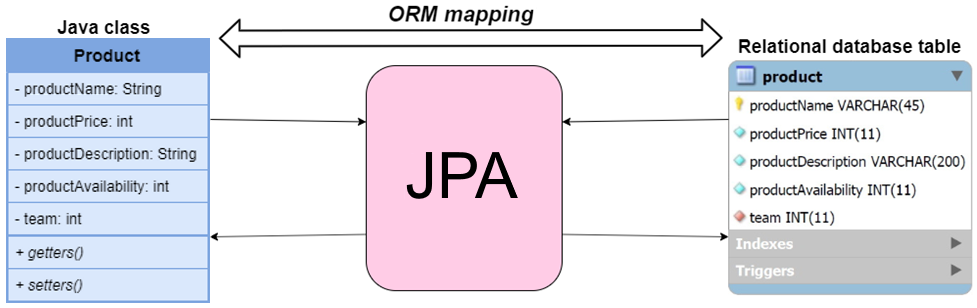
Java Persistence API

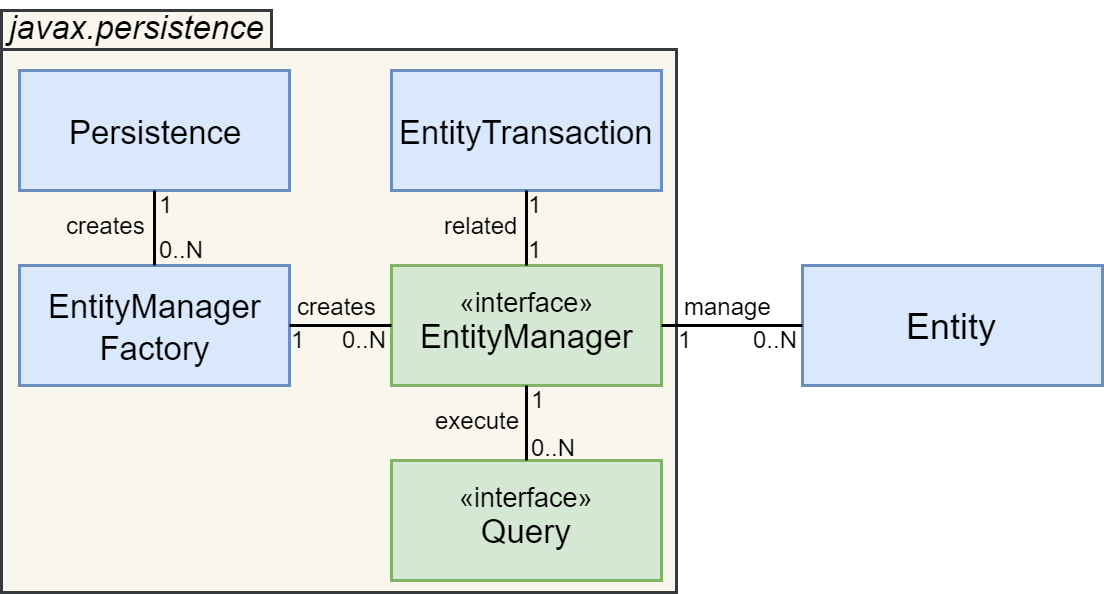
The Java Persistence API (JPA) is the standard Java specification for accessing, persisting, managing and exchanging data between Java objects (or classes) and a relational database.   
First introduced in 2006, the main goal of this API was to address the mismatch between how the data is represented and handled in an object-oriented programming language such as Java and the way it is managed in RDBMS, a discrepancy that raised the following issues during an application’s development:

* **Data Modelling –** Tables in a relational database strictly rely on concepts such as their primary key (PK) and referential integrity constraints with other tables, which have no equivalents in object-oriented languages.
* **Type Mismatch –** The fact that each RDBMS has its own set of primitive data types differing in general from the one provided in Java makes it difficult to directly map objects into tables, particularly in the case of objects with attributes of derived data types (i.e. other objects).
* **Type Specializations –** Having no notion of objects RDBMS lack the support for fundamental paradigms of object-oriented languages such as inheritance.
* **Access Control –** RDBMS lack the access-level modifiers typical of object-oriented programming languages such as *private*, *public*, *protected*, *package*.
* **Higher Development Costs –** In order to exchange data with the underlying RDBMS programmers were required to write an additional layer of logic to handle the conversion of objects into one or (typically) multiple queries that could be parsed by the database, which added to the development’s costs.

