Data Persistence Frameworks:

Introduction to Hibernate

BIO

Norman Klein is an independent software consultant with over sixteen years' experience in the Unix/J2EE environment. In previous careers, he has worked for Convex Computer, Siemens, Netscape, and Magnet Interactive. He was the web architect behind Jane's Defense Weekly, DuPont StainMaster and First Franklin's "EasyWriter" Automated Mortgage Underwriting system. Recently has started work with BrixLogic.

Email: norman_klein@sbcglobal.net

Hibernate Presentation Sections

- Overview of Data Persistence Frameworks
- Hibernate Feature Set
- Hibernate Example Code

Data Persistence Framework Overview

- JDBC
- EJB (Entity Beans)
- JDO – Java Data Objects
- ORM – Object-oriented Databases

Serialization

- Java has a built-in persistence mechanism: *serialization* which provides the ability to write a graph of objects to a byte stream, which may then be persisted to a file or database.

Serialization

Disadvantages

A serialized graph of interconnected objects can only be accessed as a whole; it is impossible to retrieve any data from the stream without deserializing the entire stream.It is not possible to access or update a single object independently.

JDBC

Advantages:

- Proven Technology
- Fastest Execution Speed
- Simple to Program
- Access Proprietary Database Features
- No App Container Required

JDBC

Disadvantages:

- 30% of Java Application spent on JDBC
- All CRUD functions need to be coded
- JDBC code is non-portable (Requires a DAO)
- **No support for Inheritance/Polymorphism of objects**

EJB (Entity Beans)

Advantages:

- Standardized API
- Proven Technology
- Tool-Support from Major Vendors
- Provides Features in Addition to Data Persistence

EJB (Entity Beans)

Disadvantages:

- Entity Beans requires Façade Session Beans
- Verbose (requires many files to implement)
- EJB QL is Static
- No CMP/BMP Finder translations
- No support for Inheritance/Polymorphism

EJB

The goals of the initial EJB specification team were ambitious—to create a persistent model with complete location transparency. A model could be deployed in one container or within many containers across distributed servers.

Unfortunately, the original designs of this model completely ignored the overhead of distributed models and local interfaces.

Performance only becomes manageable when the restrictions that entity beans must be wrapped in a session bean façade and can only be deployed to the same machine as their façades (to reduce network overhead) are adhered to.

Entity Beans Require Façade Session Beans

Issues to be Concerned about:

local interfaces vs remote interfaces (passing by value vs reference)
Network costs incurred on every field access (fine vs coarse grained)
Transaction control (fine vs coarse grained)
Security control (fine vs coarse grained)

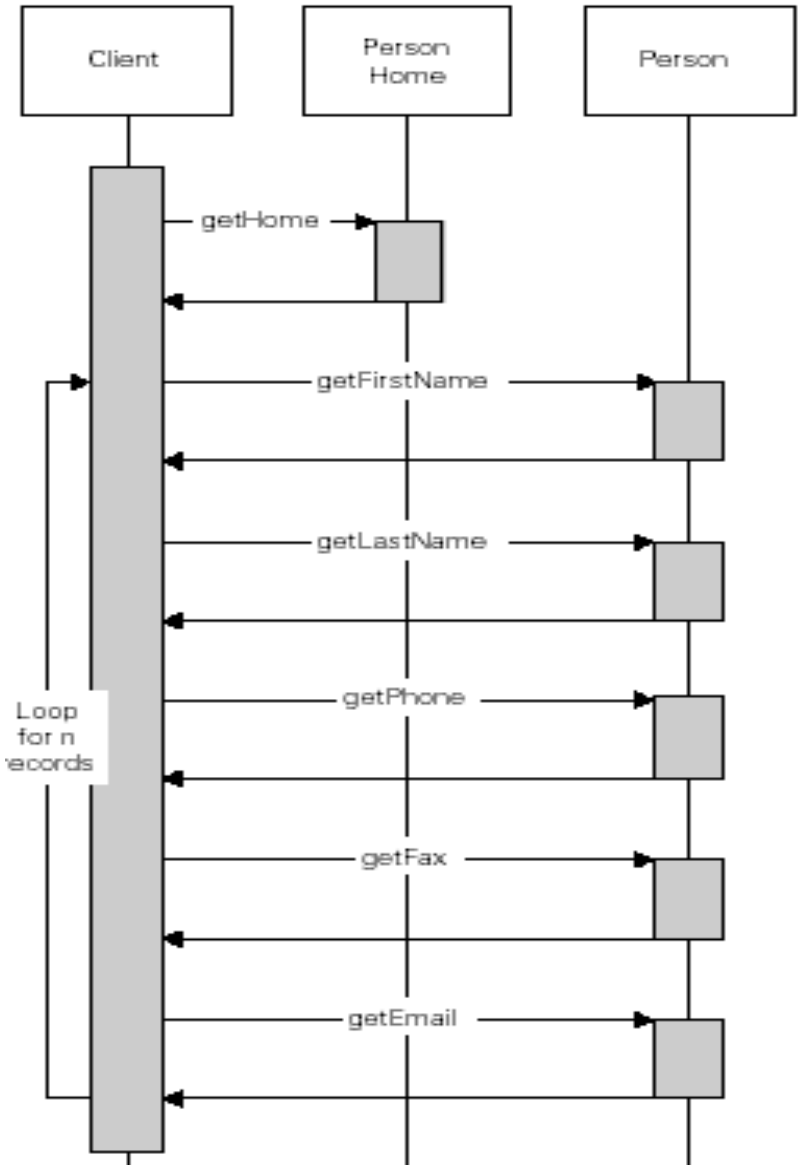
Solution:

