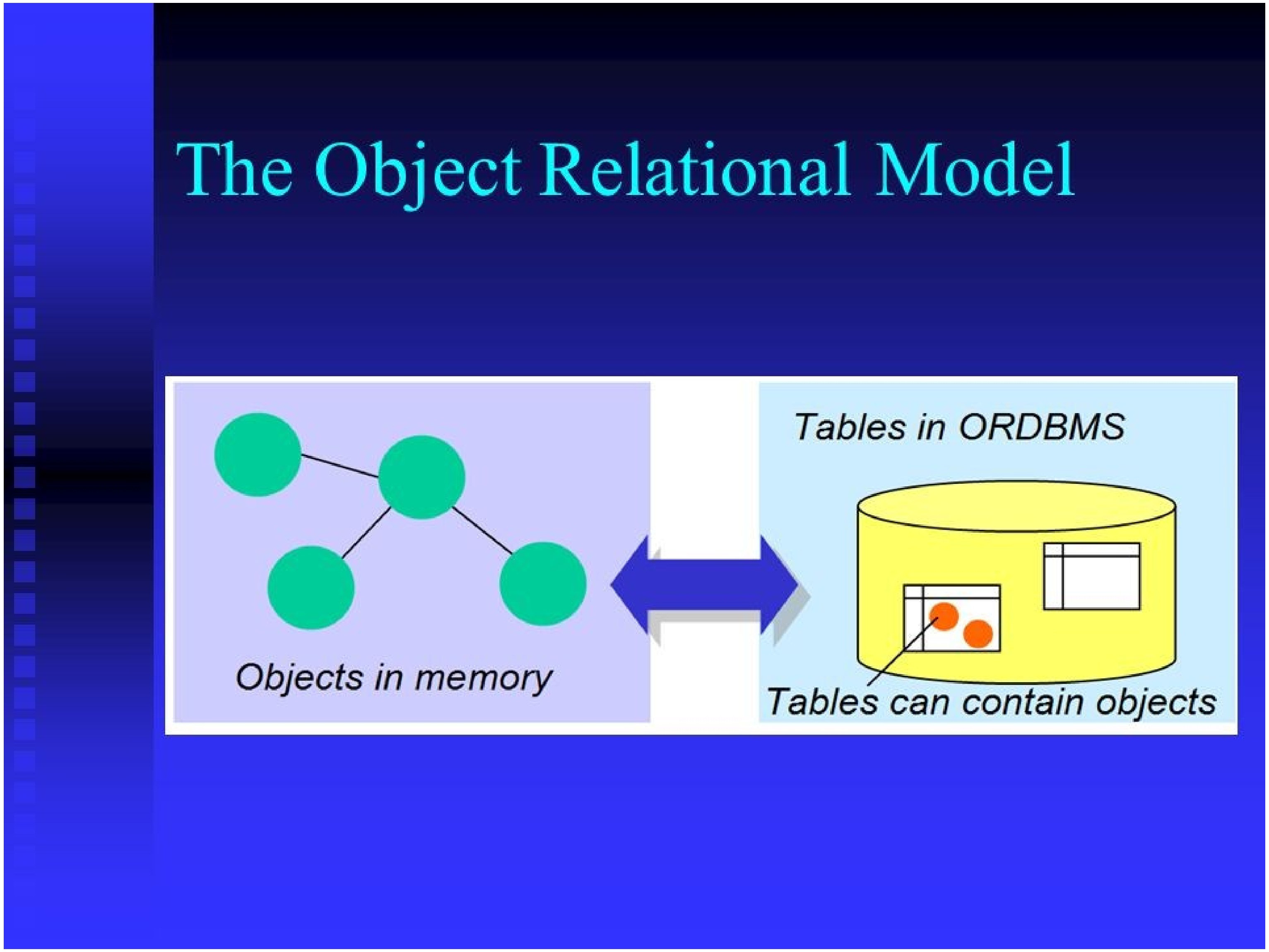
THE ENTERPRISE INFORMATION LAYER

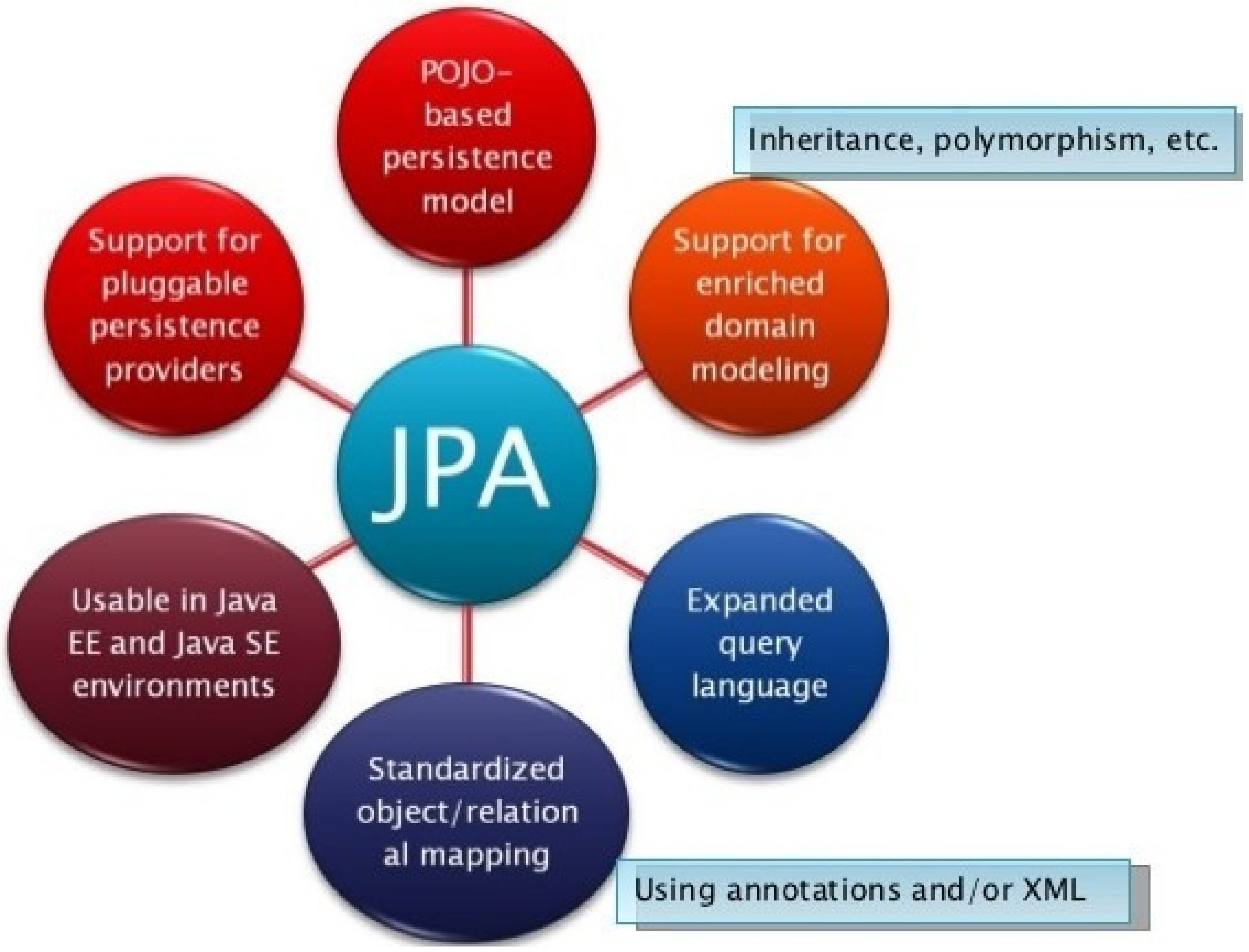
Java Persistence API - JPA

OBJECT RELATIONAL MODEL

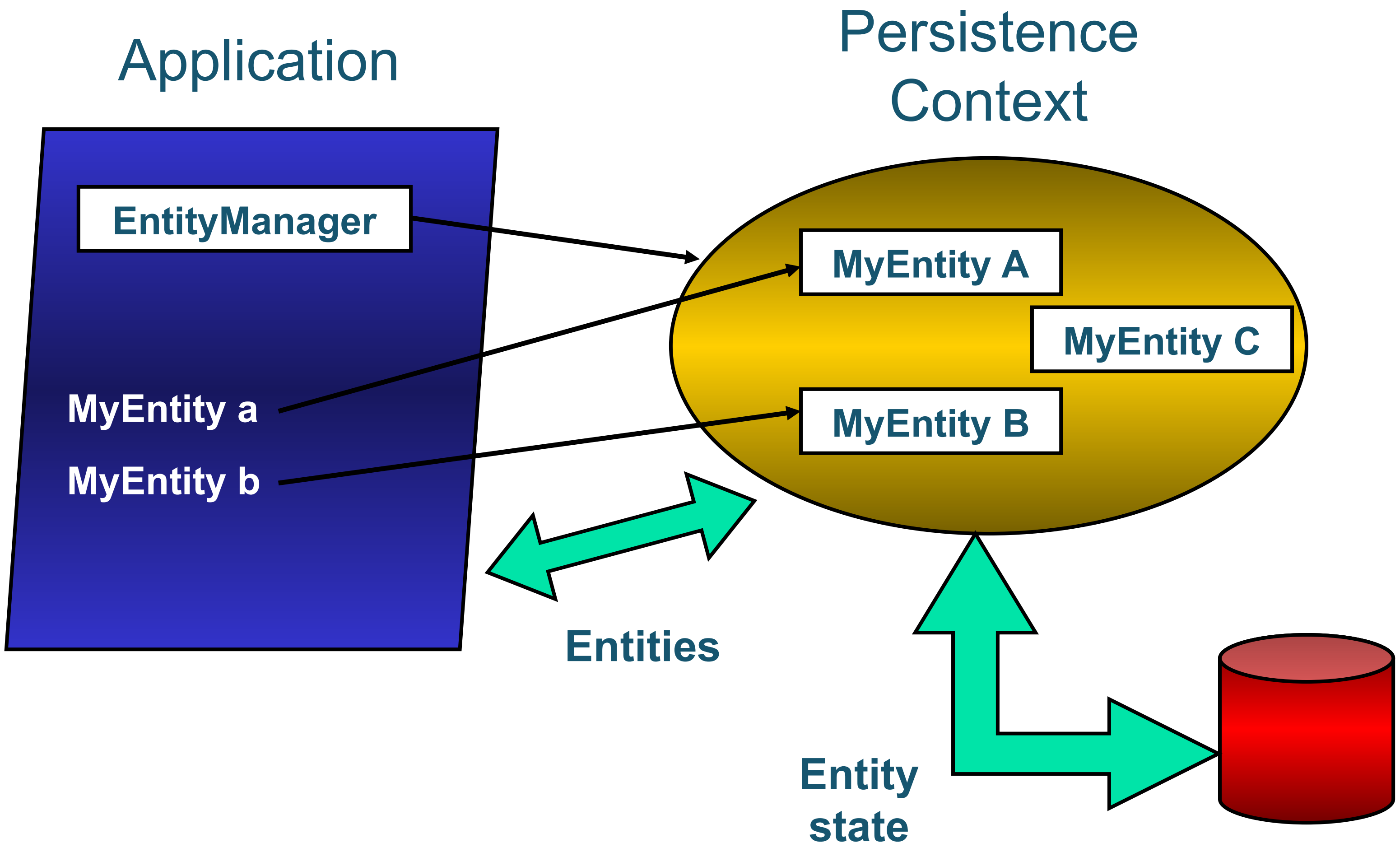

The Object Relational Model

- The JPA presents a Object Relational Model for Working with Enterprise Information System which is much in the confirmation with OOP concepts.
- It presented the capabilities to map the underlying database int Object – Relation Model or vice-versa
- It is proving to be a powerful tool to fetch, modify, manage and monitor the underlying data
- It follows all basic principles of Java and provides versatility to the code with data specific annotations and strategies

What JPA Does ?


JPA
POJO-based persistence model
Inheritance, polymorphism, etc.
Support for enriched domain modeling
Expanded query language
Standardized object/relational mapping
Using annotations and/or XML
Usable in Java EE and Java SE environments
Support for pluggable persistence providers

