JPA / Hibernate

JOHN BRYCE
Leading in IT Education
a matrix company

JPA

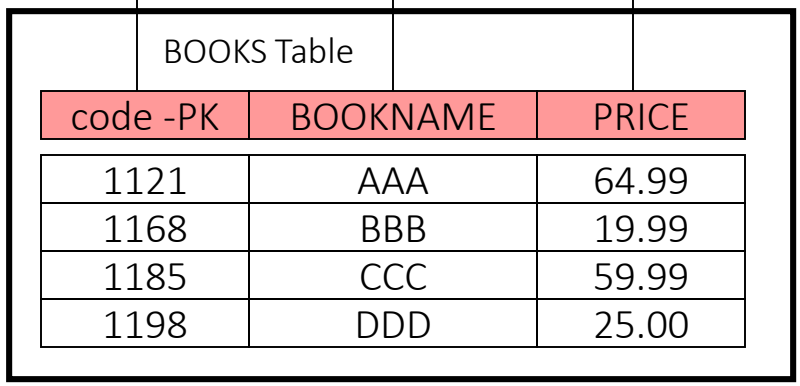
JOHN BRYCE
Leading in IT Education
a matrix company

- Represent data in memory as objects – Object / Relational mapping
- Usually one table row or related data from several tables
- POJOs are used and mapped to the DB
- Can work in attached or detached mode
- Entity-Bean will hold DB values in instance variables
- Access to the bean values will be done using get & set methods
- Container is responsible for synchronizing objects with the stored data – using naked objects pool
- Development can go in two directions:
  - Coding Entities and let the provider generate the tables in the DB
  - Code the Entities according to an existing schema

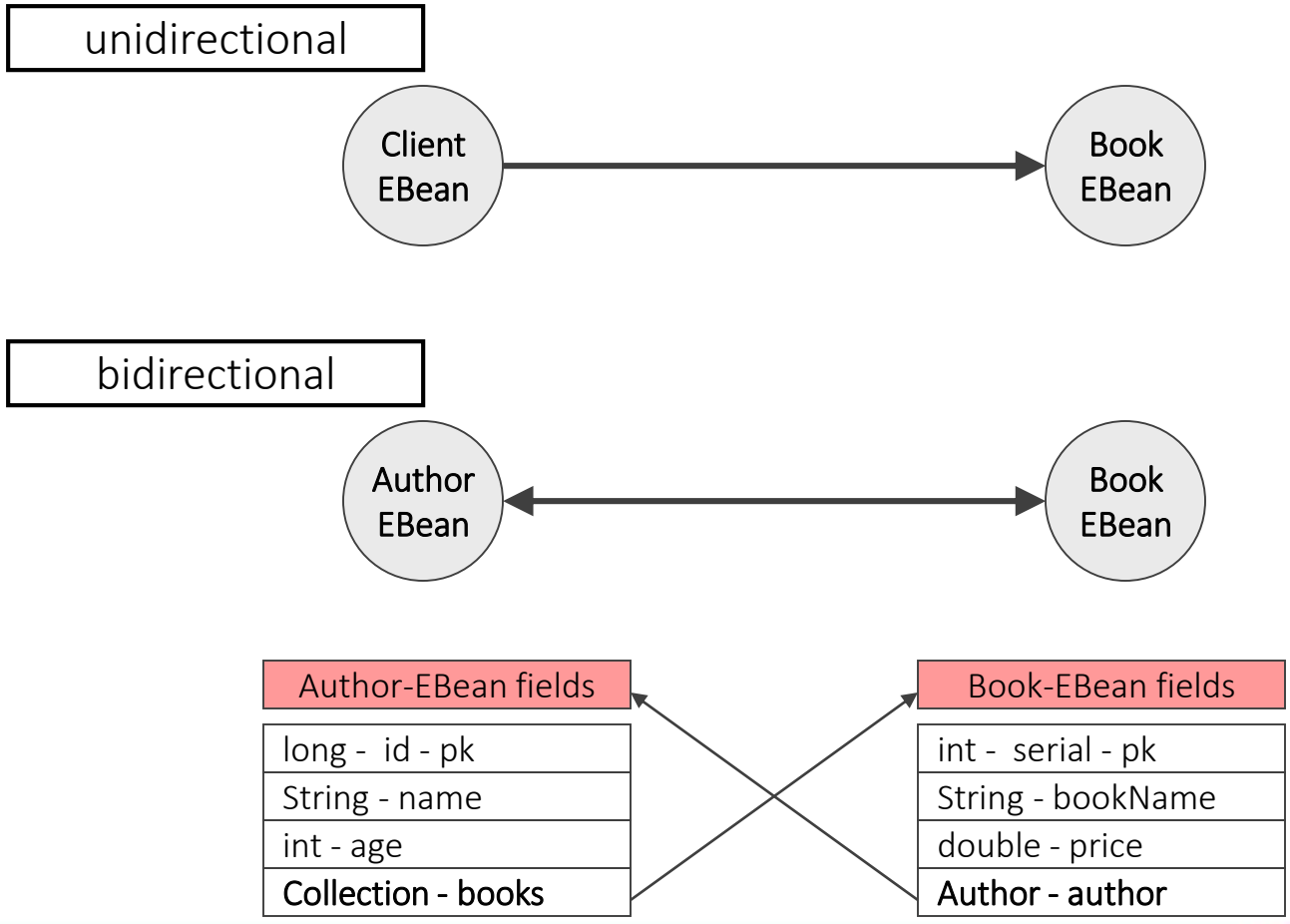
- The JPA approach:
  - Entity beans are POJOs
  - Entity Manager is the single unit responsible for all persistence activity
  - Mapping is done via annotations
  - Therefore –
    - Entity beans are much easier to code – no JDBC code at the bean level
    - Beans may be detached from the persistency context since they are POJOs
    - JPQL can be passed as a string in any query operation done via Entity Manager
    - Vendors are 100% responsible for performance

