# Java Persistence API Guide

Table of Contents

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.
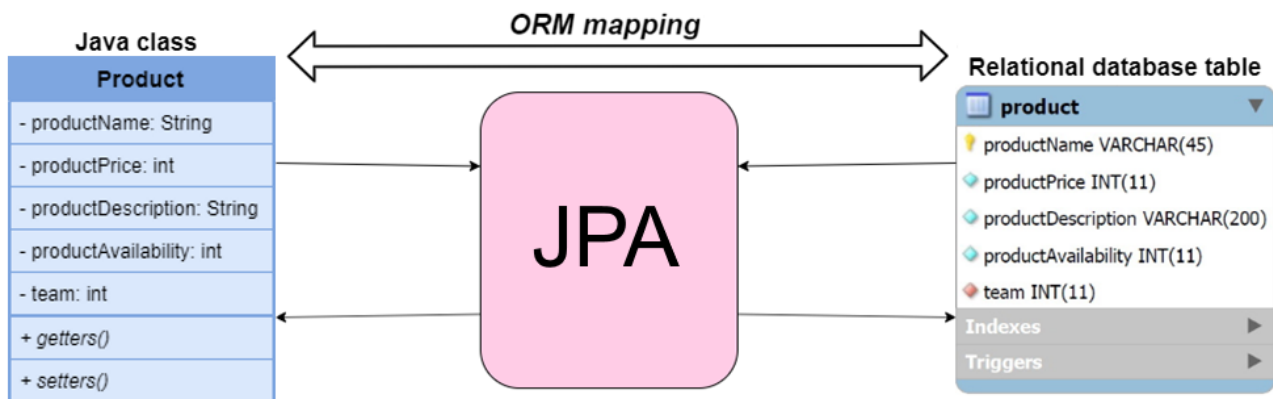
JPA Architecture

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:
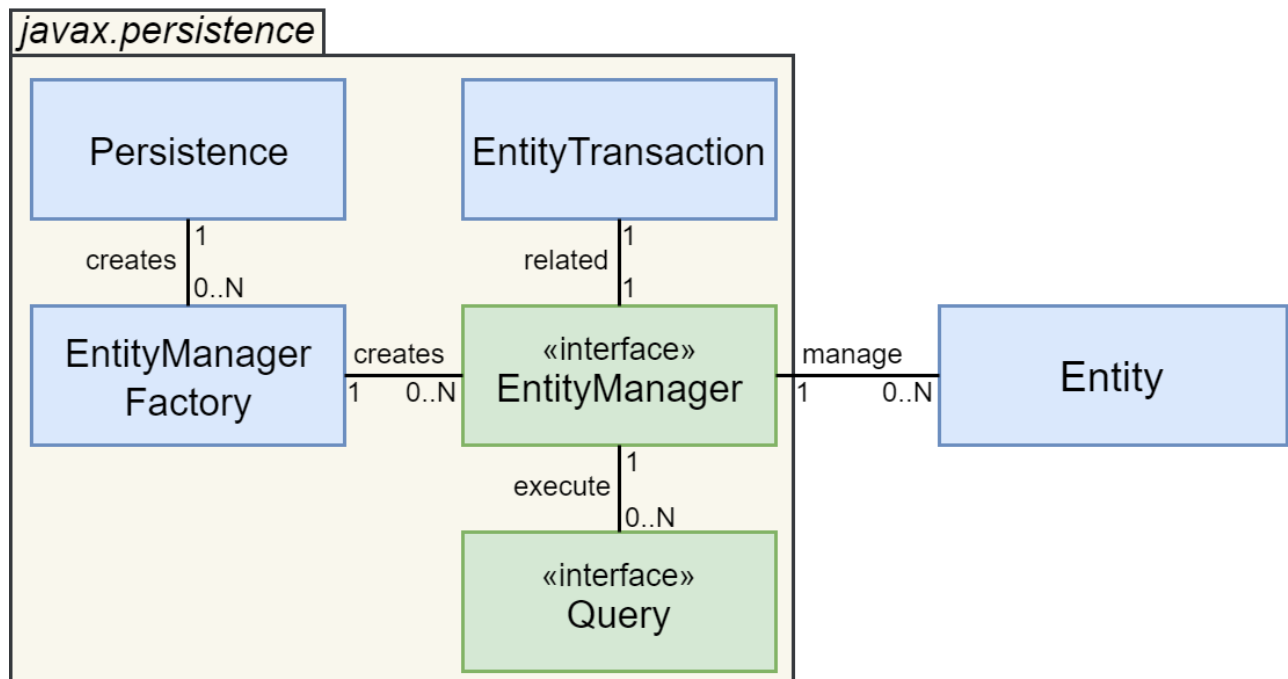
| Name	Description
Persistence	A class containing static methods which among other things allow to instantiate EntityManagerFactory objects
EntityManagerFactory	A class which is used to instantiate and manage one or more EntityManager objects
EntityManager	An interface which manages the persistence operations 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) Create an empty MySQL schema
2) Create the *persistence.xml* file
3) Create the POJO classes of the objects to persist
4) Derive the POJO classes into *persistence entities*
5) Implement the relationships between the *persistence entities*
6) Instantiate an *EntityManagerFactory* and an *EntityManager* object in the driver class of the application
7) For each operation on the persistence entities that must be persisted into the database
   7.1) Use the *EntityManager* object to initiate a transaction with the database
   7.2) Execute the set of operations required on the persistence entities in question
   7.3) Use the *EntityManager* object to persist each new or updated persistence entity into the database
   7.4) Use the *EntityManager* object to commit the transaction to the database
8) Once all operations on the persistence entities are concluded, close the *EntityManager* and *EntityManagerFactory* instances

