Object-orientated DB and persistence

In this week, we will cover the following topics:

- *Object orientated approaches to databases.*
- *Brief relevance of certain object-orientated concepts (classes and objects).*
- *Critique of relational 'objects' or entities (object fragmentation).*
- *Persistence and maintenance of object-orientated code for client-side programming using JDBC within a Hibernate framework context.*

… and will result in the following learning outcomes

- *Understanding of the relevance of object-orientation and problems with database persistence.*
- *Appreciation of relational entities as 'objects' from a java developer's perspective.*
- *Understanding of how to access relational databases without direct use of embedded queries, but by using objects.*

Table of Contents

10.1 Object orientated concepts ch 2 paterson

10.1.1 Objects

Object-oriented programming is an approach to computer programming based on the concept of an 'object'. An object, like an entity, is made up of separate data types. These types, like entities, can be primitive (integer, floating point, string etc.). However, unlike entities, objects can also be made up of other objects, which, in turn, are typically constituted from various primitive data types and other objects. Therefore, the ability to store objects inside other objects, in addition to having primitive datatypes, is for our purposes a key distinction between an *object* and what we have been referring to as an *entity*.

Furthermore, while entities are entirely made up of data, and are the *target* of programming code, objects that contain data and other objects also have *methods*, which provide objects with *behavior*. As such, objects are explicitly defined by using computer programming code typically implemented using object-orientated computer programming languages (C++, Java, C# etc.). Entities, on the other hand, exist inside a relational database, which is to be accessed via queries, either directly using MySQL, for example, or by embedding MySQL within an a programming language. Relational database entities are therefore relatively straightforward when compared with objects, which can take on arbitrary complexity.

Object-orientated languages therefore support the development of more complexity. To understand how object-orientated languages provide this support, we need to understand classes.

10.1.2 Classes

Classes provide a kind of general blueprint. Using this blueprint, objects can be created in computer memory, which are 'decorated' with specialized field values and behaviors, and exist independently in memory from other objects created using the same class. This is illustrated in Figure 1.

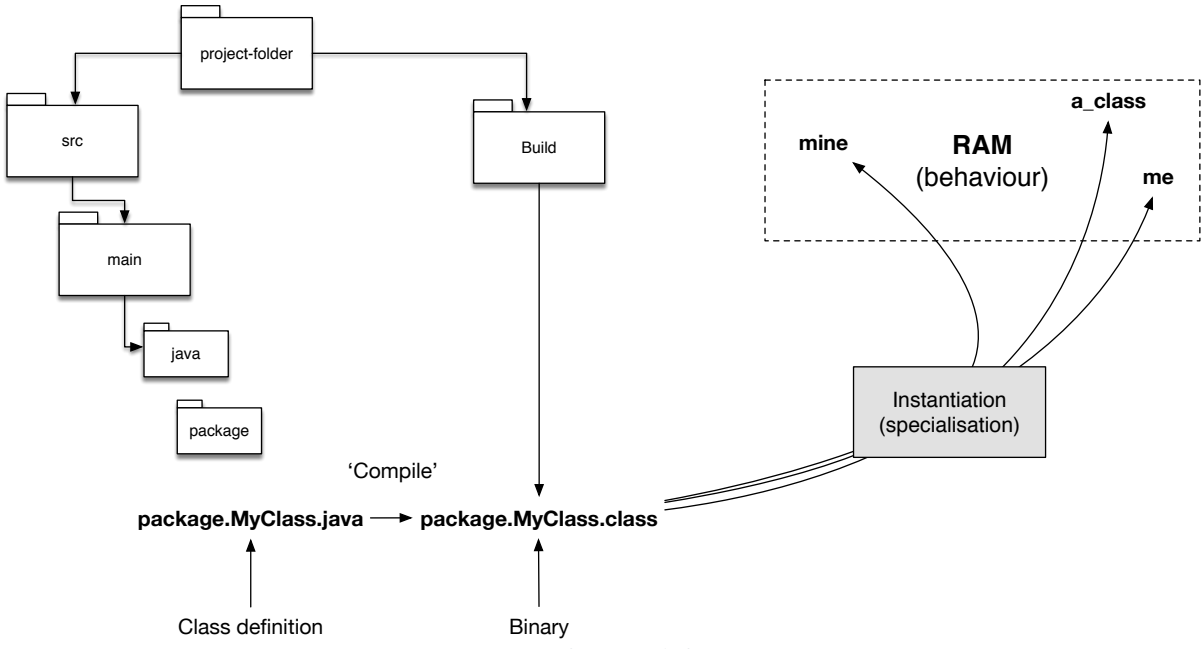

Figure 1: classes and objects

Within your java project, and within a specific package, you write the blueprint, the class definition (e.g., MyClass.java) in the .java file. Then, when this compiles it produces another file, which is simply a binary equivalent (a machine-readable version) of the .java version. When a program is launched, the MyClass representation can then be instantiated, which is the point at which the general definition of the class is specialized, depending on the data and object values. This produces separate classes in memory, which may have unique behaviors, being based on different field values. Notice the difference with entities. Entities have a number properties, which are unique to that entity. While this is a requirement for an entity, such that they can be uniquely accessed, the uniqueness of a class is not related to its data, but rather is related to its unique place in memory. In fact, instances of classes are *typically* specialized, but this is by no means a requirement.

There are quite a few technical points here, but from a pragmatic point of view the key points is that relational database entities are quite different from object orientated software objects. We will return to some the differences below

10.1.3 Inheritance

A key feature object orientated design is that of the inheritance. In everyday usage, inheritance means that the child of a given parent will inherit certain features; the child of a parent with 'brown-eyed' genes is also likely to have brown eyes; parents with dark hair are likely to have offspring with dark hair etc.

In object-orientated software design, the notion of inheritance is similar. The parent class is often referred to as the 'superclass'. A 'subclass' can then inherit properties (field data/objects etc.) from the superclass. This capacity is often depicted in UML, as indicated in Figure 2.

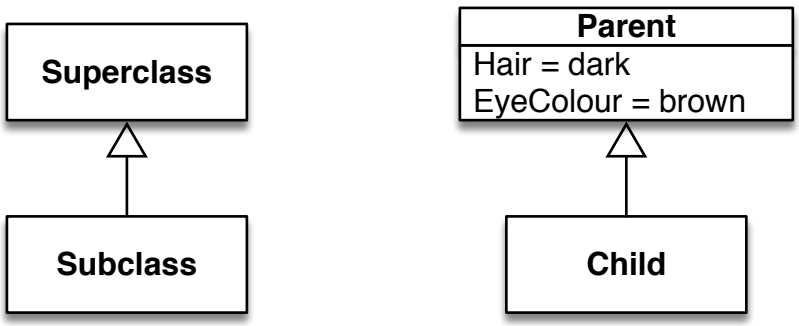

Figure 2: Inheritance diagram.

10.2 Object oriented databases

With these basic definitions concerning object-orientation in place, we can draw on these to further think about how relational databases differ from object-orientated databases.

As we noted earlier on in the course, relational databases are by far the most commonly used database paradigm. We have explained that relational databases are used to store attributes about given entities. A key feature of the relational approach, and the SQL languages that support it, is that the units of storage are primitive datatypes. For example, integer numbers, floating point numbers, strings etc. are used to declare the values of the attributes that

constitute entities. While it is true that different kinds of data types have been added to implementations of SQL, over time, the basic point is that the units of storage are relatively simple.

By way of illustration, let us take the perspective of a publisher. They are interested in storing different types of publication in their database, and we will consider 'Journal articles' and 'books' as examples. For a journal article entity, we will design a table to store appropriate attributes; *author, title, journal, year, volume, number, pages, month*. Likewise, books have the following attributes; *author, editor, title, publisher, year, volume, series, address, in addition*.

Therefore, for different kinds of publications, breaking up the entities/objects in question seems a natural thing to do. However, the data model itself harbors is a high degree of specificity in terms of the entities, when in fact it is possible to exploit some of the generality implied in the data. In other words, each entity, article and book, share common attributes. These common attributes are; author, title, volume.

The original relational database design of the tables is in one sense quite *verbose*; the common attributes are represented more than once in the schema. From a design point of view, this feels like overelaboration, which should be simplified. Of course, the relational approach does not support very well the need to represent different entities with a minimal schema. In other words, relational database designs are quite poor at exploiting the generality available within a given data domain.

In the previous section, we introduce some object-oriented concepts which we will now exploit to represent the data more effectively. Specifically, the object orientated capacity for inheritance can be used to simplify design by exploiting generalities *available* in the data.
To do this, we introduce a level of abstraction above our existing publication types of *article* and *book*. We will call this additional class a 'Publication' type. The publication type will specify the common (*author*, *title*, *year*, *volume*) fields. The classes article and book will inherit these fields from the publication class while, at the same time, specifying additional features relevant according to the subclass; remember, the classes 'article' and 'book' *subclass* the 'publication' class, which can therefore be considered the superclass. Using UML inheritance diagrams, we can easily visualize these relationships, as presented in Figure 3 and contrast this approach to the relational one.

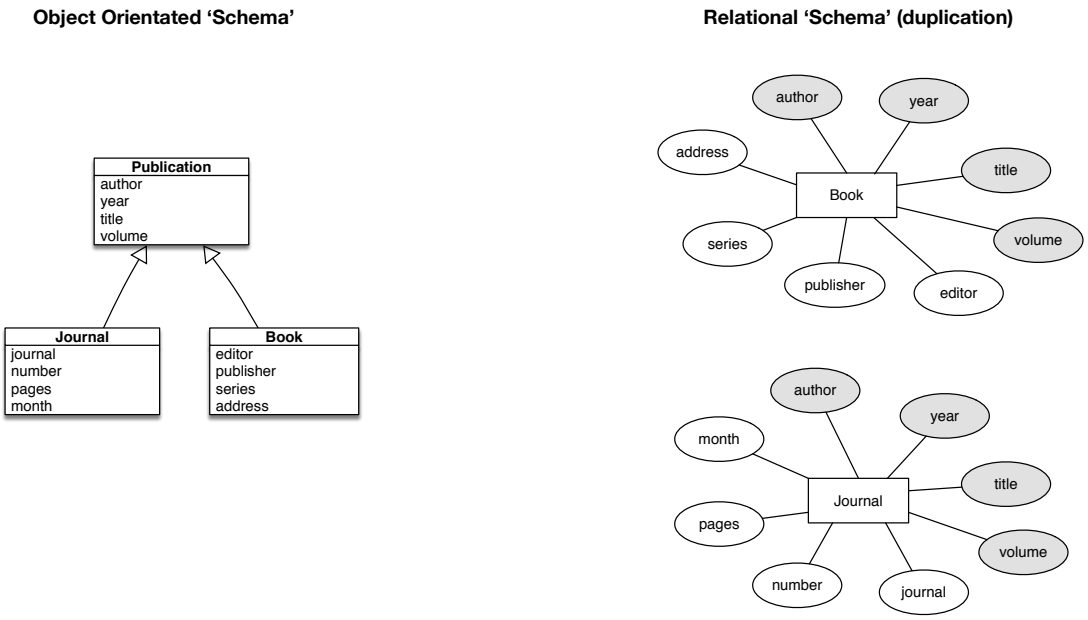

Figure 3: Publication class types v associated relational form.

In example just given, the object orientated schema contains fewer explicit fields; from this point of view it is simpler. As we have discussed previously in the course, thinking of ways to simplify the schema can be highly advantageous. You might argue that the relational approach can simplify the number of fields by placing them in additional tables. However, this requires the creation of these tables and the cross-linking between tables. Generally, when the domain data and the related model become highly complex, object orientated databases become more desirable.

10.3 Fragmentation of objects into relational entities

Furthermore, object-oriented databases are not only useful from the point of view of simplifying design. Object orientated databases are inherently more capable of storing complex objects, compared with relational databases. Relational databases must break up object complexity into primitive components whereas o-o databases can store objects 'as a whole'.

Four example, let us presume that we have an object in the real-world such as the cat. The aim is to store this object within a database. Object-oriented databases are, as they 'say on the tin', orientated towards objects. As such, the cat object could be stored in its entirety within the database as a single object. On the other hand, in a relational database, to access the 'whole cat' we would either store all its features within a single attribute, which is not a good idea, or access each separate attribute at the same time to return all the information concerning the given cat. As we have seen previously in the course, storing the entire cat within a single field is an unworkable solution in the context of relational databases; this would render the task of querying the cat's individual features impossible. In Figure 4, we present the view of the cat from object-orientated and relational points of view.

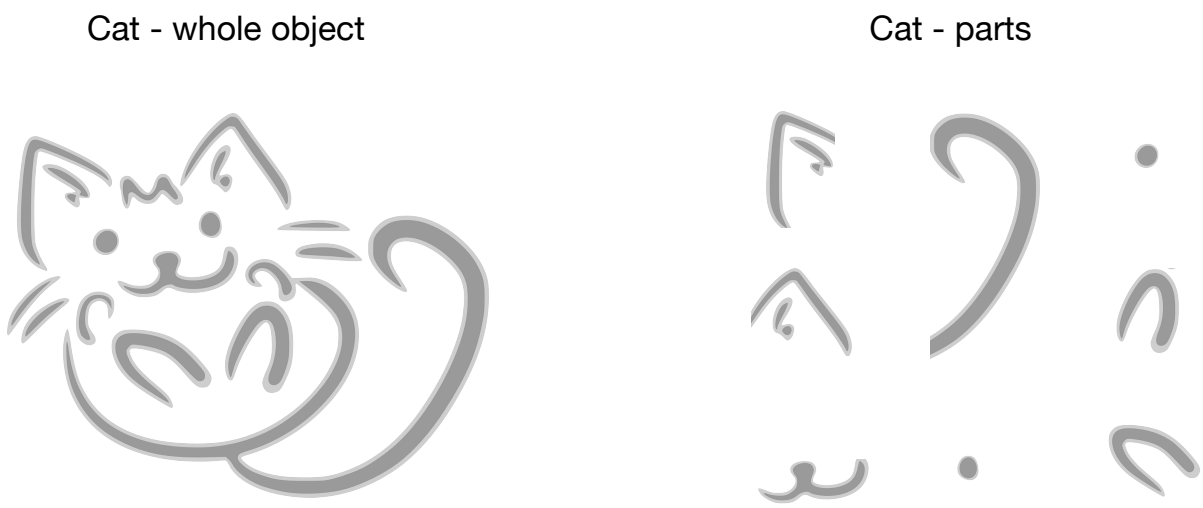

Figure 4: the whole cat and the fragmented cat

The key point here is that when working with object orientated languages, such as Java, the relational database model does not provide the best functionality for programmers. In fact, it is quite awkward. Many client-server applications use a relational database to store data and an object orientated programming language to develop the application. Consequently, to enable the persistence of objects between application launches, objects created during runtime need to be mapped to tables in the said database, then tables need to be mapped back to the objects when the application is relaunched. As we discovered, when data objects accessed this involves the embedding one language (SQL) within another (Java).

Generally, the whole process is considered quite disjointed from the point of view of an object orientated programmer, and the frustration that has led to the development of object-orientated databases is nicely captured by Paterson:

> *"Object orientation models and encapsulates the entities in a domain and the relationships between them, while relational databases aim to make data independent of the applications that use them, thereby minimizing duplication of data and providing flexibility in accessing that data…*
>
> *As a result, saving the state of a single object may involve splitting the values of its attributes between two or more relational tables. The difference between the object and relational views of data is often known as the impedance mismatch. It's a bit like taking a square peg, chopping it up, and fitting the little pieces into a bunch of round holes."*
>
> (Paterson, 2006) – The Definitive Guide to db4o

Continuing with the cat example, there is another sense in which the object might be further fragmented when represented within relational database. For example, each cat has fur, which might be represented as a single attribute within a Cat table. We might, for instance, record the colour of the fur *within* this attribute. However, fur can have other attributes associated with it (length, softness etc.). In a relational database, these additional properties

might be stored in a separate table named Cat_Fur, and the original fur attribute might now link to the Cat_Fur table. When querying the database from the client application, we would, under such circumstances, need to construct more complicated queries, like joining tables together. As the domain becomes more complicated, the joining of tables may become cumbersome programmatically, not to mention computationally expensive. For these reasons, while it is *possible* to develop client-side applications, and use our relational database to capture all the attributes, it is not necessarily desirable.

Alternatively, the idea behind an object orientated database is to avoid such fragmentation of objects and allow objects to be stored 'inside' other objects using pointers. In this way, the entire cat can be queried before specific information about the cat is used in the client side program. This reduces the tedium of having to record and reconstruct objects from the fragmented parts/attributes (see Figure 4).

However, relational databases remain useful for:

- Storing straightforward, primitive datatypes. Many applications still require interaction with very simple data, not necessarily complicated objects, and in such situations, the relational database remains a strong choice. Where a data-driven application relies on such data, there would be little point in using object-oriented database. So, even though object orientated databases merged from the concern that relational databases working adequate, this is only true incident situations.

and…

- From a purely pragmatic point of view, because object-oriented databases our more recent developments, compared to relational databases, there are fewer standards. Therefore, when one type of object-oriented databases adopted for development, it is more likely that the code gets 'tied' to the specific application programming interface. Unfortunately, with fewer standards changing between different API can result in a large degree of code rewriting, which costs time and therefore money.

- Object orientated databases are not underpinned by any formal mathematical logic, unlike relational databases. Object orientated databases do not therefore benefit from the power of mathematical analysis.

Therefore, as is often the case in software engineering, the choice over which is kind of database to use very much depends on the context… there is no absolute 'right' or 'wrong' database choice.

10.4 Persistence using hibernate

Hibernate uses XML so, and you will be learning how to implement mappings from java objects to relational storage. So, llet us start by briefly reminding ourselves about what XML is.

10.4.1 What is XML

Before answering this question, it is useful to ask what the ML part of XML stands for. It stands for mark-up language. The 'X' stands for 'extensible'.

When you begin to learn mark-up language, a good way to start is by looking at some basic examples that are visual. One such mark-up language is Latex, a typesetting mark-up language which is interpreted/compiled into a visual rendering, not to a browser but to a document type, like .pdf. So, let us look at a basic document in Figure 5.

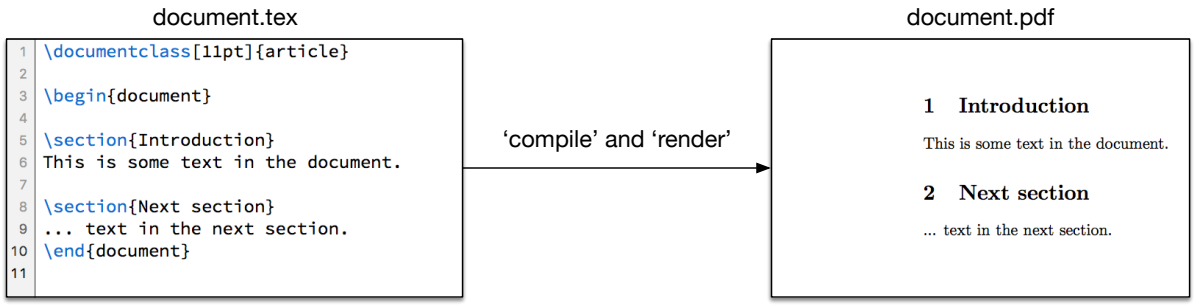

Figure 5: Basic latex mark-up.

In this example, we use the Latex mark-up language to typeset a document. The text is marked up by the commands **(\begin \end, \section** etc.). When such a document contains valid sysntax it can be compiled by a latext program, such as pdflatex, which creates the document.pdf.

The same essential process happens when we can an index.html file. In Figure 6, we present a very basic file, which is rendered in a browswer as illustrated.

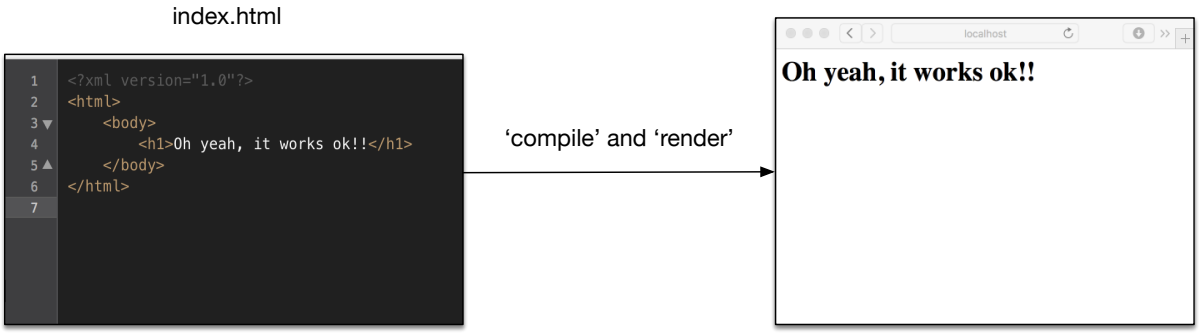

Figure 6: Basic html mark-up

10.4.2 Hibernate object-relational mappings using XML

Now we have briefly seen what mark-up language is, using examples related to the mark-up of text for visual rendering in two separate cases, we will go on to look at an example of XML usage that is specifically relevant to our course, especially in relation to some of the observations made in this week's materials (object-orientation v relational), and when we connected to a MySQL relational database using the o-o programming language of Java, in week TODO. Criticisms we have made about relational databases are that, from the point of view of an object-orientated application, they:

- tend towards the fragmentation of objects, and;
- can make the client-side code dependent on the use of specific SQL syntax.

"is an object-relational mapping framework for the Java language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions."

Wikkipedia

In other words, object orientated developers, e.g., Java programmers, like to work with objects, or in java-speak plain old java objects (**POJOs**). We will now briefly cover the *Hibernate* framework and explain how Hibernate allows for the removal of SQL syntax from java code, by mapping from POJOs to database entities/tables.

Note that, although Hibernate is not an object orientated database, it allows us to work-around some of the disadvantages of accessing relational databases directly (without a mapping) from an object orientated language like Java. Therefore, *Hibernate* is quite a popular choice for Java development teams who work in an object-orientated paradigm.

The idea of Hibernate is to provide functionality for intelligent handling of the interface between client-side code and the storage of object data. When a program is closed, the states of objects are still written to the database in the same way, but how this is achieved is more sensible from a Java developer point of view; querying of data is more straightforward, code is more decoupled from changes in the database and is handled using XML mappings and configurations. There are other advantages to using Hibernate, but we will focus on these.

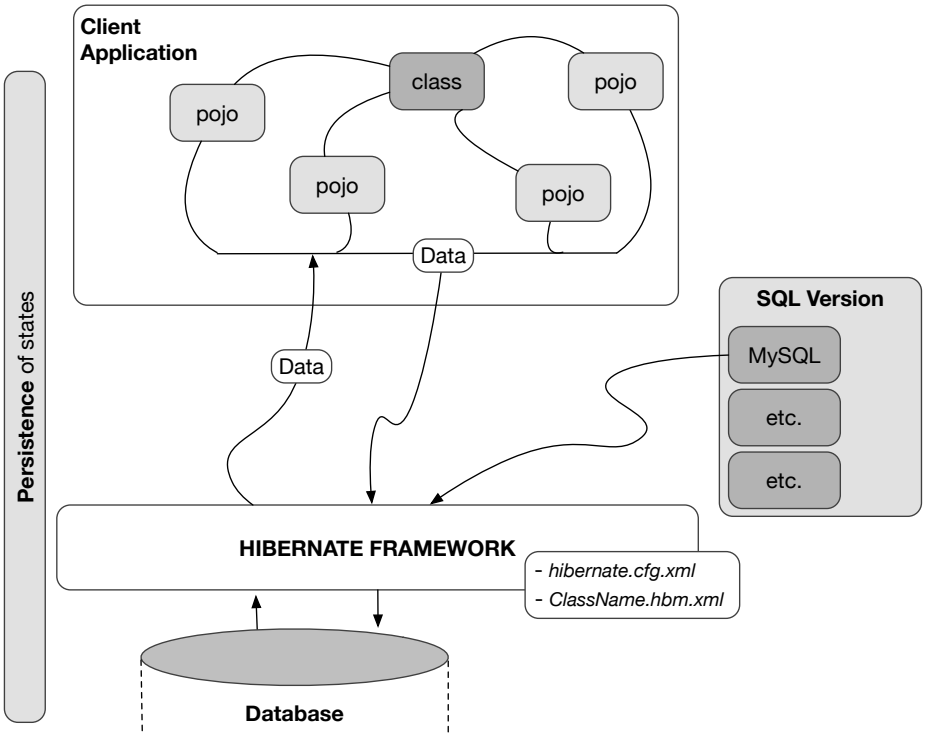

Figure 7: Client-hibernate-db architecture

An illustrative view of the overall client-sql-hibernate-database architecture is presented in Figure 7. When a client-side program is executed, its objects are created, say for the first time. When the program closes, the objects' data needs to be recorded, Data can be stored by querying the database through the Hibernate framework, depending on the connector being used, and the design of the objects in question. These aspects are defined in the Hibernate configuration and object mapping files.

- Hibernate configuration: we will see, in our practical sessions how to configure:
  - the SQL **dialect**.
  - the SQL **driver** (e.g., JDBC type).
  - The connection **url, username and password.**
  - Console **sql** output options.
  - Mapping resources, i.e., Hibernate mapping files.

- Hibernate mapping: these are defined inside mapping files which:
  - Have the file name format <**ClassName.hbm.xml**>. For example, the class declaration in <Customer.java> will be mapped inside a mapping file named <Customer.hbm.xml>.

The following xml file would be stored inside your java project with the name <**Person.hbm.xml**>. Take a look at the xml content:

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

```
<hibernate-mapping>
   <class name="Person" table="PERSON">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="name" column="FirstName" type="string"/>    </class>
</hibernate-mapping>
```

Importantly, Hibernate expects that the related java class will be a POJO with getters and setters. Therefore, the xml, above, would be used to allow Hibernate to communicate with the database as if communicating with a POJO declared in Person.java:

```
public class Person {

       private int id;

       private String name;

       public Person() {}

       public int getId() {
              return id;
       }

       public void setId(int id) {
              this.id = id;
       }

       public String getName() {
              return name;
       }

       public void setName(String name) {
              this.name = name;
       }
}
```

For example. In order, to create a person in our database through java, without any SQL commands, we could use code in the following form

```
try {
       tx = session.beginTransaction();
       Employee employee = new Employee("John");
       employeeID = (Integer) session.save(employee);
       tx.commit();
}
```

.. and we will see some more involved examples of this in the practical sessions.

10.4.3 Summary

You should take away the following points:

- xml does not do anything by itself other than structure data inside tags. This can then be used for different applications (html formatting for browser rendering) or, for containing data on specific configurations and mapping definitions within a framework like Hibernate.

- xml is therefore used in all kinds of applications and you should become familiar with reading it.

- Hibernate allows us to side-step some of the complications of communicating with databases in order to achieve persistence for objects in-between application runs.

- Basic knowledge required to developing with Hibernate includes:

    - Familiarity with the configuration file.
    - Familiarity with mapping files and;
    - Familiarity with the POJO constrains

10.5 Summary

- There are 'issues' of accessing databases from within an object-orientated perspective.

- Java programmers use object-orientated principles. Embedded queries make object-orientated development quite messy and, to a typical java developer.

- Hibernate is a framework that allows embedded queries to be removed, and object-orientated methods of database interaction possible, and removing of the problem of 'fragmentation', which has been referred by previous commentators on these issues.

- Hibernate, like many API's exploit the use of XML.

- Hibernate specifies mappings between POJOs and RDB's to help the development, flexibility and maintenance of codebases.

~~~~~
~~~~~