Example of Object / Relational Mapping

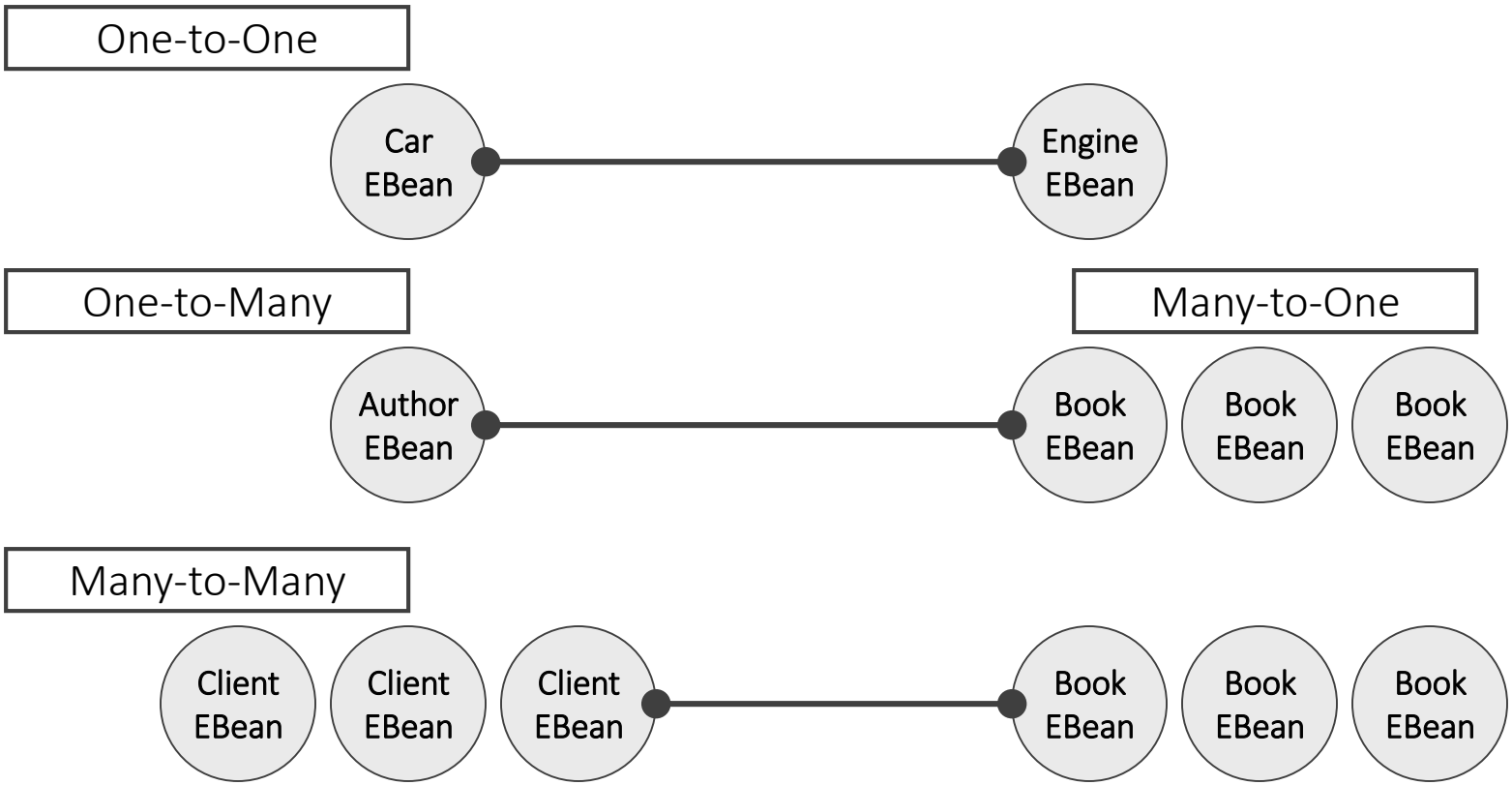
```
…
private int code;
private String bookName;
private double price;
…
public int getCode () {
        return code;
}
public String  getBookName () {
        return bookName;
}
public double getPrice () {
        return price;
}
public void setBookName (String name) {
        bookName=name;
}
public void setPrice (double newPrice) {
        price=newPrice;
}
….
```



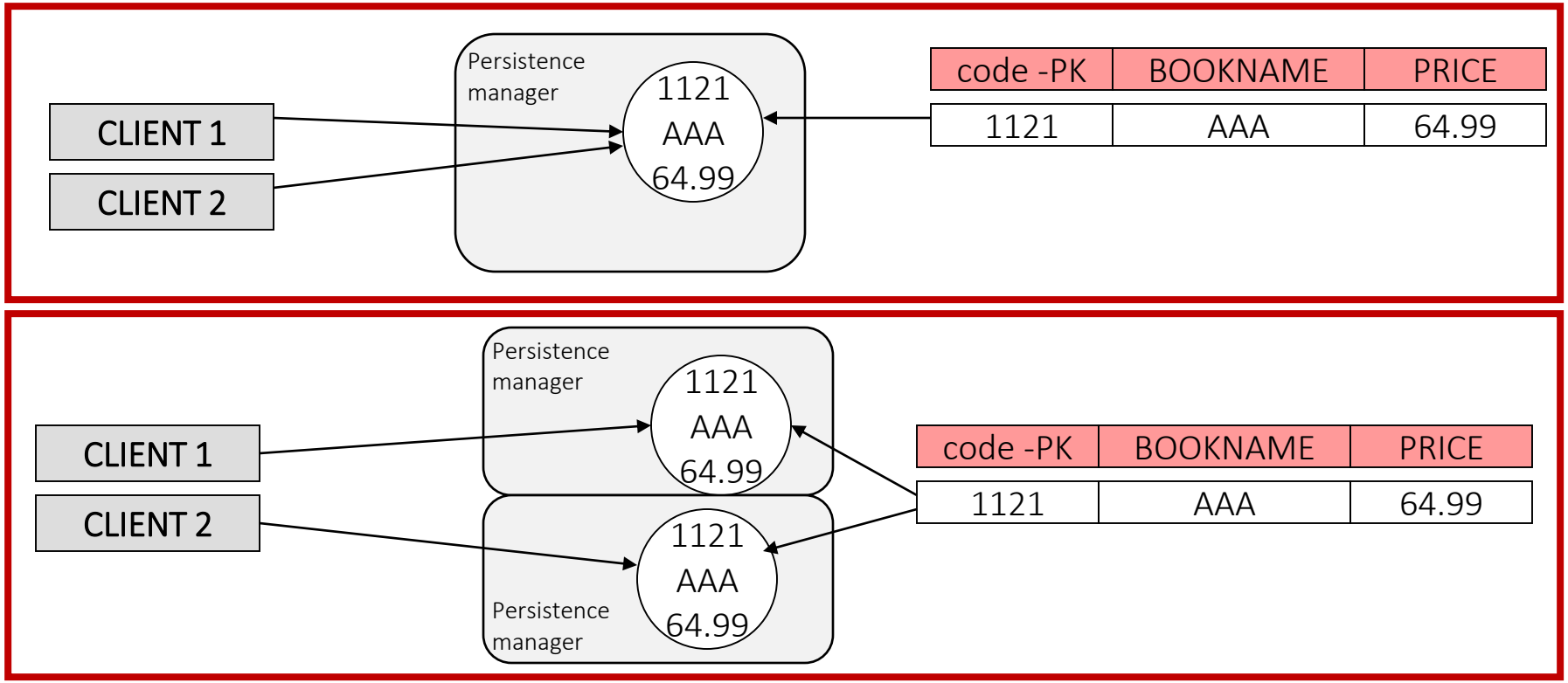
DB relations expressed as references in Objects:



DB relations expressed as references in Objects:



- Same data may be reflected by more then one instance
- Each persistence manager holds only one instance of entity mapped to a row



- Steps in coding Entity bean based flow:
  - Coding Entity beans
  - Creating a persistent unit in the persistence DD
  - Creating clients that uses the unit manager (Entity Manager) to interact with the entity beans
    - Clients might be:
      - Other enterprise beans that share the same container
      - external & stand-alone clients