The JPA addresses these issues by offering an **Object-Relational Mapping (ORM)** mechanism, which allows programmers to persist POJO (Plain Old Java Objects) into tables of a relational database while concealing the implementation details of the creation, management and data exchange between the underlying database and the application.  
 

JPA Architecture

The classes and interfaces offered by the JPA are collected in the *javax.persistance* package, where the relationships between the main classes and interfaces are outlined below:



|  |  |
| --- | --- |
| Name | Description |
| Persistence | A class containing static methods which among other things allow to instantiate EntityManagerFactory objects |
| EntityManagerFactory | A class which is used to instantiate and manage one or more EntityManager objects |
| EntityManager | An interface which manages the persistence operations of one or more Entities. It is associated with an entityTransaction class and can execute multiple queries |
| EntityTransaction | A class which handles the persistence operations of a single EntityManager |
| Query | An interface which must be implemented (by a JPA vendor) in order to establish a connection with the target RDBMS |
| Entity | The persistence object that must be stored as a table in an RDBMS, consisting in our case of a Java class |

JPA Tutorial

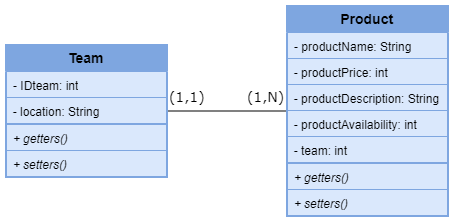
We now present a reference tutorial on how the JPA can be used in practice to persist the data of a Java application into a relational database, where the following working conditions are assumed:

* The *Hibernate* implementation of the JPA is used
* The data must be persisted in an underlying MySQL database
* The Hibernate and the Java Database Connector (JDBC) libraries are included in the project

The steps generally required to persist the data of a Java application through the JPA are:

1. Define a set of parameters relative to the database to connect to in a *persistence.xml* file.
2. Write the POJO classes relative to the objects that must be persisted.
3. Derive the POJO classes into persistence entities by appropriately adding a set of annotations to their definitions, which instruct the JPA on how such persistence entities should be mapped into database tables and their relationships with other entities.
4. Use the *persistence* class to instantiate an *EntityManagerFactory* object.
5. Use the *EntityManagerFactory* object to instantiate an *EntityManager* object.
6. For each operation that must be applied on the persistence entities:
   1. Use the *EntityManager* object to initiate a transaction with the database.
   2. Execute the set of operations on the persistence entities in question, which will be mapped by the JPA into the relative CRUD queries.
   3. Use the *EntityManager* object to commit the transaction into the database
7. Once all operations are finished, close the *EntityManager* and *EntityManagerFactory* instances.

As an example of the tutorial we suppose that we must persist the information relative to the products sold by a tech company and the teams of employees in charge of their assembly, as shown in the diagram below:



1. Create the persistence.xml file

The *persistence.xml* is an XML file containing a set of properties that are used by the JPA to connect and exchange data with the target relational database, such as its server IP address and port, username and password, the database connector to use (in this case, JDBC), the JPA implementation to use (in this case, Hibernate) and so on.  
What follows is an example of *persistence.xml* file suited to our application:

persistence.xml



1. Create the POJO classes of the objects to persist

In our example the Java classes relative to objects to persist are the following:

Team.java



Product.java



1. Derive the POJO classes into persistence entities

To derive the POJO classes previously defined into persistence entities we need to complement their definitions with a set of special annotations which instruct the JPA on how such persistence entities should be mapped into database tables and their relationships with other entities, where a set of the most commonly used JPA annotations is shown below:

|  |  |  |
| --- | --- | --- |
| Annotation | Description | Example |
| @Entity | Specifies that the following class represents a *persistence entity*, and thus should be mapped into the database as a table with the same name | @Entity **public class** Myclass   { … } |
| @Table | Used after the @Entity annotation to override the name of the table that maps the class into the database | @Entity  @Table(name = ”myname”) **public class** Myclass   { … } |
| @Column | Used to specify how the following class attribute should be translated as an attribute of the table associated with the class, such as its name, its maximum length, and more | @Column(name = ”myattr”,   length = 25,   nullable = **false**)  **private** String name; |
| @Id | Specifies that the following class attribute is part of the primary key of the table associated with the class | @Id **private int** IDproduct; |
| @Generated Value | Used after the @Id annotation to specify the strategy to adopt in the generation of the value of the following primary key attribute | @Id  @GeneratedValue(strategy   = GenerationType.AUTO)  **private int** IDproduct; |
| @Transient | Specifies that the following class attribute should NOT be persisted into the database as an attribute of the corresponding table | @Transient  **private int** boxNumber; |
| @OneToOne | Specifies that the entity has a one-to-one relationship with the entity specified as the type of the following attribute, and possibly additional properties of such referential integrity constraint | @OneToOne(cascade =   CascadeType.ALL)  **private** Employee leader; |
| @PrimaryKey  JoinColumn | Used after the @OneToOne annotation to specify that the associated entities share the same primary key | @OneToOne @PrimaryKeyJoinColumn  **private** Team team; |
| @JoinColumn | Used after a @OneToOne or @ManyToOne annotation to specify the name of the Foreign Key exported to the related entity | @OneToOne @JoinColumn(name=”leader”)  **private** Team teamLeader; |
| @OneToMany | Specifies that the entity has a one-to-many relationship with the entity specified as the type of the following attribute, and possibly additional properties of such referential integrity constraint | @OneToMany (mappedBy = ”productName”)  **private** Set products; |
| @ManyToOne | Specifies that the entity has a many-to-one relationship with the entity specified as the type of the following attribute, and possibly additional properties of such referential integrity constraint | @ManyToOne (JoinColumn = "team")  **private** Team team; |
| @ManyToMany | Specifies that the entity has a many-to-many relationship with the entity specified as the type of the following attribute, and possibly additional properties of such referential integrity constraint | @ManyToMany  (mappedBy = "projects") **private** Set Employees; |

Thus in our example the classes of the entities to persist become the following:

Team.java



Product.java



**Step 3: define entities and entity managers.**

For the product class we define the class product.java:

1. The class productManager.java
2. **import** javax.persistence.EntityManagerFactory;
3. **import** javax.persistence.EntityManager;
4. **import** javax.persistence.Persistence;
6. **public** **class** ProductManagerEM{
8. **private** EntityManagerFactory factory;
9. **private** EntityManager entityManager;
11. **public** **void** setup(){
13. factory = Persistence.createEntityManagerFactory("InnovativeSolutionsDB");
14. }
16. **public** **void** exit(){
18. factory.close();
19. }

The class teamManager.java:

1. **import** javax.persistence.EntityManagerFactory;
2. **import** javax.persistence.EntityManager;
3. **import** javax.persistence.Persistence;
5. **public** **class** TeamManagerEM{
7. **private** EntityManagerFactory factory;
8. **private** EntityManager entityManager;
10. **public** **void** setup(){
12. factory = Persistence.createEntityManagerFactory("InnovativeSolutionsDB");
13. }
15. **public** **void** exit(){
17. factory.close();
18. }

**Step 4: introduce functions for CRUD operations**

All the following functions has to be introduced in the respective EntityManager.java file.