Enterprise Computing Lecture 4

Persistence

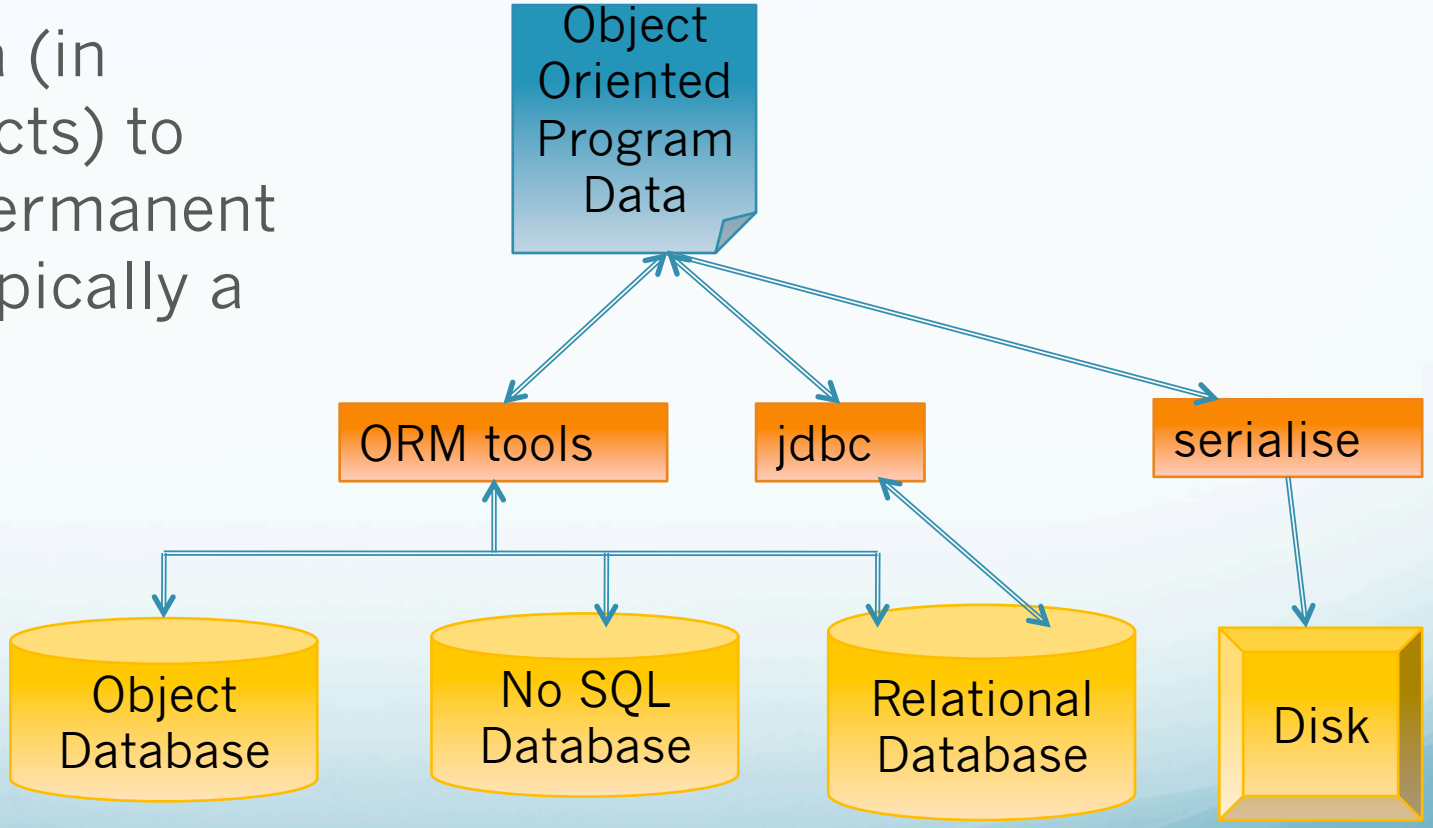
Objective

- Overview of persistence and related frameworks and technologies
- Java Persistence API
- Object Relationship Management
- Transactions
- Queries

Overview

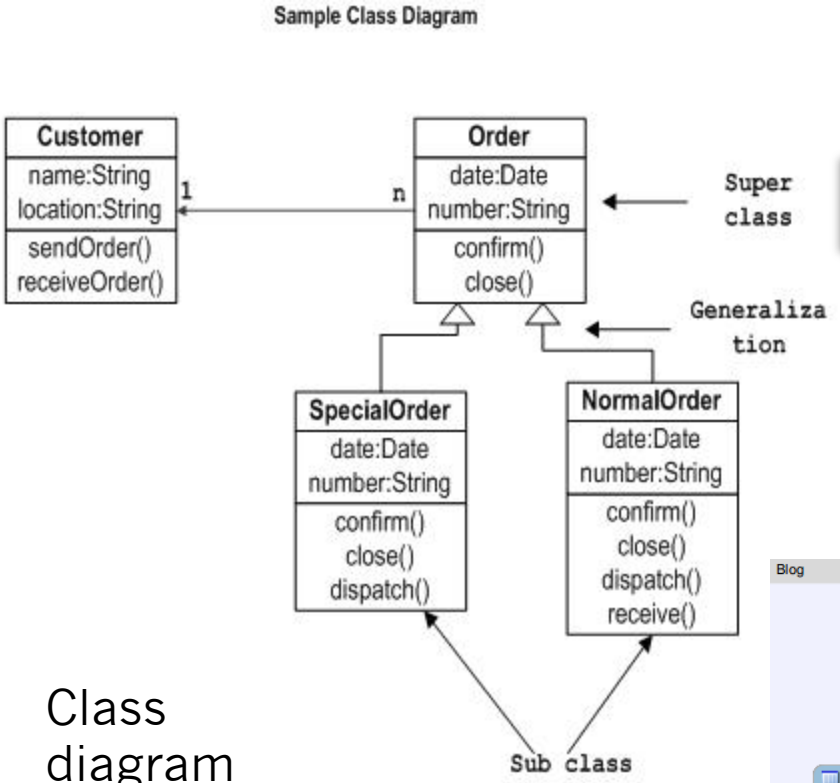
What is a persistence framework?

A persistence framework moves the program data (in memory objects) to and from a permanent data store, typically a database



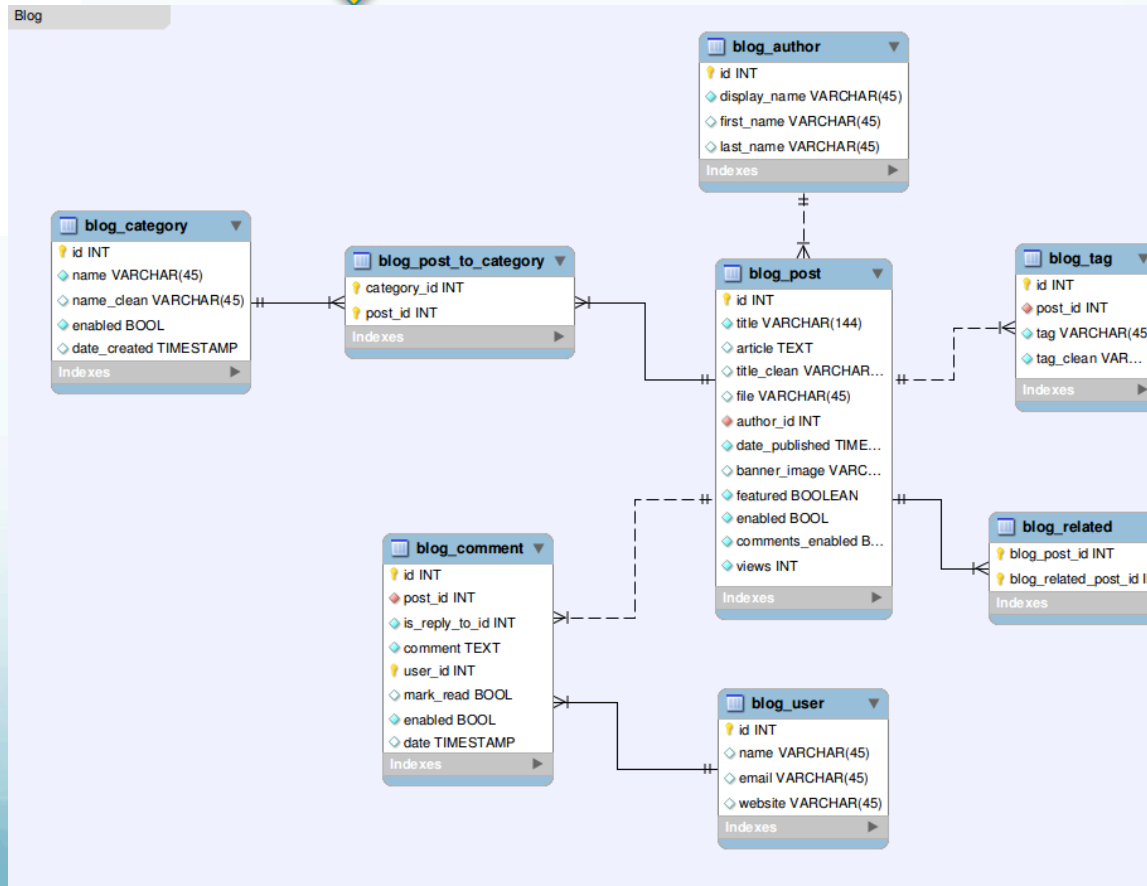
ORM tools

- The principle of **Object Relational Mapping** is to delegate access to the database to external tools which provide an object oriented view of the tables in a relational database and vice versa.
- Several frameworks provide this including:
  - Hibernate (opensource), TopLink (Oracle), Sun's Java Data Objects –JDO, and more recently **JPA** – **Java Persistence API**, which is the recommended solution for Java EE.



