Developing With Java Persistence

- Annotations
- Primary Keys
- Inheritance Strategies

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

Java Persistence Annotations

- The persistence manager needs "metadata" about your objects and tables so it can manage their persistence
- One way to provide the metadata is to write **JDK 5 annotations**

Annotation	Description
@Column	Specifies a mapped column for a property or field
@Entity	Specifies that a class is an entity
@GenereratedValue	Provides for generation of primary keys
@Id	Specifies that a field or property is a prmary key
@Inheritance	Specifies inheritance strategy
@SequenceGenerator	Defines primary keys from a sequence table
@Table	Specifies primary table for an entity
@TableGenerator	Defines primary keys from a database table

There are many other persistence annotations - this figure shows only the ones we will cover in this chapter.

Instead of using annotations, you can instead write an XML file named orm.xml that provides the metadata.

Defininging an Entity

- The **@Entity** annotation designates a class as an entity so that a persistence manager can manage its persistent state

```
1    import javax.persistence.Entity;
2
3    @Entity
4    public class Policy implements Serializable
5    {
6    . . .
7    }
```


You can optionally provide a name for the entity using this annotation.

Rules for Entities

- The class must be annotated with the **javax.persistence.Entity** annotation
- 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 beans 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 classs methods

---

Actually, using the @Entity annotation is not required - alternatively, you can write an XML configuration file.

It's possible to use abstract classes as entities.

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

Mapping State

- Java Persistence supports mapping either **fields** or **JavaBean-style** properties to database columns
- You use the @ID or @Column annotations to indicate how you want to map

```
@Id
private int policyNumber;
@Column (name="CUST_NAME")
private String holderName;
. . .
```

```
private int policyNumber;
private String holderName;

@Id
public int getPolicyNumber() {}
public void setPolicyNumber(int n) {}

@Column (name="CUST_NAME")
public String getHolderName() {}
public void setHolderName(String s) {}
. . .
```

You shouldn't mix the two approaches -- either annotate fields or properties, but not both.

If you don't write an @Column annotation, the persistence manager assumes that the column name matches the property or field name.

The @Id annotation indicates that this field or property is part or all of the entity's primary key.

Mapping a Table's Columns

- You can use the @Table annotatation to define which table the objects of an entity class should map to
- You can use the @Column annotation to indicate the column name that maps an object's properties or fields

```
1    @Entity
2    @Table(name="INS_POLICY")
3    public class Policy implements Serializable
4    {
5      @Id
6      @Column (name="POL_ID)
7      private int policyNumber;
8      @Column (name="CUST_NAME")
9      private String holderName;
10     . . .
11   }
```

If you don't write a @Table annotation, the table's name must match the entity's name.

There is also a @SecondaryTable annotation that lets you split an entity's properties amongst more than one table.

Transient Fields

- By default, the persistence manager persists all fields or properties in the entity
- You can use the @Transient annotatation to indicate that a field or property should not be persisted

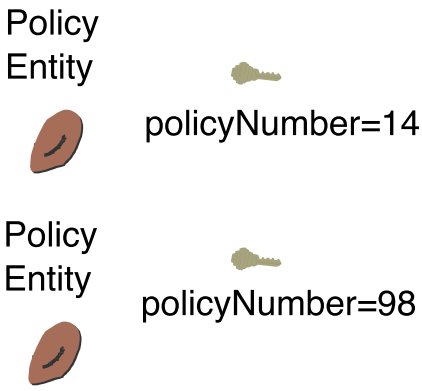
```
1    @Entity
2    @Table(name="INS_POLICY")
3    public class Policy implements Serializable
4    {
5      . . .
6      private String claimImageFileName;
7      @Transient
8      private ImageIcon claimImage;
9      . . .
10   }
```

Here we indicate that we don't want a image's binary data to be persisted since we do persist the file name from which we can load the image.

Since the persistence manager doesn't initialize transients, it's up to the entity to do so itself, typically by writing a method annotated with @PostLoad. The persistence provider calls such methods after the persistent portion of an entity is established.

