Object Relational Mapping
Lars Mortensen

Literature: https://en.wikibooks.org/wiki/Java_Persistence

This second reference is for a specific database (ObjectDB) but since this database uses JPA, you can use the tutorial as a quick way to get started.

http://www.objectdb.com/java/jpa

Object-relational mapping

- Technique for converting data between tables in relational databases to objects in an object oriented language
- Creates in effect a "virtual" object database

Why?

- Data management tasks in OO-programming are typically implemented by manipulating objects that are almost always **non-scalar** values
- Relational database management systems can only store and manipulate **scalar** values such as integers and strings organized within tables.

Without Object-relational mapping

Programmers must:

- Either convert the object values into groups of simpler values for storage in the database (and convert them back upon retrieval),

- Use only simple scalar values within the program.

Object-relational mapping is used to implement the first approach
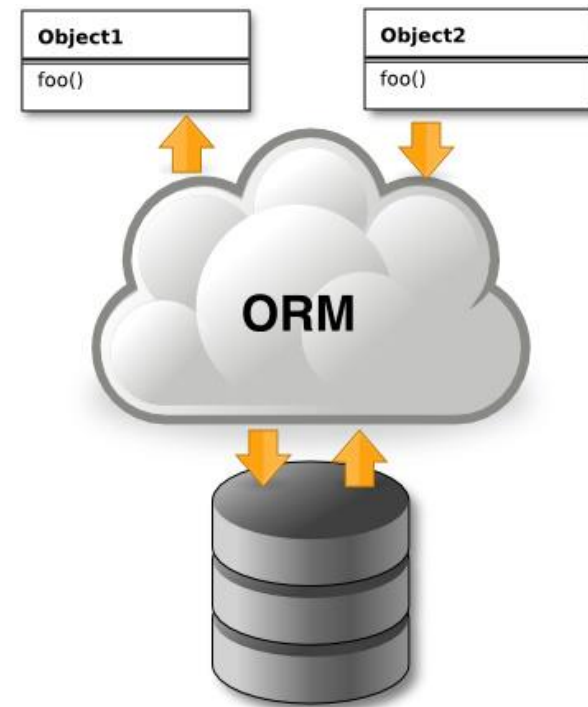
To ORM or not to ORM ☺

Comparison with traditional data access techniques

+ ORM typically reduces the amount of code that
     needs to be written

+ Avoids low level JDBC and SQL code

+ Provides database and schema independence

+ It allows us to use the OO-paradigm

- The high level of abstraction can obscure what is
     actually happening in the implementation code.

- Be aware of JPA's Convention-Over-Configuration Strategy

- Heavy reliance on ORM software has been cited as
     a major factor in producing <u>poorly designed databases</u>.

- There are a variety of difficulties that arise when
     considering how to match an object system to a
     relational database.

Object-Relational Impedance Mismatch

Our Goals

- Take advantage of all the things Relational Databases do well

- But doing it, without leaving all the all the things OO-languages do well

- Have the illusion of only "talking" OO, even when we manipulate data.

- Do less work

Object-Relational Impedance Mismatch Issues

- How are columns, rows, tables mapped to objects?
- How are relationships handled?
- How is OO inheritance mapped to relational tables?
- How is composition and aggregation handled?
- How are conflicting type systems between databases handled?
- How are objects persisted?
- How are different design goals handled?
- Relational model designed for data storage/retrieval
- Object Oriented model is about modelling behaviour

Object-Relational Impedance Mismatch Issues

- Example – mismatch in data types
  - OO Languages such as Java C# have String and int data types
  - RDMBS such as MySQL has a varchar and smallint
- Although values are stored and manipulated differently the database driver (JDBC in Java) handles conversions automatically

Object-Relational Impedance Mismatch Issues

- Example – collections versus tables
  - Java/C# use collections to manage lists of objects
  - Databases uses tables to manage lists of entities

- Example – blobs versus objects
  - Databases uses blobs to manage large objects as simple binary data
  - Java/C# use objects with behaviors
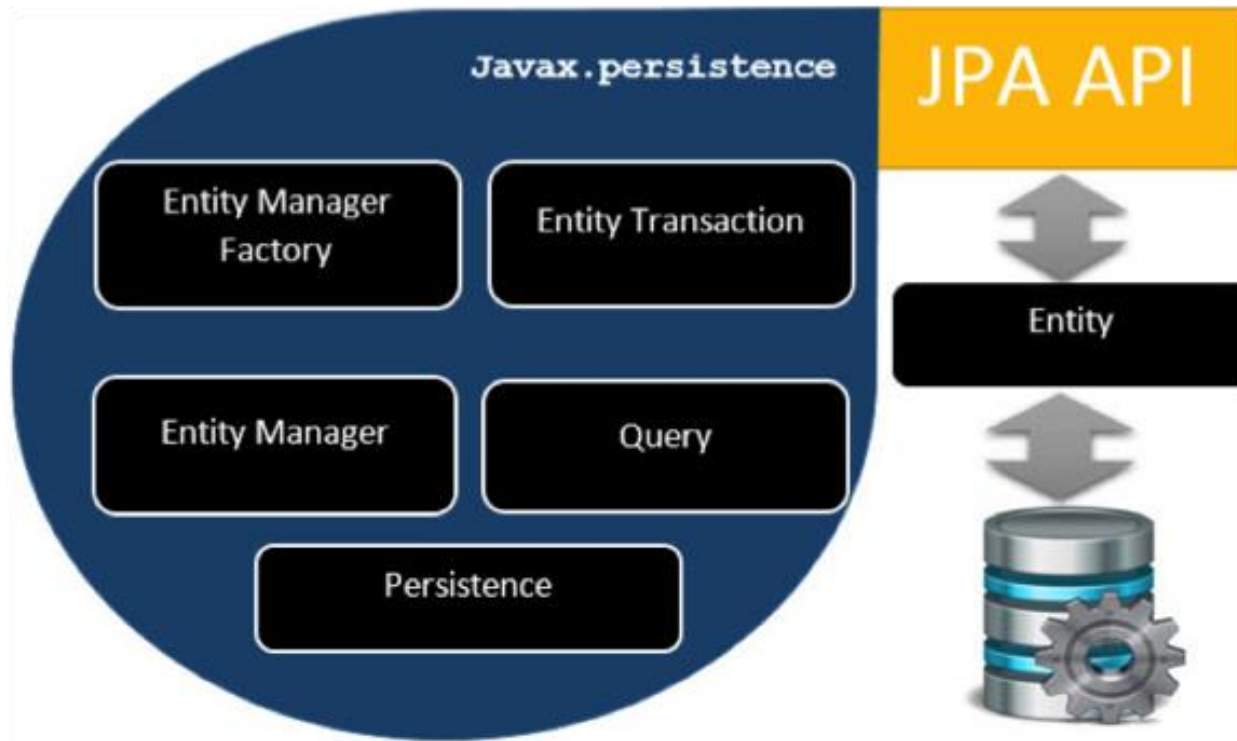
Object Relational Mapping Tools

cphbusiness

http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software#.NET

Introduction to the Java Persistence API

cphbusiness

Java Persistence consists of four areas:

- ## The Java Persistence API
  API which provides Java developers with an object/relational mapping facility for managing relational data in Java applications

- ## The query language (JPQL)
  Query language allows us to write portable queries that work regardless of the underlying data store