Class diagram

ORM tool



Java Persistence API - JPA

- The main components of JPA are:
  - **ORM**: mapping between objects and relational database tables
  - **An Entity Manager** API to perform the database operations of Create, Read, Update and Delete
  - **Callback and listeners** to hook business logic into the life cycle of a persistent object (like triggers in a database)
  - **Transaction** and locking mechanisms to support concurrent access to the database
  - **A query language**, JPQL (java persistent query language) which allows you to retrieve data using an object oriented language

Does for an object oriented language what a DBMS does for a database

Entities

What is an Entity?

- Objects that persists in a database are called Entities.
  - An object just lives in memory; an entity has its **state** saved in a database
- Entities generally match to tables in a database.
- An Entity is represented as a plain old java object (**POJO**) that is linked to a database table
  - An instance of an Entity is an instance of a Java program that is synchronised with a row in a database table.
- **Annotations** are used to map objects and attributes to database objects such as tables, attributes, primary keys, foreign keys etc.

Examples of entities:

Customer
Product
Invoice
Video
Book

Example of on Entity

```
@Entity
@Table(name = "Book")
public class Book implements Serializable {
@Id
    @GeneratedValue
    @Column
    private Integer id;
    @Column(name = "ISBN")
    private String isbn;
    @Column(name = "title")
    private String title;
    @Column(name = "author")
    private String author;
    public Book() { }
    public Integer getId() {return id; }
    public void setId(Integer id) { this.id = id;    }
    public String getIsbn() {       return isbn;    }
    public void setIsbn(Stringisbn) {       this.isbn = isbn;    }
        . . . Remaining setter & getter methods. . }
```

Corresponding SQL table definition:

```
create table Book
(id int AUTO_INCREMENT NOT NULL,
ISBN char(15),
title char(50),
author char(50),
PRIMARY KEY (id))
```

Annotation examples

@Entity – identifies a POJO as an “Entity”

@Table – maps this entity to a database table

@Id – identifies the primary key

@Column – identifies a table column

@JoinColumn– identifies a foreign key

@OneToMany– identifies the relationship type of a foreign key

@GeneratedValue– specifies this is an generated field

Full list of JPA annotations:

- http://www.objectdb.com/api/java/jpa/annotations

Entity Manager

The Entity Manager

The role of the entity manager is to:

- manage the entities;
- read from and write to a given database:
  - allow CRUD (create, read, update, delete) operations.
  - Allow complex queries to retrieve data from the database

- The code to create an entity manager, and persist a book object is as follows:

```
EntityManagerFactory  emf = Persistence.createEntityManagerFactory("name of Persistence Unit");
EntityManager   em = emf.createEntityManager();
```

Or use annotations:

```
@PersistenceContext(unitName = "name of PU")
    private EntityManager em;
```

Persistence Unit

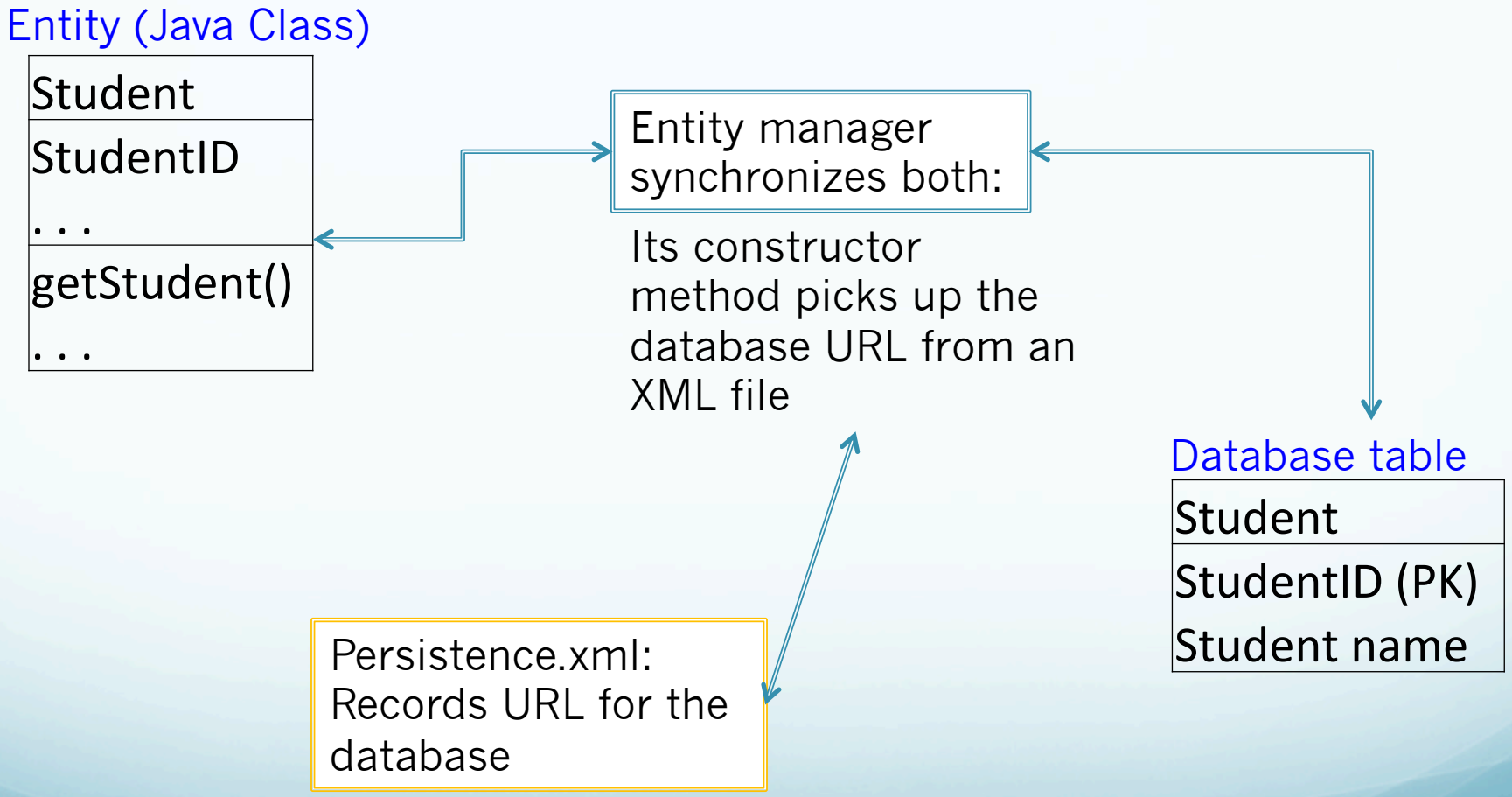
- A persistence unit is a mapping to the relational database specified in the persistence.xml configuration file.
- Creating a persistence unit adds the following code to persistence.xml

```
<persistence-unit name="ConsultingAgencyPU"
   transaction-type="JTA">
<jta-data-source>jdbc/consult</jta-data-source>
<properties/>
</persistence-unit>
```