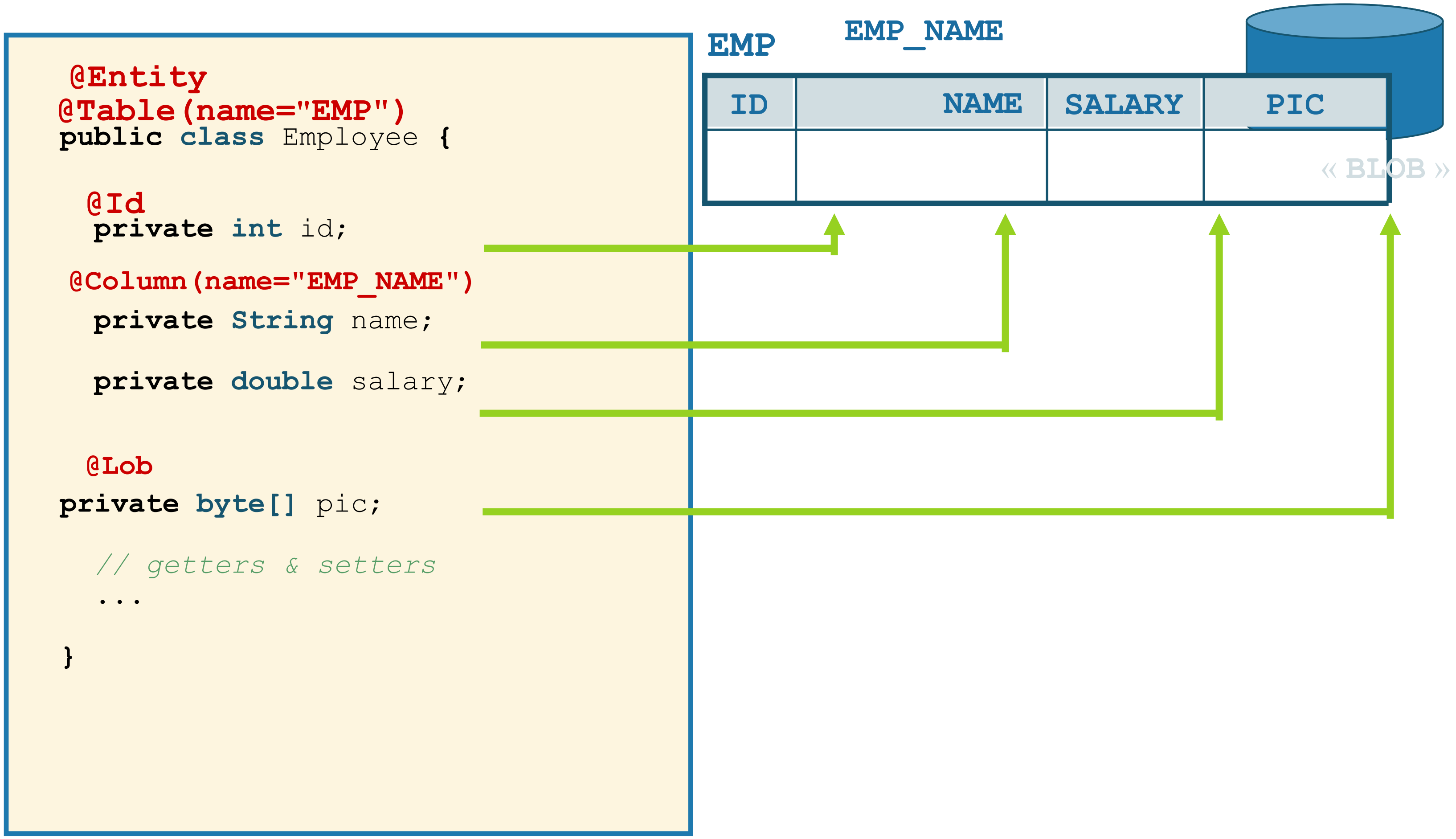
Persistence Context



Let's Laugh a Little

IT'S NOT BORING UP HERE – YOU GET TO LOOK THROUGH EVERYONE'S DATA!
© D.Fletcher for CloudTweaks.com

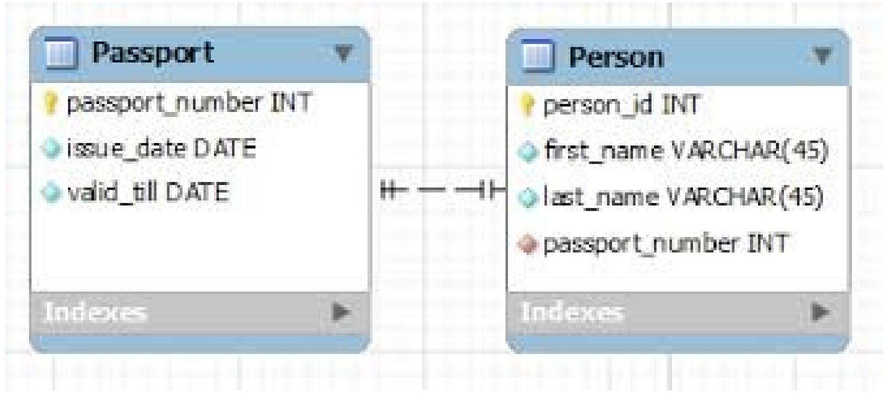
Java Persistence API: *Mappings* database