- ## The Java Persistence Criteria API (OO-Queries)
  Queries written using Java APIs, which are type safe, and portable.
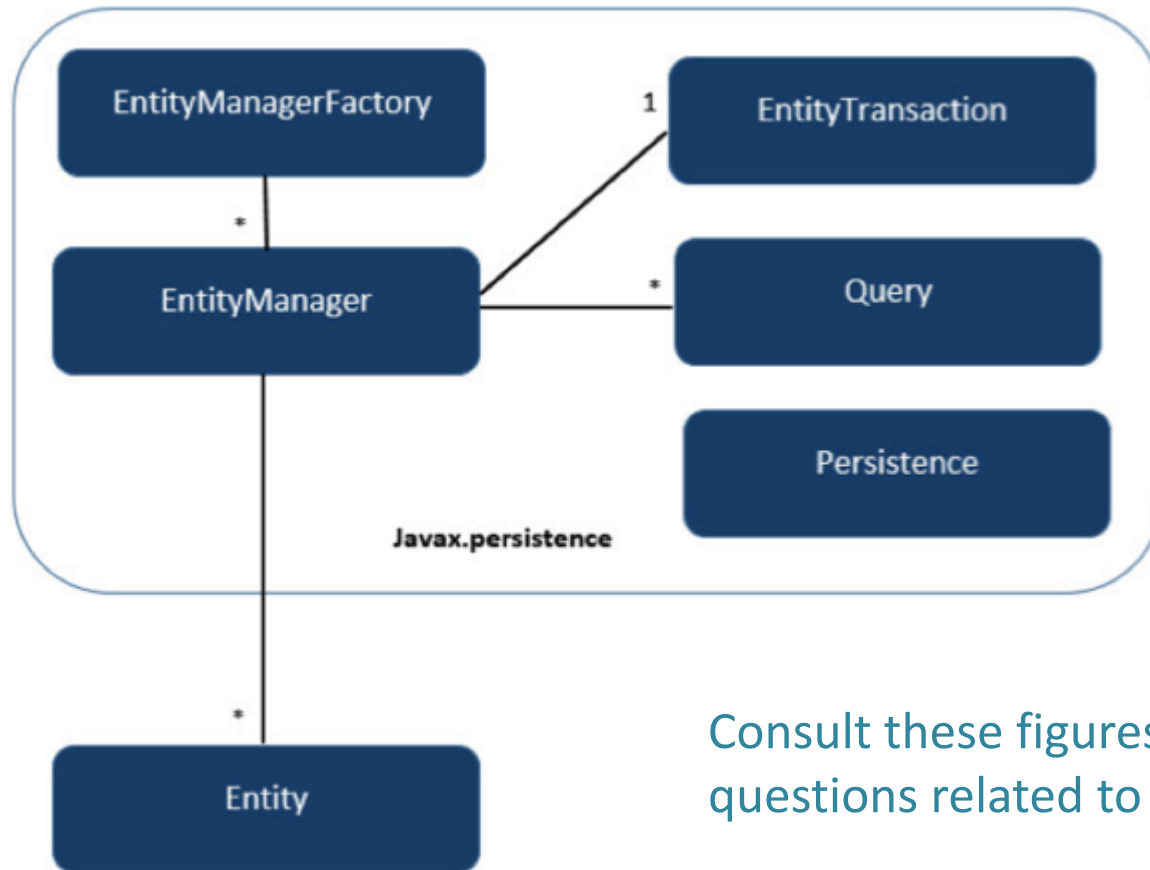
- ## Object/relational mapping metadata

http://docs.oracle.com/javaee/7/tutorial/doc/persistence-intro.htm#BNBPZ

JPA Architecture

cphbusiness

More Info: https://www.tutorialspoint.com/jpa/jpa_architecture.htm

http://docs.oracle.com/javaee/7/tutorial/doc/persistence-intro.htm#BNBPZ

JPA Class Relationships

Consult these figures for exam-questions related to JPA-architecture

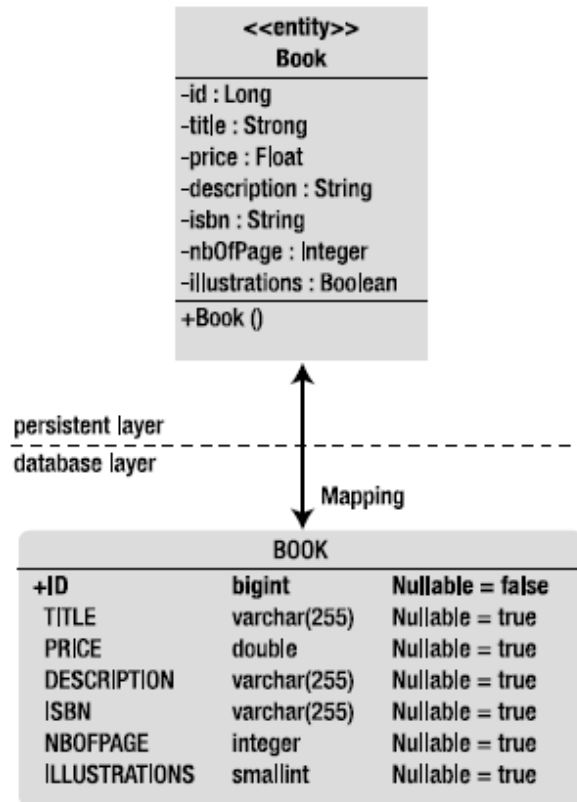More Info: https://www.tutorialspoint.com/jpa/jpa_architecture.htm

http://docs.oracle.com/javaee/7/tutorial/doc/persistence-intro.htm#BNBPZ

Which JPA to use?

JPA - Entities

An entity is a lightweight persistence domain object.
Typically, an entity represents a table in a relational
database, and each entity instance corresponds to a row
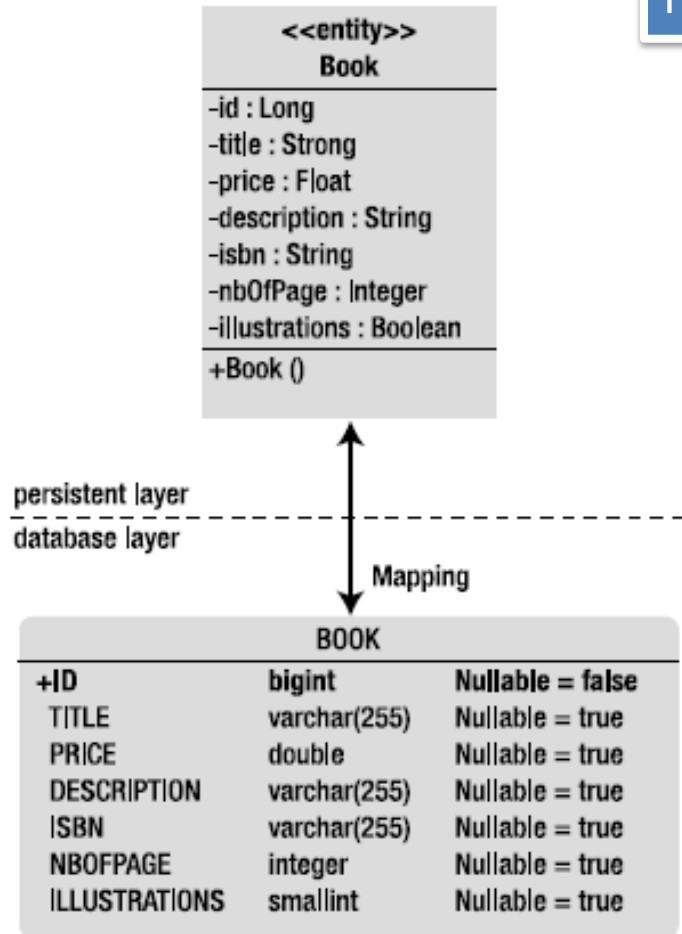in that table.

