Internet Computing Workshop

Part2: Hibernate
Contents

• Why ORM and Hibernate?
• A simple example to provide an overview
• Database patterns
• Log4j
• Object lifecycle in Hibernate
• Equality
• Session and its use
• Querying (some HQL!)
• Cascading persistence
• Transactions
• Mapping
Announcement

JDBC exam will be on Wed 27 Jan
The exam will be on JDBC ONLY!
You can bring your notes
You will have access to API
But, you are not allowed to go online or use email or any sort of chat software
A high level view of the course

- Aim: to learn engineering of multi-tired web based systems
As a java programmer

Now that you know your JDBC:
What is the most striking difference between writing a conventional Java program and writing a program that involves JDBC and database connections?
As a java programmer

What are your observations?

…

• They don’t map very well
• Writing CRUD (create, read, update, delete) primitive statements in JDBC is very time consuming and error prone
• How do we achieve portability when it comes to different dialects of SQL in DBMSs
• Imagine how hard it is to have to use different databases and different flavors of SQL
Object relational paradigm mismatch

is often caused by

• Different levels of granularity
• The problem related to subtypes
• The problem of identity, when two things are equal
• Implementation of associations
• Challenges of navigation
Simple case

No mismatch here

```java
public class User {
    private String username;
    private String name;
    private String address;
    private Set billingDetails;
}
```
Simple case

```java
public class BillingDetails {
    private String accountNumber;
    private String accountName;
    private String accountType;
    private User user;
}
```

You can create the same model in SQL with another table for BillingDetails

```sql
create table USERS (
    USERNAME varchar(15) not null primary key, 
    NAME varchar(50) not null, 
    ADDRESS varchar(100)
)
```
What if we want to include users Address?

Shall we add a new table? There are different variants of address, shall we use user-defined addresses? To Java it is just an address, yet another attribute! Different level of granularity!
Consider the above:
In Java we have subtyping
In SQL we probably use supertable and subtables. These two are different- ORM gives proper solution.
Identity

In Java there are two different notions of equality:

- Object identity (a==b i.e. Same memory location)
- Equality by value, using equals() method

In databases it is on the basis of primary key or system defined keys (surrogate keys)
OO representation of associations is via *object reference*.

In DBs it is key value matching, hence bidirectional. In OO, the object reference is a pointer and can be unidirectional (in any of the two directions) or bidirectional.

In OO navigation is via pointer mechanism, in databases it is via table join....
What is ORM?

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.

What is metadata/metamodel?

Metadata is data about data. Schema description (DDL) vs. data description, DTD vs HTML, XSD to XML, Annotation vs. java code, UML metamodel vs. UML model.
ORM

works by (reversibly) transforming data from one representation to another-like compilers:

• 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
Advantages of ORM

- **Productivity**, because of automation
- **Maintainability**, raising the level of abstraction via use of mappings
- **Performance**, by using complex optimisation techniques
- **Vendor independence**, a middleware to cater for various SQL dialects
Part 2 of the course:

We only cover basics of Hibernate:

Step 1: study of a Hello World example so that you can see a complete—although very simple application

Step 2: we will elaborate on samples of the development processes— in particular different mappings
Different Hibernate Development Processes
Step 1: Hello World application

1) Start from Java implementation
2) Create mapping XML file for the meta data
3) Use it to generate the database schema

(You can also use Java Annotations instead of the XML file)
Hello World application

A java application that store and retrieve Messages from the database
Code by Alan Sexton, available at MessageExample.zip has 6 files:
Message.java
Main.java
Message.hbm.xml
hibernate.properties
log4j.properties
ehcache.xml
Message.java

This is the class that is going to be persisted to the database

Message

Id: Long
Text: String

1..*

1

nextMessage
import java.io.Serializable;
public class Message implements Serializable {
    private Long id;
    private String text;
    private Message nextMessage;

    public Message() {
        ...
    }
}
public String getText()
{
    return text;
}

public void setText(String text)
{
    this.text = text;
}

A method that provides the next message

public void setNextMessage(Message nextMessage)
{
    this.nextMessage = nextMessage;
}
the **mapping file** that describes how the properties of a `Message.java` should be mapped to columns of tables in the database (along with other necessary information such as how keys are generated, what database constraints and indexes should be maintained etc.)

Hibernate uses this file to generate the DB schema automatically

Open the file
Notice, document says:

- **Message** object is persisted to a table called **messages**
- Attribute **id** is mapped to column called **message_id**
- `<generator class="sequence"/>` is used to specify the class name to be used to generate the primary key for new record while saving a new record.
Notice, document says:

- **nextMessage** is mapped to NEXT_MESSAGE_ID – remember association with *many-to-one multiplicity*

- You can make NEXT_MESSAGE_ID a foreign key, which hibernate takes care of, by replacing the line with

  ```xml
  <many-to-one name="nextMessage"
    cascade="all" column="next_message_id"
    foreign-key = “FK_next_message”/>
  ```

WE WILL LOOK AT THE DETAILS LATER
Hibernate properties

Line 1-5 information about jdbc, database and its dialect

Line 10:

`hibernate.show_sql=false`

You generally don’t see the sql, but for debugging and testing, you can switch on.