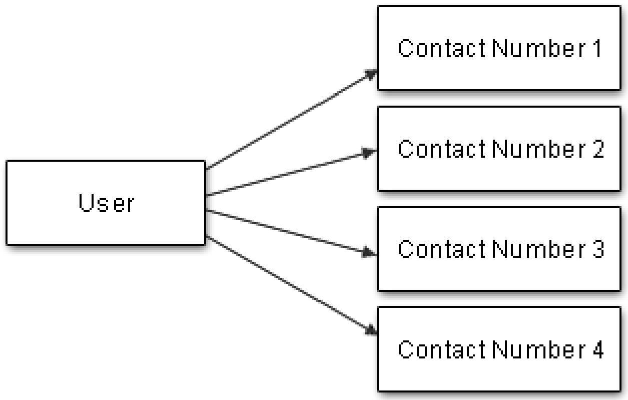


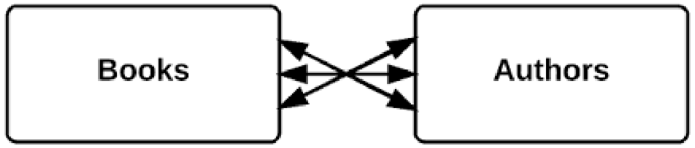
```
 @Entity
@Table(name="EMP")
public class Employee {

  @Id
  private int id;

 @Column(name="EMP_NAME")
  private String name;

  private double salary;


  @Lob
private byte[] pic;

  // getters & setters
  ...

}
```

JPA Entity Relationships

- One to One
- Many to One
- One to Many
- Many to Many



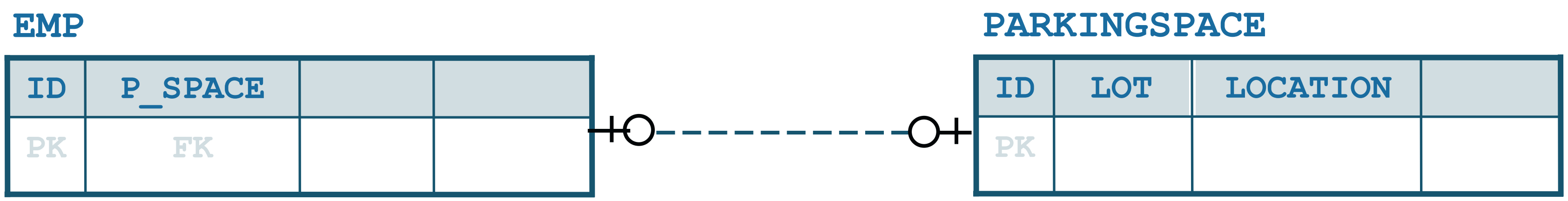




Relationships: One to One

```
@Entity
@Table(name="EMP")
public class Employee {

   @Id
   private int id;

  @OneToOne
   @JoinColumn(name="P_SPACE")
  private ParkingSpace space;

    // getters & setters
    ...

}
```

```
@Entity
public class ParkingSpace {

   @Id
   private int id;

   private int lot;

   private String location;

   @OneToOne(mappedBy="space")
  private Employee emp;

   // getters & setters
   ...

}
```



EMP

ID	P_SPACE		
PK	FK		

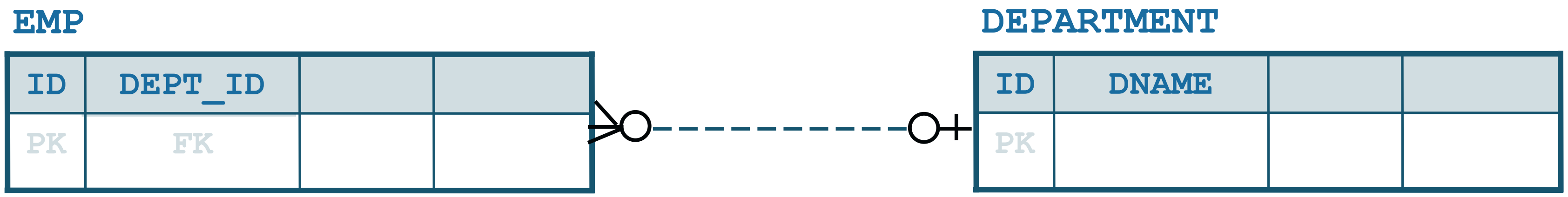
PARKINGSPACE

ID	LOT	LOCATION	
PK			

Relationship: Many to One

```
@Entity
@Table(name="EMP")
public class Employee {

  @Id
  private int id;


  @ManyToOne
 @JoinColumn(name="DEPT_ID")
  private Department d;


  // getters & setters
  ...

}
```

```
@Entity
public class Department {

  @Id
  private int id;

  private String dname;


  // getters & setters
  ...

}
```