Requirements for Entity Classes

- The class must be annotated with the `@Entity` annotation.

- The class must have (at least) a public or protected, no-argument constructor.

- The class must not be declared `final`. No methods or persistent instance variables must be declared `final`.

- If an entity instance is passed by value as a detached object, the class must implement the `Serializable` interface.

- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.

- Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods.

Mapping Entities

Because of the Convention-Over-Configuration strategy used by the EE-framework, this is all that is required to turn the class in to a fully fledged Entity Class

<<entity>>
Book

-id : Long
-title : Strong
-price : Float
-description : String
-isbn : String
-nbOfPage : Integer
-illustrations : Boolean

+Book ()

persistent layer
- -
database layer

Mapping

BOOK		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true

```java
@Entity
public class Book ...
{
    @Id private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    public Book() { }
    // Getters, setters
}
```

Entity Manager and Persistence Context

- Entities are managed by the entity manager, which is represented by javax.persistence.EntityManager instances.

- Each EntityManager instance is associated with a **persistence context**: a set of managed entity instances that exist in a particular data store.

- A **persistence context** defines the scope under which particular entity instances are created, persisted, and removed.

- The EntityManager interface defines the methods that are used to interact with the persistence context.

The Entity Manager

The entity manager is a (perhaps THE) central piece in JPA.

- It manages the state and life cycle of entities as well as querying entities within a persistence context.
- It is responsible for creating and removing persistent entity instances and finding entities by their primary key.
- It can lock entities for protecting against concurrent access by using optimistic or pessimistic locking.
- It can use JPQL queries to retrieve entities following certain criteria.

The Entity Manager

```
Book2 book = new Book2();
book.setDescription("..");
 //...
EntityManagerFactory emf;
emf = Persistence.createEntityManagerFactory("pu-x");
EntityManager em = emf.createEntityManager();
Try{
  em.getTransaction().begin();
  em.persist(book);
  em.getTransaction().commit();
}
finally{
  em.close();
}

em.remove(book);
```
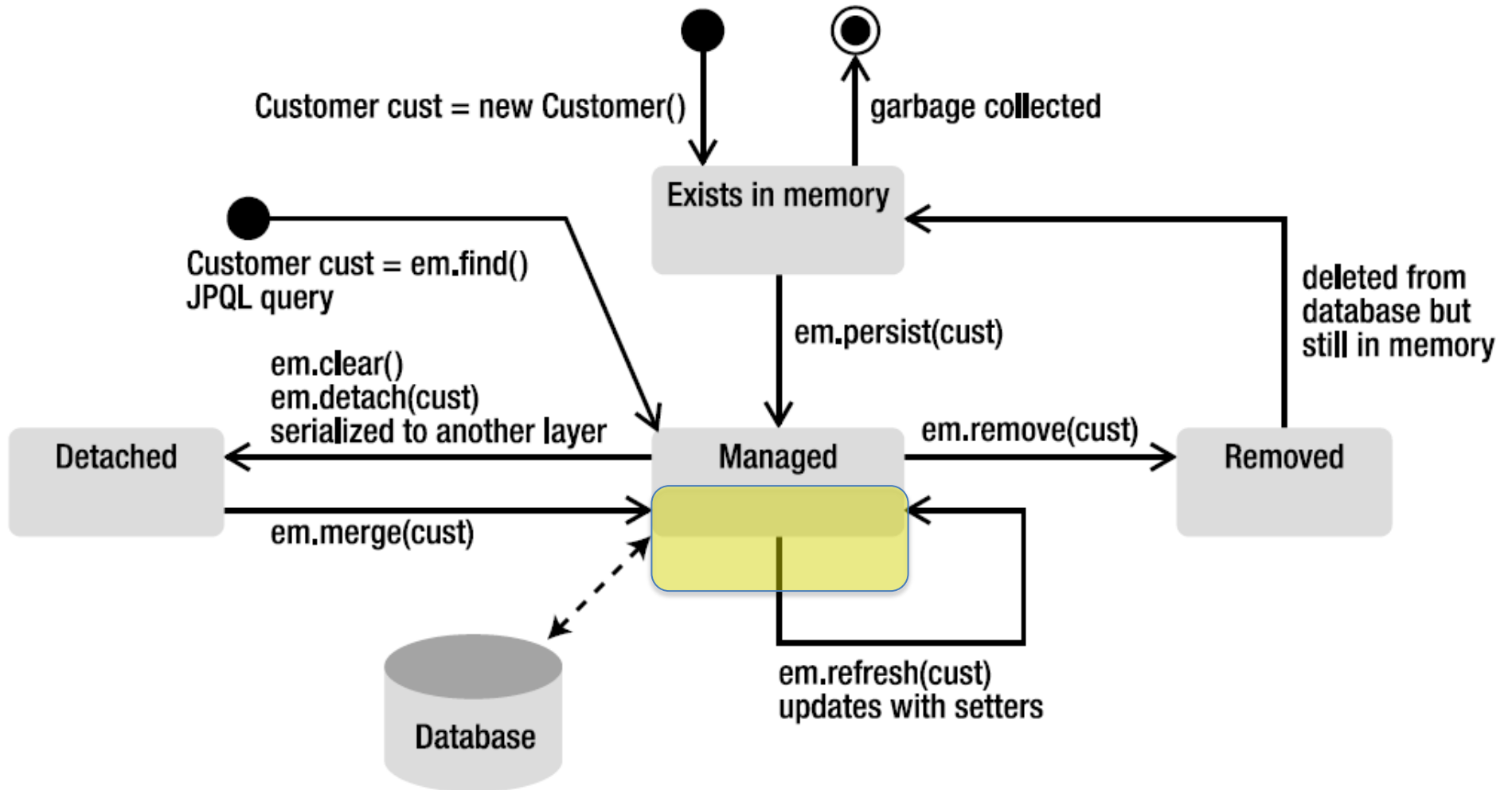
Just a POJO

Entities can be used as regular objects by different layers of an application

and become managed by the entity manager when we need to load or insert data into the database

Now the book is **Managed**

Again, just a POJO (**detached**)

Entity Life Cycle

Obtaining an Entity Manager

Obtaining an Entity Manager depends on which of the following strategies are used

Application-Managed Entity Managers

As we have seen this uses the **Persistence** class to create an **EntityManagerFactory**.
With an application Managed EntityManager, we as programmers are in charge of creating, closing and handle transactions.

Container-Managed Entity Managers

In a container-managed evironment (i.e. Glassfish for us) the usual way to obtain a EntityManager is by injection:

```
@PersistenceContext
EntityManager em;
```

The component running in a container (servlet, EJB, web service, etc.) doesn't need to create or close the entity manager, as its life cycle is managed by the container.

The Chicken or the Egg

With NetBeans and JPA we can either generate tables from existing Entity-classes or generate the Entity-classes from existing tables.