As an example that we'll use throughout the tutorial, suppose 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 by the class diagram below:
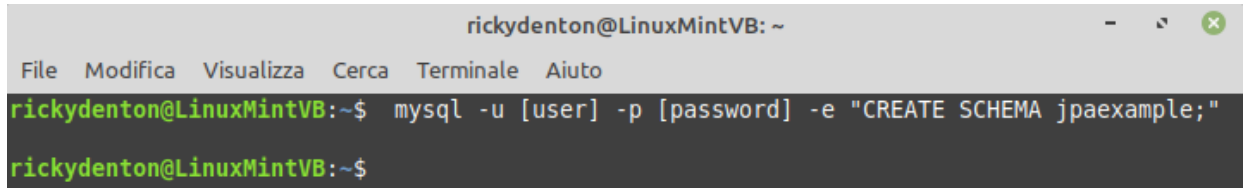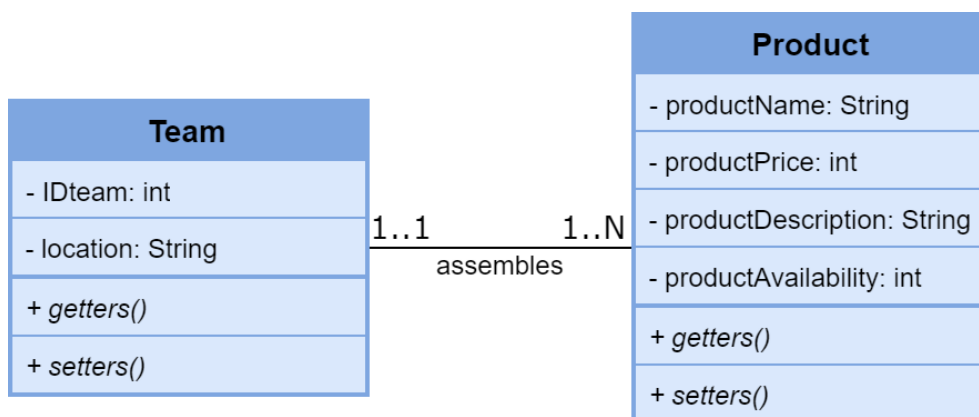
1) Create an empty MySQL Schema

Since in our example we are using MySQL as the underlying RDBMS, the first operation to perform consists in creating an empty database schema on the server side, which will be used by the JPA to persist the data of our application.