EMP

ID	DEPT_ID		
PK	FK		

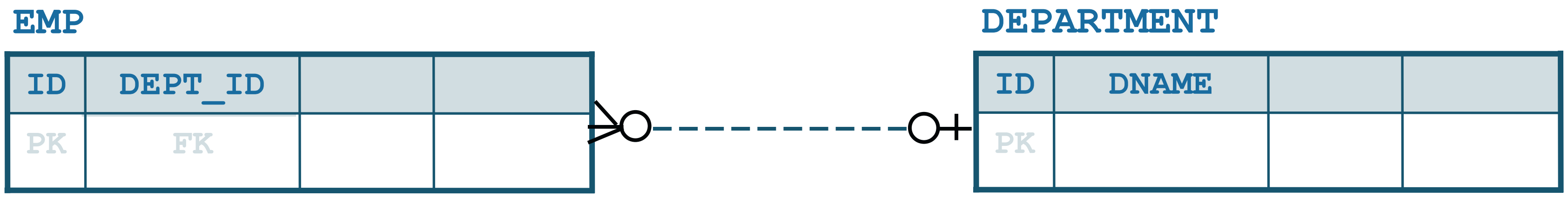
DEPARTMENT

ID	DNAME		
PK			

Relationship: One to Many

```java
@Entity
@Table(name="EMP")
public class Employee {

   @Id
   private int id;

    @ManyToOne
  @JoinColumn(name="DEPT_ID")
   private Department d;


   // getters & setters
   ...

}
```

```java
@Entity
public class Department {

   @Id
   private int id;

   private String dname;

   @OneToMany(mappedBy="d")
   private Collection<Employee> emps;

   // getters & setters
   ...

}
```



EMP

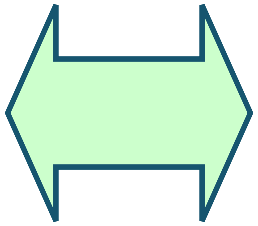
ID	DEPT_ID		
PK	FK		

DEPARTMENT

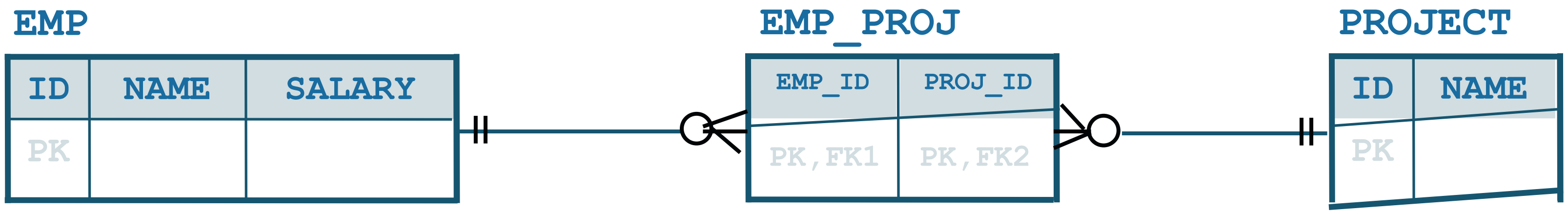
ID	DNAME		
PK			

Relationships: Many to Many

```
@Entity
@Table(name="EMP")
public class Employee {

 @Id
 private int id;

 @JoinTable(name="EMP_PROJ",
      joinColumns=
      @JoinColumn(name="EMP_ID"),
      inverseJoinColumns=
      @JoinColumn(name="PROJ_ID"))
 @ManyToMany
private Collection<Project> p;
}
```



```
@Entity
public class Project {

   @Id
   private int id;

   private String name;

   @ManyToMany(mappedBy="p")
   private Collection<Employee> e;

   // getters & setters
   ...

}
```