So which way should we go?
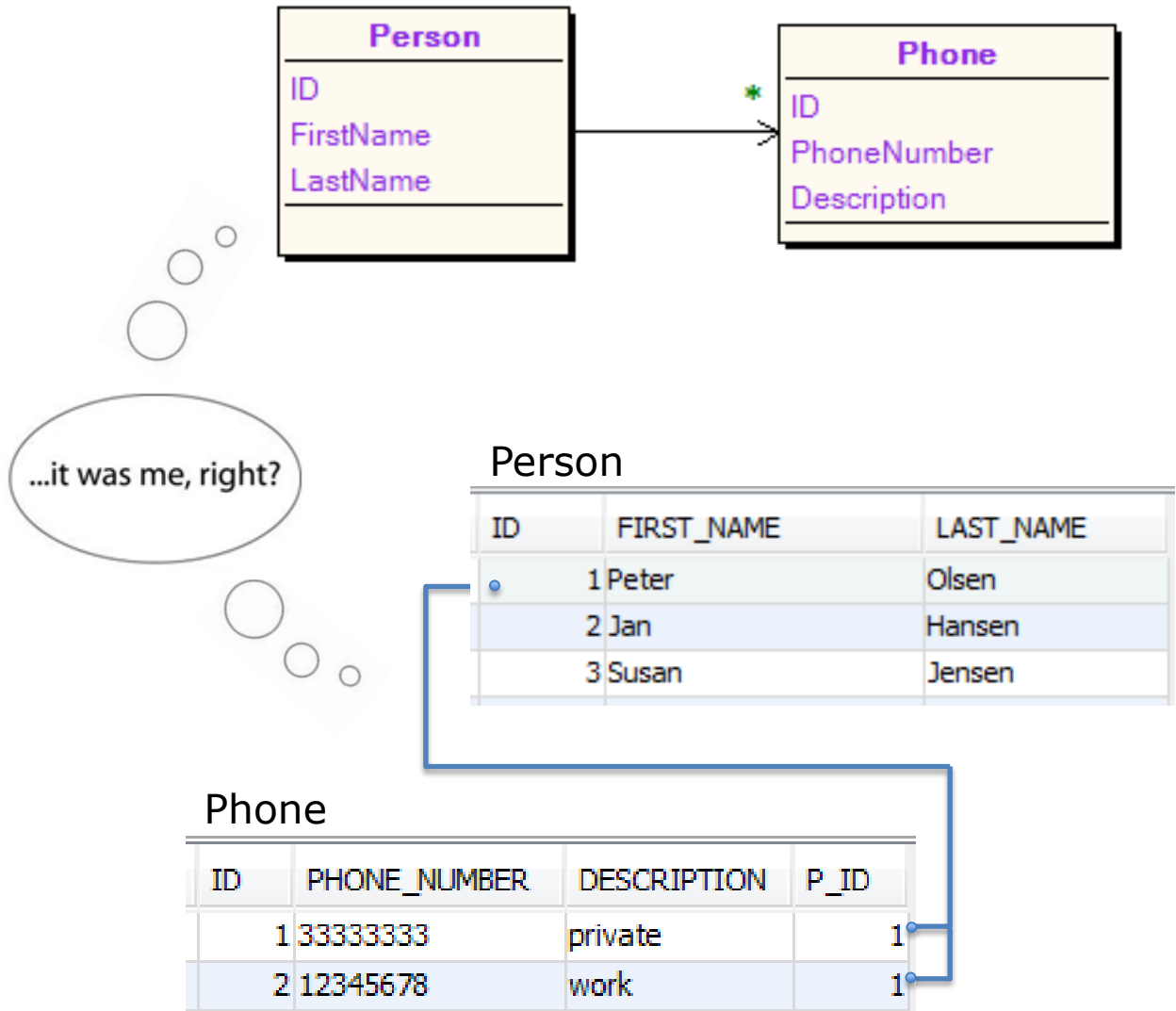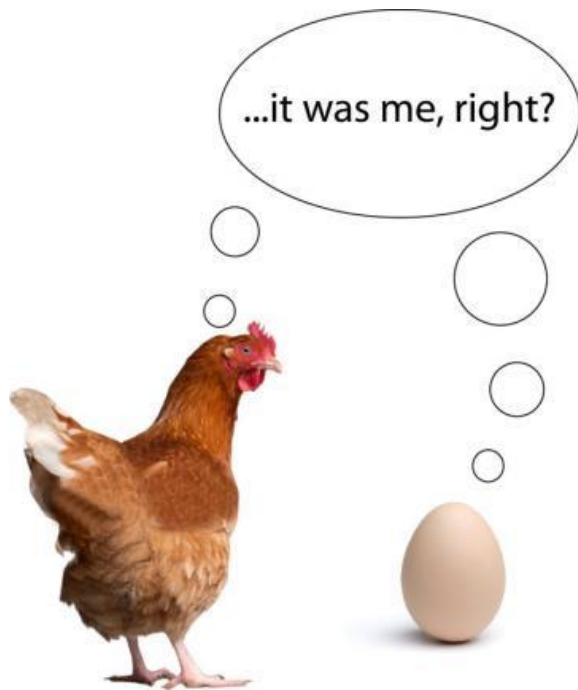
Entity Beans from existing tables

Tables from Entity Classes

JPA – Object Relational Mappings

The following will far from introduce all OR-mapping annotations but should give you a general overview of what is possible.

Use the following for details:

https://en.wikibooks.org/wiki/Java_Persistence.

The link below is for another specific database (ObjectDB) but is relevant for 99% of its content, and very easy to read
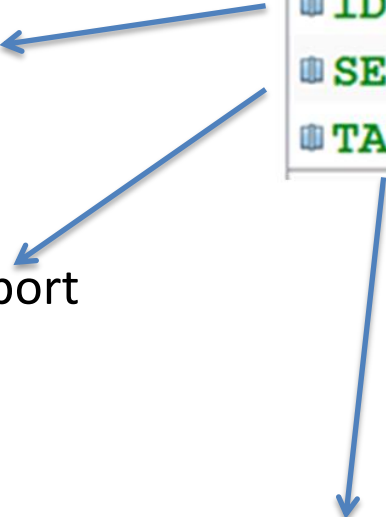http://www.objectdb.com/java/jpa

Primary Keys

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
Long id;
```

AUTO	GenerationType
IDENTITY	GenerationType
SEQUENCE	GenerationType
TABLE	GenerationType

Good choice for an MySQL database
AUTO_INCREMENT

Good choice for a Database which support
Sequences (Oracle or JavaDb)

Can be used by "all" DataBase Vendors.
Often the selected Strategy for AUTO

Auto Generation of ID's - Sequences
(for databases that supports this, like Oracle and Derby)

cphbusiness

We can control how a Sequence is generated or map it to an existing sequence.

```
Class Book{
...
  @ID
  @GeneratedValue(strategy = GenerationType.SEQUENCE,generator="s1")
  @SequenceGenerator(name="s1",sequenceName = "My_SEQ",
                     initialValue = 200000,allocationSize = 1)
```

These annotations will:
- Create a sequence as sketched below, if we are creating tables from Entities
- Map to the existing sequence if we are creating Entities from tables

Table Create Script

```
DROP SEQUENCE My_SEQ RESTRICT;
CREATE SEQUENCE My_SEQ START WITH 200000 INCREMENT BY 1;
```

Auto Generation of ID's - Tables

All databases can use a separate Table as their strategy to provide a "next id" value

This is usually the default when you select: **GenerationType.AUTO**

```
Class Book{
...
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE,generator="s1")
  @TableGenerator(name="s1",table = "My_SEQ",
  initialValue = 200000,allocationSize = 50)
```

These annotations will:
- **Create a table for auto id's if we are creating tables from Entities**
- **Map to the existing table if we are creating Entities from tables**

Auto Generation of ID's - IDENTITY

MySQL does not provide **Sequences** to generate a unique value for new Rows.
MySQL provides a construct AUTO_INCREMENT as sketched below:

```
CREATE TABLE Persons
(
    ID int NOT NULL AUTO_INCREMENT,
    Name varchar(80),
    PRIMARY KEY (ID)
)
```

This is how you Signal JPA to use this strategy for automatic key generation:

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
 private Integer id;
```

There is no way, as for the other two strategies, to provide a start value and allocation size via JPA.
Se exercises for an example script you can use to insert data without conflicting with JPA.

