Hibernate is an open source ORM framework for Java developers which provides JPA implementation along with its own native API. RebelLabs survey report of 2014 tells that about 67.5% of Java applications use Hibernate as an ORM framework. According to a survey of ITJobsWatch.co.uk the demand for the Hibernate coders is constantly high.

|  |  |  |
| --- | --- | --- |
| Java tools and technologies landscape 2014 orm framework in use graph  [View](http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/10/) the RebelLabs survey report of 2014 |  | https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_2242606999290469600_shared/web-hosted/assets/DPTH_JobDemandtreand_15Sep16_1215.png    [View](http://www.itjobswatch.co.uk/jobs/uk/hibernate.do) the hibernate jobs demand trend (Checked on 15-Sep-2016) |

Hibernate was started in 2001 by Galvin King, as a technology for persistence and object-relational mapping. In 2006 JPA was released by Java Community Process, to standardize the persistence process. JPA incorporated many features from third-party persistence frameworks like Hibernate and TopLink Essentials.

So Hibernate, in 2010, with its third version released, aligning to all the JPA specifications, became a certified implementation of JPA. Hence Hibernate can be called as a super set of JPA, as it comprises both JPA implementation and Hibernate Native APIs (including Criteria, Second-level caching, support for collection data-type, etc).

Observations on the movie ticket booking scenario

* The booking details for the movie was stored in some repository.
* Both Annie and Scott opened one application from their desktop and could check the number of available tickets.
* When Annie booked the tickets the number of available tickets got updated permanently in the repository.
* When Scott checked again he could see the latest details of availability.

In computer science terminology, we define the phenomenon of making a modification permanent to the storage system as **Data Persistence**.

Implementing data persistence is one of the most critical challenges of an Enterprise application. In any domain, the greatest concern for the enterprise application is data, which is enormous in size and critical in nature. So implementing a proper data persistence strategy will allow the application to be scalable, maintainable, and efficient.





The persistence phenomenon consists of three core components: **Data, Medium, and Storage.**



The **data**that needs to be persisted can be:

* Raw data: which is collected from a file or any other source in the form of bytes.
* Java object: which is the data contained in the object of a Java class.

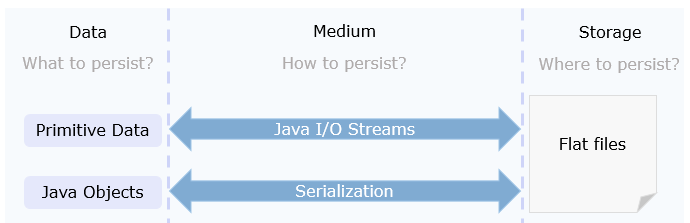
Data can be persisted using RAM or secondary storage device like a hard-drive. Further, logical storage devices like DB or Files are also be used for storing the data.

To persist the data, Java provides **mediums** like:

* I/O Streams and Serialization
* JDBC
* ORM Frameworks like Hibernate

The **Java Input-Output(I/O) API**provides classes for performing input and output operations on raw data.

* These classes are available in the java.io package.
* Java I/O API is built on four abstract classes. This depends upon the type of data it can handle (byte/character).
  + InputStream and OutputStream: deals with bytes.
  + Reader and Writer: deals with character.



**Serialization** helps in sending Java objects through the network and this also can be used to store these Java objects in a file.

* An object can be marked serializable by implementing the **java.io.Serializable** interface.
* Serializable objects can be converted into a stream of bytes.
* This stream of bytes can be written into a file.
* These bytes can be read back to re-create the object.
* Deserialization is the process of retrieving an object from the byte streams.

**Java I/O** APIs pretty much covers all the functionalities as a data persistence medium.

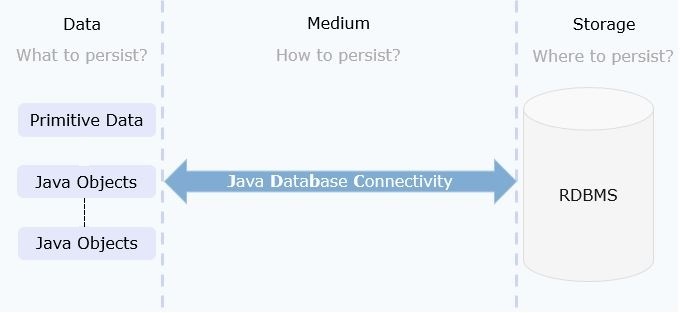
* But working with the File system is very difficult and inefficient in handling large and complex data.
* And using Java I/O also need lower-level details of the data to be retrieved, stored, or manipulated.