The Primary Key

- Each entity class must have some unique state so that objects can be differentiated
- You use the @Id annotation to mark a field or property as part or all of the primary key
- Java Persistence also supports **composite keys** and separate **key classes**



INS_POLICY

POL_ID	CUST_NAME
14	Bill Smith
98	Sue Jones

Auto Generated Keys

- Many database scenarios involve keys that are generated either by the database itself or by the persistence provider
- You use the @GeneratedValue annotation to configure the generation strategy

```
public @interface GeneratedValue
{
  GenerationType strategy() default AUTO;
  String generator() default "";
}

public enum GenerationType
{
   TABLE, SEQUENCE, IDENTITY, AUTO
}
```

Here we show the annotation definition for the @GeneratedValue annotation.

Note that both the "strategy" and "generator" elements (parameters) are optional with default values.

If you use the TABLE or SEQUENCE generation types, then you must write a separate @TableGenerator or @SequenceGenerator annotation.

Auto and Identity Key Strategies

- The **AUTO** strategy indicates that the persistence provider should should choose the best way to generate keys
- The **IDENTITY** strategy indicates that the provider must use some sort of auto-increment or identity column in the database

```
1    @Entity
2    @Table(name="INS_POLICY")
3    public class Policy implements Serializable
4    {
5      @Id
6      @Column (name="POL_ID)
7      @GeneratedValue(strategy=GenerationType.AUTO)
8      private int policyNumber;
9      . . .
10   }
```

The AUTO strategy is common and flexible, especially in top-down scenarios, since it allows the persistence provider to choose the best key-generation technique supported by the database in use.

Table and Sequence Key Strategies

- To use the **SEQUENCE** strategy, the database itself must contain a **sequence table** from which the primary keys are fetched
- To use the **TABLE** strategy, the database itself must contain a separate table from which keys are obtained

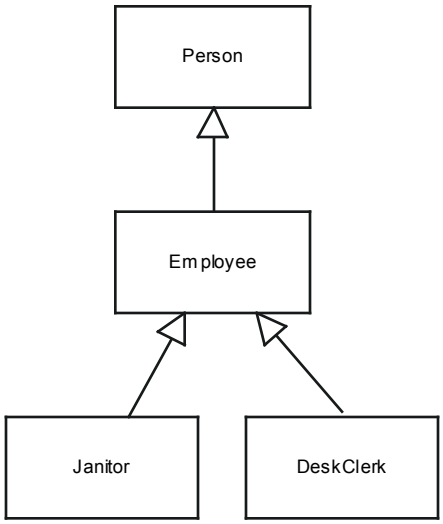
```
1    @Entity
2    @Table(name="INS_POLICY")
3    public class Policy implements Serializable
4    {
5      @Id
6      @Column (name="POL_ID)
7      @SequenceGenerator(name="policyGen",
8         sequenceName="POLICY_SEQ", allocationSize = 1)
9      @GeneratedValue(strategy=GenerationType.SEQUENCE,
10        generator="policyGen")
11     private int policyNumber;
12     . . .
13   }
```


If you choose either of these strategies, then you must write a separate @SequenceGenerator or @TableGenerator annotation. The SEQUENCE strategy is common for Oracle installations, which provide a fast and efficient sequence-table implementation.

If you choose the TABLE strategy, there must be a user-defined table in the database from which the keys are extracted.

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.

Inheritance Strategies

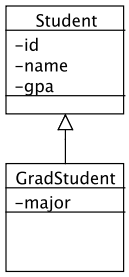
- One table per class hierarchy
- A table per concrete class
- A table per subclass (JOINED)

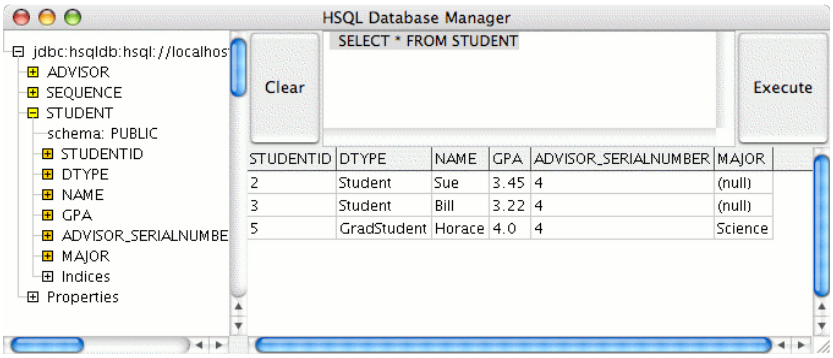
We will not cover the "table per concrete class" strategy since it's not used as often as the other two.