EMP

ID	NAME	SALARY
PK		

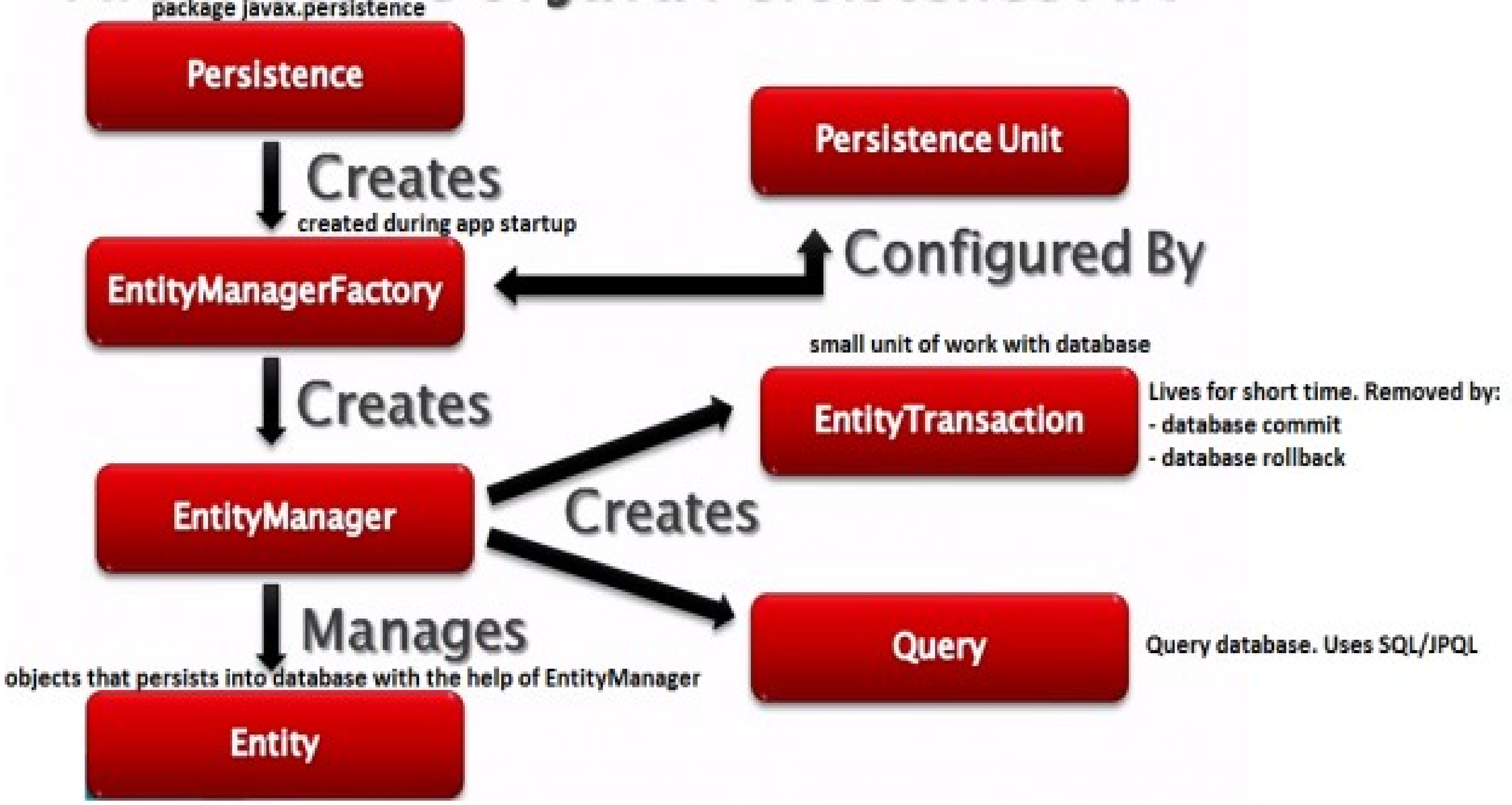
EMP_PROJ

EMP_ID	PROJ_ID
PK,FK1	PK,FK2

PROJECT

ID	NAME
PK	

Architecture of Java Persistence API


package javax.persistence
Persistence
Creates
created during app startup
EntityManagerFactory
Persistence Unit
Configured By
Creates
EntityManager
small unit of work with database
EntityTransaction
Lives for short time. Removed by:
- database commit
- database rollback
Creates
Query
Query database. Uses SQL/JPQL
Manages
objects that persists into database with the help of EntityManager
Entity

Persistence Context

- Abstraction representing a set of “managed” entity instances
  - Entities keyed by their persistent identity
  - Only one entity with a given persistent identity may exist in the PC
  - Entities are added to the PC, but are not individually removable (“detached”)
- Controlled and managed by EntityManager
  - Contents of PC change as a result of operations on EntityManager API

Entity Manager

- Client-visible artifact for operating on entities
  - API for all the basic persistence operations
- Can think of it as a proxy to a persistence context
  - May access multiple different persistence contexts throughout its lifetime
- Multi-dimensionality leads to different aspects of EntityManager (and persistence context) naming
  - Transaction type, life cycle

Operations on Entities

- EntityManager API
  - `persist()` - Insert the state of an entity into the db
  - `remove()` - Delete the entity state from the db
  - `refresh()` - Reload the entity state from the db
  - `merge()` - Synchronize the state of detached entity with the pc
  - `find()` - Execute a simple PK query
  - `createQuery()` - Create query instance using dynamic JP QL
  - `createNamedQuery()` - Create instance for a predefined query
  - `createNativeQuery()` - Create instance for an SQL query
  - `contains()` - Determine if entity is managed by pc
  - `flush()` - Force synchronization of pc to database

persist()

- Insert a new entity instance into the database
- Save the persistent state of the entity and any owned relationship references
- Entity instance becomes managed

```
public Customer createCustomer(int id, String name) {
    Customer cust = new Customer(id, name);
    entityManager.persist(cust);
    return cust;
}
```

find() and remove()