Be aware statements are optimised and may appear in unfamiliar order!
Hibernate properties

Line 12-17 about connection pool
C3P0 is a connection pooling software-c3p0.jar must be included
Announcement

JDBC exam will be on Wed 27 Jan
The exam will be on JDBC ONLY!
You can bring your notes
You will have access to API
But, you are not allowed to go online or use email or any sort of chat software
Contents-updates

• Why ORM and Hibernate?
• A simple example to provide an overview
• Database patterns
• Log4j
• Object lifecycle in Hibernate
• Equality
• Session and its use
• Querying (some HQL!)
• Cascading persistence
• Transactions
• Mapping
Recap

Quickly flip through slides
Why ORM?
What ORM/Hibernate does?
Simple example.
Message.java
Hibernate.properties
Mapping file: Message.hbm.xml
DB interaction pattern

What do we mean by a Pattern?
There are six main (vertical) steps

1. **Configuration**: Create a Configuration object, load the configuration parameters from the `hibernate.properties` and adjust them as required (for example reading a password or user name)

2. **SessionFactory**: Create a `SessionFactory` object from the Configuration object (Only one! Share it!)
3. **Session**: obtain one session per unit of work from the SessionFactory. Sessions must not be shared between different threads.

4. **Begin Transactions**: Inside a try/catch block get a Transaction object by calling `beginTransaction()` on the Session object.

5. **Interact with the database**

6. **Commit**: Call `commit()` on the Transaction object - close the try block - handling exceptions and closing the Session object.
SessionFactory

The SessionFactory object is a heavyweight, thread safe object. You would normally share one such object between all your threads in a web application.

So the following is suitable:

```java
private static SessionFactory sessions = null;
```
Session object

Hibernate Session:
A single-threaded nonshared object that represents a particular unit of work with the database.

It has the persistence manager API you call to load and store objects:
1) Begin transactions- invoke beginTransaction()
2) Save, close...
Session object

Hibernate Session:
The Session internals consist of a queue of SQL statements that need to be synchronized with the database at some point and a map of managed persistence instances that are monitored by the Session
Interaction with DB

Explicitly by calling methods of Session to associate objects to the database (i.e., map them to the database), execute queries, load, save, delete…

Implicitly by calling property mutators on mapped objects that result in database being updated. setFoo(…)

Implicitly by referencing non-mapped objects from mapped objects which (in certain circumstances) can cause the non-mapped objects to be added to the database.
Interaction with DB

*implicitly* by unreferencing mapped objects from other mapped objects which (in certain circumstances) can cause the unreferenced objects to be deleted from the database.

We will discuss this later in Query section!
Sample code for the interaction

Main_SixSteps.java is a small program explaining Interaction with DB by saving one message to a your database.

See Sample_SixSteps directory. You must

1. Include Message.java. Hibernate.properties, Message.bhm.xml and ehcache.xml
2. Don’t forget to modify Hibernate.properties by putting your dbname and username
3. To re-run the program don’t forget to drop (drop cascade option in pgadmin) both created databases AND sequence.
Sample code

Explanation follows at Main_SixSteps.java

TODO: Write a JDBC program which implements the same usecase as the one implemented in Main_SixSteps.java. How does Hibernate and JDBC compare? see earlier slides!
Back to Main.java

For the moment forget about Log4J stuff.
Input: “create” and program does the following
1) Add Message (similar to Main_SixSteps.java)
2) Retrieving a Message as a string
3) Updating saving a string of Messages and retrieving them

Notice that Six steps are followed although the above functionality is implemented as methods (refactoring)
Not essential, but you may get a warning when you start.

What is a cache?

There are three main types of cache in Hibernate

1. 1\textsuperscript{st} level cache, when the session is open
2. 2\textsuperscript{nd} level cache, when SessionFactory open
3. Query cache: identifier for individual query

Advanced topics beyond this course!
Use cases

Form follows functionality!

Key role of requirements

**Use cases** describe the requirements of a system in terms of the **interactions** that should occur between the system and various Actors, i.e. people and other systems, that can interact with it.

Use cases are captured via diagrams and text

Use cases, although often not well defined, are a main part of all software development methodologies.
Example of a use case

Name       Add book to order
Initiator  User
Goal       Given a customer email address and a book ISBN, add the book to the customer's order

Main Success Scenario
1. User requests that a book be added to a customer's order
2. System identifies customer from email address
4. System obtains current open order for customer
5. System adds new orderDetail for book to order
Extension  

- Numbers refer to steps in Succ. Scenario

2. No matching customer found
   1. Fail

3. No matching book found
   1. Fail

4. No open order for customer found
   1. Add new order for customer
   2. Resume 5

5. OrderDetail for book exists
   1. add one to quantity for orderDetail
   2. Stop
Use cases

We follow the style suggested by Cheesman and Daniel - see Alan notes.