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

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Student implements Serializable
{
. . .
}
```



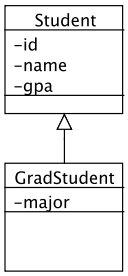


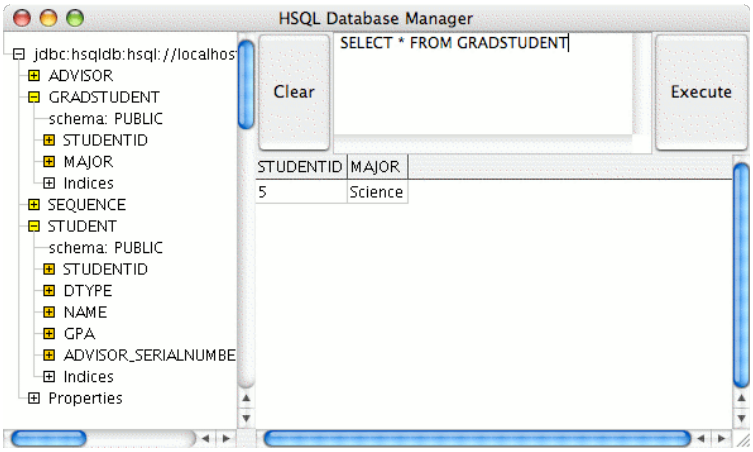
This strategy results in tables that aren't normalized, since superclass objects will have "null" column entries, for example as shown here, a "Student" has no "major" (that's a property introduced by the GradStudent subclass).

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

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Student implements Serializable
{
. . .
}
```





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 "STUDENTID" column in GRADSTUDENT is a foreign key into the STUDENT table.

Entity Lifecycle Callbacks

- You can annotate methods so that the persistence provider invokes them after certain lifecycle events

```
@PrePersist  - Called before entity saved
@PostPersist - Called after entity saved
@PreRemove   - Called before entity deleted
@PostRemove  - Called after entity deleted
@PreUpdate   - Called before entity is synchronized
@PostUpdate  - Called after entity is synchronized
@PostLoad    - Called after entity loaded into context
```

```
1    @Entity
2    public class Policy implements Serializable
3    {
4    . . .
5      @PostLoad
6      public void myLoad() {. . .}
7    }
```

You can put lifecycle callback methods either in the entity itself or in a separate class.

Callback methods in the entity class have the signature:

```
public void xxxx()
```

Callback methods in a separate class have the signature:

```
public void xxxx(Object o)  -- passed entity instance
```

Configuring the Persistence Provider

- To configure, you write a file named **persistence.xml**
- For standalone Java applications, this file must reside in a **META-INF** folder

```
<?xml version="1.0" encoding="windows-1252" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance". . .>
  <persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
   <provider>oracle.toplink.essentials.PersistenceProvider</provider>
   <class>university.Student</class>
   <class>university.Advisor</class>
   <class>university.GradStudent</class>
    <properties>
     <property name="toplink.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
     <property name="toplink.jdbc.url" value="jdbc:hsqldb:hsql://localhost"/>
     <property name="toplink.jdbc.user" value="sa"/>
     <property name="toplink.jdbc.password" value=""/>
     <property name="toplink.ddl-generation" value="drop-and-create-tables"/>
     <property name="toplink.target-database"
       value="oracle.toplink.essentials.platform.database.HSQLPlatform"/>
   </properties>
  </persistence-unit>
</persistence>
```


In a Java EE application, you don't need to explicitly name the entity classes, since the container will scan JPA JARs to discover them. The "class" elements are required for Java SE applications, however.

The properties are obviously provider-specific. Read your JPA provider's documentation for details.

Chapter Summary

In this chapter, you learned:

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