- find()
  - Obtain a managed entity instance with a given persistent identity – return null if not found
- remove()
  - Delete a managed entity with the given persistent identity from the database

```
public void removeCustomer(Long custId) {
    Customer cust =
     entityManager.find(Customer.class, custId);
    entityManager.remove(cust);
}
```

merge()

- State of detached entity gets merged into a managed copy of the detached entity
- Managed entity that is returned has a different Java identity than the detached entity

```
public Customer storeUpdatedCustomer(Customer cust) {
    return entityManager.merge(cust);
}
```

Queries

- Dynamic or statically defined (named queries)
- Criteria using JP QL (extension of EJB QL)
- Native SQL support (when required)
- Named parameters bound at execution time
- Pagination and ability to restrict size of result
- Single/multiple-entity results, data projections
- Bulk update and delete operation on an entity
- Standard hooks for vendor-specific hints

Queries

- Query instances are obtained from factory methods on EntityManager
- Query API:

`getResultList()` – execute query returning multiple results

`getSingleResult()` – execute query returning single result

`executeUpdate()` – execute bulk update or delete

`setFirstResult()` – set the first result to retrieve

`setMaxResults()` – set the maximum number of results to retrieve

`setParameter()` – bind a value to a named or positional parameter

`setHint()` – apply a vendor-specific hint to the query

`setFlushMode()`– apply a flush mode to the query when it gets run

Dynamic Queries

- Use createQuery() factory method at runtime and pass in the JP QL query string
- Use correct execution method
  - getResultList(), getSingleResult(), executeUpdate()
- Query may be compiled/checked at creation time or when executed
- Maximal flexibility for query definition and execution

Dynamic Queries

```
public List findAll(String entityName){
     return entityManager.createQuery(
      "select e from " + entityName + " e")
        .setMaxResults(100)
        .getResultList();
  }
```

- Return all instances of the given entity type
- JP QL string composed from entity type. For example, if “Account” was passed in then JP QL string would be: “**select e from Account e**”

Named Queries

- Use createNamedQuery() factory method at runtime and pass in the query name
- Query must have already been statically defined either in an annotation or XML
- Query names are “globally” scoped
- Provider has opportunity to precompile the queries and return errors at deployment time
- Can include parameters and hints in static query definition

Named Queries

```
@NamedQuery(name="Sale.findByCustId",
  query="select s from Sale s
         where s.customer.id = :custId
         order by s.salesDate")

public List findSalesByCustomer(Customer cust) {
  return
  entityManager.createNamedQuery(
                              "Sale.findByCustId")
         .setParameter("custId", cust.getId())
         .getResultList();
}
```

- Return all sales for a given customer

Object/Relational Mapping

- Map persistent object state to relational database
- Map relationships to other entities
- Metadata may be annotations or XML (or both)
- Annotations
  - Logical—object model (e.g. @OneToMany)
  - Physical—DB tables and columns (e.g. @Table)
- XML
  - Can additionally specify scoped settings or defaults
- Standard rules for default db table/column names

Object/Relational Mapping

- State or relationships may be loaded or “fetched” as EAGER or LAZY
  - LAZY - hint to the Container to defer loading until the field or property is accessed
  - EAGER - requires that the field or relationship be loaded when the referencing entity is loaded
- Cascading of entity operations to related entities
  - Setting may be defined per relationship
  - Configurable globally in mapping file for persistence-by-reachability

Entity Transactions

- Only used by Resource-local EntityManagers
- Isolated from transactions in other EntityManagers
- Transaction demarcation under explicit application control using EntityTransaction API
  - begin(), commit(), rollback(), isActive()
- Underlying (JDBC) resources allocated by EntityManager as required

Bootstrap Classes

`javax.persistence.Persistence`

- Root class for bootstrapping an EntityManager
- Locates provider service for a named persistence unit
- Invokes on the provider to obtain an EntityManagerFactory

`javax.persistence.EntityManagerFactory`

- Creates EntityManagers for a named persistence unit or configuration

EntityManager: declare the *persistence unit*

The file persistence.xml, is the place where one declares our persistence links. It is also where our persistence manager is configured.

```
<?xml version="1.0"?>

<persistence>
    <persistence-unit name="emp">
        <jta-data-source>jdbc/EmployeeDS</jta-data-source>
     <!-- autres propriétés du persistence provider -->
    </persistence-unit>
</persistence>
```

persistence.xml

Transactions: JTA

To use the JTA, it is necessary to put the hand on the transaction in progress.

```
public class MyServlet extends HttpServlet {
    …
    @Resource UserTransaction utx;

    public void doGet(…) {
    utx.begin()

    // persistence operations …

    utx.commit();
    }
}
```

Support from industry

The JPA made concensus with JavaOne2006, all the large actors find their account there, and until proof of the opposite, will support and endorse this technology.