HIBERNATE

Mr. Ashok

Facebook group: ASHOK IT SCHOOL
Email: ashok.javatraining@gmail.com

Introduction

Almost all applications require persistent data. Persistence is one of the fundamental concepts in application development. If an information system did not preserve data when it was powered off, the system would be of little practical use hence we need to persist the data

Persistence

The process of storing and maintaining the data for long time is called Persistence.

Persistence store

To store and maintain data for long time we need Persistence stores. Below two things we can use as Persistence stores

1. File System
2. Database

Database and File System are two methods used to store, retrieve, manage and manipulate data. Both systems can be used to allow the user to work with data in a similar way. A File System is a collection of raw data files stored in the hard-drive, whereas a database is intended for easily organizing, storing and retrieving large amounts of data. In other words, a database holds a bundle of organized data (typically in a digital form) for one or more users. Databases, often-abbreviated DB, are classified according to their content, such as document-text, bibliographic and statistical. It should be noted that, even in a database, data are eventually (physically) stored in some sort of files (.dbf).

File system

As mentioned above, in a typical File System electronic data are directly stored in a set of files. If only one table is stored in a file, it is called a flat file. They contain values in each row separated with a special delimiter like commas. In order to query some random data, first it is required to parse each row and load it to an array at run time, but for this file should be read sequentially (because, there is no control mechanism in files); therefore it is quite inefficient and time consuming. The burden of locating the necessary file, going through the records (line by line), checking for the existence of a certain data and remembering what files/records to edit are on the user. The user either has to perform each task manually or has to write a script that does them automatically with the help of the file management capabilities of the operating system. Because of these reasons, File Systems are easily vulnerable to serious issues like inconsistency, inability to maintain concurrency, data isolation, threats on integrity and lack of security.

Using serialization

Java has a built-in persistence mechanism: Serialization provides the ability to write a snapshot of a network of objects (the state of the application) to a byte stream, which may then be persisted to a file or database. Serialization is also used by Java's Remote Method Invocation (RMI) to achieve pass-by value semantics for complex objects. Another use of serialization is to replicate application state across nodes in a cluster of machines.

Why not use serialization for the persistence layer? Unfortunately, a serialized network of interconnected objects can only be accessed as a whole; it is impossible to retrieve any data from the stream without deserializing the entire stream. Thus, the resulting byte stream must be considered unsuitable for arbitrary search or aggregation of large datasets. It is not even possible to access or update a single object or subset of objects independently. Loading and overwriting an entire object network in each transaction is no option for systems designed to support high concurrency.

Given current technology, serialization is inadequate as a persistence mechanism for high concurrency web and enterprise applications. It has a particular niche as a suitable persistence mechanism for desktop applications.

Some enterprise applications will use below file formats to store the data (Some companies will send/receive the data in below file formats).

1. Fixed Width Field Size Text File
2. CSV (Comma Separated File)
3. XML files
4. Excel files

Fixed Width Field Size Text File

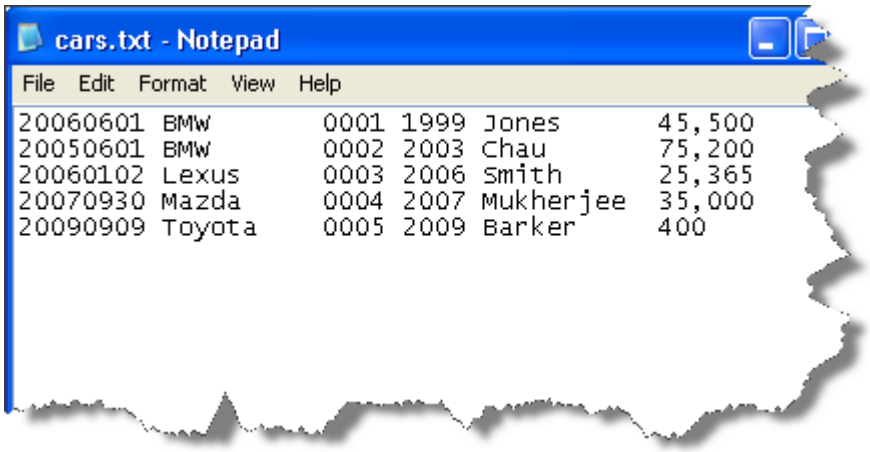
A fixed width text file is a file that has a specific format, which allows for the saving of textual information/data in an organized fashion.

Fixed width text files are special cases of text files where the format is specified by column widths, pad character and left/right alignment. Column widths are measured in units of characters. For example, if you have data in a text file where the first column always has exactly 10 characters, and the second column has exactly 5, the third has exactly 12 (and so on), this would be categorized as a fixed width text file.

To be very specific, if a text file follows the rules below, it is a fixed width text file:

- Each row (paragraph) contains one complete record of information.
- Each row contains one or many pieces of data (also referred to as columns or fields).
- Each data column has a defined width specified as a number of characters that is always the same for all rows.
- The data within each column is padded with spaces (or any character you specify) if it does not completely use all the characters allotted to it (empty space).
- Each piece of data can be left or right aligned, meaning the pad characters can occur on either side.
- Each column must consistently use the same number of characters, same pad character and same alignment (left/right).

Data in a fixed-width text file is arranged in rows and columns, with one entry per row. Each column has a fixed width, specified in characters, which determines the maximum amount of data it can contain. No delimiters are used to separate the fields in the file. Instead, smaller quantities of data are padded with spaces to fill the allotted space, such that the start of a given column can always be specified as an offset from the beginning of a line. The following file snippet illustrates characteristics common to many flat files. It contains information about cars and their owners, but there are no headings to the columns in the file and no information about the meaning of the data. In addition, the data has been laid out with a single space between each column, for readability:



CSV File Format

CSV is a simple file format used to store tabular data, such as a spreadsheet or database. Files in the CSV format can be imported to and exported from programs that store data in tables, such as Microsoft Excel or OpenOffice Calc.

CSV stands for "comma-separated values". Its data fields are most often separated, or delimited, by a comma. For example, let us say you had a spreadsheet containing the following data.

Name	Class	Dorm	Room	GPA
Sally Whittaker	2018	McCarren House	312	3.75
Belinda Jameson	2017	Cushing House	148	3.52
Jeff Smith	2018	Prescott House	17-D	3.20
Sandy Allen	2019	Oliver House	108	3.48

The above data could be represented in a CSV-formatted file as follows:

```
Sally Whittaker,2018,McCarren House,312,3.75
Belinda Jameson,2017,Cushing House,148,3.52
Jeff Smith,2018,Prescott House,17-D,3.20
Sandy Allen,2019,Oliver House,108,3.48
```

CSV is a common data exchange format that is widely supported by consumer, business, and scientific applications. Among its most common uses is moving tabular data between programs that natively operate on incompatible (often proprietary or undocumented) formats. This works despite lack of adherence because so many programs support variations on the CSV format for data import.

For example, a user may need to transfer information from a database program that stores data in a proprietary format, to a spreadsheet that uses a completely different format. The database program most likely can export its data as "CSV"; the spreadsheet program can then import the exported CSV file.

XML files

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is defined in the XML 1.0 Specification produced by the W3C, and several other related specifications, all gratis open standards.

The design goals of XML emphasize simplicity, generality, and usability over the Internet. It is a textual data format with strong support via Unicode for the languages of the world. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures, for example in web services.

```
<CustomerOrderMessage>
        <OrderNumber>12345</OrderNumber>
        <Quantity>10</Quantity>
</CustomerOrderMessage>
```

- The XML above has only two byte of information the rest of it is metadata. Using too much metadata requires more processor power and increase network traffic (Which is great news for hardware vendors, but bad news for the people who have to pay for it).
- The flexibility of XML can lead to unnecessary complexity and it can make it hard for developers to understand, therefore lead to mistakes and development time and cost increases.
- In some of the cases, it can be necessary to convert XML into simplified format so it can be loaded into the database.
- This XML can be loaded by most of the ETL tools.

XML persistence is a variation on the serialization theme; this approach addresses some of the limitations of byte-stream serialization by allowing easy access to the data through a standardized tool interface. However, managing data in XML would expose you to an object/hierarchical mismatch. Furthermore, there is no additional benefit from the XML itself, because it is just another text file format and has no inherent capabilities for data management. You can use stored procedures (even writing them in Java, sometimes) and move the problem into the database tier. So-called object-relational databases have been marketed as a solution, but they offer only a more sophisticated datatype system providing only half the solution to our problems (and further muddling terminology). We are sure there are plenty of other examples, but none of them is likely to become popular in the immediate future.

In addition, as a Conclusion here are some basic XML design tips:

- ✓ Use XML when it is necessary
- ✓ Too much metadata is a bad thing
- ✓ Keep tags short
- ✓ Keep it simple and clean
- ✓ Check ETL documentation and design XML in such a way so it can be loaded without conversion

Excel Files

In Excel files we can store the data in the form of rows and columns but the problem here is Excel files are platform dependent. Excel works only in Windows operating system.

Database

A Database may contain different levels of abstraction in its architecture. Typically, the three levels: external, conceptual and internal make up the database architecture. External level defines how the users view the data. A single database can have multiple views. The internal level defines how the data is physically stored. The conceptual level is the communication medium between internal and external levels. It provides a unique view of the database regardless of how it is stored or viewed. There are several types of databases such as Analytical databases, Data warehouses and Distributed databases. Databases (more correctly, relational databases) are made up of tables, and they contain rows and columns, much like spreadsheets in Excel. Each column corresponds to an attribute while each row represents a single record. For example, in a database, which stores employee information of a company, the columns could contain employee name, employee Id and salary, while a single row represents a single employee. Most databases come with a Database Management System (DBMS) that makes it very easy to create/manage/organize data.

Difference between File system and Database

As a summary, in a File System, files are used to store data while, a database is a collection of organized data. Although File System and databases are two ways of managing data, databases clearly have many advantages over File Systems. Typically when using a File System, most tasks such as storage, retrieval and search are done manually (even though most operating systems provide graphical interfaces to make these tasks easier) and it is quite tedious whereas when using a database, the inbuilt DBMS will provide automated methods to complete these tasks. Because of this reason, using a File System will lead to problems like data integrity, data inconsistency and data security, but these problems could be avoided by using a database. Unlike a File System, databases are efficient because reading line by line is not required, and certain control mechanisms are in place.

Database Management Systems

A Database is a collection of records. Database management systems are designed as the means of managing all the records. Database Management is a software system that uses a standard method and running queries with some of them designed for the oversight and proper control of databases.

Formally, a "database" refers to a set of related data and the way it is organized. Access to this data is usually provided by a "database management system" (DBMS) consisting of an integrated set of computer software that allows users to interact with one or more databases and provides access to all of the data contained in the database (although restrictions may exist that limit access to particular data). The DBMS provides various functions that allow entry, storage and retrieval of large quantities of information and provides ways to manage how that information is organized.

Because of the close relationship between them, the term "database" is often used casually to refer to both a database and the DBMS used to manipulate it.

Outside the world of information technology, the term database is often used to refer to any collection of related data (such as a spreadsheet or a card index). Existing DBMSs provide various functions that allow management of a database and its data, which can be classified into four main functional groups:

DDL – Creation, modification and removal of definitions that define the organization of the data.
DML – Insertion, modification, and deletion of the actual data.
DQL – Providing information in a form directly usable or for further processing by other applications. The retrieved data may be made available in a form the same as it is stored in the database or in a new form obtained by altering or combining existing data from the database.

Administration – Registering and monitoring users, enforcing data security, monitoring performance, maintaining data integrity, dealing with concurrency control, and recovering information that has been corrupted by some event such as an unexpected system failure.

Both a database and its DBMS conform to the principles of a particular database model. "Database system" refers collectively to the database model, database management system, and database.

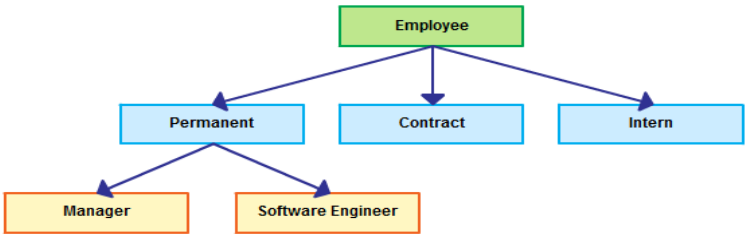
Types of Database Management Systems

- Hierarchical databases.
- Network databases.
- Relational databases.
- Object-oriented databases
- No Sql databases

Hierarchical Databases (DBMS)

In the Hierarchical Database Model, records contain information about their groups of parent/child relationships, just like as a tree structure. The structure implies that a record can have also a repeating information. In this structure, Data follows a series of records; It is a set of field values attached to it. It collects all records together as a record type. These record types are the equivalent of tables in the relational model, and with the individual records being the equivalent of rows. To create links between these record types, the hierarchical model uses these type Relationships.

The Hierarchical Data Model is a way of organizing a database with multiple one to many relationships. The structure is based on the rule that one parent can have many children but children are allowed only one parent. This structure allows information to be repeated through the parent child relations created by IBM and was implemented mainly in their Information Management System. (IMF), the precursor to the DBMS.



Advantage

The model allows easy addition and deletion of new information. Data at the top of the Hierarchy is very fast to access. It was very easy to work with the model because it worked well with linear type data storage such as tapes. The model relates very well to natural hierarchies such as assembly plants and employee organization in corporations. It relates well to anything that works through a one to many relationships. For example; there is a president with many managers below them, and those managers have many employees below them, but each employee has only one manager.

Disadvantage

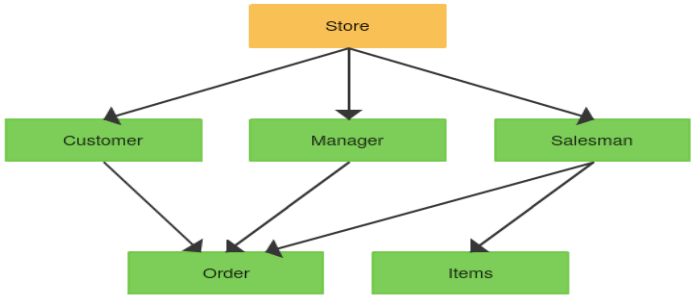
This type of database structure is that each child in the tree may have only one parent, and relationships or linkages between children are not permitted, even if they make sense from a logical standpoint. Hierarchical databases are so in their design. it can adding a new field or record requires that the entire database be redefined.

Network Databases

The network model is a database model conceived as a flexible way of representing objects and their relationships. Its distinguishing feature is that the schema, viewed as a graph in which object types are nodes and relationship types are arcs, is not restricted to being a hierarchy or lattice.

The network model replaces the hierarchical model with a graph thus allowing more general connections among the nodes. The main difference of the network model from the hierarchical model is its ability to handle many to many relationships. In other

words it allow a record to have more than one parent.



Database design done using network model. In the network model a node can have multiple parent nodes.

Advantages

- Conceptual simplicity-Just like the hierarchical model, the network model is also conceptually simple and easy to design.
- Capability to handle more relationship types-The network model can handle the one to many and many to many relationships which is real help in modeling the real life situations.
- Ease of data access-The data access is easier and flexible than the hierarchical model.
- Data integrity- The network model does not allow a member to exist without an owner.
- Data independence- The network model is better than the hierarchical model in isolating the programs from the complex physical storage details.
- Database standards

Dis-Advantages

- System complexity- All the records are maintained using pointers and hence the whole database structure becomes very complex.
- Operational Anomalies- The insertion, deletion and updating operations of any record require large number of pointers adjustments.
- Absence of structural independence-structural changes to the database is very difficult.

Relational Databases

The relational model, first proposed in 1970 by Edgar F. Codd, departed from this tradition by insisting that applications should search for data by content, rather than by following links. The relational model employs sets of ledger-style tables, each used for a different type of entity. Only in the mid-1980s did computing hardware become powerful enough to allow the wide deployment of relational systems (DBMSs plus applications). By the early 1990s, however, relational systems dominated in all large-scale data processing applications, and as of 2015, they remain dominant: IBM DB2, Oracle, MySQL, and Microsoft SQL Server are the top DBMS. The dominant database language, standardized SQL for the relational model, has influenced database languages for other data models.

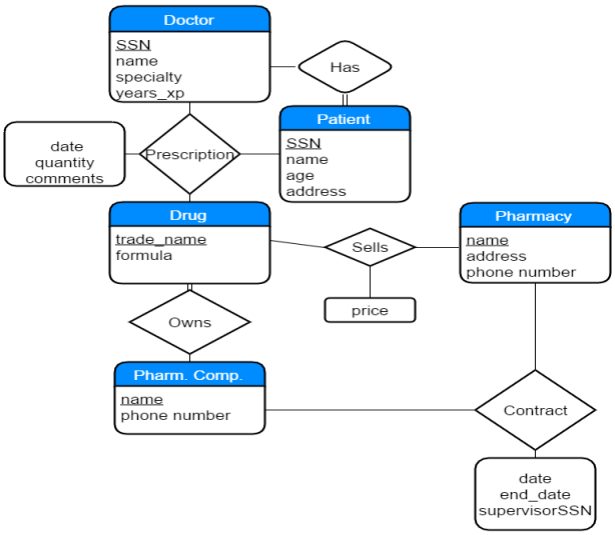
In relational databases, the relationship between data files is relational. Hierarchical and network databases require the user to pass a hierarchy in order to access needed data. These databases connect to the data in different files by using common data numbers or a key field. Data in relational databases is stored in different access control tables, each having a key field that mainly identifies each row. In the relational databases are more reliable than either the hierarchical or the network database structures. In relational databases, tables or files filled up with data are called relations (tuples) designates a row or record, and columns are referred to as attributes or fields.

Relational databases work on each table has a key field that uniquely indicates each row, and that these key fields can be used to connect one table of data to another.

Properties of Relational Tables

In the relational database, we have to follow some properties, which given below.

- It's Values are Atomic
- In Each Row is alone.
- Column Values are of the same thing.
- Columns is undistinguished.
- Sequence of Rows is Insignificant.
- Each Column has a common Name.



Relational database management systems are not specific to Java, nor is a relational database specific to a particular application. This important principle is known as data independence. In other words, and we can't stress this important fact enough, data lives longer than any application does. Relational technology provides a way of sharing data among different applications, or among different technologies that form parts of the same application, (the transactional engine and the reporting engine, for example). Relational technology is a common denominator of many disparate systems and technology platforms. Hence, the relational data model is often the common enterprise-wide representation of business entities.

How is it different from a normal DBMS?

- DBMS stores data as files whereas RDBMS stores data in a tabular arrangement.
- RDBMS allows for normalization of data.
- RDBMS maintains a relation between the data stored in its tables. A normal DBMS does not provide any such link. It blankly stores data is its files.
- Structured approach of RDBMS supports a distributed database unlike a normal database management system.

Features of RDBMS

- The system caters to a wide variety of applications and quite a few of its stand out features enable its worldwide use. The features include:
- First, its number one feature is the ability to store data in tables. The fact that the very storage of data is in a structured form can significantly reduce iteration time.
- Data persists in the form of rows and columns and allows a facility primary key to define unique identification of rows.
- It creates indexes for quicker data retrieval.
- Allows for various types of data integrity
- Also allows for the virtual table creation, which provides a safe means to store and secure sensitive content.
- Common column implementation and multi user accessibility is included in the RDBMS features.

Advantages of RDBMS

- Data is stored only once and hence multiple record changes are not required. In addition, deletion and modification of data becomes simpler and storage efficiency is very high.
- Complex queries can be carried out using the Structure Query Language. Terms like 'Insert', 'Update', 'Delete', 'Create' and 'Drop' are keywords in SQL that help in accessing a particular data of choice.
- Better security is offered by the creation of tables. Certain tables can be protected by this system. Users can set access barriers to limit access to the available content. It is very useful in companies where a manager can decide which data is provided to the employees and customers. Thus, a customized level of data protection can be enabled.
- Provision for future requirements as new data can easily be added and appended to the existing tables and can be made consistent with the previously available content. This is a feature that no flat file database has.

Disadvantages of RDBMS

Like there are two sides to a coin, RDBMS houses a few drawbacks as well.

- The prime disadvantage of this effective system is its cost of execution. To set up a relational database management system, a special software needs to be purchased. Once it is bought, setting up of data is a tedious task. There will be millions of lines of content to be transferred to the tables. Some cases require the assistance of a programmer and a team of data entry specialists. Care must be taken to ensure secure data does not slip into the wrong hands at the time of data entry.
- Simple text data can be easily added and appended. However, newer forms of data can be confusing. Complex images, numbers, designs are not easy to be categorized into tables and presents a problem.
- Structure limits are another drawback. Certain fields in tables have a character limit.
- Isolated databases can be created if large chunks of information are separated from each other. Connecting such large volumes of data is not easy.

Uses of RDBMS

They find application in disciplines like Banking, airlines, universities, manufacturing and HR.

Using RDBMS can bring a systematic view to raw data. It is easy to understand and execute and hence enables better decision making as well.

It ensures effective running of an accounting system. Moreover, ticket service and passenger information documentation in airlines, student databases in universities and product details along with consumer demand of these products in industries also comes under the wide usage scope of RDBMS.

Object-Oriented Database Model

Object databases were developed in the 1980s to overcome the inconvenience of object-relational impedance mismatch, which led to the coining of the term "post-relational" and the development of hybrid object-relational databases.

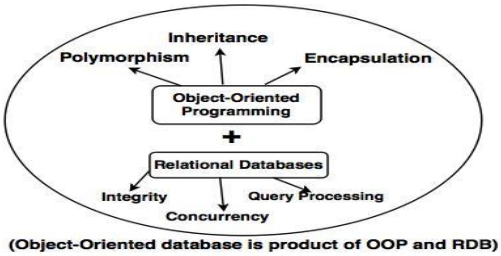
Because we work with objects in Java, it would be ideal if there were a way to store those objects in a database without having to bend and twist the object model at all. In the mid-1990s, object-oriented database systems gained attention. They are based on a network data model, which was common before the advent of the relational data model decades ago. The basic idea is to store a network of objects, with all its pointers and nodes, and to re-create the same in-memory graph later on. This can be optimized with various metadata and configuration settings.

An object-oriented database management system (OODBMS) is more like an extension to the application environment than an external data store. An OODBMS usually features a multi-tiered implementation, with the backend data store, object cache, and client application coupled and interacting via a proprietary network protocol. Object nodes are kept on pages of memory, which are transported from and to the data store.

Object-oriented database development begins with the top-down definition of host language bindings that add persistence capabilities to the programming language. Hence, object databases offer seamless integration into the object-oriented application environment. This is different from the model used by today's relational databases, where interaction with the database occurs via an intermediate language (SQL) and data independence from a particular application is the major concern.

We will not bother looking too closely into why object-oriented database technology has not been more popular; we will observe that object databases have not been widely adopted and that it does not appear likely that they will be in the near future. We are confident that the overwhelming majority of developers will have far more opportunity to work with relational technology, given the current political realities (predefined deployment environments) and the common requirement for data independence.

The object-oriented database derivation is the integrity of object-oriented programming language systems and consistent systems. The power of the object-oriented databases comes from the cyclical treatment of both consistent data, as found in databases, and transient data, as found in executing programs.



(Object-Oriented database is product of OOP and RDB)

Object-Oriented database

Object-oriented databases use small, recyclable separated of software called objects. The objects themselves are stored in the object-oriented database. Each object contains of two elements:

1. Piece of data (e.g., sound, video, text, or graphics).
2. Instructions or software programs called methods, for what to do with the data.

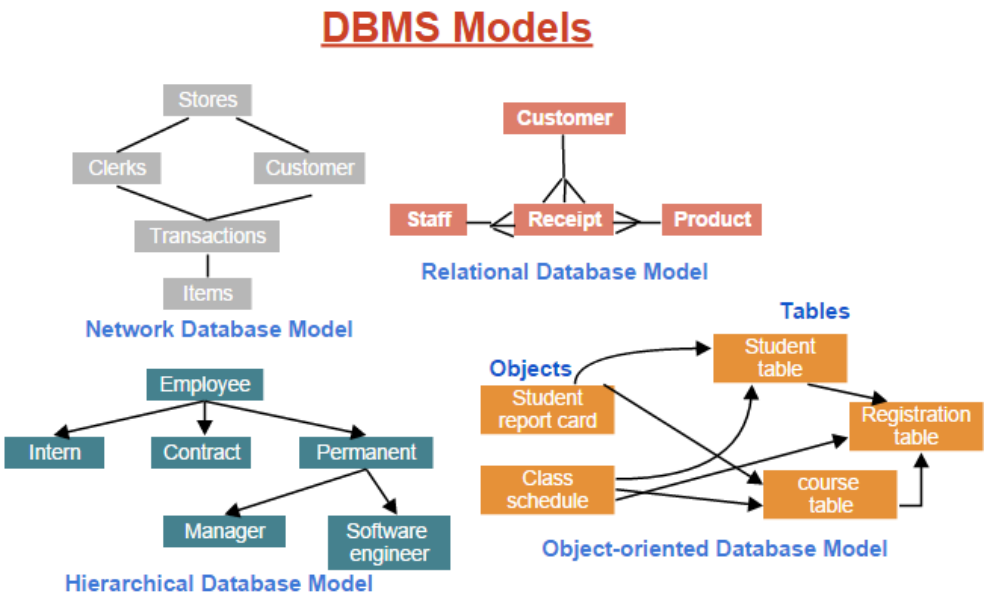
Disadvantage of Object-oriented databases

1. Object-oriented databases have these disadvantages.
2. Object-oriented database are more expensive to develop.
3. In the Most, organizations are unwilling to abandon and convert from those databases.

They have already invested money in developing and implementing.

The benefits to object-oriented databases are compelling. The ability to mix and match reusable objects provides incredible multimedia capability.

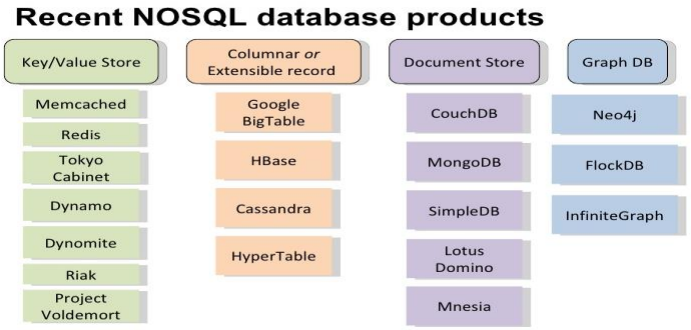
Summary of 4 generations of Databases



No SQL Databases

The next generation of post-relational databases in the late 2000s became known as NoSQL databases, introducing fast key-value stores and document-oriented databases. A competing "next generation" known as NewSQL databases attempted new implementations that retained the relational/SQL model while aiming to match the high performance of NoSQL compared to commercially available relational DBMSs.

Following the technology progress in the areas of processors, computer memory, computer storage, and computer networks, the sizes, capabilities, and performance of databases and their respective DBMSs have grown in orders of magnitude. The development of database technology can be divided into three eras based on data model or structure: navigational, SQL/relational, and post-relational.

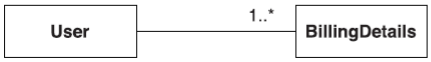


The paradigm mismatch

The object/relational paradigm mismatch can be broken into several parts, which we will examine one at a time.
Let us start our exploration with a simple example that is problem free. As we build on it, you will begin to see the mismatch appear.

Suppose you have to design and implement an online e-commerce application. In this application, you need a class to represent information about a user of the system, and another class to represent information about the user's billing details like below

```
public class User {
    private String username;
    private String name;
    private String address;
    private Set billingDetails;

    // Accessor methods (getter/setter), business methods, etc.
    ...
}
public class BillingDetails {
    private String accountNumber;
    private String accountName;
    private String accountType;
    private User user;

    // Accessor methods (getter/setter), business methods, etc.
    ...
}
```



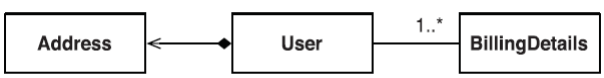
Note that we are only interested in the state of the entities with regard to persistence, so we've omitted the implementation of property accessors and businessmethods (such as getUsername () or billAuction ()).

```
create table USERS (
      USERNAME varchar(15) not null primary key,
      NAME varchar(50) not null,
      ADDRESS varchar(100)
)
```

```
create table BILLING_DETAILS (
      ACCOUNT_NUMBER varchar(10) not null primary key,
      ACCOUNT_NAME varchar(50) not null,
      ACCOUNT_TYPE varchar(2) not null,
      USERNAME varchar(15) foreign key references users
)
```

The relationship between the two entities is represented as the foreign key, USERNAME, in BILLING_DETAILS. For this simple domain model, the object/relational mismatch is barely in evidence; it's straightforward to write JDBC code to insert, update, and delete information about users and billing details. Now, let us see what happens when we consider something a little more realistic. The paradigm mismatch will be visible when we add more entities and entity relationships to our application.

The most glaringly obvious problem with our current implementation is that we have designed an address as a simple String value. In most systems, it is necessary to store street, city, state, country, and ZIP code information separately. Of course, we could add these properties directly to the User class, but because it is highly likely that other classes in the system will also carry address information, it makes more sense to create a separate Address class. The updated model is shown in below figure



Should we also add an ADDRESS table? Not necessarily. It is common to keep address information in the USERS table, in individual columns. This design is likely to perform better, because a table join is not needed if you want to retrieve the user and address in a single query. The nicest solution may even be to create a user-defined SQL datatype to represent addresses, and to use a single column of that new type in the USERS table instead of several new columns.

We have the choice of adding either several columns or a single column (of a new SQL datatype). This is clearly a problem of granularity.

The problem of granularity

Granularity refers to the relative size of the types you are working with. Let us return to our example. Adding a new datatype to our database catalog, to store Address Java instances in a single column, sounds like the best approach. A new Address type (class) in Java and a new ADDRESS SQL datatype should guarantee interoperability. However, you will find various problems if you check the support for user-defined datatypes (UDT) in today's SQL database management systems.

UDT support is one of a number of so-called object-relational extensions to traditional SQL. This term alone is confusing, because it means that the database management system has (or is supposed to support) a sophisticated datatype system— something you take for granted if somebody sells you a system that can handle data in a relational fashion. Unfortunately, UDT support is a somewhat obscure feature of most SQL database management systems and certainly is not portable between different systems. Furthermore, the SQL standard supports user-defined datatypes, but poorly.

This limitation is not the fault of the relational data model. You can consider the failure to standardize such an important piece of functionality as fallout from the object-relational database wars between vendors in the mid-1990s. Today, most developers accept that SQL products have limited type systems. However, even with a sophisticated UDT system in our SQL database management system, we would likely still duplicate the type declarations, writing the new type in Java and again in SQL. Attempts to find a solution for the Java space, such as SQLJ, unfortunately, have not had much success.

For these and whatever other reasons, use of UDTs or Java types inside an SQL database isn't common practice in the industry at this time, and it's unlikely that you'll encounter a legacy schema that makes extensive use of UDTs. We therefore cannot and will not store instances of our new Address class in a single new column that has the same datatype as the Java layer. Our pragmatic solution for this problem has several columns of built-in vendor-defined SQL types (such as Boolean, numeric, and string datatypes). The USERS table is usually defined as follows:

```
create table USERS (
      USERNAME varchar(15) not null primary key,
      NAME varchar(50) not null,
      ADDRESS_STREET varchar(50),
      ADDRESS_CITY varchar(15),
      ADDRESS_STATE varchar(15),
      The paradigm mismatch 13
      ADDRESS_ZIPCODE varchar(5),
      ADDRESS_COUNTRY varchar(15)
)
```

Classes in our domain model come in a range of different levels of granularity— from coarse-grained entity classes like User, to finer-grained classes like Address, down to simple String-valued properties such as zipcode. In contrast, just two levels of granularity are visible at the level of the SQL database: tables such as USERS, and columns such as ADDRESS_ZIPCODE. Many simple persistence mechanisms fail to recognize this mismatch and so end up forcing the less flexible SQL representation upon the object model. We have seen countless User classes with properties named zipcode! It turns out that the granularity problem is not especially difficult to solve.

The problem of subtypes

In Java, you implement type inheritance using super classes and subclasses. To illustrate why this can present a mismatch problem, let us add to our e-commerce application so that we now can accept not only bank account billing, but also credit and debit cards. The most natural way to reflect this change in the model is to use inheritance for the BillingDetails class.

We may have an abstract BillingDetails superclass, along with several concrete subclasses: CreditCard, BankAccount, and so on. Each of these subclasses defines slightly different data (and completely different functionality that acts on that data). The UML class diagram in figure 1.3 illustrates this model. SQL should probably include standard support for super tables and sub tables.

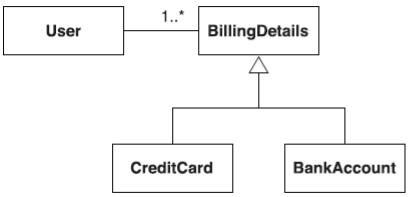


Figure 1.3
Using inheritance for different billing strategies

This would effectively allow us to create a table that inherits certain columns from introduce a new notion: virtual columns in base tables. Traditionally, we expect virtual columns only in virtual tables, which are called views. Furthermore, on a theoretical level, the inheritance we applied in Java is type inheritance. A table isn't a type, so the notion of supertables and subtables is questionable. In any case, we can take the short route here and observe that SQL database products don't generally implement type or table inheritance, and if they do implement it, they don't follow a standard syntax and usually expose you to data integrity problems (limited integrity rules for updatable views). In chapter 5, section 5.1, "Mapping class inheritance," we discuss how ORM solutions such as Hibernate solve the problem of persisting a class hierarchy to a database table or tables.

This problem is now well understood in the community, and most solutions support approximately the same functionality. But we aren't finished with inheritance. As soon as we introduce inheritance into the model, we have the possibility of polymorphism. The User class has an association to the BillingDetails superclass. This is a polymorphic association. At runtime, a User object may reference an instance of any of the subclasses of BillingDetails. Similarly, we want to be able to write polymorphic queries that refer to the BillingDetails class, and have the query return Instances of its subclasses.

SQL databases also lack an obvious way (or at least a standardized way) to represent a polymorphic association. A foreign key constraint refers to exactly one target table; it isn't straightforward to define a foreign key that refers to multiple tables. We'd have to write a procedural constraint to enforce this kind of integrity rule. The result of this mismatch of subtypes is that the inheritance structure in your model must be persisted in an SQL database that doesn't offer an inheritance strategy. The next aspect of the object/relational mismatch problem is the issue of object identity. You probably noticed that we defined USERNAME as the primary key of our USERS table. Was that a good choice? How do we handle identical objects in Java?

The problem of identity

Although the problem of object identity may not be obvious at first, we'll encounter it often in our growing and expanding e-commerce system, such as when we need to check whether two objects are identical. There are three ways to tackle The paradigm mismatch problem: two in the Java world and one in our SQL database. As expected, they work together only with some help.

Java objects define two different notions of sameness:

1. Object identity (roughly equivalent to memory location, checked with a==b)
2. Equality as determined by the implementation of the equals() method (also called equality by value)

On the other hand, the identity of a database row is expressed as the primary key value. "Object identity and equality," neither equals () nor == is naturally equivalent to the primary key value. It is common for several no identical objects to simultaneously represent the same row of the database, for example, in concurrently running application threads. Furthermore, some subtle difficulties are involved in implementing equals () correctly for a persistent class.

Let us discuss another problem related to database identity with an example. In our table definition for USERS, we used USERNAME as a primary key. Unfortunately, this decision makes it difficult to change a username; we need to update not only the USERNAME column in USERS, but also the foreign key column in BILLING_DETAILS. To solve this

problem, later in the book we'll recommend that you use surrogate keys whenever you can't find a good natural key (we'll also discuss what makes a key good). A surrogate key column is a primary key column with no meaning to the user; in other words, a key that isn't presented to the user and is only used for identification of data inside the software system. For example, we may change our table definitions to look like this:

```
create table USERS (
      USER_ID bigint not null primary key,
      USERNAME varchar(15) not null unique,
      NAME varchar(50) not null,
...
)
create table BILLING_DETAILS (
      BILLING_DETAILS_ID bigint not null primary key,
      ACCOUNT_NUMBER VARCHAR(10) not null unique,
      ACCOUNT_NAME VARCHAR(50) not null,
      ACCOUNT_TYPE VARCHAR(2) not null,
      USER_ID bigint foreign key references USER
)
```

The USER_ID and BILLING_DETAILS_ID columns contain system-generated values. These columns were introduced purely for the benefit of the data model, so how (if at all) should they be represented in the domain model

In the context of persistence, identity is closely related to how the system handles caching and transactions. Different persistence solutions have chosen different strategies, and this has been an area of confusion. We cover all these interesting topics—and show how they are related—in chapters 10 and 13. So far, the skeleton e-commerce application we've designed has identified the mismatch problems with mapping granularity, subtypes, and object identity. We're almost ready to move on to other parts of the application, but first we need to discuss the important concept of associations: how the relationships between our classes are mapped and handled. Is the foreign key in the database all you need?.

Problems relating to associations

In our domain model, associations represent the relationships between entities. The User, Address, and BillingDetails classes are all associated; but unlike Address, BillingDetails stands on its own. BillingDetails instances are stored in their own table. Association mapping and the management of entity associations are central concepts in any object persistence solution.

Object-oriented languages represent associations using object references; but in the relational world, an association is represented as a foreign key column, with copies of key values (and a constraint to guarantee integrity). There are substantial differences between the two representations.

Object references are inherently directional; the association is from one object to the other. They're pointers. If an association between objects should be navigable in both directions, you must define the association twice, once in each of the associated classes. You've already seen this in the domain model classes:

```
public class User {
       private Set billingDetails;
       ...
}
public class BillingDetails {
       private User user;
       ...
}
```

On the other hand, foreign key associations are not by nature directional. Navigation has no meaning for a relational data model because you can create arbitrary data associations with table joins and projection. The challenge is to bridge a completely open data model, which is independent of the application that works with the data, to an application-dependent navigational model, a constrained view of the associations needed by this particular application. It is not possible to determine the multiplicity of a unidirectional association by looking only at the Java classes. Java associations can have many-to-many multiplicity.

