Session: 10

Advanced Persistence Concepts

Objectives

- Explain how relationships are managed in OOP
- Explain how relationships are managed in relational databases
- Explain cardinality and directionality in object relationships
- Explain how relationships are managed in JPA
- Describe annotations provided by JPA to create object relationships
- Describe the mapping of different aspects of database to the enterprise application
- Describe the different JPA strategies to map inheritance in relational databases
- Explain implementation of inheritance among the entities

Introduction

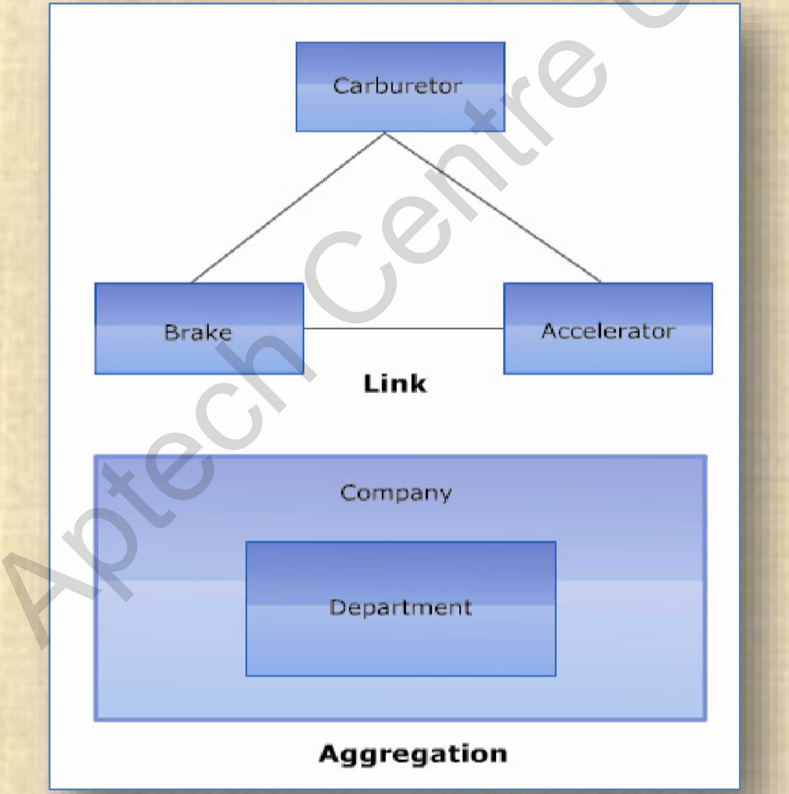
- ❑ Entity beans in an enterprise application usually relate to one another.
- ❑ For instance, the Student entity bean is related to the Teacher entity bean in an enterprise application because the students are taught by the teacher.
- ❑ Implementing the relationship between the student and teacher is handled differently by application developers and database designers.

Relationships in Object-Oriented Programming 1-4

- Objects interact with other objects to represent some concrete function.
- There are three types of association among the objects:
  - Association
  - Aggregation
  - Inheritance
- Relationships among objects describe how objects collaborate, to contribute to the behavior of the system.

Relationships in Object-Oriented Programming 2-4

- ❑ Nature of a relationship can be broadly classified as, link and aggregation.
- ❑ Following figure depicts relationships among objects:

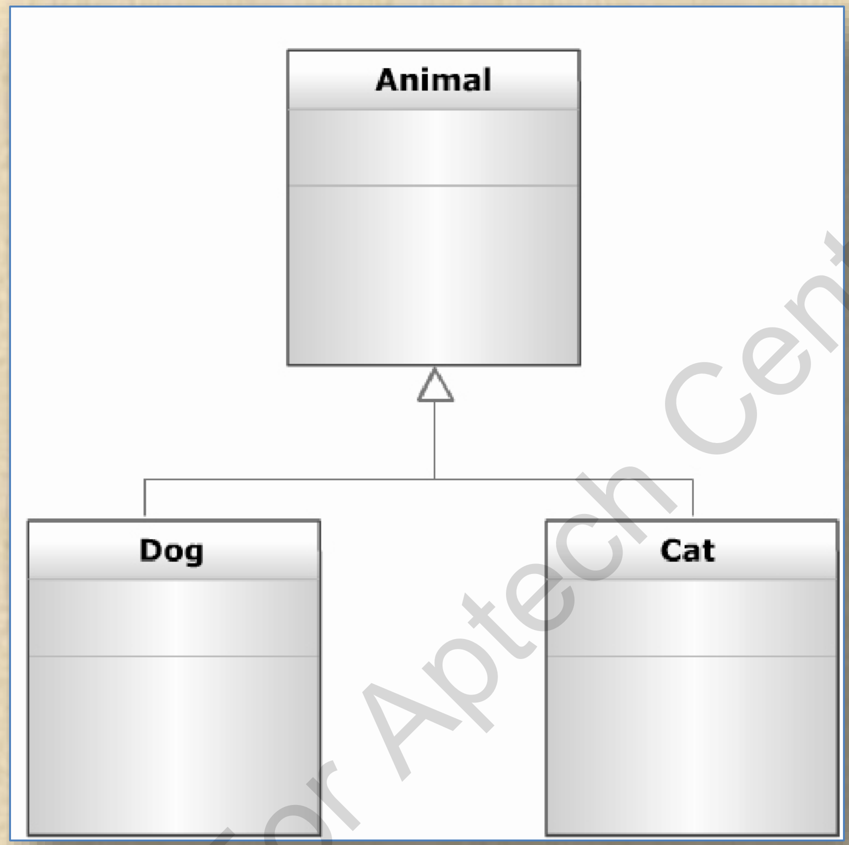


Relationships in Object-Oriented Programming 3-4

- ❑ In generalization, an object or subtype is dependent on another object or super type.
- ❑ Generalization relationship is used to reuse attributes, operations, and relationships present in the super type with one or more subtypes.
- ❑ The generalization relationship is also known as inheritance or '**is-a**' relationship.

Relationships in Object-Oriented Programming 4-4

- Following figure depicts inheritance:



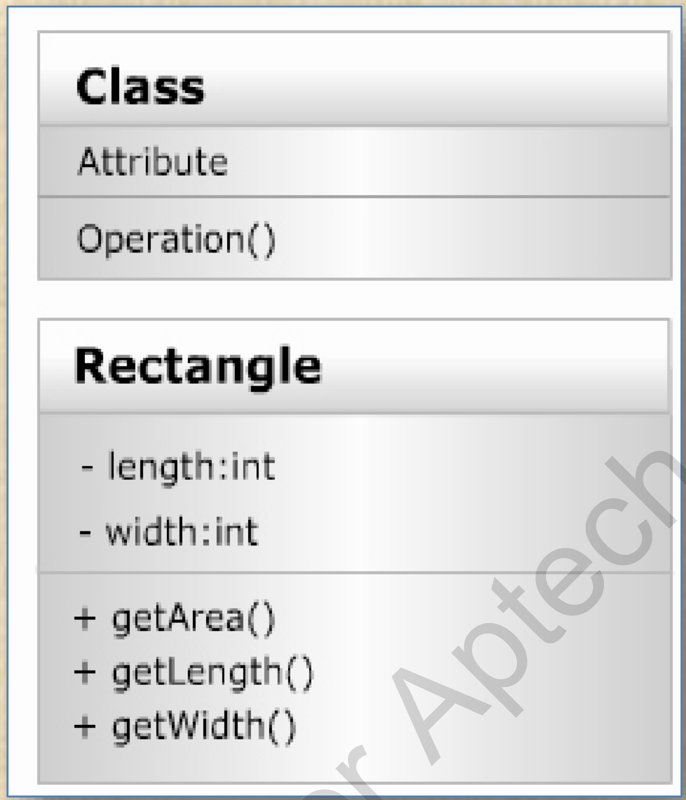
The UML graphical representation of Generalization is a hollow triangle shape on the super type end of the line that connects it to one or more subtypes.