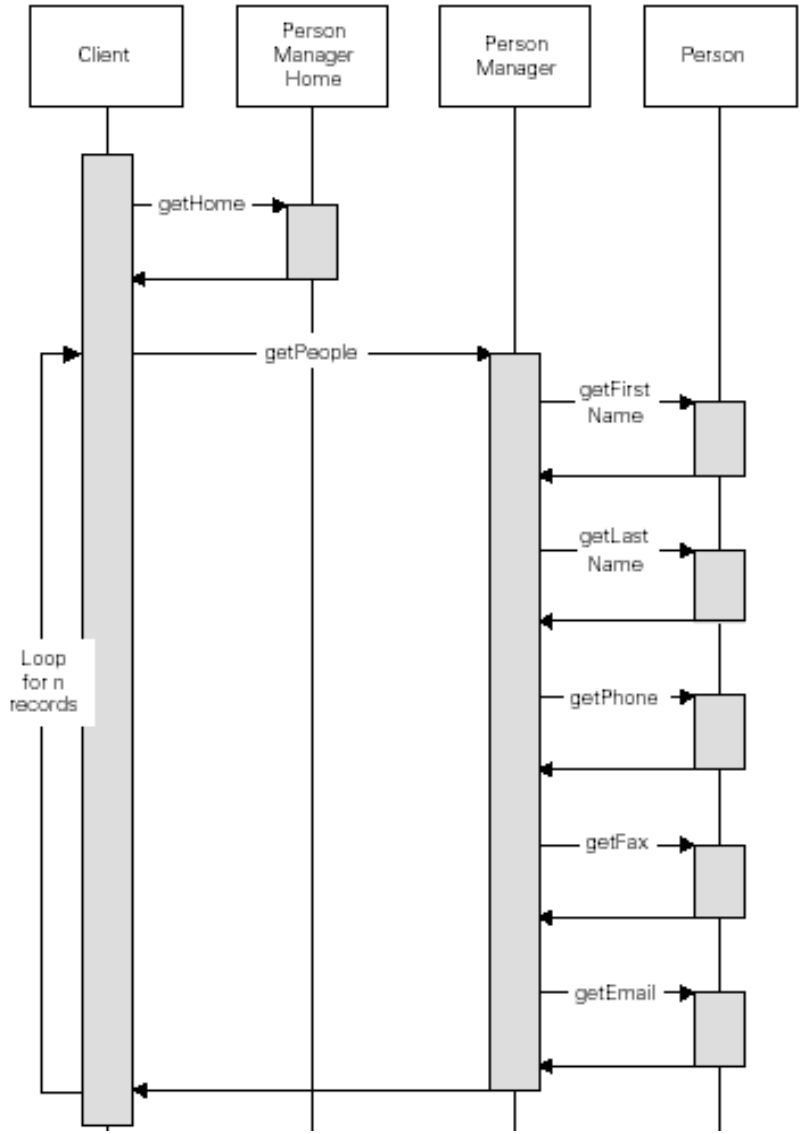
Use a Façade Session Bean Interface

Table 2.4 Many EJB services typically go to waste. The problem is that the EJB container provides coarse-grained services, but persistence needs to be a fine-grained service. Persistence, security, and distributed declarative transactions do not belong at the fine-grained level.