The schema can be created by executing the following SQL script:

CREATE SCHEMA jpaexample;

2) Create the persistence.xml file

The *persistence.xml* acts as configuration file for the JPA in the context of our application where, other than information relative to the JPA version to use, one or more *persistence units* can be defined, which represent a collection of settings that are used to configure the *EntityManagerFactory* and thus the *EntityManager* instances in our application.
The settings of each persistence unit are specified as a set of properties in the format (*name, value*), where the main settings are:

- Information on the target DBMS to use to persist data, such as its type (in our case, MySQL), its IP address and Port, the username and password to authenticate with, and the schema to use (in our case "jpaexample")

- Information on the connector to use to exchange data with the database (in our case, JDBC)

- Information and possible additional configuration relative to the JPA implementation to use (in our case, Hibernate)

An example of *persistence.xml* file suited to the requirements of our application is the following, which by default must be placed inside the META-INF folder within the classpath:

persistence.xml

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


<!-- JPA version information -->
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                                 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">


<!-- Persistence Unit definition -->
<persistence-unit name="JPAExample">
 <properties>

  <!-- General server parameters, such as its IP and port, RDBMS type and schema to use -->
  <property name="javax.persistence.jdbc.url"
            value="jdbc:mysql://localhost:3306/jpaexample?serverTimezone=UTC"/>

  <!-- JDBC parameters, such as the user and password used to connect to the database -->
  <property name="javax.persistence.jdbc.user" value="root"/>
  <property name="javax.persistence.jdbc.password" value="root"/>
  <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>

  <!-- JPA implementation-specific parameters (in this case, hibernate) -->
  <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
  <property name="hibernate.archive.autodetection" value="class"/>
  <property name="hibernate.hbm2ddl.auto" value="update"/>
  <property name="hibernate.show_sql" value="true"/>
  <property name="hibernate.format_sql" value="true"/>

 </properties>
</persistence-unit>
</persistence>
```

3) Create the POJO classes of the objects to persist

In our example the POJO classes relative to the objects to persist consist in the following:

Team.java

```java
public class Team
{
private int IDteam;
private String teamLeader;
private String location;

/* Constructors, Getters and Setters (omitted for brevity) */
}
```

Product.java

```java
public class Product
{
private String productName;
private String productDescription;
private int productPrice;
private int productAvailability;

/* Constructors, Getters and Setters (omitted for brevity) */
}
```

4) Derive the POJO classes into persistence entities

To derive the POJO classes into persistence entities we need to complement their definitions by adding a set of special annotations, which instruct the JPA on how such persistence entities should be mapped into database tables, with the most commonly used JPA annotations being 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;`

In our example the POJO classes can be derived into persistence entities by adding the following JPA annotations:

Team.java

```java
import javax.persistence.*;

@Entity
@Table(name = "team")
public class Team
{
@Id
@GeneratedValue(strategy = "GenerationType.AUTO")   //We let Hibernate automatically select
private int IDteam;                                 //a generation policy for the PK

@Column(name = "teamLeader", length = 45, nullable = false)
private String teamLeader;

@Column(name = "location", length = 45)
private String location;

/* Constructors, Getters and Setters (omitted for brevity) */
}
```

Product.java

```java
import javax.persistence.*;

@Entity
@Table(name = "product")
public class Product
{
@Id
@Column(name = "productName", length = 45, nullable = false)
private String productName;

@Column(name = "productDescription", length = 200, nullable = false)
private String productDescription;

@Column(name = "productPrice", nullable = false)
private int productPrice;

@Column(name = "productAvailability", nullable = false)
private int productAvailability;