Composite Primary Keys

Composite primary keys can be defined in two ways:

Using an **Id Class**

```
@Entity @IdClass(ProjectId.class)
public class Project {
    @Id int departmentId;
    @Id long projectId;
       :
}


Class ProjectId {
    int departmentId;
    long projectId;
}
```

Using an **Embeddable Class**

```
@Entity
public class Project {
    @EmbeddedId ProjectId id;
      :
}

@Embeddable
Class ProjectId {
    int departmentId;
    long projectId;
}
```

The main purpose of both the **IdClass** and the **Embeddable** Class is to be used as the structure passed to the `EntityManager find()` and `getReference()` AP

Date Time and Transient Properties

```java
@Temporal(TemporalType.DATE)
private Date dateOfBirth;

@Temporal(TemporalType.TIMESTAMP)
private Date creationDate;

@Transient
private int age;
```
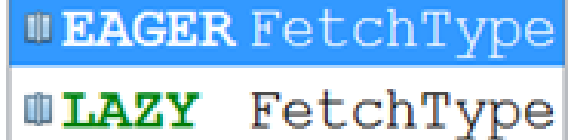
Enums

```java
public enum CustomerType {
    GOLD,
    SILVER,
    IRON,
    RUSTY
}
```

```java
public class Customer {
    ...
    @Enumerated(EnumType.STRING)
    private CustomerType customerType;

}
```

Collections and Maps of Basic Types


```java
@ElementCollection(fetch = FetchType.LAZY)
private List<String> hobbies= new ArrayList();



@ElementCollection(fetch = FetchType.LAZY)
@MapKeyColumn(name = "PHONE")
@Column(name = "Description")  //Name of the Value column
private Map<String, String> phones = new HashMap();
```

Relationships

In OO there are two ways to "connect" classes. **Composition** (almost always the right choice) and **Inheritance**
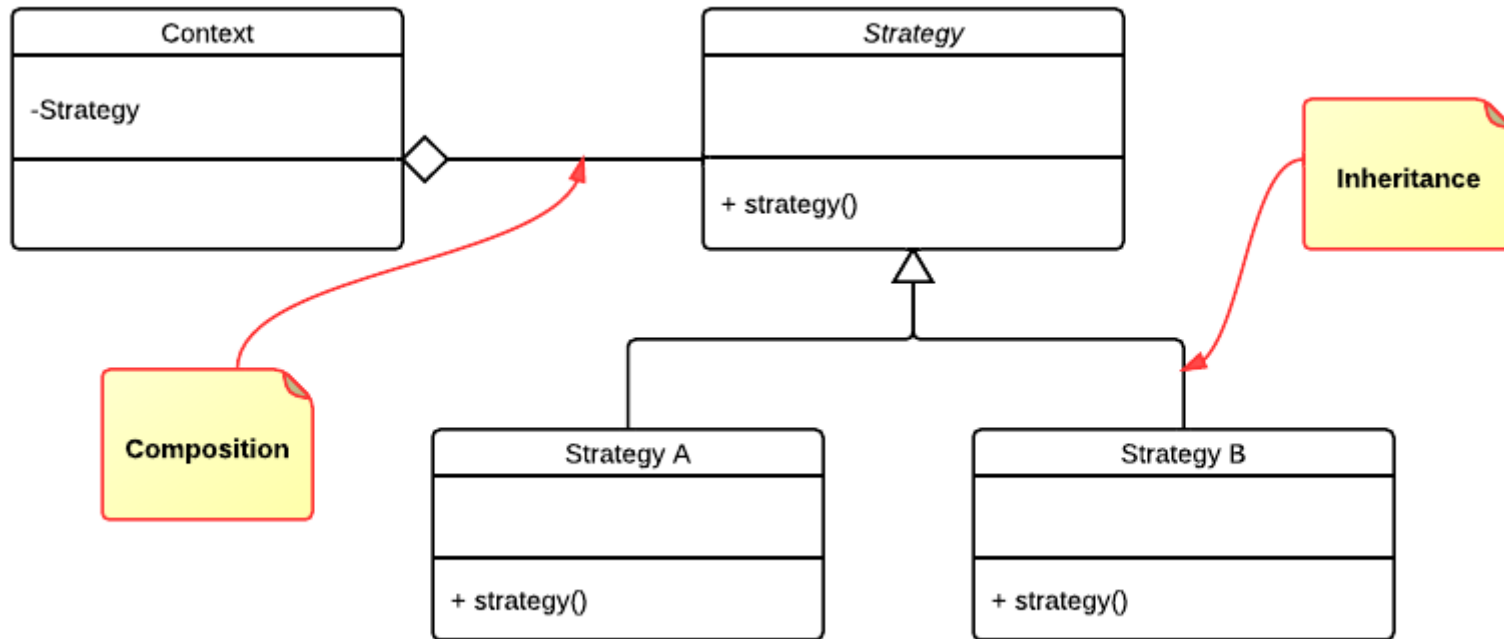
In the following we will see how these two strategies can be mapped to a DataBase, using JPA

All Possible Cardinality-Direction Combinations

Use the links below, for detailed information related to each type

Cardinality	Direction
One-to-one [details](#)	Unidirectional
One-to-one	Bidirectional
One-to-many [details](#)	Unidirectional
One-to-many	Bidirectional
Many-to-one [details](#)	Unidirectional
Many-to-one	Bidirectional
Many-to-many [details](#)	Unidirectional
Many-to-many	Bidirectional

We will investigate this in details in todays exercises

Bidirectional relationships

Rules that applies to bidirectional relationships:

The inverse side of a bidirectional relationship must refer to its owning side by use of the **mappedBy** element of the **OneToOne**, **OneToMany**, or **ManyToMany** annotation. The mappedBy element designates the property or field in the entity that is the owner of the relationship.

```
public class Customer .. {
…
  @OneToMany(mappedBy = "customer")
  private List<Address> addresses = new ArrayList();
```

```
@Entity
public class Address ..{
  private static final long serialVersionUID = 1L;
  ..
  @ManyToOne
  private Customer customer;
```

Side with the Foreign Key

Bidirectional relationships
the *mappedBy* element

The **mappedBy** element designates the property or field in the entity that is the owner of the relationship.

- The many side of one-to-many / many-to-one bidirectional relationships must be the owning side, hence the mappedBy element cannot be specified on the ManyToOne annotation.

- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.

