Developing With Hibernate

- Primary Keys
- Inheritance Strategies

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Top Down vs Bottom Up

- In a **top-down** approach, you first write the object classes and generate database schema from the object model
- In a **bottom-up** approach, you generate object classes from database table definitions
- A **meet in the middle** approach combines top-down and bottom up

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is an Entity?

- In Hibernate, an **entity** is a class that has its own identity in the database
- Hibernate also supports **components**, which are considered parts of entities – components share the table of the owning entity

```
1    public class Person  // entity
2    {
3       private int id;
4       private String firstName;
5       private Address homeAddress;
6       . . .
7    }
8
9    public class Address  // component
10   {
11      private String street;
12      . . .
13   }
```

Hibernate's support for "components" lets you define fine-grained domain models. Components share the database table of their owner and have no individual identity.

Note that since Address doesn't have its own table, a given Address cannot be shared amongst several entities.

If a entity that defines a component and the component has no current value (e.g. a Person with no current Address), Hibernate uses NULLs in the Address's database columns.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Rules for Entities

- The class must be configured in a Hibernate configuration file
- The class must have a public or protected, zero-argument constructor
- The class must not be declared final
- If an entity will be passed through a session bean's remote business interface, the entity must implement the Serializable interface
- Persistent fields must be declared private, protected, or package-private, and can only be accessed directly by the entity class's methods

Clients of the entity must access the entity's state via get/set methods, but methods in the entity class can access fields directly.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping an Entity

- You can use the **class** element to configure an entity
- This element provides numerous configuration options, only a few of which are shown here:

```
<class
  name="ClassName"
  table="tableName"                          // default: ClassName
  discriminator-value="discriminator_value"  // default: ClassName
  mutable="true|false"                       // default: true
  . . .
/>
```

```
1    Example:
2
3    <hibernate-mapping . . .>
4       <class name="model.Person" table="PERSONNEL">
5         . . .
6       </class>
7    </hibernate-mapping>
```

2 - 5

See this Web page for details:

http://docs.jboss.org/hibernate/stable/core/reference/en/html/mapping.html#mapping-declaration-class

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping State

- Hibernate supports mapping either **fields** or JavaBean-style **properties** to database columns

```
<property
  name="propertyName"
  column="colName"          // default: propertyName
  type="typeName"           // default: matching Java-Hibernate type
  access="field|property    // default: property
  . . .
/>
```

```
1    Example:
2
3    <hibernate-mapping . . .>
4       <class name="model.Pet">
5         <property name="name" access="field"
6           type="string"/>
7          . . .
8       </class>
9    </hibernate-mapping>
2 - 6
```

Using property-based access is useful if the get/set methods have additional logic that you want Hibernate to execute when it accesses the field.

Developers have benchmarked and found little performance difference between field and property access. Hibernate uses Java reflection in either case.

If you use access="field, then the class doesn't need get/set methods

Note that the "type" you specify here is neither a Java or SQL type - instead it's a type defined by Hibernate.

If you have a field or property that you don't want stored in the database (i.e. a "transient" field), then simply omit a property mapping for that field.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Hibernate Types

- In the mapping file, for state, you specify **Hibernate types**, which are a bridge between Java and SQL types
- The following table lists some of the Hibernate types and how they map

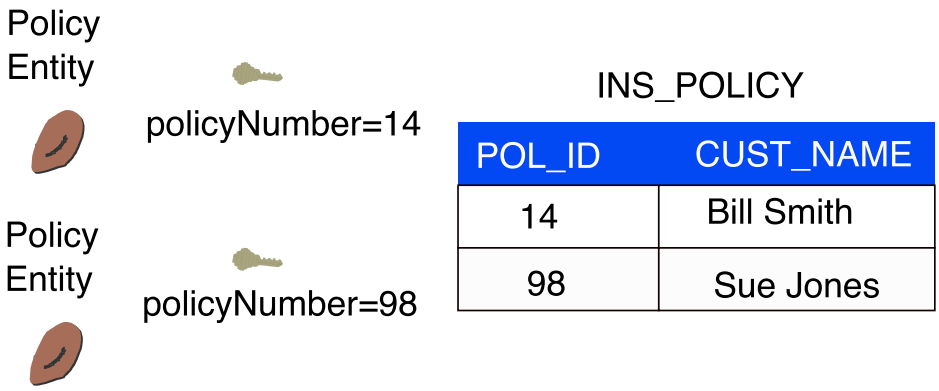
Java Type	Hibernate Type	Possible SQL Types
int	integer	INTEGER, NUMBER
double	double	DOUBLE, FLOAT
boolean	boolean, yes_no	BOOLEAN, SMALLINT
java.lang.String	string	VARCHAR, VARCHAR2
java.util.Date	date, timestamp	DATE
java.Math.BigDecimal	big_decimal	NUMERIC, NUMBER