• Each use case must have a unique name.
• Each use case must have an initiator (an actor who starts the a use case)
• Each use case must have a goal: a short description of the purpose of the use case.
• Each use case must have a single numbered sequence of steps that describe the main success scenario (desirable scenario)
• Extensions depicts alternative scenarios
Rules for steps

- Each step must be of the form "Actor does something" or "Include use case".
- First step indicates the stimulus that caused the use case to be initiated.
- The combination of Actor and Stimulus must be unique across all use cases.
- Steps cannot use parallelism, recursion, iteration, gotos or conditional statements.
- A step can invoke another use case by naming it in an "Include" statement. In such a case, when the step is executed, the included use case is executed and, when it has terminated, execution continues with the next step in the first use case.
Extensions

• A use case may have a list of extensions.
• Each extension is a mini-use case that describes an alternative or addition to the main success scenario.

An extension is made up of:

• The *step number* (*extension point*) in the main success scenario at which the extension applies.

• A *condition* that must be tested before that step is executed. If the condition evaluates to true, the extension is executed, if not, the original step is executed.
Extensions

• Each step can be of the same form and type as in the main success scenario but the last step is of the following forms:
  – "Fail": the use case is terminated with the goal unsatisfied.
  – "Stop": the use case is terminated with the goal satisfied.
  – "Resume N": the extension ends and the use case continues at step N in the main success scenario.
Apache Log4j

This will be a brief lecture: you can learn more from various tutorials and available bookd:

• http://jakarta.apache.org/log4j

• http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/log4j/log4j.html

• Samudra Gupta: Pro Apache log4j
What is logging

Gupta’s book:
Logging is a **systematic** and **controlled** way of representing the **state** of an application in a **human-readable** fashion

State: value of some/all variables
Most people use `system.out.println` or debuggers.
But logs are different from both of the above.
What are the advantages of logging?

• Problem diagnosis
• Quick debugging
• Easy maintenance
• History
• Cost and time saving

I’d like to add the following from my research on process mining. what is PM?

Load balancing, identifying bottlenecks, common scenarios.
Disadvantages

Extra runtime overhead
Programming overhead- more code can mean more errors
Bad logs confuse people
Imposes extra requirements (means allocation of resources= money)
logging is much better than println

1. You can turn a log off to focus on a special aspect of the system
2. Println can not separate log-messages of different degree of importance
3. println just writes to console
Three main objects in log4j

Logger: captures logging information.

Appender: publishes logging information to various destinations (console, file, socket, ..)
   – For example, a **ConsoleAppender** object for printing logging information to a console.

Layout: The Layout object is used to format logging information in different styles (human-readable part!)
Interaction of the three objects:

1. Logger receives a Logging Event.
2. If a filter is associated, approve the request.
3. Pass the logging event to the filter.
4. If Necessary, render and format logging information.
5. Log the rendered object.
How does it work in practice?

Three ways to **configure** log4j

- Properties file- log4j.properties
- XML style of representation
- Programmatic style

We choose the first one- for the other two methods of configuration see
http://jakarta.apache.org/log4j
How does it work in practice?

1. Name a Logger object for the class
   ```java
   private static final Log LOG = LogFactory.getLog(Main.class);
   ```

2. Insert desired txt with level logging via Level-Based Logging Methods
   ```java
   LOG.fatal(“House is on fire”);
   ```

3. Specify Level of logging for the directory and type of Appender in log4j.properties
How does it work in practice?

Example:

```
log4j.logger.org.hibernate.SQL=debug,stdout
```

Logging related to `org.hibernate.SQL` will be carried out at the level of "debug" and will be appended to `stdout`.

- How does it work?
- What happens to subpackages?
- What if `org.hibernate` has different level?

We will answer these next.
Logger object

The main object that you deal with is a Logger object- everything else behind the scene.

A convenient way is to get it is:

```java
private static final Log LOG = LogFactory.getLog(ClassNAME.class);
```

Notice: we can assign the logger in properties file and then retrieve it using

```java
public static Logger getLogger(String name);
```
```java
public static Logger getLogger();
```
public void debug(Object message);

This method prints messages with the level Level.DEBUG.

There are similar methods for other 7 Levels: ALL<DEBUG<INFO<WARN<ERROR<FATAL<OFF

ALL: means all information will be logged
OFF: means nothing will be logged

Each Level prints messages about higher level

Explain!
Level-Based Logging Methods

The configuration can be altered from the log4j.properties; So we can have different level of logging at different time. For example looking at only FATAL messages.