**Serialization** too has its own disadvantages:

* Since storing and retrieval of the entire object graph is done at once, it is not a suitable approach while working with a large amount of data.
* Concurrent access is not possible.
* It provides no query capabilities.
* The data cannot be retrieved without de-serialization.

**JDBC or 'Java Database Connectivity'** is a Java Core API for performing database interaction.

* Using JDBC API, a Java application can access a variety of databases such as MySQL, Oracle, etc.
* JDBC follows a relational database-oriented approach to work with the data using SQL queries.



The problem with Serialization is solved by JDBC, but it does not store the Java objects directly. The data from the objects need to be converted into a SQL query and then executed, for persistence.

* SQL code has to be embedded within Java Programs which makes it non-portable.
* JDBC API allows the developer to fire the SQL queries from the Java code. This means the developer needs to know the specific SQL constructs for the Relational Database Management System (RDBMS) used.
* Also, it is the responsibility of the programmer to make sure that the data model and the object model are synchronized properly.

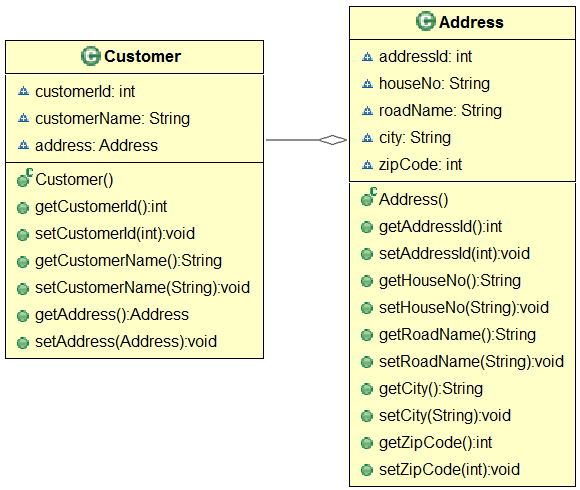
Due to this JDBC API is not a maintainable solution for enterprise applications.

DBC, I/O, Serialization do not solve the problem of data persistence effectively. For a medium to be effective, it needs to take care of the fundamental difference in the way Object-Oriented Programs(OOP) and RDBMS deals with the data.

* In Programming languages like Java, the related information or the data will be persisted in the form of hierarchical and interrelated objects.
* In the relational database, the data is persisted as table format or relations.

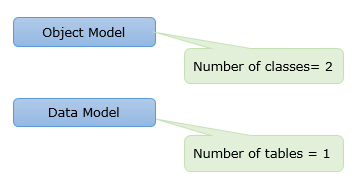
The greatest challenge in integrating the concepts of RDBMS and OOP is a mapping of the Java objects to databases. When object and relational paradigms work with each other, a lot of technical and conceptual difficulties arise, as mapping of an object to a table may not be possible in all the contexts.

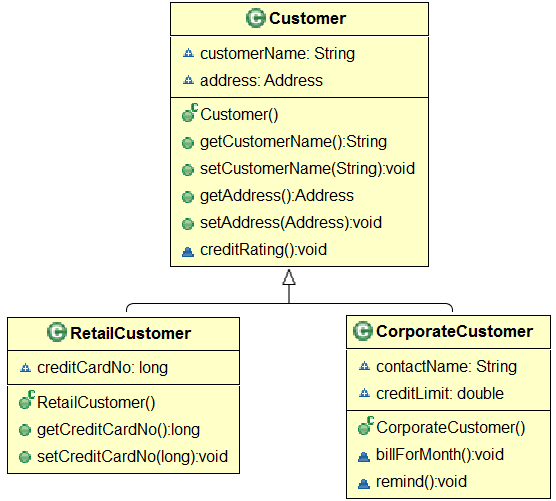
Storing and retrieving Java objects using a Relational database exposes a paradigm mismatch called "Object-Relational Impedance Mismatch". These differences are because of perception, style, and patterns involved in both the paradigms that lead to the following paradigm mismatches:

* Granularity: Mismatch between the number of classes in the object model and the number of tables in the relational model.
* Inheritance or Subtype: Inheritance is an object-oriented paradigm that is not available in RDBMS.
* Associations: In object-oriented programming, the association is represented using reference variables, whereas, in the relational model foreign keys are used for associating two tables.
* Identity: In Java, object equality is determined by the "==" operator or "equals()" method, whereas in RDBMS, uses the primary key to uniquely identify the records.
* Data Navigation: In Java, the dot(.) operator is used to travel through the object network, whereas, in RDBMS join operation is used to move between related records.
* Let us understand the problem of Granularity with the Customer and Address example.
* Consider the object model of Customer and Address as depicted below. Each Customer has an Address as shown below.
* 
* In the database, the Customer details can be represented as a single Customer table as shown below

| CUSTOMER | | | | | |
| --- | --- | --- | --- | --- | --- |
| **CUSTOMERID** | **CUSTOMERNAME** | **HOUSENO** | **ROADNAME** | **CITY** | **ZIPCODE** |
| 1001 | Rick | 237 | ABC Main Road | Texas | 77001 |

* Number of tables = 1
* In the object model, there are **two** Java classes - Customer and Address. However, the data of the classes are being pushed into only **one** table(Customer) of the database.
* The **Granularity** problem comes when the number of classes mapping to the number of tables in the database **do not match**.

* 

* In the Object-Oriented paradigm, the parent-child or base-derived class relationships are implemented using **inheritance.**Consider the object model for the types of Customers in a retail application. The Customer can be a Corporate or Retail Customer.
* The object model is represented with inheritance as shown below.
* 

In the data model, since inheritance is not possible we have to create two tables even though columns are getting repeated.

* RETAILCUSTOMER - with columns CUSTOMERNAME, CUSTOMERADDRESS and CREDITCARDNO.
* CORPORATECUSTOMER - with CUSTOMERNAME, CUSTOMERADDRESS, CONTACTNAME and CREDITLIMIT.

SQL does not support the concept of super-tables and sub-tables. The databases do not allow to create a table that inherits certain columns from its parent.

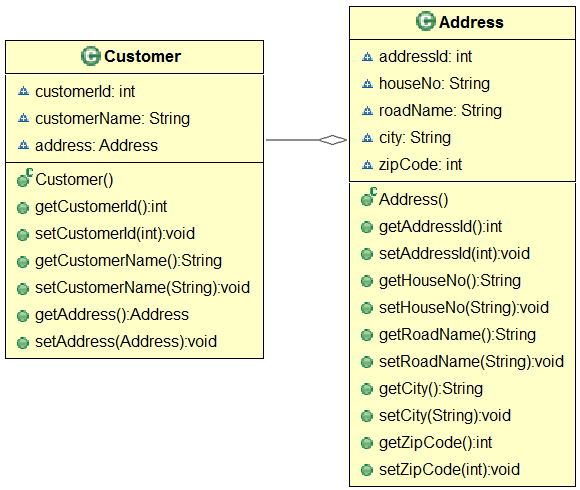
| RETAILCUSTOMER | | |
| --- | --- | --- |
| **CUSTOMERNAME** | **CUSTOMERADDRESS** | **CREDITCARDNO** |
| Rick | ABC Main Road, Texas | 5555555555554444 |

| CORPORATECUSTOMER | | | |
| --- | --- | --- | --- |
| **CUSTOMERNAME** | **CUSTOMERADDRESS** | **CONTACTNAME** | **CREDITLIMIT** |
| Sam | XYZ Main Road, Texas | Samuel | 5000 |

The **Inheritance or Subtype paradigm mismatch** occurs because inheritance is not defined explicitly in any standardized RDBMS. Most of the RDBMS does not define anything similar to inheritance.

**Association**relationship in Object-Oriented languages like Java is called has-a relationship (a reference of one class is created as an instance variable in the other). For example, a Customer **has-a** Address.

