

HIBERNATE

Mr. Ashok

Ashok IT School

Facebook Group: Ashok IT School

Email ID: ashok.javatraining@gmail.com

Introduction

All applications requires some persistent mechanism to store application generated data. In java we can store data using variables and Objects. But these variables and Objects will be stored into Secondary memory. This secondary memory will be available till the application is executing. Once application execution completed this secondary memory will be vanished, then data which is stored in secondary memory also gets vanished. So we will lose the data if store in variables and Objects. To overcome this problem we need to go for Persistence stores.

What is Persistence store?

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

Almost all applications require persistent data. Persistence is one of the fundamental concepts in application development. If an information system didn't preserve data when it was powered off, the system would be of little practical use.

To store and maintain data for long time we need Persistence stores. Below are the Persistence stores we can use

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).

What is a 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's 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 isn't 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 Text File
2. CSV (Comma Separated File)
3. XML files etc.

Fixed Width Text File Definition

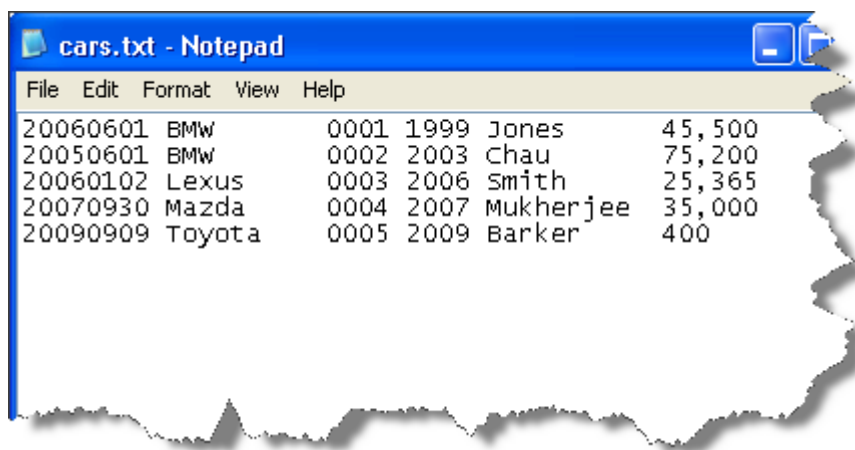
A fixed width text file is a file that has a specific format which allows for the saving of textual information/data in an organized fashion.

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's 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 exported CSV file can then be imported by the spreadsheet program.

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's 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're sure there are plenty of other examples, but none of them are likely to become popular in the immediate future.

And 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

What is a 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.

What is the 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 professional information technology, the term database is often used to refer to any collection of related data (such as a spreadsheet or a card index). This article is concerned only with databases where the size and usage requirements necessitate use of a database management system.

Existing DBMSs provide various functions that allow management of a database and its data which can be classified into four main functional groups:

Data definition – Creation, modification and removal of definitions that define the organization of the data.

Update – Insertion, modification, and deletion of the actual data.

Retrieval – Providing information in a form directly usable or for further processing by other applications. The retrieved data may be made available in a form basically 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

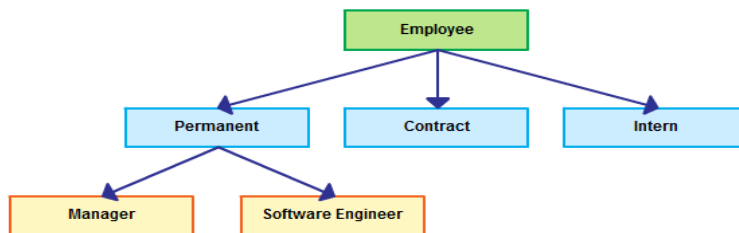
There are four structural types of database management systems:

- Hierarchical databases.
- Network databases.
- Relational databases.
- Object-oriented 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 relationship. 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.

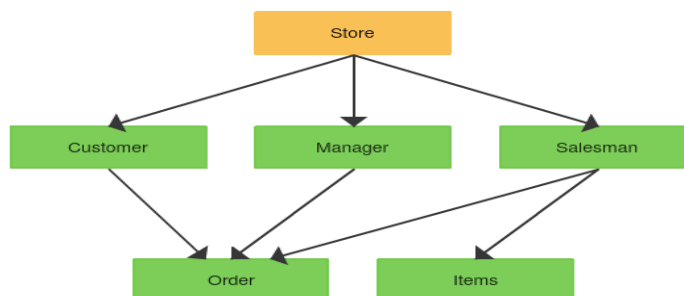
Disadvantage

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 OF NETWORK MODEL-

- ✓ 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-ADVANTAGE OF NETWORK MODEL-

- 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.

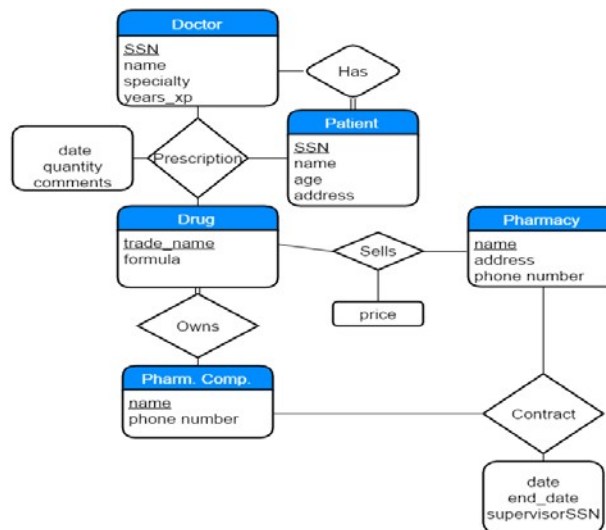
In relational databases, the relationship between data files is relational. Hierarchical and network databases require the user to pass a hierarchy in order to access needed data. These databases connect to the data in different files by using common data numbers or a key field. Data in relational databases 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 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 are 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 aren't 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 in 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 of all, 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 for a facility primary key to define unique identification of rows.
- It creates indexes for quicker data retrieval.
- Allows for various types of data integrity like (i) Entity Integrity; wherein no duplicate rows in a table exist, (ii)Domain Integrity; that enforces valid entries for a given column by filtering the type, the format, or the wide use of values, (iii)Referential Integrity; which disables the deletion of rows that are in use by other records and (iv)User Defined Integrity; providing some specific business rules that do not fall into the above three.
- Also allows for the virtual table creation which provides a safe means to store and secure sensitive content.
- Common column implementation and also 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. Also 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 also the development of hybrid object-relational databases.

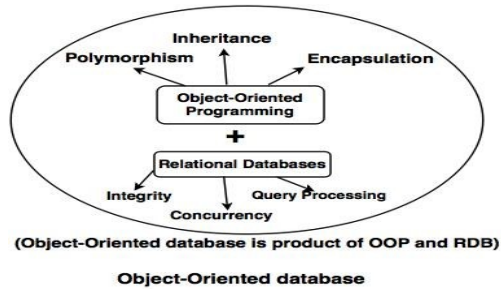
Because we work with objects in Java, it would be ideal if there were a way to store those objects in a database without having to bend and twist the object model at all. In the mid-1990s, object-oriented database systems gained attention. They're based on a network data model, which was common before 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 tightly together 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 won't bother looking too closely into why object-oriented database technology hasn't been more popular; we'll observe that object databases haven't been widely adopted and that it doesn't appear likely that they will be in the near future. We're 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 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

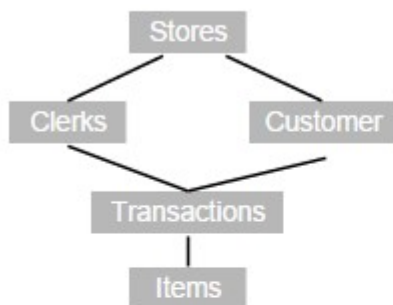
1. Object-oriented databases have these disadvantages.
2. Object-oriented database are more expensive to develop.
3. In the Most organizations are unwilling to abandon and convert from those databases.

They have already invested money in developing and implementing.

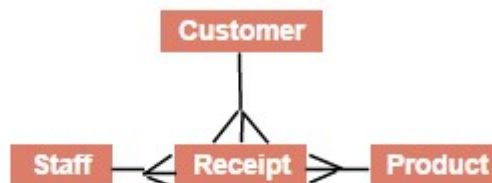
The benefits to object-oriented databases are compelling. The ability to mix and match reusable objects provides incredible multimedia capability.

Summary of 4 generations of Databases

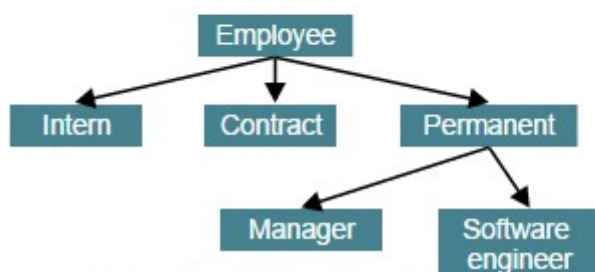
DBMS Models



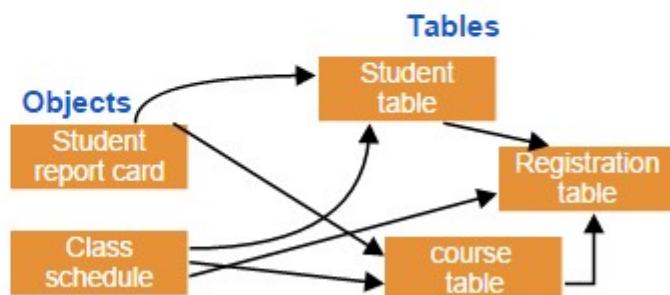
Network Database Model



Relational Database Model



Hierarchical Database Model

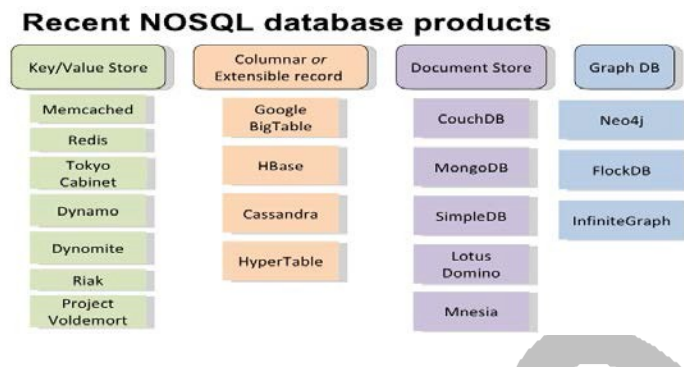


Object-oriented Database Model

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.



Database Languages

Database languages are special-purpose languages, which do one or more of the following:

Data definition language – defines data types such as creating, altering, or dropping and the relationships among them

Data manipulation language – performs tasks such as inserting, updating, or deleting data occurrences

Query language – allows searching for information and computing derived information

After storing the data into Persistence stores, we can perform some operations on Persistent data, these operations are called as Persistence operations. Below are the persistence operations

C- Create /Insert

U – Update

R – Retrieve

D – Delete

The approach to managing persistent data has been a key design decision in every software project we've worked on. Given that persistent data isn't a new or unusual requirement for Java applications, you'd expect to be able to make a simple choice among similar, well-established persistence solutions. Think of web application frameworks (Struts versus WebWork), GUI component frameworks (Swing versus SWT), or template engines (JSP versus Velocity). Each of the competing solutions has various advantages and disadvantages, but they all share the same scope and overall approach. Unfortunately, this isn't yet the case with persistence technologies, where we see some wildly differing solutions to the same problem.

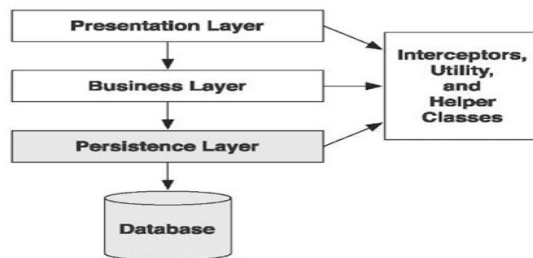
For several years, persistence has been a hot topic of debate in the Java community. Many developers don't even agree on the scope of the problem. Is persistence a problem that is already solved by relational technology and extensions such as stored procedures, or is it a more pervasive problem that must be addressed by special Java component models, such as EJB entity beans? Should we hand-code even the most primitive CRUD (create, read, update, delete) operations in SQL and JDBC, or should this work be automated? How do we achieve portability if every database management system has its own SQL dialect? Should we abandon SQL completely and adopt a different database technology, such as object database systems? Debate continues, but a solution called object/relational mapping (ORM) now has wide acceptance. Hibernate is an open source ORM service implementation.

Hibernate is an ambitious project that aims to be a complete solution to the problem of managing persistent data in Java. It mediates the application's interaction with a relational database, leaving the developer free to concentrate on the business problem at hand. Hibernate is a nonintrusive solution. You aren't required to follow many Hibernate-specific rules and design patterns when writing your business logic and persistent classes; thus, Hibernate integrates smoothly with most new and existing applications and doesn't require disruptive changes to the rest of the application.

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's 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 don't form a layer, because they don't obey the rules for interlayer dependency in a layered architecture.

Let's now take a brief look at the various ways the persistence layer can be implemented by Java applications. Don't worry—we'll get to ORM and Hibernate soon. There is much to be learned by looking at other approaches.

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 don't 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're

certain you'll be able to find a solution that meets your requirements, both business and technical. It's 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. A great many details can easily be overlooked in such a large project—as both the authors know from experience! Even if an existing tool doesn't 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

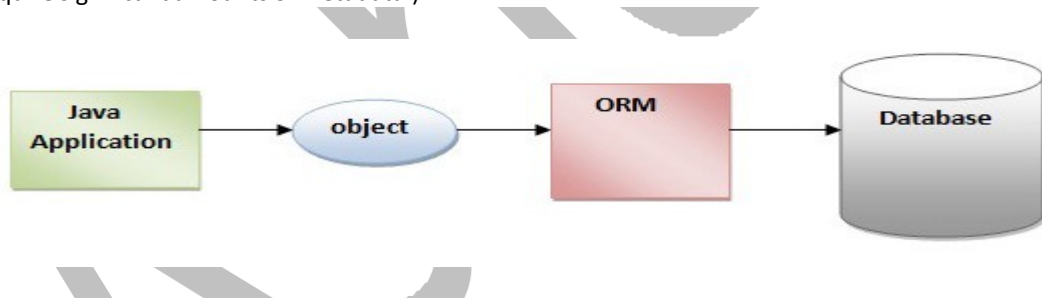
Now that we've looked at the alternative techniques for object persistence, it's time to introduce the solution we feel is the best, and the one we use with Hibernate: ORM. Despite its long history (the first research papers were published in the late 1980s), the terms for ORM used by developers vary. 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?

In a nutshell, 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 wouldn't 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.)



FAQ Isn't ORM a Visio plug-in? The acronym ORM can also mean object role modeling, and this term was invented before object/relational mapping became relevant. It describes a method for information analysis, used in database modeling, and is primarily supported by Microsoft Visio, a graphical modeling tool. Database specialists use it as a replacement or as an addition to the more popular entity-relationship modeling. However, if you talk to Java developers about ORM, it's usually in the context of object/relational mapping.

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're using the term full ORM to include any persistence layer where SQL is automatically generated from a metadata-based description. We aren't 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 Fussell (Fussell, 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. This level of functionality can hardly be achieved by a homegrown persistence layer—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're using in this topic.

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 nonbenefit.

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, you 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 almost always involve changes to the other, and often the design of one representation is compromised to accommodate the existence of the other. (What almost always 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 really 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 shouldn't 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's usually much easier to develop a cross-platform application using ORM. Even if you don't require cross-platform operation, an ORM can still help mitigate some of the risks associated with vendor lock-in.

HIBERNATE INTRODUCTION

Hibernate is a full object/relational mapping tool that provides all the previously listed ORM benefits. The API you're 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.

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	
Developer(s)	Red Hat
Stable release	v5.2.11 / September 13, 2017; 13 days ago
Repository	github.com/hibernate/hibernate-orm
Development status	Active
Written in	Java
Operating system	Cross-platform (JVM)
Platform	Java Virtual Machine
Type	Object-relational mapping
License	GNU Lesser General Public License
Website	hibernate.org

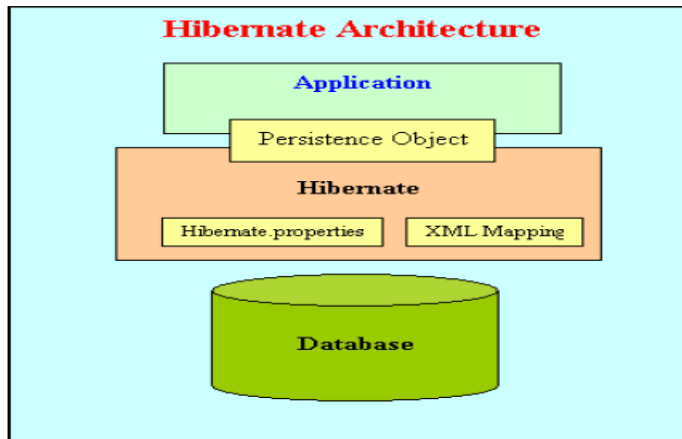
- ✓ Hibernate is an Object/Relational Mapping framework for Java environments.
- ✓ Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities.
- ✓ It can also significantly reduce 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 lots of features in the persistence data layer.

Hibernate Architecture

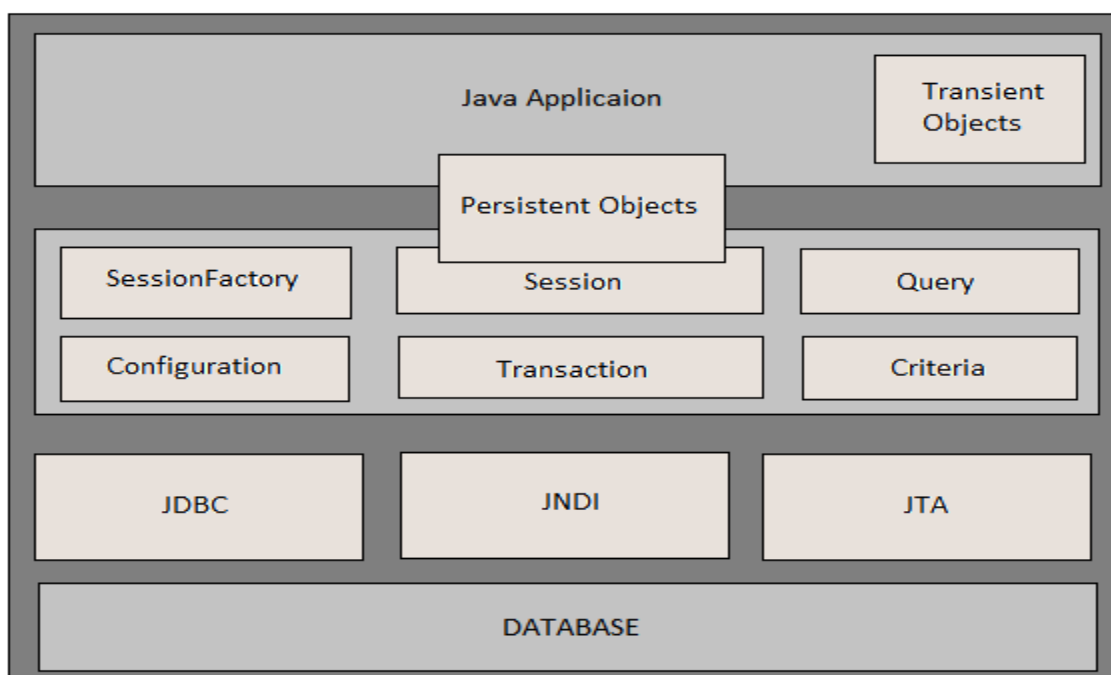
The Hibernate architecture is layered to keep us isolated from having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

There are 4 layers in hibernate architecture they are, java application layer, hibernate framework layer, backhand api layer and database layer. Let's 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



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's 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 configuration 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 Hibernate 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
```

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
```

Hibernate Application Requirements

Any hibernate application, for example consider even first hello world program must always contains 4 files totally.

- POJO class
- Mapping XML or Annotations
- Configuration XML
- One java file to write our business logic

Actually these are the minimum requirement to run any hibernate application, and in fact we may require any number of POJO classes and any number of mapping xml files (Number of POJO classes = that many number of mapping xmls), and only one configuration xml and finally one java file to write our logic.

Hibernate Persistent Classes or POJO classes

Persistent classes are those java classes whose objects have to be stored in the database tables. They should follow some simple rules of Plain Old Java Object programming model (POJO).

- a) A persistence class should have a default constructor
- b) A persistence class should have an id to uniquely identify the class objects. All attributes should be declared as private
- c) Public getter and setters should be defined to access the class attributes

```
public class Product {  
  
    private Integer pid;  
    private String pname;  
    private Double price;  
  
    public Integer getPid() {  
        return pid;  
    }  
    public void setPid(Integer pid) {  
        this.pid = pid;  
    }  
    public String getPname() {  
        return pname;  
    }  
    public void setPname(String pname) {  
        this.pname = pname;  
    }  
    public Double getPrice() {  
        return price;  
    }  
    public void setPrice(Double price) {  
        this.price = price;  
    }  
}
```

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="dialect">databaseDialectClass</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 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 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)
- ❖ ClientForSave.java (java file to write our hibernate logic)

```
public class Product {  
  
    private Integer productId;  
    private String proName;  
    private Double price;  
  
    //Setters and getters  
  
}
```

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




"-//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>
```

In this mapping file, Product class is linked with PRODUCTS table in the database, and next is the id element, means in the database table what column we need to take as primary key column, that property name we need to give here, here property name “productId” which will mapped with “pid” column in the table.

And “proName” is mapped with “pname” column of the PRODUCTS table, here we have not specified any column for the property price, this means that, our property name in the pojo class and the column name in the table both are same.

Remember: the first 3 lines is the DTD for the mapping file, as a programmer no need to remember but we need to be very careful while we are copying this DTD, program may not be executed if we write DTD wrong, actually we have separate DTD’s for Mapping xml and Configuration xml files.

```
"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>
```

```
public class SaveProduct {
    public static void main(String[] args) {
        Configuration cfg = new Configuration()
        cfg.configure("hibernate.cfg.xml");
        StandardServiceRegistry registry =
        new StandardServiceRegistryBuilder().configure().build();
        SessionFactory sessionFactory =
            new MetadataSources( registry ).buildMetadata().buildSessionFactory();
        Session hsession = sessionFactory.openSession();
        Transaction tx = hsession.beginTransaction();

        Product p = new Product();
        p.setProductId(101);
        p.setProName("Apple");
        p.setPrice(35000.50);

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

CURD Operations in Hibernate

Storing record into table

In hibernate, we generally use one of below two versions of save() method:

```
public Serializable save(Object object) throws HibernateException

public Serializable save(String entityName, Object object) throws HibernateException
```

Both **save()** methods take a transient object reference (which must not be null) as an argument. Second method takes an extra parameter '**entityName**' which is useful in case you have mapped multiple entities to a Java class. Here you can specify which entity you are saving using **save ()** method

Hibernate Select Query Example

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. It's all what load () method need from you to work it correctly.

Let's 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));  
        Product 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

Difference between load() and get() method in hibernate session

The difference between both methods lies in return value "if the identifier does not exist in database". In case of get() method you will get return value as NULL if identifier is absent; But in case of load() method, you will get a runtime exception

org.hibernate.ObjectNotFoundException: No row with the given identifier exists

Hibernate Update Query Example : 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  
    }  
}
```

What is the difference between save(), persist(), saveOrUpdate() and update() methods ?

save() method will store entity into database and returns Serializable Id

persist() method will store entity into database and will not return any value

saveOrUpdate() method, if entity already available in database it will update otherwise it will store entity into database

update() method is used to update the record, when we call update(), if that entity not available in database it will throw exception.

Hibernate Delete Query Example

```
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  
    }  
}
```

Auto DDL in Hibernate

One of the major advantage of hibernate is Auto DDL support. It will create the tables Automatically. 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 prefixed 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 created in the Java Virtual Machine. 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.

```
public class SingletonExample {  
    // Static member holds only one instance of the  
    // SingletonExample class  
  
    private static SingletonExample singletonInstance;  
  
    // SingletonExample prevents any other class from instantiating  
    private SingletonExample() {  
    }  
  
    // Providing Global point of access  
    public static synchronized SingletonExample getSingletonInstance() {
```

```
    if (null == singletonInstance) {
        singletonInstance = new SingletonExample();
    }
    return singletonInstance;
}

public void printSingleton(){
    System.out.println("Inside print Singleton");
}
}
```

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();
    }

}
```

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:

An object which is not associated with hibernate session and does not represent a row in the database is considered as transient. It will be garbage collected if no other object refers to it. An object that is created for the first time using the new() operator is in transient state. When the object is in transient state then it will not contain any identifier (primary key value). You have to use save, persist or saveOrUpdate methods to persist the transient object.

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 state then it contain identity but you can't 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.

Generators in Hibernate

When inserting a new record/row in a database table for the instantiated java object, the ID column must be populated with a unique value in order to uniquely identify that persisted object.

- ✓ 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 IdentifierGenerator 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:

increment	generates identifiers of type long, short or int that are unique only when no other process is inserting data into the same table. Do not use in a cluster.
identity	supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type long, short or int.
sequence	uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi or a generator in Interbase. The returned identifier is of type long, short or int
hilo	uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a table and column (by default hibernate_unique_key and next_hi respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database.
seqhilo	uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a named database sequence.
uuid	uses a 128-bit UUID algorithm to generate identifiers of type string, unique within a network (the IP address is used). The UUID is encoded as a string of hexadecimal digits of length 32. guid uses a database-generated GUID string on MS SQL Server and MySQL.
native	picks identity, sequence or hilo depending upon the capabilities of the underlying database. assigned lets the application to assign an identifier to the object before save() is called. This is the default strategy if no <generator> element is specified.
select	retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value
foreign	uses the identifier of another associated object. Usually used in conjunction with a <one-to-one> primary key association.

Composite Primary Key in Hibernate

- If the database table has more than one column as primary key then we call it as composite primary key.
- If the table has one primary key then in hibernate mapping file we need to configure this column by using <id> element.
- if the table has multiple primary key columns, in order to configure these primary key we need to use <composite-id=""> element in our hibernate hbm file

Let us consider there is a need for persisting a Book Object in a table BOOK_INFO. Book class contains properties such as isbn, bookName, authorName, category, price out of which isbn, bookName and authorName are part of a composite primary key fields in the table BOOK_INFO. Steps to implement this requirement is mentioned below.

To implement this application, follow the steps mentioned below.

```
import java.io.Serializable;

public class Book implements Serializable {

    private int isbn;
    private String bookName;
    private String authorName;
    private String category;
    private Double price;

    private static final long serialVersionUID = 4430953665101945676L;

    // Setter and Getters

}
```

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

```
<?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="info.ashok.Book" table="BOOK">
        <composite-id>
            <key-property name="isbn" column="ISBN" />
            <key-property name="bookName" column="BOOKNAME" />
        </composite-id>
        <property name="authorName" column="AUTHORNAME" />
        <property name="price" column="PRICE" />
    </class>
</hibernate-mapping>
```

```
public class InsertCPK {
    public static void main(String[] args) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        Book book = new Book();
        book.setIsbn(101);
        book.setBookName("Head First Java");
        book.setAuthorName("Cathie Sierra");
        book.setPrice(new Double("500.00"));
        Transaction tx = session.beginTransaction();
        session.save(book);
        tx.commit();
        session.close();
        factory.close();
    }
}
```

```
public class RetrieveCPK {
    public static void main(String[] args) {
        SessionFactory factory = HibernateUtil.buildSessionFactory();
        Session session = factory.openSession();

        Product p = new Product();
        p.setProductId(101);
        p.setPrice(25000);

        Object o = session.get(Product.class, p);
        // here p must be an serializable object,
        Product p1 = (Product) o;
        System.out.println("The price is: " + p1.getProName());
        System.out.println("Object Loaded successfully....!!");
        session.close();
        factory.close();
    }
}
```

———— Composite Primary Key Select Operation ————

Composite Primary Key Using Annotations

```
@Embeddable
public class CustomerID implements Serializable {
    @Column
    private Integer cid;

    @Column
    private Integer phno;
    //setters and getters
}
```

_____ Class to contain all PKs _____

```
@Entity
@Table
public class Customer {

    @Id
    @Embedded
    private CustomerID cid;

    @Column
    private String cname;
    // setters and getters
}
```

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");
    }
}
```

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 isn’t 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

Proxool - Distributed with Hibernate

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>
```

Explanation about the properties:

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>
```

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");
    session.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`

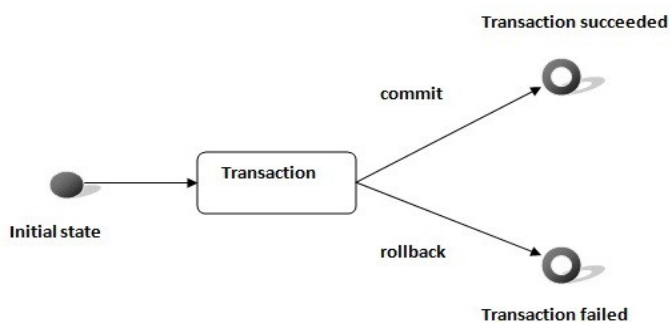
```
SessionFactory sf = HibernateUtil.getSessionFactory();

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

A **transaction** simply represents a unit of work. In such case, if one step fails, the whole transaction fails (which is termed as atomicity). A transaction can be described by ACID properties (Atomicity, Consistency, Isolation and Durability).



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 wasCommitted()** checks if the transaction is committed successfully.
8. **boolean wasRolledBack()** checks if the transaction is rolledback successfully.

```
Session session = null;
Transaction tx = null;
try {
    session = sessionFactory.openSession();
    tx = session.beginTransaction();
    // some action
    tx.commit();
} catch (Exception ex) {
    ex.printStackTrace();
    tx.rollback();
} finally {
    session.close();
}
```

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's 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

Basic Hibernate Annotations

@Entity annotation marks this class as an entity.

@Table 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 annotation marks the identifier for this entity.

@GeneratedValue 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 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 createdAt;

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

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 Blob and Clob in Hibernate**

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 and photos 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.

```
-----Employee.java-----
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Lob;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Version;

@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;

```

```

}-----ImageDemo.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();
    }
}

```


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.

```
-----Defect.java-----
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;

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



```

@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
}
-----VersionInsertDemo.java-----
public class VersionInsertDemo {

    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();
    }
}

```

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 :

```

-----VersionUpdateDemo.java-----
public class VersioningUpdateDemo {

    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(Ticket.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();
}
}
```

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 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 don't 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 and every 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
- java.sql.Timestamp

-----Employee.java-----

```
import java.time.LocalDateTime;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;

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 LocalDateTime createDateTime;

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

// setters and getters

}
```

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

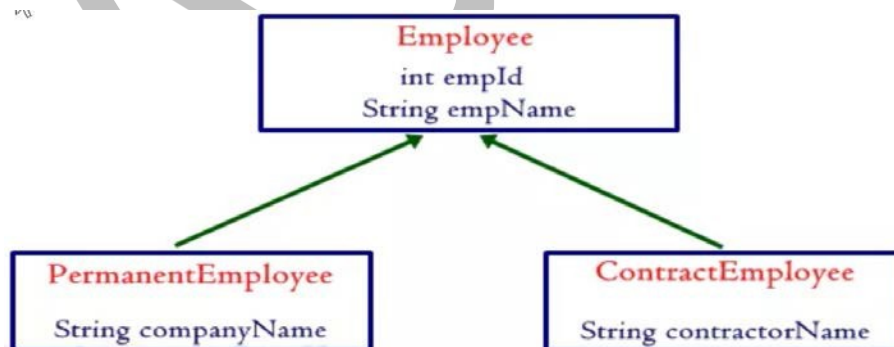
Introduction to Inheritance Mapping In Hibernate

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.

Hibernate supports 3 types of Inheritance Mappings to map classes involved in inheritance with database tables.

- Table Per Hierarchy
- Table Per Concrete class
- Table Per Subclass

Example



In the above hierarchy, three classes are involved where Employee is the super class and PermanentEmployee and ContractEmployee are subclasses.

Now the question is how many tables are required and how to link the tables so that PermanentEmployee gets three properties of empId and empName (from super class), companyName.

Table-per-class-hierarchy: Only one table is created for all the classes involved in hierarchy. Here we maintain an extra discriminator column in table to differentiate between PermanentEmployee and ContractEmployee.

EMPID	DTYPE	EMP_NAME	COMPANY_NAME	CONTRACTOR_NAME
1	Permanent employee	Ameer	CTS	-
2	Permanent employee	Lourde	TCS	-
3	Contract Employee	Prabhu	-	ABD Consultancy
4	Contract Employee	Badru	-	MN Consultancy

Table-per-concrete-class: One table for each concrete class (subclass) is created but not of super class. Here, foreign key is not maintained.

P_EMPLOYEES			C_EMPLOYEES		
EMPID	EMP_NAME	COMPANY_NAME	EMPID	EMP_NAME	CONTRACTOR_NAME
1	Ameer	CTS	3	Prabhu	ABD Consultancy
2	Lourde	TCS	4	Badru	MN Consultancy

Table-per-subclass: One table for each class is created. This hierarchy creates three tables.

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.

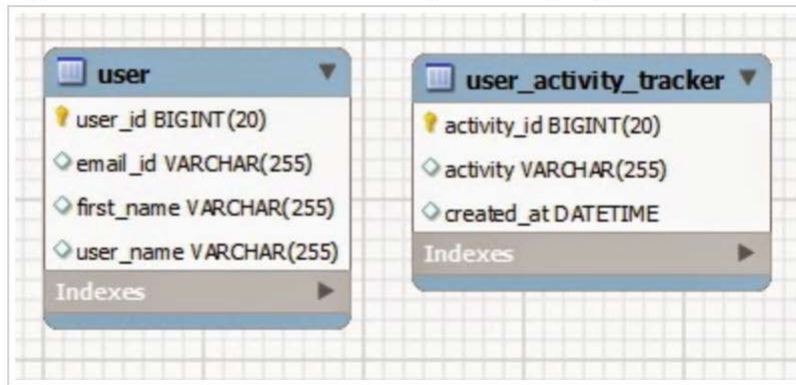
The Interceptor in Hibernate provides callback from the session to the application, allowing the application to inspect and/or manipulate the properties of a persistent object just before it is saved, update, delete, load from the database. One of the common use of interceptor is to track the activity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



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

hibernate.cfg.xml

```
@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

```
@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[] propertyNames, 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, Object[] 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[] propertyNames, 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 committing");
}

// called after committed into database
public void postFlush(Iterator iterator) {
    System.out.println("After committing");
    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 are familiar with SQL then writing HQL would be pretty easy: from SELECT, FROM, ORDER BY to arithmetic expressions and aggregate functions, etc.
- **Fully object-oriented:** HQL doesn't 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 with out doing any further changes to it
- ✓ HQL supports object oriented features like **Inheritance, polymorphism, Associations**(Relation ships)
- ✓ HQL is initially given for selecting object from database and in hibernate 3.x we can do DML operations (insert, update...) too

// In SQL

```
sql> select * from Product
```

Note: Product is the table name...!!!

// In HQL

```
hql> select p from Product p
```

[or]

```
from Product p
```

Note: here p is the reference...!!

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.

// In SQL

```
sql> select pid,pname from Product
```

Note: pid, pname are the columns Product is the table

// In HQL

```
hql> select p.productid,p.productName from Product p
```

[or]

```
from Product p ( we should not start from, from key word here because  
we selecting the columns hope you are getting me )
```

Note: here p is the reference...!!

productid,productName are POJO variables

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)

Example

// In SQL

```
sql> select * from Product where pid=?
```

Note: Product is the table

// In HQL

```
hql> select p from Product p where p.productid=?
```

[or]

```
select p from Product p where p.productid=:java4s
```

[or]

```
from Product p where p.productid=?
```

[or]

```
from Product p where p.productid=:java4s
```

Note: Here p is the reference...!!

```
@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
}
```

-----MyApp.java-----

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

        // Create the student object.
        Student student = new Student();
        // Setting the object properties.
        student.setFirstName("Vivek");
        student.setLastName("Solenki");
        student.setClassName("MCA ");
        student.setRollNo("MCA/07/70");
        student.setAge(27);
        // Get the session object.
        Session session = HibernateUtil.getSessionFactory().openSession();
        // Start hibernate transaction.
        session.beginTransaction();
```

```
// Persist the student object.
session.save(student);

// 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("First Name: " + stu1.getFirstName());
System.out.println("Last Name: " + stu1.getLastName());
System.out.println("Class: " + stu1.getClassName());
System.out.println("RollNo: " + stu1.getRollNo());
System.out.println("Age: " + stu1.getAge());

// 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("First Name: " + stu2.getFirstName());
System.out.println("Last Name: " + stu2.getLastName());
System.out.println("Class: " + stu2.getClassName());
System.out.println("RollNo: " + stu2.getRollNo());
System.out.println("Age: " + stu2.getAge());

// 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("First Name: " + stu3.getFirstName());
System.out.println("Last Name: " + stu3.getLastName());
System.out.println("Class: " + stu3.getClassName());
System.out.println("RollNo: " + stu3.getRollNo());
System.out.println("Age: " + stu3.getAge());

// 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()`. The `setFirstResult()` 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.

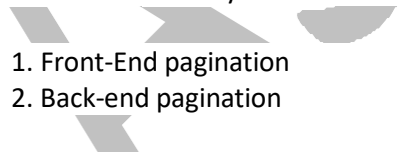
What is Pagination?

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 again and again.

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 can't 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 less performance when compared with front-end pagination.

- ✓ This type of pagination is suggestible, if number of 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.

Code Block

```
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.getId());
9     System.out.println("Name : " +account.getName());
10    System.out.println("Balance : " +account.getBalance());
11 }
```

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. This tag is given by Apache people.

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 several `createCriteria()` 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 on Restrictions, 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. The ilike() 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 is le(). 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 the `org.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();
```

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 the `org.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 later 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 of 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 CallProcDemo{  
public static void main(String[] args) {  
    getAllEmpRecords(101,'Ashok',25000,'M');  
}  
  
public void insertEmpRecord(Integer id,String name,Double salary,Character gender) {  
    try {  
        SessionFactory sf = HibernateUtil.getSessionFactory();  
        Session hsession = sf.openSession();  
        Transaction tx = hsession.beginTransaction();  
  
        StoredProcedureQuery query = hsession  
            .createStoredProcedureQuery("INSERTEMPRECORDPROC");  
  
        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();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
}
```


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 Paramter 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.aitz.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 javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

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  
  
    @Override  
    public String toString() {  
        return "Movie [id=" + id + ", name=" + name + ", year=" + year + "];"  
    }  
}  
}
```

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

First 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.

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.

First level cache in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you can not 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.

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

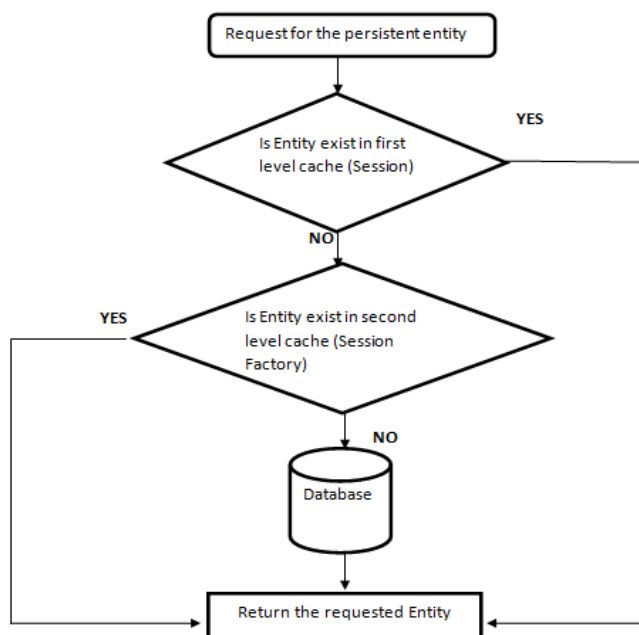
Though we cannot disable the first level cache in hibernate, but we can certainly remove some of objects from it when needed. This is done using two methods:

```
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, then there is no way that second level cache update itself until "time-ToLiveSeconds" 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 database 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.

-----Hibernate config file changes-----

```
<property name="hibernate.cache.use_second_level_cache">true</property>

<property name="hibernate.cache.region.factory_class">
    org.hibernate.cache.ehcache.EhCacheRegionFactory
</property>

<property name="net.sf.ehcache.configurationResourceName">
    ehcache.xml
</property>
```

-----ehcache.xml-----

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="true"
4     monitoring="autodetect" dynamicConfig="true">
5     <diskStore path="java.io.tmpdir/ehcache" />
6     <defaultCache maxEntriesLocalHeap="10000" eternal="false"
7         timeToIdleSeconds="120" timeToLiveSeconds="120" diskSpoolBufferSizeMB="30"
8         maxEntriesLocalDisk="10000000" diskExpiryThreadIntervalSeconds="120"
9         memoryStoreEvictionPolicy="LRU" statistics="true">
10         <persistence strategy="localTempSwap"/>
11     </defaultCache>
12     <cache name="olyanren.java.cache.ehcache.model.Admin"
13         maxElementsInMemory="500"
14         eternal="true"
15         timeToIdleSeconds="0"
16         timeToLiveSeconds="100"
17         overflowToDisk="false"
18         />
19     <cache
20         name="org.hibernate.cache.StandardQueryCache"
21         maxEntriesLocalHeap="5"
22         eternal="false"
23         timeToLiveSeconds="120">
24         <persistence strategy="localTempSwap"/>
25     </cache>
26     <cache name="org.hibernate.cache.spi.UpdateTimestampsCache"
27         maxEntriesLocalHeap="5000" eternal="true">
28         <persistence strategy="localTempSwap"/>
29     </cache>
30 </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 LRU 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 cleared after 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".

timeToldleSeconds enables cached object to be kept in as long as it is requested in periods shorter than timeToldleSeconds.

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 **timeToldleSeconds** = 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 milli-seconds 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.

timeToldleSeconds="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 up-dated**.

Transactional: If you need a fully transactional cache. Only **suitable in a JTA** environment.

```
import org.hibernate.Session;

import com.ait.s.Employee;
import com.ait.s.HibernateUtility;

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

        session.beginTransaction();

        Employee Employee = null;

        Employee = (Employee) session.load(Employee.class, 11);

        System.out.println("Employee from the Database => "+Employee);
        System.out.println();

        System.out.println("Going to print Employee *** from First Level Cache");
        // second time loading same entity from the first level cache
        Employee = (Employee) session.load(Employee.class, 11);
        System.out.println(Employee);
        // removing Employee object from the first level cache.
        session.evict(Employee);
        System.out.println("Object removed from the First Level Cache");
        System.out.println();
        System.out.println("Going to print Employee *** from Second level Cache");
        Employee = (Employee) session.load(Employee.class, 11);
        System.out.println(Employee);
        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");
        Employee = (Employee) session2.load(Employee.class, 11);
        System.out.println(Employee);
        session2.getTransaction().commit();
    }
}
```

Query caching

The Query Cache does not cache the entities unlike Second Level Cache, It caches the queries and the return identifiers of the entities. Once it cached the identifiers then, 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 cache the identifiers not the complete entities.

Enabling Query cache

```
<property key="hibernate.cache.use_query_cache">true</property>
```

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.

Association Mapping or Relationships in Hibernate

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.

Using hibernate, if we want to put relationship between two entities [Objects of two POJO classes], then in the database tables, there must exist foreign key relationship, we call it as Referential integrity.

The main advantage of putting relationship between objects is, we can do operation on one object, and the same operation can transfer onto the other object in the database [remember, object means one row in hibernate terminology]

While selecting, it is possible to get data from multiple tables at a time if there exists relationship between the tables, nothing but in hibernate relationships between the objects.

Using hibernate we can put the following 4 types of relationships

One-To-One

One-To-Many

Many-To-One

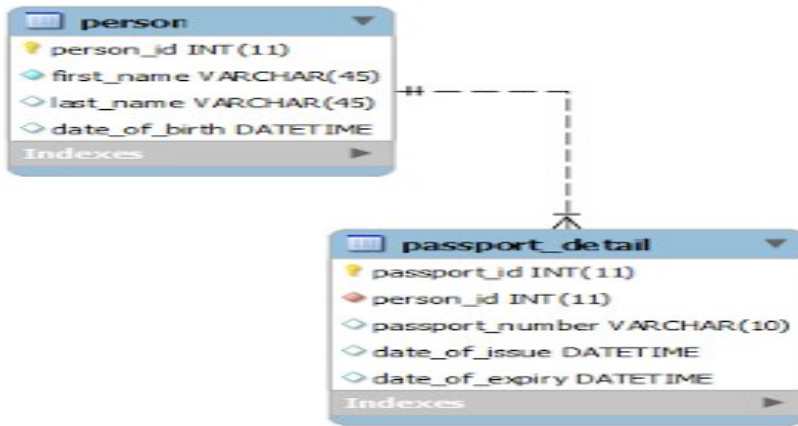
Many-To-Many

One to One Relation ship

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 PassportDetail 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-many association)
Here is the example of author and book table for our One-To-Many association example

Author table:

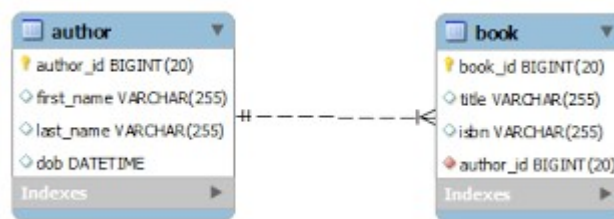
author_id	first_name	last_name	dob
101	Joshua	Bloch	1961-08-28
102	Watts	S. Humphrey	1927-07-04

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 setters
}
}
```



```
@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:

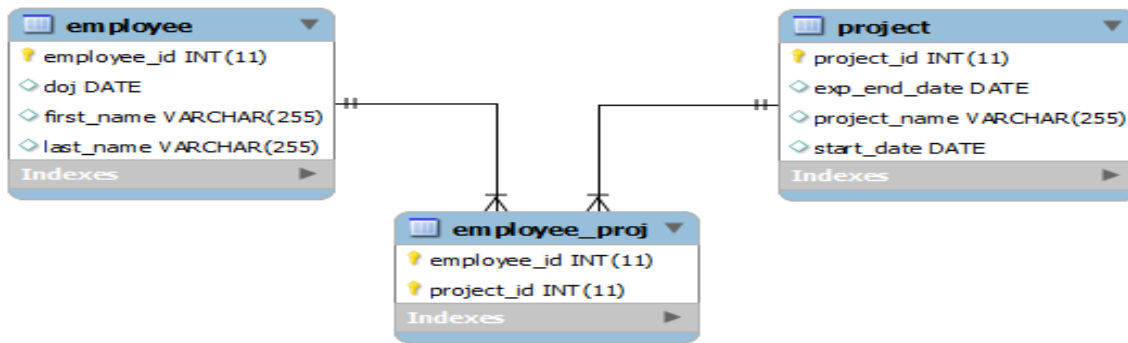
project_id	project_name	start_date	exp_end_date
101	MARS	2013-01-01	2015-08-28
102	SBA	2014-10-02	2016-07-04

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 or not 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 they are required by the program. 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 and every 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.

There are following types of cascade:

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 check 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
}
```