- For many-to-many bidirectional relationships either side may be the owning side

Relationships - Lazy Fetching

The cost of retrieving and building an object's relationships far exceeds the cost of selecting the object

The solution to this issue is **lazy fetching** (lazy loading). Lazy fetching allows the fetching of a relationship to be deferred until it is accessed

Lazy fetching involves some *magic* in the JPA provider to transparently fault in the relationships as they are accessed.

```
@OneToOne(fetch =FetchType.      EAGER FetchType
@JoinColumn(name="ADDR_ID")      LAZY  FetchType
private Address address;
```

https://en.wikibooks.org/wiki/Java_Persistence/Relationships#Lazy_Fetching

Relationships - Cascading

Relationship mappings have a cascade option that allows the relationship
to be cascaded for common operations.
Cascade is normally used to model dependent relationships, such
as Order -> OrderLine.
Cascading the orderLines relationship allows for the Order's -> OrderLines
to be persisted, removed, merged along with their parent.

```java
@OneToOne(cascade={CascadeType.
@JoinColumn(name="ADDR_ID")
private Address address;
```

	ALL	CascadeType
	DETACH	CascadeType
	MERGE	CascadeType
	PERSIST	CascadeType
	REFRESH	CascadeType
	REMOVE	CascadeType

https://en.wikibooks.org/wiki/Java_Persistence/Relationships#Cascading

Inheritance

JPA defines three inheritance strategies defined from the InheritanceType enum:

- **SINGLE_TABLE**
  A single table is used to store all of the instances of the entire inheritance hierarchy.

- **JOINED**
  In joined inheritance a table is defined for each class in the inheritance hierarchy to store only the local attributes of that class.

- **TABLE_PER_CLASS**
  Each entity is mapped to its own dedicated table like the joined-subclass strategy.
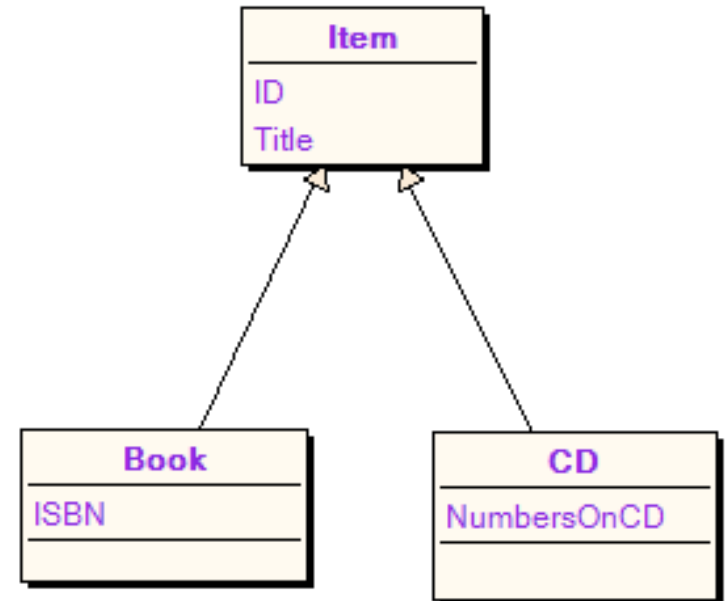 The difference is that all attributes of the root entity will also be mapped to columns of the child entity table.

Inheritance

The following slides all rely on this simple setup

```
Item item = new Item();
item.setTitle("Title for an Item");

Book book = new Book();
book.setIsbn("9780828815130");
book.setTitle("The Da Vinci Code");

CD cd = new CD();
cd.setTitle("Mylo Xyloto");
cd.setNumbersOnCD(11);
```

Inheritance - *SINGLE_TABLE*

Single table inheritance is the simplest, and default, and often the best performing solution

- A single table is used to store all of the instances of the entire inheritance hierarchy.

- The table will have a column for every attribute of every class in the hierarchy.

- A discriminator column is used to determine which class the particular row belongs to.

Item
ID
Title

Book
ISBN

CD
NumbersOnCD

#	ID	DTYPE	TITLE	ISBN	NUMBERSONCD
1	2	Book	The Da Vinci Code	9780828815130	*<NULL>*
2	3	CD	Mylo Xyloto	*<NULL>*	11
3	1	Item	Title for an Item	*<NULL>*	*<NULL>*

Inheritance - *SINGLE_TABLE*

cphbusiness
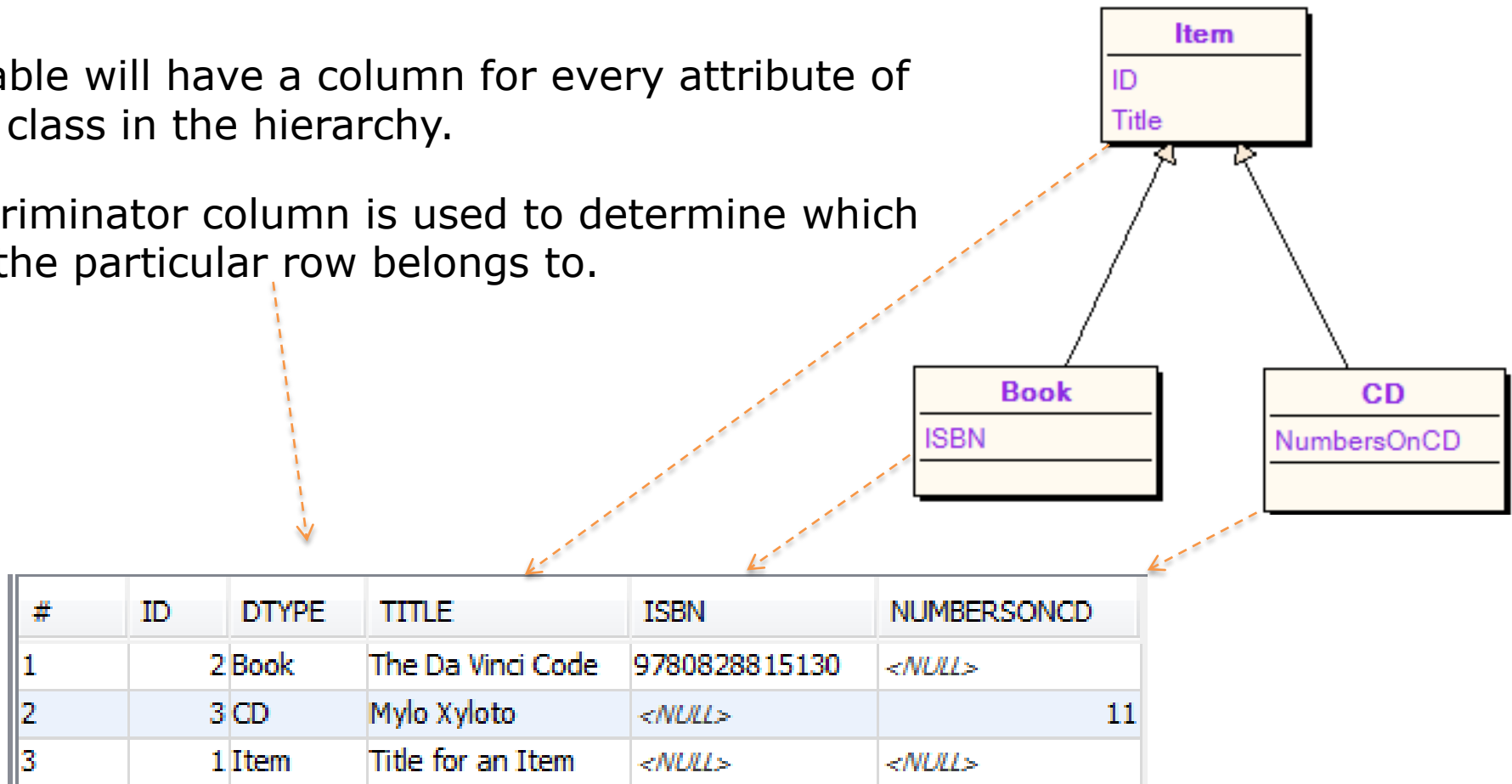
Entity Classes

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Item {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String title;
  //Getters Setters
}

  @Entity
  public class Book extends Item{
   private String isbn;
  //Getters Setters
  }


