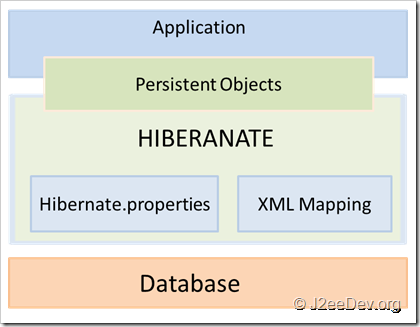
**Hibernate**

* Hibernate is an open source, lightweight object-relational mapping solution.
* The main feature of Hibernate is its support for object-based modeling, which allows it to provide a transparent mechanism for persistence.
* It uses XML to map a database to an application and supports fine-grained objects. The current version of Hibernate is 3.x, and it supports Java annotations
* Hibernate includes a very powerful query language called Hibernate Query Language, or HQL.

**Hibernate Query Language (HQL):**

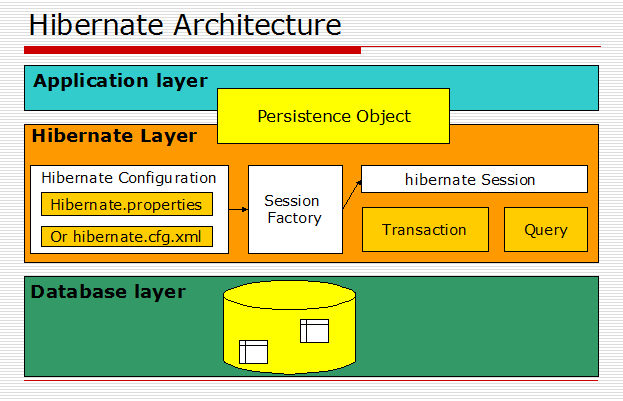
* HQL is completely object-oriented, enabling you to leverage the complete strength of the object-oriented pillars of inheritance, polymorphism and association. Also in terms of coding, it is similar to SQL.
* HQL is database independent.
* HQL queries are case insensitive, except for the names of the Java classes and properties being used.
* HQL returns query results as objects that can be directly accessed and manipulated by the programmer.
* HQL also supports many advanced features of pagination and dynamic profiling that SQL has never supported.
* HQL does not require any explicit joins when working with multiple tables.

**High Level architecture diagram**



* Itself opens connection to database
* Converts HQL (Hibernate Query Language) statements to database specific statement.
* Receives result set
* Then performs mapping of this database specific data to Java objects which are directly used by Java application.

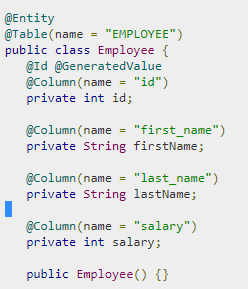
**Low level architecture diagram**



* Load the Hibernate configuration file and create configuration object. It will automatically load all hbm mapping files.
* Create session factory from configuration object.
* Get one session from this session factory.
* Create HQL query.
* Execute query to get list containing Java objects.

**Details**

* **Configuration object**
  + It aids in establishing a connection to a particular relational database.
  + The configuration file (hibernate.cfg.xml) should know which mapping file it needs to refer to.
  + The Configuration object is the first Hibernate object you create in any Hibernate application.
  + It is usually **created only once** during application initialization
  + At runtime, Hibernate reads the mapping file and then uses it to build a dynamic Java class corresponding to that table of the database.
* **org.hibernate.SessionFactory**
  + Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated
  + The SessionFactory is a **heavyweight** object; it is usually created during application start up and kept for later use.
  + You would need **one SessionFactory object per database** using a separate configuration file
  + It is basically used to obtain a session instance, and can be seen as an analogue to the connection pooling mechanism.
  + This is thread safe, as all the application threads can use a single SessionFactory (as long as Hibernate uses a single database).
  + This interface is configured through the configuration file, which determines the mapping file to be loaded.
* **org.hibernate.Session**
  + A Session is used to get a physical connection with a database.
  + The Session object is **lightweight** and designed to be instantiated each time an interaction is needed with the database.
  + It provides a single thread that determines the conversation between the application and the database.
  + This is analogous to a specific (single) connection.
  + It is not thread safe.
  + Persistent objects are saved and retrieved through a Session object.
  + The main function of the Session is to offer, create, read, and delete operations for instances of mapped entity classes.
  + Instances may exist in one of the following three states at a given point in time −
    - **transient** − A new instance of a persistent class, which is **not associated with a Session** and has no representation in the database and no identifier value is considered transient by Hibernate.
    - **persistent** − You can make a transient instance persistent by **associating it with a Session**. A persistent instance has a representation in the database, an identifier value and is associated with a Session.
    - **detached** − Once we close the Hibernate Session, the persistent instance will become a detached instance.
* **org.hibernate.Transaction**
  + A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality.
  + It provides a single-thread object that spans through the application and determines an atomic unit of work.
  + It basically abstracts JDBC, JTA, and CORBA transactions.
  + This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.
* **org.hibernate.Query** 
  + It is used to perform a query, either in HQL or in the SQL dialect of the underlying database.
  + A Query instance is lightweight, and it is important to note that it cannot be used outside the session through which it was created.
* **Persistence classes**
  + Java classes whose objects or instances will be stored in database tables are called persistent classes in Hibernate.
  + Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model
* **Mappings**
  + Mapping of collections
    - Hibernate can persist instances of java.util.Map, java.util.Set, java.util.SortedMap, java.util.SortedSet, java.util.List, and any array of persistent entities or values.
  + Mapping of associations between entity classes
    - Many-to-One: Mapping many-to-one relationship using Hibernate
    - One-to-One: Mapping one-to-one relationship using Hibernate
    - One-to-Many: Mapping one-to-many relationship using Hibernate
    - Many-to-Many: Mapping many-to-many relationship using Hibernate
  + Component Mappings.
    - If the referred class does not have its own life cycle and completely depends on the life cycle of the owning entity class, then the referred class hence therefore is called as the Component class
* **Annotations:** 
  + **Basic requirement: JDK 5, Hibernate 3.0**
  + @Entity Annotation
    - The EJB 3 standard annotations are contained in the javax.persistencepackage, so we import this package as the first step.
    - Second, we used the @Entity annotation to the Employee class, which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.
  + @Table Annotation
    - The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.
    - The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table.
  + @Id and @GeneratedValue Annotations
    - Each entity bean will have a primary key, which you annotate on the class with the @Id annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.
    - By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the @GeneratedValue annotation.
    - Letting Hibernate determine which generator type to use makes your code portable between different databases.
  + @Column Annotation
    - The @Column annotation is used to specify the details of the column to which a field or property will be mapped.
    - You can use column annotation with the following most commonly used attributes −
      * name attribute permits the name of the column to be explicitly specified.
      * length attribute permits the size of the column used to map a value particularly for a String value.
      * nullable attribute permits the column to be marked NOT NULL when the schema is generated.
      * unique attribute permits the column to be marked as containing only unique values.