/* Constructors, Getters and Setters (omitted for brevity) */
}
```

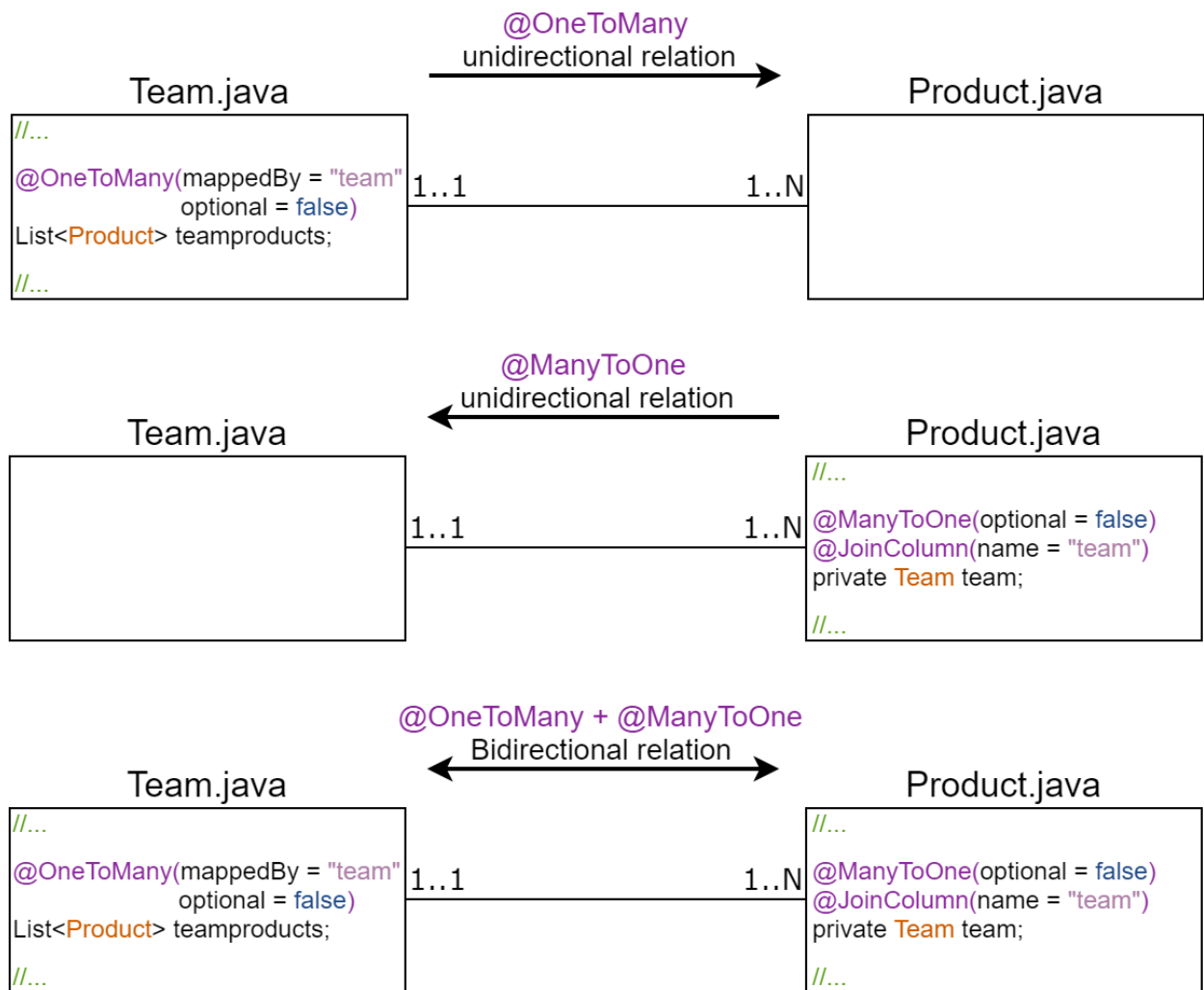# 5) Implement the relationships between the persistence entities

Next we must decide how to implement the existing relationships between the persistence entities, where in general each relationship can be realized as a <u>unidirectional relationship</u>, which is obtained by adding to one of the entities involved in the relation an attribute mapping the foreign entities objects related with each local entity object, or as a <u>bidirectional relationship</u>, where an attribute mapping the foreign objects related with each local object is added to *both* entities.

A unidirectional relationship allows to easily traverse the relation, i.e. retrieve the foreign objects related to each local object, from the entity where it is defined, whereas a bidirectional relationship allows to easily traverse the relationship in both directions, at the cost of introducing a redundancy in the database, and thus the choice of the type of relationship to use depends in general on whether we want to minimize the memory footprint or the access times of our application.

Whatever the choice, the relationship attributes introduced in the entities must also be complemented with the appropriate annotations chosen among the following, which specify to the JPA the type and other information on how to manage the relationship in question:

Annotation	Description	Example
@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;`

In our example, the One-To-Many relationship between the Team and Product entities can be implemented as follows:

@OneToMany
unidirectional relation

Team.java → Product.java

1..1   1..N

Team.java:
```
//...

@OneToMany(mappedBy = "team"
          optional = false)
List<Product> teamproducts;

//...
```

@ManyToOne
unidirectional relation

Team.java ← Product.java

1..1   1..N

Product.java:
```
//...

@ManyToOne(optional = false)
@JoinColumn(name = "team")
private Team team;

//...
```

@OneToMany + @ManyToOne
Bidirectional relation

Team.java ↔ Product.java

1..1   1..N

Team.java:
```
//...

@OneToMany(mappedBy = "team"
          optional = false)
List<Product> teamproducts;

//...
```

Product.java:
```
//...

@ManyToOne(optional = false)
@JoinColumn(name = "team")
private Team team;

//...
```

From here, as a design choice, we opt for the solution carrying the least memory occupancy, which is given by defining a @ManyToOne relationship in the Product entity:

Product.java

```java
import javax.persistence.*;

@Entity
@Table(name = "product")
public class Product
{
@Id
@Column(name = "productName", length = 45, nullable = false)
private String productName;

@Column(name = "productDescription", length = 200, nullable = false)
private String productDescription;

@Column(name = "productPrice", nullable = false)
private int productPrice;

@Column(name = "productAvailability", nullable = false)
private int productAvailability;

//Implements the Many-to-one relationship with the "Team" entity
@ManyToOne(optional = false)
@JoinColumn(name = "team")
private Team team;

/* Constructors, Getters and Setters (omitted for brevity) */
}
```

6) Instantiate an EntityManagerFactory and an EntityManager Object

Next, in order to persist the objects of our application, we need to instantiate in the driver class of our application:

- An *EntityManagerFactory* object, which is performed by using the *createEntityManagerFactory(String persistenceUnit)* static method of the *Persistence* class.

- An *EntityManager* object, which is performed by using the *CreateEntityManager()* method of the *EntityManagerFactory* object just created.

DBDriver.java

```java
import javax.persistence.*;

public class DBDriver
{
  static private EntityManagerFactory factory= Persistence.createEntityManagerFactory("DBDriver");
  static private EntityManager manager = factory.createEntityManager();

  public static void main(String[] args)
  {

   /* Application driver goes here */

  }

}
```

7) Execute the operations on the persistence entities

Next we can start executing the operations on the persistence entities as required by the logic of our application, where the following main methods of the *EntityManager* object can be used to start a transaction, retrieve, and persist the objects into the database:

ENTITYMANAGER	
Method	**Description**
EntityTransaction getTransaction()	Returns the *EntityTransaction* object associated with the *EntityManager* (typically used in conjunction with the *begin()* and *commit()* methods of the *EntityTransaction* object to start and commit a transaction)
void persist(Object entity)	Persists the persistence entity object into the database
void remove(Object entity)	Removes the persistence entity object from the database
void refresh(Object entity)	Updates the in-memory persistence entity with its state in the database
<T> T merge(T entity);	Merges the state of the persistence entity into the database
<T> T find(Class<T> entityClass, Object primaryKey)	Returns the persistence entity of the specified type and primary key
Query createQuery(String qlString)	Creates a Query object for executing a Java Persistence query language statement
void close()	Closes the Entity Manager object

What follows is a simple example where a new Team and a new Product object are created and persisted into the database:

DBDriver.java

```java
import javax.persistence.*;

public class DBDriver
{
  static private EntityManagerFactory factory= Persistence.createEntityManagerFactory("DBDriver");
  static private EntityManager manager = factory.createEntityManager();

  public static void main(String[] args)
  {
   //Starts a new transaction with the database
   manager.getTransaction().begin();

   //Creates a new team object (with an automatically generated primary key TeamID)
   Team team1 = new Team("John Smith", "London");

   //Creates a new Product object related with the previous Team object
   Product product1 = new Product("SmartWatch","A rubber smartwatch",25,0,team1);

   //Persist both objects into the database
   manager.persist(team1);
   manager.persist(product1);