For portability across databases, Hibernate mapping uses Hibernate types rather than specific SQL types. That way, you can change database dialects without (hopefully) modifying the mapping files.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

An Entity's Identity

- Each entity class must have some unique state so that objects can be differentiated
- Hibernate supports both **natural** and **synthetic** keys, but most applications use synthetic keys



A natural key is one that is part of the entity's actual data, while a synthetic key is meaningless except for its uniqueness. Hibernate developers tend to prefer synthetic keys.

Hibernate does support "composite keys", which are typically natural keys that are composed of multiple columns in the database.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring the Identity

- You use the **id** configuration element to mark a field or property as the primary key

```
<id
  name="propertyName"
  column="colName"           // default: propertyName
  type="typeName"            // default: matching Java-Hibernate type
  access="field|property     // default: property
  . . .
/>
```

```
1    Example:
2
3    <hibernate-mapping . . .>
4       <class name="model.Pet">
5         <id name="idNumber">
6           <generator class="increment"/>
7         </id>
8           . . .
9       </class>
10   </hibernate-mapping>
```

2 - 9

Hibernate supports various ways to generate synthetic keys.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Auto Generated Keys

- Many database scenarios involve keys that are generated either by the database itself or by the persistence provider
- Hibernate provides several generation strategies

Hibernate Generator	Description
native	Uses database's normal strategy
identity	Use IDENTITY column
sequence	Use database's sequence generator
increment	Hibernate generates an incremented ID
hilo	Uses Hi/Lo table generation strategy
uuid.hex	Hibernate generates a unique UUID

The "identity" strategy works well with DB2, Apache Derby, HypersonicSQL and other databases that provide IDENTITY syntax for a column when creating a table.

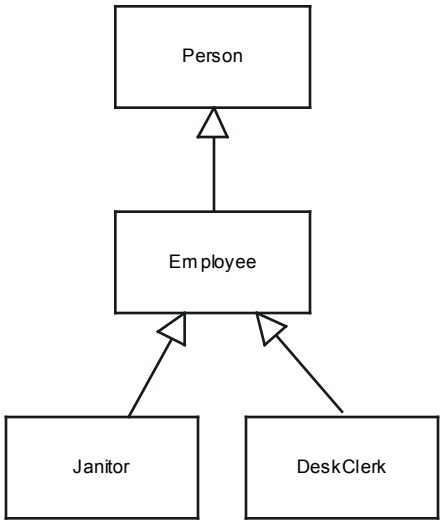
The "sequence" strategy works well with Oracle. You will also need to write a child element of <generator> to define the sequence table:

```
<generator class="sequence">
  <param name="sequence">MY_SEQ_TABLE</param>
</generator>
```

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Inheritance Overview

- Object models often exhibit inheritance and polymorphism to aid in code re-use and maintenance
- Relational databases generally don't directly support inheritance, leading to an O/R "impedance" mistmatch



There a couple of key benefits to using inheritance:

1. Inheritance lets you create models that closely approximate the real-world entity upon which the model is based. Lowering the so-called "representational gap" makes your software easier to understand and develop.