Name of Persistence Unit

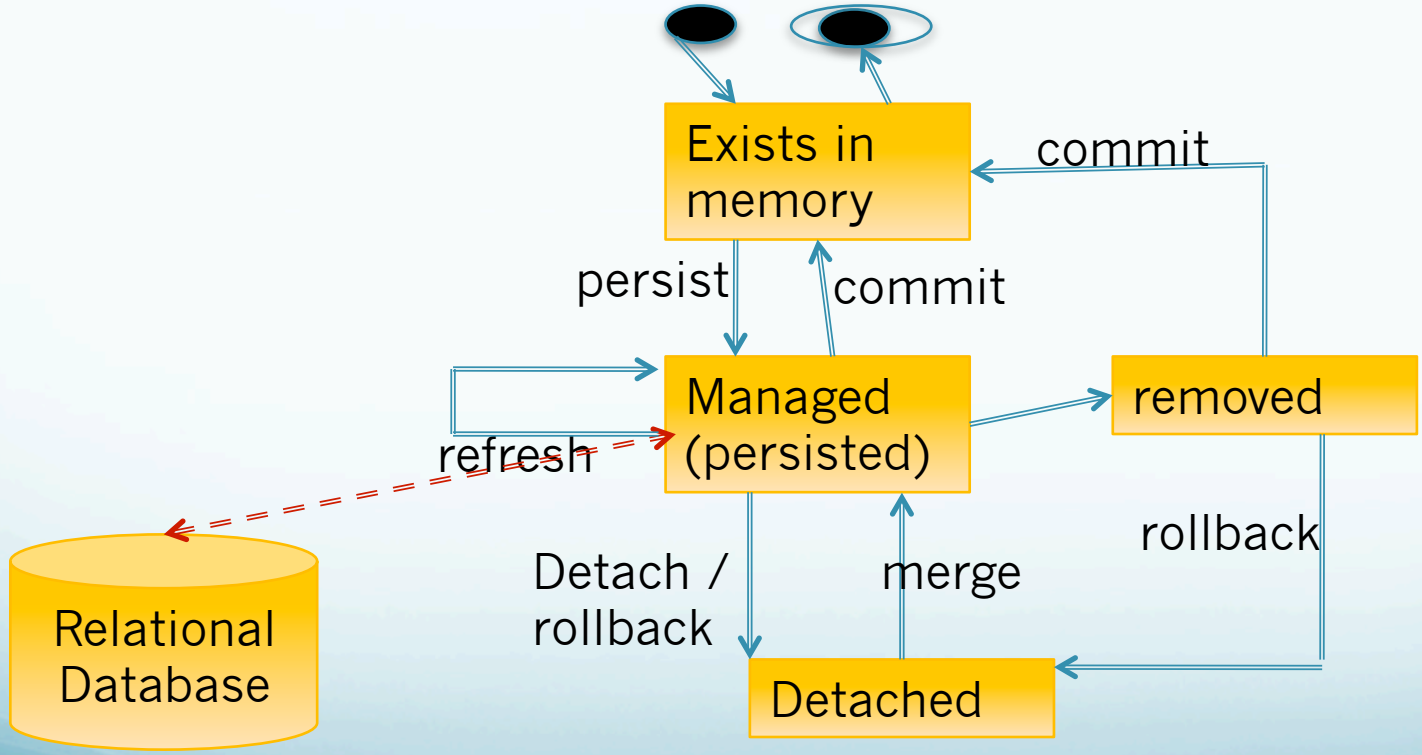
Corresponding data source

Entities and the Entity Manager



Entity Life cycle

- As with all objects, entities have a life cycle. Event specific code can be executed by the Entity Manager based on life cycle related events.



Entity Life cycle

- When an entity, such as a book, is created, it **exists in memory** only. JPA knows nothing about it. Book book=new Book();
- If that object is **managed** (persisted), its state is mapped & synchronised with a database table (e.g. book table) - em.persist(book);
- At any time, that object can be **detached** (no longer controlled by JPA) in which case is resides in memory like any other Java object - em.detach(book);
- If while being managed the object is **removed**/deleted, the data will be deleted from the database, but the object will remain in memory until garbage collection - em.remove(book);

Entity operations - **CRUD**

- Most commonly you will want to add new data, update existing data, query the data or delete data from the database. The following examples are using the book entity:
  - em.persist(book) – map the state of a new object to a database table row (**C**reate)
  - em.find(bookID)– set the state of an entity from data in the database table row (**R**ead / Select)
  - em.flush(book)– update the database table with the entities attribute values (**U**pdate / Set)
  - em.remove(book) – delete the database table row (**D**elete)

Listeners and Callback methods

- Each operation has a pre- and post- event* which can be associate with methods in the entity class (a callback method) or methods in other classes (called listeners).
  - *Except for load which doesn't have a pre- event.
- Annotations are used to associate a method with an event, for example:

```
@PrePersist
public int initialize() {        ...    }
```

Working with entities – an example

Create an instance of an entity
Persisting that instance
Updating the instance in a transaction
Implementing Locking - concurrency

The following slides are based on the Book Entity definition below:

```
@Entity
@Table(name = "Book")
public class Book implements Serializable {
      @Id
    @GeneratedValue
    @Column
    private Integer id;
    @Column(name = "ISBN")
    private String isbn;
    @Column(name = "title")
    private String title;
    @Column(name = "author")
    private String author;
    public Book() { }
    public Integer getId() {return id; }
    public void setId(Integer id) { this.id = id;    }
    public String getIsbn() {        return isbn;    }
    public void setIsbn(Stringisbn) {        this.isbn = isbn;    }
       . . . Remaining setter & getter methods. . }
```

Step 1: Create a new book object

```
Public class Main {
    Public static void main(String[] rgs) {

    Book book = new Book();
    book.setID(1234);
    book.setISBN("1234225456785")
    book.setTItle("Hitchhikers Guide to the Galaxy");
    book.setAuthor("Douglas Adams");

    OR
    Book book = new Book(1234," 1234225456785",
        "Hitchhikers Guide to the Galaxy", "Douglas Adams");
```

At this points, the book exists in memory, but the entity manager is unaware of it.

Step 2: Persist the object and save it to the database

In a container managed environment:

- The entity manager is acquired from the container using annotations
- Transactions are managed by the container, so the application does not need to issue commit and rollback statements.

In an application managed environment

- The application itself must have code to create an instance of the entity manager and transaction object
- The application issues commit and rollback statements

Step 2: Persist the object and save it to the database

In a container managed environment:

```
@PersistenceUnit(unitName
    = "BookAppPU");

private EntityManager em;

em.persist(book);
```

Note: the container issues the commit.
Em.flush(); will force updates to the database in advance of the next commit.

In an application managed environment

```
EntityManagerFactory emf =
    Persistence.createEntityMan
    agerFactory("PersistenceUni
    tXML");

EntityManager em =
    emf.createEntityManager();

EntityTransaction tx =
    em.getTransaction()

tx.begin()

em.persist(book);

tx.commit();
```

Exists in memory

Flushed to database

Glassfish – CMP (container managed persistence)

- GlassFish supports **container** managed persistence, as do many application and web servers.

Create a book object from an existing book in the database:

- The following code assumes an entity manager, **em** has already be instantiated as per the last slide.

```
Book book = new Book();
book = em.find(Book.class, 1234);
```

Find locates a book row by its primary key. 'book' is now managed.

- Make changes to the book:

```
book.setISBN("123422545678 5")
```

Changes made in memory, but not persisted in database

- Persist changes to database (optional in container manager application)

```
em.flush(book);
```

Change is now permanently persisted in the database

Flush and Refresh

- **em.flush()**: If an object is already managed (em.persist() or em.find() has been called), an em.flush() can be called at any time to update the underlying database with changes that have been made in memory.
- **em.refresh()** does the opposite. It resets all values of the entity to its current values in the underlying database, **OVER WRITING** any changes made in memory not yet persisted.

Full list of entity manager methods: http://docs.oracle.com/javaee/6/api/javax/persistence/EntityManager.html

Remove a book from the database:

Step 1: link an object to the table row

```
book = em.getReference(Book.class, id);
book.getId();
```

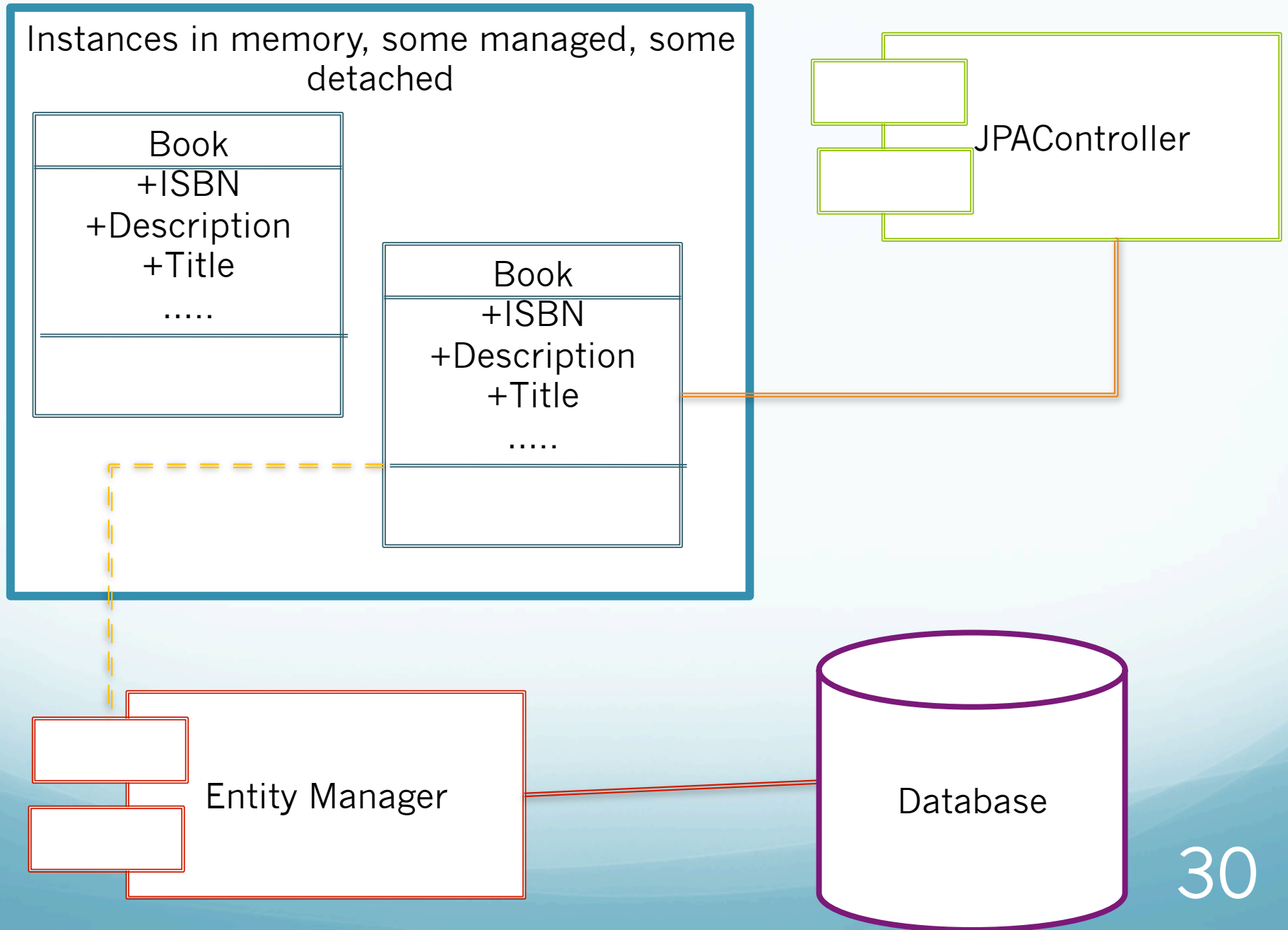
getReference is similar to find in that it links to a row in the database, but it doesn't load the data into memory (lazy fetch). The book is now managed.

```
em.remove(book);
```

If the book exists, remove it from the database. The object is no longer managed or referenced.