Object Modeling 1-2

- ❑ Object modeling is widely done through class diagrams.
- ❑ UML diagrams are used to depict objects and relationships in an object model.
- ❑ Uses various notations to represent the following entities in diagrammatic form:
  - Classes
  - Attributes
  - List of operations
  - Access modifiers

Object Modeling 2-2

- ❑ Following figure depicts a class diagram:



Class

Attribute

Operation()

Rectangle

- length:int

- width:int

+ getArea()

+ getLength()

+ getWidth()

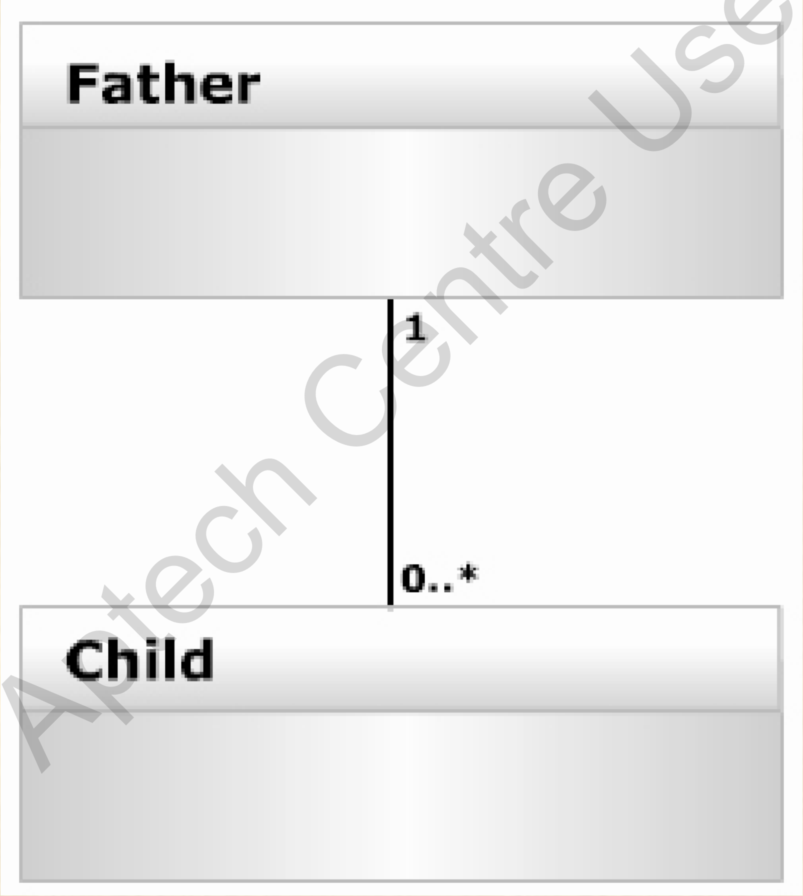
- The topmost section contains the name of the class.
- The middle section contains a list of attributes.
- The bottom section contains a list of operations.

Multiplicity in Relationships 1-2

- Relationships among objects are shown using connectors.
- Multiplicity in relationships is depicted using the following notations:
  - `0..1` No instances, or one instance
  - `1` Exactly one instance
  - `0..*` or `*` Zero or more instances
  - `1..*` At least one instance

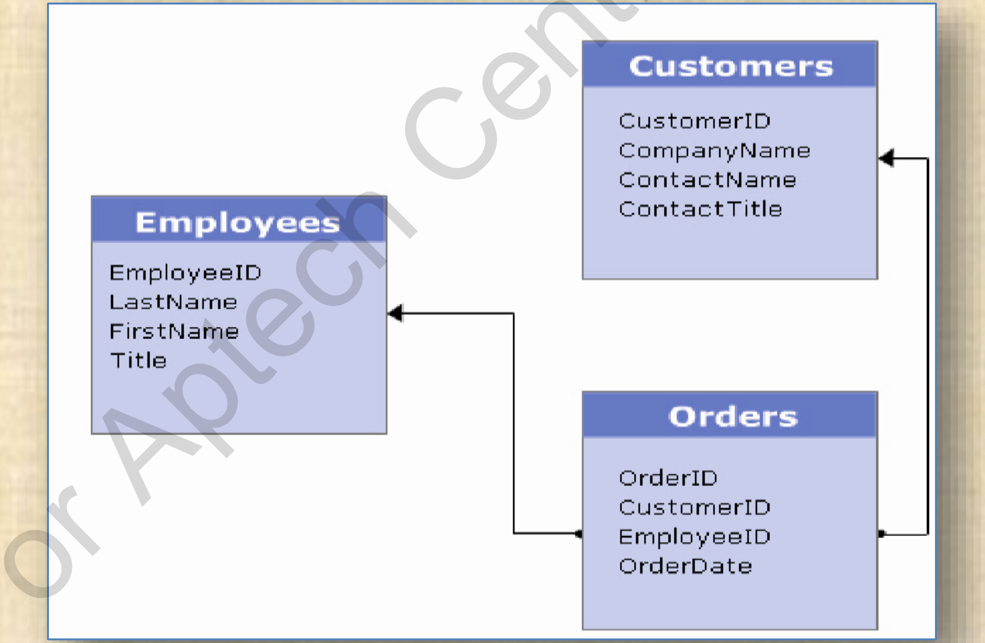
Multiplicity in Relationships 2-2

- Following figure depicts multiplicity in a relationship:



Relationships in Database 1-2

- A database schema describes the table structure, data types, and relations in a database.
- A database schema can also be a series of Structured Query Language (SQL) statements.
- Following is a relational database schema:



Relationships in Database 2-2

- Following is the terminology used when data is stored in a table:

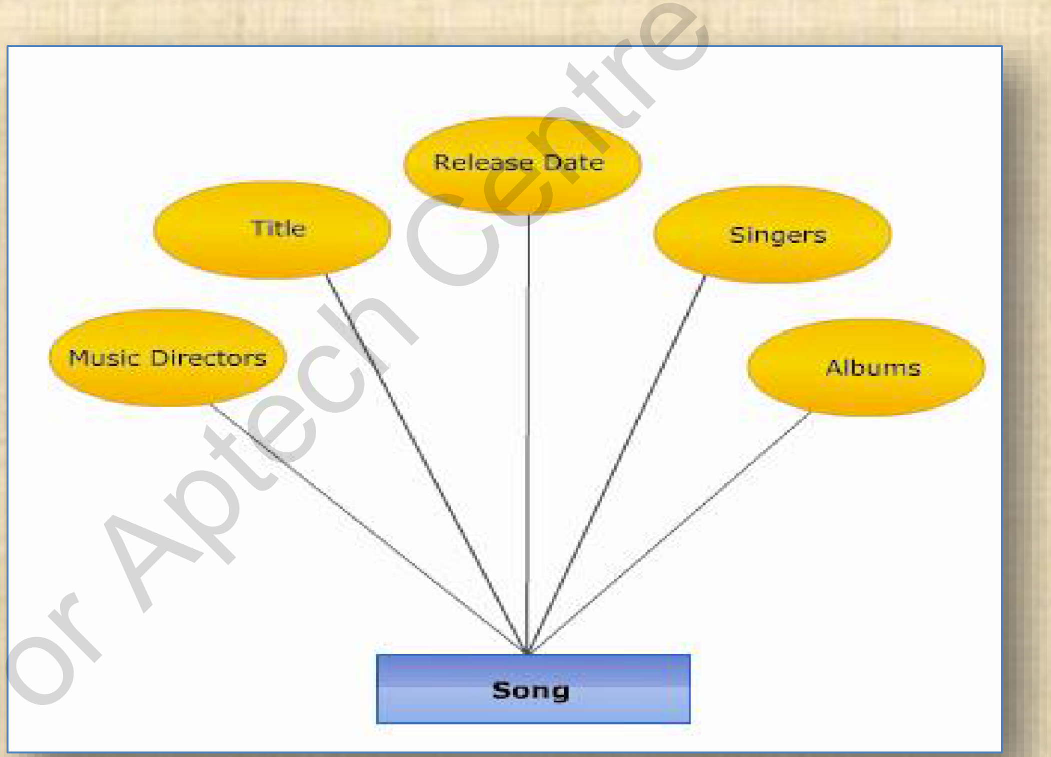
Attributes

Primary key

Foreign key

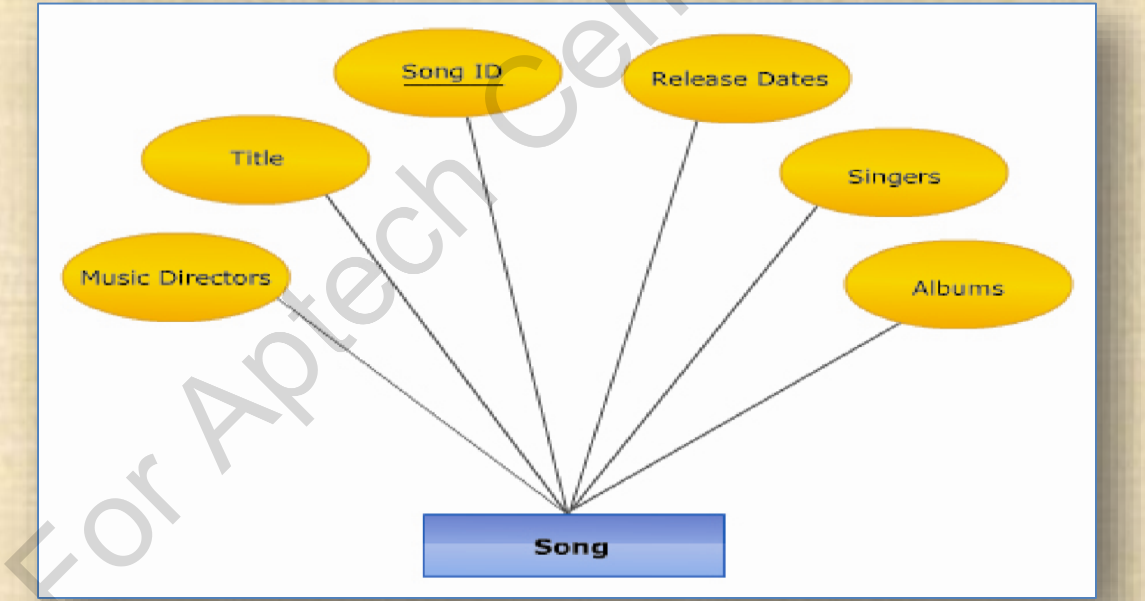
Attributes

- ❑ Each entity can have one or more attributes.
- ❑ Attribute is a specific piece of information about an entity.
- ❑ Following figure shows an example of attribute for a Song entity:



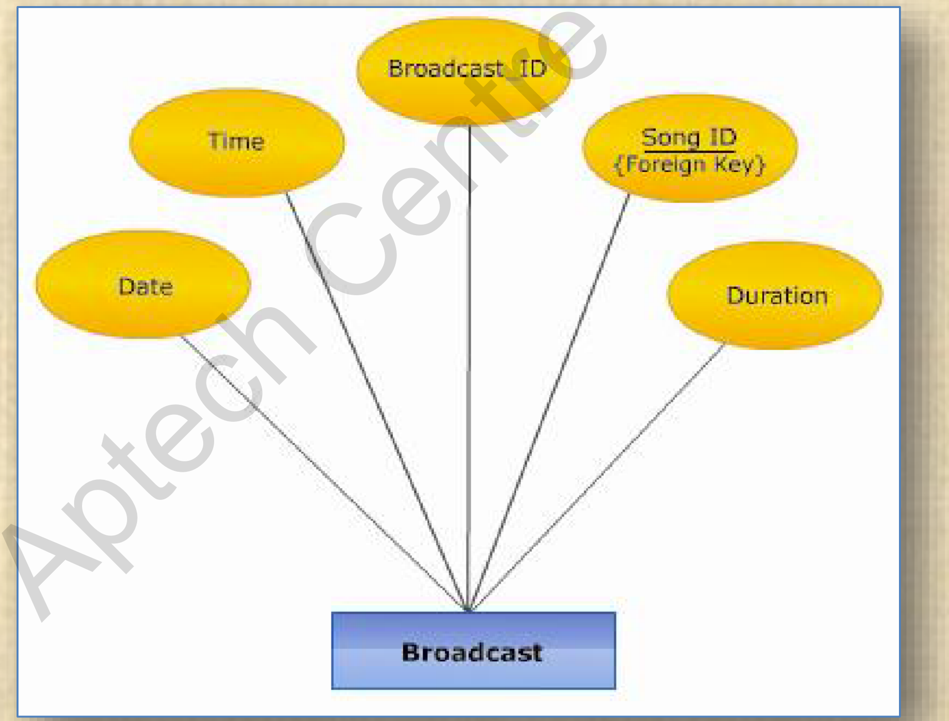
Primary Key

- ❑ An attribute which uniquely identifies the occurrence of an entity is called the primary key.
- ❑ It can be a single attribute or a combination of attributes.
- ❑ Essential attribute for an entity.
- ❑ Following figure shows primary key attribute of entity Song:



Foreign key

- ❑ Foreign key is used when two entities are to be related.
- ❑ Following figure shows a foreign key attribute in a Broadcast entity:



Relationship

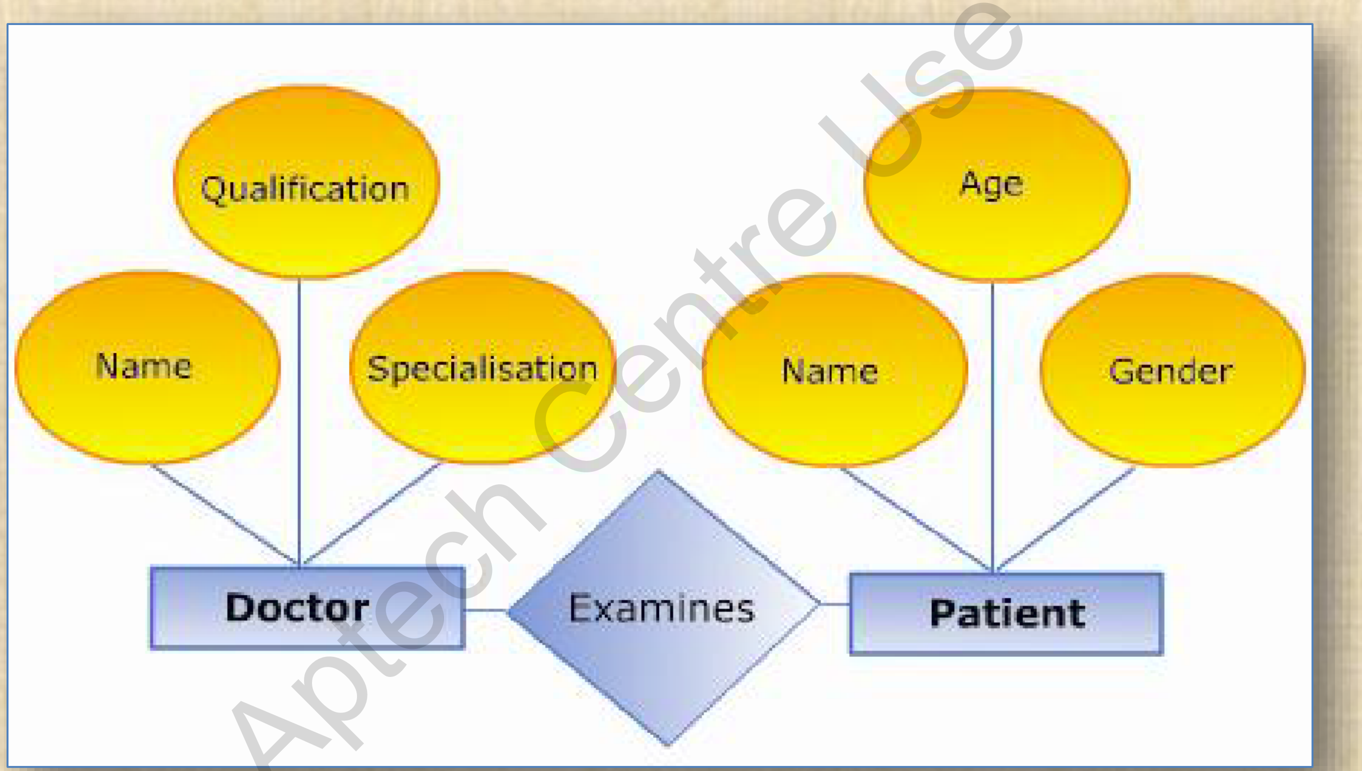
- Foreign keys help in forming relationships between entities.
- There are four types of relationships that can be created between entities in the database:
  - One-to-one relationship
  - One-to-many relationship
  - Many-to-one relationship
  - Many-to-many relationship

Data Modeling 1-2

- ❑ Data modeling is usually achieved with the help of an Entity-Relationship Diagram.
- ❑ Data modeling concepts such as entities, attributes, and relations are represented in the ER diagram using symbols.
- ❑ To create an ER diagram, you should identify the entities, attributes, and relationships of an activity for which data has to be maintained.