2. When combined with polymorphism (covered later), inheritance lets you create systems that are easy to extend without risking breakage to existing code.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Inheritance Strategies

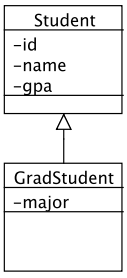
- One table per class hierarchy
- A table per concrete class
- A table per subclass (JOINED)

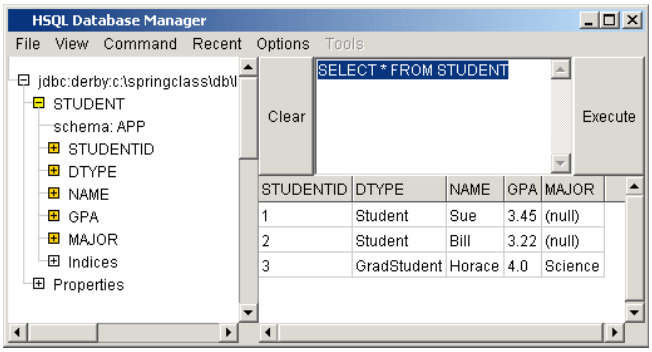
We will not cover the "table per concrete class" strategy since it's not used as often as the other two. It doesn't support polymorphic queries - i.e. queries that return objects of a superclass AND subclasses.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Single Table Inheritance

- In this strategy, a single table holds data for all classes in the hierarchy
- This is simple and fast, but isn't normalized and requires **nullable** columns
- The table must contain a "discriminator" column that indicates the object's type





This strategy results in tables that aren't normalized, since superclass objects will have "null" column entries, for example as shown here, a "Student" has no "major" (that's a property introduced by the GradStudent subclass).

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Single Table Inheritance

- To configure single-table inheritance, you configure the subclass(es) within the configuration of the superclass using the **subclass** element

```
1    <hibernate-mapping>
2      <class name="university.Student"
3        discriminator-value="Student">
4        <id name="studentID" column="STUDENTID" >
5          <generator class="identity" />
6        </id>
7        <discriminator column="DTYPE" type="string"/>
8        <property name="name" column="NAME"  />
9        <property name="gpa" column="gpa" />
10       <subclass name="university.GradStudent"
11         discriminator-value="GradStudent">
12         <property name="major"/>
13       </subclass>
14     </class>
15   </hibernate-mapping>
```

The "subclass" configuration element takes most of the same options as the "class" element.

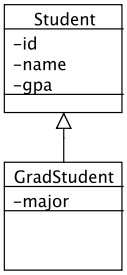
Note that it is possible to configure subclasses in their own separate configuration file and then use the "extends=" attribute on the "subclass" element.

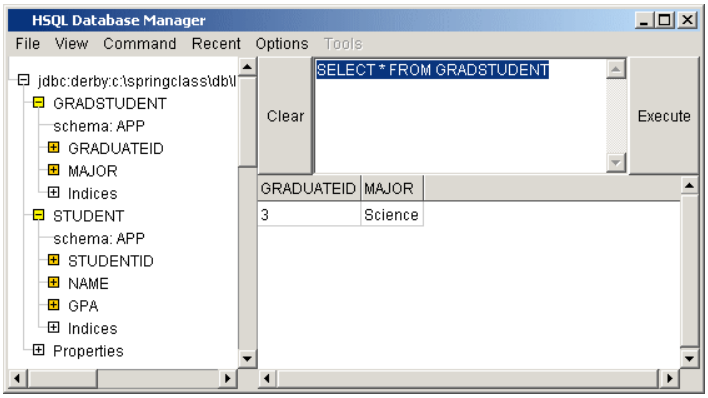
You can nest subclasses within other subclasses, too - there's no Hibernate restriction on how deep the class hierarchy can range.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Table Per Subclass Inheritance

- In this strategy, each entity has its own normalized table
- This strategy is not as fast as the single-table strategy, but allows for **not null** columns





This strategy is elegant, but is slower than the single-table strategy, since it requires that the persistence provider do a table join to retrieve all of the data for subclass objects.

Note that in the database screenshot, the "GRADUATEID" column in GRADSTUDENT is both a primary key in the GRADSTUDENT table and a foreign key into the STUDENT table.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring Table Per Subclass

- To configure table-per-subclass inheritance, you configure the subclass(es) within the configuration of the superclass using the **joined-subclass** element

```
1    <hibernate-mapping>
2      <class name="university.Student">
3        <id name="studentID" column="STUDENTID" >
4          <generator class="identity" />
5        </id>
6        <property name="name" column="NAME"  />
7        <property name="gpa" column="gpa" />
8        <joined-subclass name="university.GradStudent">
9          <key column="GRADUATEID"/>
10         <property name="major"/>
11       </joined-subclass>
12     </class>
13   </hibernate-mapping>
```

The "key" child element in "joined-subclass" is required, and configures BOTH a primary key in the subclass and a foreign key in the superclass's table.