Assuming em is an entity manager, when would you use each of the following, and is the object managed or detach after the method is executed?

- em.persist( ):_____
- em.flush():_____
- em.find(class, id):_____
- em.remove():_____


Instances in memory, some managed, some detached
Book
+ISBN
+Description
+Title
.....
Book
+ISBN
+Description
+Title
.....
JPAController
Entity Manager
Database

OBJECT RELATIONAL MAPPING

OBJECT RELATIONAL MAPPING

Language of the Relational Model

- Tables
- Primary Key
- Columns
- Rows
- Contraints (Uniqueness)
- Foreign keys
- Indexs
- Join Tables

How do you map from one to the other?

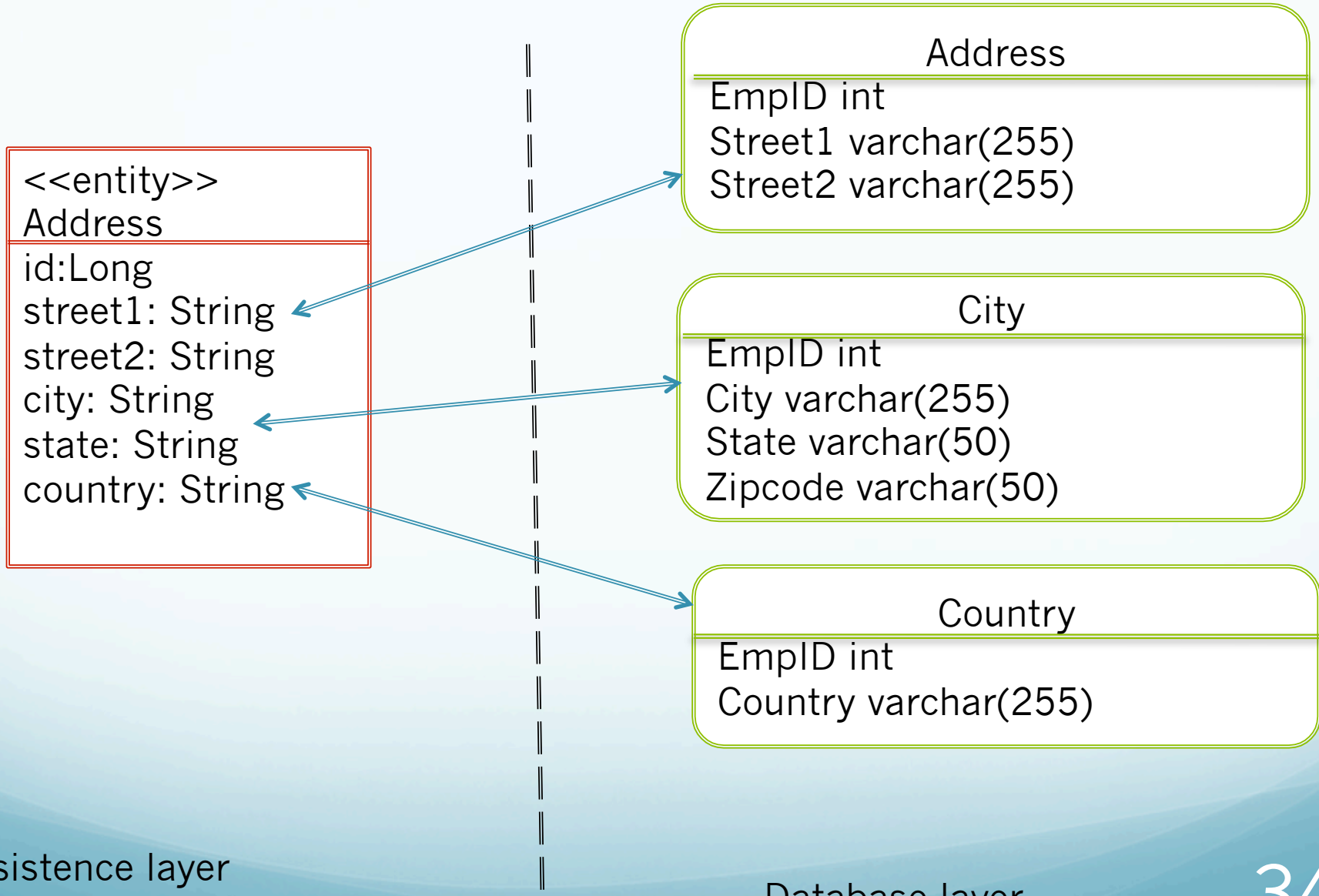
Language of the Object Model

- Objects
- Object Id's
- Attributes
- Dot notation
- embedded objects and collections
- Concrete & abstract classes
- Interfaces
- Enumerations
- Inheritance
- Annotations

1. Mapping entities to table and columns

- We have already seen @entity, @table, @column, @Id, and @JoinColumn for mapping to tables, columns, primary keys and foreign keys, which works if there is a one to one mapping between an entity and a database table
- For new systems database tables can be generated automatically from an entity class.
- With legacy databases, one to one mapping may not be feasible, in which case an entity's class may need to be mapped to a number of tables as follows . .. :

1. Mapping one entity to multiple table and columns



1. Mapping one entity to multiple table and columns

```
@entity
@table(name=Address)
@secondarytables({@(secondarytable(name="City"
    pkJoinColumns = @PrimaryKeyJoinColumn(name = "EmpID")
    ), (@secondarytable(name="Country",
    pkJoinColumns = @PrimaryKeyJoinColumn(name = "EMPID")
    ))})
Public class Address {
@Id @column
Private Long ID
@column
Private String street1
@column
private String street2
@column (table="City")
private String city
@Column(table="City")
private String State)
@column(table="City")
private String zipcode
@column(table="Country")
private String country
}
```

2. Primary keys

- We have seen mapping to simple primary keys using the @Id annotation, and generated identifiers using @GeneratedValue.
- Occasionally, tables require a composite primary key made up of two or more attributes.
  - for example news content on a web site may need to appear in different languages, so each article could have a composite primary key of two attributes: title and language.
- The primary key is therefore a composite object in its own right.
- This is handled as follows with the @Embeddable and @EmbededId annotations, and declaring the primary key structure in a separate class.

2 Primary Keys

News item **entity** class

```
@Entity
public class News {
@EmbeddedID
private NewsID id;
private String content;
//constructors, getters &
    setters
}
```

Primary key class

```
@Embeddable
public class NewsID{
private String title;
private String language;
//constructors, getters &
    setters
}
```

2 Primary Keys

- The entity class uses @EmbeddedId instead of @Id to identify the primary key.
- This identifies **NewsID** as an object rather than an attribute, which is declared in its own class with the annotation @Embeddable
- To find an entity (i.e. a particular article), first instantiate the primary key class using it's constructor, then pass this to the entity manager as follows:

```
NewsID pk = new NewsID("Clubs may revolt over tickets", "EN");
News news = em.find(News.class, pk) ;
```

Embeddables

- A class annotated with @Embeddable, like the primary key class for NewsID, doesn't have a persistent identity of its own, and can only exist as part of the 'owning' entity. It shares the identity of the owning entity.
- An entity can have a number of embeddables, not just the primary key. The entity and its embeddables all map to just one database table.

3. Relationship mapping

- In relational databases, relationships between two tables are implemented as:
  - foreign keys (1:1, or 1:m)
  - or join tables (m:n relationships)

- Relationships can be:
  - Uni-direction – one side only has a foreign key
  - Or bidirectional – each table has a foreign key to the other.

3. Relationship mapping

- In object oriented programming, associations between classes are structural, linking objects of one kind to another.
- dot (.) notation is used to navigate from one object to another, for example:
  - Order.getCustomer().getCountry() navigates from the Order object, to the Customer object to the Country attribute.
- In a **uni-directional** association, class1 would include an attribute of type class2. In a **bidirectional** association, class2 would also include an attribute of type class1.

Cardinality

- If an entity declares another entity as one of its attributes, the JPA assumes it's a uni-directional foreign key, and either OneToOne (if linking to one object) or OneToMany (if linking to a collection)
- Further specification of the relationship can be done using one of the following annotations:
  - @OneToOne, @OneToMany, @ManyToOne and @ManyToMany
- The following slides show some examples . . .

Example: OneToOne, unidriectional


```
@Entity
public class College {
@Id @GeneratedValue
Private Long id;
private String name
private Director director
//getters & setters etc
}
```

```
@Entity
public class Director{
@Id @GeneratedValue
Private Long id;
private String name
private String email
//getters & setters etc
}
```

A OneToOne mapping is assumed since Director is declared as an entity. The foreign key column will be called director_id, which is the concatenation of the name of the relationship (director) and the primary key of the destination table. The column can be renamed using @JoinColumn in College entity, e.g. @JoinColumn(name="director-fk")

Bi-directional

- In a bi-directional relationship, one side has to be the ‘owner’ of the relationship, and the other side is the ‘inverse’ of the relationship
- In a one-to-one bi-directional association:
  - The owning side has an @JoinColumn annotation
  - The inverse side has a @OneToOne(mappedby=“xxx”) annotation where mappedby refers to the ‘owning’ column.

Example: OneToOne, bi-directional


```
@Entity
public class College {
@Id @GeneratedValue
Private Long id;
private String name
@JoinColumn(name="director_fk")
private Director director
//getters & setters etc
}
```

```
@Entity
public class Director{
@Id @GeneratedValue
Private Long id;
private String name
private String email
@OneToOne(mappedBy="director")
private College college
//getters & setters etc
}
```

Example: OneToMany, unidirectional

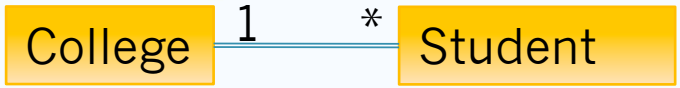

```
@Entity
public class College {
@Id @GeneratedValue
Private Long id;
private String name
private Director director
@JoinColumn(name="stu-fk")
Private Set<Student> students;
//getters & setters etc
}
```

```
@Entity
public class Student{
@Id
Private String studentId;
private String name
private String email
//getters & setters etc
}
```

OneToMany, unidirectional

- In this example, a OneToMany mapping is assumed since a collection of an entity type is being used.
- Options for collections are:
  - Collection – a group of objects
  - Set – a collection that contains no duplicates; extends collection
  - List - An ordered collection; extends collection

One To Many bidirectional




```
@Entity
public class College {
@Id @GeneratedValue
Private Long id;
private String name
private Director director
@JoinColumn(name="stu-fk")
Private Set<Student> students;
//getters & setters etc
}
```

```
@Entity
public class Student{
@Id
Private String studentId;
private String name
private String email

@ManyToOne(mappedby
="students")
Private College college
//getters & setters etc
}
```

ManyToMany bidirectional


```
@Entity
public class Module{
@Id
Private String name;
@ManyToMany
@JoinTable (name="student_module",
    joinColumns =
    @JoinColumn(name="module_fk"),
    inverseJoinColumns=@joinColumn(name="
    student_fk"))
Private Collection<Student> students
//getters & setters etc
}
```

```
@Entity
public class Student{
@Id
Private String studentId;
private String name
private String email
@ManyToMany(mappedBy="students")
private collection<Module> modules;
//getters & setters etc
}
```

A m:n relationship must define the 'join' table in the underlying database. You don't need an entity for the join table.

ManyToMany bidirectional - code explained .. .

```
@JoinTable
    (name="student_module",
    joinColumns =
    @JoinColumn(name="module_f
    k"),
    inverseJoinColumns=@JoinColu
    mn(name="student_fk"))

private Collection<Student>
    students
```

- This code is on the 'owning side' and defines the join table needed to represent the many to many relationships

```
@ManyToMany(mappedBy="s
    tudents")

private collection<Module>
    modules;
```

- On the **inverse** side, the mappedBy property points to the attribute on the owning side that defines the join table

ManyToMany bidirectional: resulting tables

Student
StudentID (PK) Name Email

Module
ID (PK) Name

Student_Module
Module_FK Student_FK

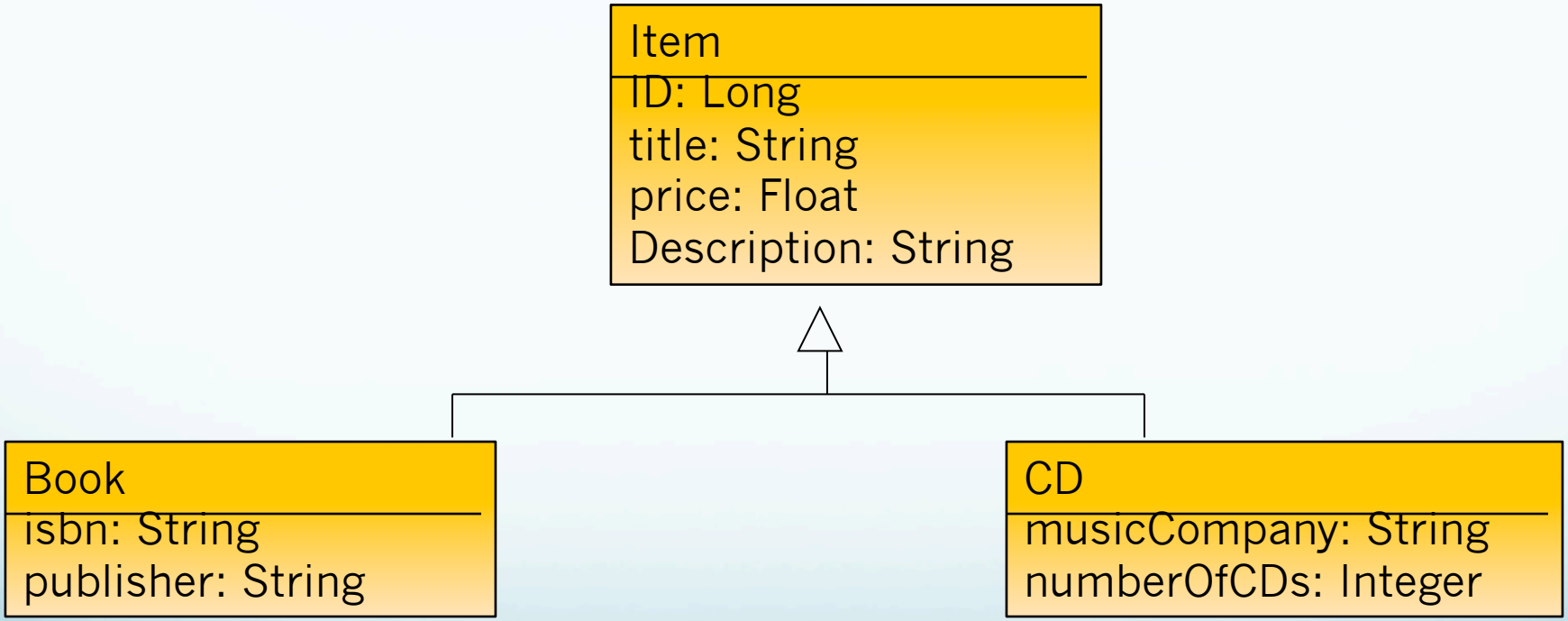
Inheritance Mapping

Inheritance Mapping

- There is no native support for inheritance in a relational database.
- There are three choices when mapping a hierarchical, inheritance based model to a flat relational model
  1. Single table strategy: One table for all classes in the entity hierarchy
  2. Joined table strategy: Concrete and abstract classes are each mapped to their own table
  3. Table per class strategy: One table for each concrete class

Inheritance Mapping

Take the following example:



1. Single table Strategy

- The default approach is a single table to hold all objects in the entity hierarchy as follows:

ID	Dtype	Title	Price	Description	Music Company	noOFCDs	Isbn	publisher
1	**Item**	Pen	2.50	Blue biro				
2	**CD**	Coldplay	21.50	Live in Phnx Park	ITB studios	2		
3	**Book**	Beginning JEE	28.60	Core text			978-1-4392...	apress
...	...							

1. Single table strategy

- Instances of **Item**, **CD** and **Book** are all added to the same table.
- Attributes which do not apply to an entity are left blank.
  - E.g. CD has NULL for ISBN
- **Dtype** is added to the table to distinguish between the types of entities in the hierarchy.
  - The values entered in Dtype default to the name of the entity.
    - This can be changed with an **@DescriminatorValue** annotation at the top of each entity.

The entity definitions would be:

```
@entity
@inheritance (strategy =
    InheritanceType.SINGLE_TABLE)
public  class Item {
@id @ generated
protected  Long id;
@column(nullable=false)
protected  String title;
protected  Float price;
protected  String description;
//constructors, getters & setters etc }
```

```
@entity
public  class Book extends
    Item {
private String isbn;
private String publisher;
//constructors, getters &
    setters etc }
```

```
@entity
public  class CD extends Item {
private String musicComany;
private Integer noOfCDs;
//constructors, getters & setters etc}
```

1. Single table strategy

- The parent class indicates the strategy to be used with the @inheritance annotation. In this case the strategy to be used is **SINGLE_TABLE**
- The child classes **EXTENDS** the parent class, linking them to the parent class, and the inheritance strategy defined in the parent class
- Note: (strategy = InheritanceType.SINGLE_TABLE) is the default inheritance strategy if a strategy is not specified.

1. Single table strategy

- ✓ The single table strategy is the easiest to understand,
- ✓ It doesn't require table joins to access the data.
- ✓ Good for a straight forward, stable, entity hierarchy.
- x It can be difficult to add new entities to the hierarchy
- x All columns in the child entities must be nullable, i.e. they can not be defined as mandatory.

2. Joined table strategy

- In the joined strategy, each entity maps to its own table. The root entity maintains the primary key to be used by all tables, as well as inherited attributes and DType.

Item

ID	Dtype	Title	Price	Description
1	**Item**	Pen	2.50	Blue biro
2	**CD**	Coldplay	21.50	Live in Phnx Park
3	**Book**	Beginning Jee	28.60	Core text
...	...			

CD

ID	Music Company	noOFCDs
2	ITB studios	2

Book

ID	Isbn	publisher
3	978-1-4392...	apress

The code to implement this strategy requires just ONE change to the **@inheritance** annotation to define the strategy to be used:

```
@entity
@inheritance (strategy =
    InheritanceType.JOINED)
public  class Item {
@id @ generated
protected  Long id;
@column(nullable=false)
protected  String title;
protected  Float price;
protected  String description;
//constructors, getters & setters etc }
```

```
@entity
public  class Book extends
    Item {
private String isbn;
private String publisher;
//constructors, getters &
    setters etc }
```

```
@entity
public  class CD extends Item {
private String musicComany;
private Integer noOfCDs;
//constructors, getters & setters etc}
```

2. Joined table strategy

- ✓ The joined table strategy is intuitive as each entity maps to a table.
- ✓ A new entity can be added with out changing the definition or existing tables. (e.g. entity: DVD)
- x Accessing all attributes of a particular object will **ALWAYS** need a **join** between the entity class and the parent class, making database access inefficient. The deeper the hierarchy, the more joins required.

3. Table per class strategy

- This approach creates a table for each CONCRETE class in the hierarchy, with all attributes for an entity located in its table, including inherited attributes. This would create the following three tables:

Item

ID	Dtype	Title	Price	Description
1	**Item**	Pen	2.50	Blue biro
2	**CD**	Coldplay	21.50	Live in Phnx Park
3	**Book**	Beginning Jee	28.60	Core text
...	...			

3. Table per class strategy

CD

ID	Dtype	Title	Price	Description	Music Company	noOFCDs
2	**CD**	Coldplay	21.50	Live in Phnx Park	ITB studios	2

Book

ID	Dtype	Title	Price	Description	Isbn	publisher
3	**Book**	Beginning Jee	28.60	Core text	978-1-4392...	apress

The code to implement this strategy also requires just ONE change to the @inheritance annotation to define the strategy to be used:

```
@entity
@inheritance (strategy =
    InheritanceType.TABLE_PER_CLASS)
public  class Item {
@id @ generated
protected  Long id;
@column(nullable=false)
protected  String title;
protected  Float price;
protected  String description;
//constructors, getters & setters etc }
```

```
@entity
public  class Book extends
    Item {
private String isbn;
private String publisher;
//constructors, getters &
    setters etc }
```

```
@entity
public  class CD extends Item {
private String musicComany;
private Integer noOfCDs;
//constructors, getters & setters etc}
```

3. Table per Class Strategy

- ✓ Access to items in the tables no longer requires a join.
- ✓ A new entity can be added without changing the definition or existing tables. (e.g. entity: DVD)
- x This approach denormalises the model – all root entity attributes are repeated in the tables of the leaf entities.
  - Note: annotating **Item** as a **@MappedSuperclass** rather than **@Entity** means item would not have its own table, but attributes inherited from Item would appear in the Book and CD tables. This would remove the need for duplicating attributes provided ITEM is an abstract class.

More on transactions – concurrency

Handling concurrency

- JPA supports concurrency, but does not implement concurrency by default.
- If an application is likely to have multiple users updating the same data on the database, the application must specify that locking is to be used.
- There are two options:
  - Optimistic locking – assume conflicts will not arise. Acquire the lock during COMMIT. If the database has been updated by another user since the last read, ROLLBACK the transaction.
  - Pessimistic locking – assume conflicts will arise. Acquire a lock when the data is read, and do not release the lock until a COMMIT is issued. This degrades performance, but is necessary if the chance of conflict is high.

Optimistic Locking

- Optimistic locking is implemented by maintaining a version number with each entity.
- When an update/commit is issued, the version number is checked against the current version number on the database. If the database version number is higher, the transaction is ROLLEDBACK.

Code:

Entity class:

```
@Entity
public class Book {
@Id @GeneratedValue
Private Long id;
@Version
private Integer version;
. . . . . .
```

Transaction:

```
book = em.find(Book.class, 1234);

em.lock(book, LockModeType.OPTIMISTIC);
book.setISBN("123422545678")
em.update(book)
.
```

OptimisticLockException is raised if the version numbers are not in sync

Pessimistic Locking

- Pessimistic locking does not require a version number, so no additional annotations are required in the entity class. The lock is acquired as follows:

```
book = em.find(Book.class, 1234);
em.lock(book, LockModeType.PESSIMISTIC);
book.setISBN("1234225456785")
em.update(book)
```

Querying the database

Main Options:

1. JPQL - Java Persistence Query Language, builds a query as a string, much like SQL. Advantage: easier to use
2. CriteriaBuilder – instantiating Java objects that represent elements of the query. Advantage: errors detected at compile time

JPQL

- JPQL is based on the syntax of SQL with one important difference:
  - SQL returns rows and columns
  - JPQL returns entities (objects) and collections of entities
- JPQL syntax is objected oriented. The domain model is navigated using **dot notation**.
- Under the hood, all JPQL queries are translated into SQL and executed using JDBC calls. Entity instances have their attribute values set, based on the SQL results returned.

Some simple queries:

- SELECT c FROM Customer c
- SELECT c.firstname, c.lastname FROM Customer c
- SELECT DISTINCT c.firstname FROM Customer c WHERE c.age BETWEEN 16 AND 65
- SELECT c.firstname FROM Customer c WHERE c.email LIKE '%gmail.com'
- SELECT c.firstname FROM Customer c WHERE c.country IN ('USA', 'Ireland')

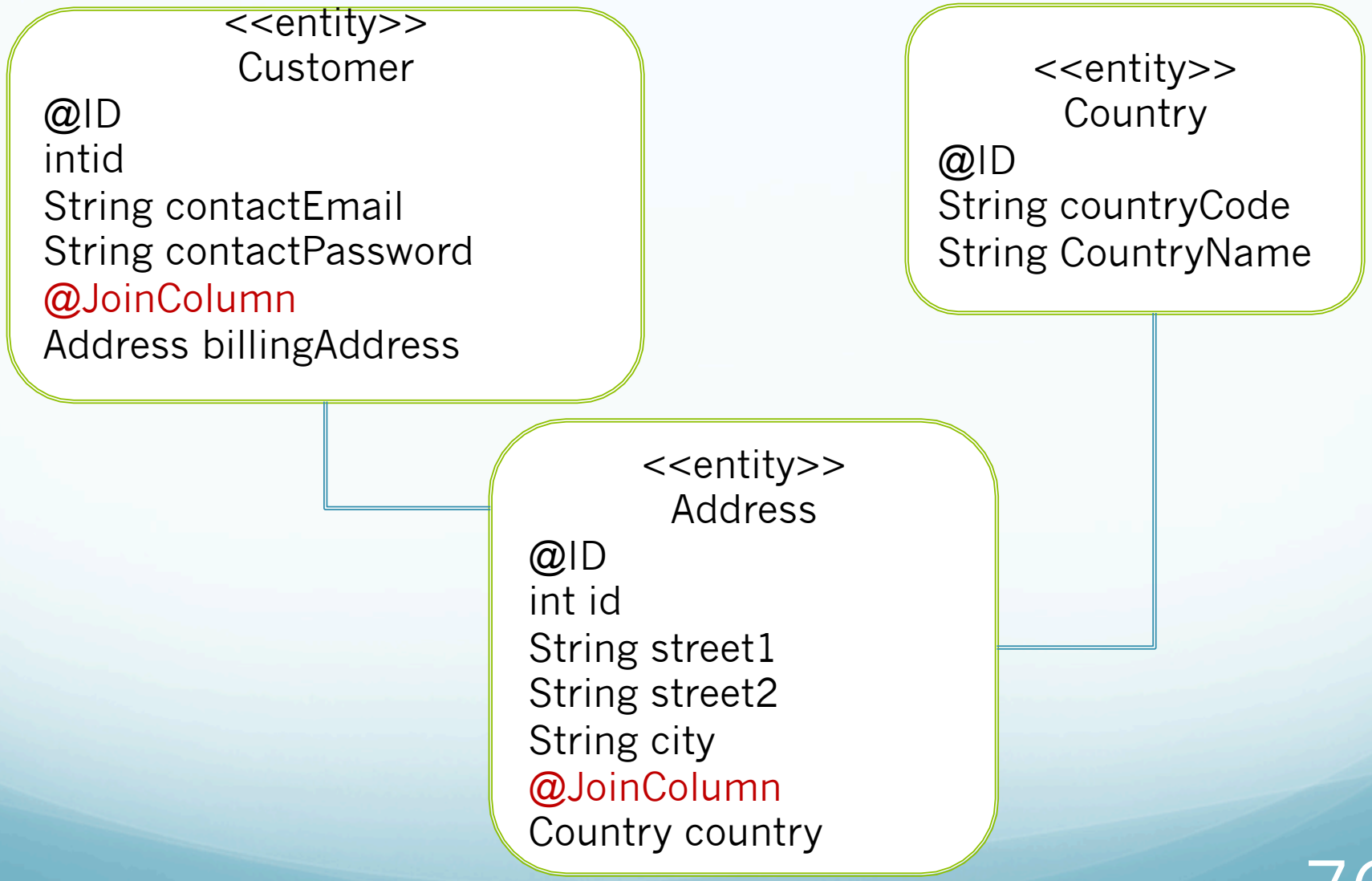
Some simple queries:

- SELECT COUNT(c) FROM Customer c
- SELECT c.county, COUNT(c) FROM Customer c GROUP BY c.county
- SELECT max(c.age) FROM Customer c
- SELECT c FROM Customer c WHERE age > 18 ORDER BY c.age DESC

Some simple queries:

- DELETE FROM Customer c WHERE c.age<18
- UPDATE Customer c SET c.firstname="TOO YOUNG" WHERE c.age< 18

Navigating objects



Navigating objects

Assuming the entity model on the last slide represents OneToOne relationships, table joins are coded using dot notation as follows:

- SELECT c.contactEmail, c.billingAddress.street1
  FROM Customer c
- SELECT c.contactEmail,
  c.billingAddress.country.countryName
  FROM Customer c

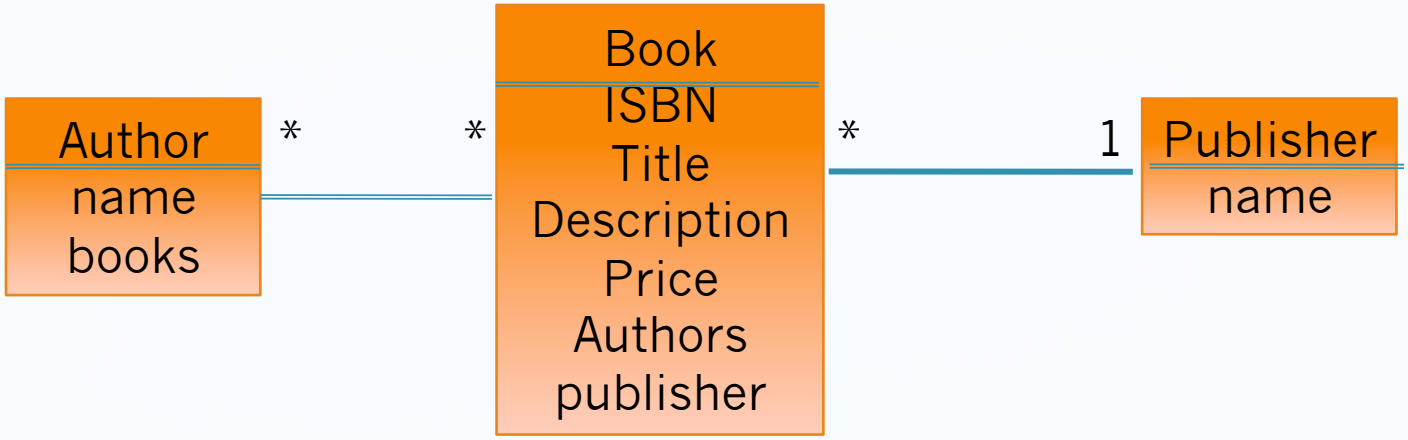
Navigating objects

- Dot notation can navigate from OneToOne or ManyToOne, but can not navigate to a collection (OneToMany or ManyToMany).
- To handle such navigation, use a join operator as follows:

- Select all authors who have had books published by "XYZ Press" :

```
SELECT DISTINCT a
 FROM Author a INNER JOIN a.book b
 WHERE b.publisher.name = 'XYZ Press'
```

Write the JPQL for the following:



- Return the title and description for the Book whose ISBN is '12335'
- Return all ISBNs for books cheaper that €20
- Return the publisher's names for books with JEE in their title
- Return the author's names for books with JEE in their title

Methods to execute a query

- getResultList() – for queries that return a collection of entities
- getSingleResult() – for queries that return a single entity
- executeUpdate() – for queries that make changes to the database (update, delete)

Building queries dynamically

1. A simple query to return all customers:

```
Query query = em.createQuery("SELECT c FROM Customer
    c");
List<Customer> customers = query.getResultList();
```

2. The query string can be built up dynamically as follows:

```
String jpqlQuery= "SELECT c FROM Customer c";
if (check some condition)
jpqlQuery+= " WHERE c.firstName="Joe";
Query query = em.createQuery(jpqlQuery);
List<Customer> customers = query.getResultList();
```

Binding Attributes

- JPQL query can accept two types of parameters:

1. Positional parameters, designated by a **question mark ?**

- SELECT c FROM Customer c WHERE c.firstname = ?1

2. Named parameters, designated by a **colon:**

SELECT c FROM Customer c WHERE c.firstname = :fname

Adding parameters

3. Adding positional parameters:

```
Query query = em.createQuery("SELECT c FROM
    Customer c WHERE c.firstname = ?1");
query.setParameter(1, Joe);
List<Customer> customers = query.getResultList();
```

4. Adding named parameters:

```
Query query = em.createQuery("SELECT c FROM
    Customer c WHERE c.firstname = :fname");
query.setParameter("fname", Joe);
List<Customer> customers = query.getResultList();
```

Building JPQL into an applications

There are fours ways to code JPQL statements, each with a different purpose:

1. Dynamic queries: a JPQL query string dynamically specified at run time, generally in the session bean
2. Named queries – static JPQL query string added to the entity bean. Optimised at compile time.
3. Native queries – native SQL queries
4. Criteria API – new more recent API that is object based rather than using a query string.

Dynamic versus Named queries

- Dynamic JPQL queries are translated in SQL at run time, which has cost implications.
- Some queries need to be dynamic, because it will not be known until run time what the parameter values will be, or possibly what the structure of the query itself will be.
- If a query is known in advance, then it is more efficient to implement it as a Named Query in the entity class. The persistence provider translates named queries once the application starts.
- NOTE: Each named query in an application must have a unique name, and is created using the @NamedQuery annotation.

Example 1 of using a named query

Entity class:

```
@Entity
@NameQuery (name="findAllCustomers",
   query="SELECT c FROM Customer c");
Public class Customer { . . . . . }
```

Using this in the application:

```
Query query =
   em.createNamedQuery("findAllCustomers");
List<Customer> customers = query.getResultList();
```

Example 2 of using a named query

Entity class:

```
@Entity
@NamedQuery(name="findAllCustomersWithName",
query="SELECT c FROM Customer c WHERE c.name
LIKE :custName")
```

...

Session bean:

```
Query query =
    em.createNamedQuery("findAllCustomersWithName"
    );
query.setParameter("custName", "Smith")
List<Customer> customers = query.getResultList();
```

Note: the value for customer name is hardcoded in the session bean.

Short hand . . .

Note: the query in the session bean on the last slide could also have been written as one line of code as follows:

```
customers =
em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName", "Smith")
    .getResultList();
```

Adding this query to the book entity gives . . .

```
@Entity
@Table(name = "Book")
@namedquery (name="findBookByTitle", query="SELECT b FROM book
    WHERE b.title='H1N1'")
public class Book implements Serializable {
    @Id
    @GeneratedValue
    @Column
    private Integer id;
    @Column(name = "ISBN")
    private String isbn;
    @Column(name = "title")
    private String title;
    @Column(name = "author")
    private String author;
    public Book() { }
    public Integer getId() {return id; }
    public void setId(Integer id) { this.id = id;    }
    public String getIsbn() {       return isbn;    }
    public void setIsbn(Stringisbn) {       this.isbn = isbn;    }
       . . . Remaining setter & getter methods. . }
```

Calling the query from a Session bean:
List books =
em.createNamedQuery("findBookByTitle
");

Example of using a Native SQL query

JPQL queries are portable across databases.
However if you need to use a specific, non-portable, feature of a database, for efficiency you can specify the query in SQL as follows:

```
Query query = em.createNativeQuery("SELECT *
    FROM t_Customer, Customer.class");
List<Customer> customers = query.getResultList();
```

Criteria API

- The criteria API is designed to support more complex query construction, and is particularly suited to building dynamic queries in session beans.
- Queries are written in Java using programming API's.
- The queries themselves can be run over any type of data store, not just SQL based databases.
- Main advantage: it's a typesafe API (data types are known at compile time as query is composed from objects rather than a query string)
- Main disadvantage: more complex

More information is available here:
http://docs.oracle.com/javaee/7/tutorial/doc/persistence-criteria.htm#GJITV

Criteria API

- Steps:
  1. Create a CriteriaBuilder via an Entity Manager method: CriteriaBuilder cb = em.getCriteriaBuilder();
  2. Use the CriteriaBuilder to create an instance of a criteria query: CriteriaQuery<*Book*> cq = cb.createQuery(Book.class);
  3. Add clauses to the query
     - from is the **query root**, and defines the object(s) to be queried: Root<*Book*> myObject = cq.from(Book.class);
     - select defines the return type: cq.select(book);
     - Other methods define additional clauses, as per SQL
  4. Create the query
     - createQuery constructs a query based on the clauses specified: Query q = em.createQuery(cq);
  5. Run the query
     - getResults runs the query and returns the results: q.getResultList();

Putting it all together

Criteria API:

```
EntityManager em = ...;

CriteriaBuilder cb =
em.getCriteriaBuilder();

CriteriaQuery<Book> cq =
cb.createQuery(Book.class);

Root<Book> book =
cq.from(Book.class);

cq.select(book);

TypedQuery<Book> q =
em.createQuery(cq);

List<Book> allBooks =
q.getResultList();
```

JPQL:

```
EntityManager em = ...;

Query query =
      em.createQuery("SELECT b
      FROM Book b");

List<Book>allBooks=
      query.getResultList();
```

Criteria API – query root

- All queries must have a root, which is the object that all navigation will originate from.
- A query will generally have just one root, defined using the from method, but it may have more than one root if the query navigates from more than one entity.

```
CriteriaQuery<Book> cq = cb.createQuery(Book.class);

Root<Book> book = cq.from(Book.class);

Root<Book> book2 = cq.from(Book.class);
```

- Any object the query navigates to must be specified in the JOIN method of the root object as follows:

```
Join<Book, Author> author= book.join(book.author);
```

Criteria API – select

- The select method of CriteriaQuery defines what is returned by the query which can be an object:

```
cq.select(book);
```

- An attribute of that object, specified using the get method of the root object as follows:

```
cq.select(book.get("name"));
```

- Or a collection of attributes as you might do in an SQL query

```
cq.multiselect(book.get("name"), book.get("author"));
```

Criteria API – other clauses

- As with SQL or JPQL, other clauses can be added. These are done via CriteriaBuilder methods:
- cq.where(cb.equal(book.get(“Author”), ”Kamber"));
- cq.where(book.get(“Type”).isNull());
- cq.where(book.get(”keyWord”).in(”JPQL”, ”JEE"));
- .and(cb.gt(book.get(“pubDate”), date));
- cq.orderBy(cb.desc(book.get(“Price”))));
- cq.groupBy(book.get(“publisher”));

Criteria API

Example from NewsApp:

```
public List<NewsEntity> findRange(int[] range) {
    CriteriaQuery cq =
    em.getCriteriaBuilder().createQuery();
     cq.select(cq.from(NewsEntity.class));
     Query q = em.createQuery(cq);
     q.setMaxResults(range[1] - range[0]);
     q.setFirstResult(range[0]);
     return q.getResultList();}
```

Criteria API

Javadocs for:

- CriteriaBuilder methods: http://docs.oracle.com/javaee/6/api/javax/persistence/criteria/CriteriaBuilder.html
- CriteriaQuery methods: http://docs.oracle.com/javaee/6/api/javax/persistence/criteria/CriteriaQuery.html
- All classes listed here, select the package: javax.persistence.criteria

Exercise on JPAall (lab 3)

- Objective: add a query to the JPAapp creates in Lab3 to implement a filter by course in the student/list.xhtml page.

Select by course:

- Using the lab3-code handout, answer the following:

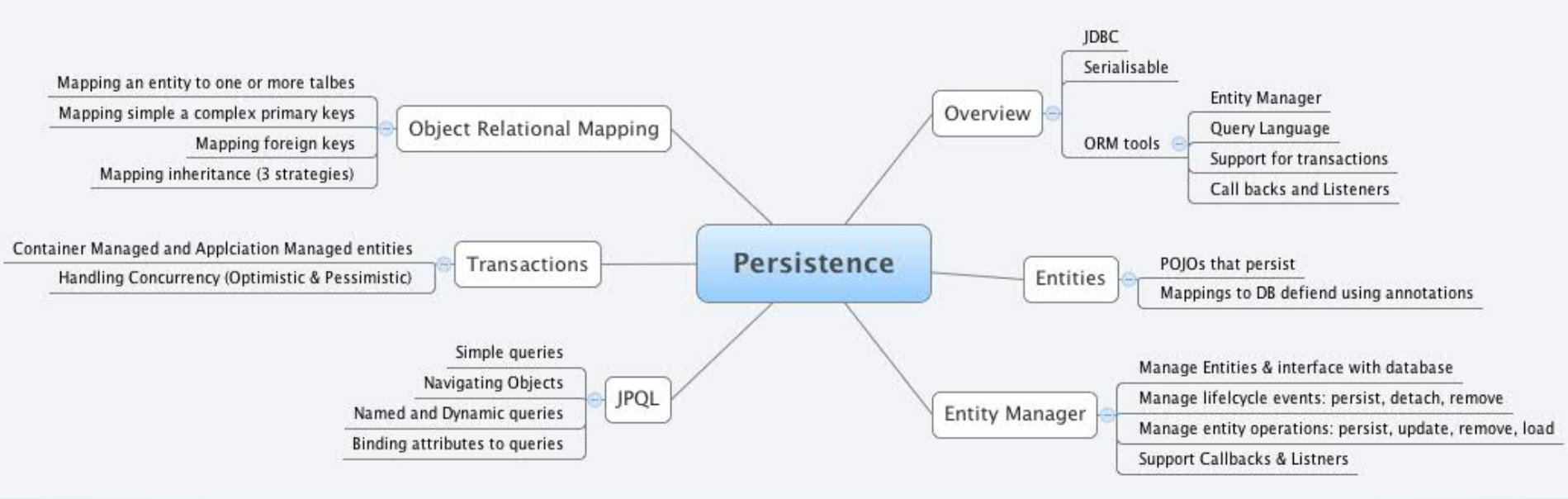
1. In list.xhtml, how is the dataTable populated?

2. Where can you change howdataTable it is populated to check if the filter is being used?

3. Write the query to implement the filter.

Note: For this exercise, ignore the fact that the result of the query needs to be paginated when creating the new query.

Summary (see mindmap gif on moodle)



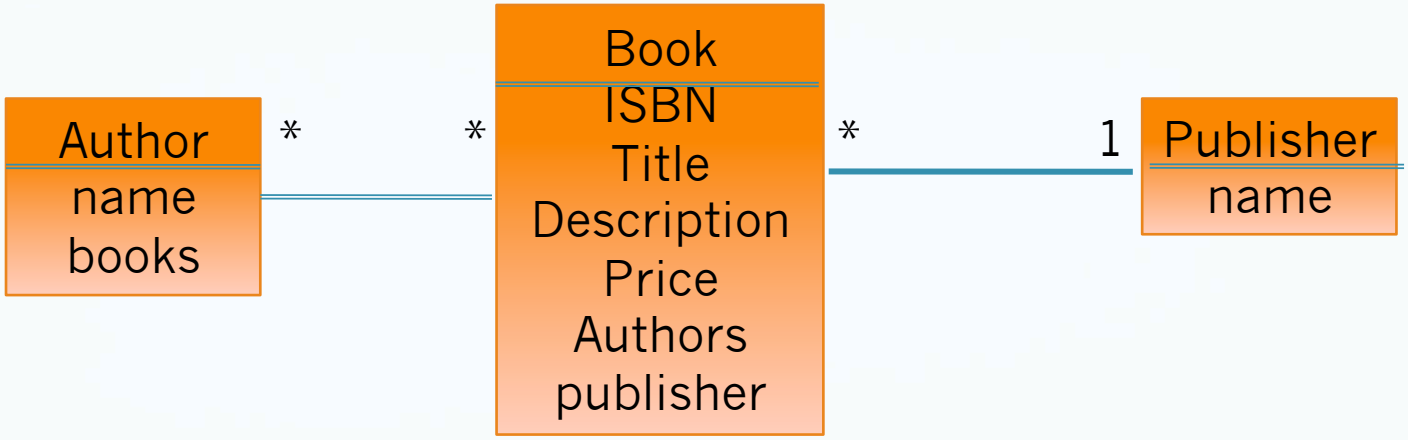
Solution slide 66: Assuming em is an entity manager, when would you use each of the following, and is the object managed or detach after the method is executed?

- Persist – changes the status to managed, generally used to create a new row in the database from an instance
- Flush – synchonised a managed instance with the database
- Find – create a new managed instance from an existing row in the database table
- Remove – remove a row from the database table linked with the managed instance, after which it is not managed (removed)

Other commands:

- Merge – synchonised an unmanaged instance with an existing row in the database, changing its status to managed
- Commit – commit a transaction (series of updates), after which the entity is detached

Solution: Write the JPQL for the following:



- Return the title and description for the Book whose ISBN is '12335'

```
SELECT b.title, b.description FROM book b WHERE
    b.IBSN='12335'
```

- Return all ISBNs for books cheaper that €20

```
SELECT b.ISBN FROM book b WHERE b.price< 20.0
```

- Return the publishers names for book with JEE in their title

```
SELECT b.publisher.name FROM book b WHERE b.title like
    '%JEE%'
```

- Return the authors names for books with JEE in their title

Solution to Exercise on JPAapp

The following solution does not use pagination, and to would need to be changed to a range query for pagination.

Student controller needs to call findByCourse rather than findRange.
SESSION BEAN:

```
public List<entity.Student> findByCourse(String selectedCourse) {
//Calling a named querys defined in the entity bean
        if (selectedCourse.contentEquals(null) || selectedCourse.isEmpty()) {
            Query query = em.createNamedQuery(”Student.findAll");
            return query.getResultList();
        } else {
            Query query = em.createNamedQuery(”Student.findByCourse");
            query.setParameter(”course", selectedCourse);
            return query.getResultList() } }
```

ENTITY BEAN:

```
@NamedQueries({
    @NamedQuery(name = ”Student.findAll", query = "SELECT s FROM
Student s"),
    @NamedQuery(name = ”Student.findByCourse", query = "SELECT s
FROM Student s WHERE s.course.id = :course")})
```