- Entity bean class must:
  - be annotated with ***@Entity*** annotation ('name' attribute is optional)
  - have no-argument constructor
  - not be Enum or interface
  - have non final Class and data members
- If can be passed as detached object – must implement *Serializable*
- May be abstract
- May be top-level class
- May extend non-Entities classes
- Accessor methods (getters/ setters) mustn't be private & are optional

- Entity beans may use:
  - Property based access
    - means that persistent fields can be accessed directly
  - Method based access
    - means that access is done via getters & setters
      - Both set() & get () must be present
      - Get() is the one that should be annotated
      - Bean field type can be different than types defined in the DB [later]

- Persistent fields – are the Entity attributes mapped to a table columns
  - mustn't be *transient*
  - are declared using DD or annotations
  - if both DD & annotations are used – definition must be consistent
  - type may be:
    - Primitive, primitive array
    - Wrapper class (String, Integer, etc..)
    - java.math.BigInteger, java.math.BigDecimal
    - Java.util.Date, java.sql.Date, java.util.Calendar
    - Java.sql.Timestamp, java.sql.Time
    - Enum
    - Other entities

- Accessor methods
  - Must follow Java Bean naming convention
  - May work with the following collections:
    - Collection<OtherE>
    - Set<OtherE>
    - List<OtherE>
    - **Map<PK><OtherE>**

```
private double value;

public double getValue (){
        return value;
}

public void setValue (double value){
        this.value=value;
}
```

  - If application exception is thrown by accessor methods during container load & store operations will
    - throw *PersistenceException*
    - rollback the existing transaction
  - If any Runtime exception is thrown – the transaction is rolled back

Example:

```
@Entity
@Table(name="COUNTRIES")
public class Country implements Serializable{

    private int id;
    private String name;
    private String language;
    private double area;
    private Collection<City> cities = new HashSet();

    @Id  //denotes PK
@GeneratedValue
@Column(name="CNT_ID", nullable=false, columnDefinition="integer")
public  int getId() { return id; }
public void setId() { this.id=id; }

    @Column(name="NAME", nullable=false)
public String getName() { return name; }
    public void setName(String name) { this.name=name; }

    @Column(name="LANG", nullable=false)
    public String getLanguage() { return language; }
    public void setLanguage(String language) { this. language = language; }

@Column(name="AREA", nullable=false, columnDefinition="double")
    public double getArea() { return area; }
    public void setArea(double area) { this. area = area; }

    ...
```

Example cont.

```
        …

        @OneToMany
        public Collection<City> getCities(){ return cities; }

        public void setCities(Collection<City> cities){ this. cities = cities; }

        //business method to add a single city
        public void addCity(City city) {
                getCities().add(city) ;
                city.setCountry(this);  //since it is a bidirectional relations
        }
}
```

Table

- Entities may be denoted by *@Table* in order to specify the table named to be mapped to.
- *@Table*
  - Appears at class level
  - uses the ‘name’ attribute to specify the table name
  - is optional – if not specified, the not qualified bean name will be used

Primary Key

```
...
private int personID;

@Id
@GeneratedValue(GenerationType.AUTO)
public int getPersonID() { ....  }

public void setPersonID( int personID ) {  ...  }
...
```

- Entity beans must have a PK
- Simple PK
  - Is one of the persistent fields
  - Annotated with *@Id*
  - Can be automatically generated when annotated *@GeneratedValue*
    - *strategy* attribute may specify any of the supported generation types
    - *GenerationType* enum supports: *TABLE, SEQUENCE, IDENTITY, AUTO*
- Composite PK
  - When PK is more than one column a PK class should be composed
  - Must be public *Serializable* object
  - Must have no-argument constructor
  - May be properties-based (use public accessor methods)
  - Must override *equals() & hashcode()* to become logically comparable
  - Fields and properties must have the same name & type as in the Entity bean class
  - Cannot be automatically generated
  - Use *@EmbeddedId* & *@Embeddable - for PK classes that are part of the beans state*

```
@Embeddable
public class CityPK implements Serializable{
        private String name;
        private int code;

        public CityPK() {}

        public CityPK (String name, int code){
              this.name=name;
              this.code=code;
        }
        public int hashCode () {
                 return name.hashCode()+code;
        }
        public boolean equals (Object o) {
                if (o instanceof CityPK)
                  if (((CityPK)o).name.equals(this.name) && ((CityPK)o).code==this.code)
                     return true;
                return false;
        }
        @Column(name="CITY_NAME")
       public  String getName(){ return name; }
       public void setName(String name){ this.name=name;}
      @Column(name="CITY_CODE")
       public  int getCode(){ return code; }
       public void setCode(int code){ this.code=code;}
}
```

PK class example

In this case – the PK is not part of the state of the Entity bean – it is used internally.
requirements:

- class is annotated as embeddable
- implements Serializable
- overrides hashCode() & equals()
- data members are actual CMP fields
- denoted getters & setters

```
@Entity
@Table(name="CITIES")
public class City implements Serializable

    private CityPK pk;
    private Country country;
    …

    @EmbeddedId()
     public CityPK getPK() { return pk; }
     public void setPK(CityPK pk) { this.pk=pk; }

    …

}
```

PK class example

The entity bean uses CityPK class as PK externally.
This means that it will use CityPK objects for maintaining both code & name CMP fields.
requirements:

- declare the composite PK class
- denote each CMP field that is part of the PK

- Entity CMP Fields
  - Annotations for CMP fields:
    - @Column
      - name – name of the mapped table column (String)
      - nullable – specifies whether the column may take null values or not (true/false)
      - unique – specifies whether the column holds unique values or not (true/false)
      - columnDefinition - specifies the DDL type of the column.

        When not specified – mapping is done according to system defaults (for example – java.lang.String is mapped to a VARCHAR)
      - table – used for multi-table mapping
      - insertable – specified whether this column included in INSERT operation (true/false)
      - updatable - specified whether this column included in UPDATE operation (true/false)
      - length – specified the permitted VARCHAR length
    - @Column is used at the set/get method level

```
@Column(name="EMP_ID", nullable=false, columnDefinition="integer")
```

- Entity CMP Fields
  - More annotation for CMP fields:
    - @Transient

```
@Transient
```

      - means that the CMP field is not persisted
      - relevant for 'read only' attribute that only have 'get' methods
      - is used at the get() method level
    - @Basic

```
@Basic(fetch=FetchType.LAZY)
```

      - specifies the default mapping between Java types & DDL type
      - therefore – usually never used but:
      - help in determining fetch policy via *fetch* attribute
      - *FetchType* enum supports: *LAZY, EAGER*
        - LAZY – data is loaded only when the denoted column is read
        - EAGER – data is loaded on first fetch from the DB

- Entity CMP Fields
  - More annotation for CMP fields:

```
@Temporal(TemporalType.TIME)
```