Let us re-look at the object model of Customer and Address. Each Customer has an Address as shown below.



| CUSTOMER | | |
| --- | --- | --- |
| **CUSTOMERID** | **CUSTOMERNAME** | **ADDRESSID** |
| 1001 | Rick | 5001 |

addressId is Foreign Key in the Customer table

| ADDRESS | | | | |
| --- | --- | --- | --- | --- |
| **ADDRESSID** | **HOUSENO** | **ROADNAME** | **CITY** | **ZIPCODE** |
| 5001 | 237 | ABC Main Road | Texas | 77001 |

addressId is Primary Key in the Address table

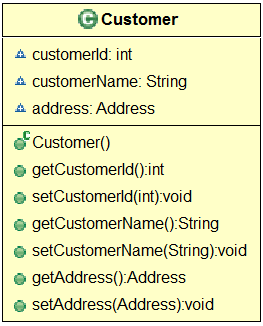
In the relational model, the association between tables is represented using the primary key and foreign keys.

In the case of a customer having multiple addresses, the relational model supports one-to-one, one-to-many, and many-to-many relationship mappings.

The **Association paradigm mismatch** exists because Java represents associations (has-a relationship) using object references and in RDBMS association is by a **foreign key** column.

In Object-Oriented terminologies, **Identity**is a feature that determines the equality of two comparable units.

Consider the object model of Customer depicted below:



However, SQL gives exactly one notion of 'sameness': the primary key. The equality of two rows of a table is determined by checking the primary key value. The customers are identified by primary key values (1001 and 1002). Two customers with the same customerId are treated as equal in SQL.

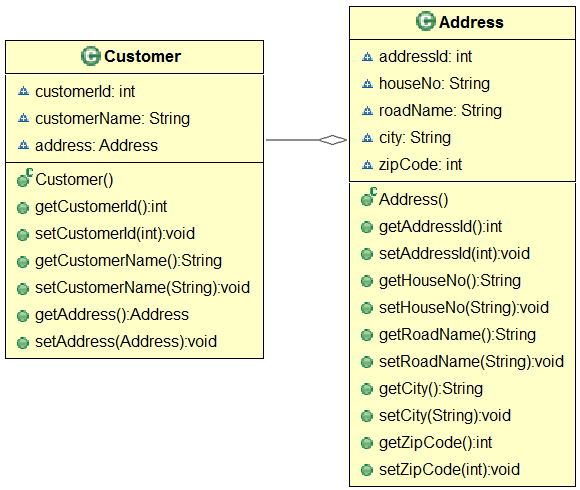
| CUSTOMER | | |
| --- | --- | --- |
| **CUSTOMERID** | **CUSTOMERNAME** | **ADDRESSID** |
| 1001 | Rick | 3001 |
| 1002 | Sam | 3001 |

CustomerId is the primary key

The**Identity paradigm mismatch** occurs because Java defines similarity using == and equals() whereas RDBMS uses a primary key.

**Data Navigation**refers to the procedure of traversing through the Java object network.

Let us take the same example of an object model of Customer and Address. If we need to retrieve the zipCode of a customer, which is available in Address, in Java, data is accessed by navigating through the object network. For example: **customer.getAddress().getZipCode();**



| CUSTOMER | | |
| --- | --- | --- |
| **CUSTOMERID** | **CUSTOMERNAME** | **ADDRESSID** |
| 1001 | Rick | 5001 |

addressId is Foreign Key in the Customer table

| ADDRESS | | | | |
| --- | --- | --- | --- | --- |
| **ADDRESSID** | **HOUSENO** | **ROADNAME** | **CITY** | **ZIPCODE** |
| 5001 | 237 | ABC Main Road | Texas | 77001 |

addressId is Primary Key in the Address table

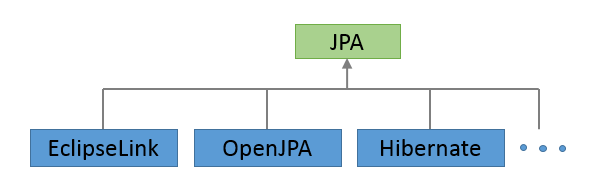
RDBMS uses **SQL JOINS** to navigate from one database table to another. To retrieve the zipCode of the customer Rick, the information is available in two tables- CUSTOMER and ADDRESS, a join operation has to be done as below:

**select add.zipCode from Address add, Customer cust where add.addressId=cust.addressId and cust.customerName='Rick'**

The **Data Navigation paradigm mismatch** occurs due to the dissimilarities in the ways we access the data in Java using objects and in an RDBMS.