   //Commits the transaction with the database
   manager.getTransaction().commit();

   /* … */

  }

}
```

8) Close the EntityManager and EntityManagerFactory instances

Once all the required operations on the persistence entities have been performed, the *EntityManager* and *EntityManagerFactory* objects can be closed using their *close()* methods, as follows:

DBDriver.java

```java
import javax.persistence.*;

public class DBDriver
{
  static private EntityManagerFactory factory= Persistence.createEntityManagerFactory("DBDriver");
  static private EntityManager manager = factory.createEntityManager();

  public static void main(String[] args)
  {
   //Transaction 1
   manager.getTransaction().begin();
   Team team1 = new Team("John Smith", "London");
   Product product1 = new Product("SmartWatch","A rubber smartwatch",25,0,team1);
   manager.persist(team1);
   manager.persist(product1);
   manager.getTransaction().commit();

   /* … */

   //Transaction N
   manager.getTransaction().begin();
   Team team4 = manager.find(Team.class, 4);
   Team team5 = manager.find(Team.class, 5);
   team4.setLocation(team5.location);
   manager.merge(team4);
   manager.remove(team5);
   manager.getTransaction().commit();

   //Close the EntityManager and EntityManagerFactory objects
   manager.close();
   factory.close();
  }

}
```

JPA Complete Example

A more comprehensive example of an application using the JPA to persist data into a database, where an instance of each CRUD (Create, Read, Update, Delete) operation is performed, is shown below:

Team.java

```java
package entities;

import javax.persistence.*;
import dbdriver.DBDriver;

@Entity
@Table(name = "team")
public class Team
{
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int IDteam;

 @Column(name = "teamLeader", length = 45, nullable = false)
 private String teamLeader;

 @Column(name = "location", length = 45)
 private String location;

 /****************************************************************
  *                         Constructors
  ****************************************************************/
 public Team()
  {}

 public Team(String teamLeader,String location)
  {
    this.teamLeader = teamLeader;
    this.location = location;
  }

 public Team(int iDteam,String teamLeader,String location)
  {
    IDteam = iDteam;
    this.teamLeader = teamLeader;
    this.location = location;
  }

 /****************************************************************
  *                         Getters
  ****************************************************************/
 public int getIDteam()
  { return IDteam; }

 public String getTeamLeader()
  { return teamLeader; }

 public String getLocation()
  { return location; }

 /****************************************************************
  *                         Setters
  ****************************************************************/
 public void setIDteam(int id)
  { IDteam = id; }

 public void setTeamLeader(String leader)
  { teamLeader = leader; }

 public void setLocation(String loc)
  { location = loc; }
```

```java
/****************************************************************
 *                        JPA Utilities
 ****************************************************************/
public boolean addTeamProduct(String productName,String productDescription,int productPrice,int
productAvailability)
{
  try
    {
      DBDriver.getEntityManager().getTransaction().begin();
      Product p = new Product(productName,productDescription,productPrice,productAvailability,this);
      DBDriver.getEntityManager().merge(this);
      DBDriver.getEntityManager().persist(p);
      DBDriver.getEntityManager().getTransaction().commit();
      System.out.println("[Team "+ IDteam +"]: Added product ("+ p.getProductName() +")");
      return true;
    }
  catch(IllegalStateException|RollbackException e)
    {
      e.printStackTrace();
      System.out.println("[Team "+ IDteam +"]: error in adding product (" + productName + ")");
      return false;
    }
}


public boolean changeLocation(String loc)
{
  String temp =  location;
  try
    {
      DBDriver.getEntityManager().getTransaction().begin();
      location = loc;
      DBDriver.getEntityManager().merge(this);
      DBDriver.getEntityManager().getTransaction().commit();
      System.out.println("[Team "+ IDteam +"]: location changed from \"" + temp + "\" to \"" + loca-
tion + "\"");
      return true;
    }
  catch(IllegalStateException|RollbackException e)
    {
      e.printStackTrace();
      System.out.println("[Team "+ IDteam +"]: error in changing location from \"" + temp + "\" to
\"" + location + "\"");
      return false;
    }
}