Data Modeling 2-2

- Following figure shows an ER diagram:



Managing Entities Relationship in JPA

The JPA specification provides support to the following aspects of entity relationships:

- Entity inheritance
- Polymorphism
- Managing relationships and associations
- Polymorphic queries

Entities in the enterprise applications can be associated based on two things:

- Cardinality
- Directionality

Cardinality 1-3

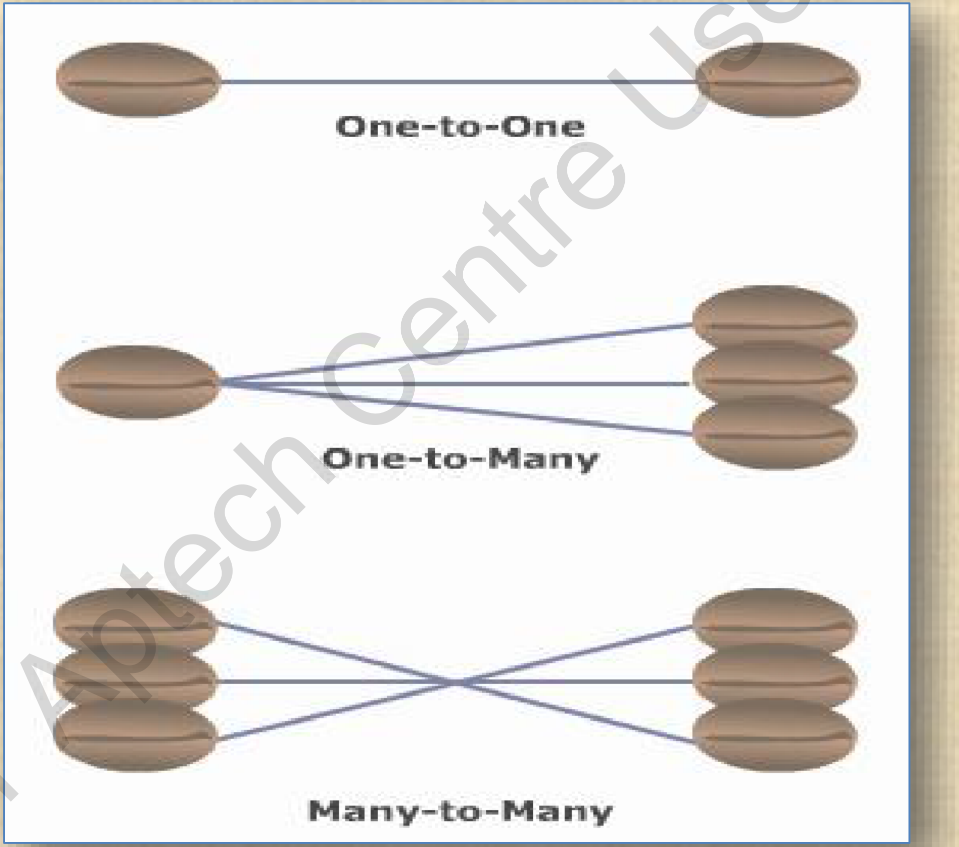
- The number of instances of an entity bean that relates to the number of instances of another bean is cardinality.
- JPA supports three types of cardinality relationships:
  - **One-to-one**
    - In one-to-one relationship, one bean instance relates to only one bean instance.
    - Relationship between a student bean and score card bean is an example of one-to-one relationship, as one student can be related to only one score card.

Cardinality 2-3

- **One-to-Many/Many-to-One**
  - In one-to-many relationship, one bean instance relates to multiple bean instances.
  - For instance, one customer bean instance can relate to multiple invoice beans but one invoice cannot be related to multiple customers.
- **Many-to-Many**
  - In many-to-many relationship, many bean instances relate to many instances of another bean.
  - The fact that many actors work in many movies is an instance of many-to-many relationship.

Cardinality 3-3

- Following figure demonstrates different types of cardinality:



Directionality

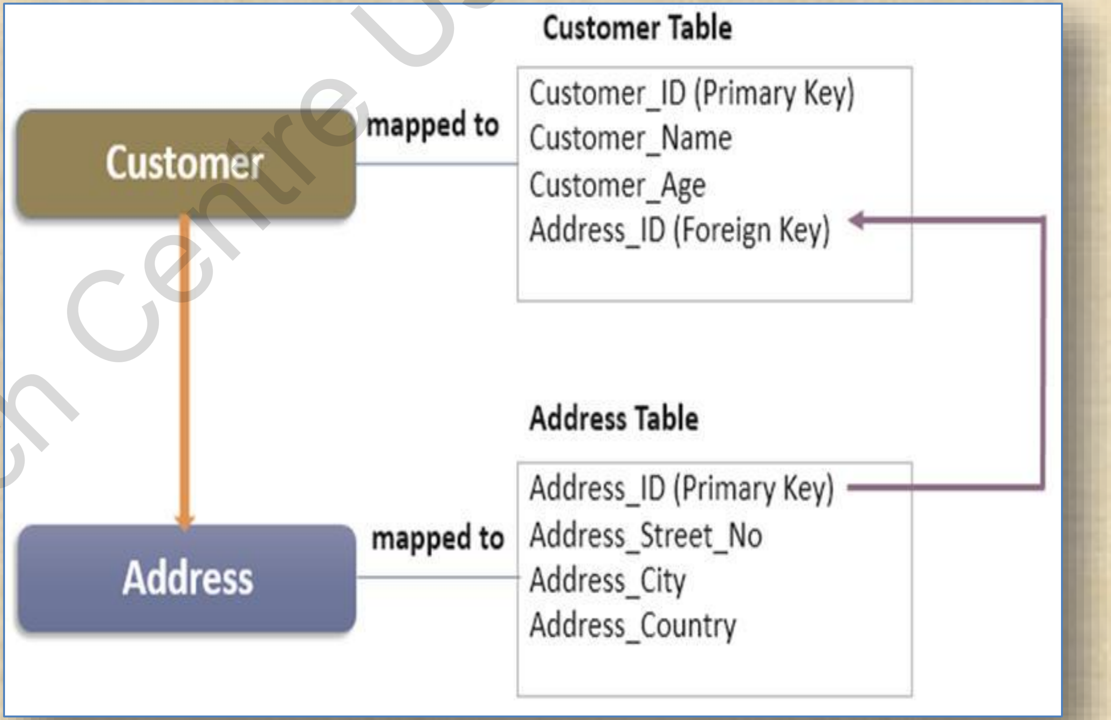
- ❑ Directionality defines the navigation pattern between two beans.
- ❑ The navigation pattern can be unidirectional or bidirectional.

One-to-One Unidirectional Relationship 1-3

- Following figure shows one-to-one unidirectional relationship:

The entity whose state defines the state of the other entity is said to be the owning side of the relationship.

Here, **Customer** entity will determine the corresponding address, but the **Address** entity cannot refer to the customer. Therefore, customer is the owning side of the relationship.



Customer
mapped to
Customer Table
Customer_ID (Primary Key)
Customer_Name
Customer_Age
Address_ID (Foreign Key)

Address
mapped to
Address Table
Address_ID (Primary Key)
Address_Street_No
Address_City
Address_Country

One-to-One Unidirectional Relationship 2-3

- One-to-one relationships are mapped using primary key and foreign key associations.
- They are annotated through `javax.persistence.OnetoOne`.
- `@OnetoOne` annotation has the following attributes:
  - `targetEntity`
  - `cascade`
  - `fetch`
  - `mappedBy`
  - `orphanRemoval`