Note that no discriminator column is needed in this approach.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Complete Example

Student.java

GradStudent.java

CreateStudents.java

Student.hbm.xml

hibernate.cfg.xml

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package university;
2
3    public class Student
4    {
5        private int studentID;
6        private String name;
7        private double gpa;
8
9        public double getGpa()
10       {
11           return gpa;
12       }
13
14       public void setGpa(double gpa)
15       {
16           this.gpa = gpa;
17       }
18
19       public String getName()
20       {
21           return name;
22       }
23
24       public void setName(String name)
25       {
26           this.name = name;
27       }
28
29       public int getStudentID()
30       {
31           return studentID;
32       }
33
34       private void setStudentID(int studentID)
35       {
36           this.studentID = studentID;
37       }
38
39       @Override
40       public String toString()
41       {
42           return "Name: " + name +
43                  " ID: " + studentID +
44                  " GPA: " + gpa;
45       }
```

```
46
47    }
```

```
1    package university;
2
3    public class GradStudent extends Student
4    {
5        private String major;
6
7        public String getMajor()
8        {
9            return major;
10       }
11
12       public void setMajor(String major)
13       {
14           this.major = major;
15       }
16
17
18   }
```

```
1    package client;
2
3    import org.hibernate.Session;
4    import org.hibernate.SessionFactory;
5    import org.hibernate.Transaction;
6    import org.hibernate.cfg.Configuration;
7
8    import university.GradStudent;
9    import university.Student;
10
11   public class CreateStudents
12   {
13
14       public static void main(String[] args)
15       {
16           SessionFactory factory =
17               new Configuration().configure().buildSessionFactory();
18           Session session = factory.openSession();
19           Transaction tx = session.beginTransaction();
20
21           Student s = new Student();
22           s.setGpa(3.45);
23           s.setName("Sue");
24           session.save(s);
25           System.out.println("New student's ID: " + s.getStudentID());
26
27           s = new Student();
28           s.setGpa(3.22);
29           s.setName("Bill");
30           session.save(s);
31           System.out.println("New student's ID: " + s.getStudentID());
32
33           GradStudent gs = new GradStudent();
34           gs.setGpa(4.00);
35           gs.setName("Horace");
36           gs.setMajor("Science");
37           session.save(gs);
38           System.out.println("New student's ID: " + gs.getStudentID());
39
40           tx.commit();
41           session.close();
42           factory.close();
43       }
44   }
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="university.Student" table="STUDENT">
        <id name="studentID" column="STUDENTID" >
            <generator class="identity" />
        </id>
        <property name="name" column="NAME"  />
        <property name="gpa" column="gpa" />
        <joined-subclass name="university.GradStudent">
          <key column="GRADUATEID"/>
          <property name="major"/>
        </joined-subclass>
    </class>
</hibernate-mapping>

<!--
<hibernate-mapping>
  <class name="university.Student"
    discriminator-value="Student">
    <id name="studentID" column="STUDENTID" >
      <generator class="identity" />
    </id>
    <discriminator column="DTYPE" type="string"/>
    <property name="name" column="NAME"  />
    <property name="gpa" column="gpa" />
    <subclass name="university.GradStudent"
      discriminator-value="GradStudent">
      <property name="major"/>
    </subclass>
  </class>
</hibernate-mapping>
 -->
```

```
1    <!DOCTYPE hibernate-configuration
2      SYSTEM "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
3    <hibernate-configuration>
4        <session-factory>
5            <property name="hibernate.connection.driver_class">
6                org.apache.derby.jdbc.EmbeddedDriver
7            </property>
8            <property name="hibernate.connection.url">
9                jdbc:derby:c:\springclass\db\InheritDB
10           </property>
11           <property name="hibernate.dialect">
12               org.hibernate.dialect.DerbyDialect
13           </property>
14           <property name="show_sql">true</property>
15           <property name="format_sql">true</property>
16           <property name="hibernate.hbm2ddl.auto">create</property>
17           <mapping resource="university/Student.hbm.xml" />
18       </session-factory>
19   </hibernate-configuration>
```

Chapter Summary

In this chapter, you learned:

- About primary keys and auto generation
- Different ways of implementing inheritance

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum