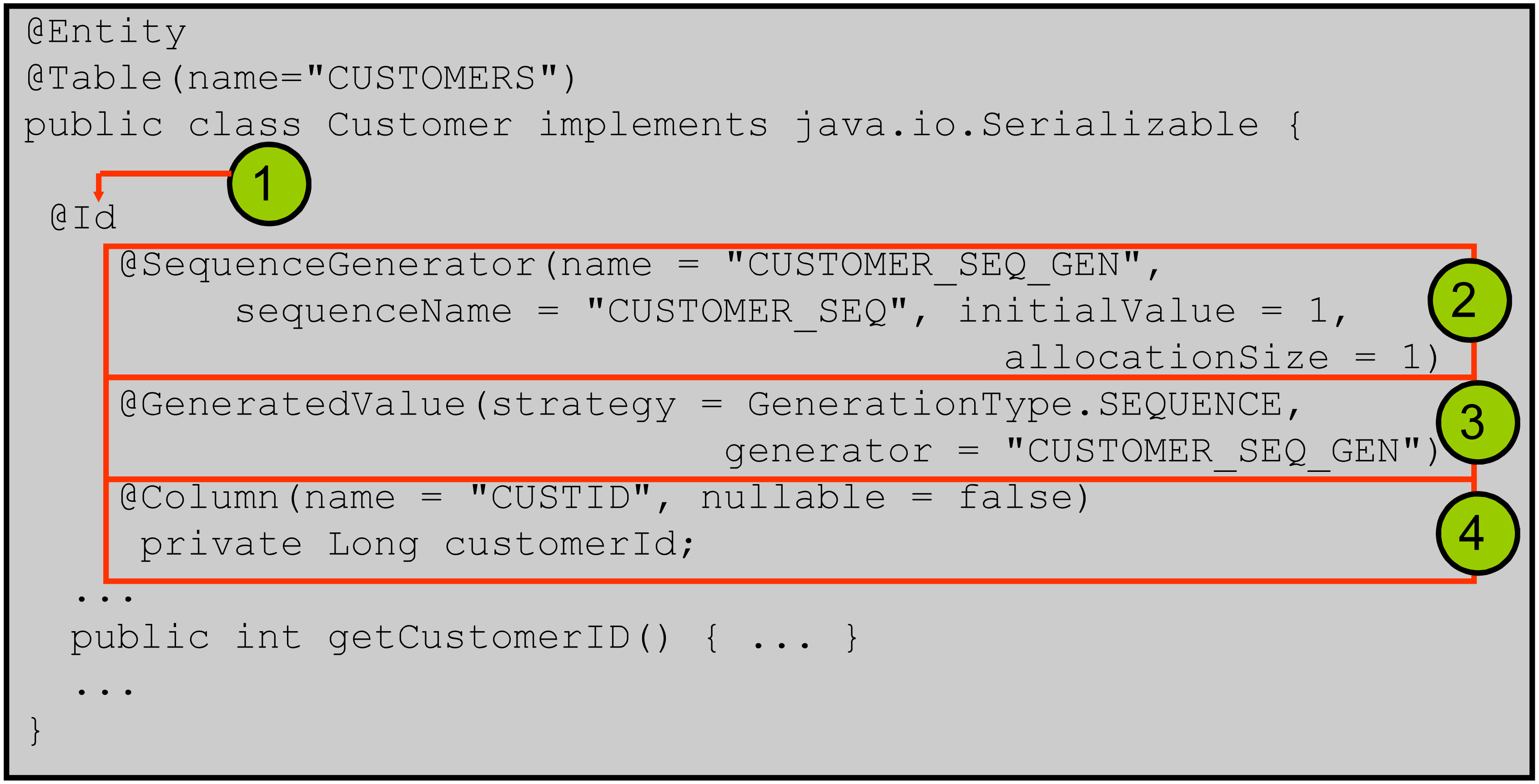
- The `@Id` annotation
- The `@IdClass` annotation

Entity Component Primary Key Association



Generating Primary Key Values

Use the `@GeneratedValue` annotation.

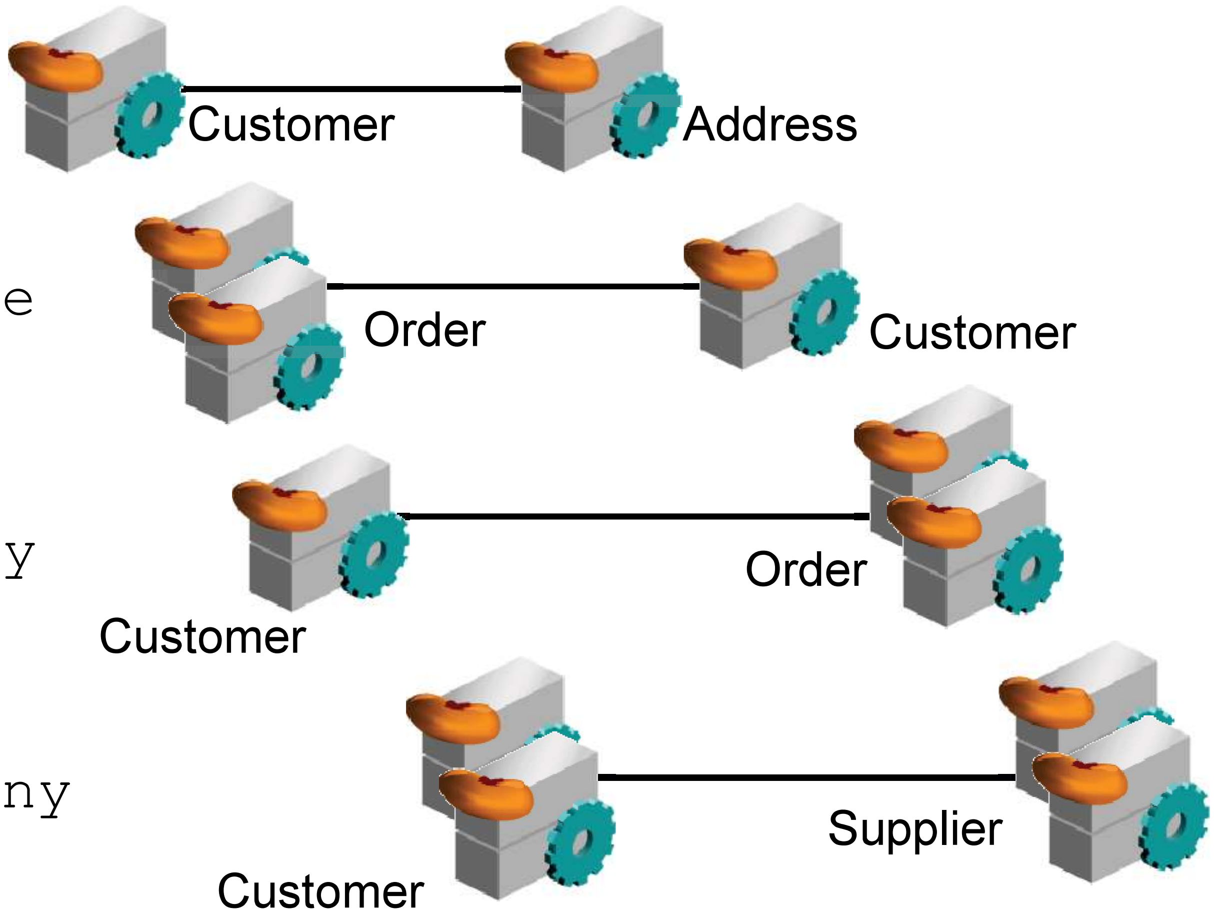


```
@Entity
@Table(name="CUSTOMERS")
public class Customer implements java.io.Serializable {

  @Id
    @SequenceGenerator(name = "CUSTOMER_SEQ_GEN",
         sequenceName = "CUSTOMER_SEQ", initialValue = 1,
                                      allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                              generator = "CUSTOMER_SEQ_GEN")
    @Column(name = "CUSTID", nullable = false)
     private Long customerId;
   ...
   public int getCustomerID() { ... }
   ...
}
```

Mapping Relationships Between Entities

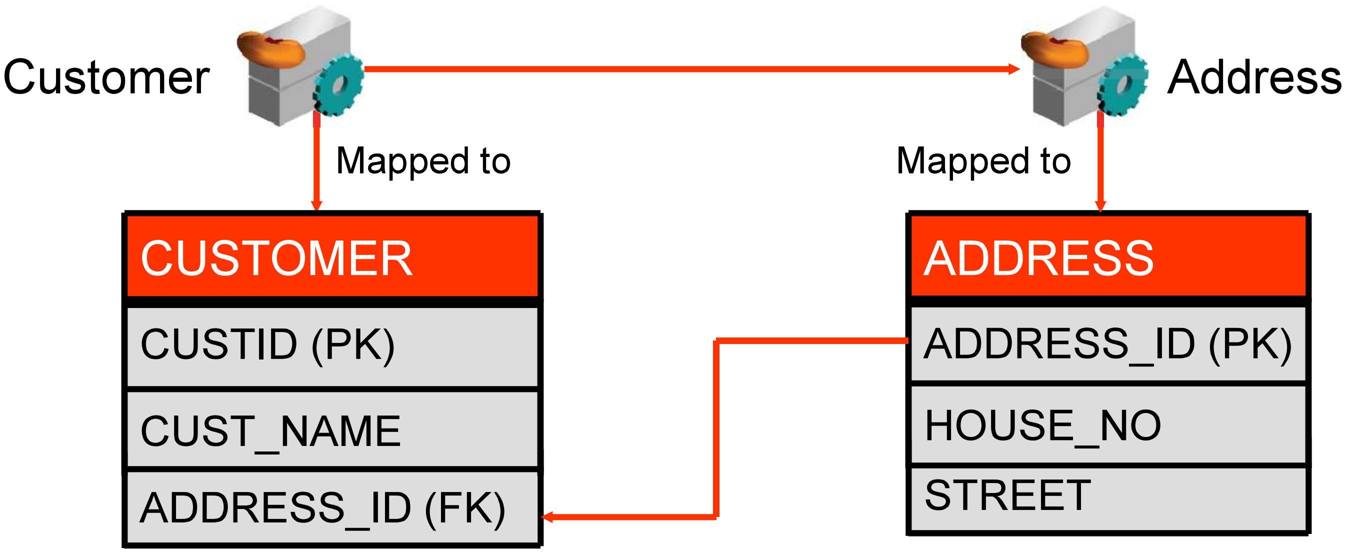
Annotations for entity relationships:

- `@OneToOne`
- `@ManyToOne`
- `@OneToMany`
- `@ManyToMany`



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



Customer

Address

Mapped to

Mapped to

CUSTOMER
CUSTID (PK)
CUST_NAME
ADDRESS_ID (FK)

ADDRESS
ADDRESS_ID (PK)
HOUSE_NO
STREET

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="MAILING_ADDRESS_REF",
                    referencedColumnName="ADDRESS_PK")
protected Address address;
...

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

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", updatable=false)
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;
...
```

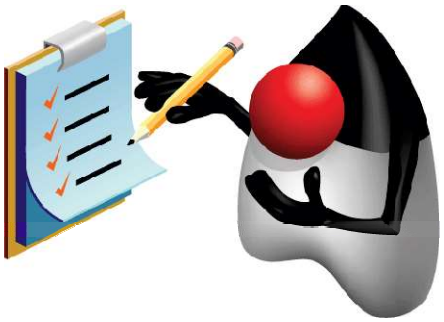
Managing Entities

- Entities are managed by using the `EntityManager` API.
- `EntityManager` performs the following tasks for the entities:
  - Implements the object-relational mapping between Java objects and database
  - Performs the CRUD operations for the entities
  - Manages the life cycle of the entities
  - Manages transactions

Summary

In this lesson, you should have learned how to:

- What are JPA Entities?
- Domain Modeling with JPA
- Creating an Entity (a POJO with annotations)
- Specifying Object Relational (OR) Mapping
- Mapping Relationships between Entities
- Inheritance Mapping Strategy (Singe Table, Joined Subclass)



Practice: Overview

These practice covers the following topics:

- Creating a simple entity bean by coding the bean
- Using the JDeveloper wizards to create a set of entity beans
- Creating and managing a session bean that provides client access to the entity beans
- Creating a test client to invoke the session bean