Service	Granularity
Declarative, distributed transactions	Coarse
Security and authentication	Coarse
Distributed object access • failover • scalability	Coarse
Persistence	Fine

Need for a Façade Session Bean





EJB -- Verbose

10 files for an EJB solution

2 files for an JDBC solution (client and code)

- Must run within an app container

Classification	File name
Entity bean	LogEntry.java
	LogEntryBean.java
	LogEntryHome.java
Session bean	Logger.java
	LoggerBean.java
	LoggerHome.java
Sample client code	TestLogger.java
Deployment descriptor	ejb-jar.xml
Vendor-specific deployment configuration	weblogic-cmp-rdbms-jar.xml
	weblogic-ejb-jar.xml

EJB QL Statements are Static

EJB QL is a query language defined in the EJB 2.0 specification for performing queries on container-managed persistence entity beans. It is an object-based query language that looks much like a subset of SQL.

But

EJB QL queries must be defined at deploy time within a deployment descriptor. Therefore, queries **cannot** be dynamically generated based on user input or other dynamic state. Although EJB QL queries can be parameterized at runtime.

Ex.

```
<ejb-ql>
    <![CDATA[SELECT OBJECT(a) FROM ProductBean AS pb
                        WHERE name = ?1]]>
</ejb-ql>
```

BMP/CMP Finder Limitations

CMP supports mapping of domain objects to a data store. In CMP, the mapping is specified in deployment descriptor of the entity bean. The EJB compiler generates a concrete class definition from the abstract CMP bean defined by the developer.

EJB QL is defined within the scope of CMP entity beans. So you **cannot** use EJB QL to find entity beans defined using BMP.

No Support for Inheritance/Polymorphism

Inheritance is the capability of an object-oriented programming language for one class to specialize or extend a base class.

- the CMP design paradigm does not allow classes to be inherited.

Polymorphism is the capability of an object-oriented programming language for a subclass to override a method's behavior in one of its superclasses.

- The relationship to other domain objects (beans) cannot be *polymorphic*

Example – No Inheritance

Writing an entity bean that extends a parent entity bean is possible. However, when executing an entity bean finder method, the container returns only objects of the same class as the entity bean's object interface.

Ex.

The domain object model comprises three classes:

Boat, Kayak, and Canoe.

Where both ***Kayak*** and ***Canoe*** extend ***Boat***.

But

You can't define a ***findAllBoats*** method that provides a list of **all** Boat objects in the database. The Boat entity bean cannot define a finder method that returns anything other than Boat entity beans.

Example -- No Polymorphism

Customer might have a SavingsAccount or CheckingAccount or both.

A **polymorphic** relationship would represent this as a single Customer.***getAccount***() method that returns either a SavingsAccount or CheckingAccount.

However, if these domain objects are modeled as CMP beans, then two explicit methods are required on CustomerBean:
getSavingsAccount() and ***getCheckingAccount***()

rather

than the single, polymorphic ***getAccount***()

Object-Oriented Databases

Vendors: Poet, Progress and Versant

Object-oriented database management systems (OODBMS) represent data in the database as lists of objects, rather than as tables containing rows and columns.

Applications: CAD and architecture

Object-Oriented Databases

Advantages:

- Full Support for Objects

Object-Oriented Databases

Disadvantages

- Unproven Technology
- Limited Market Acceptance
- Unknown Performance

JDO

Vendors: SolarMetric, LIBeLIS

JDO-based implementations typically run the compiled bytecode, of their domain object model through a tool that enhances the source or bytecode to add persistence.

JDO

Vendors: SolarMetric, LIBeLIS

Advantages:

- Standardized API
- Supports Inheritance and Polymorphism
- Advanced Query Functionality (JDOQL)
- Runs outside of an App Container

JDO

Disadvantages:

- Unproven Technology
- Requires byte-code modification
- JDO 2.0 specification still incomplete
- Currently Requires Proprietary Extensions

Object-Relational Mappers

- **Products**:

  Commercial: TopLink, CocoBase

  Open Source: Hibernate, iBATIS, Turbine

  A persistence layer where SQL is auto-generated from a metadata-based description which defines the mapping from persistent classes (objects) to relational database tables.