For example, the classes could look like this:

```
public class User {
        private Set billingDetails;
        ...
}

public class BillingDetails {
        private Set users;
        ...
}
```

Table associations, on the other hand, are always one-to-many or one-to-one. You can see the multiplicity immediately by looking at the foreign key definition. The following is a foreign key declaration on the BILLING_DETAILS table for a one-tomany association (or, if read in the other direction, a many-to-one association):

USER_ID bigint foreign key references USERS

These are one-to-one associations:

USER_ID bigint unique foreign key references USERS BILLING_DETAILS_ID bigint primary key foreign key references USERS

If you wish to represent a many-to-many association in a relational database, you must introduce a new table, called a link table. This table doesn't appear anywhere in the domain model. For our example, if we consider the relationship between the user and the billing information to be many-to-many, the link table is defined as follows:

```
create table USER_BILLING_DETAILS (
        USER_ID bigint foreign key references USERS,
        BILLING_DETAILS_ID bigint foreign key references BILLING_DETAILS,
        PRIMARY KEY (USER_ID, BILLING_DETAILS_ID)
)
```

The cost of the mismatch

We now have quite a list of object/relational mismatch problems, and it will be costly (in time and effort) to find solutions, as you may know from experience. This cost is often underestimated, and we think this is a major reason for many failed software projects. In our experience (regularly confirmed by developers we talk to), the main purpose of up to 30 percent of the Java application code written is to handle the tedious SQL/JDBC and manual bridging of the object/relational paradigm mismatch. Despite all this effort, the result still does not feel quite right. We have seen projects nearly sink due to the complexity and inflexibility of their database abstraction layers. We also see Java developers (and DBAs) quickly lose their confidence when design decisions about the persistence strategy for a project have to be made.

One of the major costs is in the area of modeling. The relational and domain models must both encompass the same business entities, but an object-oriented purist will model these entities in a different way than an experienced relational data modeler would. The usual solution to this problem is to bend and twist the domain model and the implemented classes until they match the SQL database schema. (Which, following the principle of data independence, is certainly a safe long-term choice.)

This can be done successfully, but only at the cost of losing some of the advantages of object orientation. Keep in mind that relational modeling is underpinned by relational theory. Object orientation has no such rigorous mathematical definition or body of theoretical work, so we cannot look to mathematics to explain how we should bridge the gap between the two paradigms—there is no elegant transformation waiting to be discovered. (Doing away with Java and SQL, and starting from scratch is not considered elegant.) The domain-modeling mismatch is not the only source of the inflexibility and the lost productivity that lead to higher costs. A further cause is the JDBC API itself. JDBC and SQL provide a statement-oriented (that is, command-oriented) approach to moving data to and from an SQL database. If you want to query or manipulate data, the tables and columns involved must be specified at least three times (insert, update, select), adding to the time required for design and implementation. The distinct dialects for every SQL database management system do not improve the situation.

To round out our understanding of object persistence, and before we approach possible solutions, we need to discuss application architecture and the role of a persistence layer in typical application design.

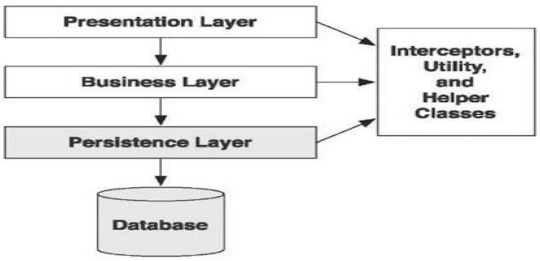
Persistence layers and alternatives

In a medium- or large-sized application, it usually makes sense to organize classes by concern. Persistence is one concern; others include presentation, workflow, and business logic.1 A typical object-oriented architecture includes layers of code that represent the concerns. It is normal and certainly best practice to group all classes and components responsible for persistence into a separate persistence layer in a layered system architecture. In this section, we first look at the layers of this type of architecture and why we use them. After that, we focus on the layer we are most interested in—the persistence layer—and some of the ways it can be implemented.

Layered architecture

- ✓ A layered architecture defines interfaces between codes that implements the various concerns, allowing changes to be made to the way one concern is implemented without significant disruption to code in the other layers. Layering also determines the kinds of interlayer dependencies that occur. The rules are as follows:
- ✓ Layers communicate from top to bottom. A layer is dependent only on the layer directly below it.
- ✓ Each layer is unaware of any other layers except for the layer just below it.
- ✓ Different systems group concerns differently, so they define different layers. A typical, proven, high-level application architecture uses three layers: one each for presentation, business logic, and persistence.

Let us take a closer look at the layers and elements in the diagram:



Presentation layer: - The user interface logic is topmost. Code responsible for the presentation and control of page and screen navigation is in the presentation layer.

Business layer: - The exact form of the next layer varies widely between applications. It's generally agreed, however, that the business layer is responsible for implementing any business rules or system requirements that would be understood by users as part of the problem domain. This layer usually includes some kind of controlling component—code that knows when to invoke which business rule. In some systems, this layer has its own internal representation of the business domain entities, and in others it reuses the model defined by the persistence layer.

Persistence layer: - The persistence layer is a group of classes and components responsible for storing data to, and retrieving it from, one or more data stores. This layer necessarily includes a model of the business domain entities (even if it's only a metadata model).

Database: - The database exists outside the Java application itself. It's the actual, persistent representation of the system state. If an SQL database is used, the database includes the relational schema and possibly stored procedures.

Helper and utility classes: - Every application has a set of infrastructural helper or utility classes that are used in every layer of the application (such as Exception classes for error handling). These infrastructural elements do not form a layer, because they do not obey the rules for interlayer dependency in a layered architecture.

Let us now take a brief look at the various ways the persistence layer can be implemented by Java applications.

Hand coding a persistence layer with SQL/JDBC

The most common approach to Java persistence is for application programmers to work directly with SQL and JDBC. After all, developers are familiar with relational database management systems, they understand SQL, and they know how to work with tables and foreign keys. Moreover, they can always use the well-known and widely used data access object (DAO) pattern to hide complex JDBC code and no portable SQL from the business logic.

The DAO pattern is a good one—so good that we often recommend its use even with ORM. However, the work involved in manually coding persistence for each domain class is considerable, particularly when multiple SQL dialects are supported. This work usually ends up consuming a large portion of the development effort. Furthermore, when requirements change, a hand-coded solution always requires more attention and maintenance effort.

Why not implement a simple mapping framework to fit the specific requirements of your project? The result of such an effort could even be reused in future projects. Many developers have taken this approach; numerous homegrown object/relational persistence layers are in production systems today. However, we do not recommend this approach. Excellent solutions already exist: not only the (mostly expensive) tools sold by commercial vendors, but also open source projects with free licenses. We are certain you will be able to find a solution that meets your requirements, both business and technical. It is likely that such a solution will do a great deal more, and do it better, than a solution you could build in a limited time.

Developing a reasonably full-featured ORM may take many developers' months. For example, Hibernate is about 80,000 lines of code, some of which is much more difficult than typical application code, along with 25,000 lines of unit test code. This may be more code than is in your application. Details can easily be overlooked in such a large project—as both the authors know from experience! Even if an existing tool does not fully implement two or three of your more exotic requirements, it's still probably not worth creating your own tool. Any ORM software will handle the tedious common cases—the ones that kill productivity.

Object/relational mapping

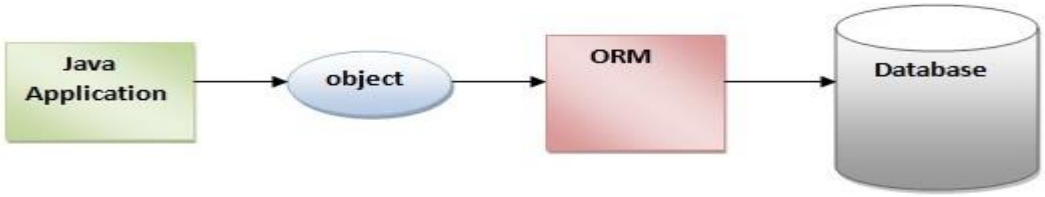
Now that we have looked at the alternative techniques for object persistence, it is time to introduce the solution we feel is the best, and the one we use with Hibernate: ORM. Despite its long history (the first research papers were published in the late 1980s), the terms for ORM used by developers vary. Some call it object relational mapping, others prefer the simple object mapping; we exclusively use the term object/relational mapping and its acronym, ORM. The slash stresses the mismatch problem that occurs when the two worlds collide.

In this section, we first look at what ORM is. Then we enumerate the problems that a good ORM solution needs to solve. Finally, we discuss the general benefits that ORM provides and why we recommend this solution.

What is ORM?

Briefly, object/relational mapping is the automated (and transparent) persistence of objects in a Java application to the tables in a relational database, using metadata that describes the mapping between the objects and the database.

ORM, in essence, works by (reversibly) transforming data from one representation to another. This implies certain performance penalties. However, if ORM is implemented as middleware, there are many opportunities for optimization that would not exist for a hand-coded persistence layer. The provision and management of metadata that governs the transformation adds to the overhead at development time, but the cost is less than equivalent costs involved in maintaining a hand-coded solution. (And even object databases require significant amounts of metadata.)



An ORM solution consists of the following four pieces:

- An API for performing basic CRUD operations on objects of persistent classes
- A language or API for specifying queries that refer to classes and properties of classes
- A facility for specifying mapping metadata
- A technique for the ORM implementation to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions

We are using the term full ORM to include any persistence layer where SQL is automatically generated from a metadata-based description. We are not including persistence layers where the object/relational mapping problem is solved manually by developers hand-coding SQL with JDBC. With ORM, the application interacts with the ORM APIs and the domain model classes and is abstracted from the underlying SQL/JDBC. Depending on the features or the particular implementation, the ORM engine may also take on responsibility for issues such as optimistic locking and caching, relieving the application of these concerns entirely.

Let's look at the various ways ORM can be implemented. Mark Fussel (Fussel, 1997), a developer in the field of ORM, defined the following four levels of ORM quality. We have slightly rewritten his descriptions and put them in the context of today's Java application development.

Pure relational

The whole application, including the user interface, is designed around the relational model and SQL-based relational operations. This approach, despite its deficiencies for large systems, can be an excellent solution for simple applications where a low level of code reuse is tolerable. Direct SQL can be fine-tuned in every aspect, but the drawbacks, such as lack of portability and maintainability, are significant, especially in the long run. Applications in this category often make heavy use of stored procedures, shifting some of the work out of the business layer and into the database.

Light object mapping

Entities are represented as classes that are mapped manually to the relational tables. Hand-coded SQL/JDBC is hidden from the business logic using well-known design patterns. This approach is extremely widespread and is successful for applications with a small number of entities, or applications with generic, metadata-driven data models. Stored procedures may have a place in this kind of application.

Medium object mapping

The application is designed around an object model. SQL is generated at build time using a code-generation tool, or at runtime by framework code. Associations between objects are supported by the persistence mechanism, and queries may be specified using an object-oriented expression language. Objects are cached by the persistence layer. A great many ORM products and homegrown persistence layers support at least this level of functionality. It's well suited to medium-sized applications with some complex transactions, particularly when portability between different database products is important. These applications usually don't use stored procedures.

Full object mapping

Full object mapping supports sophisticated object modeling: composition, inheritance, polymorphism, and persistence by reachability. The persistence layer implements transparent persistence; persistent classes do not inherit from any special base class or have to implement a special interface. Efficient fetching strategies (lazy, eager, and prefetching) and caching strategies are implemented transparently to the application. A homegrown persistence layer can hardly achieve this level of functionality—it's equivalent to years of development time. A number of commercial and open source Java ORM tools have achieved this level of quality. This level meets the definition of ORM we are using in this topic.

Let us look at the problems we expect to be solved by a tool that achieves full object mapping.

Generic ORM problems

The following list of issues, which we will call the ORM problems, identifies the fundamental questions resolved by a full object/relational mapping tool in a Java environment. Particular ORM tools may provide extra functionality (for example, aggressive caching), but this is a reasonably exhaustive list of the conceptual issues and questions that are specific to object/relational mapping.

1. What do persistent classes look like? How transparent is the persistence tool? Do we have to adopt a programming model and conventions for classes of the business domain?

2. How is mapping metadata defined? Because the object/relational transformation is governed entirely by metadata, the format and definition of this metadata is important. Should an ORM tool provide a GUI interface to manipulate the metadata graphically? Or are there better approaches to metadata definition?

3. How do object identity and equality relate to database (primary key) identity? How do we map instances of particular classes to particular table rows?

4. How should we map class inheritance hierarchies? There are several standard strategies. What about polymorphic associations, abstract classes, and interfaces?

5. How does the persistence logic interact at runtime with the objects of the business domain? This is a problem of generic programming, and there are a number of solutions including source generation, runtime reflection, runtime bytecode generation, and build-time bytecode enhancement. The solution to this problem may affect your build process (but, preferably, should not otherwise affect you as a user).

6. What is the lifecycle of a persistent object? Does the lifecycle of some objects depend upon the lifecycle of other associated objects? How do we translate the lifecycle of an object to the lifecycle of a database row?

7. What facilities are provided for sorting, searching, and aggregating? The application could do some of these things in memory, but efficient use of relational technology requires that this work often be performed by the database.

8. How do we efficiently retrieve data with associations? Efficient access to relational data is usually accomplished via table joins. Object-oriented applications usually access data by navigating an object network. Two data access patterns should be avoided when possible: the n+1 selects problem, and its complement, the Cartesian product problem (fetching too much data in a single select).

Two additional issues that impose fundamental constraints on the design and architecture of an ORM tool are common to any data access technology:

- Transactions and concurrency
- Cache management (and concurrency)

As you can see, a full object/relational mapping tool needs to address quite a long list of issues. By now, you should be starting to see the value of ORM. In the next section, we look at some of the other benefits you gain when you use an ORM solution.

Why ORM?

An ORM implementation is a complex beast—less complex than an application server, but more complex than a web application framework like Struts or Tapestry. Why should we introduce another complex infrastructural element into our system? Will it be worth it?

It will take us most of this topic to provide a complete answer to those questions, but this section provides a quick summary of the most compelling benefits. First, though, let's quickly dispose of a no benefit.

A supposed advantage of ORM is that it shields developers from messy SQL. This view holds that object-oriented developers can't be expected to understand SQL or relational databases well, and that they find SQL somehow offensive. On the contrary, we believe that Java developers must have a sufficient level of familiarity with—and appreciation of—relational modeling and SQL in order to work with ORM. ORM is an advanced technique to be used by developers who have already done it the hard way. To use Hibernate effectively, we must be able to view and interpret the SQL statements it issues and understand the implications for performance.

Now, let's look at some of the benefits of ORM and Hibernate.

Productivity

Persistence-related code can be perhaps the most tedious code in a Java application. Hibernate eliminates much of the grunt work (more than you'd expect) and lets you concentrate on the business problem.

No matter which application-development strategy you prefer—top-down, starting with a domain model, or bottom-up, starting with an existing database schema—hibernate, used together with the appropriate tools, will significantly reduce development time.

Maintainability

Fewer lines of code (LOC) make the system more understandable, because it emphasizes business logic rather than plumbing. Most important, a system with less code is easier to refactor. Automated object/relational persistence substantially reduces LOC. Of course, counting lines of code is a debatable way of measuring application complexity.
However, there are other reasons that a Hibernate application is more maintainable. In systems with hand-coded persistence, an inevitable tension exists between the relational representation and the object model implementing the domain. Changes to one usually involve changes to the other, and often the design of one representation is compromised to accommodate the existence of the other. (What usually happens in practice is that the object model of the domain is compromised.) ORM provides a buffer between the two models, allowing more elegant use of object orientation on the Java side, and insulating each model from minor changes to the other.

Performance

A common claim is that hand-coded persistence can always be at least as fast, and can often be faster, than automated persistence. This is true in the same sense that it's true that assembly code can always be at least as fast as Java code, or a handwritten parser can always be at least as fast as a parser generated by YACC or ANTLR—in other words, it's beside the point. The unspoken implication of the claim is that hand-coded persistence will perform at least as well in an actual application. But this implication will be true only if the effort required to implement at-least-as-fast hand-coded persistence is similar to the amount of effort involved in utilizing an automated solution. The interesting question is what happens when we consider time and budget constraints?

Given a persistence task, many optimizations are possible. Some (such as query hints) are much easier to achieve with hand-coded SQL/JDBC. Most optimizations, however, are much easier to achieve with automated ORM. In a project with time constraints, hand-coded persistence usually allows you to make some optimizations. Hibernate allows many more optimizations to be used all the time. Furthermore, automated persistence improves developer productivity so much that you can spend more time hand optimizing the few remaining bottlenecks.

Finally, the people who implemented your ORM software probably had much more time to investigate performance optimizations than you have. Did you know, for instance, that pooling PreparedStatement instances results in a significant performance increase for the DB2 JDBC driver but breaks the InterBase JDBC driver? Did you realize that updating only the changed columns of a table can be significantly faster for some databases but potentially slower for others? In your handcrafted solution, how easy is it to experiment with the impact of these various strategies?

Vendor independence

An ORM abstracts your application away from the underlying SQL database and SQL dialect. If the tool supports a number of different databases (and most do), this confers a certain level of portability on your application. You should not necessarily expect write-once/run-anywhere, because the capabilities of databases differ, and achieving full portability would require sacrificing some of the strength of the more powerful platforms. Nevertheless, it is usually much easier to develop a cross-platform application using ORM. Even if you do not require cross-platform operation, an ORM can still help mitigate some of the risks associated with vendor lock-in.

Introducing Hibernate, EJB3, and JPA

Hibernate is a full object/relational mapping tool that provides all the previously listed ORM benefits. The API you are working with in Hibernate is native and designed by the Hibernate developers. The same is true for the query interfaces and query languages, and for how object/relational mapping metadata is defined.

Before you start your first project with Hibernate, you should consider the EJB 3.0 standard and its sub specification, Java Persistence. Let us go back in history and see how this new standard came into existence.

Many Java developers considered EJB 2.1 entity beans as one of the technologies for the implementation of a persistence layer. The whole EJB programming and persistence model has been widely adopted in the industry, and it has been an important factor in the success of J2EE (or, Java EE as it's now called). However, over the last years, critics of EJB in the developer community became more vocal (especially with regard to entity beans and persistence), and companies realized that the EJB standard should be improved. Sun, as the steering party of J2EE, knew that an overhaul was in order and started a new Java specification request (JSR) with the goal of simplifying EJB in early 2003. This new JSR, Enterprise JavaBeans 3.0 (JSR 220), attracted significant interest. Developers from the Hibernate team joined the expert group early on and helped shape the new specification. Other vendors, including all major and many smaller companies in the Java industry, also contributed to the effort. An important decision made for the new standard was to specify and standardize things that work in practice, taking ideas and concepts from existing successful products and projects. Hibernate, therefore, being a successful data persistence solution, played an important role for the persistence part of the new standard.

Understanding the standards

First, it is difficult (if not impossible) to compare a specification and a product. The questions that should be asked are, "Does Hibernate implement the EJB 3.0 specification, and what is the impact on my project? Do I have to use one or the other?"

The new EJB 3.0 specification comes in several parts: The first part defines new EJB programming model for session beans and message-driven beans, the deployment rules, and so on. The second part of the specification deals with persistence exclusively: entities, object/relational mapping metadata, persistence manager interfaces, and the query language. This second part is called Java Persistence API (JPA), probably because its interfaces are in the package javax.persistence. We will use this acronym throughout the book. This separation also exists in EJB 3.0 products; some implement a full EJB 3.0 container that supports all parts of the specification, and other products may implement only the Java Persistence part. Two important principles were designed into the new standard:

■ JPA engines should be pluggable, which means you should be able to take out one product and replace it with another if you are not satisfied—even if you want to stay with the same EJB 3.0 container or Java EE 5.0 application server.

■ JPA engines should be able to run outside of an EJB 3.0 (or any other) runtime environment, without a container in plain standard Java.

The consequences of this design are that there are more options for developers and architects, which drives competition and therefore improves overall quality of products. Of course, actual products also offer features that go beyond the specification as vendor-specific extensions (such as for performance tuning, or because the vendor has a focus on a particular vertical problem space). Hibernate implements Java Persistence, and because a JPA engine must be pluggable, new and interesting combinations of software are possible. You can select from various Hibernate software modules and combine them depending on your project's technical and business requirements.

Hibernate Core

The Hibernate Core is also known as Hibernate 3.2.x, or Hibernate. It is the base service for persistence, with its native API and its mapping metadata stored in XML files. It has a query language called HQL (almost the same as SQL), as well as programmatic query interfaces for Criteria and Example queries. There are hundreds of options and features available for everything, as Hibernate Core is really the foundation and the platform all other modules are built on. You can use Hibernate Core on its own, independent from any framework or any particular runtime environment with all JDKs.

It works in every Java EE/J2EE application server, in Swing applications, in a simple servlet container, and so on. As long as you can configure a data source for Hibernate, it works. Your application code (in your persistence layer) will use Hibernate APIs and queries, and your mapping metadata is written in native Hibernate XML files.

Hibernate Annotations

A new way to define application metadata became available with JDK 5.0: type-safe annotations embedded directly in the Java source code. Many Hibernate users are already familiar with this concept, as the XDoclet software supports Javadoc metadata attributes and a preprocessor at compile time (which, for Hibernate, generates XML mapping files). With the Hibernate Annotations package on top of Hibernate Core, you can now use type-safe JDK 5.0 metadata as a replacement or in addition to native Hibernate XML mapping files. You'll find the syntax and semantics of the mapping annotations familiar once you've seen them side-by-side with Hibernate XML mapping files. However, the basic annotations aren't proprietary.

The JPA specification defines object/relational mapping metadata syntax and semantics, with the primary mechanism being JDK 5.0 annotations. (Yes, JDK 5.0 is required for Java EE 5.0 and EJB 3.0.) Naturally, the Hibernate Annotations are a set of basic annotations that implement the JPA standard, and they're also a set of extension annotations you need for more advanced and exotic Hibernate mappings and tuning. You can use Hibernate Core and Hibernate Annotations to reduce your lines of code for mapping metadata, compared to the native XML files, and you may like the better refactoring capabilities of annotations. You can use only JPA annotations, or you can add a Hibernate extension annotation if complete portability isn't your primary concern

Hibernate EntityManager

The JPA specification also defines programming interfaces, lifecycle rules for persistent objects, and query features. The Hibernate implementation for this part of JPA is available as Hibernate EntityManager, another optional module you can stack on top of Hibernate Core. You can fall back when a plain Hibernate interface or even a JDBC Connection is needed. Hibernates native features are a superset of the JPA persistence features in every respect. (The simple fact is that Hibernate EntityManager is a small wrapper around Hibernate Core that provides JPA compatibility.)

Working with standardized interfaces and using a standardized query language has the benefit that you can execute your JPA-compatible persistence layer with any EJB 3.0 compliant application server. Or, you can use JPA outside of any particular standardized runtime environment in plain Java (which really means everywhere Hibernate Core can be used).

Hibernate Annotations should be considered in combination with Hibernate EntityManager. It's unusual that you'd write your application code against JPA interfaces and with JPA queries, and not create most of your mappings with JPA annotations.

Java EE 5.0 application servers

Hibernate is also part of the JBoss Application Server (JBoss AS), an implementation of J2EE 1.4 and (soon) Java EE 5.0. A combination of Hibernate Core, Hibernate Annotations, and Hibernate EntityManager forms the persistence engine of this application server. Hence, everything you can use stand-alone, you can also use inside the application server with all the EJB 3.0 benefits, such as session beans, message-driven beans, and other Java EE services.

To complete the picture, you also have to understand that Java EE 5.0 application servers are no longer the monolithic beasts of the J2EE 1.4 era. In fact, the JBoss EJB 3.0 container also comes in an embeddable version, which runs inside other application servers, and even in Tomcat, or in a unit test, or a Swing application.

Summary

We have discussed the concept of object persistence and the importance of ORM as an implementation technique.
Object persistence means that individual objects can outlive the application process; they can be saved to a data store and be re-created at a later point in time. The object/relational mismatch comes into play when the data store is an SQL-based relational database management system. For instance, a network of objects cannot be saved to a database table; it must be disassembled and persisted to columns of portable SQL datatypes. A good solution for this problem is object/relational mapping (ORM), which is especially helpful if we consider richly typed Java domain models.

A domain model represents the business entities used in a Java application. In a layered system architecture, the domain model is used to execute business logic in the business layer (in Java, not in the database). This business layer communicates with the persistence layer beneath in order to load and store the persistent objects of the domain model.

ORM is the middleware in the persistence layer that manages the persistence. ORM is not a silver bullet for all persistence tasks; its job is to relieve the developer of 95 percent of object persistence work, such as writing complex SQL statements with many table joins, and copying values from JDBC result sets to objects or graphs of objects. A full-featured ORM middleware solution may provide database portability, certain optimization techniques like caching, and other viable functions that aren't easy to hand-code in a limited time with SQL and JDBC It's likely that a better solution than ORM will exist someday. We (and many others) may have to rethink everything we know about SQL, persistence API standards, and application integration. The evolution of today's systems into true relational database systems with seamless object-oriented integration remains pure speculation. However, we cannot wait, and there is no sign that any of these issues will improve soon (a multibillion dollar industry is not very agile). ORM is the best solution currently available, and it is a timesaver for developers facing the object/relational mismatch every day. With EJB 3.0, a specification for full object/relational mapping software that is accepted in the Java industry is finally available.

HIBERNATE

Hibernate is a full object/relational mapping tool that provides all the previously listed ORM benefits. Hibernate was started in 2001 by Gavin King with colleagues from Cirrus Technologies as an alternative to using EJB2-style entity beans. The original goal was to offer better persistence capabilities than those offered by EJB2, by simplifying the complexities and supplementing certain missing features.

In early 2003, the Hibernate development team began Hibernate2 releases, which offered many significant improvements over the first release.

JBoss, Inc. (now part of Red Hat) later hired the lead Hibernate developers in order to further its development.

Hibernate ORM

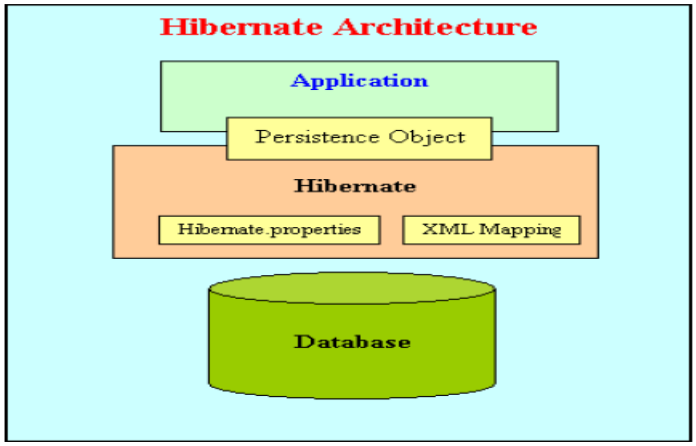
- ✓ Hibernate is an Object/Relational Mapping framework for Java environments.
- ✓ Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities.
- ✓ It can also significantly reduce development time otherwise spent with manual data handling in SQL and JDBC.
- ✓ Hibernate can certainly help you to remove or encapsulate vendor-specific SQL code.
- ✓ In JDBC all exceptions are checked exceptions, so we must write code in try, catch and throws, but in hibernate we only have Un-checked exceptions, so no need to write try, catch, or no need to write throws.
- ✓ Hibernate has capability to generate primary keys automatically while we are storing the records into database.
- ✓ Hibernate supports caching mechanism by this, the number of round trips between an application and the database will be reduced, by using this caching technique an application performance will be increased automatically.
- ✓ Hibernate supports annotations based programming.
- ✓ Hibernate provided Dialect classes which create Queries Dynamically.
- ✓ Hibernate has its own query language, i.e hibernate query language which is database independent.
- ✓ Hibernate supports collections like List, Set, Map (Only new collections).

Actually, Hibernate is much more than ORM Tool (Object - Relational Mapping) because today it is providing many features in the persistence data layer.

Hibernate Architecture

The Hibernate architecture is layered to keep us isolated from having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

There are 4 layers in hibernate architecture they are, java application layer, hibernate framework layer, backhand api layer and database layer. Let us see the diagram of hibernate architecture:



Java program will expose the data in the form of objects, Using Hibernate we can store those objects into the database directly.

Internal Structure of Hibernate application

The programming interfaces are the first thing you have to learn about Hibernate in order to use it in the persistence layer of your application. A major objective of API design is to keep the interfaces between software components as narrow as possible. In practice, however, ORM APIs are not especially small. Do not worry, though; you do not have to understand all the Hibernate interfaces at once. Figure 2.1 illustrates the roles of the most important Hibernate interfaces in the business and persistence layers. We show the business layer above the persistence layer, since the business layer acts as a client of the persistence layer in a traditionally layered application. Note that some simple applications might not cleanly separate business logic from persistence logic; that is okay—it merely simplifies the diagram.

The Hibernate interfaces shown in figure 2.1 may be approximately classified as follows:

■ Interfaces called by applications to perform basic CRUD and querying operations. These interfaces are the main point of dependency of application business/control logic on Hibernate. They include Session, Transaction, and Query.

■ Interfaces called by application infrastructure code to configure Hibernate, most importantly the Configuration class.

■ Callback interfaces that allow the application to react to events occurring inside Hibernate, such as Interceptor, Lifecycle, and Validatable.

■ Interfaces that allow extension of Hibernate powerful mapping functionality, such as UserType, CompositeUserType, and IdentifierGenerator. These interfaces are implemented by application infrastructure code (if necessary).

Hibernate makes use of existing Java APIs, including JDBC), Java Transaction API (JTA, and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

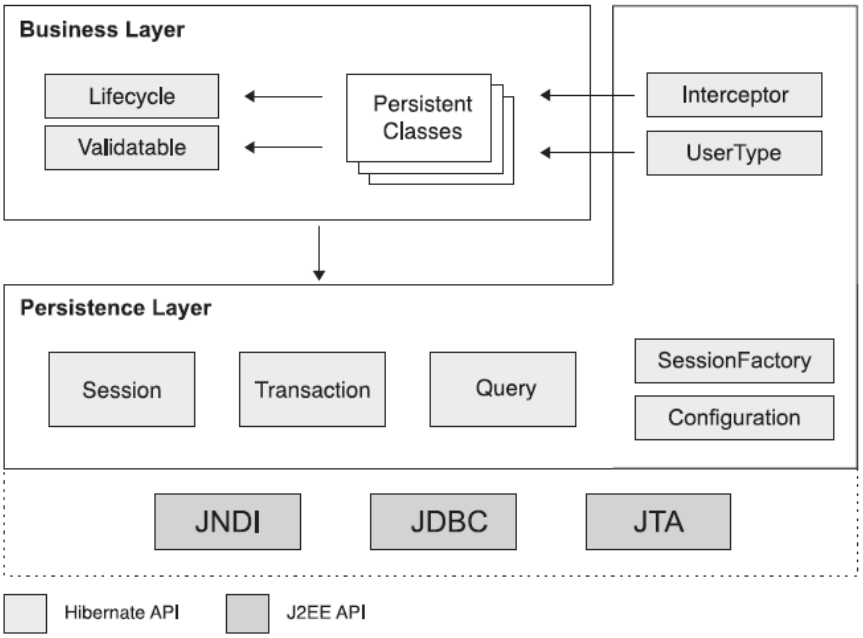


Figure 2.1 High-level overview of the HIbernate API in a layered architecture

The core interfaces

The five core interfaces are used in just about every Hibernate application. Using these interfaces, you can store and retrieve persistent objects and control transactions.

Session interface

The Session interface is the primary interface used by Hibernate applications. An instance of Session is lightweight and is inexpensive to create and destroy. This is important because your application will need to create and destroy sessions all the time, perhaps on every request. Hibernate sessions are not threadsafe and should by design be used by only one thread at a time. The Hibernate notion of a session is something between connection and transaction. It may be easier to think of a session as a cache or collection of loaded objects relating to a single unit of work. Hibernate can detect changes to the objects in this unit of work. We sometimes call the Session a persistence manager because it's also the interface for persistence-related operations such as storing and retrieving objects. Note that a Hibernate session has nothing to do with the web-tier HttpSession. When we use the word session in this book, we mean the Hibernate session.

SessionFactory interface

The application obtains Session instances from a SessionFactory. Compared to the Session interface, this object is much less exciting. The SessionFactory is certainly not lightweight! It's intended to be shared among many application threads. There is typically a single SessionFactory for the whole application—created during application initialization, for example. However, if your application accesses multiple databases using Hibernate, we need a SessionFactory for each database. The SessionFactory caches generated SQL statements and other mapping metadata that Hibernate uses at runtime. It also holds cached data that has been read in one unit of work and may be reused in a future unit of work (only if class and collection mappings specify that this second-level cache is desirable).

Configuration interface

The Configuration object is used to configure and bootstrap Hibernate. The application uses a Configuration instance to specify the location of mapping documents and Hibernate-specific properties and then create the SessionFactory. Even though the Configuration interface plays a relatively small part in the total scope of a Hibernate application, it is the first object we will meet when we begin using Hibernate.

Transaction interface

The Transaction interface is an optional API. Hibernate applications may choose not to use this interface, instead managing transactions in their own infrastructure code. A Transaction abstracts application code from the underlying transaction implementation—which might be a JDBC transaction, a JTA UserTransaction, or even a Common Object Request Broker Architecture (CORBA) transaction— allowing the application to control transaction boundaries via a consistent API. This helps to keep Hibernate applications portable between different kinds of execution environments and containers.

Query and Criteria interfaces

The Query interface allows you to perform queries against the database and control how the query is executed. Queries are written in HQL or in the native SQL dialect of your database. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query. The Criteria interface is very similar; it allows you to create and execute object-oriented criteria queries. To help make application code less verbose, Hibernate provides some shortcut methods on the Session interface that let you invoke a query in one line of code. We won't use these shortcuts in the book; instead, we'll always use the Query interface. A Query instance is lightweight and can't be used outside the Session that created it.

Callback interfaces

Callback interfaces allow the application to receive a notification when something interesting happens to an object—for example, when an object is loaded, saved, or deleted. Hibernate applications don't need to implement these callbacks, but they're useful for implementing certain kinds of generic functionality, such as creating audit records. The Lifecycle and Validatable interfaces allow a persistent object to react to events relating to its own persistence lifecycle. The persistence lifecycle is encompassed by an object's CRUD operations. The Hibernate team was heavily influenced by other ORM solutions that have similar callback interfaces. Later, they realized that having the persistent classes implement Hibernate-specific interfaces probably isn't a good idea, because doing so pollutes our persistent classes with nonportable code. Since these approaches are no longer favored, we don't discuss them in this book. The Interceptor interface was introduced to allow the application to process callbacks without forcing the persistent classes to implement Hibernate-specific APIs. Implementations of the Interceptor interface are passed to the persistent instances as parameters

Types

A fundamental and very powerful element of the architecture is Hibernate's notion of a Type. A Hibernate Type object maps a Java type to a database column type (actually, the type may span multiple columns). All persistent properties of persistent classes, including associations, have a corresponding Hibernate type. This design makes Hibernate extremely flexible and extensible. There is a rich range of built-in types, covering all Java primitives and many JDK classes, including types for java.util.Currency, java.util.Calendar, byte[], and java.io.Serializable. Even better, Hibernate supports user-defined custom types. The interfaces UserType and CompositeUserType are provided to allow you to add your own types. You can use this feature to allow commonly used application classes such as Address, Name, or MonetaryAmount to be handled conveniently and elegantly. Custom types are considered a central feature of Hibernate, and you're encouraged to put them to new and creative uses!

Extension interfaces

Much of the functionality that Hibernate provides is configurable, allowing you to choose between certain built-in strategies. When the built-in strategies are insufficient, Hibernate will usually let you plug in your own custom implementation by implementing an interface. Extension points include:

- **Primary key generation (IdentiferGenerator interface)**
- **SQL dialect support (Dialect abstract class)**
- **Caching strategies (Cache and CacheProvider interfaces)**
- **JDBC connection management (ConnectionProvider interface)**
- **Transaction management (TransactionFactory, Transaction, and TransactionManagerLookup interfaces)**
- **ORM strategies (ClassPersister interface hierarchy)**
- **Property access strategies (PropertyAccessor interface)**
- **Proxy creation (ProxyFactory interface)**

Hibernate ships with at least one implementation of each of the listed interfaces, so you do not usually need to start from scratch if you wish to extend the built-in functionality. The source code is available for you to use as an example for your own implementation. By now, you can see that before we can start writing any code that uses Hibernate, we must answer this question: How do we get a Session to work with?

Configuration

- The org.hibernate.cfg.Configuration class is the basic element of the Hibernate API that allows us to build SessionFactory.
- That means we can refer to Configuration as a factory class that produces SessionFactory.
- The Configuration object encapsulates the Hibernate configuration details such as connection properties, dialect and mapping details described in the Hibernate mapping documents which are used to build the SessionFactory.
- The Configuration object takes the responsibility to read the Hibernate configuration and mapping XML documents, resolves them and loads the details into built-in Hibernate objects.

Since this step is loading of configurations, it generally happens at the application initialization time.

```
Configuraton cfg = new Configuration ( )
      cfg.configure ( )
```

- When new Configuration () is called, Hibernate searches for a file named hibernate.properties in the root of the classpath. If it has found, all hibernate.* properties are loaded and added to the Configuration object.
- When configure () is called, Hibernate searches for a file named hiber-nate.cfg.xml in the root of the classpath, and an exception is thrown if it can't be found. You don't have to call this method if you don't have this con-figuration file, of course. If settings in the XML configuration file are duplicates of properties set earlier, the XML settings override the previous ones.

SessionFactory

- The org.hibernate.SessionFactory interface provides an abstraction for the application to obtain Hiber-nate Session objects.
- Creating a SessionFactory object involves a huge process that includes the following operations:
  - Start the cache (second level)
  - Initialize the identifier generators
  - Pre-compile and cache the named queries (HQL and SQL

- Obtain the JTA Transaction Manager

As Session factory is a heavy weight object, creation of the session factory object is an expensive operation and recommended to get it created at application start up.

Note: It is generally recommended to use single SessionFactory per JVM instance. There is no problem in using a single SessionFactory for an application with multiple threads also.

```
  // loads configuration and creates a session factory
Configuration configuration = new Configuration().configure();
ServiceRegistryBuilder registry = new ServiceRegistryBuilder();
registry.applySettings(configuration.getProperties());
ServiceRegistry serviceRegistry = registry.buildServiceRegistry();
SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);

  //business logic goes here
```

Note: In hibernate 3.x, 4.x and 5.x versions have different ways for SessionFactory creation.

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. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed. The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes.

```
Session hsession = sessionFactory.openSession();
            // logic goes here
            hsession.close ( );
```

Transaction

The transaction object specifies the atomic unit of work. The org.hibernate.Transaction interface provides methods for transaction management.

```
Transaction tx = session.beginTransaction();//start
              //operations goes here
            tx.commit(); //end
```

Setting up the Project

In some projects, developers analyzing the business domain in object-oriented terms drive the development of an application. In others, an existing relational data model heavily influences it: either a legacy database or a brand-new schema designed by a professional data modeler. There are many choices to be made, and the following questions need to be answered before you can start:

Can you start from scratch with a clean design of a new business requirement, or is legacy data and/or legacy application code present?

Can some of the necessary pieces be automatically generated from an existing artifact (for example, Java source from an existing database schema)? Can the database schema be generated from Java code and Hibernate Mapping? Metadata?

What kind of tool is available to support this work? What about other tools to support the full development cycle?

We will discuss these questions in the following sections as we set up a basic Hibernate project. This is our road map:

1) **Select a development process**
2) **Set up the project infrastructure**
3) **Write application code and mappings**
4) **Configure and start Hibernate**
5) **Run the application**

Hibernate Tools for Eclipse IDE—

The Hibernate Tools are plug-ins for the Eclipse IDE (part of the JBoss IDE for Eclipse—a set of wizards, editors, and extra views in Eclipse that help you develop EJB3, Hibernate, JBoss Seam, and other Java applications based on JBoss middleware). The features for forward and reverse engineering are equivalent to the Ant-based tools.

The additional Hibernate Console view allows you to execute adhoc Hibernate queries (HQL and Criteria) against your database and to browse the result graphically. The Hibernate Tools XML editor supports automatic completion of mapping files, including class, property, and even table and column names.

The following development scenarios are common:

■ **Top down**: in top-down development, you start with an existing domain model, its implementation in Java, and (ideally) complete freedom with respect to the database schema. You must create mapping metadata— either with XML files or by annotating the Java source—and then optionally let Hibernate' s hbm2ddl tool generate the database schema. In the absence of an existing database schema, this is the most comfortable development Style for most Java developers. You may even use the Hibernate Tools to automatically refresh the database schema on every application restart in development.

■ **Bottom up**: Conversely, bottom-up development begins with an existing database schema and data model. In this case, the easiest way to proceed is to use the reverse-engineering tools to extract metadata from the database. This metadata can be used to generate XML mapping files, with hbm2hbmxml for example. With hbm2java, the Hibernate mapping metadata is used to generate Java persistent classes, and even data access objects—in other words, a skeleton for a Java persistence layer. Or, instead of writing to XML mapping files, annotated Java source code (EJB 3.0 entity classes) can be produced directly by the tools. However, not all class association details and Java-specific metainformation can be automatically generated from an SQL database schema with this strategy, so expect some manual work.

■ **Middle out**: The Hibernate XML mapping metadata provides sufficient information to completely deduce the database schema and to generate the Java source code for the persistence layer of the application. Furthermore, the XML mapping document isn't too verbose. Hence, some architects and developers prefer middle-out development, where they begin with handwritten Hibernate XML mapping files, and then generate the database schema using hbm2ddl and Java classes using hbm2java. The Hibernate XML mapping files are constantly updated during development, and other artifacts are generated from this master definition. Additional business logic or database objects are added through sub classing and auxiliary DDL. This development style can be recommended only for the seasoned Hibernate expert.

■ **Meet in the middle**: The most difficult scenario is combining existing Java classes and an existing database schema. In this case, there is little that the Hibernate toolset can do to help. It is, of course, not possible to map arbitrary Java domain models to a given schema, so this scenario usually requires at least some refactoring of the Java classes, database schema, or both. The mapping metadata will almost certainly need to be written by hand and in XML files (though it might be possible to use annotations if there is a close match). This can be an incredibly painful scenario, and it is, fortunately, exceedingly rare.

We now explore the tools and their configuration options in more detail and set up a work environment for typical Hibernate application development.

Project setup

Hibernate can be downloaded from http://www.hibernate.org/downloads

Hibernate Application Requirements

- POJO class (Entity)
- Mapping XML or Annotations
- Configuration XML
- DAO class to write Persistence logic
- Test class for testing DAO class methods

Writing POJO classes

Developers have found entity beans to be tedious, unnatural, and unproductive. As a reaction against entity beans, many developers started talking about Plain Old Java Objects (POJOs), a back-to-basics approach that essentially revives JavaBeans, a component model for UI development, and reapplies it to the business layer. (Most developers are now using the terms POJO and JavaBean almost synonymously.)

```
import java.io.Serializable;                                    (1)

public class User implements Serializable {

        public String username;
        public Address address;

        public User() {         }                               (2)

        public String getUsername() {
                return username;
        }

        public void setUsername(String username) {
                this.username = username;
        }                                                       (3)

        public Address getAddress() {
                return address;
        }

        public void setAddress(Address address) {
                this.address = address;
        }

        public Double computeShippingCost(long zipCode) {
                // logic
        }                                                       (4)
}
```

User.java

1. Hibernate does not require that persistent classes implement Serializable. However, when objects are stored in an HttpSession or passed by value using RMI, serialization is necessary. (This is very likely to happen in a Hibernate application.)
2. Unlike the JavaBeans specification, which requires no specific constructor, Hibernate requires a constructor with no arguments for every persistent class. Hibernate instantiates persistent classes using Constructor.newInstance(), a feature of the Java reflection API. The constructor may be non-public, but it should be at least package-visible if runtime-generated proxies will be used for performance optimization
3. The properties of the POJO implement the attributes of our business entities—for example, the username of User. Properties are usually implemented as instance variables, together with property accessor methods: a method for retrieving the value of the instance variable and a method for changing its value. These methods are known as the getter and setter, respectively. Our example POJO declares getter and setter methods for the private username instance variable and also for address. The JavaBean specification defines the guidelines for naming these methods. The guidelines allow generic tools like Hibernate to easily discover and manipulate the property value. A getter method name begins with get, followed by the name of the property (the first letter in uppercase); a setter method name begins with set. Getter methods for Boolean properties may begin

with is instead of get. Hibernate does not require that accessors methods be declared public; it can easily use private accessors for property management. Some getter and setter methods do something more sophisticated than simple instance variables access (validation, for example). Trivial accessors' methods are common, however.

4. This POJO also defines a business method that calculates the cost of shipping an item to a particular user (we left out the implementation of this method).

Defining the mapping metadata

ORM tools require a metadata format for the application to specify the mapping between classes and tables, properties and columns, associations and foreign keys, Java types and SQL types. This information is called the object/relational mapping metadata. It defines the transformation between the different data type systems and relationship representations. It is our job as developers to define and maintain this metadata. We discuss various approaches in this section.

Metadata in XML

Any ORM solution should provide a human-readable, easily hand-editable mapping format, not only a GUI mapping tool. Currently, the most popular object/ relational metadata format is XML. Mapping documents written in and with XML are lightweight, are human readable, are easily manipulated by version-control systems and text editors, and may be customized at deployment time (or even at runtime, with programmatic XML generation).

However, is XML-based metadata really the best approach? A certain backlash against the overuse of XML can be seen in the Java community. Every framework and application server seems to require its own XML descriptors. In our view, there are three main reasons for this backlash:

■ Many existing metadata formats weren't designed to be readable and easy to edit by hand. In particular, a major cause of pain is the lack of sensible defaults for attribute and element values, requiring significantly more typing than should be necessary.

■ Metadata-based solutions were often used inappropriately. Metadata is not, by nature, more flexible or maintainable than plain Java code.

■ Good XML editors, especially in IDEs, are not as common as good Java coding environments. Worst, and most easily fixable, a document type declaration (DTD) often is not provided, preventing auto-completion and validation. Another problem are DTDs that are too generic, where every declaration is wrapped in a generic "extension" of "meta" element. There is no getting around the need for text-based metadata in ORM. However, Hibernate was designed with full awareness of the typical metadata problems. The metadata format is extremely readable and defines useful default values. When attribute values are missing, Hibernate uses reflection on the mapped class to help determine the defaults. Hibernate comes with a documented and complete DTD. Finally, IDE support for XML has improved lately, and modern IDEs provide dynamic XML validation and even an auto-complete feature. If that's not enoughfor you, in chapter 9 we demonstrate some tools that may be used to generate Hibernate XML mappings.

Hibernate mapping file

Hibernate mapping file is used by hibernate framework to get the information about the mapping of a POJO class and a database table.

It mainly contains the following mapping information

- Mapping information of a POJO class name to a database table name.
- Mapping information of POJO class properties to database table columns.

Elements of the Hibernate mapping file:

1. **hibernate-mapping:** It is the root element.
2. **Class:** It defines the mapping of a POJO class to a database table.
3. **Id:** It defines the unique key attribute or primary key of the table.
4. **generator:** It is the sub element of the id element. It is used to automatically generate the id.
5. **property:** It is used to define the mapping of a POJO class property to database table column.

```
<hibernate-mapping>
    <class name="POJO class name" table="table name in database">
        <id name="propertyName" column="columnName" type="propertyType">
            <generator class="generatorClass" />
        </id>
        <property name="propertyName1" column="colName1" type="propertyType " />
        <property name="propertyName2" column="colName2" type="propertyType " />
        ….
    </class>
</hibernate-mapping>
```

Hibernate configuration file

Hibernate works as an intermediate layer between java application and relational database. So hibernate needs some configuration setting related to the database and other parameters like mapping files.

A hibernate configuration file mainly contains three types of information

- Connection Properties related to the database.
- Hibernate Properties related to hibernate behavior.
- Mapping files entries related to the mapping of a POJO class and a database table.

Note: No. of hibernate configuration files in an application is dependence upon the no. of database uses. No. of hibernate configuration files are equals to the no. of database uses.

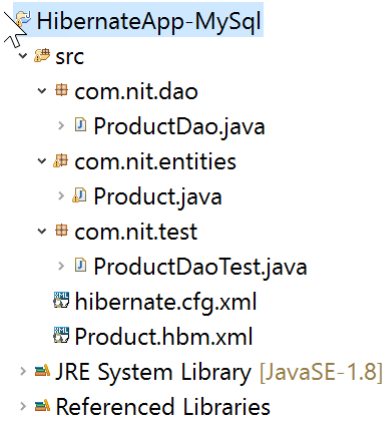
```
<hibernate-configuration>
    <session-factory>
        // Connection Properties
        <property name="connection.driver_class">driverClassName</property>
        <property name="connection.url">jdbcConnectionURL</property>
        <property name="connection.user">databaseUsername</property>
        <property name="connection.password">databasePassword</property>
        // Hibernate Properties
        <property name="show_sql">true/false</property>
        <property name="dialet">databaseDialetClass</property>
        <property name="hbm2ddl.auto">like create/update</property>
        // Mapping files entries
        <mapping resource="mappingFile1.xml" />
        <mapping resource="mappingFile2.xml" />
    </session-factory>
</hibernate-configuration>
```

Hibernate First Application

To create Hibernate application we need to create below files

- Product.java (POJO class)
- Product.hbm.xml (Xml mapping file)
- hibernate.cfg.xml (Xml configuration file)
- ProductDao.java (java file to write our persistence logic)
- ProductDaoTest.java (To test Dao class functionality)

Step 1: Create a Java Project in Eclipse IDE and add Hibernate Jars and JDBC Driver jar file to project build path



Step 2: Create POJO class. This class is used to hold the data

```
public class Product {

	private Integer productId;
	private String proName;
	private Double price;

	//Setters and getters

}
```

Step 3: Create Mapping xml file. This file used to map java class to database table

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
	<class name="Products" table="PRODUCTS">
		<id name="productId" type="int" column="PID">
		</id>
		<property name="procName" column="PRODUCT_NAME" />
		<property name="price" column="PRICE" />
	</class>
</hibernate-mapping>
```

Step 4: Create hibernate configuration file. This file is used to configure database properties, hibernate properties and hbm files or annotated classes.

```
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
	<session-factory>
		<!-- Related to Database information -->
		<property name="hibernate.connection.driver_class">
			oracle.jdbc.driver.OracleDriver
		</property>

		<property name="hibernate.connection.url">
			oracle:jdbc:thin:@localhost:1521/XE
		</property>
		<property name="hibernate.connection.username">
			hibernate
		</property>
		<property name="hibernate.connection.password">
			hibernate@123
		</property>

		<!-- Related to Hibernate Properties -->
		<property name="show_sql">true</property>
		<property name="dialect">org.hibernate.dialect.OracleDialect</property>

		<!-- List of XML mapping files -->
		<mapping resource="Prodcut.hbm.xml" />
	</session-factory>
</hibernate-configuration>
```

Step 5: Create ProductDao.java class. This class is used to perform persistence operations

```
public class ProductDao {

	public boolean insert(Product entity) {
		boolean isInserted = false;
		try {
			Configuration cfg = new Configuration().configure();
			SessionFactory sf = cfg.buildSessionFactory();
			Session hs = sf.openSession();
			Transaction tx = hs.beginTransaction();
			Serializable ser = hs.save(entity);
			if (ser != null) {
				isInserted = true;
			}
			tx.commit();
			hs.close();
			sf.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
		return isInserted;
	}
}
```

ProductDao.java

Step 6: Create ProductDaoTest.java to test Dao class method

```
public class ProductDaoTest {
	public static void main(String[] args) {

		// Setting data to Entity object
		Product entity = new Product();
		entity.setPid(1001);
		entity.setName("Keyboard");
		entity.setPrice(14500.00);

		// Calling Dao method to persist entity data
		ProductDao dao = new ProductDao();
		boolean isInserted = dao.insert(entity);
		System.out.println("Record inserted : " + isInserted);
	}
}
```

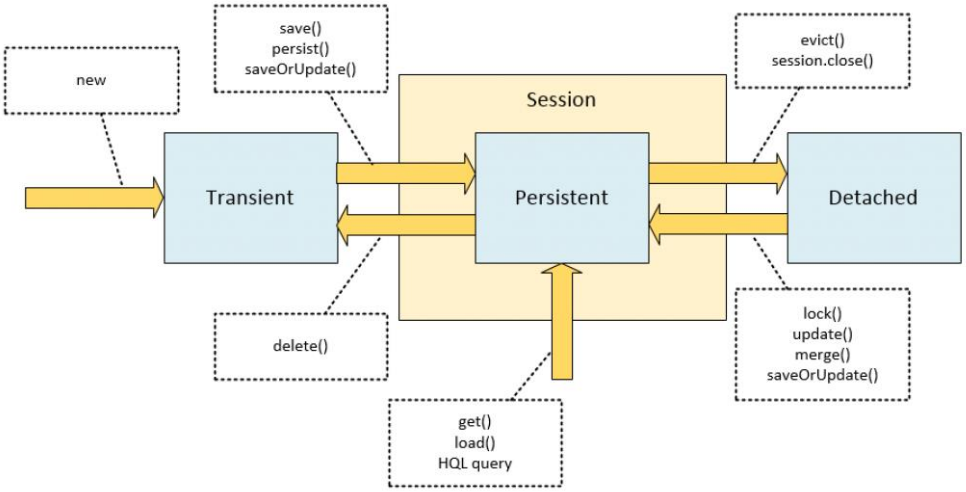
ProductDaoTest.java

Lifecycle of POJO Class Objects

Object states in Hibernate plays a vital role in the execution of code in an application. Hibernate has provided three different states for an object of a pojo class. These three states are also called as life cycle states of an object.

There are three types of Hibernate object states.

1. Transient Object State:
In Hibernate, objects instantiated using the new operator are not immediately persistent. Their state is transient, which means they are not associated with any database table row, and so their state is lost as soon as they are dereferenced (no longer referenced by any other object) by the application. These objects have a lifespan that effectively ends at that time, and they become inaccessible and available for garbage collection.



Hibernate considers all transient instances to be no transactional; a modification to the state of a transient instance is not made in the context of any transaction. This means Hibernate does not provide any rollback functionality for transient objects. (In fact, Hibernate does not roll back any object changes, as you will see later.)

Objects that are referenced only by other transient instances are, by default, also transient. For an instance to transition from transient to persistent state requires either a save () call to the persistence manager or the creation of a reference from an already persistent instance.

2. Persistent Object State
An object that is associated with the hibernate session is called as Persistent object. When the object is in persistent state, then it represent one row of the database and consists of an identifier value. You can make a transient instance persistent by associating it with a Session.

3. Detached Object State
Object which is just removed from hibernate session is called as detached object. The state of the detached object is called as detached state. When the object is in detached sate then it contain identity but you cannot do persistence operation with that identity.

Any changes made to the detached objects are not saved to the database. The detached object can be reattached to the new session and save to the database using update, saveOrUpdate and merge methods.

When the Session is closed, this entire subgraph (all objects associated with a persistence manager) becomes detached.

CURD Operations in Hibernate

A CRUD operation deals with creating, retrieving, updating and deleting records from the table. We are going to discuss about 4 main functionality:

1. Create a record
2. Read a record
3. Update a Record
4. Delete a Record

Hibernate Session is the interface between java application and hibernate framework. Session in hibernate framework provided several methods to perform these operations.

Creating / Inserting Record: To create / insert a record, we have 2 methods in hibernate. They are save and persist methods.

```
public Serializable save ( Object entity )

public void persist ( Object entity )
```

Here is the difference between save and persist method:

1. First difference between save and persist is there return type. The return type of persist method is void while return type of save method is Serializable object. But both of them also INSERT records into database
2. Another difference between persist and save is that both methods make a transient object to persistent state. However, persist () method doesn't guarantee that the identifier value will be assigned to the persistent state immediately, the assignment might happen at flush time.
3. Third difference between save and persist method in Hibernate is behavior on outside of transaction boundaries. persist () method will not execute an insert query if it is called outside of transaction boundaries. Because save () method returns an identifier so that an insert query is executed immediately to get the identifier, no matter if it are inside or outside of a transaction.
4. Fourth difference between save and persist method in Hibernate: persist method is called outside of transaction boundaries, it is useful in long-running conversations with an extended Session context. On the other hand save method is not good in a long-running conversation with an extended Session context.

Retrieving Record: To retrieve a record we have 2 methods in hibernate. They are get and load methods.

1) Loading an hibernate entity using session.load () method

Hibernate' s Session interface provides several load () methods for loading entities from your database. Each load () method requires the object's primary key as an identifier, and it is mandatory to provide it. In addition to the ID, Hibernate also needs to know which class or entity name to use to find the object with that ID. After the load () method returns, you need to cast the returned object to suitable type of class to further use it.

Let us look at different flavors of load () method available in hibernate session:

```
1. public Object load(Class theClass, Serializable id) throws HibernateException
2. public Object load(String entityName, Serializable id) throws HibernateException
3. public void load(Object object, Serializable id) throws HibernateException
```

```
public class RetriveProduct {
	public static void main(String[] args) {
		Configuration cfg = new Configuration()
		cfg.configure("hibernate.cfg.xml");
		StandardServiceRegistryBuilder serviceRegistryBuilder = new StandardServiceRegistryBuilder();
		serviceRegistryBuilder.applySettings(configuration.getProperties());
		ServiceRegistry serviceRegistry = serviceRegistryBuilder.build();
		SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);
		Session hsession = sessionFactory.openSession();
		Transaction tx = hsession.beginTransaction();

		Object obj =  hsession.load(Product.class,new Integer(1));
		Prodcut p = (Product) obj;
		System.out.println(p);

		tx.commit();
		hsession.close();
		sessionFactory.close();
	}
}
```

2) Loading an hibernate entity using session.get () method: The get () method is very much similar to load() method. The get() methods take an identifier and either an entity name or a class. There are also two get() methods that take a lock mode as an argument, but we will discuss lock modes later. The rest get() methods are as follows:

1. **public Object get(Class clazz, Serializable id) throws HibernateException**
2. **public Object get(String entityName, Serializable id) throws HibernateException**

The get() method is special because the identifier uniquely identifies a single instance of a class. Hence it's common for applications to use the identifier as a convenient handle to a persistent object. Retrieval by identifier can use the cache when retrieving an object, avoiding a database hit if the object is already cached.

User user = (User) session.load(User.class, userID);

The load() method is older; get() was added to Hibernate's API due to user request. The difference is trivial:

- If load () cannot find the object in the cache or database, an exception is thrown. The load () method never returns null. The get () method returns null if the object cannot be found.
- The load () method may return a proxy instead of a real persistent instance. A proxy is a placeholder that triggers the loading of the real object when it is accessed for the first time; we discuss proxies later in this section. On the other hand, get () never returns a proxy.
- Choosing between get () and load () is easy: If you are certain the persistent object exists, and nonexistence would be considered exceptional, load () is a good option. If you aren't certain there is a persistent instance with the given identifier, use get () and test the return value to see if it's null. Using load () has a further implication: The application may retrieve a valid reference (a proxy) to a persistent instance without hitting the database to retrieve its persistent state. So load () might not throw an exception when it does not find the persistent objecting the cache or database; the exception would be thrown later, when the proxy is accessed.

Updating Record: To update entity in the database we have two methods in hibernate

```
public void update(Object obj)

public void saveOrUpdate(Object obj)
```

```
public class UpdateProduct {

      public static void main(String[] args) {
            //Get the SessionFactory Object
            //Get the Session Object
      Product p =  (Product)hsession.get(Product.class,new Integer(1));
            p.setProductId(101);
            p.setPrice(35000.89);
            hsession.update(p);
            //close operations
      }
}
```

Update: Update method in the hibernate is used for updating the object using identifier (primary key). If the identifier is missing or doesn't exist, it will throw exception**.**

SaveOrUpdate: This method calls save () or update () based on the operation. If the identifier exists, it will call update method else the save method will be called.

Deleting Record: To delete a record we have delete method

```
public class DeleteProduct {
      public static void main(String[] args) {
            //Get the SessionFactory Object
            //Get the Session Object
            //Begin Transaction
            Product p =  new Product( );
            p.setProductId(101);
            hsession.delete(p);
            //commit the transaction
            //close operations
      }
}
```

Note: When we call delete method, first hibernate executes select query with given identifier. If record is available then it will execute delete query otherwise delete query will not be executed.

Auto DDL in Hibernate

One of the major advantage of hibernate is Auto DDL support. hibernate.hbm2ddl.auto Automatically validates or exports schema DDL to the database when the SessionFactory is created. To enable this feature we have to configure "**hibernate.hbm2ddl.auto"** property in Hibernate configuration file.

For this hbm.2ddl.auto property we can use below 4 values

validate: validate the schema, makes no changes to the database.
update: update the schema.
create: creates the schema, destroying previous data.
create-drop :drop the schema at the end of the session.

Configure this property in hibernate.cfg.xml file like below

```
<property name="hibernate.hbm2ddl.auto">create</property>
<property name="hibernate.hbm2ddl.auto">validate</property>
<property name="hibernate.hbm2ddl.auto">update</property>
<property name="hibernate.hbm2ddl.auto">create-drop</property>
```

Enable Logging and Commenting

Hibernate can output the underlying SQL behind your HQL queries into your application's log file. This is especially useful if the HQL query does not give the results you expect, or if the query takes longer than you wanted. This is not a feature you will have to use frequently, but it is useful should you have to turn to your database administrators for help in tuning your Hibernate application.

1) Logging: The easiest way to see the SQL for a Hibernate HQL query is to enable SQL output in the logs with the "show_sql" property. Set this property to true in your hibernate.cfg.xml configuration file and Hibernate will output the SQL into the logs. When you look in your application's output for the Hibernate SQL statements, they will be pre-fixed with "Hibernate".

```
<property name="hibernate.show_sql"> true <property>
```

2) Commenting: Tracing your HQL statements through to the generated SQL can be difficult, so Hibernate provides a commenting facility on the Query object that lets you apply a comment to a specific query. The Query interface has a setComment() method that takes a String object as an argument, as follows:

```
public Query setComment(String comment)
```

Hibernate will not add comments to your SQL statements without some additional configuration, even if you use the setComment() method. You will also need to set a Hibernate property, hibernate.use_sql_comments, to true in your Hibernate configuration. If you set this property but do not set a comment on the query programmatically, Hibernate will include the HQL used to generate the SQL call in the comment. I find this to be very useful for debugging HQL.

Use commenting to identify the SQL output in your application's logs if SQL logging is enabled.

Singleton Design Pattern

Singleton pattern will ensure that there is only one instance of a class is createdd. It is used to provide global point of access to the object

Singleton class rules

- Static member: This contains the instance of the singleton class.
- Private constructor: This will prevent anybody else to instantiate the Singleton class.
- Static public method: This provides the global point of access to the Singleton object and returns the instance to the client calling class

- Override clone method to avoid cloning
- Override readResolve method to avoid de-serialization

```
import java.io.Serializable;

public class DateFormatter implements Cloneable, Serializable {

	private static DateFormatter instance = null;

	private DateFormatter() {
	}

	public static DateFormatter getInstance() {
		if (instance == null) {
			synchronized (DateFormatter.class) {
				if (instance == null) {
					instance = new DateFormatter();
				}
			}
		}
		return instance;
	}

	@Override
	protected Object clone() throws CloneNotSupportedException {
		try {
			throw new Exception("Cloning not supported..!!");
		} catch (Exception e) {
			e.printStackTrace();
		}
		return null;
	}

	protected Object readResolve() {
		return instance;
	}
}
```

DateFormatter.java

Now Create HibernateUtil class which provides one instance of SessionFactory, from now onwards we will get SessionFactory object from this class.

```
public class HibernateUtil {

        private static SessionFactory sessionFactory = buildSessionFactory();
        private static SessionFactory buildSessionFactory() {
           try {
              if (sessionFactory == null) {
                 Configuration configuration = new Configuration().configure(
                                                    HibernateUtil.class.getResource("/hibernate.cfg.xml"));
                 StandardServiceRegistryBuilder serviceRegistryBuilder = new StandardServiceRegistryBuilder();
                 serviceRegistryBuilder.applySettings(configuration.getProperties());
                 ServiceRegistry serviceRegistry = serviceRegistryBuilder.build();
                 sessionFactory = configuration.buildSessionFactory(serviceRegistry);
              }
              return sessionFactory;
           } catch (Throwable ex){
              System.err.println("Initial SessionFactory creation failed." + ex);
              throw new ExceptionInInitializerError(ex);
           }
        }

        public static SessionFactory getSessionFactory(){
           return sessionFactory;
        }

        public static void close(){
           getSessionFactory().close();
        }

}
```

Hibernate Bootstrapping

The term bootstrapping refers to initializing and starting a software component. In Hibernate, we are specifically talking about the process of building a fully functional `SessionFactory` instance.

Native Bootstrapping (3.x)

To create a SessionFactory we can define the configuration in hibernate.properties or hibernate.cfg.xml or create it programmatically. In this example we'll use the hibernate.cfg.xml configuration file, which is mostly used when creating Hibernate application.

```
<?xml version="1.0" encoding="UTF-8"?>
<hibernate-configuration>
        <session-factory>
                <property name="hibernate.connection.driver_class">
                        oracle.jdbc.driver.OracleDriver
                </property>
                <property name="hibernate.connection.url">
                        jdbc:oracle:thin:@localhost:1521/XE
                </property>
                <property name="hibernate.connection.username">
                        system
                </property>
                <property name="hibernate.connection.password">
                        admin
                </property>

                <property name="dialect">org.hibernate.dialect.OracleDialect</property>
                <property name="show_sql">true</property>
                <property name="format_sql">true</property>

                <!-- HBM files or Annotated Classes >
        </session-factory>
</hibernate-configuration>
```

hibernate.cfg.xml

Now we have the configuration file done, let's create a helper class that will configure and build the SessionFactory object. This helper will be used in other Hibernate examples in this material.

```
public class HibernateUtils {

        private HibernateUtils() {  }
        private static SessionFactory sf = null;
        public static Session getSession() {

                if (sf == null) {
                        Configuration cfg = new Configuration();
                        cfg.configure();
                        sf = cfg.buildSessionFactory();
                }
                return sf.openSession();
        }
}
```

HibernateUtils.java

Hibernate 4.x Boot strapping

Services and Registries are new as a formalized concept starting in 4.0. But the functionality provided by the different Services have actually been around in Hibernate much, much longer. What is new is managing them, their lifecycles and dependencies through a lightweight, dedicated container we call a ServiceRegistry.

What is a Service?

Services provide various types of functionality, in a pluggable manner. Specifically they are interfaces defining certain functionality and then implementations of those service contract interfaces. The interface is known as the service role; the implementation class is known as the service implementation. The pluggability comes from the fact that the service implementation adheres to contract defined by the interface of the service role and that consumers of the service program to the service role, not the implementation.

Note : All Services are expected to implement the org.hibernate.service.Service "marker" interface. Hibernate uses this internally for some basic type safety; it defines no methods (at the moment).

Let's look at an example to better define what a Service is. Hibernate needs to be able to access JDBC Connections to the database. The way it obtains and releases these Connections is through the ConnectionProvider service. The service is defined by the interface (service role) org.hibernate.engine.jdbc.connections.spi.ConnectionProvider which declares methods for obtaining and releasing the Connections. There are then multiple implementations of that service contract, varying in how they actually manage the Connections:

- org.hibernate.engine.jdbc.connections.internal.DatasourceConnectionProviderImpl for using a javax.sql.DataSource
- org.hibernate.c3p0.internal.C3P0ConnectionProvider for using a C3P0 Connection pool etc.

Internally Hibernate always references org.hibernate.engine.jdbc.connections.spi.ConnectionProvider rather than specific implementations in consuming the service (we will get to producing the service later when we talk about registries). Because of that fact, other ConnectionProvider service implementations could be plugged in.

There is nothing revolutionary here; programming to interfaces is generally accepted as good programming practice. What's interesting is the ServiceRegistry and the pluggable swapping of the different implementors.

What is a ServiceRegistry?

A ServiceRegistry, at its most basic, hosts and manages Services. Its contract is defined by the org.hibernate.service.ServiceRegistry interface. Services have a lifecycle, they have a scope and Services might depend on other services. And they need to be produced (choose using one implementation over another). The ServiceRegistry fulfills all these needs.

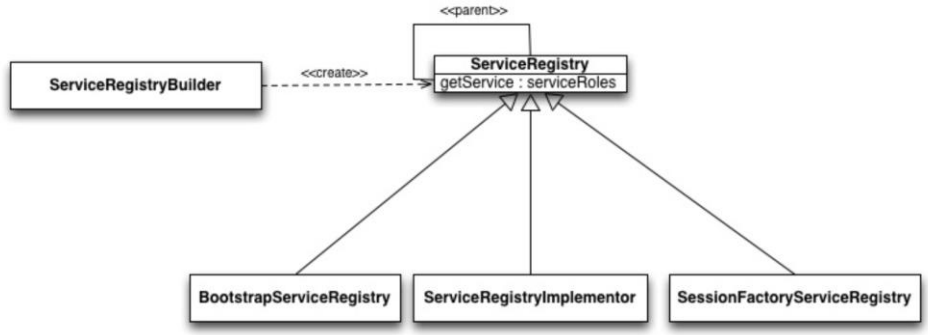
In a concise definition, the ServiceRegistry acts as a inversion-of-control (IoC) container.

Why not just use an existing IoC framework? The main reason was that this had to be as light-weight and as small of a footprint as possible. The initial design also had called for Services to be swappable at runtime, which unfortunately had to be removed due to performance problems in the proxy-based solution to swapping (the plan is to investigate alternate ways to achieve swap-ability with better performance at a later date).

A Service is associated with a ServiceRegistry. The ServiceRegistry scopes the Service. The ServiceRegistry manages the lifecycle of the Service. The ServiceRegistry handles injecting dependencies into the Service (actually both a pull and a push/injection approach are supported). ServiceRegistries are also hierarchical, meaning a ServiceRegistry can have a parent ServiceRegistry. Services in one registry can depend on and utilize services in that same registry as well as any parent registries.

Currently Hibernate utilizes 3 different ServiceRegistry implementations forming a hierarchy.

1. BootstrapServiceRegistry
2. StandardServiceRegistry
3. SessionFactoryServiceRegistry org.hibernate.service.spi.SessionFactoryServiceRegistry is the 3rd standard Hibernate ServiceRegistry. Typically, its parent registry is the StandardServiceRegistry. SessionFactoryServiceRegistry is designed to hold Services which need access to the SessionFactory. Currently that is just 3 Services.



Below is the sample application to demonstrate Hibernate 4.x Bootstrapping with Serviceregistry

```
public class HibernateUtil {

        private static SessionFactory sf = null;

        private HibernateUtil() {
        }

        public static Session buildSession() {
                if (sf == null) {
                        Configuration cfg = new Configuration();
                        cfg.configure();
                        ServiceRegistry registry =
                                        new ServiceRegistryBuilder()
                                        .applySettings(cfg.getProperties())
                                        .buildServiceRegistry();
                        sf = cfg.buildSessionFactory(registry);
                }
                return sf.openSession();
        }
}
```

HibernateUtil.java

```java
public class RobotDao {
	public boolean insert(Robot entity) {
		boolean isInserted = false;
		Session hs = null;
		Transaction tx = null;
		try {
			hs = HibernateUtil.buildSession();
			tx = hs.beginTransaction();
			Serializable id = hs.save(entity);
			if (id != null)
				isInserted = true;
			tx.commit();
		} catch (Exception e) {
			e.printStackTrace();
		}
		return isInserted;
	}
}
```

RobotDao.java

```java
@Entity
public class Robot {

	@Id
	@GeneratedValue
	private Integer robotId;
	private String name;
	private String type;

	/setters && getters
}
```

Robot.java

```java
public class RobotDaoTest {
	public static void main(String[] args) {

		Robot r = new Robot();
		r.setName("Chitti-2.0");
		r.setType("Entertainement");

		RobotDao dao = new RobotDao();
		boolean isInserted = dao.insert(r);
		System.out.println("Record Inserted ?:"+isInserted);
	}
}
```

RobotDaoTest.java

Custom Service Configuration

Instead of depending on default ConnectionProvider service, we can register third party service. Below class serves as custom connection provider class

```java
public class CustomConnProviderService implements ConnectionProvider {

	@Override
	public boolean isUnwrappableAs(Class arg0) {
		return false;
	}

	@Override
	public <T> T unwrap(Class<T> arg0) {
		return null;
	}

	@Override
	public void closeConnection(Connection con) throws SQLException {
		if (con != null) con.close();
	}

	@Override
	public Connection getConnection() throws SQLException {
		try {
			Class.forName("oracle.jdbc.driver.OracleDriver");
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}
		Connection con = DriverManager.getConnection(
		"jdbc:oracle:thin:@locahost:1521/XE", "system", "admin");
		return con;
	}

	@Override
	public boolean supportsAggressiveRelease() {
		return false;
	}
}
```

CustomConnProviderService.java

Registering CustomConnectionProvider service using addService( ) method in ServiceRegistry

```
public class HibernateUtil {

        private static SessionFactory sf = null;

        private HibernateUtil() {
        }

        public static Session buildSession() {
                if (sf == null) {
                        Configuration cfg = new Configuration();
                        cfg.configure();
                        ServiceRegistry registry =
                                        new ServiceRegistryBuilder()
                                        .applySettings(cfg.getProperties())
                                        .addService(ConnectionProvider.class, new CustomConnProviderService())
                                        .buildServiceRegistry();
                        sf = cfg.buildSessionFactory(registry);
                }
                return sf.openSession();
        }
}
```

HibernateUtil.java

Hibernate 5.x Bootstrapping

Method of creating SessionFactory object is changed significantly in Hibernate 5. Following code was used in Hibernate 4 for creating the SessionFactory object:

```
public class HibernateUtil {

        private static SessionFactory sf = null;

        private HibernateUtil() {
        }

        public static Session buildSession() {
                if (sf == null) {
                        StandardServiceRegistry registry =
                                        new StandardServiceRegistryBuilder()
                                        .configure("hibernate.cfg.xml")
                                        .build();
                        MetadataSources sources = new MetadataSources(registry);
                        Metadata m = sources.getMetadataBuilder(registry).build();
                        sf = m.getSessionFactoryBuilder()
                                .build();
                }
                return sf.openSession();
        }
}
```

HibernateUtil.java

Understanding Object Identity

It is vital to understand the difference between object identity and object equality before we discuss terms like database identity and how Hibernate manages identity.

Identity versus equality

Java developers understand the difference between Java object identity and equality. Object identity, ==, is a notion defined by the Java virtual machine. Two object references are identical if they point to the same memory location. On the other hand, object equality is a notion defined by classes that implement the equals () method, sometimes also referred to as equivalence. Equivalence means that two different (non-identical) objects have the same value. Two different instances of String are equal if they represent the same sequence of characters, even though they each have their own location in the memory space of the virtual machine. (We admit that this is not entirely true for Strings, but you get the idea.)

Persistence complicates this picture. With object/relational persistence, a persistent object is an in-memory representation of a particular row of a database table. Therefore, along with Java identity (memory location) and object equality, we pick up database identity (location in the persistent data store). We now have three methods for identifying objects:

■ **Object identity—**Objects are identical if they occupy the same memory location in the JVM. This can be checked by using the == operator.

■ **Object equality—**Objects are equal if they have the same value, as defined by the equals (Object o) method. Classes that do not explicitly override this method inherit the implementation defined by java.lang.Object, which compares object identity.

■ **Database identity—**Objects stored in a relational database are identical if they represent the same row or, equivalently, share the same table and primary key value. You need to understand how database identity relates to object identity in Hibernate.

Database identity with Hibernate

Hibernate exposes database identity to the application in two ways:

- **The value of the identifier property of a persistent instance**
- **The value returned by session.getIdentifier (Object o)**

The identifier property is special: Its value is the primary key value of the database row represented by the persistent instance. We don't usually show the identifier property in our domain model—it's a persistence-related concern, not part of our business problem. In our examples, the identifier property is always named id. So if myCategory is an instance of Category, calling myCategory.getId() returns the primary key value of the row represented by myCategory in the database. Should you make the accessor methods for the identifier property private scope or public? Well, database identifiers are often used by the application as a convenient handle to a particular instance, even outside the persistence layer. For example, web applications often display the results of a search screen to the user as a list of summary information. When the user selects a particular element, the application might need to retrieve the selected object. It's common to use a lookup by identifier for this purpose—you've probably already used identifiers this way, even in applications using direct JDBC. It's therefore usually appropriate to fully expose the database identity with a public identifier property accessor. On the other hand, we usually declare the setId() method private and let Hibernate generate and set the identifier value. The exceptions to this rule are classes with natural keys, where the value of the identifier is assigned by the application before the object is made persistent, instead of being generated by Hibernate. (We discuss natural keys in the next section.) Hibernate doesn't allow you to change the identifier value of a persistent instance after its first assigned.

Remember, part of the definition of a primary key is that its value should never change. Let's implement an identifier property for the Category class:

```
public class Category {
private Long id;
...
public Long getId() {
return this.id;
}
private void setId(Long id) {
this.id = id;
}...
}
```

The property type depends on the primary key type of the CATEGORY table and the Hibernate mapping type. This information is determined by the <id> element in the mapping document:

```
<class name="Category" table="CATEGORY">
  <id name="id" column="CATEGORY_ID" type="long">
        <generator class="native"/>
  </id>
...
</class>
```

The identifier property is mapped to the primary key column CATEGORY_ID of the table CATEGORY. The Hibernate type for this property is long, which maps to a BIGINT column type in most databases and which has also been chosen to match the type of the identity value produced by the native identifier generator. (We discuss identifier generation strategies in the next section.) So, in addition to operations for testing Java object identity (a == b) and object equality (a.equals(b)), you may now use a.getId().equals(b.getId()) to test database identity.

An alternative approach to handling database identity is to not implement any identifier property, and let Hibernate manage database identity internally. In this case, you omit the name attribute in the mapping declaration:

```
<id column="CATEGORY_ID">
      <generator class="native"/>
</id>
```

Hibernate will now manage the identifier values internally. You may obtain the identifier value of a persistent instance as follows:

```
Long catId = (Long) session.getIdentifier(category);
```

This technique has a serious drawback: You can no longer use Hibernate to manipulate detached objects effectively (see chapter 4, section 4.1.6, “Outside the identity scope”). So, you should always use identifier properties in Hibernate. (If you don’t like them being visible to the rest of your application, make the accessor methods private.) Using database identifiers in Hibernate is easy and straightforward. Choosing a good primary key (and key generation strategy) might be more difficult. We discuss these issues next.

Choosing primary keys

You have to tell Hibernate about your preferred primary key generation strategy. First, let us define primary key. The candidate key is a column or set of columns that uniquely identifies a specific row of the table. A candidate key must satisfy the following properties:

1. **The value or values are never null.**
2. **Each row has a unique value or values.**
3. **The value or values of a particular row never change.**

For a given table, several columns or combinations of columns might satisfy these properties. If a table has only one identifying attribute, it is by definition the primary key. If there are multiple candidate keys, you need to choose between them (candidate keys not chosen, as the primary key should be declared as unique keys in the database). If there are no unique columns or unique combinations of columns, and hence no candidate keys, then the table is by definition not a relation as defined by the relational model (it permits duplicate rows), and you should rethink your data model.

Many legacy SQL data models use natural primary keys. A natural key is a key with business meaning: an attribute or combination of attributes that is unique by virtue of its business semantics. Examples of natural keys might be a U.S. Social Security Number or Australian Tax File Number. Distinguishing natural keys is simple: If a candidate key attribute has meaning outside the database context, it is a natural key, whether or not it is automatically generated.

Experience has shown that natural keys usually cause problems in the long run. A good primary key must be unique, constant, and required (never null or unknown). Very few entity attributes satisfy these requirements, and some that do are not efficiently indexable by SQL databases. In addition, you should make certain that a candidate key definition could never change throughout the lifetime of the database before promoting it to a primary key. Changing the definition of a primary key and all foreign keys that refer to it is a frustrating task. For these reasons, we strongly recommend that new applications use synthetic identifiers (also called surrogate keys). Surrogate keys have no business meaning— they are unique values generated by the database or application. There are a number of well-known approaches to surrogate key generation.

Hibernate has several built-in identifier generation strategies.

Generators in Hibernate

- ✓ The <generator> element helps define how to generate a new primary key for a new instance of the class.
- ✓ The <generator> element accepts a java class name that will be used to generate unique identifiers for instances of the persistent class.
- ✓ The <generator> element is the child element of <id> element.

- Hibernate provides different primary key generator algorithms.
- All hibernate generator classes implements hibernate.id.IdentifierGenerator interface, and overrides the generate (SessionImplementor, Object) method to generate the 'identifier or primary key value'.
- If we want our own user defined generator, then we should implement IdentiferGenerator interface and override the generate()
- <generator /> tag (which is sub element of <id /> tag) is used to configure generator class in mapping file.

All generators implement the interface org.hibernate.id.IdentifierGenerator. This is a very simple interface; some applications may choose to provide their own specialized implementations. However, Hibernate provides a range of built-in implementations. There are shortcut names for the built-in generators:

Hibernate provided below generators

1) Assigned
2) Increment
3) Identity
4) Native
5) Sequence
6) Hilo
7) Seqhilo
8) Guid
9) Uuid
10) Select
11) Foreign

Assigned:

This algorithm uses programmer assigned value in primary key filed member variable of given pojo class object as identity value of the object

For every identity value generator algorithm, there will be a class and shortcut name. The class name of assigned algorithm is "org.hibernate.id.Assigned" and short cut name is assigned. All the classes that are acting as identity value generators implement a common interface called "org.hibernate.id.IdentifierGenerator".

```
<id name="customerId" column="CUSTOMER_ID">
        <generator class="org.hibernate.id.Assigned" or "assigned"/>
</id>
```

Increment:

This algorithm generated identity values of type long or short or int that are unique. This algorithm works with all database software but can generate only numeric values as identity values. The class name of this algorithm is "org.hibernate.id.IncrementGenerator". This uses "(Max identity value + 1) as the formula to generate next identity value.

```
<id name="customerId" column="CUSTOMER_ID">
        <generator class="increment/>
</id>
```

Sequence: (doesn't work with mysql)

This algorithm uses sequence concept available in DB2, postgreSQL, oracle, SAPDB …, and it generates identity values of type long,short,int. the class name of this algorithm is "org.hibernate.id.SequenceGenerator". To specify explicitly created sequence in database software as input values to this algorithm, use <param> having parameter name & value.

The sequence algorithm works only with the above listed database software

Procedure:

i) Create sequence in underlying database software.

SQL> create sequence my_seq increment by 2;

ii) Configure sequence algorithm by passing the sequence name as parameter value to the algorithm

```
<id name="no" column="eid">
        <generator class="native">
            <param name="sequence">my_seq</param>
        </generator>
</id>
```

Identity:

This algorithm use identity value generated by identity columns available in DB2, mysql, SQL server, Sybase database software. This algorithm returns identifier value of type Long, short or int.

Identity column means the column that generates that is its next record value by sing "Max value+1" formula

Note: To make ordinary column as identity column in MySQL database software, we need to apply auto increment constraint on the table column. To apply increment constraint the column must be numeric data type column and the column should have primary key constraint.

The class name of this identifier algorithm is "org.hibernate.id.IdentityGenerator"

```
<id name="customerId" column="CUSTOMER_ID">
        <generator class="identity"/>
</id>
```

With respective to the above code the EID column of the table should act as identity column by having primary key, auto increment constraints.

Difference between increment and identity algorithm:

While working with increment algorithm the id field related table column need not to have any constraint [constraints are optional]. Whereas working with identity algorithm the id field related table column must having primary key, auto increment constraint.

Hibernate software generates increment algorithm based identity value.

The database software generates identity algorithm based identity value on identity column and hibernate software collects & uses that value.

Increment algorithm works with all database software

Identity algorithm works with only database software, which support identity column (mysql, DB2 …)

Native:

This algorithm is not having its own individual behavior. This algorithm is capable of picking up identity, sequence or hilo algorithm based on the capabilities of underlying database software.

Due to this, algorithm works with all database software. While working with this algorithm, passing parameters is optional. But we can pass all the regular prams of sequence, hilo algorithms.

When native algorithm is configured by taking mysql underlying database software then it looks to work with identity algorithm.

When native algorithm is used by taking oracle as the underlying database software then it use sequence algorithm to generates identity value.

If underlying database software does not support identity columns and sequence then the native algorithm looks to use hilo algorithm to generate id value.

Example:

```
<id name="no" column="customerId">
      <generator class="native"/>
</id>
```

It uses identity algorithm

```
<id name="no" column="eid">
        <generator class="native">
            <param name="sequence">my_seq</param>
        </generator>
</id>
```

It uses sequence algorithm

```
<id name="sno" column="eid">
    <generator class="native">
        <param name="table">mytable</param>
        <param name="column">mycol</param>
        <param name="max_lo">10</param>
    /genertor>
</id>
```

It uses hilo algorithm

Note In order to work with native algorithm first try to notice which algorithm it is using internally for the underlying database software. Based on this you start configuring parameters while working with native algorithm.

Universal Unique Id: (UUID)

This algorithm generates identifiers of type string, which is unique with in a network (uses IP address). This algorithms use identifier values as encoded string of hexadecimal digits of length 32. This algorithm works with all database software. The class name of this algorithm is "org.hibernate.id.UUIDHexGenerator". This algorithm generated id value is dynamic & random value.

```
<id name="customerId" column="CUSTOMER_ID">
        <generator class="uuid"/>
</id>
```

Global Unique Id (guid)

This algorithm uses "sys guid()" function available in sql to generate string based dynamic & random identifier value. The class name of this algorithm is "org.hibernate.id.GUIDGenerator"

Note: uuid algorithm makes hibernate software to generate String based id value. guid algorithm make underlying database software to generate id value and hibernate software uses that value.

```
<id name="customerId" column="CUSTOMER_ID">
        <generator class="guid"/>
</id>
```

Select:

This algorithm take the database trigger generated value as id value of hibernate pojo class object. The class name is "org.hibernate.id.SelectGenerator"

Example

```
-- Creating CUSTOMER_MASTER TABLE
create table CUSTOMER_MASTER (
       CUST_ID number(10,0) not null,
        CUST_NAME varchar2(255 char),
        MOBILE_NUM number(19,0),
        CUST_EMAIL varchar2(255 char),
        IS_ACTIVE varchar2(255 char),
        primary key (CUST_ID)
)
```

```
-- Creating Sequence
CREATE SEQUENCE C_ID_SEQ  START WITH 1  INCREMENT BY 1;
```

```
-- Creating Trigger For CUSTOMER_MASTER table to insert CUST_ID value from Sequence
CREATE OR REPLACE TRIGGER CUSTOMER_ID_TRIGGER BEFORE
  INSERT ON CUSTOMER_MASTER
  REFERENCING NEW AS NEW OLD AS OLD
  FOR EACH ROW
  BEGIN
  SELECT C_ID_SEQ.NEXTVAL INTO :NEW.CUST_ID FROM DUAL;
END CUSTOMER_ID_TRIGGER;
```

```
-- Configuring Select generator in Hibernate Mapping file
<id name="customerId" column="CUST_ID">
       <generator class="select">
              <param name="key">mobileNum</param>
       </generator>
</id>
```

```
public class Customer {

       private Integer customerId;
       private String customerName;
       private Long mobileNum;

       //setters and getters
}
```

Customer.java

Note: Select generator will not generate the value. When Database trigger inserted the value select generator will fetch primary key column value using natural key from the table (need to configure unique column above it is "mobileNum") and will populate into Entity object.

Foreign:

This algorithm is useful to generate id value in one pojo class object by collecting identity value form another pojo class object. This algorithm is very useful in one-to-one association mapping. The class name of this algorithm is “org.hibernate.id.For-eignGenerator”.

Hilo (high and low):

This algorithm generates identifiers of type long, short, int by using given table and its column as the source of ‘high’ values. The class name of this algorithm is “org.hibernate.id.TableHiLoGenerator”

This algorithm requires 3 mandatory parameters. Table, column, Max_lo

The column in the special table keeps track of the number of records that are inserted using this HiLo algorithm.

When multiple identity values are generated using this Hilo algorithm, the difference between each two values is (“Max_lo value+1”). This algorithm works with all database software.

Example: SQL> select * from mytable;

Mycol

400 —-> you can insert any value

```
<id name="sno" column="eid">
        <generator class="hilo">
                <param name="table">mytable</param>
                <param name="column">mycol</param>
                <param name="max_lo">10</param>
        </generator>
</id>
```

Note: The first id value that has come through this algorithm is dynamic and random value. For this algorithm table must be empty.

Seqhilo:

This algorithm uses given database sequence and max_lo parameter value to generate identifiers of type long, short or int. This algorithms works with only in that database software, which supports sequence

This algorithm having 2 mandatory parameters

1. -sequence
2. -max_lo

When multiple values (identity) are generated using this algorithm, the difference between two id values will be [increment by value of (sequence) * max_lo value] + [increment value of (sequence)]

The first id value generated by this algorithm is random value.

```
<id name="sno" column="eid'>
        <generator class="seqhilo">
                <param name="sequence">my_seq</param>
                <param name="max_lo">10</param>
        </generator>
</id>
```

Custom ID generation

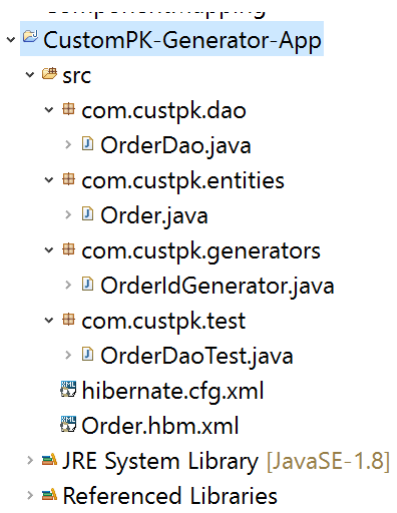
Sometimes circumstances force us to generate ID values that are not covered by existing Hibernate generators. For example, we might be forced to create a mapping for a legacy database that cannot be changed, or simply the requirements force us to do so. In such case, custom ID generation strategies may come in handy. Instead of using existing Hibernate generators, you can implement your own ID generator. You can do that either by implementing one of Hibernate generator interfaces like the most basic org.hibernate.id.IdentifierGenerator or other interfaces like org.hibernate.id.PersistentIdentifierGenerator. If all you need to do is apply some tweaks to an existing strategy, you can also extend existing generators.

Let us consider a (very unlikely) scenario that we need to create a String ID that is a concatenation of values originating from two database sequences. Let us assume that the format would be ID1_ID2.

In order to create our own generator class, we need to implement with the org.hibernate.id.IdentifierGenerator interface. This interface is having one method. So this method we need override.

```
import org.hibernate.id.IdentifierGenerator;

public class MyGenerator implements IdentifierGenerator {

   @Override

   public Serializable generate(SessionImplementor session, Object object)   {

      // your logic comes here.

   }

}
```

Example



Step 1: Create Order.java entity class

```
public class Order {

        private String orderId;
        private String orderBy;
        private Double orderPrice;

        //setters & getters
}
```

Order.java

Step 2: Create a sequence and use the sequence in CustomGenerator.java class like below (This class is the once which generates custom primary key for every record insertion).

SQL > CREATE SEQUENCE ORDER_ID_SEQ START WITH 1 INCREMENT BY 1;

```
public class OrderIdGenerator implements IdentifierGenerator {
	@Override
	public Serializable generate(SharedSessionContractImplementor session,
			Object arg1) throws HibernateException {
		String prefix = "OD";
		int suffix = 1;
		try {
			Connection con = session.connection();
			String sql = "SELECT ORDER_ID_SEQ.NEXTVAL FROM DUAL";
			Statement stmt = con.createStatement();
			ResultSet rs = stmt.executeQuery(sql);
			if (rs.next()) {
				suffix = rs.getInt(1);
			}
		} catch (Exception e) {
			e.printStackTrace();
		}
		return prefix.concat(String.valueOf(suffix));
	}
}
```

OrderIdGenerator.java

Step 3: Configure OrderIdGenerator (Custom Generator) in mapping file

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
	<class name="com.custpk.entities.Order" table="ORDER_DETAILS">
		<id name="orderId" column="O_ID">
			<generator class="com.custpk.generators.OrderIdGenerator" />
		</id>
		<property name="orderBy" column="O_BY" />
		<property name="orderPrice" column="O_PRICE"></property>
	</class>
</hibernate-mapping>
```

Order.hbm.xml

Step 4: Create OrderDao.java file

```
public class OrderDao {
	public boolean insert(Order entity) {
		boolean isInserted = false;
		Configuration cfg = new Configuration().configure();
		cfg.configure();
		SessionFactory sf = cfg.buildSessionFactory();
		Session hs = sf.openSession();
		Transaction tx = hs.beginTransaction();

		Serializable id = hs.save(entity);
		if (id != null) {
			isInserted = true;
		}
		tx.commit();
		hs.close();
		sf.close();
		return isInserted;
	}
}
```

OrderDao.java

Step 5: Create OrderDaoTest.java class and test insert functionality.

Composite Primary Key in Hibernate

A composite key, in the context of relational databases, is a combination of two or more columns in a table that can be used to uniquely identify each row in the table. Uniqueness is only guaranteed when the columns are combined; when taken individually the columns do not guarantee uniqueness.

Any column(s) that can guarantee uniqueness is called a candidate key; however, a composite key is a special type of candidate key that is only formed by a combination of two or more columns. Sometimes the candidate key is just a single column, and sometimes it is formed by joining multiple columns.

A composite key can be defined as the primary key. This is done using SQL statements at the time of table creation. It means that data in the entire table is defined and indexed on the set of columns defined as the primary key.

Note: Composite Primary Key class should implement java.io.Serializable interface.

```
CREATE TABLE PROJECT_RESOURCE_ALLOCATION (
  RESOURCE_ID NUMBER,
  PROJECT_ID NUMBER,
  ALLOC_START_DT DATE,
  ALLOC_END_DT DATE,
  ALLOC_PERCENTAGE NUMBER
  PRIMARY KEY (RESOURCE_ID, PROJECT_ID)
);
```

```
public class ProjectResourceAllocation implements Serializable{

      private Integer resourceId;
      private Integer projectId;
      private Date allocStartDt;
      private Date allocEndDt;
      private Integer allocPercentage

      //setters & getters
}
```

ProjectResourceAllocation.java

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
      <class name="com.nit.entities.ProjectResourceAllocation" table="PROJECT_RESOURCE_ALLOCATION">
            <composite-id>
                  <key-property name="resourceId" column="RESOURCE_ID" />
                  <key-property name="projectId" column="PROJECT_ID" />
            </composite-id>
            <property name="allocStartDt" column="ALLOC_START_DT"/>
            <property name="allocEndDt" column="ALLOC_END_DT"/>
            <property name="allocPercentage" column="ALLOC_PERCENTAGE"/>
      </class>
</hibernate-mapping>
```

ProjectResourceAllocation.hbm.xml

```
public class ProjectResourceAllocDao {

	public boolean insert(ProjectResourceAllocation entity) {
		boolean isInserted = false;
		SessionFactory sf = HibernateUtil.getSf();
		Session hs = sf.openSession();
		Transaction tx = hs.beginTransaction();
		Serializable id = hs.save(entity);
		if (id != null) {
			isInserted = true;
		}
		tx.commit();
		hs.close();
		return isInserted;
	}
	public ProjectResourceAllocation findByIds(ProjectResourceAllocation pra) {
		SessionFactory sf = HibernateUtil.getSf();
		Session hs = sf.openSession();
		pra = hs.get(ProjectResourceAllocation.class,pra);
		hs.close();
		return pra;
	}
}
```

ProjectResourceAllocDao.java

Composite-Primary Mapping using Annotations

```
@Entity
@Table(name = "RESOURCE_ALLOCATION")
public class ResourceAllocation implements Serializable {

	@Column(name = "ALLOC_START_DT")
	private Date allocStartDt;

	@Column(name = "ALLOC_END_DT")
	private Date allocEndDt;

	@Column(name = "ALLOC_PERCENTAGE")
	private Double allocPercentage;

	@Id
	@Embedded
	private ResourceAllocationPK pks;

	//setters & getters
}
```

ResourceAllocation.java

```
@Embeddable
public class ResourceAllocationPK implements Serializable {

	@Column(name = "RESOURCE_ID")
	private Integer resourceId;

	@Column(name = "PROEJCT_ID")
	private Integer projectId;

	//setters & getters
}
```

ResourceAllocationPK.java

Component Mapping

A Component mapping is a mapping for a class having a reference to another class as a member variable.

A component is a contained object that is persisted as a value type and not an entity reference. The term "component" refers to the object-oriented notion of composition and not to architecture-level components.

```
public class Address implements java.io.Serializable {

	private String city;
	private String state;
	private String country;
	//setters and getters
}
```

```
public class Customer implements java.io.Serializable {
	private Integer custId;
	private String custName;
	private int age;
	private Address address;
	//setters and getters
}
```

In this case, Address.java is a “component” represent the “city”, “state” and "country" columns for Customer.java

```
<hibernate-mapping>
	<class name="com.aits.Customer" table="customer">
		<id name="custId" type="java.lang.Integer">
			<column name="CUST_ID" />
			<generator class="identity" />
		</id>
		<property name="custName" type="string">
			<column name="CUST_NAME" length="10" not-null="true" />
		</property>
		<component name="Address" class="com.aits.Address">
			<property name="city" type="string">
				<column name="CITY" not-null="true" />
			</property>
			<property name="state" type="string">
				<column name="STATE" not-null="true" />
			</property>
			<property name="country" type="string">
				<column name="COUNTRY" not-null="true" />
			</property>
		</component>
	</class>
</hibernate-mapping>
```

```
public class ComponentMappingApp {
	public static void main(String[] args) {

	Session session = HibernateUtil.getSessionFactory().openSession();
	session.beginTransaction();

	Address address = new Address();
	address.setCity("HYD");
	address.setState("TG");
	address.setCountry("INDIA");

	 Customer cust = new Customer();
	 cust.setCustName("Sunil");
	 cust.setAddress(address);
	 session.save(cust);
	  session.getTransaction().commit();
	System.out.println("Done");
   }
}
```

Batch Processing

Many applications within the enterprise domain require bulk processing to perform business operations in mission critical environments. These business operations include automated, complex processing of large volumes of information that is most efficiently processed without user interaction. These operations typically include time based events (e.g. month-end calculations, notices or correspondence), periodic application of complex business rules processed repetitively across very large data sets (e.g. Insurance benefit determination or rate adjustments), or the integration of information that is received from internal and external systems that typically requires formatting, validation and processing in a transactional manner into the system of record. Batch processing is used to process billions of transactions every day for enterprises.

Suppose there is one situation in which we have to insert 1000000 records in to database in a time. So what to do in this situation...

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<1000000; i++ )
{
    Student student = new Student(.....);
    session.save(student);
}
tx.commit();
session.close();
```

By default, Hibernate will cache all the persisted objects in the session-level cache and ultimately your application would fall over with an OutOfMemoryException somewhere around the 50,000th row. You can resolve this problem if you are using batch processing with Hibernate.

To use the batch processing feature, first set hibernate.jdbc.batch_size as batch size to a number either at 20 or 50 depending on object size. This will tell the hibernate container that every X rows to be inserted as batch. To implement this in your code we would need to do little modification as follows:

```
Session session = SessionFactory.openSession();

Transaction tx = session.beginTransaction();
for ( int i=0; i<1000000; i++ ){
    Student student = new Student(.....);
    session.save(employee);
    if( i % 50 == 0 ) // Same as the JDBC batch size
    {
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}
tx.commit();
session.close();
```

Above code will work fine for the INSERT operation, but if you want to make UPDATE operation then you can achieve using the following code:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
ScrollableResults studentCursor = session.createQuery("FROM STUDENT").scroll();
int count = 0;
while(studentCursor .next()){
   Student student = (Student) studentCursor.get(0);
   student.setName("DEV");
   seession.update(student);
   if ( ++count % 50 == 0 ) {
      session.flush();
      session.clear();
   }
}
tx.commit();
session.close();
```

If you are undertaking batch processing you will need to enable the use of JDBC batching. This is absolutely essential if you want to achieve optimal performance. Set the JDBC batch size to a reasonable number (10-50).

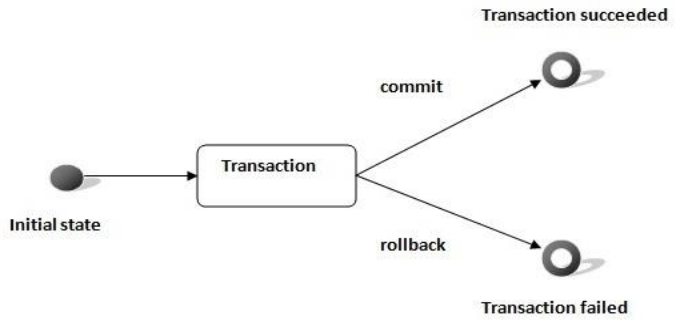
`hibernate.jdbc.batch_size 50`

```
SessionFactory sf =    HibernateUtil.getSessionFactory();

  Session session = sf.openSession();
  Transaction transaction = session.beginTransaction();

  for ( int i=0; i<100000; i++ ){
            String studentName = "Ashok " + i;
            int rollNumber = 9 + i;
            Student student = new Student();
            student.setStudentName(studentName);
            student.setRollNumber(rollNumber);
            session.save(student);
          if( i % 50 == 0 ){
               session.flush();
               session.clear();
          }
  }
  transaction.commit();
  session.close();
  sf.close();
```

Transaction management in Hibernate

Databases implement the notion of a unit of work as a database transaction (sometimes called a system transaction).

A database transaction groups data-access operations. A transaction is guaranteed to end in one of two ways: it's either committed or rolled back. Hence, database transactions are always truly atomic. In below figure we can see this graphically. If several database operations should be executed inside a transaction, you must mark the boundaries of the unit of work. You must start the transaction and, at some point, commit the changes. If an error occurs (either while executing operations or when committing the changes), you have to roll back the transaction to leave the data in a consistent state. This is known as transaction demarcation, and (depending on the API you use) it involves more or less manual intervention.



Transaction Interface in Hibernate

In hibernate framework, we have **Transaction** interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC).

A transaction is associated with Session and instantiated by calling **session.beginTransaction()**.

The methods of Transaction interface are as follows:

1. **void begin()** starts a new transaction.
2. **void commit()** ends the unit of work unless we are in FlushMode.NEVER.
3. **void rollback()** forces this transaction to rollback.
4. **void setTimeout(int seconds)** it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
5. **boolean isAlive()** checks if the transaction is still alive.
6. **void registerSynchronization(Synchronization s)** registers a user synchronization callback for this transaction.
7. **boolean wasCommited()** checks if the transaction is commited successfully.
8. **boolean wasRolledBack()** checks if the transaction is rolledback successfully.

```
Session session = sessions.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();

    concludeAuction();

    tx.commit();
} catch (Exception e) {
    if (tx != null) {
        try {
            tx.rollback();
        } catch (HibernateException he) {
            //log he and rethrow e
        }
    }
    throw e;
} finally {
    try {
        session.close();
    } catch (HibernateException he) {
        throw he;
    }
}
```

The call to session.beginTransaction() marks the beginning of a database transaction. In the case of a non-managed environment, this starts a JDBC transaction on the JDBC connection. In the case of a managed environment, it starts a new JTA transaction if there is no current JTA transaction, or joins the existing current JTA transaction. This is all handled by Hibernate—you houldn't need to care about the implementation.

The call to tx.commit()synchronizes the Session state with the database. Hibernate then commits the underlying transaction if and only if beginTransaction() started a new transaction (in both managed and non-managed cases). If begin- Transaction() did not start an underlying database transaction, commit() only synchronizes the Session state with the database; it's left to the responsible party (the code that started the transaction in the first place) to end the transaction. This is consistent with the behavior defined by JTA. If concludeAuction() threw an exception, we must force the transaction to roll back by calling tx.rollback(). This method either rolls back the transaction immediately or marks the transaction for "rollback only" (if you're using CMTs).

It is critically important to close the Session in a finally block in order to ensure that the JDBC connection is released and returned to the connection pool. (This step is the responsibility of the application, even in a managed environment.)

Annotations

Java Annotations allow us to add metadata information into our source code, although they are not a part of the program itself. Annotations were added to the java from JDK 5. Annotation has no direct effect on the operation of the code they annotate (i.e. it does not affect the execution of the program) .Annotations are given by SUN as replacement to the use of xml files in java Every annotations is internally an Interface, but the key words starts with @ symbol.

What is the use of Annotations?

- ✓ Instructions to the compiler: There are three built-in annotations available in Java (@Deprecated, @Override & @SuppressWarnings) that can be used for giving certain instructions to the compiler. For example the @override annotation is used for instructing compiler that the annotated method is overriding the method. More about these built-in annotations with example is discussed in the next sections of this article.
- ✓ Compile-time instructors: Annotations can provide compile-time instructions to the compiler that can be further used by software build tools for generating code, XML files etc.
- ✓ Runtime instructions: We can define annotations to be available at runtime which we can access using java reflection and can be used to give instructions to the program at runtime. We will discuss this with the help of an example, later in this same post.

Annotations basics
An annotation always starts with the symbol @ followed by the annotation name. The symbol @ indicates to the compiler that this is an annotation.

For e.g. @Override

Where we can use annotations?

Annotations can be applied to the classes, interfaces, methods and fields. For example the below annotation is being applied to the method.

Built-in Annotations in Java

- @Override
- @Deprecated
- @SuppressWarnings

1) @Override:

While overriding a method in the child class, we should use this annotation to mark that method. This makes code readable and avoid maintenance issues, such as: while changing the method signature of parent class, you must change the signature in child classes (where this annotation is being used) otherwise compiler would throw compilation error. This is difficult to trace when you haven't used this annotation.

```
public class Parent {

    public void justaMethod() {
        System.out.println("Parent class method");
    }
}

public class Child extends Parent {

    @Override
    public void justaMethod() {
        System.out.println("Child class method");
    }
}
```

2) @Deprecated

@Deprecated annotation indicates that the marked element (class, method or field) is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field that has already been marked with the @Deprecated annotation. When an element is deprecated, it should also be documented using the Javadoc @deprecated tag, as shown in the following example. Make a note of case difference with @Deprecated and @deprecated. @deprecated is used for documentation purpose.

```
@Deprecated
public void anyMethodHere(){
    // Do something
}
```

Now, whenever any program would use this method, the compiler would generate a warning

@SuppressWarnings

This annotation instructs compiler to ignore specific warnings. For example in the below code, I am calling a deprecated method (lets assume that the method deprecatedMethod() is marked with @Deprecated annotation) so the compiler should generate a warning, however I am using @SuppressWarnings annotation that would suppress that deprecation warning.

```
@SuppressWarnings("deprecation")
    void myMethod() {
        myObject.deprecatedMethod();
}
```

Let us see few points regarding annotations in hibernate

- ✓ In hibernate annotations are given to replace hibernate mapping [xml] files
- ✓ While working with annotations in hibernate, we do not require any mapping files, but hibernate xml configuration file is must
- ✓ Hibernate borrowed annotations from java persistence API but hibernate itself doesn't contain its own annotations

Hibernate Annotations

@Entity: This Annotation marks this class as an entity.
@Table: This Annotation specifies the table name where data of this entity is to be persisted. If you are not using @Table annotation in Entity class, hibernate will use the class name as the table name by default.
@Id: This Annotation marks the identifier for this entity.
@GeneratedValue: This Annotation is used to specify the primary key generation strategy to use. If the strategy is not specified by default AUTO will be used.
@Column: This Annotation specifies the details of the column for this property or field. If @Column annotation is not specified, property name will be used as the column name by default. We 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.

```
 import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "PRODUCTS")
public class Product {

	@Id
	@Column(name = "proid")
	private Integer productId;

	@Column(name = "proName", length = 10)
	private String proName;

	@Column(name = "price")
	private Double price;

	// setters and getters
}
```

```
@GeneratedValue
@GeneratedValue(strategy=GenerationType.AUTO)
@GeneratedValue(strategy=GenerationType.IDENTITY)
@GeneratedValue(strategy=GenerationType.SEQUENCE)
@GeneratedValue(strategy=GenerationType.TABLE)

@GeneratedValue(strategy=GenerationType.SEQUENCE,generator="sample")
@SequenceGenerator(name="sample",sequenceName="emp_seq")

@GeneratedValue(generator="sample")
@GenericGenerator(name="sample",strategy="com.app.model.MyGen")
```

Primary key generators

```
DATE AND TIME:(java.util.DATE)

@Temporal(TemporalType.DATE)
private Date dateOfBirth;

@Temporal(TemporalType.TIME)
private Date createdDate;

@Temporal(TemporalType.TIMESTAMP)
private Date updatedDate;
```

```
LOB and CLOB

@Lob
private byte[] image;
@Lob
private char[] doc;
```

```
@NotNull(message="Employee name must not be null")
@Size(min=3,max=6,message="Employee Name must be in 3-6 chars")
@Pattern(regexp="SAT[A-Z]*")
private String empName;

@Min(value=3,message="EmpSal minimum nuber is 3")
@Max(4)
@Column(name="esal")
private double empSal;

@AssertTrue
private boolean isEnabled;

@AssertFalse
private boolean isFinished;

@Past
@NotNull
private Date date1;

@Future
private Date date2;
```

Validations

Working with Large Objects

Sometimes, our data is not limited to strings and numbers. We need to store a large amount of data in Database table like documents, raw files, XML documents etc. To store these files or photos our table column datatype should be BLOB or CLOB.

To Support for BLOB or CLOB hibernate provided **@Lob** annotation

@Lob saves the data in BLOB or CLOB

CLOB(Character Large Object): If data is text and is not enough to save in VARCHAR, then that data should be saved in CLOB.

BLOB(Binary Large Object): In case of double byte character large data is saved in BLOB data type.

In case of Character[],char[] and String data is saved in CLOB. And the data type Byte[], byte[] will be stored in BLOB.

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

	@Id
	@GeneratedValue
	@Column(name = "EMP_ID")
	private Integer empId;

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

	@Lob
	@Column(name = "EMP_IMAGE")
	private byte[] empPhoto;

	@Version
	@Temporal(TemporalType.TIMESTAMP)
	private Date update_dt;

	 //setters & getters
}
```

Employee.java

```java
public class ImageDemo {

	public static void main(String[] args) throws Exception {
		insertImage();
		readImage();
	}

	public static void readImage() throws Exception {
		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();

		Employee emp = hsession.get(Employee.class, 1);

		FileOutputStream fos = new FileOutputStream("image2.jpeg");
		fos.write(emp.getEmpPhoto());
		fos.flush();
		fos.close();

		tx.commit();
		hsession.close();
		sf.close();
	}

	public static void insertImage() throws Exception {
		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();

		File f = new File("image1.jpeg");
		byte[] brr = new byte[(int) f.length()];

		FileInputStream fis = new FileInputStream(f);
		fis.read(brr);

		Employee emp = new Employee();
		emp.setEmpName("Ashok");
		emp.setEmpPhoto(brr);
		hsession.save(emp);

		tx.commit();
		hsession.close();
		sf.close();
	}
}
```

ImageDao.java

Versioning and Timestamping in Hibernate

Let's assume that, two users are working on a project and they both are currently viewing a bug or an enhancement in a project management application. Let's say one user knows that it is a duplicate bug while the other does not know and thinks that this needs to be done. Now one user changes its status as **IN PROGRESS** and the other user marks it as **DUPLICATE** at the same time but the request with status as **DUPLICATE** goes a bit early to the server and the request with status IN PROGRESS arrives later. What will be the current status?

It will be IN PROGRESS as the later request will overwrite the previous status. These kind of scenarios are common when multiple users work on same application and should be prevented.

In general, following is the scenario which should be prevented

- Two transactions read a record at the same time.
- One transaction updates the record with its value.
- Second transaction, not aware of this change, updates the record according to its value.

End Result is, the update of first transaction is completely lost.

Solution: Hibernate has a provision of version based control to avoid such kind of scenarios. Under this strategy, a version column is used to keep track of record updates. This version may either be a timestamp or a number.

If it is a number, Hibernate automatically assigns a version number to an inserted record and increments it every time the record is modified.

If it is a timestamp, Hibernate automatically assigns the current timestamp at which the record was inserted / updated.

Hibernate keeps track of the latest version for every record (row) in the database and appends this version to the where condition used to update the record. When it finds that the entity record being updated belongs to the older version, it issues an **org.hibernate.StaleObjectStateException** and the wrong update will be cancelled.

How to Implement Versioning??

There are only three steps required for versioning to work for an entity. They are :

1. Add a column with any name (usually it is named as Version) in the database table of the entity. Set its type either to a number data type (int, long etc.) or a timestamp as you want to store its value.

2. Add a field of the corresponding type to entity class. If we have made the version column in the database table of a number type , then this field should be either int, long etc. and if the column type in the database is of type timestamp, then this field should be a java.sql.Timestamp.

3. Annotate the above field in your entity class with @Version or provide the definition of version column in the <class> tag if you are using XML based entity definitions.

```java
@Entity
@Table(name = "Defects")
public class Defect {

	@Id
	@GeneratedValue
	@Column(name="Id")
	private Integer id;
	@Column(name = "type")
	private String type;
	@Column(name = "description")
	private String description;
	@Column(name = "status")
	private String status;
	@Version
	private Integer version;

	// getter and setter methods
}
```

Defect.java

```
public class VersionInsert {

	public static void main(String[] args) throws Exception {

		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();

		Defect defect = new Defect();
		defect.setType("Bug");
		defect.setDescription("Core defect");
		defect.setStatus("DUPLICATE");
		//We did not set the version field value

		hsession.save(defect);

		tx.commit();
		hsession.close();
		sf.close();
	}
}
```

VersionInsert.java

Executing the above code will generate a database record as shown below

	Id	Type	Description	Status	Version
▶	1	Bug	Core defect	DUPLICATE	0

The value 0 in the version column is automatically generated by Hibernate. **Note that we did not set version in the above code**. Below is the query generated by Hibernate while saving the record.

```
Hibernate: insert into defect(Description, Status, Type, version) values (?, ?, ?, ?)
```

See the field version is included in the insert query generated by Hibernate though we did not set its value.

Now let's make a change in this record and update it using the below code:

```
public class VersioningUpdate {

	public static void main(String[] args) throws Exception {

		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();

		Transaction tx = hsession.beginTransaction();

		// fetch record with id as 1
		Defect defect = session.get(Defect.class, 1);
		// change a field value
		defect.setStatus("CLOSED");
		// We did not set the version field value again
		session.update(defect);

		tx.commit();
		hsession.close();
		sf.close();
	}

}
```

VersioningUpdate.java

Executing the above code will generate a database record as shown below

	Id	Type	Description	Status	Version
▶	1	Bug	Core defect	CLOSED	1

The value 1 in the version column is automatically generated by Hibernate. **Note that we did not update version in the above code**.

How versioning will solve the above scenario?

Two Users are viewing at a defect ticket at the same time. We will call them User1 and User2. Let's say the defect is a new entry in database table. Post versioning implemented, it will have version as 0. User1 and User2 change the status of a bug at the same time but the request of User1 arrives a bit early to the server. User1 has changed the status to DUPLICATE, the query issued is :

update defect set Description=?, Status=DUPLICATE, Type=?, version=1 where id=? and version=0

Now the request of User2 to change the status to IN PROGRESS arrives, the query issued will be :

update defect set Description=?, Status=IN PROGRESS, Type=?, version=1 where id=? and version=0

Note the version in the WHERE clause is 0 since both the users were looking at version 0 of the defect. Now there is NO record matching the id of that defect and version since the version has been changed by the previous update. Hibernate is smart enough to detect an object mismatch and issues a org.hibernate.StaleObjectStateException thus preventing the update on an already updated record.

- If the type of version column in database and the field type in java class are set to timestamp, then every time a record is updated, the current timestamp is set as the version value.
- When timestamp is set as the type of value for version column, then its value can indicate the time at which the entity was updated.
- It is not mandatory to name the java field-representing version as version itself.

Timestamping

Storing the creation timestamp or the timestamp of the last update is a common requirement for modern applications. It sounds like a simple requirement, but for a huge application, we do not want to set a new update timestamp in every use case that changes the entity.

We need a simple, fail-safe solution that automatically updates the timestamp for each change. As so often, there are multiple ways to achieve that:

- We can use a database update trigger that performs the change on a database level. Most DBAs will suggest this approach because it's easy to implement on a database level. But Hibernate needs to perform an additional query to retrieve the generated values from the database.
- We can use an entity lifecycle event to update the timestamp attribute of the entity before Hibernate performs the update.
- We can use an additional framework, like Hibernate Envers, to write an audit log and get the update timestamp from there.
- We can use the Hibernate-specific **@CreationTimestamp** and **@UpdateTimestamp** annotations and let Hibernate trigger the required updates.

It's obvious that the last option is the easiest one to implement if we can use Hibernate-specific features. So let's have a more detailed look at it.

@CreationTimestamp and @UpdateTimestamp

Hibernate's @CreationTimestamp and @UpdateTimestamp annotations make it easy to track the timestamp of the creation and last update of an entity.

When a new entity gets persisted, Hibernate gets the current timestamp from the VM and sets it as the value of the attribute annotated with @CreationTimestamp. After that, Hibernate will not change the value of this attribute.

The value of the attribute annotated with @UpdateTimestamp gets changed in a similar way with every SQL Update statement. Hibernate gets the current timestamp from the VM and sets it as the update timestamp on the SQL Update statement.

Supported attribute types

We can use the @CreationTimestamp and @UpdateTimestamp with the following attribute types:

- java.time.LocalDate (since Hibernate 5.2.3)
- java.time.LocalDateTime (since Hibernate 5.2.3)
- java.util.Date
- java.util.Calendar
- java.sql.Date
- java.sql.Time

```
import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

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

	@Id
	@GeneratedValue
	@Column(name = "EMP_ID", updatable = false, nullable = false)
	private Long empId;

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

	@CreationTimestamp
	@Column(name = "CREATE_DT")
	private Date createDateTime;

	@UpdateTimestamp
	@Column(name = "UPDATE_DT")
	private Date updateDateTime;

	// setters and getters
}
```

When we persist a new Employee, Hibernate will get the current time from the VM; store it as the creation, and update timestamp values in Employee table.

Interceptors in Hibernate

Hibernate is all about Entity persistence and quite often we would want to intercept the request or perform some tasks when state of an object changes. With the help of interceptors , we get an opportunity to inspect the state of an object and if needed we can change the state as well.To support this Hibernate does support the concept of interceptor and provides a Interface and a implementation of Interface. Developers can either directly implement the Interface or may extend the implementation.

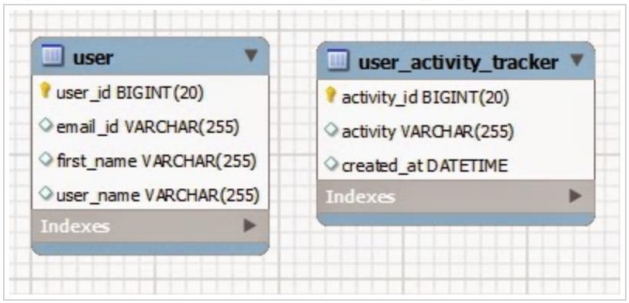
The Interceptor in Hibernate provides callback from the session to the application, allowing the application to inspect and/or manipulate the properties of a persistent object just before it is saved,update,delete,load from the database. One of the common use of interceptor is to track the activity of the application.

Two ways to implement interceptor in Hibernate, by extending EmptyInterceptor or by implementing Interceptor interface. In this tutorial we are using EmptyInterceptor class with few common methods of Interceptor interface.

- public void onDelete(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types)
- public boolean onFlushDirty(Object entity, Serializable id, Object[] currentState, Object[] previousState, String[] propertyNames, Type[] types)
- public boolean onLoad(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types)
- public boolean onSave(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types)
- public void preFlush(Iterator iterator)
- public void postFlush(Iterator iterator)

For example, we are going to track the activity of the operation performed on the user object, like if any new user is added/updated/delete into the user table, at the same time it will also create a new record inside the user_activity_tracker table.

Below are the ER diagram of the tables we are using for this project.



```xml
<hibernate-configuration>
	<session-factory>
		<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
		<property name="connection.url">jdbc:mysql://localhost:3306/hibernate
		</property>
		<property name="connection.username">root</property>
		<property name="connection.password">root</property>
		<property name="dialect">org.hibernate.dialect.MySQL5InnoDBDialect
		</property>
		<!-- Echo all executed SQL to stdout -->
		<property name="show_sql">true</property>
		<property name="format_sql">true</property>
		<property name="hbm2ddl.auto">update</property>

		<mapping class="com.aits.User" />
		<mapping class="com.aits.UserActivityTracker" />
	</session-factory>
</hibernate-configuration>
```

hibernate.cfg.xml

```java
@Entity
@Table(name = "user")
public class User {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "user_id")
	private long userId;

	@Column(name = "user_name")
	private String userName;

	@Column(name = "first_name")
	private String firstName;

	@Column(name = "email_id")
	private String emailId;

	// setters && getters

}
```

User.java

```java
@Entity
@Table(name = "user_activity_tracker")
public class UserActivityTracker {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "activity_id")
	private long activityId;

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

	@Column(name = "created_at")
	private Date createdAt;

	public UserActivityTracker(String activity, Date createdAt) {
		super();
		this.activity = activity;
		this.createdAt = createdAt;
	}

}
```

UserActivityTracker.java

----------ActivityTrackerInterceptor.java----------

```
public class ActivityTrackerInterceptor extends EmptyInterceptor {

	private static final long serialVersionUID = 1L;
	private String operation = "INSERT";
	boolean isMainEntity = false;

	// called when record deleted.
	public void onDelete(Object entity, Serializable id, Object[] state, String[] proper-
tyNames, Type[] types) {
		operation = "DELETE";
		isMainEntity = true;
		System.out.println("Delete function called");
	}

	// called when record updated
	public boolean onFlushDirty(Object entity, Serializable id, Object[] currentState, Ob-
ject[] previousState,
			String[] propertyNames, Type[] types) {
		if (entity instanceof User) {
			System.out.println("Update function called");
			operation = "UPDATE";
			isMainEntity = true;
			return true;
		}
		isMainEntity = false;
		return false;
	}

	// called on load events
	public boolean onLoad(Object entity, Serializable id, Object[] state, String[] proper-
tyNames, Type[] types) {

		// log loading events
		System.out.println("load function called");
		return true;
	}



public boolean onSave(Object entity, Serializable id, Object[] state, String[] propertyNames,
Type[] types) {
		if (entity instanceof User) {
			System.out.println("save function called");
			operation = "INSERT";
			isMainEntity = true;
			return true;
		}
		isMainEntity = false;
		return false;
	}

	// called before commit into database
	public void preFlush(Iterator iterator) {
		System.out.println("Before commiting");
	}

	// called after committed into database
	public void postFlush(Iterator iterator) {
		System.out.println("After commiting");
		if (isMainEntity) {
			Session session = HibernateUtility.getSessionFactory().openSession();
			session.getTransaction().begin();
```

```
                    UserActivityTracker activityTracker = new UserActivityTracker(operation,
Calendar.getInstance().getTime());
                    session.save(activityTracker);
                    session.getTransaction().commit();
              }
              isMainEntity = false;
       }

}
```

```
public class Main {
       public static void main(String[] args) {
              Session session = HibernateUtility.getSessionFactory().openSession();

              session.beginTransaction();
              User user = new User();

              user.setEmailId("info@ashokitschool.com");
              user.setFirstName("Ashok");
              long insertedID = (Long) session.save(user);
              session.getTransaction().commit();

              session.beginTransaction();
              // fetching user
              User user2 = (User) session.load(User.class, insertedID);
              user2.setUserName("Kumar");
              session.update(user);
              session.getTransaction().commit();
       }
}
```

Main.java

HQL (Hibernate Query Language)

The Hibernate ORM framework provides its own query language called Hibernate Query Language or HQL for short. It is very powerful and flexible and has the following characteristics:

- **SQL similarity**: HQL's syntax is very similar to standard SQL. If you were familiar with SQL, then writing HQL would be easy: from SELECT, FROM, ORDER BY to arithmetic expressions and aggregate functions, etc.
- **Fully object-oriented**: HQL does not use real names of table and columns. It uses class and property names instead. HQL can understand inheritance, polymorphism and association.
- **Case-insensitive for keywords:** Like SQL, keywords in HQL are case-insensitive. That means SELECT, select or Select are the same.
- **Case-sensitive for Java classes and properties:** HQL considers case-sensitive names for Java classes and their properties, meaning Person and person are two different objects.

Advantages of HQL

- ✓ HQL is database independent, means if we write any program using HQL commands then our program will be able to execute in all the databases without doing any further changes to it
- ✓ HQL supports object oriented features ike **Inheritance**, **polymorphism**, **Associations**(Relationships)
- ✓ HQL is initially given for selecting object from database and in hibernate 3.x we can do DML operations (insert, update…) too

SQL	HQL
SQL > SELECT * FROM PRODUCT Note : Here PRODUCT is Table name	HQL > From Product Note : Here Product is Entity class name

- If we want to load the **Partial Object** from the database that is only selective properties (selected columns) of an objects then we need to replace column names with POJO class variable names.

SQL	HQL
SELECT P_ID, P_NAME FROM PRODUCT Note : Here P_ID,P_NAME are column names in PRODUCT table	SELECT p.pid, p.pname FROM Product Note: Here pid, pname are variable names in Product Entity class

- It is also possible to **load** or **select** the object from the database **by passing run time values** into the query, in this case we can use either " ? " symbol or label in an HQL command, the index number of " ? " will starts from zero but not one (Remember this, little important regarding interview point of view)

SQL	HQL
SELECT * FROM PRODUCT WHERE P_ID = ?	FROM Product where pid = ? or FROM Product where pid = :id or FROM Product where pid = 101

```
@Entity
@Table(name="STUDENTS")
public class Student {

	@Id
	@Column(name="student_id")
	private int studentId;
	@Column(name="first_name")
	private String firstName;
	@Column(name="last_name")
	private String lastName;
	@Column(name="roll_no")
	private String rollNo;

	// Generate Setters and Getters
}
```

```
public class MyApp {
        public static void main(String args[]) {
                // Get the session object.
                Session session = HibernateUtil.getSessionFactory().openSession();

                // Update the student object.
                Query query1 = session.createQuery("update Student set className = 'MCA final'"
                                + " where rollNo = 'MCA/07/70'");
                query1.executeUpdate();
                // select a student record
                Query query2 = session.createQuery("FROM Student where rollNo = 'MCA/07/70'");
                Student stu1 = (Student) query2.uniqueResult();
                System.out.println(stu1);
                // select query using named parameters
                Query query3 = session.createQuery("FROM Student where rollNo = :rollNo");
                query3.setParameter("rollNo", "MCA/07/70");
                Student stu2 = (Student) query3.uniqueResult();
                System.out.println(stu2);
                // select query using positional parameters
                Query query4 = session.createQuery("FROM Student where rollNo = ?");
                query4.setString(0, "MCA/07/70");
                Student stu3 = (Student) query4.uniqueResult();
                System.out.println(stu3);
                // delete a student record
                Query query5 = session.createQuery("delete Student where rollNo = 'MCA/07/70'");
                query5.executeUpdate();
                // Commit hibernate transaction.
                session.getTransaction().commit();
                // Close the hibernate session.
                session.close();
        }
}
```

Pagination in Hibernate

Pagination through the result set of a database query is a very common application pattern. Typically, you would use pagination for a web application that returned a large set of data for a query. The web application would page through the database query result set to build the appropriate page for the user. The application would be very slow if the web application loaded all of the data into memory for each user. Instead, you can page through the result set and retrieve the results you are going to display one chunk at a time.

There are two methods on the Query interface for paging: setFirstResult() and setMaxResults(). ThesetFirstResult() method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to only retrieve a fixed number of objects with the setMaxResults() method. Your HQL is unchanged—you need only to modify the Java code that executes the query.

```
Query query = session.createQuery("from Student");
query.setFirstResult(1);
query.setMaxResults(5);
List results = query.list();
displayStudentsList(results);
```

In general when we are doing search, the search may return huge number of results. But at a time if we try to display those results in the web page, it leads to the following problem

1. Takes more time to load
2. Takes more memory to hold all results
3. Difficult to navigate over huge number of records

To solve this problem we can go for pagination.

What is Pagination?

Pagination is the process of dividing content into multiple pages.

Example

In google search when we search, we will get the huge number of search results. But they will display only 10 links per page. And below the page they will display page numbers to navigate to other pages. This is nothing but pagination.

Pagination can be implemented in two ways.

1. Front-End pagination
2. Back-end pagination

1. Front-End pagination: In Front-End pagination, we will hit the database only once, and get the results and store them in the memory (session/application scope). When user navigating from one page to another, we retrieve data from memory but not hitting the database repeatedly.

With this style of pagination, we can achieve easy navigation, fast loading of the data. But still we have the problems are there like

- Takes more memory to hold all results
- If data is updated in the database by other application, we cannot get updated data, as we are taking the data from memory instead from database.

2. Back-End Pagination In Back-End Pagination, we will hit the database again and again, and get the requested page details from the database. With this style of pagination we can achieve easy navigation, fast loading of the data, takes memory to hold the result data and it will get always updated data. But it gives lessa performance when compared with front-end pagination.

- This type of pagination is suggestible, if number if resulted records are unlimited and the data is not constant data.
- For back-end pagination, we need to write a query in such a way that, it returns only the requested page details.

To implement this queries different databases using different queries. Like the database to database the query information is different. To overcome this problem, we can use Hibernate support. To apply pagination hibernate provides two methods in org.hibernate.Query interface.

setFirstResult(int firstResult) : Take the argument which specify from where records has to take.
setMaxResult(int maxResults) : Take the argument which specify how many records has to take.

```
1
2     String hqlQuery = "From Account a";
3     Query query = session.createQuery(hqlQuery);
4     query.setFirstResult(3); // Means start from 4th record
5     query.setMaxResults(5); // Means per page 5 records to be displayed
6     List<Account> accounts = query.list();
7     for(Account account: accounts) {
8        System.out.println("Account ID : " +account.getAccountId());
9        System.out.println("Name : " +account.getName());
10       System.out.println("Balance : " +account.getBalance());
     }
```

While using the pagination, in web page we are responsible to display the page numbers, And when the user click on some page number, we need to send the request to server, and get that corresponding page results. To do all these things we need to implement some logic. This logic on the web page if want we can implement on our own or we can use some third party provided custom tags. One of such custom tags is <display> tag. Apache people give this tag.

Hibernate Criteria Query Language (HCQL)

HCQL stands for Hibernate Criteria Query Language. As we discussed HQL provides a way of manipulating data using objects instead of database tables. Hibernate also provides more object oriented alternative ways of HQL. Hibernate Criteria API provides one of these alternatives. HCQL is mainly used in search operations and works on filtration rules and logical conditions.

Unlike HQL, Criteria is only for selecting the data from the database, that to we can select complete objects only not partial objects, in fact by combining criteria and projections concept we can select partial objects too. We can't perform non-select operations using this criteria. Criteria is suitable for executing dynamic queries too, let us see how to use this criteria queries in the hibernate..

The Criteria API allows you to build up a criteria query object programmatically; the org.hibernate.Criteria interface defines the available methods for one of these objects. The Hibernate Session interface contains severalcreateCriteria() methods. Pass the persistent object's class or its entity name to the createCriteria() method, and Hibernate will

create a Criteria object that returns instances of the persistence object's class when your application executes a criteria query.

The simplest example of a criteria query is one with no optional parameters or restrictions—the criteria query will simply return every object that corresponds to the class.

```
Criteria crit = session.createCriteria(Student.class);
List<Student> results = crit.list();
```

Using Restrictions with Criteria

The Criteria API makes it easy to use restrictions in your queries to selectively retrieve objects; for instance, your application could retrieve only products with a price over $30. You may add these restrictions to a Criteria object with the add() method. The add() method takes an org.hibernate.criterion.Criterion object that represents an individual restriction. You can have more than one restriction for a criteria query.

Hibernate criteria restrictions query example

Restrictions class provides the methods to restrict the search result based on the restriction provided.

1. Restrictions.eq: Make a restriction that the property value must be equal to the specified value.

Syntax: Restrictions.eq("property", specifiedValue)

2. Restrictions.lt: Make a restriction that the property value must be less than the specified value.

Syntax: Restrictions.lt("property", specifiedValue)

3. Restrictions.le: Make a restriction that the property value must be less than or equal to the specified value.

Syntax: Restrictions.le("property", specifiedValue)

4. Restrictions.gt: Make a restriction that the property value must be greater than the specified value.

Syntax: Restrictions.gt("property", specifiedValue)

5. Restrictions.ge: Make a restriction that the property value must be greater than or equal to the specified value.

Syntax: Restrictions.ge("property", specifiedValue)

6. Restrictions.like: Make a restriction that the property value follows the specified like pattern.

Syntax: Restrictions.like("property", "likePattern")

7. Restrictions.between: Make a restriction that the property value must be between the start and end limit values.

Syntax: Restrictions.between("property", startValue, endValue)

8. Restrictions.isNull: Make a restriction that the property value must be null.

Syntax: Restrictions.isNull("property")

9. Restrictions.isNotNull: Make a restriction that the property value must not be null.

Syntax: Restrictions.isNotNull("property")

i) Restrictions.eq() Example

To retrieve objects that have a property value that "**equals**" your restriction, use the `eq()` method on `Restrictions`, as follows:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.eq("description","Mouse"));
List<Product> results = crit.list()
```

Above query will search all products having description as "Mouse".

ii) Restrictions.ne() Example

To retrieve objects that have a property value "not equal to" your restriction, use the ne() method onRestrictions, as follows:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.ne("description","Mouse"));
List<Product> results = crit.list()
```

Above query will search all products having description anything but not "Mouse".

iii) Restrictions.like() and Restrictions.ilike() Example

Instead of searching for exact matches, we can retrieve all objects that have a property matching part of a given pattern. To do this, we need to create an SQL LIKE clause, with either the like() or the ilike() method. Theilike() method is case-insensitive.

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.like("name","Mou%",MatchMode.ANYWHERE));
List<Product> results = crit.list();
```

Above example uses an org.hibernate.criterion.MatchMode object to specify how to match the specified value to the stored data. The MatchMode object (a type-safe enumeration) has four different matches:

ANYWHERE: Anyplace in the string
END: The end of the string
EXACT: An exact match
START: The beginning of the string

v) Restrictions.isNull() and Restrictions.isNotNull() Example

The isNull() and isNotNull() restrictions allow you to do a search for objects that have (or do not have) null property values.

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.isNull("name"));
List<Product> results = crit.list();
```

v) Restrictions.gt(), Restrictions.ge(), Restrictions.lt() and Restrictions.le() Examples

Several of the restrictions are useful for doing math comparisons. The greater-than comparison is gt(), the greater-than-or-equal-to comparison is ge(), the less-than comparison is lt(), and the less-than-or-equal-to comparison isle(). We can do a quick retrieval of all products with prices over $25 like this, relying on Java's type promotions to handle the conversion to Double:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.gt("price", 25.0));
List<Product> results = crit.list();
```

vi) Combining Two or More Criteria Examples

Moving on, we can start to do more complicated queries with the Criteria API. For example, we can combine AND and OR restrictions in logical expressions. When we add more than one constraint to a criteria query, it is interpreted as an AND, like so:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.lt("price",10.0));
crit.add(Restrictions.ilike("description","mouse", MatchMode.ANYWHERE));
List<Product> results = crit.list();
```

If we want to have two restrictions that return objects that satisfy either or both of the restrictions, we need to use the or() method on the Restrictions class, as follows:

```
Criteria crit = session.createCriteria(Product.class);
Criterion priceLessThan = Restrictions.lt("price", 10.0);
Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);
LogicalExpression orExp = Restrictions.or(priceLessThan, mouse);
crit.add(orExp);
List results=crit.list();
```

Obtaining a Unique Result

Sometimes you know you are going to return only zero or one object from a given query. This could be because you are calculating an aggregate or because your restrictions naturally lead to a unique result. If you want obtain a single Object reference instead of a List, the uniqueResult() method on the Criteria object returns an object or null. If there is more than one result, the uniqueResult() method throws a HibernateException.

The following short example demonstrates having a result set that would have included more than one result, except that it was limited with the setMaxResults() method:

```
Criteria crit = session.createCriteria(Product.class);
Criterion price = Restrictions.gt("price",new Double(25.0));
crit.setMaxResults(1);
Product product = (Product) crit.uniqueResult();
```

Hibernate criteria ordering query example

Order class provides the methods for performing ordering operations.

Methods of Order class:

1. Order.asc: To sort the records in ascending order based on the specified property.

Syntax: Order.asc("property")

2. Order.desc: To sort the records in descending order based on the specified property.

Syntax: Order.desc("property")

Hibernate criteria projections query example

Instead of working with objects from the result set, you can treat the results from the result set as a set of rows and columns, also known as a projection of the data. This is similar to how you would use data from a SELECT query with JDBC.

To use projections, start by getting the org.hibernate.criterion.Projection object you need from theorg.hibernate.criterion.Projections factory class. The Projections class is similar to the Restrictions class in that it provides several static factory methods for obtaining Projection instances. After you get a Projection object, add it to your Criteria object with the setProjection() method. When the Criteria object executes, the list contains object references that you can cast to the appropriate type.

Example 1: Single Aggregate (Getting Row Count)

```
Criteria crit = session.createCriteria(Product.class);
crit.setProjection(Projections.rowCount());
List<Long> results = crit.list();
```

Projections class provides the methods to perform the operation on a particular column.

Other aggregate functions available through the Projections factory class include the following:

1. **avg(String propertyName)**: Gives the average of a property's value
2. **count(String propertyName)**: Counts the number of times a property occurs
3. **countDistinct(String propertyName)**: Counts the number of unique values the property contains
4. **max(String propertyName)**: Calculates the maximum value of the property values
5. **min(String propertyName)**: Calculates the minimum value of the property values
6. **sum(String propertyName)**: Calculates the sum total of the property values

Getting Selected Columns

Another use of projections is to retrieve individual properties, rather than entities. For instance, we can retrieve just the name and description from our product table, instead of loading the entire object representation into memory.

```
Criteria crit = session.createCriteria(Product.class);
ProjectionList projList = Projections.projectionList();
projList.add(Projections.property("name"));
projList.add(Projections.property("description"));
crit.setProjection(projList);
crit.addOrder(Order.asc("price"));
List<object[]> results = crit.list();
```

Query by Example

As part of the QBC facility, Hibernate supports query by example (QBE). The idea behind QBE is that the application supplies an instance of the queried class with certain property values set (to nondefault values). The query returns all persistent instances with matching property values. QBE isn't a particularly powerful approach, but it can be convenient for some applications. The following code snippet demonstrates a Hibernate QBE:

```
User u = new User();
u.setFirstname("Ashok");
Criteria criteria = session.createCriteria(User.class);
criteria.add( Example.create(u) );
List result = criteria.list();
//Process result
```

A typical use case for QBE is a search screen that allows users to specify a range of property values to be matched by the returned result set. This kind of functionality can be difficult to express cleanly in a query language; string manipulations would be required to specify a dynamic set of constraints.

Dynamic queries

It is common for queries to be built programmatically by combining several optional query criteria depending on user input. For example, a system administrator may wish to search for users by any combination of first name or last name, and to retrieve the result ordered by username. Using HQL, we could build the query using string manipulations:

```
public List findUsers(String firstname,
                      String lastname)
    throws HibernateException {

    StringBuffer queryString = new StringBuffer();
    boolean conditionFound = false;

    if (firstname != null) {
        queryString.append("lower(u.firstname) like :firstname ");
        conditionFound=true;
    }
    if (lastname != null) {
        if (conditionFound) queryString.append("and ");
        queryString.append("lower(u.lastname) like :lastname ");
        conditionFound=true;
    }

    String fromClause = conditionFound ?
                        "from User u where " :
                        "from User u ";

    queryString.insert(0, fromClause).append("order by u.username");

    Query query = getSession().createQuery( queryString.toString() );

    if (firstname != null)
        query.setString( "firstname",
                         '%' + firstname.toLowerCase() + '%' );
    if (lastname != null)
        query.setString( "lastname",
                         '%' + lastname.toLowerCase() + '%' );

    return query.list();
}
```

This code is tedious and noisy, so let us try a different approach using Criteria API

```
public List findUsers(String firstname,String lastname) throws HibernateException {

        Criteria crit = getSession().createCriteria(User.class);

        if (firstname != null) {
                crit.add( Expression.ilike("firstname",firstname, MatchMode.ANYWHERE) );
        }
        if (lastname != null) {
                crit.add( Expression.ilike("lastname",lastname, MatchMode.ANYWHERE) );
        }

        crit.addOrder( Order.asc("username") );

        return crit.list();
}
```

This code is much shorter. Note that the ilike() operator performs a case-insensitive match. There seems no doubt that this is a better approach. However, for search screens with many optional search criteria, there is an even better way. First, observe that as we add new search criteria, the parameter list of findUsers() grows. It would be better to capture the searchable properties as an object. Since all the search properties belong to the User class, why not use an instance of User.

QBE uses this idea. You provide an instance of the queried class with some properties initialized, and the query returns all persistent instances with matching property values. Hibernate implements QBE as part of the Criteria query API:

```
public List findUsers(User u) throws HibernateException {

 Example exampleUser = Example.create(u).ignoreCase().enableLike(MatchMode.ANYWHERE);
        return getSession().createCriteria(User.class)
                        .add(exampleUser)
                        .list();
}
```

The call to create () returns a new instance of Example for the given instance of User. The ignoreCase() method puts the example query into a case-insensitive mode for all string-valued properties. The call to enableLike() specifies that the SQL like operator should be used for all string-valued properties, and specifies a MatchMode.

Subqueries

Sub selects are an important and powerful feature of SQL. A sub select is a select query embedded in another query, usually in the select, from, or where clause.

HQL supports subqueries in the where clause. We cannot think of many good uses for subqueries in the from clause, although select clause subqueries might be a nice future extension. (You might remember from chapter 3 that a derived property mapping is in fact a select clause sub select.) Note that some platforms supported by Hibernate do not implement sub selects. In particular, only the latest versions of MySQL support subqueries. If you desire portability among many different databases, you should not use this feature. The result of a subquery might contain either a single row or multiple rows. Typically, subqueries that return single rows perform aggregation. The following subquery returns the total number of items sold by a user; the outer query returns all users who have sold more than 10 items:

```
from User u where 10 < (
        select count(i) from u.items i where i.successfulBid is not null
)
```

This is a correlated subquery—it refers to an alias (u) from the outer query. The next subquery is an uncorrelated subquery:

```
from Bid bid where bid.amount + 1 >= ( select max(b.amount) from Bid b )
```

The subquery in this example returns the maximum bid amount in the entire system; the outer query returns all bids whose amount is within one (dollar) of that amount. Note that in both cases, the subquery is enclosed in parentheses. This is always required.

Uncorrelated subqueries are harmless; there is no reason not to use them when convenient, although they can always be rewritten as two queries (after all, they don't reference each other). You should think more carefully about the performance impact of correlated subqueries. On a mature database, the performance cost of a simple correlated subquery is similar to the cost of a join. However, it isn't necessarily possible to rewrite a correlated subquery using several separate queries. If a subquery returns multiple rows, it's combined with quantification. ANSI SQL (and HQL) defines the following quantifiers:

- **any**
- **all**
- **some (a synonym for any)**
- **in (a synonym for = any)**

For example, the following query returns items where all bids are less than 100:

```
from Item item where 100 > all ( select b.amount from item.bids b )
```

The next query returns all items with bids greater than 100:

```
from Item item where 100 < any ( select b.amount from item.bids b )
```

This query returns items with a bid of exactly 100:

```
from Item item where 100 = some ( select b.amount from item.bids b )
```

So does this one:

```
from Item item where 100 in ( select b.amount from item.bids b )
```

HQL supports a shortcut syntax for subqueries that operate on elements or indices of a collection. The following query uses the special HQL elements() function:

```
List list = session.createQuery("from Category c  where :item in elements(c.items)")
                    .setEntity("item", item)
                     .list();
```

The query returns all categories to which the item belongs and is equivalent to the following HQL, where the subquery is more explicit:

```
List results = session.createQuery("from Category c " +
"where :item in (from c.items)")
.setEntity("item", item)
.list();
```

Along with elements(), HQL provides indices(), maxelement(), minelement(), maxindex(), minindex(), and size(), each of which is equivalent to a certain correlated subquery against the passed collection. Refer to the Hibernate documentation for more information about these special functions; they're rarely used. Subqueries are an advanced technique; you should question their frequent use, since queries with subqueries can often be rewritten using only joins and aggregation.

However, they're powerful and useful from time to time. By now, we hope you're convinced that Hibernate's query facilities are flexible, powerful, and easy to use. HQL provides almost all the functionality of ANSI standard SQL. Of course, on rare occasions you do need to resort to handcrafted SQL, especially when you wish to take advantage of database features that go beyond the functionality specified by the ANSI standard.

Hibernate Native SQL Queries

Hibernate does provide a way to use native SQL statements directly through Hibernate. One reason to use native SQL is that your database supports some special features through its dialect of SQL that are not supported in HQL. Another reason is that you may want to call stored procedures from your Hibernate application.

You can modify your SQL statements to make them work with Hibernate's ORM layer. You do need to modify your SQL to include Hibernate aliases that correspond to objects or object properties. You can specify all properties on an object with {objectname.*}, or you can specify the aliases directly with {objectname.property}. Hibernate uses the mappings to translate your object property names into their underlying SQL columns. This may not be the exact way you expect Hibernate to work, so be aware that you do need to modify your SQL statements for full ORM support. You will especially run into problems with native SQL on classes with subclasses—be sure you understand how you mapped the inheritance across either a single table or multiple tables, so that you select the right properties off the table.

Underlying Hibernate's native SQL support is the org.hibernate.SQLQuery interface, which extends theorg.hibernate.Query interface.

Your application will create a native SQL query from the session with the createSQLQuery() method on the Session interface.

```
public SQLQuery createSQLQuery(String queryString) throws HibernateException
```

Hibernate Named Queries

Named queries in hibernate is a **technique to group the HQL statements in single location**, and lately refer them by some name whenever need to use them. It **helps largely in code cleanup**because these HQL statements are no longer scattered in whole code.

Apart from above, below are some minor **advantages** of named queries

1. **Fail fast:** Their syntax is checked when the session factory is created, making the application fail fast in case of an error.
2. **Reusable**: They can be accessed and used from several places

Hibernate mapping file example

```
<hibernate-mapping>
      <query name="GET_PRODUCT_BY_PID">
            from Product p where p.pid = :pid
      </query>
</hibernate-mapping>
```

Mapping file

```
Query qry = session.getNamedQuery("GET_PRODUCT_BY_PID");
qry.setParameter("pid", new Integer(1022));
List resList = qry.getResultList();
```

Syntax Of hibernate mapping file [For Native SQL]

```
<hibernate-mapping>
      <sql-query name="GET_PRODUCTS">
            select * from PRODUCTS
      </sql-query>
</hibernate-mapping>
```

Named queries with Annotations

```
@Entity
@Table(name = "DEPARTMENT", uniqueConstraints = {@UniqueConstraint(columnNames = "ID"),
                    @UniqueConstraint(columnNames = "NAME") })
@NamedQueries
(
    {
        @NamedQuery(name=DepartmentEntity.GET_DEPARTMENT_BY_ID, query=DepartmentEntity.GET_DEPARTMENT_BY_ID_QUERY),
        @NamedQuery(name=DepartmentEntity.UPDATE_DEPARTMENT_BY_ID, query=DepartmentEntity.UPDATE_DEPARTMENT_BY_ID_QUERY)
    }
)
public class DepartmentEntity implements Serializable {

    static final String GET_DEPARTMENT_BY_ID_QUERY = "from DepartmentEntity d where d.id = :id";
    public static final String GET_DEPARTMENT_BY_ID = "GET_DEPARTMENT_BY_ID";

    static final String UPDATE_DEPARTMENT_BY_ID_QUERY = "UPDATE DepartmentEntity d SET d.name=:name where d.id = :id";
    public static final String UPDATE_DEPARTMENT_BY_ID = "UPDATE_DEPARTMENT_BY_ID";

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", unique = true, nullable = false)
    private Integer id;

    @Column(name = "NAME", unique = true, nullable = false, length = 100)
    private String name;
      //setters and getters
}
```

Working with Procedures in Hibernate

A stored procedure or in simple a proc is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.
A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.

A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

We can pass parameters to procedures in three ways.

1) IN-parameters
2) OUT-parameters
3) IN OUT-parameters

Syntax to create a procedure is:

```
CREATE [OR REPLACE] PROCEDURE proc_name    [list of parameters]
IS

BEGIN
        Execution section
EXCEPTION
        Exception section
END;
```

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

Procedure With IN Parameters

```
CREATE OR REPLACE  PROCEDURE INSERT_EMP(
        EID IN NUMBER,  ENAME IN VARCHAR2,
        ESAL  IN NUMBER, EGENDER IN CHARACTER)
AS
BEGIN
 INSERT INTO EMPLOYEES (EMP_ID,EMP_NAME,EMP_SALARY,EMP_GENDER)
        VALUES (EID,ENAME,ESAL,EGENDER);
END INSERT_EMP;
```

```
public class InsertRecordUsingProcedure{

public static void main(String[] args) {
	insertEmpRecord(101,'Ashok',75000.00,'M');
}

public static void insertEmpRecord(Integer id,String name,Double salary,Character gender) {
	SessionFactory sf = HibernateUtil.getSessionFactory();
	Session hsession = sf.openSession();
	Transaction tx = hsession.beginTransaction();

	StoredProcedureQuery query = hsession.createStoredProcedureQuery("INSERT_EMP");

	query.registerStoredProcedureParameter(0, Integer.class,ParameterMode.IN);
	query.registerStoredProcedureParameter(1, String.class,ParameterMode.IN);
	query.registerStoredProcedureParameter(2, Integer.class,ParameterMode.IN);
	query.registerStoredProcedureParameter(3, Character.class,ParameterMode.IN);

		query.setParameter(0, id);
		query.setParameter(1, name);
		query.setParameter(2, salary);
		query.setParameter(3, gender);

		query.executeUpdate();

		tx.commit();

		hsession.close();
		sf.close();
	}

}
```

InsertRecordUsingProcedure.java

Procedure with IN OUT parameters – Get_Emp_Name_By_ID

```
CREATE OR REPLACE PROCEDURE GET_EMP_NAME_BY_ID(eid in number,ename out varchar2)
 AS
BEGIN
 SELECT EMP_NAME INTO ENAME FROM EMPLOYEES WHERE EMP_ID=EID;
END GET_EMP_NAME_BY_ID;
```

```
public class CallProcDemo {

	public static void main(String[] args) {
		getEmpNameById(101);
	}
	public static void void getEmpNameById(Integer empId) {
		try {
			SessionFactory sf = HibernateUtil.getSessionFactory();
			Session hsession = sf.openSession();
			Transaction tx = hsession.beginTransaction();
			StoredProcedureQuery spQuery = hsession
					.createStoredProcedureQuery("GET_EMP_NAME_BY_ID");
			spQuery.registerStoredProcedureParameter(0, Integer.class,
					ParameterMode.IN);
			spQuery.registerStoredProcedureParameter(1, String.class,
					ParameterMode.OUT);
			spQuery.setParameter(0, empId);
			spQuery.execute();
			String name = (String) spQuery.getOutputParameterValue(1);
			System.out.println(name);
			tx.commit();
			hsession.close();
			sf.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Procedure with OUT Parameter as RefCursor – GET_ALL_EMPS

```
CREATE OR REPLACE PROCEDURE GET_ALL_EMPS

(EMPS OUT SYS_REFCURSOR)

AS

BEGIN

 OPEN EMPS FOR SELECT * FROM EMPLOYEES;

END GET_ALL_EMPS;
```

```
public class CallProcDemo {

	public static void main(String[] args) {
			SessionFactory sf = HibernateUtil.getSessionFactory();
			Session hsession = sf.openSession();
			Transaction tx = hsession.beginTransaction();
			StoredProcedureQuery spQuery = hsession
					.createStoredProcedureQuery("GET_ALL_EMPS");
			spQuery.registerStoredProcedureParameter(1, Class.class,
					ParameterMode.REF_CURSOR);
			spQuery.execute();
			List empList = spQuery.getResultList();
			Iterator itr = empList.iterator();
			while (itr.hasNext()) {
				Object[] objArr = (Object[]) itr.next();
				for (Object obj : objArr) {
					System.out.print(obj + "\t");
				}
				System.out.println();
			}
			System.out.println(empList.size());
			tx.commit();
			hsession.close();
			sf.close();
	}
}
```

Filters in Hibernate

With Hibernate3 there is a new way to filtering the results of searches. Sometimes it is required to only process a subset of the data in the underlying Database tables. Hibernate filters are very useful in those situations. Other approaches for these kind of problems is to use a database view or use a WHERE clause in the query or Hibernate Criteria API.

But Hibernate filters can be enabled or disabled during a Hibernate session. Filters can be parameterized also. This way one can manage the 'visibility' rules within the Integration tier. They can be used in the scenarios where you need to provide the capability of security roles, entitlements or personalization.

When to Use Hibernate Filters

Let's take an example, consider a web application that does the reporting for various flights. In future course there is a change in requirement such that flights are to be shown as per their status (on time, delayed or cancelled).

This can also be done using a WHERE clause within the SQL SELECT query or Hibernate's HQL SELECT query. For a small application it is okay to do this, but for a large and complex application it might be a troublesome effort. Moreover it will be like searching each and every SQL query and making the changes in the existing code which have been thoroughly tested.

This can also be done using Hibernate's Criteria API but that also means changing the code at numerous places that is all working fine. Moreover in both the approaches, one need to be very careful so that they are not changing existing working SQL queries in inadvertent way.

Filters can be used like database views, but parameterized inside the application. This way they are useful when developers have very little control over DB operations. Here I am going to show you the usage of Hibernate filters to solve this problem. When the end users select the status, your application activates the flight's status for the end user's Hibernate session. Any SQL query will only return the subset of flights with the user selected status. Flight status is maintained at two locations- Hibernate Session and flight status filter.

One of the other use cases of filters I can think of is in the user's view of the organization data. A user can only view the data that he/she is authorized to. For example an admin can see data for all the users in the organization, manager can see the data for all the employees reporting to him/her in his/her group while an employee can only see his/her data. If a user moves from one group to another-with a very minimal change using Hibernate Filters this can be implemented.

How to Use Hibernate Filters

Hibernate filters are defined in Hibernate mapping documents (hbm.xml file)-which are easy to maintain. One can programmatically turn on or off the filters in the application code. Though filters can't be created at run time, they can be parameterized which makes them quite flexible in nature. We specify the filter on the column which is being used to enable/disable visibility rules. Please go thru the example application in which the filter is applied on flight status column and it must match a named parameter. After that at run time we specify one of the possible values.

```
<hibernate-mapping>
	<class name="com.aits.Student" table="student">
		<id name="movieId" type="java.lang.Integer">
			<column name="MOVIE_ID" />
			<generator class="identity" />
		</id>
		<property name="movieName" type="string">
			<column name="MOVIE_NAME" length="10" not-null="true"
				unique="true" />
		</property>
		<property name="releasedYr" type="string">
			<column name="RELEASED_YEAR" length="20" not-null="true"/>
		</property>

		 <filter name="movieFilter" condition="RELEASED_YEAR >= :movieReleasedFilter"/>
	</class>

	<filter-def name="movieFilter">
		<filter-param name="movieReleasedFilter" type="java.lang.Integer" />
	</filter-def>

</hibernate-mapping>
```

Student.hbm.xml

Now attach the filters to class or collection mapping elements. You can attach a single filter to more than one class or collection. To do this, you add a <filter> XML element to each class or collection. The <filter> XML element has two attributes viz. name and condition. The name references a filter definition (in the sample application it's : statusFilter) while condition is analogous to a WHERE clause in HQL. Please go thru the complete hibernate mapping file from the HibernateFilters.zip archive.

Note: Each <filter> XML element must correspond to a <filter-def> element. It is possible to have more than one filter for each filter definition, and each class can have more than one filter. Idea is to define all the filter parameters in one place and then refer them in the individual filter conditions.

In the java code, we can programmatically enable or disable the filter. By default the Hibernate Session doesn't have any filters enabled on it.

The Session interface contains the following methods:

```
public Filter enableFilter(String filterName)

public Filter getEnabledFilter(String filterName)

public void disableFilter(String filterName)
```

The Filter interface contains some of the important methods:

```
public Filter setParameter(String name, Object value)

public Filter setParameterList(String name, Collection values)

public Filter setParameterList(String name, Object[] values)
```

setParameter() method is mostly used. Be careful and specify only the type of java object that you have mentioned in the parameter at the time of defining filter in the mapping file.

The two setParameterList() methods are useful for using IN clauses in your filters. If you want to use BETWEEN clauses, use two different filter parameters with different names.

At the time of enabling the filter on session-use the name that you have provided in the mapping file for the filter name for the corresponding column in the table. Similarly condition name should contain one of the possible values for that column. This condition is being set on the filter.

```
import java.util.List;

import org.hibernate.Filter;
import org.hibernate.Query;
import org.hibernate.Session;

import info.aits.HibernateUtil;

public class App {
	public static void main(String[] args) {
		Session session = HibernateUtil.getSessionFactory().openSession();
		session.beginTransaction();
		Filter filter = session.enableFilter("movieFilter");
		filter.setParameter("movieReleasedFilter", 2015);
		Query query = session.createQuery("from movie");
		List<?> list = query.list();
		for (int i = 0; i < list.size(); i++) {
			Movie mv = (Student) list.get(i);
			System.out.println(mv);
		}
		session.getTransaction().commit();
	}
}
```

Hibernate Filters with Annotations

```
import org.hibernate.annotations.Filter;
import org.hibernate.annotations.FilterDef;
import org.hibernate.annotations.ParamDef;

@Entity
@Table(name = "movie")
@FilterDef(name = "movieFilter", parameters = @ParamDef(name = "yearFilter", type = "java.lang.String"))
@Filter(name = "movieFilter", condition = "released_in_year = :yearFilter")
public class Movie {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private long id;

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

 @Column(name = "released_in_year")
 private String year;

      // setters & getters

      //toString
}
```

Movie.java

Hibernate Cache Mechanism

Caching is facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate also provide this caching functionality.

First level cache - This is enabled by default and works in session scope
Second level cache - This is apart from first level cache which is available to be used globally in session factory scope
Query cache – It is used cache the the queries

Fist level cache in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you canot disable it even forcefully.

Its easy to understand the first level cache if we understand the fact that it is associated with Session object. As we know session object is created on demand from session factory and it is lost, once the session is closed. Similarly, first level cache associated with session object is available only till session object is live. It is available to session object only and is not accessible to any other session object in any other part of application.

Understanding Hibernate First Level Cache with Example

Caching is a facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate achieves the second goal by implementing first level cache.

Fist level cache in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you cannot disable it even forcefully.

It's easy to understand the first level cache if we understand the fact that it is associated with Session object. As we know session object is created on demand from session factory and it is lost, once the session is closed. Similarly, first level cache associated with session object is available only till session object is live. It is available to session object only and is not accessible to any other session object in any other part of application.

Hibernate first level cache

- First level cache is associated with "session" object and other session objects in application can not see it.
- The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
- First level cache is enabled by default and you can not disable it.
- When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
- If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.
- The loaded entity can be removed from session using evict() method. The next loading of this entity will again make a database call if it has been removed using evict() method.
- The whole session cache can be removed using clear() method. It will remove all the entities stored in cache.

```
SessionFactory sf = HibernateUtil.getSessionFactory();
Session hsession = sf.openSession();
Transaction tx = hsession.beginTransaction();

Employee emp1 = (Employee) hsession.load(Employee.class, 1);
System.out.println(emp1);

Employee emp2 = (Employee) hsession.load(Employee.class, 1);
System.out.println(emp2);

tx.commit();
hsession.close();
sf.close();
```

In Above program when we try to load Employee object twice in same session, we can see that second "session.load()" statement does not execute select query again and load the department entity directly.

Removing cache objects from first level cache example

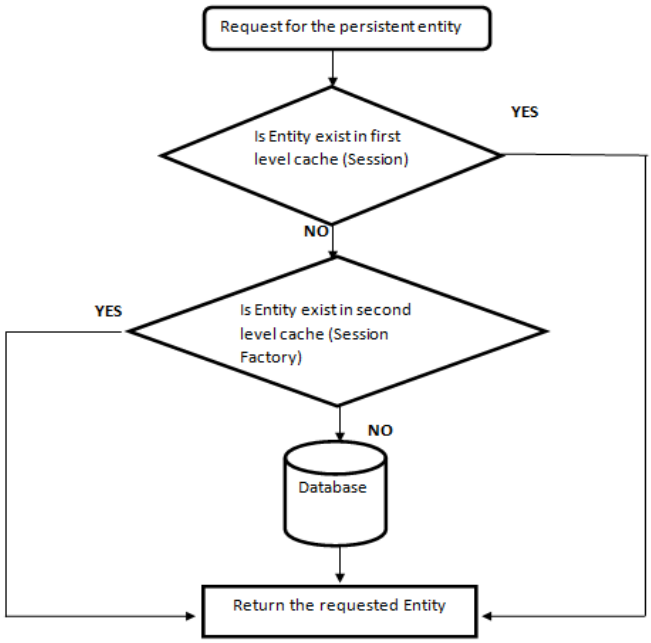
Though we cannot disable the first level cache in hibernate, but we can certainly remove some of objects from it when needed. This is done using two methods:

```
public void evict(Object obj)
public void  clear()
```

Here evict () is used to remove a particular object from cache associated with session, and clear() method is used to remove all cached objects associated with session. So they are essentially like remove one and remove all.

Second level cache

First level cache is implemented by Hibernate Framework. However, second level cache is implemented by some third party jars such as ehcache. After Hibernate 4, ehcache became default second level cache of Hibernate.



How second level cache works?

Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).

If cached copy of entity is present in first level cache, it is returned as result of load method.
If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.

If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.

Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.

If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.

Every fresh session having its own cache memory, Caching is a mechanism for storing the loaded objects into a cache memory. The advantage of cache mechanism is, whenever again we want to load the same object from the data-base then instead of hitting the database once again, it loads from the local cache memory only, so that the no. of round trips between an application and a database server got decreased. It means caching mechanism increases the performance of the application.

Second level cache in the hibernate implemented by **4** vendors…

- Easy Hibernate [EHCache] Cache from hibernate framework
- Open Symphony [OS] cache from Open Symphony
- SwarmCache
- TreeCache from JBoss

Ehcache is an **open source,** standards-based cache for boosting performance, **offloading** your database, and simplifying scalability. It's the most widely-used Java-based cache because it's **robust, proven,** and **full-featured.** Ehcache scales from in-process, with one or more nodes, all the way to mixed in-process/out-of-process configurations with terabyte-sized caches.

How to enable second level cache (EH Cache) in hibernate

To enable second level cache in the hibernate, then the following **3** changes are required

1. Add second level cache related properties in hibernate.cfg.xml file
2. Create EHCache.xml (in classpath folder)
3. Add @Cache annotation in POJO class

hibernate.cache.use_second_level_cache is used to enable second level cache, we should set hibernate.cache.use_second_level_cache property value to true, default is false.

hibernate.cache.use_query_cache is used to enable query caching, so we should set hibernate.cache.use_query_cache property value to true.

hibernate.cache.region.factory_class is used to define the Factory class for Second level caching.

EhCache will ensure that all instances of SingletonEhCacheRegionFactory use the same actual CacheManager internally, no matter how many instances of SingletonEhCacheRegionFactory you create, making it a crude version of the Singleton design pattern.

The plain EhCacheRegionFactory, on the other hand, will get a new CacheManager every time.

If you have two Hibernate session factories in Spring, each using their own instance of SingletonEhCacheRegionFactory, then they're actually going to end up sharing a lot of their cache state, which may account for your problem.

This isn't really a good match for Spring, where the singletons are supposed to be managed by the container. If you use EhCacheRegionFactory, then you'll likely get more predictable results. I suggest giving it a go and see how you get on.

If you are using Hibernate 3, corresponding classes will be net.sf.ehcache.hibernate.EhCacheRegionFactory and net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory.

net.sf.ehcache.configurationResourceName is used to define the EHCache configuration file location, it is optional parameter and if it's not used, EHCache will try to find ehcache.xml file in the classpath. Therefore, we should create ehcache.xml file in the classpath. If you're using Maven, you can add this file into resources folder.

```
<property name="hibernate.cache.use_second_level_cache">true</property>

<property name="hibernate.cache.region.factory_class">
      org.hibernate.cache.ehcache.EhCacheRegionFactory
</property>

<propertyname="net.sf.ehcache.configurationResourceName">
      ehcache.xml
</property>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="true"
         monitoring="autodetect" dynamicConfig="true">
    <diskStore path="java.io.tmpdir/ehcache" />
    <defaultCache maxEntriesLocalHeap="10000" eternal="false"
                  timeToIdleSeconds="120" timeToLiveSeconds="120" diskSpoolBufferSizeMB="30"
                  maxEntriesLocalDisk="10000000" diskExpiryThreadIntervalSeconds="120"
                  memoryStoreEvictionPolicy="LRU" statistics="true">
        <persistence strategy="localTempSwap"/>
    </defaultCache>
    <cache name="olyanren.java.cache.ehcache.model.Admin"
           maxElementsInMemory="500"
           eternal="true"
           timeToIdleSeconds="0"
           timeToLiveSeconds="100"
           overflowToDisk="false"
           />
    <cache
        name="org.hibernate.cache.StandardQueryCache"
        maxEntriesLocalHeap="5"
        eternal="false"
        timeToLiveSeconds="120">
        <persistence strategy="localTempSwap"/>
    </cache>
    <cache name="org.hibernate.cache.spi.UpdateTimestampsCache"
           maxEntriesLocalHeap="5000" eternal="true">
        <persistence strategy="localTempSwap"/>
    </cache>
</ehcache>
```

diskStore is used when cached objects to be stored in memory exceed maxEntriesLocalHeap value, then the objects which cannot be stored into memory will be saved **diskStore** path.

maxEntriesLocalHeap specifies how many entity which is serializable or not will be saved into memory.

eternal is used to decide cached entities life will be eternal or not. **Notice** that when set to "true", overrides **timeToLive** and **timeToIdle** so that no expiration can take place.

diskExpiryThreadIntervalSeconds sets the interval between runs of the expiry thread. Setting diskExpiryThreadIntervalSeconds to a low value is not recommended. It can cause excessive DiskStore locking and high CPU utilization. The default value is 120 seconds. If a cache's DiskStore has a limited size, Elements will be **evicted** from the DiskStore when it exceeds this limit. The LFU algorithm is used for these **evictions.** It is not configurable or changeable. **maxEntriesLocalDisk** is used to decide how many entity will be stored in the local disk when overflow is happend. This property works with **localTempSwap** option.

localTempSwap strategy allows the cache to overflow to disk during cache operation, providing an extra tier for cache storage. This disk storage is **temporary and is clearedafter a restart.** If the disk store path is not specified, a default path is used, and the default will be auto-resolved in the case of a conflict with another CacheManager. The TempSwap DiskStore creates a data file for each cache on startup called **"<cache_name>.data"**.

timeToIdleSeconds enables cached object to be kept in as long as it is requested in periods shorter than timeToIdleSeconds.

timeToLiveSeconds will make the cached object be invalidated after that many seconds irregardless of how many times or when it was requested.

Let's say that `timeToIdleSeconds = 3`. Object will be invalidated if it hasn't been requested for 4 seconds.

If `timeToLiveSeconds = 90` then the object will be removed from cache after 90 seconds even if it has been requested few milliseconds in the 90th second of its short life.

maxElementsInMemory="2" informs the Hibernate to place 2 records (in Hibernate style, 2 objects) in **RAM.** If the **object** size grater then specified max size, in this **case** max size is others will be in the hard disk
eternal="false" indicates the records should not be stored in hard disk permanently.
timeToIdleSeconds="120" the records may live in RAM and afterwards they will be moved to hard disk

timeToLiveSeconds="300" The maximum storage period in the hard disk will be 300 seconds and afterwards the records are permanently deleted from the hard disk (not from database table).
overflowToDisk="true" When the records retrieved are more than **maxElementsInMemory** value, the excess records may be stored in hard disk specified in the **ehcache.xml**file.

Possible **directories** where the records can be **stored:**
user.home: User's home directory
user.dir: User's current working directory
java.io.tmpdir: Default temp file path

The user.home, user.dir and java.io.tmpdir are the folders where the second-level cache records can be stored on the hard disk. The programmer should choose one.

```
@Entity
@Table(name = "EMPLOYEES")
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY)
public class Employee {

	//body
}
```

As seen above Hibernate entity, we used **@org.hibernate.annotations.Cache** annotation with **READ_WRITE** concurrency strategy. Also, we declared this entity in the ehcache.xml file.

There are four strategies we can use:

Read-only: Useful for data that is **read frequently but never updated** (e.g. referential data like Countries). It is simple. It has the best performances of all (obviously).

Read/write: Desirable if your data **needs to be updated**. But it doesn't provide a SERIALIZABLE isolation level, phantom reads can occur (you may see at the end of a transaction something that wasn't there at the start). It has more overhead than read-only.

Nonstrict read/write: Alternatively, if it's unlikely two separate transaction threads could update the same object, you may use the nonstrict–read–write strategy. It has less overhead than read-write. This one is useful for data that are **rarely updated**.

Transactional: If you need a fully transactional cache. Only **suitable in a JTA** environment.

```
Session session = HibernateUtility.getSessionFactory().openSession();
session.beginTransaction();
Employee emp = null;

emp = (Employee) session.load(Employee.class, 11);
System.out.println("Employee from the Database => "+Employee);
System.out.println("Going to print Employee *** from First Level Cache");
// second time loading same entity from the first level cache
emp = (Employee) session.load(Employee.class, 11);
System.out.println(emp);
// removing Employee object from the first level cache.
session.evict(emp);
System.out.println("Object removed from the First Level Cache");
System.out.println("Going to print Employee *** from Second level Cache");
Employee = (Employee) session.load(Employee.class, 11);
System.out.println(emp);
session.getTransaction().commit();
// loading object in another session
Session session2 = HibernateUtility.getSessionFactory().openSession();
session2.beginTransaction();
System.out.println();
System.out.println("Printing Employee *** from Second level");
emp = (Employee) session2.load(Employee.class, 11);
System.out.println(emp);
session2.getTransaction().commit();
```

Query caching

The Query Cache does not cached the entities unlike Second Level Cache; it cached the queries and the return identifiers of the entities. Once it cached the identifiers than, It took help from Hibernate Second level cache to load the entities based on the identifiers value. So Query cache always used with Second level cache to get better result, because only query cache will cached the identifiers not the complete entities.

For applications that perform many queries and few inserts, deletes, or updates, caching queries can have an impact on performance. However, if the application performs many writes, the query cache will not be utilized efficiently. Hibernate expires a cached query result set when there is any insert, update, or delete of any row of a table that appears in the query.

Just as not all classes or collections should be cached, not all queries should be cached or will benefit from caching. For example, if a search screen has many different search criteria, then it's unlikely that the user will choose the same criterion twice. In this case, the cached query results won't be utilized, and we'd be better off not enabling caching for that query.

Note that the query cache does not cache the entities returned in the query result set, just the identifier values. Hibernate will, however, fully cache the value typed data returned by a projection query. For example, the projection query "select u, b.created from User u, Bid b where b.bidder = u" will result in caching of the identifiers of the users and the date object when they made their bids. It's the responsibility of the second-level cache (in conjunction with the session cache) to cache the actual state of entities. So, if the cached query you just saw is executed again, Hibernate will have the bid-creation dates in the query cache but perform a lookup in the session and second-level cache (or even execute SQL again) for each user that was in the result. This is similar to the lookup strategy of iterate(), as explained in the previous section.

The query cache must be enabled using a Hibernate property setting:

<property key="hibernate.cache.use_query_cache">true</property>

However, this setting alone isn't enough for Hibernate to cache query results. By default, Hibernate queries always ignore the cache. To enable query caching for a particular query (to allow its results to be added to the cache, and to allow it to draw its results from the cache), you use the Query interface:

Where queries are defined in our code, add the method call **setCacheable (true)** to the queries that should be cached:

```
Session hsession = sessionFactory.openSession();
Query query = hsession.createQuery("...")
query.setCacheable(true);
List list = query.getResultList();
```

```
public class QueryCacheApp {
 public static void main(String[] args) {
  System.out.println(getCountry(1));
  System.out.println("Fetch Country from Query Cache*************");
  System.out.println(getCountry(1));
 }

 public static Country getCountry(long id) {
  Session session = HibernateUtility.getSessionFactory().openSession();
  Query query = session.createQuery("from Country c where c.cId = :cId ");
  query.setParameter("cId", id);
  query.setMaxResults(1);
  query.setCacheable(true);
  return query.list() == null ? null : (Country) query.list().get(0);
 }
}
```

In the above program, although we are calling getCountry(1) methods twice in main method, but it will generate only one sql query and next time it will fetched the records from the query cached and second level cache.

Inheritance Mapping In Hibernate

Java, being a OOPs language, supports inheritance for reusability of classes. Hibernate comes with a provision to create tables and populate them as per the Java classes involved in inheritance. Suppose if we have base and derived classes, now if we save derived class object, then base class object too will be stored into the database. In order to do so, you need to choose certain mapping strategy based on your needs to save objects.

A simple strategy for mapping classes to database tables might be “one table for every entity persistent class.” This approach sounds simple enough and, indeed, works well until we encounter inheritance.

Inheritance is such a visible structural mismatch between the object-oriented and relational worlds because object-oriented systems model both is-a and has-a relationships. SQL-based models provide only has a relationships between entities;

Relational databases don’t have a straightforward way to map class hierarchies onto database tables. To address this, the Hibernate provides several strategies:

Hibernate supports 3 types of Inheritance strategies to map classes involved in inheritance with database tables.

- Table Per Class Hierarchy
- Table Per Concrete class
- Table Per Subclass

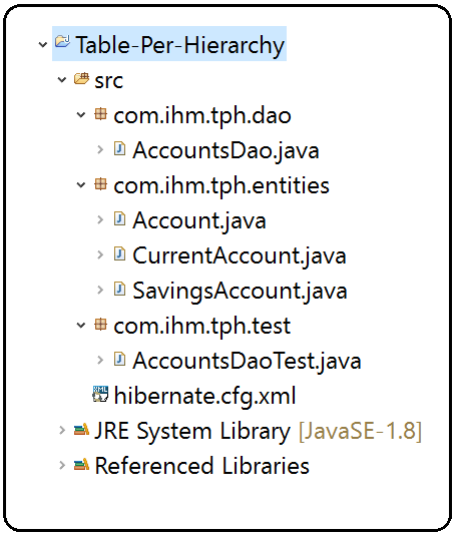
Table per class hierarchy (Single Table)

An entire class hierarchy can be mapped to a single table. This table includes columns for all properties of all classes in the hierarchy. Since the records for all entities will be in the same table. As all the data goes in one table, a discriminator is used to differentiate between different types of data. By default, this is done through a discriminator column which has the name of the entity as a value. To customize the discriminator column, we can use the **@DiscriminatorColumn** annotation:

This mapping strategy is a winner in terms of both performance and simplicity. It is the best-performing way to represent polymorphism—both polymorphic and no polymorphic queries perform well—and it is even easy to implement by hand. Ad-hoc reporting is possible without complex joins or unions. Schema evolution is straightforward.

Below example demonstrates Table per Hierarchy

Step-1: Create a project and add required libraries (Hibernate + jdbc driver)



Step-2: Create Entity classes Account.java, SavingsAccount.java and CurrentAccount.java

```
@Entity
@Table(name = "ACCOUNT_DETAILS")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "ACC_TYPE", discriminatorType =
DiscriminatorType.STRING)
public class Account {

	@Id
	@GeneratedValue
	@Column(name = "ACC_ID")
	private Integer accId;

	@Column(name = "BANK_NAME")
	private String bankName;

	@Column(name = "BRANCH_NAME")
	private String branchName;

	@Column(name = "HOLDER_NAME")
	private String holderName;

	//setters & getters

}
```

Account.java

```
@Entity
@DiscriminatorValue("C")
public class CurrentAccount extends Account {

	@Column(name = "TX_LIMIT")
	private Double txLimit;

	//setters & getters

}
```

CurrentAccount.java

```
@Entity
@DiscriminatorValue("S")
public class SavingsAccount extends Account {

	@Column(name = "MIN_BAL")
	private Double minBal;

	//setters & getters

}
```

SavingsAccount.java

Step-3: Creating hibernate configuration file with DB properties, hibernate properties and annotated classes details

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
                "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
                "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
        <session-factory>
                <property name="hibernate.connection.driver_class">
                        oracle.jdbc.driver.OracleDriver
                </property>
                <property name="hibernate.connection.url">
                        jdbc:oracle:thin:@localhost:1521/XE
                </property>
                <property name="hibernate.connection.username">
                        system
                </property>
                <property name="hibernate.connection.password">
                        admin
                </property>

                <property name="show_sql">true</property>
                <property name="format_sql">true</property>
                <property name="hbm2ddl.auto>update</property>

                <mapping class="com.nit.entities.Account" />
                <mapping class="com.nit.entities.SavingsAccount" />
                <mapping class="com.nit.entities.CurrentAccount" />


        </session-factory>
</hibernate-configuration>
```

hibernate.cfg.xml

Step-4: Create Dao class and its corresponding test class

```
public class AccountDao {

        public void insert(SavingsAccount sa) {
                Session hs = HibernateUtils.getSession();
                Transaction tx = hs.beginTransaction();
                Serializable id = hs.save(sa);
                System.out.println("Record inserted : " + id);
                tx.commit();
                hs.close();
        }

        public void insert(CurrentAccount ca) {
                Session hs = HibernateUtils.getSession();
                Transaction tx = hs.beginTransaction();
                Serializable id = hs.save(ca);
                System.out.println("Record inserted : " + id);
                tx.commit();
                hs.close();
        }

}
```

AccountDao.java

```
public class AccountDaoTest {

        public static void main(String[] args) {

                AccountDao dao = new AccountDao();

                /*SavingsAccount sa = new SavingsAccount();
                sa.setBankName("Axis");
                sa.setBranchName("S.R.Nagar");
                sa.setHolderName("John");
                sa.setMinBal(500.00);

                dao.insert(sa);*/
                CurrentAccount ca = new CurrentAccount();
                ca.setBankName("ICICI");
                ca.setBranchName("S.R.Nagar");
                ca.setHolderName("Suman");
                ca.setTxLimit(40000.00);
                dao.insert(ca);

        }
}
```

AccountDaoTest.java

Advantages of Single Table per class hierarchy

- ✓ Simplest to implement.
- ✓ Only one table to deal with.
- ✓ Performance wise better than all strategies because no joins or sub-selects need to be performed.

Disadvantages:

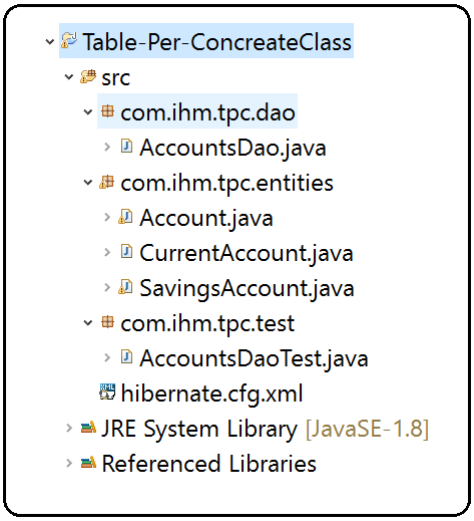
- Most of the column of table are nullable so the NOT NULL constraint cannot be applied.
- Tables are not normalized.

Table per Concrete Class

The Table per Concrete Class strategy maps each entity to its table which contains all the properties of the entity, including the ones inherited. In this case every entity class has its own table i.e. table per class.

Example on Table per Concreate class

Step-1: Create a project and add required libraries (hibernate+jdbc driver)



Step-2: Create Entity classes

```
@Entity
@Table(name = "ACCOUNT_DETAILS")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Account {

		@Id
		@GeneratedValue
		@Column(name = "ACC_ID")
		private Integer accId;

		@Column(name = "HOLDER_NAME")
		private String holderName;

		@Column(name = "BRANCH_NAME")
		private String branchName;

		@Column(name = "OPEN_DT")
		@Temporal(TemporalType.DATE)
		private Date openDt;

		//setters & getters

}
```

Account.java

```
@Entity
@Table(name = "CURRENT_ACCOUNT")
public class CurrentAccount extends Account {

	@Column(name = "TX_LIMIT")
	private Double txLimit;

	//setters & getters
}
```

CurrentAccount.java

```
@Entity
@Table(name = "SAVINGS_ACCOUNT")
public class SavingsAccount extends Account {

		@Column(name = "MIN_BAL")
		private Double minBal;

		//setters & getters

}
```

SavingsAccount.java

Step-3: Create a Dao class and test it

```
public class AccountDao {

	public void insert(SavingsAccount sa) {
		Session hs = HibernateUtils.getSession();
		Transaction tx = hs.beginTransaction();
		Serializable id = hs.save(sa);
		System.out.println("Record inserted : " + id);
		tx.commit();
		hs.close();
	}

	public void insert(CurrentAccount ca) {
		Session hs = HibernateUtils.getSession();
		Transaction tx = hs.beginTransaction();
		Serializable id = hs.save(ca);
		System.out.println("Record inserted : " + id);
		tx.commit();
		hs.close();
	}

}
```

AccountDao.java

```
public class AccountDaoTest {

	public static void main(String[] args) {

		AccountDao dao = new AccountDao();

		/*SavingsAccount sa = new SavingsAccount();
		sa.setBankName("Axis");
		sa.setBranchName("S.R.Nagar");
		sa.setHolderName("John");
		sa.setMinBal(500.00);

		dao.insert(sa);*/
		CurrentAccount ca = new CurrentAccount();
		ca.setBankName("ICICI");
		ca.setBranchName("S.R.Nagar");
		ca.setHolderName("Suman");
		ca.setTxLimit(40000.00);
		dao.insert(ca);

	}
}
```

AccountDaoTest.java

This strategy is not popular and also have been made optional in Java Persistence API.

Advantage:

- ✓ Possible to define NOT NULL constraints on the table.

Disadvantage:

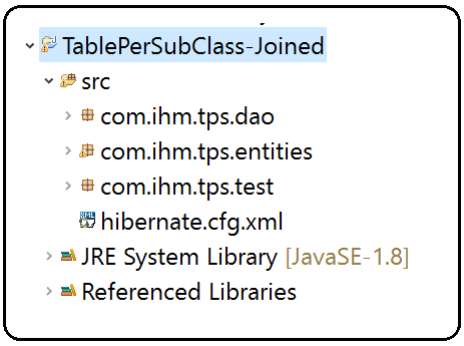
- Tables are not normalized.
- To support polymorphism either container has to do multiple trips to database or use SQL UNION kind of feature.

Note: In this case there no need for the discriminator column because all entity has own table.

Table per Sub Class (Joined)

Using this strategy, each class in the hierarchy is mapped to its table. The only column which repeatedly appears in all the tables is the identifier, which will be used for joining them when needed.

Step-1: Create a project and add the required libraries (Hibernate + JDBC Driver)



Step-2: Create Entity classes Employee.java, PermanentEmployee.java and ContratEmployee.java

```java
@Entity
@Table(name = "EMP_DETAILS")
@Inheritance(strategy = InheritanceType.JOINED)
public class Employee {

	@Id
	@GeneratedValue
	@Column(name = "emp_id")
	private Integer empId;

	@Column(name = "emp_name")
	private String empName;

	@Column(name = "EMP_PHNO")
	private Long empPhno;

	//setters & getters

}
```

Employee.java

```java
@Entity
@Table(name = "PERMANENT_EMP")
public class PermanentEmployee extends Employee{

	@Column(name = "COMPANY_NAME")
	private String companyName;

	//setters & getters
}
```

PermanentEmployee.java

```java
@Entity
@Table(name = "CONTRACT_EMP")
public class ContractEmployee extends Employee {

	@Column(name = "CONTRACTOR_NAME")
	private String contractorName;

	//setters & getters
}
```

ContractEmployee.java .

Step-3: Create Dao class and its test class

```java
public class EmpDao {

	public void insert(PermanentEmployee pe) {
		Session hs = HibernateUtils.getSession();
		Transaction tx = hs.beginTransaction();
		Serializable id = hs.save(pe);
		System.out.println("Record inserted : " + id);
		tx.commit();
		hs.close();
	}

	public void insert(ContractEmployee pe) {
		Session hs = HibernateUtils.getSession();
		Transaction tx = hs.beginTransaction();
		Serializable id = hs.save(pe);
		System.out.println("Record inserted : " + id);
		tx.commit();
		hs.close();
	}

}
```

EmpDao.java

```java
public class EmpDaoTest {

	public static void main(String[] args) {
		EmpDao dao = new EmpDao();
		PermanentEmployee pe = new PermanentEmployee();
		pe.setCompanyName("IBM");
		pe.setEmpName("Dean");
		pe.setEmpPhno(797979791);
		dao.insert(pe);

		ContractEmployee ce = new ContractEmployee();
		ce.setEmpName("Catlin");
		ce.setEmpPhno(797979791);
		ce.setContractorName("Magna Infotech");
		dao.insert(ce);
	}
}
```

EmpDaoTest.java

The disadvantage of this inheritance mapping method is that retrieving entities requires joins between tables, which can result in lower performance for large numbers of records.

The number of joins is higher when querying the parent class as it will join with every single related child – so performance is more likely to be affected the higher up the hierarchy we want to retrieve records.

Choosing a Strategy

Choosing the right inheritance strategy is not an easy task. As so often, you have to decide which advantages you need and which drawback you can accept for your application. Here are a few recommendations:

If you require the best performance and need to use polymorphic queries and relationships, you should choose the single table strategy. But be aware, that you can't use not null constraints on subclass attributes which increase the risk of data inconsistencies.

If data consistency is more important than performance and you need polymorphic queries and relationships, the joined strategy is probably your best option.

If you don't need polymorphic queries or relationships, the table per class strategy is most likely the best fit. It allows you to use constraints to ensure data consistency and provides an option of polymorphic queries. But keep in mind, that polymorphic queries are very complex for this table structure and that you should avoid them.

Association Mapping or Relationships in Hibernate

In today's connected world, databases are invaluable. They store data for many purposes, from people organizing their record collections to Amazon managing products and customers to the FBI tracking criminals and terrorists.

When we are developing an application, instead of storing complete data into single table we will store the data into different tables.

A relationship is established between two database tables when one table has a foreign key that references the primary key of another table. This is the basic concept behind the term relational database.

How a Foreign Key Works to Establish a Relationship?

Let's review the basics of primary and foreign keys. A primary key uniquely identifies each record in the table. It is a type of candidate key that is usually the first column in a table and can be automatically generated by the database to ensure that it is unique.

A foreign key is another candidate key (not the primary key) used to link a record to data in another table.

For example, consider these two tables that identify which teacher teaches which course.

Here, the Courses table's primary key is Course_ID. Its foreign key is Teacher_ID:

Courses

Course_ID	Course_Name	Teacher_ID
Course_001	Biology	Teacher_001
Course_002	Math	Teacher_001
Course_003	English	Teacher_003

You can see that the foreign key in Courses matches a primary key in Teachers:

Teachers

Teacher_ID	Teacher_Name
Teacher_001	Carmen
Teacher_002	Veronica
Teacher_003	Jorge

We can say that the **Teacher_ID** foreign key has helped to establish a relationship between the **Courses** and the **Teachers** tables.

Types of Database Relationships

Using foreign keys, or other candidate keys, you can implement three types of relationships between tables:

One-to-one: This type of relationship allows only one record on each side of the relationship.

The primary key relates to only one record – or none – in another table. For example, in a marriage, each spouse has only one other spouse. This kind of relationship can be implemented in a single table and therefore does not use a foreign key.

One-to-many: A one-to-many relationship allows a single record in one table to be related to multiple records in another table.

Consider a business with a database that has Customers and Orders tables.

A single customer can purchase multiple orders, but a single order could not be linked to multiple customers. Therefore the Orders table would contain a foreign key that matched the primary key of the Customers table, while the Customers table would have no foreign key pointing to the Orders table.

Many-to-many: This is a complex relationship in which many records in a table can link to many records in another table. For example, our business probably needs not only Customers and Orders tables, but likely also needs a Products table.

Again, the relationship between the Customers and Orders table is one-to-many, but consider the relationship between the Orders and Products table. An order can contain multiple products, and a product could be linked to multiple orders: several customers might submit an order that contains some of the same products. This kind of relationship requires a minimum three tables.

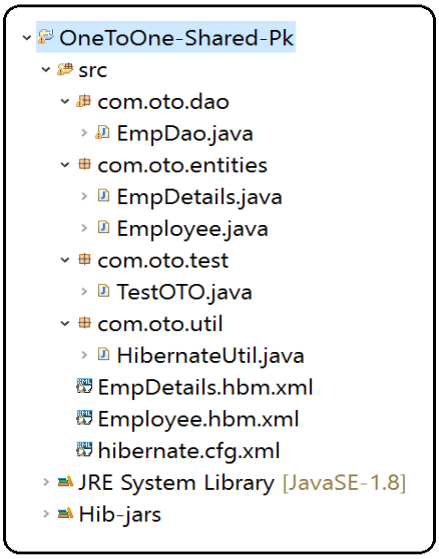
The mapping of associations between entity classes and the relationships between tables is the soul of ORM. Following are the four ways in which the cardinality of the relationship between the objects can be expressed. An association mapping can be unidirectional as well as bidirectional.

One to One Uni-Directional Mapping with Shared Primary Key using HBM – Application

- One to One Unidirectional Shared Primary Key
- A one-to-one mapping means that one object can have only one relation – at most.
- With shared primary key, the primary key of both tables are equal. The foreign key constraint is the primary key of the reference table. (**Two tables share the same primary key**).
- A unidirectional relationship means that only one side (the owning side) is able to navigate to the relationship. In the following example only the Employee to Employee Details navigation is possible but not vice versa.

Below Example depicts Employee and Employee_Details will share same primary key EMP_ID

Step-1: Create a project and add required jar files



Step-2: Create Entity classes

```
public class Employee {

        private Integer empId;
        private String empName;
        private String email;

        private EmpDetails empDetails;

        //setters & getters
}
```
Employee.java

```
public class EmpDetails {

        private Integer empId;
        private String fname;
        private String lname;
        private String mname;

        private Employee emp;

        //setters && getters
}
```
EmpDetails.java

Step-3: Create mapping files for entity classes

```
<hibernate-mapping>
  <class name="com.oto.entities.Employee" table="EMPLOYEE">
    <id name="empId" type="java.lang.Integer">
      <column name="EMP_ID" />
      <generator class="increment" />
    </id>
    <property name="empName" type="java.lang.String">
      <column name="EMP_NAME" />
    </property>
    <property name="email" type="java.lang.String">
      <column name="EMAIL" />
    </property>
    <one-to-one name="empDetails"
          class="com.oto.entities.EmpDetails"
          cascade="all" />
  </class>
</hibernate-mapping>
```
Employee.hbm.xml

```
<hibernate-mapping>
  <class name="com.oto.entities.EmpDetails" table="EMPDETAILS">
    <id name="empId" type="java.lang.Integer">
      <column name="EMP_ID" />
      <generator class="foreign">
          <param name="property">emp</param>
      </generator>
    </id>
    <property name="fname" type="java.lang.String">
      <column name="FNAME" />
    </property>
    <property name="lname" type="java.lang.String">
      <column name="LNAME" />
    </property>
    <property name="mname" type="java.lang.String">
      <column name="MNAME" />
    </property>
    <one-to-one
    name="emp"
    class="com.oto.entities.Employee"/>
  </class>
</hibernate-mapping>
```
EmpDetails.hbm.xml

Step-4: Create Dao class

```
public class EmpDao {

        public void insertEmpData() {
                Session hs = HibernateUtil.getSession();
                Transaction tx = hs.beginTransaction();

                Employee emp = new Employee();
                emp.setEmpName("Raj");  emp.setEmail("raj@in.com");

                EmpDetails edet = new EmpDetails();
                edet.setFname("Raj");   edet.setLname("kumar");
                edet.setMname("B");

                // associate parent to child and child to parent
                emp.setEmpDetails(edet);
                edet.setEmp(emp);

                hs.save(emp);

                tx.commit();
                hs.close();

        }

        public void findByEid(int eid) {
                Session hs = HibernateUtil.getSession();
                Employee e = hs.get(Employee.class, eid);
                hs.close();
        }
}
```
EmpDao.java

Step-5: Create Test class for Testing Dao methods functionality

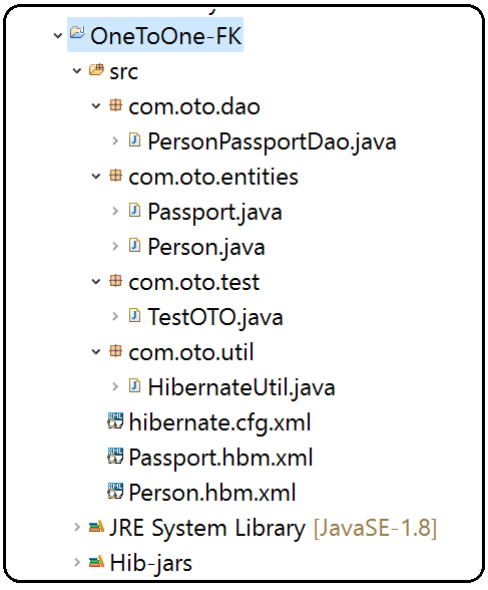
```
public class TestOTO {
	public static void main(String[] args) throws Exception {

		EmpDao dao = new EmpDao();

		// dao.insertEmpData();

		// Getting Employee By Id
		dao.findByEid(1);

		// Closing Session Factory
		HibernateUtil.closeSf();
	}
}
```

TestOTO.java

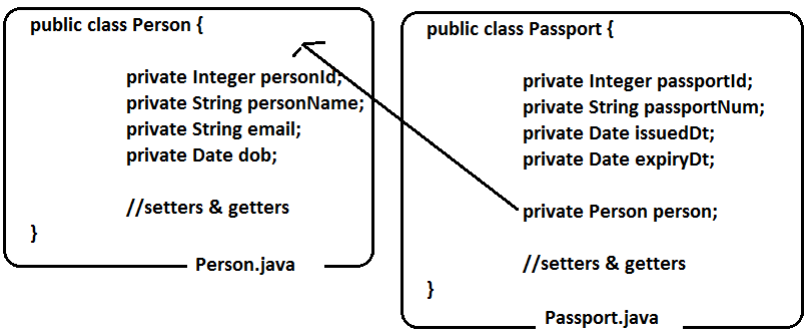
One to one Relationship with Foreign Key – Application

- A one-to-one mapping means that one object can have only one relation – at most.
- In a foreign key association, one table has a foreign key column that references the primary key of the associated table. In the following example PASSPORT table will contain PERSON_ID references from PERSON table.
- A unidirectional relationship means that only one side (the owning side) is able to navigate to the relationship.

Step-1: Create a project and add required libraries (Hibernate + JDBC Driver)



Step-2: Create Entity classes (Person.java and Passport.java)



Step-3: Create Mapping files

```
<hibernate-mapping>
<class name="com.oto.entities.Person" table="PERSON">
        <id name="personId" type="java.lang.Integer">
                <column name="PERSON_ID" />
                <generator class="increment" />
        </id>
        <property name="personName" type="java.lang.String">
                <column name="PERSON_NAME" />
        </property>
        <property name="email" type="java.lang.String">
                <column name="EMAIL" />
        </property>
        <property name="dob" type="java.util.Date">
                <column name="DOB" />
        </property>
</class>
</hibernate-mapping>
```

Person.hbm.xml

```
<hibernate-mapping>
        <class name="com.oto.entities.Passport" table="PASSPORT">
                <id name="passportId" type="java.lang.Integer">
                        <column name="PASSPORT_ID" />
                        <generator class="increment" />
                </id>
                <property name="passportNum" type="java.lang.String">
                        <column name="PASSPORT_NUM" />
                </property>
                <property name="issuedDt" type="java.util.Date">
                        <column name="ISSUED_DT" />
                </property>
                <property name="expiryDt" type="java.util.Date">
                        <column name="EXPIRY_DT" />
                </property>

                <many-to-one name="person"
                class="com.oto.entities.Person"
                unique="true" not-null="true" cascade="all"
                column="PERSON_ID"/>
        </class>
</hibernate-mapping>
```

Passport.hbm.xml

Step-4: Create Dao class to perform Persistence operations

```
public class PersonPassportDao {

        public void savePersonWithPassport() throws Exception {
                Session hs = HibernateUtil.getSession();
                Transaction tx = hs.beginTransaction();

                SimpleDateFormat sdf = new SimpleDateFormat("dd/MMM/yyyy");

                Passport passport = new Passport();
                passport.setPassportNum("JHLJL79979");
                passport.setIssuedDt(new Date());
                passport.setExpiryDt(sdf.parse("01/Jan/2028"));

                Person p = new Person();
                p.setDob(sdf.parse("01/Jun/1995"));
                p.setEmail("smith@in.com");
                p.setPersonName("Smith");

                passport.setPerson(p);

                hs.save(passport);

                tx.commit();
                hs.close();
        }

}
```

PersonPassportDao.java

Step-5: Create Test class to test Dao method functionality

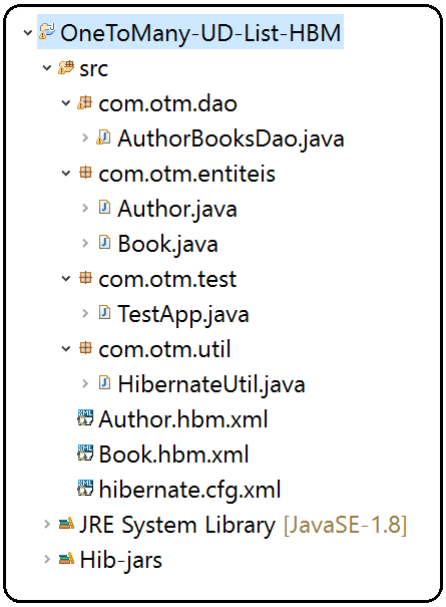
```
public class TestOTO {
	public static void main(String[] args) throws Exception {

		PersonPassportDao dao = new PersonPassportDao();

		// Calling dao method
		dao.savePersonWithPassport();

		// Closing SessionFactory
		HibernateUtil.closeSf();
	}
}
```

TestOTO.java

One to Many with Uni-Directional Using List – HBM – Application

- One To Many association represents one record in parent table can contain multiple records in child table. In Below example One Author can contain multiple Book records.
- In One to many unidirectional association, we will only be able to navigate in one direction. In Below example we will be able to navigate only from Author to Book but vice versa is not possible.
- Only one side of the association will have the reference to the other side. The one side of the association will implement one of the collection interfaces, if it has the reference to the other entity.
- **List** guarantees order of child records using new column in the child table. By default index will start from zero (If needed we can customize that index value).
- List mapping requires the addition of an index column to the database table. The index column defines the position of the element in the collection. Thus, Hibernate can preserve the ordering of the collection elements when retrieving the collection from the database.

Step-1: Create Project and add required libraries (Hibernate + JDBC driver)



Step-2: Create Entity classes

```
public class Author {

	private Integer authorId;
	private String authorName;
	private String email;

	private List<Book> books;

	//setters & getters

}
```

Author.java

```
public class Book {

	private Integer bookId;
	private String bookName;
	private String isbn;
	private Double price;
	//setters && getters
}
```

Book.java

Step-3: Create Mapping files

```
<hibernate-mapping>
  <class name="com.otm.entiteis.Author" table="AUTHOR">
    <id name="authorId" type="java.lang.Integer">
      <column name="AUTHORID" />
      <generator class="assigned" />
    </id>
    <property name="authorName" type="java.lang.String">
      <column name="AUTHORNAME" />
    </property>
    <property name="email" type="java.lang.String">
      <column name="EMAIL" />
    </property>
    <list name="books" inverse="false" table="BOOK" >
      <key>
        <column name="AUTHORID" />
      </key>
      <list-index></list-index>
      <one-to-many class="com.otm.entiteis.Book" />
    </list>
  </class>
</hibernate-mapping>
```

Author.hbm.xml

```
<hibernate-mapping>
  <class name="com.otm.entiteis.Book" table="BOOK">
    <id name="bookId" type="java.lang.Integer">
      <column name="BOOKID" />
      <generator class="assigned" />
    </id>
    <property name="bookName" type="java.lang.String">
      <column name="BOOKNAME" />
    </property>
    <property name="isbn" type="java.lang.String">
      <column name="ISBN" />
    </property>
    <property name="price" type="java.lang.Double">
      <column name="PRICE" />
    </property>
  </class>
</hibernate-mapping>
```

Book.hbm.xml

Step-4: Create Dao class

```
public class AuthorBooksDao {

	public boolean saveAuthorWithBooks() {
		boolean isInserted = false;  Session hs = null;  Transaction tx = null;
		try {
			hs = HibernateUtil.getSession();   tx = hs.beginTransaction();

			// Insert Author object
			Author a = new Author();
			a.setAuthorName("Gaven King");
			a.setEmail("gvs@sun.com");

			Book b1 = new Book(); b1.setBookName("JSE"); b1.setIsbn("ISBN001");
			b1.setPrice(100.00);

			Book b2 = new Book(); b2.setBookName("JEE"); b2.setIsbn("ISBN002");
			b2.setPrice(200.00);

			List<Book> booksSet = new ArrayList<Book>();
			booksSet.add(b1); booksSet.add(b2);

			a.setBooks(booksSet);

			Serializable id = hs.save(a);
			if (id != null) {
			    isInserted = true;
			}
			tx.commit();
		} catch (Exception e) {
			e.printStackTrace();  tx.rollback();
		} finally {
			if (hs != null)  hs.close();
		}
		return isInserted;
	}
}
```

```
public Author findAuthorById(Integer authorId) {
        Session hs = null;
        Author a = null;
        try {
            hs = HibernateUtil.getSession();
            a = hs.get(Author.class, authorId);
        } catch (Exception e) {
                e.printStackTrace();
        } finally {
                if (hs != null) hs.close();
        }
        return a;
}
```

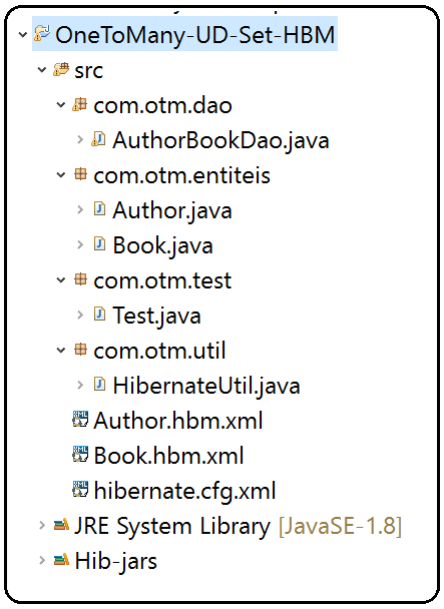
```
public void deleteAllChilds(Integer aid) {
        Session hs = null;
        Transaction tx = null;
        try {
            hs = HibernateUtil.getSession();
            tx = hs.beginTransaction();
             Author a = hs.get(Author.class, aid);
             List<Book> books = a.getBooks();
              books.clear();
               tx.commit();
        } catch (Exception e) {
                e.printStackTrace();
                if (tx != null)   tx.rollback();
        } finally {
                if (hs != null)  hs.close();
        }
}
```

```
public void deleteBookById(Integer aid, Integer bid) {
        Session hs = null;
        Transaction tx = null;
        try {
                hs = HibernateUtil.getSession();
                tx = hs.beginTransaction();
                Author a = hs.get(Author.class, aid);
                List<Book> books = a.getBooks();
                Book b = hs.get(Book.class, bid);
                books.remove(b);
                tx.commit();
        } catch (Exception e) {
                e.printStackTrace();
                if (tx != null) tx.rollback();
        } finally {
                if (hs != null) hs.close();
        }
}
```

Step-5: Create Test class for testing Dao class methods

```
public class TestApp {

        public static void main(String[] args) {
                AuthorBooksDao dao = new AuthorBooksDao();
                dao.saveAuthorWithBooks();
                // dao.findAuthorWithBooks(1);

                // dao.deleteBookById(1, 2);

                //dao.deleteAllChilds(1);

                // closing session factory
                HibernateUtil.closeSf();
        }
}
```

One to Many – Uni Directional – Set – HBM – Application




```
public class Author {
        private Integer aid;
        private String name;
        private String email;
        private Set<Book> books;
        //setters & getters
}
```

Author.java

```
public class Book {
        private Integer bid;
        private String name;
        private Double price;
        private String isbn;
        //setters & getters
}
```

Book.java

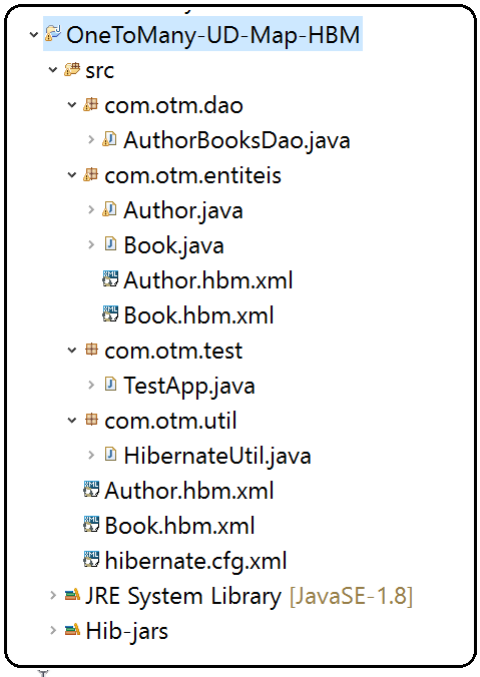
```
<hibernate-mapping>
        <class name="com.otm.entiteis.Author" table="AUTHOR_DETAILS">
                <id name="aid" type="java.lang.Integer">
                        <column name="A_ID" />
                        <generator class="increment" />
                </id>
                <property name="name" type="java.lang.String">
                        <column name="A_NAME" />
                </property>
                <property name="email" type="java.lang.String">
                        <column name="A_EMAIL" />
                </property>
                <set name="books" table="BOOK_DETAILS" cascade="all">
                        <key>
                            <column name="A_ID" />
                        </key>
                        <one-to-many class="com.otm.entiteis.Book" />
                </set>
        </class>
</hibernate-mapping>
```

Author.hbm.xml

```
<hibernate-mapping>
	<class name="com.otm.entiteis.Book" table="BOOK_DETAILS">
		<id name="bid" type="java.lang.Integer">
			<column name="B_ID" />
			<generator class="increment" />
		</id>
		<property name="name" type="java.lang.String">
			<column name="B_NAME" />
		</property>
		<property name="price" type="java.lang.Double">
			<column name="PRICE" />
		</property>
		<property name="isbn" type="java.lang.String">
			<column name="ISBN" />
		</property>
	</class>
</hibernate-mapping>
```

Book.hbm.xml

One to Many – Uni-Directional – Map – HBM -Application




```
public class Author {

	private Integer authorId;
	private String authorName;
	private String email;

	private Map<String, Book> books;

	//setters & getter

}
```

Author.java

```
public class Book {

	private Integer bookId;
	private String bookName;
	private String isbn;
	private Double price;
	//setters & getters
}
```

Book.java

```xml
<hibernate-mapping>
  <class name="com.otm.entiteis.Author" table="AUTHOR">
    <id name="authorId" type="java.lang.Integer">
      <column name="AUTHORID" />
      <generator class="assigned" />
    </id>
    <property name="authorName" type="java.lang.String">
      <column name="AUTHORNAME" />
    </property>
    <property name="email" type="java.lang.String">
      <column name="EMAIL" />
    </property>
    <map name="books" table="BOOK" lazy="true">
      <key>
        <column name="AUTHORID" />
      </key>
      <map-key type="java.lang.String"></map-key>
      <one-to-many class="com.otm.entiteis.Book" />
    </map>
  </class>
</hibernate-mapping>
```

Author.hbm.xml

```xml
<hibernate-mapping>
  <class name="com.otm.entiteis.Book" table="BOOK">
    <id name="bookId" type="java.lang.Integer">
      <column name="BOOKID" />
      <generator class="assigned" />
    </id>
    <property name="bookName" type="java.lang.String">
      <column name="BOOKNAME" />
    </property>
    <property name="isbn" type="java.lang.String">
      <column name="ISBN" />
    </property>
    <property name="price" type="java.lang.Double">
      <column name="PRICE" />
    </property>
  </class>
</hibernate-mapping>
```

Book.hbm.xml

```
public class AuthorBooksDao {

	public boolean saveAuthorWithBooks() {
		boolean isInserted = false;
		Session hs = null; Transaction tx = null;
		try {
			hs = HibernateUtil.getSession();
			tx = hs.beginTransaction();

			// Insert Author object
			Author a = new Author();a.setAuthorName("Gaven King");
			a.setEmail("gvs@sun.com");

			a.setBooks(booksMap);

			Serializable id = hs.save(a);
			if (id != null) {
				isInserted = true;
			}
			tx.commit();
		} catch (Exception e) {
			e.printStackTrace(); tx.rollback();
		} finally {
			if (hs != null) hs.close();
		}
		return isInserted;
	}
}
```

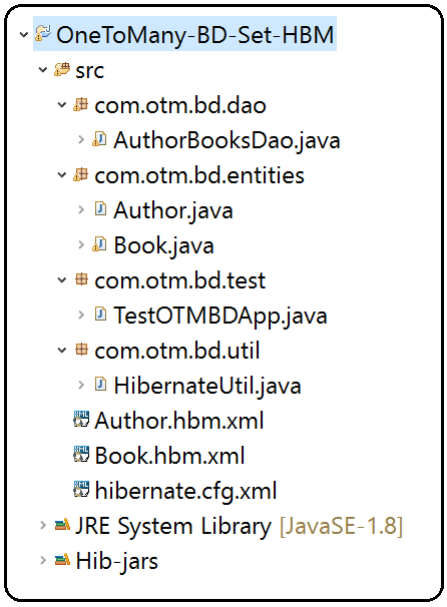
```
Book b1 = new Book();
b1.setBookName("JSE");
b1.setIsbn("ISBN001");
b1.setPrice(100.00);

Book b2 = new Book();
b2.setBookName("JEE");
b2.setIsbn("ISBN002");
b2.setPrice(200.00);


Map<String, Book> booksMap = new HashMap<String, Book>();
booksMap.put("Child-1", b1);
booksMap.put("Child-2", b2);
```

AuthorBooksDao.iava

```java
public class TestApp {

	public static void main(String[] args) {

		AuthorBooksDao dao = new AuthorBooksDao();
		dao.saveAuthorWithBooks();

		// closing session factory

		HibernateUtil.closeSf();
	}
}
```

TestApp.java

One to Many – Bi-Directional – Set – HBM - Application




```java
public class Author {

	private Integer authorId;
	private String authorName;
	private String authorEmail;

	private Set<Book> books;

	//setters & getters
}
```

Author.java

```java
public class Book {

	private Integer bookId;
	private String bookName;
	private Double bookPrice;

	private Author author;

	//setters & getters

}
```

Book.java

```xml
<hibernate-mapping>
	<class name="com.otm.bd.entities.Author" table="AUTHOR">
		<id name="authorId" type="java.lang.Integer">
			<column name="AUTHOR_ID" />
			<generator class="increment" />
		</id>
		<property name="authorName" type="java.lang.String">
			<column name="AUTHORNAME" />
		</property>
		<property name="authorEmail" type="java.lang.String">
			<column name="AUTHOREMAIL" />
		</property>
		<set name="books" table="BOOK" cascade="all-delete-orphan">
			<key column="AUTHOR_ID" />
			<one-to-many class="com.otm.bd.entities.Book" />
		</set>
	</class>
</hibernate-mapping>
```

Author.hbm.xml

```
<hibernate-mapping>
	<class name="com.otm.bd.entities.Book" table="BOOK">
		<id name="bookId" type="java.lang.Integer">
			<column name="BOOKID" />
			<generator class="increment" />
		</id>
		<property name="bookName" type="java.lang.String">
			<column name="BOOKNAME" />
		</property>
		<property name="bookPrice" type="java.lang.Double">
			<column name="BOOKPRICE" />
		</property>
		<many-to-one name="author"
			class="com.otm.bd.entities.Author"
			fetch="select" >
			<column name="AUTHOR_ID" />
		</many-to-one>
	</class>
</hibernate-mapping>
```

Book.hbm.xml

```
public class AuthorBooksDao {

	public boolean saveAuthorWithBooks() {
		boolean isInserted = false;
		Session hs = null;
		Transaction tx = null;
		try {
			hs = HibernateUtil.getSession();
			tx = hs.beginTransaction();

			Author a = new Author();
			a.setAuthorName("Gaven King");
			a.setAuthorEmail("gk@jboss.org");
			a.setBooks(booksSet);

			Serializable ser = hs.save(a);

			if (ser != null) {
				isInserted = true;
			}
			tx.commit();
		} catch (Exception e) {
			e.printStackTrace();
			tx.rollback();
		} finally {
			if (hs != null)
				hs.close();
		}
		return isInserted;
	}
```

```
Book b1 = new Book();
b1.setBookName("Hibernate");
b1.setBookPrice(100.00);

Book b2 = new Book();
b2.setBookName("JPA");
b2.setBookPrice(200.00);

Set<Book> booksSet = new HashSet<Book>();
booksSet.add(b1);
booksSet.add(b2);
```

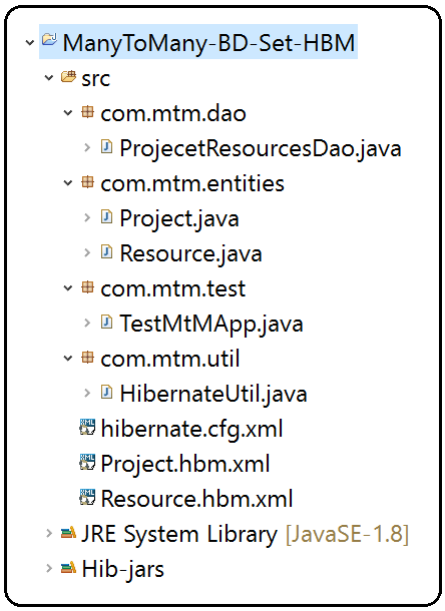
```
public void findBookById(int bid) {
	Session hs = null;
	Transaction tx = null;
	try {
		hs = HibernateUtil.getSession();
		Book b = hs.get(Book.class, bid);
		Author a = b.getAuthor();
		System.out.println(a.getAuthorEmail());
	} catch (Exception e) {
		e.printStackTrace();
		tx.rollback();
	} finally {
		if (hs != null)
			hs.close();
	}
}
```

AuthorBooksDao.java

```
public class TestOTMBDApp {

	public static void main(String[] args) {
		AuthorBooksDao dao = new AuthorBooksDao();
		// dao.saveAuthorWithBooks();
		dao.findBookById(1);
	}
}
```

TestOTMBDApp.java

Many to Many – Bi –Directional – Set – HBM- Application




```java
public class Project {

	private Integer projId;
	private String projName;
	private Date startDt;
	private Date endDt;

	private Set<Resource> resources;

	//setters && getters

}
```

Project.java

```java
public class Resource {

	private Integer resourceId;
	private String resourceName;
	private String resourceEmail;

	private Set<Project> projects;

	//setters & getters

}
```

Resource.java

```xml
<hibernate-mapping>
	<class name="com.mtm.entities.Project" table="PROJECT_DETAILS">
		<id name="projId" type="java.lang.Integer">
			<column name="PROJECT_ID" />
			<generator class="increment" />
		</id>
		<property name="projName" type="java.lang.String">
			<column name="PROJ_NAME" />
		</property>
		<property name="startDt" type="java.util.Date">
			<column name="START_DT" />
		</property>
		<property name="endDt" type="java.util.Date">
			<column name="END_DT" />
		</property>
		<set name="resources" table="PROJECT_RESOURCES" cascade="all">
			<key>
				<column name="PROJECT_ID" />
			</key>
			<many-to-many class="com.mtm.entities.Resource" column="RESOURCE_ID" />
		</set>
	</class>
</hibernate-mapping>
```

Project.hbm.xml

```xml
<hibernate-mapping>
	<class name="com.mtm.entities.Resource" table="RESOURCE_DETAILS">
		<id name="resourceId" type="java.lang.Integer">
			<column name="RESOURCE_ID" />
			<generator class="increment" />
		</id>
		<property name="resourceName" type="java.lang.String">
			<column name="RESOURCE_NAME" />
		</property>
		<property name="resourceEmail" type="java.lang.String">
			<column name="RESOURCEEMAIL" />
		</property>
		<set name="projects" table="PROJECT_RESOURCES" cascade="all">
			<key>
				<column name="RESOURCE_ID" />
			</key>
			<many-to-many class="com.mtm.entities.Project" column="PROJECT_ID" />
		</set>
	</class>
</hibernate-mapping>
```

Resource.hbm.xml

```java
public boolean saveResourceWithProjects() {
	boolean isInserted = false;
	Session hs = null; Transaction tx = null;
	SimpleDateFormat sdf = new SimpleDateFormat("dd/MMM/yyyy");
	try {
		hs = HibernateUtil.getSession(); tx = hs.beginTransaction();

		Resource r1 = new Resource();
		r1.setResourceName("Smith"); r1.setResourceEmail("smit@in.com");

		Resource r2 = new Resource();
		r2.setResourceName("Dean"); r2.setResourceEmail("de@in.com");

		Project p1 = new Project();
		p1.setProjName("IRCTC"); p1.setStartDt(new Date());
		p1.setEndDt(sdf.parse("27/Jan/2019"));

		Project p2 = new Project();
		p2.setProjName("Gmail"); p2.setStartDt(new Date());
		p2.setEndDt(sdf.parse("27/Jun/2019"));

		Set<Resource> resSet = new HashSet<Resource>();
		resSet.add(r1); resSet.add(r2);

		Set<Project> projSet = new HashSet<Project>();
		projSet.add(p1); projSet.add(p2);

		// Associate parent with child
		r1.setProjects(projSet); r2.setProjects(projSet);

		// Saving
		hs.save(r1);  hs.save(r2);

		tx.commit();
		hs.close();
		isInserted = true;
	} catch (Exception e) {
		e.printStackTrace(); tx.rollback();
	}
	return isInserted;
}
```

saveResourceWithProjects - method

```java
public void findResourceByProjId(int pid) {
	Session hs = null;
	try {
		hs = HibernateUtil.getSession();
		Project p = hs.get(Project.class, pid);
		Set<Resource> resSet = p.getResources();
		if (!resSet.isEmpty()) {
			for (Resource r : resSet) {
				System.out.println(r.getResourceName());
			}
		}
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

findResourceByProjectId - Method

```
public void findProjectByResId(int rid) {
		Session hs = null;
		try {
			hs = HibernateUtil.getSession();
			Resource r = hs.get(Resource.class, rid);
			Set<Project> projects = r.getProjects();
			if (!projects.isEmpty()) {
				for (Project p : projects) {
					System.out.println(p.getProjName());
				}
			}
		} catch (Exception e) {
			e.printStackTrace();
		}
}
```

findProjectByResourceId - Method

```
public void deleteResourceFromProjet(int rid,int pid) {
		Session hs = null;
		Transaction tx = null;
		try {
			hs = HibernateUtil.getSession();
			tx = hs.beginTransaction();
			Resource r = hs.get(Resource.class, rid);
			Set<Project> projects = r.getProjects();
			Project p = hs.get(Project.class, pid);
			projects.remove(p);
			tx.commit();
			hs.close();

		} catch (Exception e) {
			e.printStackTrace();
		}
}
```

deleteResourceFromProject - Method

```
public class TestMtMApp {

	public static void main(String[] args) {
		ProjecetResourcesDao dao = new ProjecetResourcesDao();
		// dao.saveResourceWithProjects();
		// dao.findResourceByProjId(2);
		// dao.findProjectByResId(2);
		dao.deleteResourceFromProjet(1, 2);
	}
}
```

TestMtMApp.java

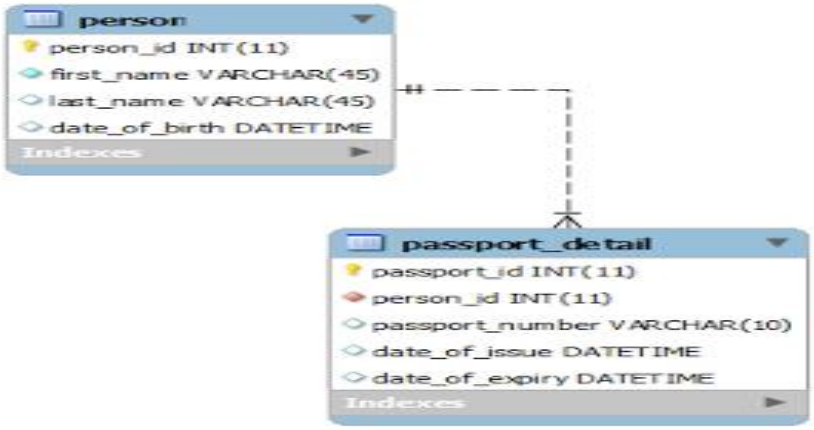
Relationships Using Annotations

One to One Relationship

One-to-One association means, each row of table is mapped with exactly one and only one row with another table

For example, we have a passport_detail and person table each row of person table is mapped with exactly one and only one row of passport_detail table. One person has only one passport.

One-One relationship ER Diagram



==========Person.java==========

```
@Entity
@Table(name = "person")
public class Person {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "person_id", unique = true, nullable = false)
	private Integer personId;

	@Column(name = "first_name")
	private String firstName;

	@Column(name = "last_name")
	private String lastName;

	@Temporal(TemporalType.DATE)
	@Column(name = "date_of_birth")
	private Date dateOfBirth;

	@OneToOne(mappedBy = "person", cascade = CascadeType.ALL)
	private PassportDetail passportDetail;

	//getters and setters

 }
```

==========PassportDetails.java==========

```
@Entity
@Table(name = "passport_details")
public class PassportDetails {

	@Id
	@GeneratedValue
	@Column(name = "PASSPORT_ID")
	private Integer passportId;

	@Column(name = "PASSPORT_NO")
	private String passportNo;

	@OneToOne
	@JoinColumn(name = "PERSON_ID")
	private Person person;

	//getters and setters

 }
```

```
public class OneToOneDemo {

	public static void main(String[] args) {
```

```
		OneToOneDemo otd = new OneToOneDemo();
		otd.insertRecord();
		//otd.retriveRecord();
	}
	public void retriveRecord() {
		try {
			SessionFactory sf = HibernateUtil.getSessionFactory();
			Session hsession = sf.openSession();
			Transaction tx = hsession.beginTransaction();
			Person p = hsession.get(Person.class, new Integer(25));
			tx.commit();
			hsession.close();
			sf.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public void insertRecord() {
		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();
		Person person = new Person();
		person.setFirstName("Tony");
		person.setLastName("Leo");
		SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
		PassportDetails passportDetail = new PassportDetails();
		try {
			person.setDateOfBirth(sdf.parse("1990-10-10"));
			passportDetail.setDateOfIssue(sdf.parse("2010-10-10"));
			passportDetail.setDateOfExpiry(sdf.parse("2020-10-09"));
		} catch (Exception e) {
			e.printStackTrace();
		}

		passportDetail.setPassportNo("Q12345678");

		person.setPassportDetails(passportDetail);
		passportDetail.setPerson(person);

		// saving person it will generate two insert query.
		System.out.println(hsession.save(person));
		tx.commit();
		hsession.close();
		sf.close();
	}
}
```

That's it, now run the PersonPassportService class you will get output at your console where two insert query will executed one for Person and one for PasspportDetail table.

One-To-Many Relationship

One to many association means each row of a table can be related to many rows in the relating tables.

For example, we have an author and book table, we all know that a particular book is written by a particular author. An author of a book is always one and only one person, co-author of a book may be multiple but author always only one. So many books can be written by an author, here many books an author relationship is one-to-many association example.

an author has many books that is. 1----------* books(one-to-may association)
Here is the example of author and book table for our One-To-Many association example

Author table:

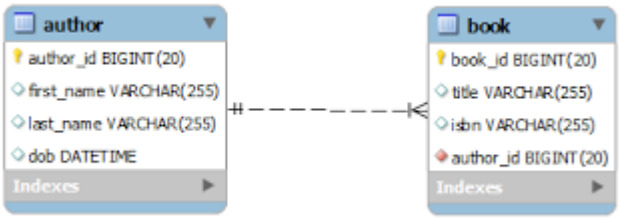
author_id	first_name	last_name	dob
101	Joshua	Bloch	1961-08-28
102	Watts	S. Humphrey	1927-07-04

Book table:

book_Id	title	isbn	author_id
1001	A Discipline for Software Engineering	0201546108	102
1002	Managing the Software Process	8177583301	102
1003	Effective Java	9780321356680	101
1004	Java Puzzlers	032133678X	101

From the above two tables we have seen that author id 101 has written two books(1003,1004) and author 102 is also written two books(1001,1002) but the same book is not written by each other. So it is clear that an author has many books. Lets start implementation of these in hibernate practically.

Here is an One-To-Many association mapping of an author and book table ER diagram:



==========Author.java==========

```
@Entity
@Table(name = "author")
public class Author {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "author_id", unique = true, nullable = false)
	private long authorId;

	@Column(name = "first_name")
	private String firstName;

	@Column(name = "last_name")
	private String lastName;

	@Temporal(TemporalType.DATE)
	@Column(name = "dob")
	private Date dateOfBirth;

	@OneToMany(mappedBy = "author", cascade = CascadeType.ALL)
	private Set<Book> books = new HashSet<>();

	//getters and settes

}
```

```
@Entity
@Table(name = "book")
public class Book {
```

```
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "book_id", unique = true, nullable = false)
	private long bookId;

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

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

	@ManyToOne
	@JoinColumn(name = "author_id", nullable = false)
	private Author author;

	//setters and getters

}
```

```
public class OneToMany {
	public static void main(String[] args) {
		 insertRecord();
		//retriveBooks();
		//deleteBook();
	}
	public static void insertRecord() {
		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();

		Author author = new Author();
		author.setAuthorName("Ramu");
		author.setAuthorDob(new Date());

		Book book1 = new Book();
		book1.setBookName("Design Patterns");
		book1.setIsbn("DP1234");
		book1.setAuthor(author);

		Book book2 = new Book();
		book2.setBookName("Spring In Action");
		book2.setIsbn("SP0987");
		book2.setAuthor(author);

		Set<Book> books = new HashSet<Book>();
		books.add(book1);
		books.add(book2);

		author.setBooks(books);
		hsession.save(author);

		tx.commit();
		hsession.close();
		sf.close();
	}
	public static void retriveBooks() {
		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Author author = hsession.get(Author.class, new Integer(33));
		System.out.println(author);
		hsession.close();
		sf.close();
	}
	public static void deleteBook() {
```

```
            SessionFactory sf = HibernateUtil.getSessionFactory();
            Session hsession = sf.openSession();
            Transaction tx = hsession.beginTransaction();

            Author auth = new Author();
            auth.setAuthorId(33);
            Query query = hsession.createQuery("from Author where authorId=39");
            hsession.delete(query.getResultList().get(0));

            tx.commit();
            hsession.close();
            sf.close();
      }

}
```

Many to Many Relationship

Many-To-Many association means, one or more row(s) of a table are associated one or more row(s) in another table. For example an employee may assigned with multiple projects, and a project is associated with multiple employees

Employee table:

employee_id	first_name	last_name	doj
101	Tony	Jha	2014-08-28
102	Zeneva	S. Humphrey	2014-07-04

Project table:

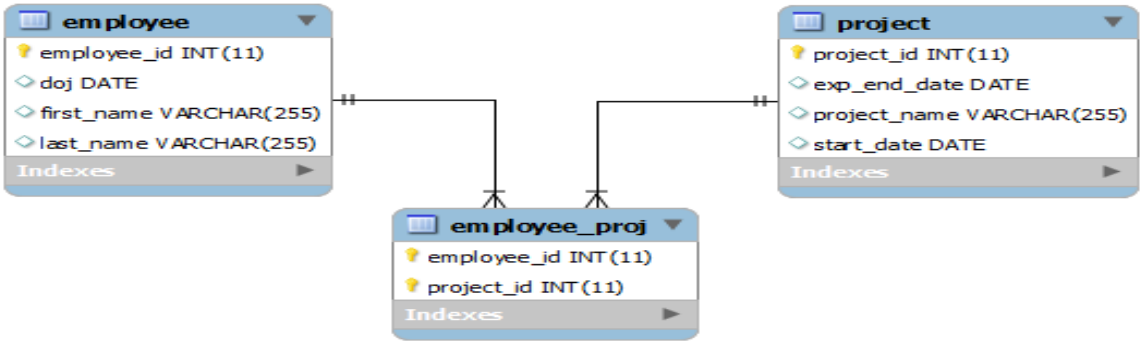
project_id	project_name	start_date	exp_end_date
101	MARS	2013-01-01	2015-08-28
102	SBA	2014-10-02	2016-07-04

Employee_Proj table:

employee_id	project_id
101	101
101	102
102	102

From the above Employee_Proj table it's clear that an employee 101 is associated with project(101,102) and project 102 is associated with employee(101,102). So it's clear that employee and project relationship is fall in Many-To-Many association categories.

Many-To-Many association mapping table ER diagram.



----------Employee.java----------

```
@Entity
@Table(name = "employee")
public class Employee {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "employee_id")
	private int employeeId;

	@Column(name = "first_name")
	private String firstName;

	@Column(name = "last_name")
	private String lastName;

	@Column(name = "doj")
	@Temporal(TemporalType.DATE)
	private Date doj;

	@ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
	@JoinTable(name = "employee_proj", joinColumns = { @JoinColumn(name = "em-
ployee_id", nullable = false, updatable = false) }, inverseJoinColumns = { @JoinCol-
umn(name = "project_id", nullable = false, updatable = false) })
	private Set<Project> projects;

	//setters and getters
}
```

```
@Entity
@Table(name = "project")
public class Project {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "project_id")
	private int projectId;

	@Column(name = "project_name")
	private String projectName;

	@Column(name = "start_date")
	@Temporal(TemporalType.DATE)
	private Date startDate;

	@Column(name = "exp_end_date")
	@Temporal(TemporalType.DATE)
	private Date expectedEndDate;

	@ManyToMany(fetch = FetchType.LAZY, mappedBy = "projects")
	private Set<Employee> employees;

		//setters and getters
}
```

```
public class ManyToManyDemo {
	public static void main(String[] args) {
		Employee employee = new Employee();

		SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
		try {

			SessionFactory sf = HibernateUtil.getSessionFactory();
			Session hsession  = sf.openSession();
			Transaction tx = hsession.beginTransaction();

			// employee details
			employee.setDoj(sdf.parse("2014-08-28"));
			employee.setFirstName("Tony");
			employee.setLastName("Jha");

			// project details
			Project mars = new Project();
			mars.setProjectName("MARS");
			mars.setStartDate(sdf.parse("2013-01-01"));
			mars.setExpectedEndDate(sdf.parse("2015-08-28"));

			// project 2 details
			Project sba = new Project();
			sba.setProjectName("SBA");
			sba.setStartDate(sdf.parse("2014-10-02"));
			sba.setExpectedEndDate(sdf.parse("2016-07-04"));

			Set<Project> projects = new HashSet<>();
			projects.add(mars);
			projects.add(sba);

			employee.setProjects(projects);

			hsession.save(employee);

			tx.commit();
			hsession.close();
			sf.close();
		} catch (ParseException e) {
			e.printStackTrace();
		}
	}
}
```

Fetching Strategies in Hibernate

The fetch type essentially decides whether to load all of the relationships of a particular object/table as soon as the object/table is initially fetched.

There are two types of the fetching strategies in the hibernate.

1. Lazy Fetch type – Load the child entities when needed
2. Eager Fetch type – Load all child entities immediately when parent object is loaded

Lazy Fetch Type

Hibernate defaults to a lazy fetching strategy for all entities and collections. Suppose you have a parent and that parent has a collection of children. Now hibernate can lazy load these children which means that hibernate does not load all the children while loading the parent. Instead it loads children only when it is requested to do so. It prevents a huge load since entity is loaded only once the program requires them. Hence it increase the performance.

Syntax :

```
@OneToMany(mappedBy = "author", fetch = FetchType.LAZY)
private Set<Book> books = new HashSet<>();
```

Eager Fetch Type

In hibernate 2 this is default behavior of the retrieving an object from the database.

Join Fetch strategy the eager fetching of associations. The purpose of Join Fetch strategy is optimization in terms of time. I mean even associations are fetched right at the time of fetching parent object. So in this case we don't make database call again and again . So this will be much faster. Agreed that this will bad if we are fetching too many objects in a session because we can get java heap error.

Syntax :

```
@OneToMany(mappedBy = "author", fetch = FetchType.EAGER)
private Set<Book> books = new HashSet<>();
```

Cascade Types in Hibernate

Cascading means that if we do any operation on an object it acts on the related objects as well. If you insert, update or delete an object the related objects (mapped objects) are inserted, updated or deleted.

Hibernate does not navigate to object associations when you are performing an operation on the object. We have to do the operations like save, update or delete on each object explicitly. To overcome this problem Hibernate is providing Cascade Types.

To enable cascading for related association, the cascade attribute has to be configured for association in mapping metadata.

- **cascade="none",** the default, tells Hibernate to ignore the association.
- **cascade="save-update"** tells Hibernate to navigate the association when the transaction is committed and when an object is passed to save () or update () and save newly instantiated transient instances and persist changes to detached instances.
- **cascade="delete"** tells Hibernate to navigate the association and delete persistent instances when an object is passed to delete ().
- **cascade="all"** means to cascade both save-update and delete, as well as calls to evict and lock.
- **cascade="all-delete-orphan"** means the same as cascade="all" but, in addition, Hibernate deletes any persistent entity instance that has been removed (dereferenced) from the association (for example, from a collection).
- **cascade="delete-orphan"** Hibernate will delete any persistent entity instance that has been removed (dereferenced) from the association (for example, from a collection).

Using Annotations we can use below options

- **CascadeType.PERSIST:** means that save () or persist () operations cascade to related entities.
- **CascadeType.MERGE**: means that related entities are merged into managed state when the owning entity is merged.
- **CascadeType.REFRESH**: does the same thing for the refresh () operation.
- **CascadeType.REMOVE:** removes all related entities association with this setting when the owning entity is deleted.
- **CascadeType.DETACH**: detaches all related entities if a "manual detach" occurs.
- **CascadeType.ALL**: is shorthand for all of the above cascade operations.

The cascade configuration option accepts an array of CascadeTypes; thus, to include only refreshes and merges in the cascade operation for a One-to-Many relationship as in our example, you might see the following:

```
@OneToMany (cascade={CascadeType.REFRESH, CascadeType.MERGE}, fetch =
FetchType.LAZY)
@JoinColumn (name="AUTHOR_ID")

private Set<Book> books;
```

Hibernate Validator framework

A good data validation strategy is an important part of every application development project. Being able to consolidate and generalize validation using a proven framework can significantly improve the reliability of your software, especially over time

Hibernate Validator provides a solid foundation for building lightweight, flexible validation code for Java SE and Java EE applications. Hibernate Validator is supported by a number of popular frameworks, but its libraries can also be used in a standalone implementation. Standalone Java SE validation components can become an integral part of any complex heterogeneous server-side application. In order to follow this introduction to using Hibernate Validator to build a standalone component, you will need to have JDK 6 or higher installed. All use cases in the article are built using Validator version 5.0.3. You should download the Hibernate Validator 5.0.x binary distribution package, where directory \hibernate-validator-5.0.x.Final\dist contains all the binaries required for standalone implementation.

- **@Size** : This annotation is used to set the size of the field. It has three properties to configure, the `min`, `max` and the `message` to be set.
- **@Min** : This annotation is used to set the min size of a field
- **@NotNull** : With this annotation you can make sure that the field has a value.
- **@Length** : This annotation is similar to **@Size**.
- **@Pattern** : This annotation can be used when we want to chack a field against a regular expression. The `regex` is set as an attribute to the annotation.
- **@Range** : This annotation can be used to set a range of min and max values to a field.

```
----------------------------------Form.java-----------------------------------------------
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
import org.hibernate.validator.constraints.Length;
import org.hibernate.validator.constraints.Range;

public class Form {

	@Size(min = 5, max = 10, message = "Your name should be between 5 - 10 characters.")
	private String name;
	@Min(value = 5, message = "Please insert at least 5 characters")
	private String lastname;
	@NotNull(message = "Please select a password")
	@Length(min = 5, max = 10, message = "Password should be between 5 - 10 charactes")
	private String password;
	@Pattern(regexp = "[0-9]+", message = "Wrong zip!")
	private String zip;
	@Pattern(regexp = ".+@.+\\..+", message = "Wrong email!")
	private String email;
	@Range(min = 18, message = "You cannot subscribe if you are under 18 years old.")
	private String age;

	// Setters and getters
}
```

Connection Pooling in Hibernate

By default, Hibernate uses JDBC connections in order to interact with a database. Creating these connections is expensive—probably the most expensive single operation Hibernate will execute in a typical-use case. Since JDBC connection management is so expensive that possibly you will advise to use a pool of connections, which can open connections ahead of time (and close them only when needed, as opposed to “when they’re no longer used”).

Thankfully, Hibernate is designed to use a connection pool by default, an internal implementation. However, Hibernate’s built-in connection pooling is not designed for production use. In production, you would use an external connection pool by using either a database connection provided by JNDI or an external connection pool configured via parameters and classpath.

Hibernate supports a variety of connection pooling mechanisms. If you are using an application server, you may wish to use the built-in pool (typically a connection is obtaining using JNDI). If you can't or don't wish to use your application server's built-in connection pool,

Hibernate Supports for below External Connection pools

c3p0 - Distributed with Hibernate

Apache DBCP - Apache Pool

Proxocol - Distributed with Hibernate

Maven dependencies for configuring connection pool

```
<dependencies>
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-core</artifactId>
			<version>5.2.6.Final</version>
		</dependency>
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-c3p0</artifactId>
			<version>5.2.6.Final</version>
		</dependency>
		<dependency>
			<groupId>com.oracle</groupId>
			<artifactId>ojdbc14</artifactId>
			<version>10.1</version>
		</dependency>
		<dependency>
			<groupId>c3p0</groupId>
			<artifactId>c3p0</artifactId>
			<version>0.9.1.2</version>
		</dependency>

		<!--For DBCP-->
			<dependency>
			<groupId>commons-dbcp</groupId>
			<artifactId>commons-dbcp</artifactId>
			<version>1.4</version>
		</dependency>
		<!-- ENd -->
</dependencies>
```

Configure C3P0 Connection Pool

```
<hibernate-configuration>
<session-factory>
  <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
  <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/myschema</property>
  <property name="hibernate.connection.username">user</property>
  <property name="hibernate.connection.password">password</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
  <property name="show_sql">true</property>

  <property name="hibernate.c3p0.min_size">5</property>
  <property name="hibernate.c3p0.max_size">20</property>
  <property name="hibernate.c3p0.timeout">300</property>
  <property name="hibernate.c3p0.max_statements">50</property>
  <property name="hibernate.c3p0.idle_test_period">3000</property>

  . . . .
</session-factory>
</hibernate-configuration>
```

hibernate.c3p0.min_size – Minimum number of JDBC connections in the pool. Hibernate default: 1
hibernate.c3p0.max_size – Maximum number of JDBC connections in the pool. Hibernate default: 100
hibernate.c3p0.timeout – When an idle connection is removed from the pool (in second). Hibernate default: 0, never expire.
hibernate.c3p0.max_statements – Number of prepared statements will be cached. Increase performance. Hibernate default: 0, caching is disable.
hibernate.c3p0.idle_test_period – idle time in seconds before a connection is automatically validated. Hibernate default: 0

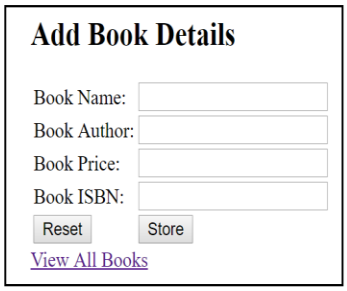
Configure Apache DBCP Connection Pool
Apache Connection Pool can be downloaded from http://commons.apache.org/dbcp/
In order to integrate this pool with Hibernate you will need the following jars: **commons-dbcp.jar** and **commons-pool-1.5.4.jar**.

```
<hibernate-configuration>
 <session-factory>
  <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
  <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/myschema</property>
  <property name="hibernate.connection.username">user</property>
  <property name="hibernate.connection.password">password</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
  <property name="show_sql">true</property>

  <property name="hibernate.dbcp.initialSize">8</property>
  <property name="hibernate.dbcp.maxActive">20</property>
  <property name="hibernate.dbcp.maxIdle">20</property>
  <property name="hibernate.dbcp.minIdle">0</property>

  . . . .
</session-factory>
</hibernate-configuration>
```

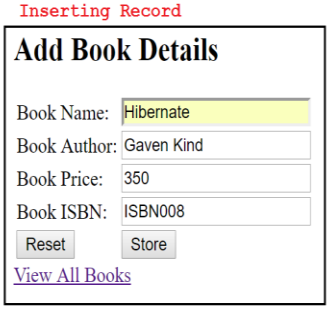
Hibernate Integration with Web Application

We are developing simple web application to perform CURD operations with below screens

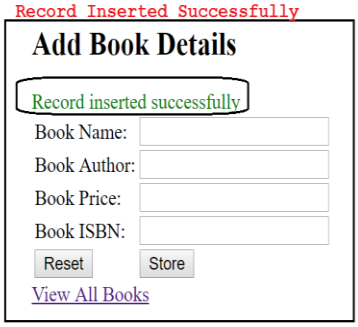
Home Screen



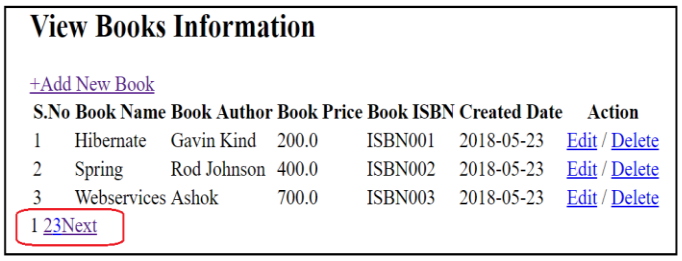
Inserting Record



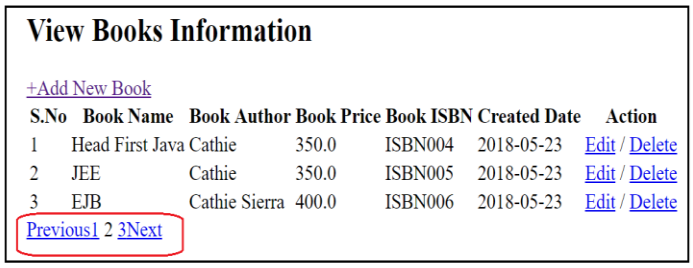
Record Inserted Successfully



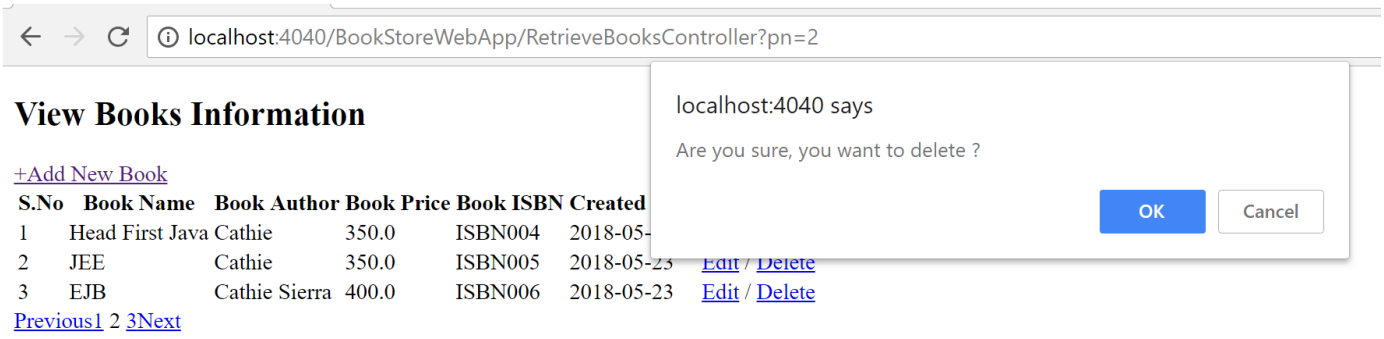
Viewing all books



Accessing books through pagination links



Deleting Record with User confirmation



Web App Project Code

Step-1: Create Dynamic web project and add required jar files (Hibernate, Servlet, Jsp, JSTL and JDBC Driver jar file) to project build path

Below is the Controller class and corresponding index.jsp to insert record

```java
@WebServlet("/BookStoreController")
public class BookStoreController extends HttpServlet {
	private static final long serialVersionUID = 1L;

	protected void doPost(HttpServletRequest request,
			HttpServletResponse response) throws ServletException, IOException {

		// capture form data and storing in model object
		BookModel model = new BookModel();
		model.setBookName(request.getParameter("bookName"));
		model.setAuthorName(request.getParameter("authorName"));
		model.setIsbn(request.getParameter("bookIsbn"));
		model.setPrice(Double.parseDouble(request.getParameter("bookPrice")));

		// calling service method
		BookStoreService service = new BookStoreService();
		boolean isInserted = service.save(model);

		if (isInserted) {
			// display success msg
			request.setAttribute("succMsg", "Record inserted successfully");
		} else {
			// display failure msg
			request.setAttribute("errMsg", "Failed to insert");
		}

		// Redirect to same page
		request.getRequestDispatcher("index.jsp").forward(request, response);
	}

}
```

BookStoreController.java

onsubmit

```jsp
<h2>Add Book Details</h2>
		<font color='green'>${succMsg}</font>
		<font color='red'>${errMsg}</font>
		<form action="BookStoreController" method="POST">
		  <table>
			<tr>
				<td>Book Name:</td>
				<td><input type="text" name="bookName" /></td>
			</tr>
			<tr>
				<td>Book Author:</td>
				<td><input type="text" name="authorName" /></td>
			</tr>
			<tr>
				<td>Book Price:</td>
				<td><input type="text" name="bookPrice" /></td>
			</tr>
			<tr>
				<td>Book ISBN:</td>
				<td><input type="text" name="bookIsbn" /></td>
			</tr>
			<tr>
				<td><input type="reset" value="Reset" /></td>
				<td><input type="submit" value="Store" /></td>
			</tr>
			</table>
		</form>

<a href="RetrieveBooksController">View All Books</a>
```

index.jsp

Below is the Controller class to retrieve record with Server side pagination

```java
@WebServlet("/RetrieveBooksController")
public class RetrieveBooksController extends HttpServlet {
	private static final long serialVersionUID = 1L;

	protected void doGet(HttpServletRequest request,
			HttpServletResponse response) throws ServletException, IOException {
		BookStoreService service = new BookStoreService();

		int pageNo = 1;

		String pn = request.getParameter("pn");
		if (pn != null && !"".equals(pn)) {
			pageNo = Integer.parseInt(pn);
		}
		request.setAttribute("currPageNo", pageNo);

		long totRecords = service.retriveRecordsCnt();
		int pageSize = 3;
		long totalPagesReq = (totRecords / pageSize) + (totRecords % pageSize > 0 ? 1 : 0);
		request.setAttribute("pagesReq", totalPagesReq);

		List<BookModel> modelsList = service.retrieveAllBooks(pageNo, pageSize);

		// Storing book models in request scope
		request.setAttribute("books", modelsList);

		// Dispatch the request to jsp to print the data
		RequestDispatcher rd = request.getRequestDispatcher("viewBooks.jsp");
		rd.forward(request, response);

	}
}
```

RetrieveBooksController.java

Below is the viewBooks.jsp file to display records with Edit and Delete hyperlinks

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
<script>
        function confirmDelete() {
                var status = confirm("Are you sure, you want to delete ?");
                if (status) {
                        return true;
                } else {
                        return false;
                }
        }
</script>
</head>
<body>

        body content

</body>
</html>
```

```
<h2>View Books Information</h2>
        <a href="index.jsp">+Add New Book</a>
<table>
        <thead>
                <th>S.No</th>
                <th>Book Name</th>
                <th>Book Author</th>
                <th>Book Price</th>
                <th>Book ISBN</th>
                <th>Created Date</th>
                <th>Action</th>
        </thead>
        <tbody>
                <c:forEach items="${books}" var="b" varStatus="cnt">
                        <tr>
                                <td><c:out value="${cnt.index+1}" /></td>
                                <td><c:out value="${b.bookName}" /></td>
                                <td><c:out value="${b.authorName}" /></td>
                                <td><c:out value="${b.price}" /></td>
                                <td><c:out value="${b.isbn}" /></td>
                                <td><c:out value="${b.createDt}" /></td>
                                <td><a href="EditBookController?bookId=${b.bookId}">Edit</a> /
                                <a href="DeleteBookController?bookId=${b.bookId}"
                                onclick="return confirmDelete()">Delete</a>
                        </tr>
                </c:forEach>
        </tbody>
</table>
```

viewBooks.jsp

Below is the controller class to delete book record using book id

```
@WebServlet("/DeleteBookController")
public class DeleteBookController extends HttpServlet {
        private static final long serialVersionUID = 1L;

        protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException, IOException {

                String bookId = request.getParameter("bookId");
                if (bookId != null && !"".equals(bookId)) {
                        int bid = Integer.parseInt(bookId);
                        BookStoreService service = new BookStoreService();
                        boolean isDeleted = service.deleteBookById(bid);

                        request.getRequestDispatcher("RetrieveBooksController").forward(
                                        request, response);
                }
        }
}
```

DeleteBookController.java

Below are the Entity and Model classes

```java
@Entity
@Table(name = "BOOK_DETAILS")
public class BookEntity {

	@Id
	@GeneratedValue
	@Column(name = "BOOK_ID")
	private Integer bookId;

	@Column(name = "BOOK_NAME")
	private String bookName;

	@Column(name = "AUTHOR_NAME")
	private String authorName;

	@Column(name = "BOOK_ISBN")
	private String isbn;

	@Column(name = "BOOK_PRICE")
	private Double bookPrice;

	@Column(name = "IS_ACTIVE")
	private String isActive;

	@Column(name = "CREATE_DT")
	@Temporal(TemporalType.DATE)
	@CreationTimestamp
	private Date createDt;

	@Column(name = "UPDATE_DT")
	@Temporal(TemporalType.DATE)
	@UpdateTimestamp
	private Date updateDt;

	//Setters & getters
}
```

BookEntity.java

```java
public class BookModel {

	private Integer bookId;
	private String bookName;
	private String authorName;
	private String isActive;
	private String isbn;
	private Double price;
	private Date createDt;

	//setters & getters

}
```

BookModel.java

Entity class mapped with DB table

Model class is used for data transferring to and fro

Below is the Service class (It calls Dao class method for DB operations)

```java
package com.web.bs.service;

import java.util.ArrayList;
import java.util.List;

import com.web.bs.dao.BookStoreDao;
import com.web.bs.entities.BookEntity;
import com.web.bs.model.BookModel;

/**
 * This class is written to handle all business operations for Book_store
 * functionality
 *
 * @author Ashok
 *
 */
public class BookStoreService {

	BookStoreDao dao = new BookStoreDao();

	/**
	 * This method is used to store book data by calling dao method
	 *
	 * @param model
	 * @return boolean
	 */
	public boolean save(BookModel model) {
		BookEntity entity = new BookEntity();

		// Converting model to Entity
		entity.setBookId(model.getBookId());
		entity.setBookName(model.getBookName());
		entity.setAuthorName(model.getAuthorName());
		entity.setIsbn(model.getIsbn());
		entity.setBookPrice(model.getPrice());
		entity.setIsActive("Y");

		// Calling dao method and returning status
		return dao.insertBook(entity);
	}

	/**
	 * This method will retrive all books from dao and will return data in model
	 * format
	 *
	 * @return
	 */
	public List<BookModel> retrieveAllBooks(int pageNo, int pageSize) {

		List<BookModel> models = new ArrayList<BookModel>();

		// retrieving data from dao method
		List<BookEntity> entitiesList = dao.findAll(pageNo, pageSize);

		// Convert each entity to model
		for (BookEntity entity : entitiesList) {

			BookModel model = new BookModel();
			model.setBookId(entity.getBookId());
			model.setBookName(entity.getBookName());
			model.setAuthorName(entity.getAuthorName());
			model.setIsbn(entity.getIsbn());
			model.setIsActive(entity.getIsActive());
			model.setPrice(entity.getBookPrice());
			model.setCreateDt(entity.getCreateDt());
			// Adding each model to collection
			models.add(model);
		}
		return models;
	}

	public long retriveRecordsCnt() {
		return dao.findTotalRecordsCnt();
	}

	public boolean deleteBookById(int bid) {
		return dao.deleteBook(bid);
	}
}
```

BookStoreService.java

Below is the Dao class to perform CURD operations in Database

```
/**
 * This class is written to handle database operations for Book_store table
 *
 * @author Ashok
 *
 */
public class BookStoreDao {

	/**
	 * This method is used to store book data
	 *
	 * @param entity
	 * @return
	 */
	public boolean insertBook(BookEntity entity) {
		boolean isInserted = false;
		Session hs = null;
		Transaction tx = null;
		try {
			hs = HibernateUtil.getSession();
			tx = hs.beginTransaction();
			Serializable id = hs.save(entity);
			if (id != null) {
				isInserted = true;
			}
			tx.commit();
			hs.close();
		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			if (hs != null)
				hs.close();
		}

		return isInserted;
	}

	/**
	 * This method is used to retrieve all books details
	 *
	 * @return
	 */
	public List<BookEntity> findAll(int pageNo, int pageSize) {
		List<BookEntity> booksList = null;
		Session hs = null;
		try {
			hs = HibernateUtil.getSession();
			String hqlQuery = "From BookEntity where isActive='Y'";
			Query query = hs.createQuery(hqlQuery);

			// Pagination methods
			query.setFirstResult((pageNo - 1) * pageSize);
			query.setMaxResults(pageSize);

			booksList = query.getResultList();
		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			if (hs != null)
				hs.close();
		}
		return booksList;
	}
}
```

BookStoreDao.java

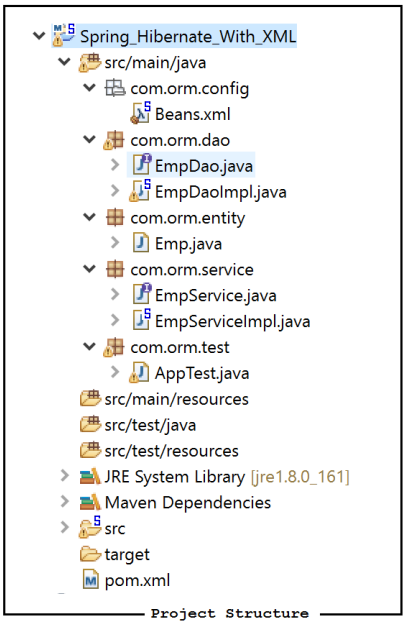
```
/**
 * This method is used to make book record as in-active
 *
 * @param bid
 * @return
 */
public boolean deleteBook(int bid) {
	boolean isDeleted = false;
	Session hs = null;
	Transaction tx = null;
	try {
		hs = HibernateUtil.getSession();
		tx = hs.beginTransaction();
		String hql = "update BookEntity set isActive='N' where bookId=:bid";
		Query query = hs.createQuery(hql);
		query.setParameter("bid", bid);
		int no = query.executeUpdate();
		if (no > 0) {
			isDeleted = true;
		}
		tx.commit();
		hs.close();
	} catch (Exception e) {
		e.printStackTrace();
	} finally {
		if (hs != null)
			hs.close();
	}
	return isDeleted;
}
```

deleteBook(int bid) from BookStoreDao

Hibernate Integration with spring (XML approach)

Step-1: Create Standalone Maven project and configure spring, Hibernate and ojdbc Dependencies like below

Note: Observe package structure



Project Structure

Spring_Hibernate_With_XML/pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.hib</groupId>
    <artifactId>Spring_Hibernate_With_XML</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.3.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>4.3.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>5.2.5.Final</version>
        </dependency>
        <dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc14</artifactId>
            <version>1.1.1</version>
        </dependency>
    </dependencies>
</project>
```

Maven Dependencies

Step-2: Create Entity class and Dao interface and its Implementation class like below

EmpDao.java

```java
package com.orm.dao;

import java.util.List;

public interface EmpDao {

    public boolean save(Emp e);

    public Emp findById(int id);

    public List<Emp> findAll();
}
```

EmpDao.java

EmpDaoImpl.java

```java
package com.orm.dao;

import java.io.Serializable;

@Repository
public class EmpDaoImpl implements EmpDao {

    @Autowired(required = true)
    private HibernateTemplate ht;

    @Transactional(readOnly = false)
    public boolean save(Emp e) {
        Serializable ser = ht.save(e);
        return (ser != null) ? true : false;
    }

    public Emp findById(int id) {
        return ht.get(Emp.class, id);
    }

    public List<Emp> findAll() {
        String hql = "From Emp";
        return (List<Emp>) ht.find(hql, null);
    }
}
```

EmpDaoImpl.java

Emp.java

```java
package com.orm.entity;

import javax.persistence.Entity;

@Entity
@Table(name = "EMP_TBL")
public class Emp {

    @Id
    @GeneratedValue
    private Integer eid;
    private String ename;
    private Double salary;

    // setters & getters

    // toString
}
```

Emp.java

Step - 3: Create Service interface and its Implementation like below

EmpService.java

```
package com.orm.service;

import java.util.List;

public interface EmpService {

    public boolean save(Emp e);

    public Emp findById(int id);

    public List<Emp> findAll();
}
```

EmpService.java

EmpServiceImpl.java

```
package com.orm.service;

import java.util.List;

@Service
public class EmpServiceImpl implements EmpService {

    @Autowired(required = true)
    private EmpDao dao;

    public boolean save(Emp e) {
        return dao.save(e);
    }

    public Emp findById(int id) {
        return dao.findById(id);
    }

    public List<Emp> findAll() {
        return dao.findAll();
    }

}
```

EmpServiceImpl.java

Step – 4: Create Spring Bean Configuration file with DataSource, SessionFactory, Transaction Manager and Hibernate Template like below

Beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd
        http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.3.xsd">
    <context:component-scan base-package="com.orm" />
    <tx:annotation-driven transaction-manager="txManager" />
    <bean id="ds"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
        <property name="url" value="jdbc:oracle:thin:@localhost:1521/XE" />
        <property name="username" value="SYSTEM" />
        <property name="password" value="admin" />
    </bean>
    <bean id="sf"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <property name="dataSource" ref="ds" />
        <property name="annotatedClasses">
            <list>
                <value>com.orm.entity.Emp</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.show_sql">true</prop>
            </props>
        </property>
    </bean>
    <bean id="txManager"
        class="org.springframework.orm.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="sf" />
    </bean>
    <bean id="ht" class="org.springframework.orm.hibernate5.HibernateTemplate">
        <property name="sessionFactory" ref="sf" />
    </bean>
</beans>
```

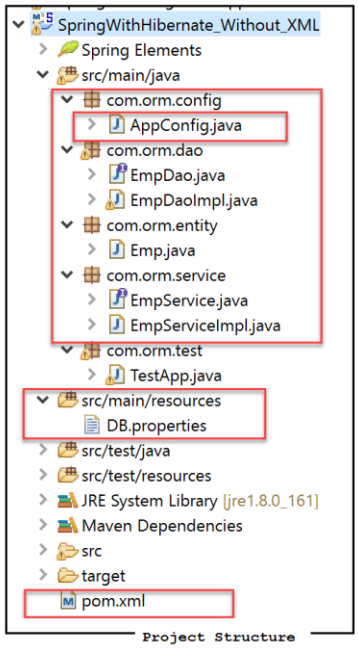
Step – 5: Create Test class to test our application

AppTest.java

```
package com.orm.test;

import java.util.List;

public class AppTest {

    public static void main(String[] args) {

        // Starting IOC
        @SuppressWarnings("resource")
        ApplicationContext context = new ClassPathXmlApplicationContext("com/orm/config/Beans.xml");

        // Getting EmpService bean obj
        EmpService service = context.getBean(EmpService.class);

        // Setting Data into Entity class obj
        Emp e = new Emp();
        e.setEname("Gita");
        e.setSalary(35000.00);

        // Saving record
        System.out.println(" Record Saved ? : " + service.save(e));

        // Getting One Record using Id
        System.out.println(service.findById(109));

        // Getting All Records
        List<Emp> emps = service.findAll();

        if (!emps.isEmpty()) {
            for (Emp emp : emps) {
                System.out.println(emp);
            }
        }
    }
}
```

Hibernate Integration with spring (Annotation Approach)

Step-1: Create Maven Standalone project and configure maven dependencies for Spring, Hibernate and ojdbc



SpringWithHibernate_Without_XML/pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.aits</groupId>
    <artifactId>SpringWithHibernate-WithoutXML</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.3.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>4.3.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>5.2.5.Final</version>
        </dependency>
        <dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc14</artifactId>
            <version>1.1.1</version>
        </dependency>
    </dependencies>
</project>
```

Maven Dependencies

Step-2: Create Application Configuration class like below

```java
@Configuration
@PropertySource("classpath:DB.properties")
@EnableTransactionManagement
@ComponentScan(basePackages = { "com.orm.dao", "com.orm.service" })
public class AppConfig {

	@Autowired
	private Environment env;

	@Bean
	public DataSource getDataSource() {
		DriverManagerDataSource dataSource = new DriverManagerDataSource();
		dataSource.setDriverClassName(env.getProperty("db.driver"));
		dataSource.setUrl(env.getProperty("db.url"));
		dataSource.setUsername(env.getProperty("db.username"));
		dataSource.setPassword(env.getProperty("db.password"));
		return dataSource;
	}
```

DataSource Creation

```java
	@Bean
	public LocalSessionFactoryBean getSessionFactory() {
		LocalSessionFactoryBean factoryBean = new LocalSessionFactoryBean();
		factoryBean.setDataSource(getDataSource());
		Properties props = new Properties();
		props.put("hibernate.show_sql", env.getProperty("hibernate.show_sql"));
		props.put("hibernate.hbm2ddl.auto", env.getProperty("hibernate.hbm2ddl.auto"));
		factoryBean.setHibernateProperties(props);
		factoryBean.setAnnotatedClasses(Emp.class);
		return factoryBean;
	}
```

SessionFactory Creation

```java
	@Bean
	public HibernateTransactionManager getTransactionManager() {

		HibernateTransactionManager transactionManager = new HibernateTransactionManager();

		transactionManager.setSessionFactory(getSessionFactory().getObject());

		return transactionManager;
	}
```

TransactionManager Creation

```java
	@Bean
	public HibernateTemplate getHibernateTemple() {

		HibernateTemplate hibernateTemplate = new HibernateTemplate(getSessionFactory().getObject());

		return hibernateTemplate;
	}
```

HibernateTemplate Creation

AppConfig.java (This class works as a replacement for xml file)

Step-3: Create Properties file in resource folder to configure database configuration details

DB.properties

```
# Database properties
db.driver=oracle.jdbc.driver.OracleDriver
db.url=jdbc:oracle:thin:@localhost:1521/XE
db.username=system
db.password=admin

# Hibernate properties
hibernate.show_sql=true
hibernate.hbm2ddl.auto=update
```

Database Properties

Step-4: Create Entity class and Dao interface and its Implementation class like below

EmpDao.java

```java
package com.orm.dao;

import java.util.List;

public interface EmpDao {

    public boolean save(Emp e);

    public Emp findById(int id);

    public List<Emp> findAll();
}
```

EmpDao.java

EmpDaoImpl.java

```java
package com.orm.dao;

import java.io.Serializable;

@Repository
public class EmpDaoImpl implements EmpDao {

    @Autowired(required = true)
    private HibernateTemplate ht;

    @Transactional(readOnly = false)
    public boolean save(Emp e) {
        Serializable ser = ht.save(e);
        return (ser != null) ? true : false;
    }

    public Emp findById(int id) {
        return ht.get(Emp.class, id);
    }

    public List<Emp> findAll() {
        String hql = "From Emp";
        return (List<Emp>) ht.find(hql, null);
    }
}
```

EmpDaoImpl.java

Emp.java

```java
package com.orm.entity;

import javax.persistence.Entity;

@Entity
@Table(name = "EMP_TBL")
public class Emp {

    @Id
    @GeneratedValue
    private Integer eid;
    private String ename;
    private Double salary;

    // setters & getters

    // toString
}
```

Emp.java

Step-5: Create Service interface and its Implementation like below

EmpService.java

```java
package com.orm.service;

import java.util.List;

public interface EmpService {

    public boolean save(Emp e);

    public Emp findById(int id);

    public List<Emp> findAll();
}
```

EmpService.java

EmpServiceImpl.java

```java
package com.orm.service;

import java.util.List;

@Service
public class EmpServiceImpl implements EmpService {

    @Autowired(required = true)
    private EmpDao dao;

    public boolean save(Emp e) {
        return dao.save(e);
    }

    public Emp findById(int id) {
        return dao.findById(id);
    }

    public List<Emp> findAll() {
        return dao.findAll();
    }

}
```

EmpServiceImpl.java

Step-6: Create Test class to test our application

AppTest.java

```java
package com.orm.test;

import java.util.List;

public class AppTest {

    public static void main(String[] args) {

        // Starting IOC
        @SuppressWarnings("resource")
        ApplicationContext context = new ClassPathXmlApplicationContext("com/orm/config/Beans.xml");

        // Getting EmpService bean obj
        EmpService service = context.getBean(EmpService.class);

        // Setting Data into Entity class obj
        Emp e = new Emp();
        e.setEname("Gita");
        e.setSalary(35000.00);

        // Saving record
        System.out.println(" Record Saved ? : " + service.save(e));

        // Getting One Record using Id
        System.out.println(service.findById(109));

        // Getting All Records
        List<Emp> emps = service.findAll();

        if (!emps.isEmpty()) {
            for (Emp emp : emps) {
                System.out.println(emp);
            }
        }
    }
}
```

====0000====