There are also Filter objects to analyze logging information. Filter are beyond the level of this course, see log4j manual.
Logger Hierarchy

There is a hierarchy in

```
log4j.logger.org.hibernate.tool.hb
  m2ddl=debug,stdout
```

as follows:

```
org.hibernate.
org.hibernate.tool
org.hibernate.tool.hbm2ddl
```

What if Level for some layers are not specified?
Level Inheritance

The *inherited level* for a given logger C, is equal to the first non-null level in the logger hierarchy, starting at C and proceeding upwards in the hierarchy towards the root logger.

Draw a picture and explain
Conditions of Successful Logging

Using the logging methods does not guarantee publishing the info

**Logging messages with level less than logger level will be ignored:** Any logging information will be approved by a logger if and only if the level $p$ associated with the logging message is greater than or equal to the level $q$ assigned to the logger.

The `Appender` objects can use `Filter` and further reject— but this does not apply to us.
Biolerplate code for writing to file

```java
log4j.appender.file =
    org.apache.log4j.FileAppender
log4j.appender.stdout.Target = hibernate.log
log4j.appender.stdout.layout = org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern = %d{ABSOLUTE} %5p %c{1}:%L - %m%n

Then we can write:
log4j.rootLogger = info,file
```
Biolerplate code for writing to console

```java
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

Then we can write:
log4j.logger.org.hibernate.SQL=debug,stdout
```
Conversion character for PatternLayout

\textbf{d} date of the logging request. \%d{yyyy-MM-dd} or absolute date \%d{ABSOLUTE}.

\textbf{c} precision request. So if the logger name is \texttt{org.hibernate.tool.hbm2ddl}, then \texttt{c{2}} will print the last two: \texttt{tool.hbm2ddl}

\textbf{F} The name of the file from which the logging request was issued.
Conversion character for PatternLayout

I  The location information.
L  The line number in the program file from which the logging request was issued.
m  The logging message.
M  The method in the program from which the logging request was issued.
n  Platform-dependent line separator- next line
p  The level associated with the logging request.
r  number of milliseconds elapsed from the start of the application before this logging request was issued.
t  invoking thread.
Conversion character for PatternLayout

So what does this mean?

=\%d\{\text{ABSOLUTE}\} \ \%5p \ \%c\{1\}:\%L - \ %m\%n

Notice these characters are case sensitive! Be warned upper case of some characters may mean different things which may put unnecessary pressure on the resources.
Back to Hibernate

• Why ORM and Hibernate?
• A simple example to provide an overview
• Database patterns
• Log4j

At this point we look at hibernate in details:
• Object lifecycle in Hibernate
• Equality
• Session and its use
• Querying (some HQL!)
• Cascading persistence
• Transactions
• Mapping
Hibernate Object Life Cycle

Each object in hibernate is in one of the following states:

- Transient
- Persistent
- Detached
Transient object

No associations with the database
The get garbage collected when they are not referenced to.
Not effected by commits and rollback
A transient object becomes persistent if
1) Invoking `save` on the `session` object
2) Being referenced by a persistent object
Persistent objects

- Object which will be saved to a database
- All persistent objects have primary keys values
- Hibernate follows sophisticated optimisation techniques which means that database can be updated at any time between the update and the end of the transactions
- Notice: Queries are sometime seem to be executed in a different order to what they are formulated.
How to make a Persistent object transient

1. Calling the `delete` method of the `Session` object on a persistent object, removes it from the database and makes it transient

2. If we know the identifier of an object, executing `load` or `get` methods of `Session` results in a transient object

3. Creating a query (`createQuery`) and extracting the results from it can also result in transient objects
Detached object

When a Session is closed, the persistent becomes a detached object.

Clearly, it is possible that an object which is detached become stale.

A detached object may be re-attached later to another Session object.

Idea behind Data Transfer Objects (DTO)
Hibernate Object life cycle

Note: close() and clear() affect all instances in a Session

Adapted from "Hibernate in Action" by C. Bauer & G. King
Recap

• Why ORM and Hibernate?
• A simple example to provide an overview
• Database patterns
• Log4j

At this point we look at hibernate in details:
• Object lifecycle in Hibernate
• Equality
• Session and its use
• Querying (some HQL!)
• Cascading persistence
• Transactions
• Mapping
a==b  Identical objects, point to the same memory location within one JVM

Object class has two methods:

equals(): a.equals(b) checks if two object have the same value

HashCode(): if two objects are equal, they must have the same Hashcode() value.

Note: default implementation of equals() is with the help of ==, i.e.
public boolean equals(Object obj) {
    return (this == obj);
}

Sometime you need to define a new notion of equality, then
1. Override equals()
2. Override Hashcode()

YOU MUST DO BOTH!

BTW, why we need Hashcode() at all?
Identity and equality

Object in a relational database are equal if they belong to the same table and have the same primary key.