/****************************************************************
 *                        Overrides
 ****************************************************************/
@Override
public String toString()
  { return new String("[Team " + IDteam +"]: Team Leader: \"" + teamLeader + "\", Location: \"" +
location + "\""); }

}
```

Product.java

```java
package entities;

import javax.persistence.*;
import dbdriver.DBDriver;

@Entity
@Table(name = "product")
public class Product
{
 @Id
 @Column(name = "productName", length = 45, nullable = false)
 private String productName;

 @Column(name = "productDescription", length = 200, nullable = false)
 private String productDescription;

 @Column(name = "productPrice", nullable = false)
 private int productPrice;

 @Column(name = "productAvailability", nullable = false)
 private int productAvailability;

 @ManyToOne(optional = false)
 @JoinColumn(name = "team")
 private Team team;

 /***************************************************************
  *                      Constructors
  ***************************************************************/

 public Product()
  {}

 public Product(String productName,String productDescription,int productPrice,int productAvailabil-
ity,Team team)
  {
    this.productName = productName;
    this.productDescription = productDescription;
    this.productPrice = productPrice;
    this.productAvailability = productAvailability;
    this.team = team;
  }

 /***************************************************************
  *                        Getters
  ***************************************************************/
 public String getProductName()
  { return productName; }

 public String getProductDescription()
  { return productDescription; }

 public int getProductPrice()
  { return productPrice; }

 public int getProductAvailability()
  { return productAvailability; }

 public Team getTeam()
  { return team; }
```

```java
/***************************************************************
 *                        Setters
 ***************************************************************/
public void setProductName(String productName)
 { this.productName = productName; }

public void setProductDescription(String productDescription)
 { this.productDescription = productDescription; }

public void setProductPrice(int productPrice)
 { this.productPrice = productPrice; }

public void setProductAvailability(int productAvailability)
 { this.productAvailability = productAvailability; }

public void setTeam(Team team)
 { this.team = team; }


/***************************************************************
 *                        JPA Utilities
 ***************************************************************/
public void printTeamInformation()
 {
  System.out.println("["+ productName + "] --> "+ team);
 }


public boolean deleteProduct()
 {
  try
   {
    DBDriver.getEntityManager().getTransaction().begin();
    DBDriver.getEntityManager().remove(this);
    DBDriver.getEntityManager().getTransaction().commit();
    System.out.println("[" + productName + "]: product removed from the database");
    return true;
   }
  catch(IllegalStateException|RollbackException e)
   {
    e.printStackTrace();
    System.out.println("[" + productName + "]: Error in removing product from the database");
    return false;
   }
 }


/***************************************************************
 *                        Overrides
 ***************************************************************/
@Override
public String toString()
 { return new String("["+ productName +"]: ProductDescription: \"" + productDescription + "\",
productPrice: " + productPrice + ", productAvailability: " + productAvailability + ", Team: " +
team.getIDteam() ); }

}
```

DBDriver.java

```java
package dbdriver;

import javax.persistence.*;
import entities.*;

public class DBDriver
{
  static private EntityManagerFactory factory= Persistence.createEntityManagerFactory("JPAExam-
ple");
  static private EntityManager manager = factory.createEntityManager();

  static public EntityManager getEntityManager()
   { return manager; }

  public static void main(String[] args)
  {
   //Add a new team product (CREATE)
   Team t1 = manager.find(Team.class,1);
   t1.addTeamProduct("ICameraDrone","A flying drone with a 20MPX camera",150,2);

   //Print the leader of the team in charge of the assembly of a certain product (READ)
   Product p1 = manager.find(Product.class,"ISmartLock");
   p1.printTeamInformation();

   //Change the location of a team (UPDATE)
   Team t2 = manager.find(Team.class,2);
   t2.changeLocation("Paris");

   //Delete an existing product (DELETE)
   Product p2 = manager.find(Product.class,"ICameraDrone");
   p2.deleteProduct();

   manager.close();
   factory.close();
  }

}
```

output.txt

```
[Team 1]: added product (ICameraDrone)

[ISmartLock] --> [Team 2]: Team Leader: "James", Location: "Paris"

[Team 2]: location changed from "Washington" to "Paris"

[ICameraDrone]: product removed from the database
```