    - @Temporal
      - Helps in determining which DB type is mapped to java.util.Date or java.util.Calendar
      - Default mapping is to 'timestamp'
      - Attribute value is *TemporalType* enum
      - *TemporalType* enum supports: *DATE,TIME,TIMESTAMP*
      - Can be used in addition to *@Basic* annotation & fetch policy

```
@Lob
```

    - @Lob

      Useful when the persisted types are:
      - Blob (byte[], Byte[], Serializable objects)
      - Clob (char[], Character[], long Strings)
      - Can be used in addition to *@Basic* annotation & fetch policy

- Entity CMP Fields
  - More annotation for CMP fields:
    - @Enumerated
      - Useful for mapping enums to a table (get/set takes and returns enum)
      - Enums are stored by default as number (ordinal value) or as string
      - The value of @Enumerated is specified via *EnumType* enum
      - *EnumType* enum supports: *ORDINAL,STRING*

```
@Enumerated(EnumType.STRING)
```

- Persistence DD – *persistence.xml*
  - Specifies the persistence units
    - Which entity beans are included (if not – all entities are mapped automatically)
    - Which DataSource is used per unit
    - Persistence unit name – used by Entity bean clients
    - Provider & Properties settings

<persistence-unit>
- defines a single unit
- may be repeated for multiple units
- the name of the unit is use by the unit clients that obtains its *EntityManager*

<jta-data-source>
- specifies the DataSource JNDI name

<provider>
- specifies the fully qualified provider class name
- is optional – when using the default provider

<properties><property>
- sets vendor specific attributes

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence>
  <persistence-unit name="globe">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

- Persistence DD – *persistence.xml*
  - In most cases several units will be defined
  - Each entity should be mapped to a unit
  - Entities may be assigned as jars or as single classes

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence>
  <persistence-unit name="globe">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <jar-file>lib/globe.jar</jar-file>
    <properties>
        ....
    </properties>
  </persistence-unit>
</persistence>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence>
  <persistence-unit name="globe">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <class>org.globe.Country</class>
    <class>org.globe.Capital</class>
    <properties>
        ....
    </properties>
  </persistence-unit>
</persistence>
```

- *EntityManager*
  - A unit that manages Entity beans of the same persistent context
    - Persistent context
      - Is set of entity instances
      - Each persistence entity has a unique instance
      - Works with a single *DataSource* specified in the *persistence.xml*
  - *EntityManager* interface defines the way of interacting with the persistent context
    - Encapsulates all JDBC activity
    - Makes interaction simple
    - Allows entity beans to remain POJOs
    - Supports attached and detached entities

      for example:
      - when working with an attached object changes done via 'set' methods are delegated to the DB
      - when working with a detached object a 'merge' operation is required in order for all 'set' changes to be delegated – usually relevant for remote client like Swing applications

- *EntityManager*

  Supported operations:

EntityBean interface :

persist (Object entity)	Inserts a new table row where the entity is mapped to
find(Class<T> entity class, Object pk)	Returns entity <T> or null if not found
getReference(Class<T> entity class, Object pk)	Returns entity <T> or *EntityNotFoundException*
createQuery(String ejbQL)	returns a query object that allows to find entity beans
createNamedQuery(String queryName)	same, but uses a pre-defined query specified in the DD
createNativeQuery(String SQL)	same, but uses a vendor specific query or SQL query
remove (Object entity)	Deletes a table row where the entity is mapped to
merge (Object entity)	Attaches the entity to the persistent context - update
refresh (Object entity)	Overrides the managed entity state with DB data
contains (Object entity)	Returns 'true' if the entity is currently managed
flush ()	Forces DB updates done via persist, merge, remove
clear ()	Detaches all managed entities, unsaved changes are lost

- Persist operation combined with @GeneratedValue raises an issue:
  - How will the client get the generated value ?
  - Answer – the entity sent to persist – gets updated
    - The container sets its PK before inserting
    - Simply return the entity instance or its PK

```
public int addNewCountry(Country country ){
        entityManager.persist(country);
        return country.getId();
}
```

- *EntityManager - Exceptions*
  - Persist
    - *EntityExistsException* – If entry already exists
    - *IllegalArgumentException* – If sent parameter is not an Entity
  - Merge, Remove, Find
    - *IllegalArgumentException* – If sent parameter is not an Entity
  - GetReference, Refresh
    - *IllegalArgumentException* – If sent parameter is not an Entity (class)
    - *EntityNotFoundException* – if the Entity cannot be accessed
  - Query execution
    - *IllegalArgumentException* – If JPQL raises problems of invalid
  - Native Query execution
    - *PersistenceException* - wraps any underlying exception
- Are all system exceptions

- *EntityManager*
  - Obtaining *EntityManager* via Dependency Injection
    - Both *EntityManagerFactory* & *EntityManager* can be injected
    - For *EntityManagerFactory* injection use *@PersistenceUnit* annotation
    - For *EntityManager* injection use *@PersistenceContext* annotation
    - Usually there is no need in working with the factory
    - More than one persistence unit may be used
    - The factory is closed by the container when the bean is discarded

- *EntityManager*
  - Obtaining *EntityManager & EntityManagerFactory*
    - J2EE – clients - example

```
@Stateless
public MyBean implements MyBusinessInterface{

    @PersistenceUnit(unitName="globe")
    private EntityManagerFactory factory;

    …
```

```
@Stateless
public MyBean implements MyBusinessInterface{

    @PersistenceContext(unitName="globe")
    private EntityManager manager;

    …
```

- Entity Beans Relationships
  - Annotations for relationship:
    - @OneToOne - Unidirectional, Bidirectional
    - @OneToMany – Unidirectional, Bidirectional
    - @ManyToOne - Unidirectional
    - @ManyToMany - Unidirectional, Bidirectional

    - *Each annotation supports:*
      - *fetch policy settings - fetch=FetchType.EAGER/LAZY*
      - *cascade*
      - *mapped by*
  - *Join table*
    - When using @ManyToMany a 3rd table must be created
    - If JPA is generating DB schema – a join table and columns can be specified
    - Use @JoinTable or @JoinColumn for that
    - Is generated automatically

- Entity Beans Relationships
  - *cascade* value
    - Specifies how the persistent manager treats dependent entities in each action
    - Supported actions: *'PERSIST', 'MERGE', 'REMOVE', 'REFRESH', 'ALL'*
    - Permitted only for OneToOne, OneToMany , ManyToOne

```
@Entity
public class Country implements Serializable

        private Capital capital;

        @OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
        public Capital getCapital(){
                return capital;
        }
        public void setCapital(Capital capital){
                this. capital = capital;
        }
}
```

Means that every time a Country entity is persisted or removed, the same happens to its dependent entity called Capital

```
@Entity
public class Country implements Serializable

        private Capital capital;

        @OneToOne(cascade={CascadeType.ALL})
        public Capital getCapital(){
                return capital;
        }
        public void setCapital(Capital capital){
                this. capital = capital;
        }
}
```

- Entity Beans Relationships
  - *cascade* value