  @Entity
  public class CD extends Item {
    private int numbersOnCd;
  //Getters Setters
  }
```

This is the default, so it can be left out

Database Script

```sql
CREATE TABLE ITEM (
  ID BIGINT NOT NULLPRIMARY KEY,
  DTYPE VARCHAR(31),
  TITLE VARCHAR(255),
  ISBN VARCHAR(255),
  NUMBERSONCD INTEGER
);
```

The Discriminator column

NO @ID → its inherited from the base class

IMPORTANT
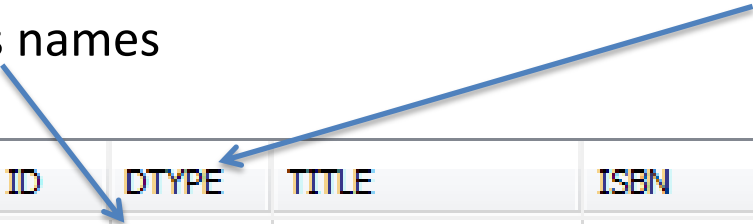
Inheritance - *SINGLE_TABLE* Pros & Cons

+ The default

+ Simple to understand

+ Works well when the hierarchy is relatively simple and stable.

- Adding new entities to the hierarchy, or adding attributes to existing entities, involves adding new columns to the table, migrating data, and changing indexes.

- All columns of the child entities must be **Nullable**.

- If mapping to an existing database schema, our table may not have a class discriminator column

The Discriminator Column

The default name for the Discriminator Column is DTYPE and the default values used are the class names

#	ID	DTYPE	TITLE	ISBN	NUMBERSONCD
1	2	Book	The Da Vinci Code	9780828815130	<NULL>
2	3	CD	Mylo Xyloto	<NULL>	11
3	1	Item	Title for an Item	<NULL>	<NULL>

As for almost everything else, we can take control over the mapping using annotations:

Adding this annotation to the base class will use the column name DT and assume column content is of type Char.

@DiscriminatorColumn(name = "DT", discriminatorType = DiscriminatorType.CHAR)

@DiscriminatorValue("I")   Add this to the **Item** class
@DiscriminatorValue("B")   Add this to the **Book** class
@DiscriminatorValue("C")   Add this to the **CD** class

ID	DT	TITLE	ISBN	NUMBERSONCD
1	B	The Da Vinci Code	9780828815130	<NULL>
2	C	Mylo Xyloto	<NULL>	11
3	I	Title for an Item	<NULL>	<NULL>

Inheritance - *JOINED*

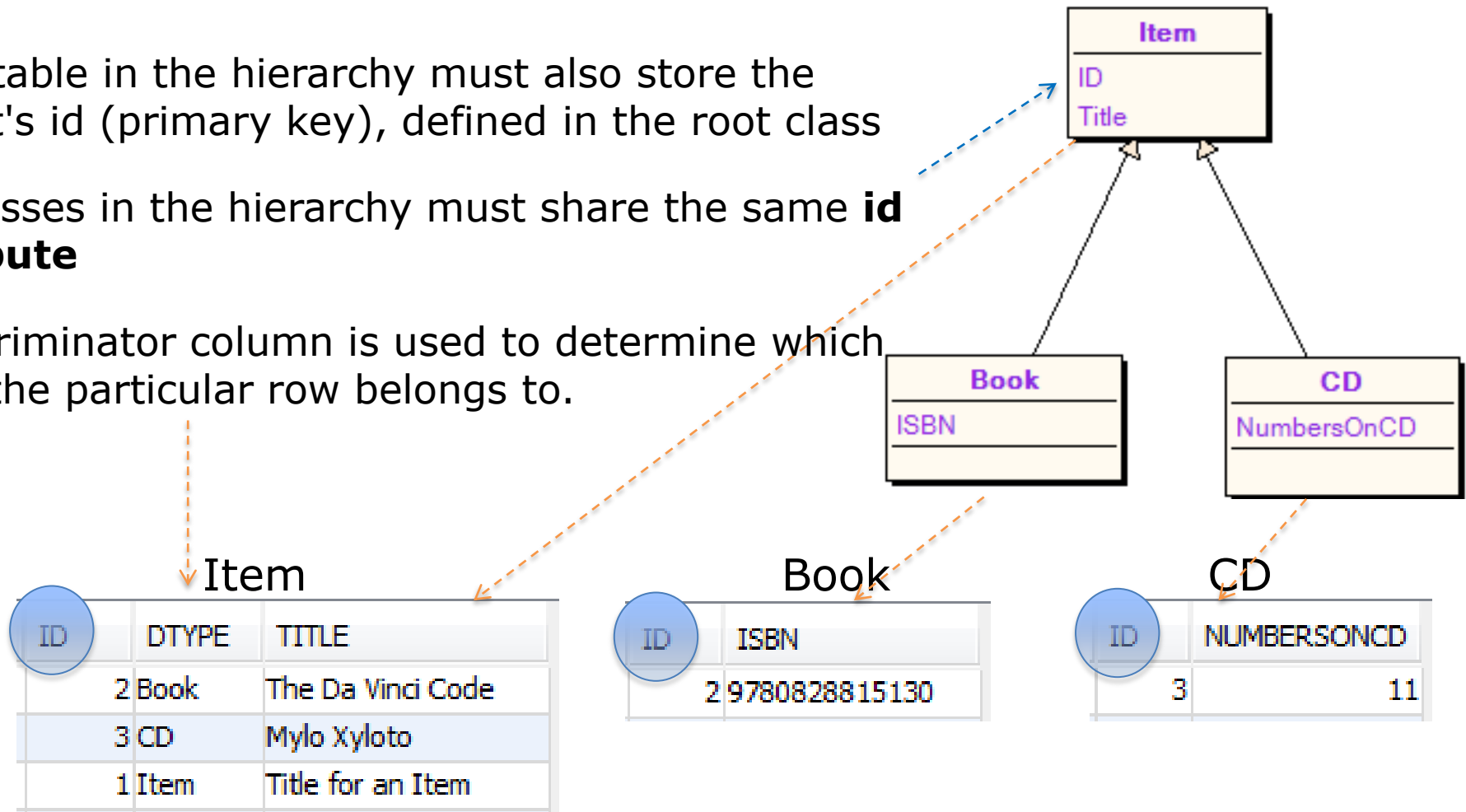Joined inheritance is the inheritance strategy that most closely mirrors the object model into the data model

- A table is defined for each class in the inheritance hierarchy to store only the local attributes of that class

- Each table in the hierarchy must also store the object's id (primary key), defined in the root class

- All classes in the hierarchy must share the same **id attribute**

- A discriminator column is used to determine which class the particular row belongs to.

Item

ID
Title

Book

ISBN

CD

NumbersOnCD

Item

ID	DTYPE	TITLE
2	Book	The Da Vinci Code
3	CD	Mylo Xyloto
1	Item	Title for an Item

Book

ID	ISBN
2	9780828815130

CD

ID	NUMBERSONCD
3	11

Inheritance - *JOINED*

Entity Classes

```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Item {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String title;
  //Getters Setters
}

  @Entity
  public class Book extends Item{
   private String isbn;
  //Getters Setters
  }


  @Entity
  public class CD extends Item {
    private int numbersOnCd;
  //Getters Setters
  }
```

Still NO @ID
Only relevant for the tables (mapping details are handled by the entity manager)

Database Script

```sql
CREATE TABLE ITEM (
  ID BIGINT NOT NULL PRIMARY KEY,
  DTYPE VARCHAR(31),
  TITLE VARCHAR(100)
);