**Advantages of Hibernate over JDBC**

* Hibernate is data base independent, same code will work for all data bases like ORACLE, MySQL, SQLServer etc.  
  In case of JDBC query must be data base specific.
* As Hibernate is set of Objects, you don't need to learn SQL language.  
  You can treat TABLE as an Object.   
  In case of JDBC you need to learn SQL.
* Don't need Query tuning in case of Hibernate. If you use Criteria Quires in Hibernate then hibernate automatically tuned your query and return best result with performance.  
  In case of JDBC you need to tune your queries.
* You will get benefit of Cache. Hibernate support two level of cache. First level and 2nd level. So you can store your data into Cache for better performance.  
  In case of JDBC you need to implement your java cache.
* Hibernate supports Query cache and It will provide the statistics about your query and database status.  
  JDBC Not provides any statistics.
* Development fast in case of Hibernate because you don't need to write queries.
* No need to create any connection pool in case of Hibernate. You can use c3p0.  
  In case of JDBC you need to write your own connection pool.
* In the xml file you can see all the relations between tables in case of Hibernate. Easy readability.
* You can load your objects on start up using lazy=false in case of Hibernate.  
  JDBC don't have such support.
* Hibernate Supports automatic versioning of rows but JDBC do not.
* In JDBC all exceptions are checked exceptions, so we must write code in try, catch and throws, but in hibernate we only have unchecked exceptions, so no need to write try, catch, or no need to write throws.  Actually in hibernate we have the translator which converts checked to unchecked

**Advantages of Hibernate over other ORM’s**

* Hibernate makes object-relational mapping simple by mapping the metadata in an XML file that defines the table in the database that needs to be mapped to a particular class.
* In other persistence frameworks, you need to modify the application class to achieve object-relational mapping; this is not necessary in Hibernate.
* With Hibernate, you needn't worry about database changes, as manual changes in the SQL script files are avoided.
* If you ever need to change the database your application uses, that can be easily accommodated by altering the **dialect** property in the configuration file.
* Hibernate gives you the complete power of SQL, something that was never offered by earlier commercial ORM frameworks.
* Hibernate also supports many databases, including MySQL, Oracle, Sybase, Derby, and PostgreSQL, and works well with plain old Java object (POJO)-based models, too.
* **Hibernate generates JDBC code based on the underlying database chosen**, and so saves you the trouble of writing JDBC code. It also **supports connection pooling**.
* The APIs that are used by Hibernate are very simple and easy to learn.
* Developers with very little knowledge of SQL can make use of Hibernate, as it lessens the burden of writing SQL queries.

### When to use Hibernate?

* Hibernate is best used to leverage end-to-end OR mapping.
* It provides a complete ORM solution, but leaves you control over queries.
* Hibernate is an ideal solution for situations where you have complete control over both the application and the database design.
* In such cases you may modify the application to suit the database, or vice versa. In these cases you could use Hibernate to build a fully object-relational application.
* Hibernate is the best option for object-oriented programmers who are less familiar with SQL.

**When not to use Hibernate?**

* If you **schema is very simple** (less than 8-10 tables) and does not have very complex relationship among them, then hibernate usage will be slightly overkill for this scenario.
* If the **size of data** you are trying work on is very large (in the order of mn/bn), there too usage of hibernate usage is discouraged. Good old JDBC is better over here
* If you application uses a lot of **static data** then **caching** would be a good option and hibernate support caching transparently.