```
Country c=new Country();
Capital cap=new Capital ();
cap.setCountry(c);
c.setCapital(cap);

entityManager.persist(c);
```

Since cascade was set to ALL,
Any operation, including 'persist'
includes all dependent entities.
In this case, adding a new Country entry
will also create a new Capital entry

- When NOT to use cascading ?
  - When entities have different lifespan
    - a deletion of one should not be followed by a deletion of other
    - For example – when City is removed – Country should stay
  - When merging entity with little changes / updates
    - If most of the CMP & CMR fields haven't change – there is no reason to cascade the 'merge' operation
    - Might effect performance - save round trips to the DB
  - You should know how your entities are going to be used before you set cascading

- Entity Beans Relationships

  - *mappedBy* value
    - Used only in bidirectional relationship
    - Specifies the field or property in the dependent entity – not the owner
    - Forbidden for ManyToOne (the 'ManyToOne' is in the owner)
    - In OneToOne – the owner is the one that hold the foreign key

    - The '*ejbPostCreate()*' issue is solved easily

- Entity Beans Relationships
  - OneToOne unidirectional example
  - In this example – Country holds a reference to a single Capital entity

```
Country c=new Country();

Capital cap=new Capital();

c.setCapital(cap);
```

```
@Entity
public class Country implements Serializable

        private Capital capital;

        @OneToOne
        public Capital getCapital(){
                return capital;
        }
        public void setCapital(Capital capital){
                this. capital = capital;
        }
}
```

```
@Entity
public class Capital implements Serializable
        …
}
```

- Entity Beans Relationships
  - OneToOne bidirectional example

```
Country c=new Country();

Capital cap=new Capital ();

cap.setCountry(c);

c.setCapital(cap);
```

- Country is created with a 'null' Capital value
- Capital is created & set with a country
- Capital is assigned to the country

```
@Entity
public class Country implements Serializable

        private Capital capital;

        @OneToOne
        public Capital getCapital(){
                return capital;
        }
        public void setCapital(Capital capital){
                this. capital = capital;
        }
}
```

```
@Entity
public class Capital implements Serializable

        private Country country;

        @OneToOne(mappedBy="capital")
        public Country getCountry(){
                return country;
        }
        public void setCountry(Country country){
                this.country =country;
        }
}
```

- Entity Beans Relationships
  - OneToMany unidirectional example
  - In this example – single Country entity can hold references to different City entities

```
@Entity
public class Country implements Serializable

        private Collection<City> cities =new ArrayList<City>();

        @OneToMany
        public Collection<City> getCities(){
                return cities;
        }
        public void setCities(Collection<City> cities){
                this. cities = cities;
        }
}
```

```
@Entity
public class City implements Serializable
        …
}
```

Some issues when working with 'Many' relationship

- The Collection CMR must be instantiated in the declaration line – Otherwise, when clients create POJOs offline, the collection will be null.
- Most server uses 'LAZY' loading for Collection CMR fields. If users detaches entity beans – policy should change to 'EAGER' – otherwise, without calling the appropriate get() method and force collection loading - the instances are not fully loaded when handed to remote clients!

- Entity Beans Relationships
  - ManyToOne unidirectional example
  - In this example – many Cities can point to the same Country

```
@Entity
public class City implements Serializable

        private Country c;

        @ManyToOne
        public Country getCountry(){
                return c;
        }
        public void setCountry( Country c){
                this. c= c;
        }
}
```

```
@Entity
public class Country implements Serializable
        …
}
```

- Entity Beans Relationships
  - OneToMany / ManyToOne bidirectional example
  - The owner is City
    - Uses 'ManyToOne'

```
@Entity
public class Country implements Serializable

        private Collection<City> cities = new HashSet<City>();

        @OneToMany(mappedBy="country")
        public Collection<City> getCities(){
                return cities;
        }
        public void setCities(Collection<City> cities){
                this. cities = cities;
        }
}
```

```
@Entity
public class City implements Serializable

        private Country country;

        @ManyToOne
        public Country getCountry(){
                return country;
        }
        public void setCountry(Country country){
                this.country =country;
        }
}
```

- Entity Beans Relationships
  - ManyToMany unidirectional example
  - In this example – several Country entities can hold same references to different City entities

```
@Entity
public class Country implements Serializable
        private Collection<City> cities = new HashSet<City>();

        @ManyToMany
        public Collection<City> getCities(){
                return cities;
        }
        public void setCities(Collection<City> cities){
                this. cities = cities;
        }
}
```

```
@Entity
public class City implements Serializable
        …
}
```

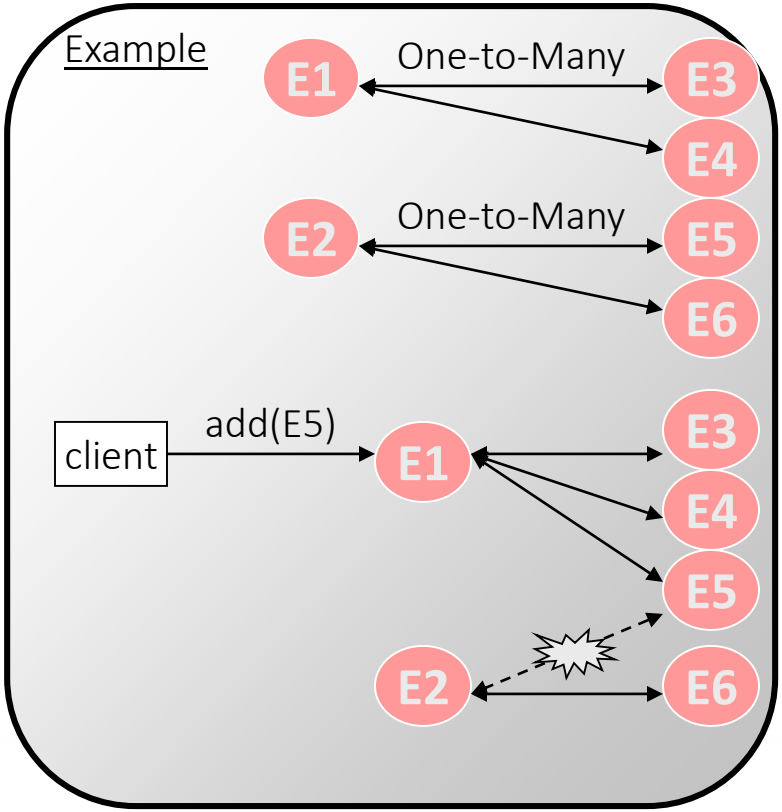
- Entity Beans Relationships
  - ManyToMany bidirectional example
  - The owner may be anyone
  - In this example Country entities can be referenced by different City entities & vise versa

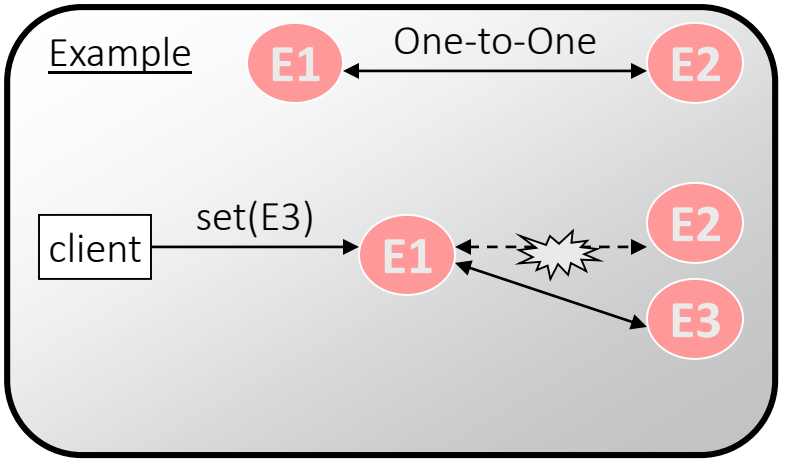
```
@Entity
public class Country implements Serializable

        private Collection<City> cities = new HashSet<City>();