Hibernate provides database identity by:

1. The value of the identifier property of a persistent instance
2. The value returned by `Session.getIdentifier(Object entity)`
Identity and equality

Remember:
public class Message {
    private Long id;
    ...
    public Long getId() {
        return this.id;
    }
    private void setId(Long id) {
        this.id = id;
    }
}
In a Session: same db ref. means same objects

within a single Session, you request two objects which have the same database identifier, then you will get references to the same actual objects.

```
a.getId().equals(b.getId())
```

This is because Hibernate uses cache for persistent objects

This means you can use `==` instead of `equals()` for testing two objects are equal.
A subtle point:

In a Session, if two objects have the same id, they are equal

BUT

it is possible that two objects are equal without having the same identifier

So we sometimes need to define a Business Key for a class to establish equality conceptually
Example of when two equal object have different identifiers

Assume that id is a *surrogate* key. Assume Object \texttt{obj1} is saved so the db generates and id, say \texttt{obj1ID}.

Then you create a java Collection object (set, map,...) and save \texttt{obj1} into it.

This will result in a **new id for obj1** which is different from \texttt{obj1ID}.

**Same object has different id in different places**

Q: can’t I reset the new id of the Collection to \texttt{obj1ID}?  
NO, using the set collection class is that the contained object's identity must not change while it is in the collection.
Example of when two equal object have different identifiers

This is a common situation when one-to-many or many-to-many relations

**Question:**

How about I use all the attributes of an object and check for the equality, i.e. two objects are equal if all attributes have the same value?

How about a customer changes her passwd, is it the same customer?

**Conclusion:** we need a conceptual idea of equality.
Business key

Similar to database key
Unique identifier of the object.
Example: for a Customer an email address would be suitable.
If email changes a different Customer is obtained
public class Customer {
    ...  
    public boolean equals(Object other) {
        // Check if within the Session they are equal
        if (this == other)
            return true;
        if (other == null)
            return false;
        if (!(other instanceof Customer))
            return false;
        return false;
    }
}
Example: equality for Customers

// if not null and instance of Customer
// then check for emails
final Customer o = (Customer) other;
return this.emailAddress.equals(o.
    getEmailAddress());
}

// hasCode is the hashCode() of email
public int hashCode() {
    return emailAddress.hashCode();
}
Hibernate Objects

Object that maps into db. You must have:

• default constructor for the class.
• accessors and mutators for all the instance variables of the class.

Good practices (not must):

• implement as Serializable.
• have an id instance variable, usually of type Long, to automatically generated surrogate keys.
• Mutator of the id should be private

IMPORTANT: leave updating of id to Hibernate!
Hibernate Objects

• decide on a business key for the object and implement the equals and hashCode
• When adding type specific constructors leave the id field null
• Don’t make the class final. Hibernate uses lazy loading for objects- before they actual execution a proxy mechanism is used

Do you know how to implement associations?
Use Set or HashMap for reference to Collections of object?
Sessions

The following method when applied to an object $O$, they will apply to any objects referenced by $O$

- `session.get` creates a new persistent obj by id from the db. It returns null if no such object in db
- `session.load` is similar except, if there was no such object in the db it throws exception.
- `session.delete` makes the database row corresponding to a persistent (or even a detached!) object to be deleted - the object becomes transient.
- `session.save` on a transient item will assign it an id and make it persistent: Similar to `INSERT`
Sessions

- `session.update` reattaching a detached object. Similar to SQL UPDATE
- `session.lock` reattaches the object (to the session) without checking or updating the database (assumes db and application are fully sync). Generally, do not use this method
- `session.saveOrUpdate` clear from name
- `session.merge` checks for a persistent object with the same identifier in the session. If it finds one, it copies the data from the detached object onto the persistent one, Otherwise it creates a new persistent object from the data
- Check the API if you are not sure!
Querying

If we know the object id:

X x = z.getY().getX()

If we don’t know we need to execute a query.

1) use session.createQuery to return Query object.

2) HQL to write a query (HQL similar to SQL)

3) Execute Query object by invoking list() or iterate() to return a list of results or an iterator over the results.
Example of HQL

session.createQuery("from customer cust
where cust.city=:cityName")
 .setString("cityName", "Birmingham")
 .list() ;

Returns a list of Customers.

• Method chaining
• select is optional. Can write “select fr...
• Customer cust instead of “?” in PreparedStatement in SQL
• from/where used for projection
What does the following do?

```java
Query query = createQuery("select cust, sa " +
"from Customer cust, SalesAgent sa " +
"where cust.city = sa.city");
```
So, you can write

```java
// get an iterator
ArrayList results = query.iterate();
while ( results.hasNext() ){
    Object[] row = (Object[]) results.next();
    Customer cust = (Customer) row[0];
    SalesAgent sa = (SalesAgent) row[1];
    // print the result
    System.out.format("Customer: %20s, Sales Agent: %20s\n", cust.getName(), sa.getName());
}
```
A little bit of HQL

You can use binary operations with exact semantics as SQL

=, <> , < , > , >=, <=, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL,

Binary operators for HQL collections
IS [NOT] EMPTY, [NOT] MEMBER [OF]

Following returns index of corresponding element from the collection
IS [NOT] EMPTY, [NOT] MEMBER [OF]

For further details see the manual
Pagination

Used when a query result in large number of rows- Google give only one page at a time
If a query object results in ordering of the rows we can use

• setFirstResult() and
• setMaxResults()
int pageSize = 10;
int pageNo = 2; // page 3
Query query =
createQuery("select cust, sa " +
   "from Customer cust, SalesAgent sa " +
   "where cust.city = sa.city" +
   "order by cust.name asc, sa.name asc" ); // order
query.setFirstResult(pageNo * pageSize);
query.setMaxResults(pageSize);
List customerSalesagentList = query.list();
when an object is made persistent, the objects it refers to are also made persistent. 

cascade lets us control how much automatically persisted, deleted etc.

Value for cascade are:

none: no automatic action on the referenced object takes place (default behaviour)
persist, merge, lock, evict, : Cascade any persist(), merge(), lock(), evict() operation across this relationship.
Cascading Persistence

Similarly

replicate, replace, save-update:
  Cascade any replicate(), replace() ... operation across this relationship.

delete: automatically removes the reference to the object(s) when delete() is called on the referencing object- note it is not deleted.
Cascading Persistence

delete-orphan: automatically delete any object for whom the reference has been removed from the referencing object. This option is only available for one-to-many and one-to-one relationships.

all: cascade all operations except delete-orphan.

all-delete-orphan: cascade all operations, and take the action of delete-orphan as well.

For further info see Hibernate manual
Recap

• Why ORM and Hibernate?
• A simple example to provide an overview
• Database patterns
• Log4j

At this point we look at hibernate in details:
• Object lifecycle in Hibernate
• Equality
• Session and its use
• Querying (some HQL!)
• Cascading persistence
• Transactions
• Mapping
DB interaction pattern

You may remember there are six main steps in a simple Hibernate program:

1. Configuration
2. SessionFactory
3. Session
4. Begin Transactions
5. Interact with the database
6. Commit
Delegate the synchronisation with DB to a Transaction object to commit or abort (rollback)

What if more than one transactions interact with the data?

Transaction

Application

Transaction1

Application

Transaction2
Different levels of locking

In Hibernate four level of locking considered

• Serializable
• Read Uncommitted (dirty read)
• Repeatable Read
• Read Committed (default with postgresql JDBC)
Serializable

Very resources intensive-no concurrency: one transition must have completely finishes (committed or rolled back) before the other had starts

For example a SELECT query sees only the data before transactions begins. It does not see the uncommitted data or changes committed by other concurrent transactions

Now, if other concurrent transaction commits, the transactions will rollback! Start again
Read Committed

Default used in PostgreSQL JDBC connection

One transaction can not see any value which has been written by another transaction if that other transaction has not yet committed.

Three concurrency problems:

• Lost Updates
• Unrepeatable Reads
• Phantom Read
Lost Updates

Transactions: tx1 tx2

read row A
Write to row A
Commit

read row A
write to row A
Commit

So value written by tx1 is lost, hence related transitions has not happened
Lost Updates

In effect the old data is lost. Versioning will solve this problem (two different version of the same row - with an identifier)
Unrepeatabile Reads

Reading is not repeatable!

**Transactions:** tx1 tx2

read row A

Write to row A

Commit

read row A

The second read produces different value

This can cause problem.

Caching and versioning can take care of this.
Phantom Read

Transactions: tx1  tx2

SELECT sth

DELETE a row

Commit

Same SELECT

tx1 gets a different result!

This is a problem as there is no versioning info left to work with
Phantom Read

Common problem with pagination
Run a SELECT and retrieve the first batch
Another transaction deletes a row from the second batch
We end up with the second set of result wrong— you never see the deleted result.
Be aware of this— although you can’t do much!
Dirty read

Suppose a two transactions
1\textsuperscript{st}: book flight and hotel to Birmingham- say opodo
1.1 Book flight to BHX
1.2 Book hotel
2\textsuperscript{nd}: book car at BHX (local car hire)- not opodo

Scenario:
Flight is booked (1.1 commit)- so car is hired (2 commit)
No hotel is available- so 1.1 compensated (not Rollback)
Hence a car hired for a customer who will not arrive. Not much can be done- manual checking!?!
Long transactions

1) break the long transaction up into two (or more) database transactions
2) use detached objects from the first transaction to carry the necessary information to the presentation layer
3) Reattach the object to do necessary updates (as objects get modified by others)
Long transactions

Transaction management is a complex subject outside the scope of the course, but:

Safe solution:

1. read from the database in the early transactions,
2. allow database writes in the last database transaction of a long transaction,
3. use *versioning* to check if your writes still make sense.
Versioning

Persistent object:
Add a version instance variable of type `int` or `long` to the objects.
Don’t forget accessor and mutator
Type can also be `TimeStamp` or `Calendar`
Advantage: you can see the time of the version
Disadvantage: timing problem (close updates will be treated as identical versions)
Versioning

Mapping file XX.bhm.xml:

Update mapping file:
If the version attribute is called “version” add this after <id>

<version name="version"
column="version">

Your database will have one extra column for versioning.
Versioning

Hibernate will take care of the versioning: whenever you make an object dirty in memory, Hibernate will update its version (in memory).