One-to-One Unidirectional Relationship 3-3

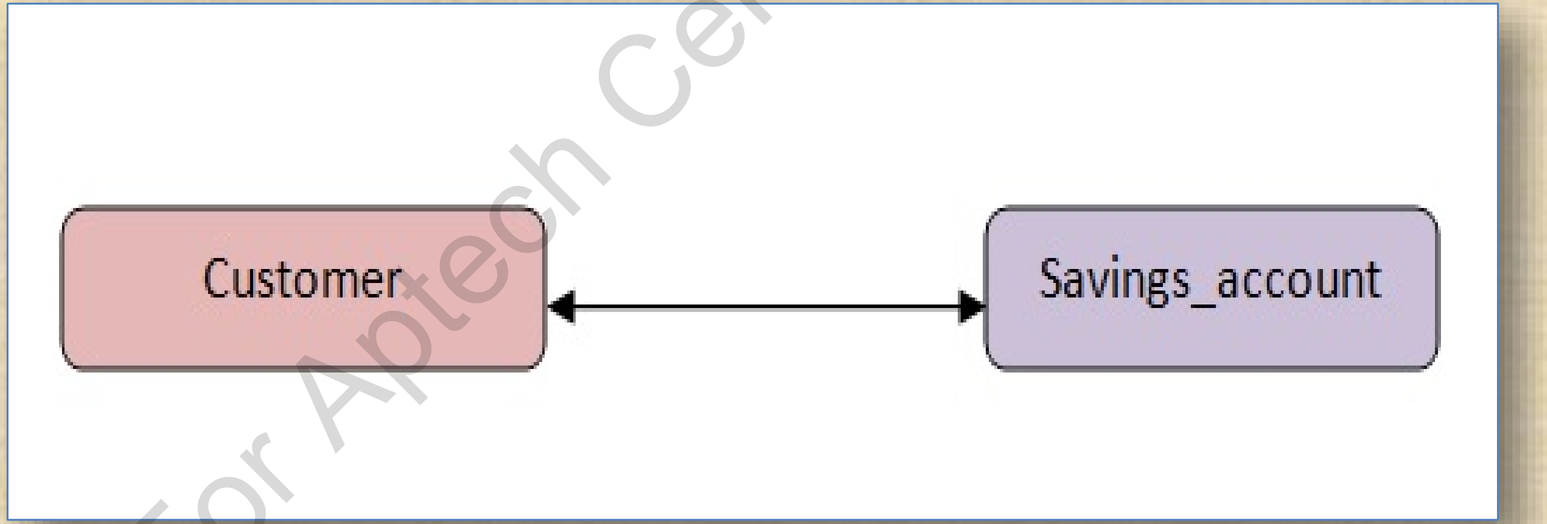
- Following code snippet demonstrates the one-to-one mapping for `Customer` and `Address` entities:

```
@Entity
@Table(name=”Customer”)
public class Customer {
     @Id
     @Column(name=”Customer_ID”)
      Protected String Customer_ID;


    // Mapping Foreign Key
    @OneToOne
    @JoinColumn(name=”Cus_address_id”,
      referencedColumnName=”Address_ID”, updatable=false)
      protected Address address; // reference of Address object
}
@Entity
@Table(name=”Address”)
public class Address {
   @Id
   @Column(name=”Address_ID”)
    protected String Address_ID;
     . . .
     . . .
}
```

- The `@JoinColumn` annotation contains the attribute name which refers to the name of the foreign key in the Customer table.
- The attribute `updatable` is set to false, which means that the persistence provider would not update the foreign key, even if the address reference were changed.

One-to-One Bidirectional Relationship 1-2

- Entity on either side of the relationship can determine the entity on the other side of the relationship. This is termed as a one-to-one bidirectional relationship.
- Following figure shows a one-to-one bidirectional relationship:



One-to-One Bidirectional Relationship 2-2

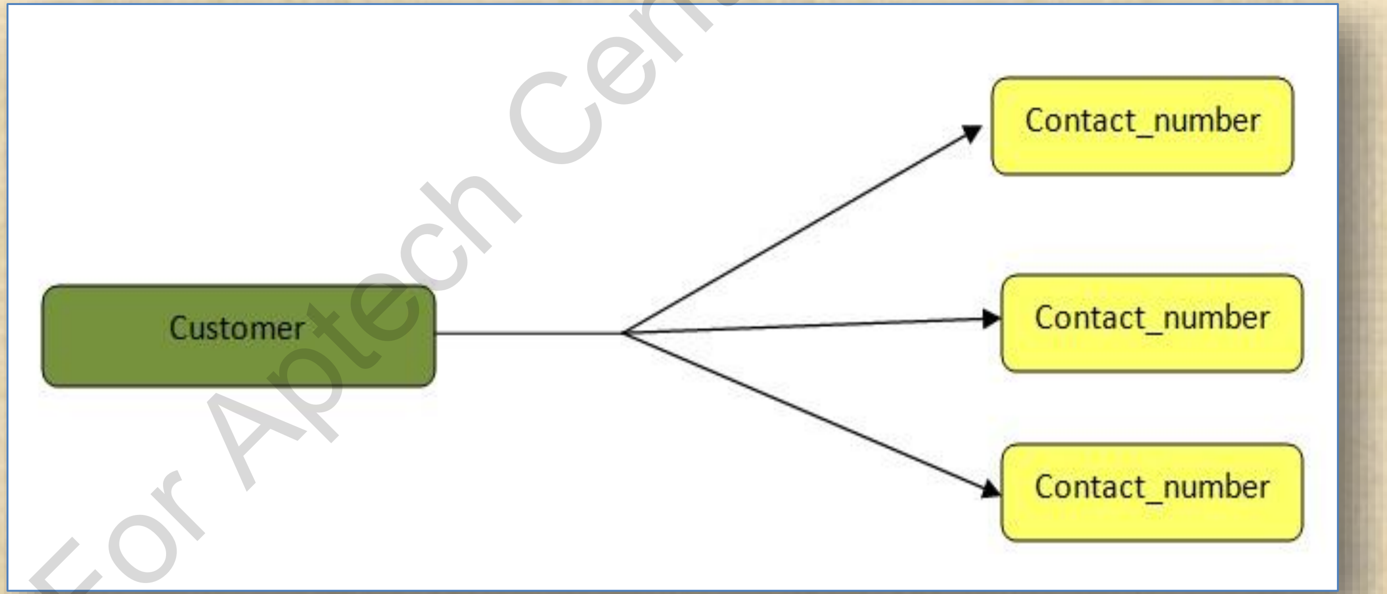
❑ Following code snippet shows the bidirectional mapping of Customer and Address entity:

```
@Entity
@Table(name=”Address”)
public class Address {

@OneToOne(mappedBy=”address”)
protected Customer customer;

   @Id
   @Column(name=”Address_ID”)
   protected String Address_ID;
   . . .
   . . .
}
```

- The `mappedBy` element identifies the corresponding association field on the owing side of the relationship.

One-to-Many/Many-to-One Relationship 1-4

- ❑ Each entity of a type is associated with more than one entity of another type.
- ❑ Following figure shows an example of one-to-many relationship:

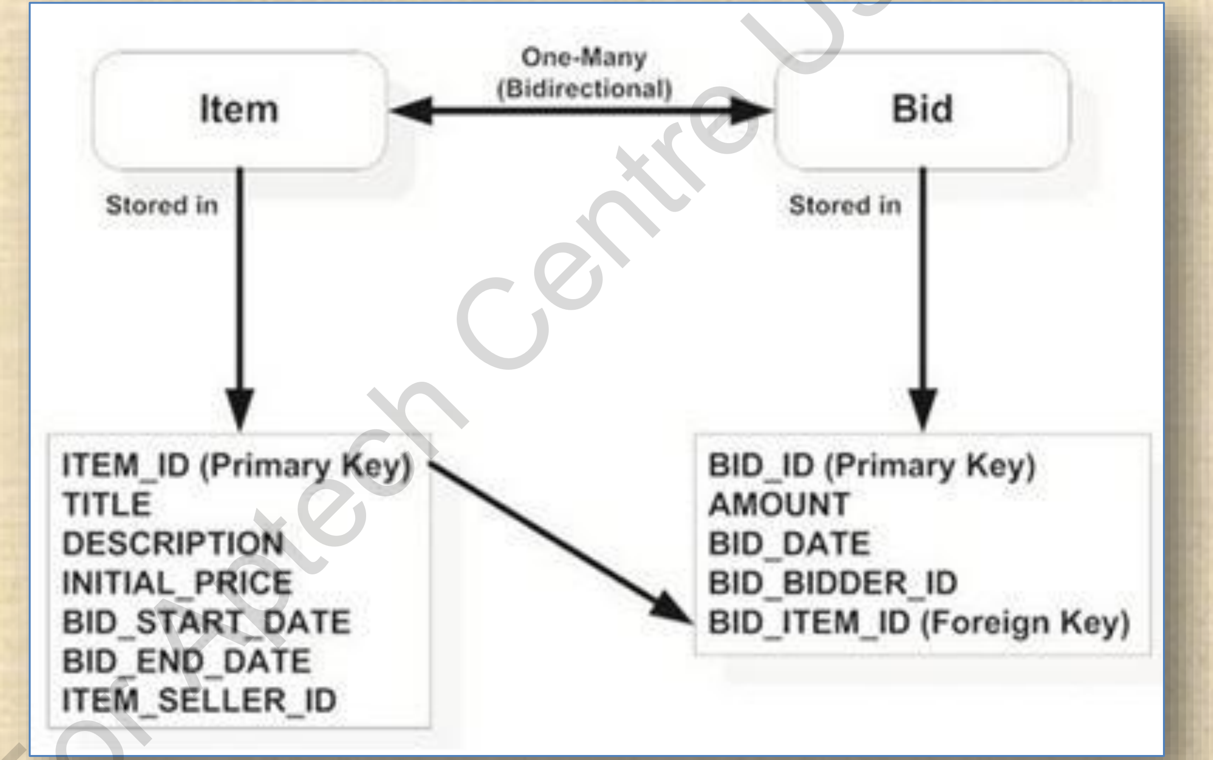


One-to-Many/Many-to-One Relationship 2-4

- javax.persistence.OnetoMany annotation is used to set the relationship between two entities.
- @OnetoMany has the following attributes:
  - targetEntity
  - cascade
  - fetch
  - mappedBy
  - orphanRemoval
- To map the relationship between the two entities in the inverse, then Many-to-One associations can be built.
- javax.persistence.ManytoOne annotation is used to represent many-to-one relationships.