Resolving Object-Relational Impedance Mismatch is one of the key challenges in data persistence. **Object Relational Mapping**helps to achieve the same.

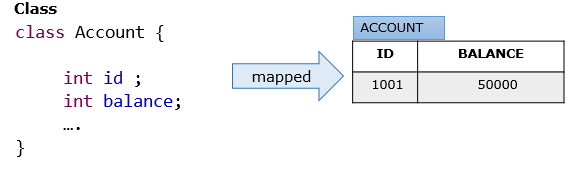
We use **ORM** mainly for the following reasons:

* ORM resolves the Object-Relational Paradigm mismatch.
* The lower level interaction with the database is handled by the ORM. Framing and executing the database dependent queries is taken care of by the ORM framework.
* ORM helps the developer to get rid of "messy SQL". The developer need not waste time in writing the plumbing code.
* ORM allows the developer to concentrate on the business logic and work with the object model.
* ORM is database independent. All database vendors give the necessary support for ORM. Hence the application code becomes portable without worrying about the underlying database.
* ORM frameworks are preferred as an elegant persistence solution for Enterprise applications. There are a lot of ORM frameworks from different vendors available in the market.
* Many third-party persistence frameworks like Hibernate, EclipseLink, are available in the market. These frameworks helped the developers to achieve Object Relational Mapping and perform database operations in the object-oriented approach. But it became challenging to the port application from one ORM framework to another, as every framework addressed the Object-Relational Impedance mismatch in its own way.
* In 2006, **Java Persistence API (JPA)** was released by Java Community Process, to standardize the persistence process. JPA incorporated many features from the existing frameworks like Hibernate and TopLink Essentials.
* JPA became the standard specification for ORM in Java. As the name indicates, JPA is just a specification (having a set of interfaces), which provides the standards and specifications to be followed while mapping the Java objects to the database tables.
* Each vendor, who provides an ORM framework, implements JPA. A few of the ORM frameworks are EclipseLink, OpenJPA, and Hibernate. Any of these ORM frameworks can be used to connect a Java application to the database.
* 
* **Note:**In this course, we will study in details about the **Hibernate framework.**

Let us understand Object Relational Mapping concepts in detail.

In Object Relational Mapping, the Java entity classes are mapped to relational database tables. In this technique the entity classes are mapped to the database tables, the data members are mapped to the database table columns, objects of Java entity classes are mapped to the records of the database tables.

Consider a Java Entity class "Account" with a mapping database table "Account" as below:

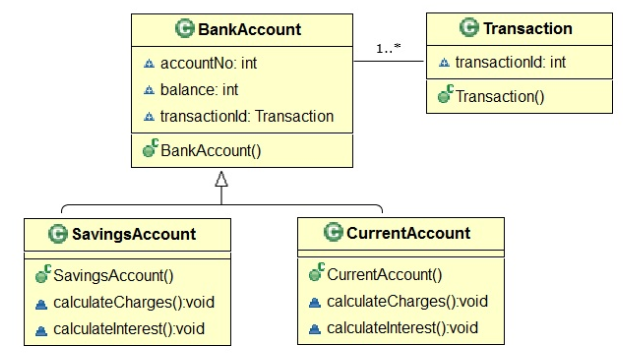


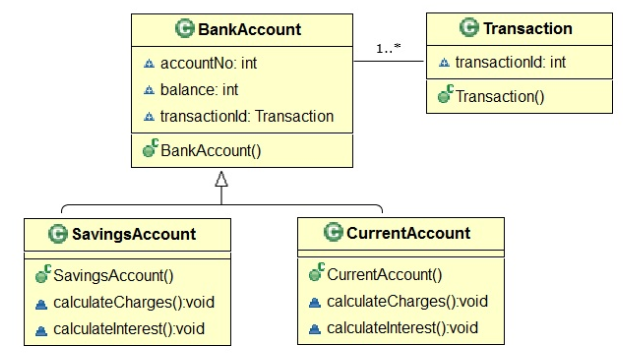
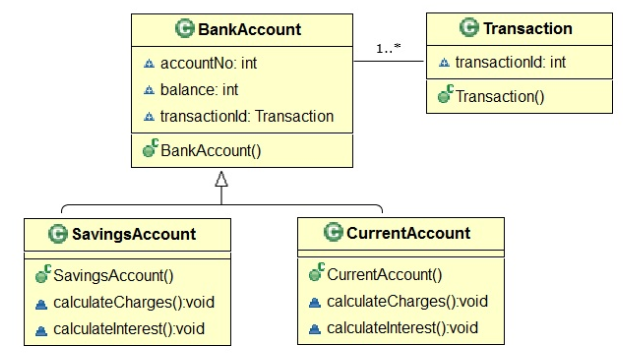
In the provided example,

* Java Entity class Account is mapped to a table Account
* Data members id and balance of Account class are mapped to the table columns Id and Balance
* An instance of Account class with values 1001(id) and 50000(balance) is a record(row) in a table

**Problem Statement:**

Consider the object and data model of a banking application shown below.





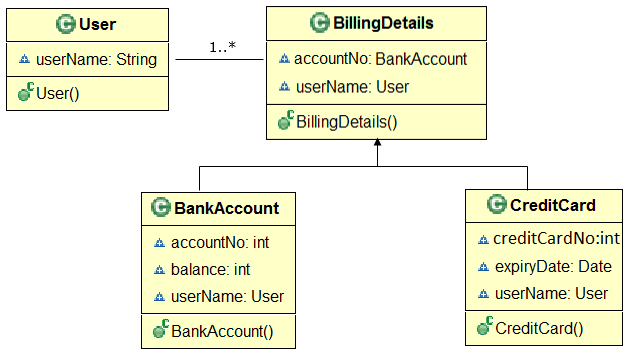
Identify the Object-Relational Impedance Mismatches that can occur in the below relationships:

* Account and Transaction
* Account, CurrentAccount and Savings Account

**Summary**: In this exercise, you learned to identify the Object-Relational Impedance Mismatch.

**Problem Statement:**

Consider the below object model.



**Problem Statement:**

Below is a database table with some sample data. Identify the ORM elements and create the Java entity class for this table.

                                          EMP\_DETAILS

|  |  |  |
| --- | --- | --- |
| ID | Name | Salary |
| 1001 | Matt | 20000 |
| 1002 | Lisa | 30000 |

Summary: In this exercise, you learned to code the entity class based on database information provided.

 this module we have learned:

* Different challenges while mapping an Object model to a relational model
* Object Relational Mapping(ORM)
* How ORM is helping in Object-Relational impedance mismatch.

Hibernate is a pure Java Persistence Framework that supports Object Relational Mapping. The main goal of this framework is to release the programmers from the common data persistence related works. It is an open-source framework.

* Hibernate provides an implementation for JPA Specification.
* Hibernate is a powerful ORM solution that maps user-defined Java classes to DB tables.
* Hibernate has a strong query language which is called Hibernate Query Language. It supports native SQL as well.
* Hibernate reduces the number of lines in the code by keeping object-table mapping itself and gives the result to an application as Java objects. It ensures the programmer doesn't have to manually handle persistent data, this way reducing the time of development and cost of maintenance.
* Hibernate uses SQL based schema for mapping object model to the relational model.
* Hibernate allows developers to emphasize the domain model and not on the persistence plumbing (e.g.: connection management).

**History of Hibernate framework:**

* Hibernate Framework was started by Gavin King and his colleagues in the year 2001 with the aim of offering better persistence support than those offered by the JEE component Enterprise Java Bean 2.  It tried to simplify the complexities and supplement certain missing features of EJB2.
* JBoss later took the lead developers of Hibernate for the further development of this framework.
* The subsequent versions provided various new features and enhancements for better performance.
* The current version is 5.0.2, released in Sep 2015.

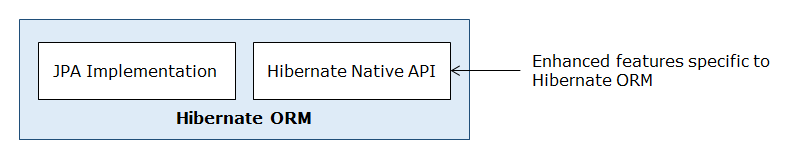
**Features of Hibernate:**

* Object Relational Mapping: Hibernate, being an ORM framework aims to resolve the Object-Relational Impedance Mismatches proving itself as an effective data persistence medium.
* Scalability and Reliability: Hibernate works well in the client-server based environment and delivers a scalable architecture. Hibernate provides good stability and quality, hence it is reliable.
* Extensible: Hibernate is highly configurable and extensible.
* High Performance: Hibernate has high performance due to various features like multiple fetch strategies, optimistic locking with automatic versioning and time stamping, caching, etc.
* Idiomatic Persistence: Hibernate enables the development of persistent classes that follows object-oriented idioms like inheritance, composition, polymorphism, association, and the Java collections framework.

**Benefits of Hibernate:**

* Lightweight: Hibernate implements ORM using simple POJO classes.
* Open Source: Hibernate is freely available and may be redistributed and modified
* Vendor Independent: Hibernate, or in general JPA, prevents writing code according to the database vendor. Hence it is vendor-independent.
* Non-Invasive: Hibernate does not force the developer to extend or implement any class or interface.

**Note :**  
POJO, or Plain Old Java Object, is nothing but a normal Java class and does not implement any special interfaces of any of the Java frameworks.

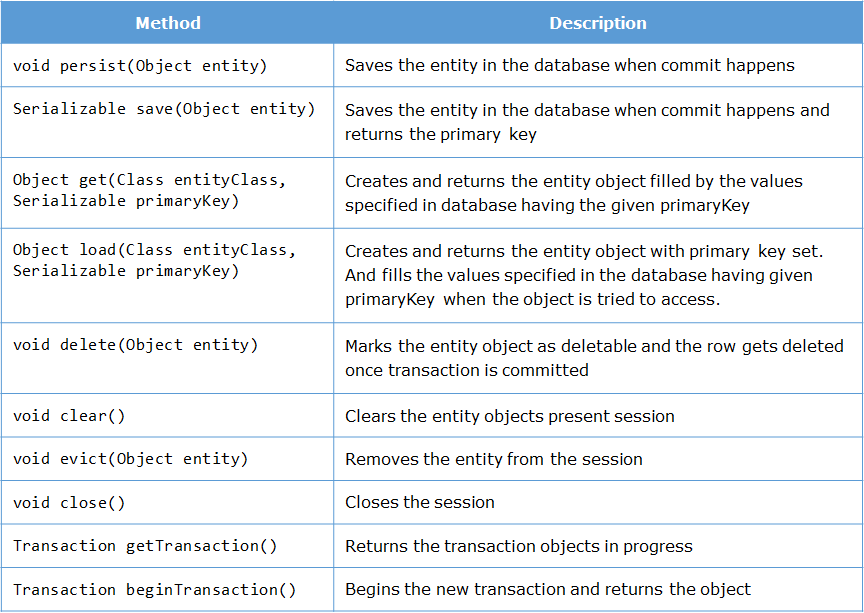


Now lets us see some important interfaces and classes of Hibernate API.

Some important interfaces and classes of Hibernate API that you are going to learn in this course are as follows:

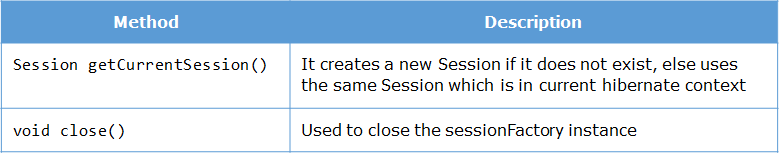
**Session**

The **org.hibernate.Session** interface represents the connection between the application and database. It provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query, and Criteria. Few methods of this interface are as follows:



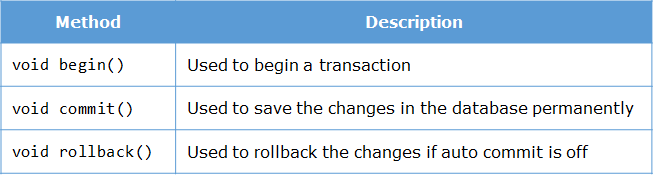
**SessionFactory**

The **org.hibernate.SessionFactory** is a factory class to get objects of Session. The methods available here are:



**Transaction**

The **org.hibernate.Transaction** interface provides methods for transaction management. Few methods of this interface are as follows:



Now let us learn how to perform CRUD operations using Hibernate.