Hibernate throws StaleObjectStateException if the version number of the object on disk is not the same as it was when the object was loaded.
Versioning

Catch the exception and do something about the conflict

For example, report back to the user that the choice he/she has just made is, in fact, no longer available and could they please make another one
Mapping-overall view

Objects to be persisted

Mapping files
X.hbm.xml, Y.hbm.xml
...

Runtime

Hibernate

Database

Configuration related
hibernate.properties
log4j.properties
ehcachexml ...
Hibernate objects

- **Entity beans**: objects with persistent identity
  - (identifier field managed by Hibernate).
  - Example: User, Customer, Order etc.

- **Value beans**: objects which only exist in relationship to an entity bean.
  - Example: Address, CreditCardDetails etc.
Hibernate objects

• For each entity beans sth.java create a mapping file sth.hbm.xml to describe the mapping

• Put sth.bhm.xml in the same directory as sth.class

• The connection between any value beans related to sth.java is described in the mapping file of the corresponding entity bean, i.e. sth.hbm.xml
Simple Entity Classes without Relationships

Consider a class `User` with three attributes:
- `dateOfBirth`: `java.util.Date`
- `Username`: `String`
- `Gender`: `String` // or enum

We add `id` and `version` and all setters and getters to create an entity bean.
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="a.b.c">
<class name="User" table="user" lazy="true">
  <id name="id" column="id" type="long">
    <generator class="sequence"/>
  </id>
  <version name="version" column="version"/>
</class>
</hibernate-mapping>
Simple Entity Classes without Relationships

```
<property name="dateOfBirth" column="dob" type="date"/>
<property name="username" not-null="true" unique="true"/>
<property name="gender"/>
</class>

</hibernate-mapping>
```
Mapping Value Objects within Entities

Consider the following:
What does this mean?
What is the black diamond shape?
What does the arrow mean?
Mapping Value Objects within Entities

Technically not different from Person class with an attribute Address

So we can include in `Person.hbm.xml`

```xml
<property name="address"
    type="string"/>
```

But what if Address has its attributes such as street, postcode...?

*Use components*
Mapping Value Objects within Entities

Instead of a property in Person.hbm.xml include a components with its own property items

```xml
<component name="address" class="Address">
    <property name="street" column="user_street"/>
    <property name="postcode" column="user_postcode"/>
</component>
```
Mapping Value Objects within Entities

1) child objects are wholly owned by their parents: you cannot have two different parents.

2) A null child property is represented in the database by setting to null all the fields corresponding to the child object. Thus loading such a row will result in a parent object with a null child property, not a parent object with a child object whose properties are all null.

3) Not only can you have multiple components in a class, but one can have multiple components of the same (child) class in a class: such HomeAddress and WorkAddress
Mapping Simple Entity Classes without Relationships

What does this mean?

Implement the following and save a few Customer and Account objects. You need to create java files and bhm.xml files.

For solution see part01.zip
Mapping Simple Entity Classes without Relationships

Account object might be accessed by various applications, Modify Account.java and Account.hbm.xml

Look at both files

Solution:
   part01_version_only_only_account_files.zip
Mapping Value Objects within Entities

What does this mean?
Exercise

Implement this model and test it by saving a Customer object and its two addresses.

[No sample code for this one- it is straightforward]
Mapping inheritance

What does this mean?

[Diagram showing inheritance relationship between Address, HomeAddress, and workAddress classes with attributes street, postcode, houseNo, and officeNo]
Mapping inheritance

Use a single table
Each row of the table can hold an object of any type from the hierarchy.
One additional column called discriminator to distinguish between different types in hierarchy
You don’t need to add a new field called discriminator- it is for Hibernate book keeping
Another solution: delegate
Mapping inheritance

```xml
<class name="Address" table="addresses"
discriminator-value="only-address">
  <id name="id" column="id" type="long">
    <generator class="sequence"/>
  </id>
  <discriminator column="subclass" type="string"/>
  <property name="streetName" column="street" type="string"/>
  <property name="postcode"/>

  <subclass name="HomeAddress"
    discriminator-value="home-address">
    <property name="houseNo"/>
  </subclass>
</class>

... the rest
Mapping inheritance

Be careful with using “not null”
For a sample see part3.zip
Many-to-One, Unidirectional Associations

What does this mean?
What is the meaning of the arrow?
Many-to-One, Unidirectional Associations

Implement this model and write a small program to save the following students and lecturers

Student s1 = new Student("Jim", 123345);
Student s2 = new Student("John", 4321);
Lecturer l1 = new Lecturer("Achim");
s1.setSupervisedBy(l1);
s2.setSupervisedBy(l1);

Solution: see part4.zip
We will examine the code now.
Many-to-One, Unidirectional Associations

