**Hibernate Prerequisites:**

1. Install the DB Server (preferably MySQL)
2. Create a DB (by name COLLEGE / INSTITUTE)
3. Create a tables in the above DB (by name STUDENTS\_INFO & STUDENTS\_OTHERINFO)

**CREATE** **TABLE** students\_info

( regno **INT**(50) **NOT** **NULL**,

firstname **VARCHAR**(50),

middlename **VARCHAR**(50) **DEFAULT** ‘Not Avilable’,

lastname **VARCHAR**(50),

**PRIMARY** **KEY** (regno)

);

**CREATE** **TABLE** students\_otherinfo

( regno **INT**(50) **NOT** **NULL**,

isadmin **VARCHAR**(1) **DEFAULT** ‘N’,

password **VARCHAR**(50) **DEFAULT** ‘qwerty’

**PRIMARY** **KEY** (regno)

);

1. Insert some records in to the above tables (preferably 5 student’s info). Make one student as Admin & rest as normal students.

**Assignment 1:**

1. Create the dynamic web application, which will have the login page. Login page accepts register number & password, passes it to a servlet.
2. Servlet using JDBC interacts with DB validates the user credentials
3. If user has provided invalid credentials, then redirect the request to login page with error information
4. If user has provided valid credentials, then check is admin rights
5. If user is admin then display all the students’ information
6. If user is non-admin then display his own information

**Note:** Use MVC, DTO, DAO & Factory design patterns for this assignment

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Singleton Design Pattern**

Talk about connection Pooling & ask them to design & tell them the importance of Single instance.

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

There are many ways where we can implement singleton design pattern. Few are

1. Lazy initialization
2. Eager initialization
3. Static block initialization
4. Bill Pugh Approach

**1.** **Lazy initialization**

It’s not the best thread-safe implementation because synchronization is very expensive when we are talking about the performance. We can see that the synchronized method getInstance() does not need to be checked for synchronization after the object is initialized.

**public** **class** Singleton

{

**private** **static** Singleton *instance* = **null**;

**private** Singleton(){ }

/\*public static Singleton getInstance()

{

if (instance == null)

{

synchronized (Singleton.class)

{

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}//End of getInstance()

\*/

**public** **synchronized** **static** Singleton getInstance()

{

**if** (*instance* == **null**)

{

*instance* = **new** Singleton();

}

**return** *instance*;

}//End of getInstance()

}//End of Class

**2.** **Eager initialization**

If the program will always need an instance, or if the cost of creating the instance is not too large in terms of time/resources, the programmer can switch to eager initialization, which always creates an instance

**Advantages:**

* There is no need to synchronize the getInstance() method, meaning all threads will see the same instance and no (expensive) locking is required.
* The final keyword means that the instance cannot be redefined, ensuring that one (and only one) instance ever exists.

**public** **class** Singleton

{

**private** **static** **final** Singleton *instance* = **new** Singleton();

**private** Singleton() {}

**public** **static** Singleton getInstance()

{

**return** *instance*;

}

}

**3.** **Static Block Initialization**

**public** **class** Singleton

{

**private** **static** **final** Singleton *instance*;

**private** Singleton() { }

**static**

{

**try** {

*instance* = **new** Singleton();

} **catch** (Exception e) {

**throw** **new** RuntimeException("Darn, an error occurred!", e);

}

}

**public** **static** Singleton getInstance() {

**return** *instance*;

}

}

**4.** **Bill Pugh Approch**

[University of Maryland](http://en.wikipedia.org/wiki/University_of_Maryland,_College_Park) Computer Science researcher [Bill Pugh](http://en.wikipedia.org/wiki/William_Pugh) has written about the code issues underlying the Singleton pattern when implemented in Java.Pugh's efforts on the "[Double-checked locking](http://en.wikipedia.org/wiki/Double-checked_locking)" idiom led to changes in the Java memory model in Java 5 and to what is generally regarded as the standard method to implement Singletons in Java. The technique known as the [initialization on demand &](http://en.wikipedia.org/wiki/Initialization_on_demand_holder_idiom) is as lazy as possible, and works in all known versions of Java. It takes advantage of language guarantees about class initialization, and will therefore work correctly in all Java-compliant compilers and virtual machines.

The nested class is referenced no earlier (and therefore loaded no earlier by the class loader) than the moment that getInstance() is called. Thus, this solution is [thread-safe](http://en.wikipedia.org/wiki/Thread_safety) without requiring special language constructs (i.e. volatile or synchronized).

**public** **class** Singleton

{

//Private constructor prevents instantiation from other classes

**private** Singleton() { }

/\*\*

\* SingletonHolder is loaded on

\* the first execution of Singleton.getInstance()

\* or the first access to SingletonHolder.INSTANCE, not before.

\*/

**private** **static** **class** SingletonHolder {

**public** **static** **final** Singleton *INSTANCE* = **new** Singleton();

}

**public** **static** Singleton getInstance() {

**return** SingletonHolder.*INSTANCE*;

}

}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

* Consider designing an internal “object pool” that will allow objects to be reused instead of created from scratch.
* Consider making all constructors private or protected.