CREATE TABLE CD (
  ID BIGINT NOT NULL PRIMARY KEY,
  NUMBERSONCD INTEGER,
  CONSTRAINT FK_CD_ID FOREIGN KEY (ID)
REFERENCES ITEM(ID)
);


CREATE TABLE BOOK (
  ID BIGINT NOT NULL PRIMARY KEY ,
  ISBN VARCHAR(14),
  CONSTRAINT FK_BOOK_ID FOREIGN KEY (ID)
REFERENCES ITEM(ID)
);
```
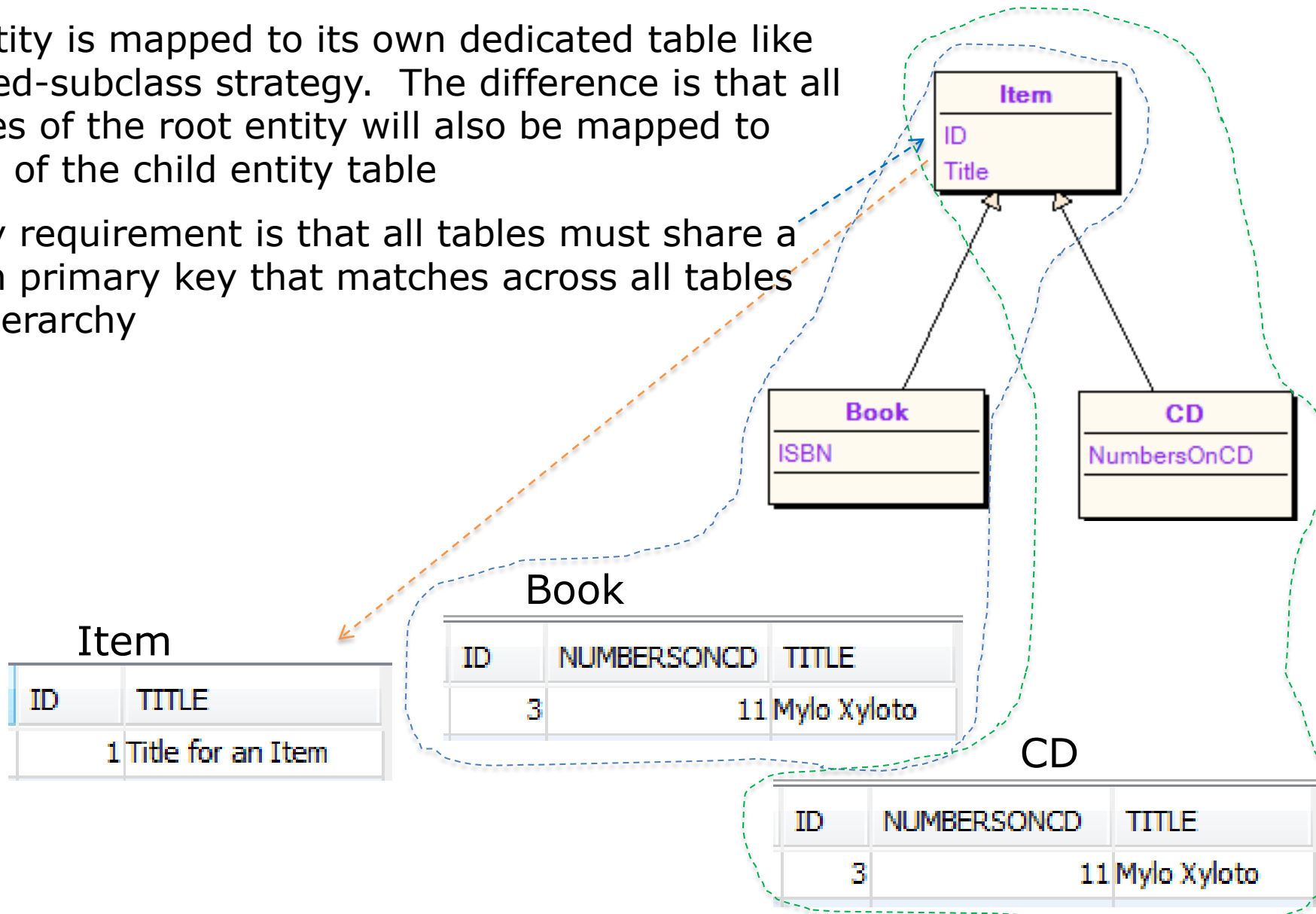
Inheritance - *JOINED* Pros & Cons

+ The joined-subclass strategy is intuitive and is close to what we know from OO-inheritance

- Querying can have a performance impact. The strategy is called joined because, to reassemble an instance of a subclass, the subclass table has to be joined with the root class table.
The deeper the hierarchy, the more joins needed to assemble a leaf entity

- If mapping to an existing database schema, our table may not have a class discriminator column

Inheritance - *TABLE_PER_CLASS*

- Each entity is mapped to its own dedicated table like the joined-subclass strategy. The difference is that all attributes of the root entity will also be mapped to columns of the child entity table

- The only requirement is that all tables must share a common primary key that matches across all tables in the hierarchy

Item
ID
Title

Book
ISBN

CD
NumbersOnCD

Item

ID	TITLE
1	Title for an Item

Book

ID	NUMBERSONCD	TITLE
3	11	Mylo Xyloto

CD

ID	NUMBERSONCD	TITLE
3	11	Mylo Xyloto

Inheritance - *TABLE_PER_CLASS*

From a database point of view, this strategy denormalizes the model and causes all root entity attributes to be redefined in the tables of all leaf entities that inherit from it.

With the table-per-concrete-class strategy, there is:
- no shared table
- no shared columns
- no discriminator column

Inheritance - *TABLE_PER_CLASS*
Pros & Cons

+ Performs well when querying instances of one entity

- Polymorphic queries across a class hierarchy more expensive than the other strategies (e.g., finding all the items, including CDs and books); it must query all subclass tables using a **UNION** operation, which is expensive when a large amount of data is involved.

- Entity Support for this strategy is optional in JPA 2.0.

Inheritance - Demo

Querying with JPA

JPA provides several querying mechanisms:

- **JPQL**
- Criteria API
- Native SQL Queries

We will focus only on

JPQL (Java Persistence Query Language)

JPQL is the query language defined by JPA. It is similar to SQL, but operates on **objects**, **attributes** and **relationships** instead of *tables* and *columns*

IMPORTANT

Query Types in JPA

There are two main types of queries in JPA:

- Named Queries
- Dynamic Queries

N**amed queries** are used for a static queries that will be used many times in the application. The advantage of a named query is that it can be defined once, in one place, and reused in the application.

Dynamic queries are normally used when the query depends on the context. For example, depending on which items in the query form were filled in, the query may have different parameters. Dynamic queries are also useful for uncommon queries.

Named Queries - Example

Declaring a Named Query (one place, reusable)

```java
@NamedQuery(
    name="findAllEmployeesInCity",
    query="Select emp from Employee emp where emp.address.city = :city"
)
public class Employee { ... }
```

Executing a Named Query

```java
EntityManager em = getEntityManager();
Query query = em.createNamedQuery("findAllEmployeesInCity");
query.setParameter("city", "Ottawa");
List<Employee> employees = query.getResultList();
```

```
Query q1 = em.createQuery("SELECT c FROM Country c");
List<Country> countries = em.getResultList();
```

Dynamic Queries – All the Details

cphbusiness

https://en.wikibooks.org/wiki/Java_Persistence/JPQL