Object-Relational Mappers

Advantages:

- Familiar technology (JDBC)
- Comparable performance to hand-coded JDBC (less than 10% slower)
- Fully object-oriented: supports inheritance, polymorphism
- Supports Dynamic Queries
- Doesn’t require byte-code modification to work

Object-Relational Mappers

Disadvantages:

- No Standard API
- Further work needed to fully support stored procedures

Section II

Hibernate Features

Hibernate and Struts

You should think in terms of using Hibernate to persist your Business Model. Struts works with your Business Model, and your Business Model works with Hibernate, but Struts and Hibernate don't need to know that each other exists. (Layers pattern.)

-- Ted Husted (author of Struts in Action)

Struts Plugin for Hibernate

Add the following configuration for the Hibernate plugin to the struts-config.xml file

```
<plug-in className="edu.arbor.util.plugin.HibernatePlugIn">
      <set-property property="configFilePath" value="path-to-config-file" />
      <set-property property="storeInServletContext" value="true-or-false" />
</plug-in>
```

Compile the source file HibernatePlugIn.java and copy into an accessible library path.

Find complete directions and source files at http://hibernate.bluemars.net

How Hibernate Works

Rather than utilize bytecode processing or code generation, Hibernate uses runtime reflection to determine the persistent properties of a class.

The objects to be persisted are defined in a mapping document, which serves to describe the persistent fields and associations, as well as any subclasses or proxies of the persistent object.

Hibernate

A Hibernate application consists of 4 parts

1. Hibernate Properties File
2. Hibernate Mapping (XML) File
3. Hibernate Java Library
4. HQL (Hibernate Query Language)

and

1. Java Class Files
2. Database Schema

Message Class

```
package hello;

public class Message {

        private Long id;
        private String text;
        private Message nextMessage;

        private Message() {}                // Hibernate requires this

        public Message(String text) {
                this.text = text;
        }

        public Long getId() {
                return id;
        }

        private void setId(Long id) {
                this.id = id;
        }
```

Message Class (continued)

```
    public String getText() {
            return text;
    }

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

    public Message getNextMessage() {
            return nextMessage;
    }

    public void setNextMessage(Message nextMessage) {
            this.nextMessage = nextMessage;
    }
}
```

Database Schema

SQL*Plus: Release 9.2.0.1.0 - Production on Wed Aug 20 13:41:13 2003

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production

```
SQL> desc MESSAGES;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------
 ID                                        NOT NULL NUMBER(10)
 MESSAGE_TEXT                              NOT NULL VARCHAR2(80)
 NEXT_MESSAGE_ID                                    NUMBER(10)

SQL>
```

Hibernate Property File

```
# put in the root of the classpath
# or set System properties using  Dproperty=value

# Example configuration for Oracle
hibernate.dialect=net.sf.hibernate.dialect.OracleDialect
hibernate.connection.driver_class=oracle.jdbc.driver.OracleDriver
hibernate.connection.username=SCOTT
hibernate.connection.password=TIGER
hibernate.connection.url=jdbc:oracle:thin:@localhost:1521:test
hibernate.show_sql=true
```

Hibernate Mapping File

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sf.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
    <class name="hello.Message table="MESSAGES">
        <id name="id”  column="ID">
            <generator class="native"/>
        </id>
        <property name="text”  column="MESSAGE_TEXT"/>
        <many-to-one name="nextMessage
                                        column="NEXT_MESSAGE_ID"/>
    </class>
</hibernate-mapping>
```

Creating the Mapping File

Defining a class

- Class
- Id
- Properties
- Relationships

Class

```
<class
        name="ClassName"
        table="tableName"
        discriminator-value="discriminator_value"
        mutable="true|false"
        schema="owner"
        polymorphism="implicit|explicit"
        persister="PersisterClass" />
```

name: fully qualified Java class name of the persistent class (or interface).
table: name of its database table

Ex.

```
        class name="hello.Message" table="MESSAGES">
```

Id

Mapped classes *must* declare the primary key column of the database table.

```
<id
        name="propertyName"
        type="typename”
        column="column_name"
        unsaved-value="any|none|null|id_value">
        <generator class="generatorClass"/>
</id>
```

name: Name of the identifier property.
column: Name of the primary key column.

Ex.

```
        <id name="id”  column="MESSAGE_ID">
            <generator class=“sequence"/>
        </id>
```

Property

The <property> element declares a persistent, JavaBean style property of the class.

```
<property
    name="propertyName"
    column="column_name"
    type="typename"
    update="true|false"
    insert="true|false"
/>
```

name: Name of the property
column: Name of the mapped database table

Ex.

<property name="text" column="MESSAGE_TEXT"/>

Relationships

- one-to-one
- many-to-one
- one-to-many
- many-to-many
- components
- subclass
- joined-subclass
- set
- map
- composite-id

Many-to-one Mapping

Bar Foo.getBar() // returns corresponding Bar instance

```
<class name="Foo" table="foo">
    ...
    <many-to-one name="bar" column="bar_id"/>
</class>
```

<many-to-one name="nextMessage" column="NEXT_MESSAGE_ID"/>

A **many-to-one** reference is analogous to a simple Java reference. It is the same as the one-to-one situation except there is no requirement to have a shared PK. Instead a FK is used.

One-to-one Mapping

Java usage:

Bar Foo.getBar(); // Can only be one Bar

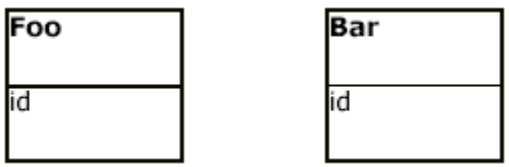
Mapping file:

```
<class name="Foo">
    ...
    <one-to-one name="bar" class="Bar" />

</class>
```

Schema:

Both Foo and Bar must share the same PK values to be part of a one-to-one assocation. If you create suitable instances of Foo and Bar with a shared PK, then retrieving a Foo will automatically retrieve the corresponding Bar.



One-to-many

Set Foo.getBar() // returns a collection of Bar instances

```
<class name="Foo" table="foo">
    ...
    <set role="bars" table="bar">
        <key column="foo_id"/>
        <one-to-many class="Bar"/>
    </set>
</class>
```

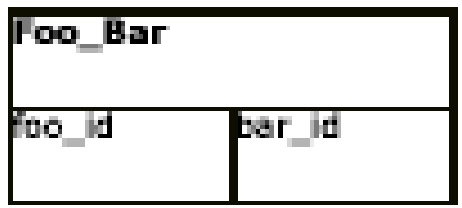
There is an extra column in Bar's table which holds the FK to Foo. This allows Foo to be assigned a collection of Bars based on the value of the foo_id in Bar.

Many-to-many Mapping

Set Foo.getBars() // returns a collection of Bar instances

```
<class name="Foo" table="foo">
    ...
    <set role="bars" table="foo_bar">
        <key column="foo_id"/>
        <many-to-many column="bar_id" class="Bar"/>
    </set>
</class>
```

We cannot have an extra column on Bar as that would dictate that each Bar has only one Foo. So we are forced to have an extra table, *foo_bar*, which holds the relationship between instances.



Initialization

```
// Looks for mapping files in same classpath  location as class
Configuration  cfg  = new Configuration().addClass(hello.Message.class);

// This binds you to a database configuration
SessionFactory  sessions = cfg.buildSessionFactory();

// obtain a JDBC connection and instantiate a new Session
Session sess = sessions.openSession();

// start a new transaction (optional)
Transaction  tx =  sess.beginTransaction();
```

Creating a Message

```
Session session = getSessionFactory().openSession();
Transaction trans = session.beginTransaction();

Message message = new Message("Hello World");
session.save(message);

trans.commit();
session.close();
```

Generated SQL:

```
insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (1, 'Hello World', null)
```

Retrieving a Message

```
Session newSession = getSessionFactory().openSession();
Transaction newTrans = newSession.beginTransaction();

List messages = newSession.find("from Message message order by message.text asc");
System.out.println( messages.size() + " message(s) found:" );

for ( Iterator iter = messages.iterator(); iter.hasNext(); ) {
      Message message = (Message) iter.next();
      System.out.println( message.getText() );
}

newTrans.commit();
session.close();
```

Generated SQL:

```
select message.MESSAGE_ID, message.MESSAGE_TEXT, message.NEXT_MESSAGE_ID
from MESSAGES message
order by message.MESSAGE_TEXT asc
```

Updating a Message

Session session = **getSessionFactory().openSession**();
Transaction trans = **session.beginTransaction**();

// 1 is the generated id of the first message
Message message = **session.load**(Message.class, new Long(1));
message.setText("Greetings Earthling");
message.setNextMessage(new Message("Take me to your leader (please)");

trans.commit();
session.close();

Generated SQL:

```
select message.MESSAGE_ID, message.MESSAGE_TEXT, message.NEXT_MESSAGE_ID
from MESSAGES message where message.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (2, 'Take me to your leader (please)', null)

update MESSAGES set MESSAGE_TEXT = 'Greetings Earthling', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```

Deleting a Message

Session session = **getSessionFactory().openSession**();
Transaction trans = **session.beginTransaction**();

Message message = **session.load**(Message.class, new Long(1));

// method 1 – deleting the loaded message
session.delete(message);

// method 2 – delete all Messages after the tenth message
session.delete("from player in class example.message where player.id > 10");

trans.commit();
session.close();

Generated SQL:

```
Delete from MESSAGES message where message.MESSAGE_ID = 1
```

Hibernate Query Language

HQL has significantly more features than EJB QL.

- from
- select
- where
- having
- ‘order by’ and ‘group by’
- Aggregate functions

HQL -- from

- **from ex.Cat**

  Which returns all instances of the class ex.Cat

- **from ex.Cat as cat**

  This allows you to use the alias in subsequent queries.
  This returns instances of all persistent classes that extend this class or implement the interface.

- **from java.lang.object obj**

  This returns all persistent objects

- **from ex.Apple as apple, ex.Bears as bears**

  This results in a cartesian product or “cross” join

HQL -- select

select cust.name from ex.Customer cust

This query selects the names of all customers.

select distinct cust.name from ex.Customer cust

This query selects the distinct names of all the customers

select cust.fname, cust.lname from ex.Customer cust

This query returns multiple objects as an array of type Object[]

HQL -- where

- **from ex.Cat cat where cat.id = 123**
  - Selects all instances of cat whose id is 123

- **from ex.Cat cat where cat.mate.name is not null**
  - This is a compound path expression, where the inner table join is automatically generated

- **from ex.Cat cat, Dog dog**

  where upper(cat.mate.name) like 'Tab%' and cat.friend.dog = dog

  Supports logical operations

HQL – Aggregate Functions

The supported aggregate functions include:

- avg(…), sum(…), min(…), max(…)
- count(*), count(distinct …)
- In, between, is null
- any, some, all, exists

HQL – ‘order by’ and ‘group by’

- **from ex.Cat cat order by cat.name asc, cat.weight desc, cat.birthdate**

  - asc indicates ascending order and desc indicates descending order

- **select cat.color, sum(cat.weight), count(cat)**
  from ex.Cat cat
  group by cat.color

  - A query that returns aggregate values may be grouped by any property of a returned class

HQL -- having

```
select cat from ex.Cat cat
      join cat.kittens kitten
group by cat
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

Notice that SQL functions such as aggregate functions are not allowed in the ***having*** and ***order by*** clauses, if this query is to be executed against MySQL.

HQL – Named Parameters

Query q = s.createQuery("from cat in class Cat where cat.name=**:name** and foo.size=**:size**");
q.setProperties(fooBean); // fooBean has getName() and getSize()

Contrast this with the use of '?' in JDBC statements with parameters

Hibernate Supported Databases

RDBMS support

- DB2 7.1, 7.2;
- MySQL 3.23;
- PostgreSQL 7.1.2, 7.2, 7.3;
- Oracle 8i, 9i;
- Sybase 12.5 (JConnect 5.5);
- Interbase 6.0.1 (Open Source) with Firebird
- HypersonicSQL 1.61, 1.7.0;
- Microsoft SQL Server 2000;
- Mckoi SQL 0.93
- Progress 9
- Pointbase Embedded 4.3
- SAP DB 7.3
- Informix
- Ingres
- FrontBase

Web Resources

Main Site

hibernate.org

User Forum

forum.hibernate.org

Mapping Diagrams

www.xylax.net/hibernate/

Section III

Hibernate Code Example

Polymorphism

```
package eg;

public class DomesticCat extends Cat {
        private String name;

        public String getName() {
                return name;
        }

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

Discriminator

The **<discriminator>** element is required for polymorphic persistence using the table-per-class-hierarchy mapping strategy and declares a discriminator column of the table.