One-to-Many/Many-to-One Relationship 3-4

- Following figure shows database schema of Item and Bid relationship in an auction:



Item

One-Many
(Bidirectional)

Bid

Stored in

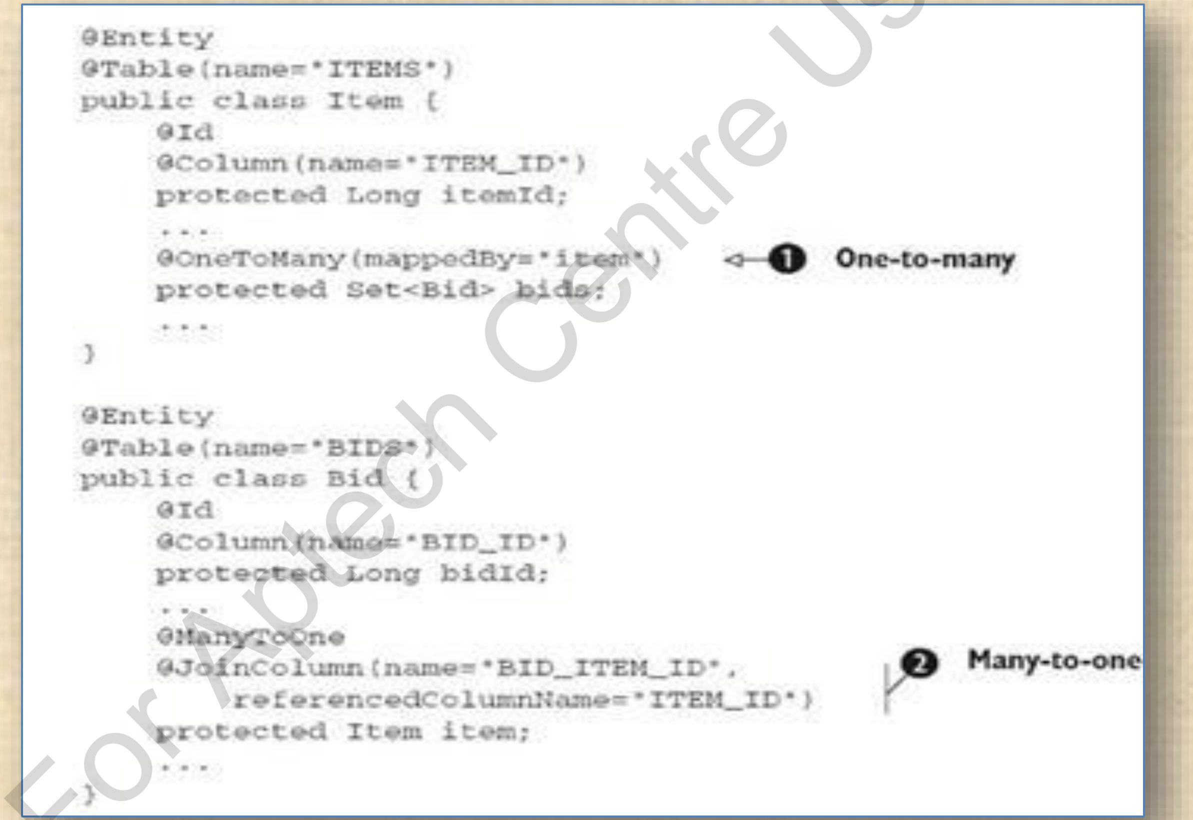
Stored in

ITEM_ID (Primary Key)
TITLE
DESCRIPTION
INITIAL_PRICE
BID_START_DATE
BID_END_DATE
ITEM_SELLER_ID

BID_ID (Primary Key)
AMOUNT
BID_DATE
BID_BIDDER_ID
BID_ITEM_ID (Foreign Key)

One-to-Many/Many-to-One Relationship 4-4

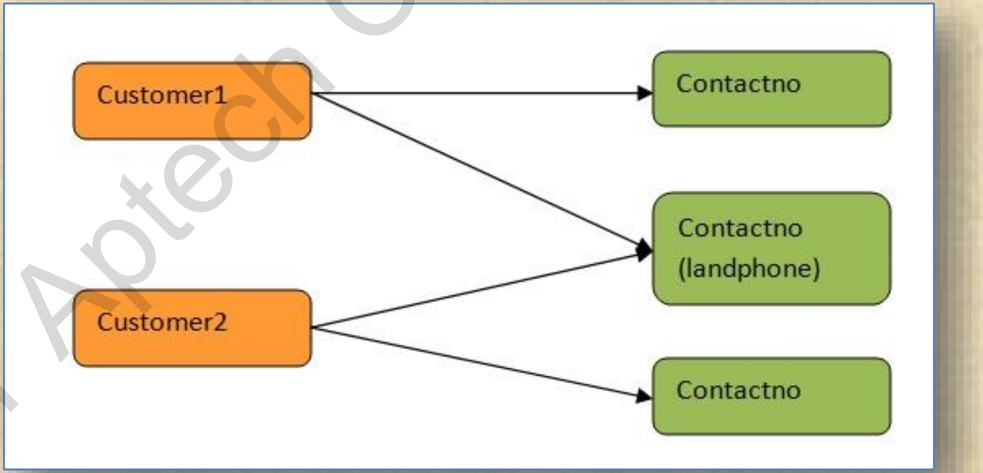
- Following figure shows the bidirectional implementation of Item and Bid relationship:



```
@Entity
@Table(name="ITEMS")
public class Item {
    @Id
    @Column(name="ITEM_ID")
    protected Long itemId;
    ...
    @OneToMany(mappedBy="item")      <-❶ One-to-many
    protected Set<Bid> bids;
    ...
}

@Entity
@Table(name="BIDS")
public class Bid {
    @Id
    @Column(name="BID_ID")
    protected Long bidId;
    ...
    @ManyToOne
    @JoinColumn(name="BID_ITEM_ID",                 ❷ Many-to-one
        referencedColumnName="ITEM_ID")
    protected Item item;
    ...
}
```

Many-to-Many Relationship 1-3

- ❑ Each entity on either side of the relationship is associated with more than one entity on the other side.
- ❑ Many-to-many relationship can be a unidirectional or bidirectional relationship.
- ❑ Following figure shows a many-to-many unidirectional relationship:

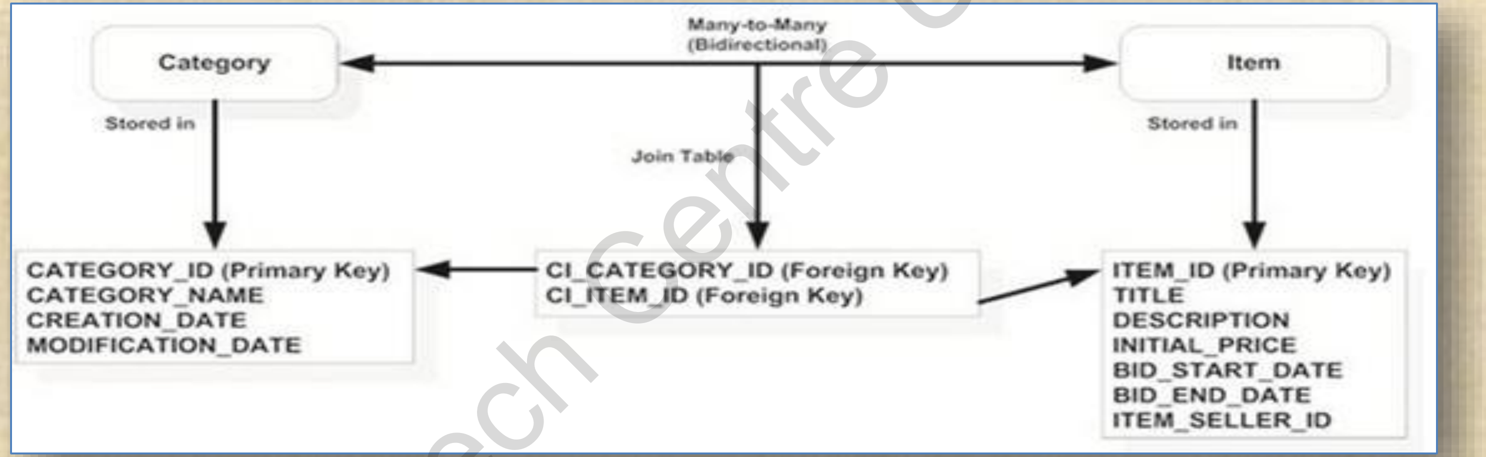


Many-to-Many Relationship 2-3

- ❑ In a many-to-many bidirectional relationship both the entities in the association can be referenced by more than one entity of the other type.
- ❑ Both unidirectional and bidirectional many-to-many relationships are annotated with `javax.persistence.ManytoMany`.