```xml
<hibernate-mapping>
    <class name="Student" table="students" lazy="false">
        <id name="id" column="student_id">
            <generator class="sequence"/>
        </id>
        <property name="studentName" /> 
        <property name="studentID" />
        <many-to-one name="supervisedBy" cascade="save-update" column="supervisor"/>
    </class>
</hibernate-mapping>
```
Many-to-One, Unidirectional Associations

```xml
<hibernate-mapping>
  <class name="Lecturer" table="lecturers"
    lazy="false">
    <id name="id" column="lecturer_id">
      <generator class="sequence"/>
    </id>
    <property name="lecturersName" column="lecturers_name" />
  </class>
</hibernate-mapping>
```
cascade="save-update"

If no value assigned to `cascade`, changes will not updated. For example change to the supervisor

- Restricting only the behaviour of `delete()`, `load()`... does not seem applicable
- `save-update` is convenient as it will reattach the Lecturer object if a lecturer object gets detached.
Modify the Main method of the previous example to add a new student who is supervised by a lecturer other than Achim, say Uday.

Student s3 = new Student("Sue", 5467);
Lecturer l2 = new Lecturer("Uday");
s3.setSupervidedBy(l2);

Then write a new transaction to update the new student to be supervised by Achim
What to do with 0..1 and *?

specify unique="true" as an attribute of the many-to-one element to have multiplicity * (no zero)
specifying not-null="true" to end up with multiplicity "1" instead of "0..1"

You can also take care of this by writing Assertions or using other constraint related libraries (OCL) - very much out of the scope of the course
One-to-Many, Unidirectional Associations

What does this mean?
Equality

You may have different Account, so let us first deal with equality.

Exercise:
Take Customer and Account in part01. Add a new field email to Customer and a new field IBAN (here int) to Account and then implement equals and hasCode to produce Business Key.

What are the implications of your decisions?
See part5.zip for sample solution
Implementation of association via a set

Exercise:
Create a new field `Accounts` to `Customer` to keep the list of accounts as a `Set`

```java
private Set Accounts = new HashSet();
```

Create a method `AddToAccounts (Account acc)` for adding to the `Set` a new `Account`
<class name="Customer" table="customers" lazy="true">
    <id name="id" column="customer_id">
        <generator class="sequence" />
    </id>
    <property name="customerName" column="customer_name" />
    <property name="age" />
    <property name="emailAddress" />
    <set name="Accounts" cascade="save-update" lazy="true">
        <key column="customer_id" />
        <one-to-many class="Account" />
    </set>
</class>
Discussion on one-to-many

Using Set

Set does not allow repetition of objects, i.e. we can’t have two of the same object in as Set.

Solution:

Use other implementations of List. HashMap() is suitable, so is LinkedList()...

There are over 30 implementation of associations in Java

Exercise: re-implement the previous example with HashMap().
Discussion on one-to-many

One-to-many is inefficient:

```java
Customer cust = new Customer("bxb" ...);

cust.getHasAccount().add(new Account(50, 1234));
cust.getHasAccount().add(new Account(10, 4321));

session.save(cust);
```
Discussion on one-to-many

This indeed will save Customer and two account.

**But how about saving the value of the association?**

Hibernate needs to do extra update statements to adjust the value of the association.

**Good news:**

- almost always you can implement bidirectional association **Many-to-one**
- From conceptual point of view we rarely need to insist on mono-directional (password!, which are handled differently)
Many-to-One, bidirectional Association

What does it mean?
Implement the following and write a small example to save some Customer and Account objects.
For a sample solution see part6.zip
Many-to-One, bidirectional Association

A new field must be set in the “one” end
inverse="true"

Which signals Hibernate that the same foreign key column is used for both ends

```xml
<set name="Accounts" inverse="true" cascade="save-update" lazy="true">
    <key column="customer_id"/>
    <one-to-many class="Account"/>
</set>
```
Many-to-One, bidirectional Association

We need to add another field to the “many” end, Account to keep the customer

```java
private Customer customerWhoOwnsAccount;
```

Adjust the hbm.xml file for the “many” side. It is many-to-one from this side:

```xml
<many-to-one
    name="customerWhoOwnsAccount"
    column="customer_id" cascade="save-update"/>
```
Many-to-One, bidirectional Association

When adding an Account, we must remove any previous Customers associated with it

```java
public void addToAccounts(Account acc) {
    Customer anotherOwner = acc.getCustomerWhoOwnsAccount();
    // if another owner is assigned remove acc from it
    if (anotherOwner != this && anotherOwner != null) {
        anotherOwner.getAccounts().remove(acc);
    }
    acc.setCustomerWhoOwnsAccount(this);
    Accounts.add(acc);
}
```