        @ManyToMany
        public Collection<City> getCities(){
                return cities;
        }
        public void setCities(Collection<City> cities){
                this. cities = cities;
        }
}
```

```
@Entity
public class City implements Serializable

        private Collection<Country> countries = new HashSet<Country>()

        @ManyToMany(mappedBy="cities")
        public Collection<Country> getCountries(){
                return countries;
        }
        public void setCountries(Collection<Country> countries){
                this. countries = countries;
        }
}
```

Relations Integrity

- The programmer is responsible for updating all the involved components according to the relationship setting in every change






```
Customer newCust= new Customer();
CreditCard cc = oldCust.getCreditCard();

oldCust.setCreditCard(null);
newCust.setCreditCard(cc);
```

- Working with Maps
  - Usually for holding a key to an entity bean
  - Key is one of the unique fields of the entity class (pk)
    - Use @MapKey to specify field name

```
@Entity
public class Country implements Serializable

        private Map<String,City> cities = new HashTree<String,City>();

        @ManyToMany
        @MapKey(name="cityName")
        public Map<String,City> getCities(){
                return cities;
        }
        public void setCities(Map<String,City> cities){
                this. cities = cities;
        }
}
```

- Querying
  - JPQL
    - query language for objects
    - is used upon abstract schema names representing actual entities
    - queries may result in Objects and CMP field (including primitives)
  - In previous versions
    - JPQL statements were part of the DD
    - Therefore, kind of hard coded
  - Now
    - Are assigned to the Entity Manager via

createQuery(String ejbQL)	returns a query object that allows to find entity beans

- Querying
  - findByPrimaryKey (key)
    - No need to specify it in each Entity bean anymore
    - Is provided by the *EntityManager*

find(Class<T> entity class, Object pk)	Returns entity <T> or null if not found
getReference(Class<T> entity class, Object pk)	Returns entity <T> or *EntityNotFoundException*

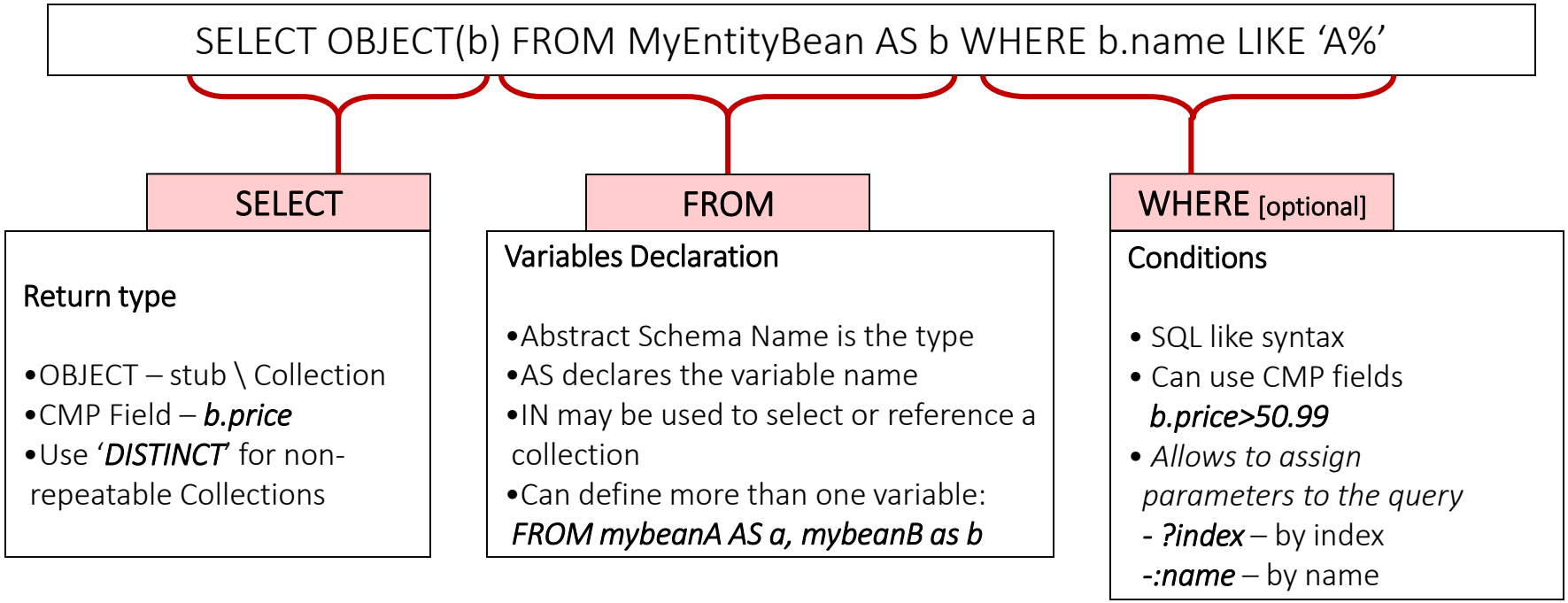
- Query API
  - Query interface
    - Allows to define queries programmatically
    - Supports
      - Assigning parameters
      - Set hints [vendor specifics like *timeout* in hibernate]
      - Set max results
      - Set single result
      - Set first result index

      } Result paging

Query interface :

List getResultList()
Object getSingleResult()
Query setMaxResults(int max)
Query setFirstResult(int index)
setHint(String hintName, Object value)
Query setParameter(String name, Object value)

- JPQL statements has 3 parts:



- Querying
  - Simple query examples

```
Query query= entityManager.createQuery("SELECT OBJECT(c)  FROM Customer AS c");
List<Customer> list = query.getResultList();
for(Customer cust : list){
        ....
}
```

- Returns a List of Objects
- Each object is an instance of Customer