Many-to-Many Relationship 3-3

- Following figure shows an example of many-to-many relationship:



CATEGORY_ID (Primary Key)
CATEGORY_NAME
CREATION_DATE
MODIFICATION_DATE

CI_CATEGORY_ID (Foreign Key)
CI_ITEM_ID (Foreign Key)

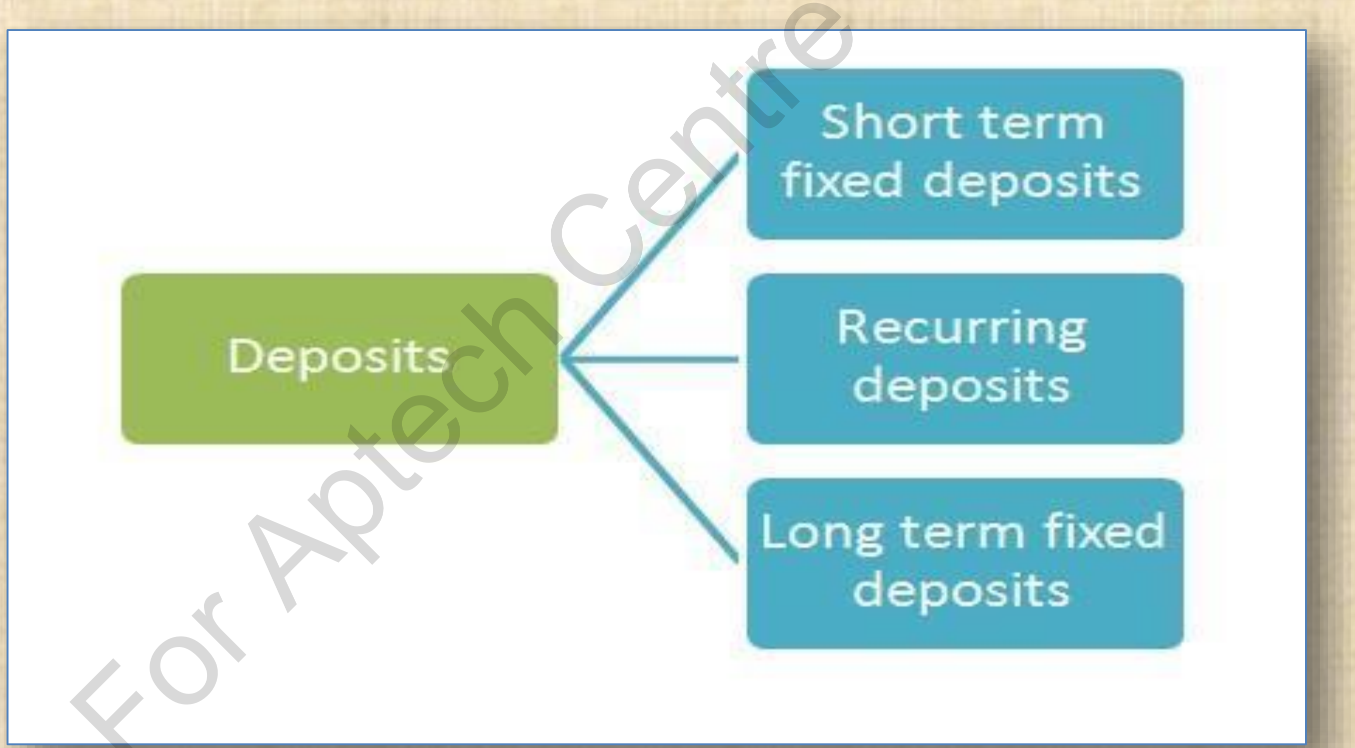
ITEM_ID (Primary Key)
TITLE
DESCRIPTION
INITIAL_PRICE
BID_START_DATE
BID_END_DATE
ITEM_SELLER_ID

Entity Inheritance

- Enterprise applications use inheritance to achieve code reuse and polymorphism.
- The inheritance in the application design has to be translated to the database design.
- JPA provides three different strategies for translating entity inheritance onto database:
  - A single table per class hierarchy
  - A table per concrete entity class
  - A table per sub class

Single Table Per Class Hierarchy 1-4

- All the entities in the class hierarchy are mapped onto a single table.
- Consider the hierarchy as shown in the following figure:



Single Table Per Class Hierarchy 2-4

□ All the entities shown in the hierarchy are mapped onto a single table as shown in the given table.

Fixed deposit number	Linked savings Account number	Customer name	Amount

□ The table definition requires a way for discriminating between the child classes of the `FixedDeposit` class.

□ Following is the table definition with discriminator column:

Fixed deposit number	Linked savings account number	Customer name	Amount	Fixed deposit type

Single Table Per Class Hierarchy 3-4

- The discriminator column for the mapping strategy is defined through `javax.persistence.DiscriminatorColumn`.
- Following are the attributes of the `DiscriminatorColumn`:
  - `name`
  - `Discriminator type`
  - `column definition`
  - `length`

Single Table Per Class Hierarchy 4-4

- Following code snippet demonstrates the single table per class hierarchy:

```
@Entity(name = "Account")
@DiscriminatorColumn(name = "DISCRIMINATOR",
discriminatorType =DiscriminatorType.STRING)
@DiscriminatorValue("account_type")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
class Account{
 @Id
  String account_type;
 @Id
  long account_number;
...
}
```

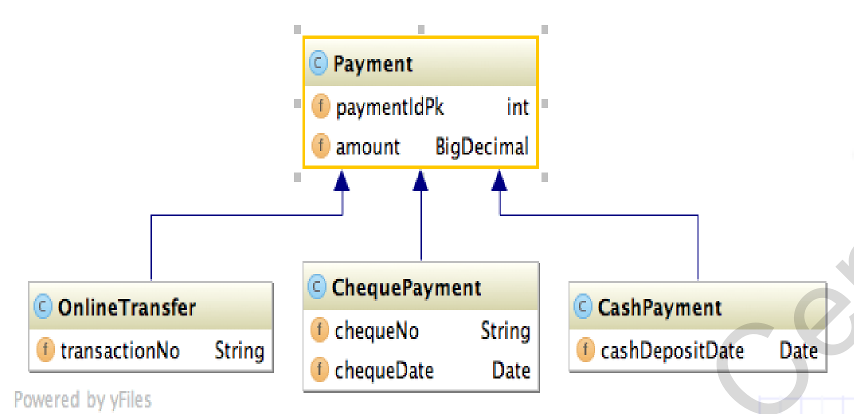
- The discriminator is of type String. The property **`account_type`** of the entity is used to discriminate different types of deposits in the table.

Table Per Concrete Entity Class Strategy 1-2

- A table is created for every entity class in the hierarchy.
  - This strategy corresponds to **`InheritanceType.TABLE_PER_CLASS`**.
- Each of these tables has columns which are properties of the sub class.
- This strategy does not require a discriminator column as in the case of single table per class hierarchy.
- In order to extract data from individual classes:
  - **A separate query is written on the individual tables or SQL UNION queries are used.**

Table Per Concrete Entity Class Strategy 2-2

- Following figure shows the class design and database table design for the Table per Concrete class:



Payment
paymentIdPk int
amount BigDecimal

OnlineTransfer
transactionNo String

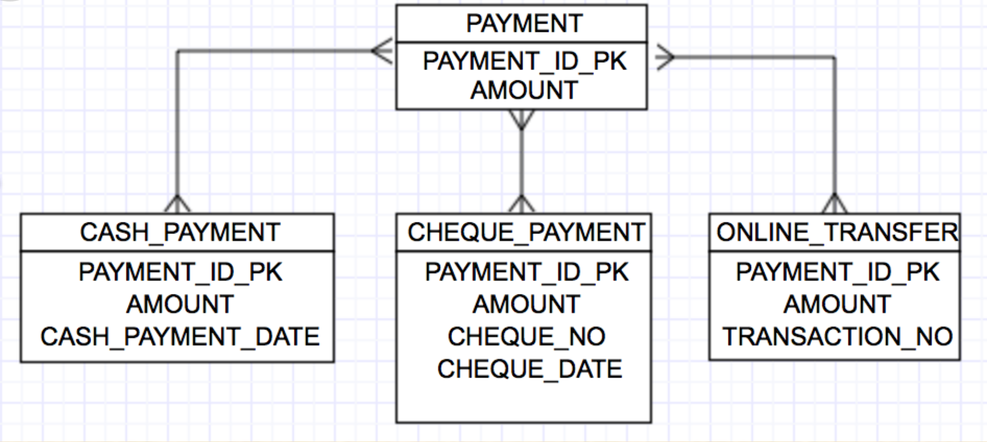
ChequePayment
chequeNo String
chequeDate Date

CashPayment
cashDepositDate Date

Powered by yFiles

Class Diagram

Database Table Design



PAYMENT
PAYMENT_ID_PK
AMOUNT

CASH_PAYMENT
PAYMENT_ID_PK
AMOUNT
CASH_PAYMENT_DATE

CHEQUE_PAYMENT
PAYMENT_ID_PK
AMOUNT
CHEQUE_NO
CHEQUE_DATE

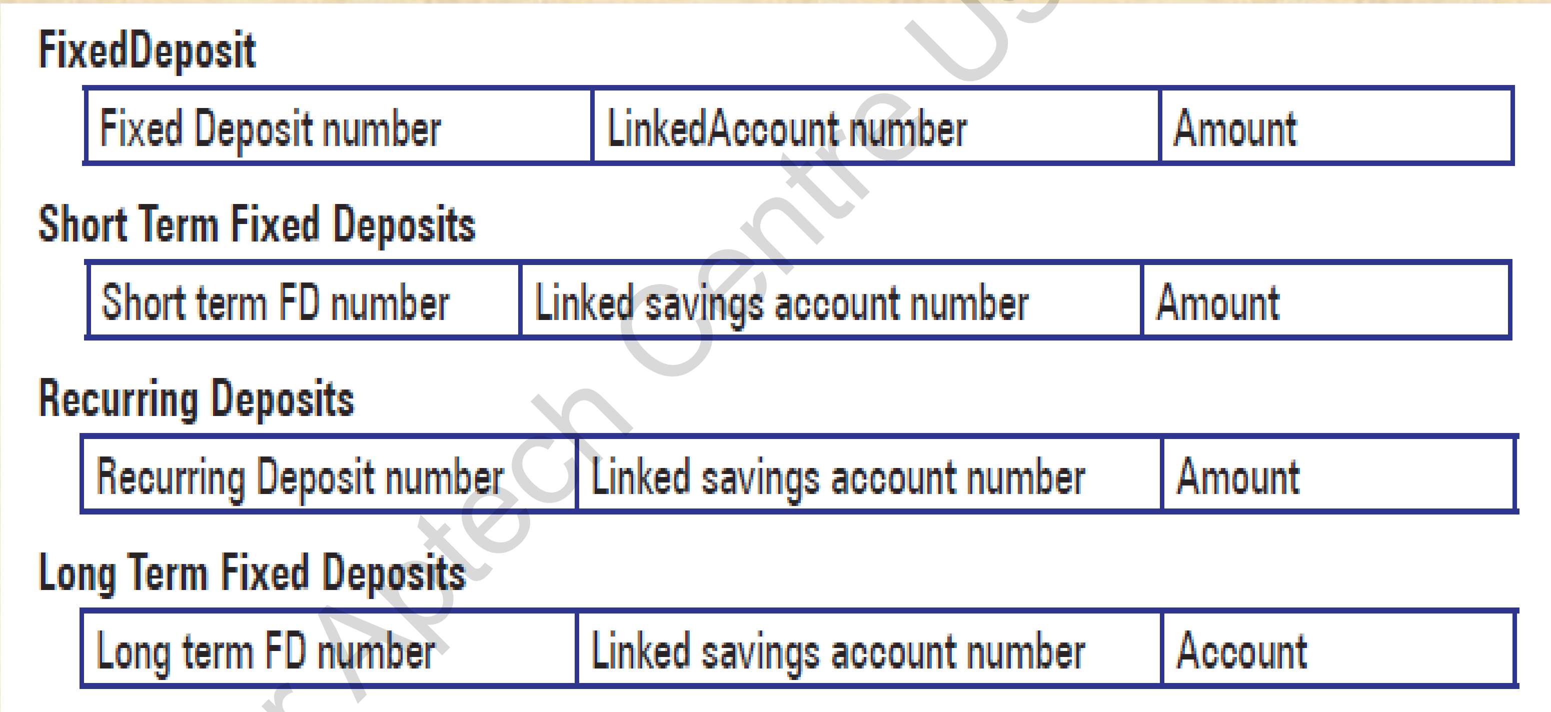
ONLINE_TRANSFER
PAYMENT_ID_PK
AMOUNT
TRANSACTION_NO

Table Per SubClass Strategy 1-2

- ❑ Is also known as joined class strategy.
- ❑ Annotated with `javax.persistence.Joined`.
- ❑ The super class or the root of the class hierarchy is represented by a single table.
- ❑ Each subclass of the hierarchy is mapped as a different table.
  - ▪ Columns of these tables are the properties of the subclass.
- ❑ Each sub class has its own primary key.
- ❑ All the entities of the hierarchy can be aggregated by applying a **JOIN** operation on all the subclasses.
- ❑ This strategy provides good support for polymorphic relationships.

Table Per SubClass Strategy 2-2

- The hierarchy can be implemented using the following tables:



FixedDeposit

Fixed Deposit number	LinkedAccount number	Amount

Short Term Fixed Deposits

Short term FD number	Linked savings account number	Amount

Recurring Deposits

Recurring Deposit number	Linked savings account number	Amount

Long Term Fixed Deposits

Long term FD number	Linked savings account number	Account

Annotations Used For Entity Inheritance Mapping

- The hierarchy in an application can be identified while deploying by annotating the root class of the hierarchy with `javax.persistence.Inheritance.`
- The mapping strategy is defined through the annotation `javax.persistence.InheritanceType.`
- The `InheritanceType` annotation can assume any one of the following values:
  - `SINGLE_TABLE` (default) corresponds to single table per class hierarchy.
  - `JOINED` value corresponds to a table per subclass strategy.
  - `TABLE_PER_CLASS` value corresponds to a table per concrete entity class strategy.

Non-Entity Base Classes

❑ Entity classes can also be inherited from non-entity base classes.

Abstract Entity classes

- Cannot be instantiated.
- Concrete entity classes can extend and define the functionality of these abstract entity classes.
- Prefixed with ‘`abstract`’ keyword.

Mapped super classes

- Are those classes in the enterprise application which are not
- persisted.
- Applications may inherit the behavior and properties of such super class but the state of the mapped super classes are not persisted.
- Annotated with `@MappedSuperClass`.

Summary

- ❑ Relationships describe how objects collaborate with one another, to contribute to the behavior of the system.
- ❑ A database schema describes the table structure, data types, and relations in a database.
- ❑ Some of the common terms in relational database are attributes, primary key, and foreign key.
- ❑ There are four types of relationships that can be created between the entities in the database that includes one-to-one, one-to-many, many-to-one, and many-to-many.
- ❑ Entity beans represent objects in OOAD.
- ❑ In order to support entity relationships, the object-to-relational mapping engine must provide support for object-oriented features such as inheritance, polymorphism, and so on.
- ❑ Enterprise applications use inheritance among entities for code reuse and to implement polymorphism.
- ❑ Mapping of persistent application objects onto the database can be defined through annotations in JPA.
- ❑ The association among entities is defined through relationships that can be unidirectional or bidirectional.
- ❑ JPA defines three strategies for mapping the entity inheritance onto the database.
- ❑ javax.persistence.Inheritance and javax.persistence.InheritanceType are the annotations used to map inheritance onto the database.
- ❑ Abstract Entity classes and Mapped super classes are non-entity base classes.