```
Query query= entityManager.createQuery("SELECT c.firstName, c.lastName FROM Customer AS c");
List list = query.getResultList();
for(Object [] data : list){
        Object firstName=data[0];
        Object lastName=data[1];
        ....
}
```

- Returns a List of arrays (Object[ ])
- Each array holds the firstName & lastName of record a customer

- Querying
  - JOIN operator
    - Useful when querying a collection CMR field

SELECT city from Country AS c **join c.cities as city**

Returns a List of all cities from all countries

SELECT city.name from Country AS c **join c.cities as city**

Returns a List of all city names from all countries

SELECT city from Country AS c **join c.cities as city ORDER BY city.name DESC**

Returns a sorted List of all cities from all countries. The list is ordered by city names in a descending order

- Querying
  - WHERE clause
    - Syntax is similar to SQL

Constants

string	‘Hello’ ‘Hello World’ - use ‘’ as quote within a string
integer	145 , -981
float	0.7 , -13.678 , 5.1E4
boolean	TRUE , FALSE

Operators

Arithmetic	+ , - , * , / , BETWEEN xxx AND xxx , ABS(num) , SQRT(num)
Casting	From int to float is done automatically
Strings	LIKE , __ (underscore) replaces one character , % replace many characters
Conditions	=,<,>,<=,>=,<>,[NOT]LIKE, [NOT]BETWEEN..AND.., IS, IS[NOT], AND, OR, NOT , IN[NOT]

- Querying
  - WHERE clause examples

SELECT c from Country AS c **WHERE c.age > 50**

Returns a List of all countries that exist more than 50 years

SELECT city from Country AS c, join c.cities as city **WHERE city.name LIKE 'A%'**

Returns a List of all cities that starts with the letter 'A'

SELECT c FROM Country AS c , join c.cities as city **WHERE city.name IN (?1, ?2, 'A%', 'B%')**

Returns a List of all countries with the assigned city names (as query parameters) & with cities that starts with 'A' and 'B' letters
Setting parameters:
Query.setParameter(int paramPosition, Object value)

SELECT c FROM Country AS c **WHERE c.name = :name**

Returns a Country with the specified name (as query parameter)
Setting parameters:
Query.setParameter(String paramName, Object value)

- Querying
  - WHERE clause
    - Functional expressions

String Functions

LOWER(string s)
UPPER(string s)
TRIM(string s)
CONCAT (string s1,string s2)
SUBSTRING (string s , int start, int end)
LOCATE (string origin, string toFind, int start)
LENGTH (string)

Numeric Functions

ABS (number) – absolute value
SQRT(double) – returns the square root of a double
MOD(int x, int y) – returns x%y

Dates & Times

CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP

- Querying
  - WHERE clause with functional expressions examples

SELECT c from Country AS c WHERE **LENGTH(c.name) >6**

Returns a List of all countries with a name that is longer than 6 chars

SELECT c from Customer AS c WHERE **ABS(c.rate) > 4**

Returns a List of all customers with a rate higher than 4

SELECT r FROM Reservation AS r WHERE **r.date = CURRENT_DATE**

Returns a List of all reservations made today

- Querying
  - Named Queries

A mechanism that allows to predefine queries (JPQL & native) for later use

EntityManager generates named Queries via

`createNamedQuery(String queryName)`

```
@NamedQueries({
    @NamedQuery(name="cityList", query="SELECT city.name FROM Country  AS c join c.cities as city"),
    @NamedQuery(name="avgPopulation", query=SELECT AVG(city.population) FROM Country AS c join c.cities as  city"),
    @NamedQuery(name="population", query=SELECT SUM(city.population) FROM Country AS c join c.cities as city")
})
@Entity
public class Country implements Serializable
        private Collection<City> cities = new HashSet();          …
        …
}
```

```
Query query = entityManager.createNamedQuery("avgPopulation");
int avgPopulation = query.getSingleResult();
```

Done outside of the entity bean

Detached Entities

- Entity beans can be located by client via:
  - find(..)
  - getReference(..)
  - Query
- As long as clients are not downloading [serializing] entity beans – there are no detached entities
  - Local clients and other EJBs may live in the same persistent context
  - Means that set() operations are synchronized with the DB directly
- But – when a remote client is involved that the entity replica is actually detached from the persistent context

Detached Entities

- Detached entities raises several issues:
  - Set operations changes the object state – but are not persisted
  - Lazy fetch policy causes CMP fields to remain un-initialized
    - In this case the spec is not clear
    - Getting a lazy fetched data might result in some vendor specific lazy-exception
    - This exception can be inserted to the Entity POJO by byte-code manipulation

- Detached nodes becomes attached after the client performs merge() operation

Cache & Locking Strategies

EntityManager

- holds a unique optimized set of cached entities
- this cache is known as “First Level Cache”
- manages locks on first level cache in two ways:
  - Optimistic Locking – allows multiple Tx on an entity – but prevents collisions
  - Pessimistic Locking – prevents other Tx on an entity when reading a row

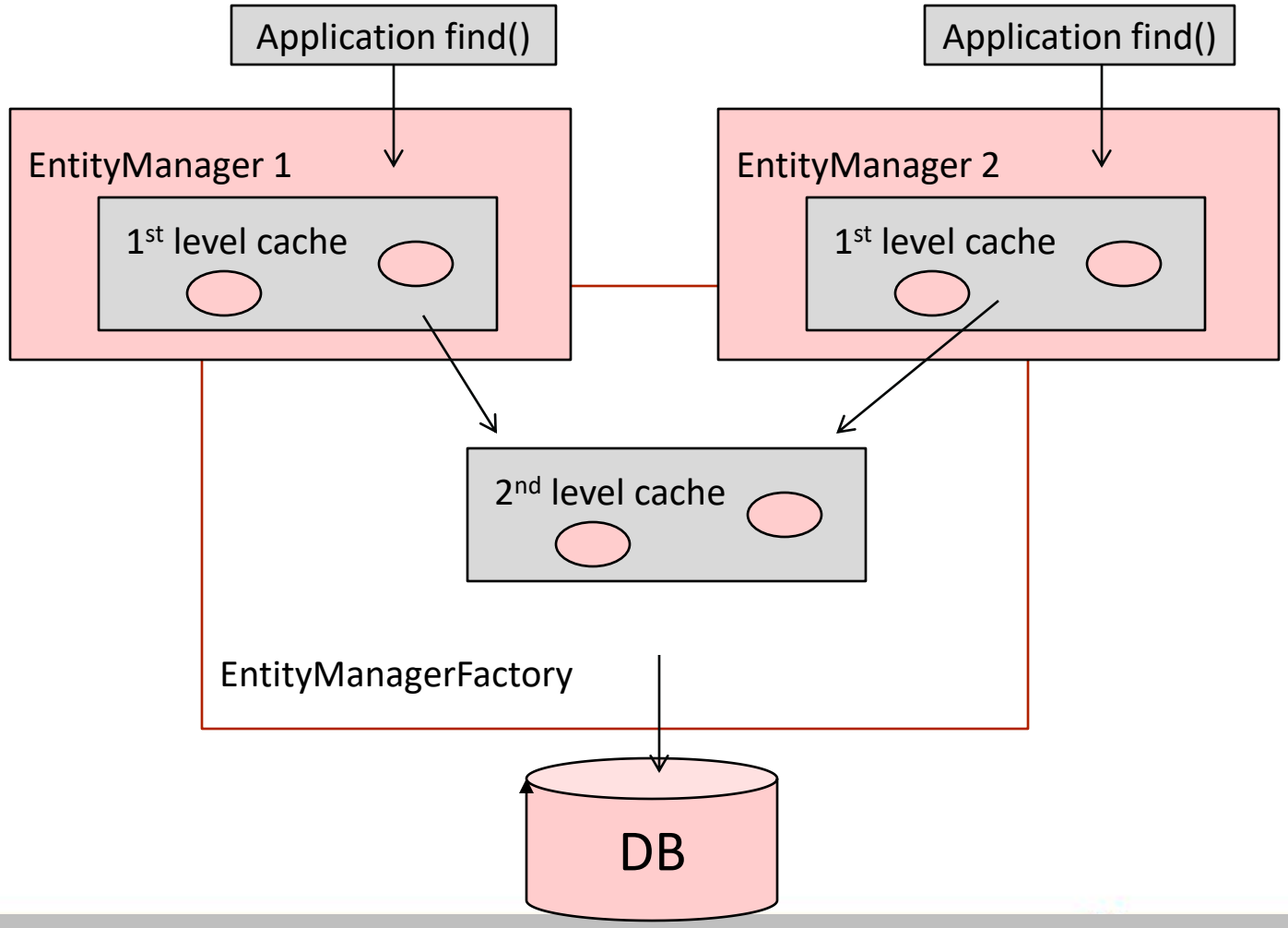
EntityManagerFactory – can generate multiple EntityManagers

- holds a unique optimized set of cached entities collected from several entity managers
- known as ‘Shared Cache’ or ‘Second Level Cache’ (2L Cache)

Entity Beans

JOHN BRYCE
Leading in IT Education
a matrix company

Cache & Locking Strategies



Entity Beans

JOHN BRYCE
Leading in IT Education
a matrix company

1st Level Cache - Optimistic Locking

- Means that you will be able to store data only if not changed during current Tx
- Helps in keeping high integrity when data is accessed by multiple Tx
- Not relevant when working in 'read only' mode
- When updating, optimistic lock is done by versioning
  - Each entity that is loaded gets a version id
  - The id value in the entity bean matches the current value in the DB
  - Each time a commit is performed on that record – the version value is incremented
  - Any attempt to store entity object with older version throws *OptimisticLockException*

Entity Beans

JOHN BRYCE
Leading in IT Education
a matrix company

1st Level Cache - Optimistic Locking

- To use optimistic locking
  - Add a column to the relevant tables to hold the version values
  - Update Entity class
    - Define and map CMP field to hold version value
    - Define only get() method – version values must be 'read only' fields
    - Denote CMP field with *Version* annotation

```
private int version;

@Version
@Column(name = "VERSION")
public int getVersion() {
        return version;
}
```

Entity Beans

JOHN BRYCE
Leading in IT Education
a matrix company

1st Level Cache - Optimistic Locking

- Entity manager supports lock operation for versioned entities
- It is done via *EntityManager.lock(entity, lockType)* operation
- Lock operation takes one of the following Enums:
  - LockType.READ
  - LockType.WRITE
- Read
  - No dirty reads
  - No no-repeatable reads
- Write
  - Same as READ
  - Version update on flush, commit

Entity Beans

JOHN BRYCE
Leading in IT Education
a matrix company

1st Level Cache - Pessimistic Locking

- Means that no one can perform any transaction on an entity already participating in one
- Harms performance – but good for high-risk data access
- Not relevant when working in 'read only' mode
- Two ways of locking:
  1. Read than Lock - Find the entity and then lock before updating

     Less locking time, but might end up with OptimisticLockException if was changed between fetching and locking
  2. Read and Lock - Lock starts during 'finding' – so when fetched, the entity is already locked for you

     Longer lock time – might cause bottlenecks…

Entity Beans

JOHN BRYCE
Leading in IT Education
a matrix company

1st Level Cache - Pessimistic Locking

- To use Pessimistic locking in the 2 different ways:
  1. Read than Lock

```
Employee e = em.find(Employee.class, id)
em.lock(e, PESSIMISTIC);
e.setSalary(10000);
```

  2. Read and Lock

```
Employee e = em.find(Employee.class, id, PESSIMISTIC)
float salary = e.getSalary();
e.setSalary(salary+100);
```

Entity Beans

JOHN BRYCE
Leading in IT Education
a matrix company

2L Cache

- Shared cache (or 2L cache) is maintained by EntityMasnagerFactory
- To set 2L Cache edit persistence.xml :

```
<persistence-unit name="myUnit">
   ...
   <properties>
      <property name="javax.persistence.sharedCache.mode" value="ALL"/>
   </properties>
   ...
 </persistence-unit>
```

- Property *javax.persistence.sharedCache.mode* accepts:
  - ALL – default, cache is enabled for all entities
  - NONE – no 2L cache
  - ENABLE_SELECTIVE – active only for @Cacheable entities
  - DISABLE_SELECTIVE – active for all entities except @Cacheable entities

Entity Beans

JOHN BRYCE
Leading in IT Education
a matrix company

2L Cache

- For value ENABLE_SELECTIVE, cache is active only for cacheable entities:

```
@Cacheable
@Entity
public class Employee {
  …
```

or

```
@Cacheable(true)
@Entity
public class Employee {
  …
```

- For value DISABLE_SELECTIVE, cache is active for all except entities where cache is disabled:

```
@Cacheable(false)
@Entity
public class Employee {
  …
```

Entity Beans

JOHN BRYCE
Leading in IT Education
a matrix company

2L Cache

- Shared cache is updated automatically by default when
  - On retrieval – when entity is not found in 1st & 2nd levels cache – it is loaded and added
  - On commit (store) – when entity is updated or created

- Each mode can be configured to enable/disable cache
  - For retrieve mode set EntityManager *javax.persistence.cache.retrieveMode* & *javax.persistence.cache.storeMode* properties
    - USE – to enable
    - BYPASS – to disable

```
em.setPropertry("javax.persistence.cache.retrieveMode ", CacheRetrieveMode.BYPASS);
```

Entity Beans

2L Cache

- Override settings on querying:

```
query.setHint("javax.persistence.cache.retrieveMode ", CacheRetrieveMode.BYPASS);
```

- We can override setting on find operation as well:

```
em.find(Employee.class, id, Collections.singletonMap
        ("javax.persistence.cache.retrieveMode ", CacheRetrieveMode.BYPASS));
```

Entity Beans

JOHN BRYCE
Leading in IT Education
a matrix company

2L Cache

- Using Cache interface
  - Obtained from EntityManagerFactory
  - Allows to check and remove from cache

```
Cache cache = emf.getCache();
```

Cache Interface:

boolean contains(Class c, Object pk)	Return true if entity found in cache
void evict(Class c)	Removes entities of specified class from cache (subclasses are included)
void evict(Class c, Object pk)	Removes the give n entity from cache
void evictAll()	Clear the cache

  - Cache is used mainly by the provider & its hosting environment