JPA and Hibernate

Mike Calvo

mike@citronellasoftware.com

What is JPA?

- Java Persistence API
- A specification for generic ORM
  - Not an implementation
  - Vendor-neutral
- Based on Annotations
  - Metadata
- Developed out of EJB 3 Specification
  - Replacement for train wreck EJB 2 persistence
  - Stand alone specification (does not require JEE container)

What is Hibernate?

- Very popular open source Java ORM tool
- Originally developed by Gavin King
- Now maintained by Redhat (JBoss Group)
- Implements JPA specification
- Predates JPA
  - Has its own custom API for persisting mapped objects

Choosing an API

- JPA Benefits
  - Standard
  - Vendor-neutral
  - Works with EJB 3
- Hibernate Benefits
  - More mature (fully developed)
  - Works with Java 1.4 and older
- My Policy
  - Use JPA when you can
  - Use Hibernate API when you need it

Enabling Hibernate & JPA with Maven

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate</artifactId>
   <version>3.2.5.ga</version>
 </dependency>

 <dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-annotations</artifactId>
   <version>3.3.0.ga</version>
 </dependency>

 <dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>3.3.1.ga</version>
 </dependency>
```

Configuring Mapping

- Hibernate legacy
  - XML configuration files
  - Each file mapped a class and it’s persistent fields
- JPA and Hibernate 3.2 and above
  - Annotations (javax.persistence package)
  - Class-level: marks a class as persistent
  - Method/File level: configure field and relationship persistence
- Hibernate still supports XML configuration

Core JPA Annotations

- @Entity
- @Id
- @GeneratedValue
- @Column
- @JoinColumn
- @OneToMany
- @ManyToOne
- @ManyToMany

JPA Annotation Rules

- Any persistent class must be annotated with @Entity (or inherit from a class that does)
- All persistent classes must have a field annotated by @Id signifying the primary key
- All instance fields in a persistent class are assumed to be mapped to columns with the same name as the field
  - @Transient will remove a field from mapping
- Relationships are not automatically mapped
  - Relationships are modeled as aggregate members
  - Require an @OneToMany, @ManyToOne, or @ManyToMany

Overriding Defaults

- Class annotation @Table defines specific table mappings
  - Name
- Field annotation @Column defines specific column mappings
  - Name
  - Nullability
  - Size
  - Uniqueness

Relationships

- @OneToMany
  - Annotate a member variable which is an @Entity annotated type
- @ManyToOne or @ManyToMany
  - Annotate a member variable which is a standard Java collection with parameter of type which is an @Entity
  - Type of collection is significant
- Use @JoinColumn to define specifics about the join column

Relationship Features

- Lazy Loading and Fetch
- Cascading
- Fetch
- Polymorphic

Lazy Loading

- Performance optimization
- Related items are not retrieved until they are first accessed
  - Field level access
- Limited to work only within the Session or EntityManager that loaded the parent object
  - Causes a big problem in web applications

Fetch

- Fetch mode
  - Lazy
  - Eager – disable lazy loading
- Mode can be configured on each relationship
- Consider performance and use when configuring fetch

Cascade

- Tells Hibernate whether to follow the path of the relationship on
  - Insert
  - Update
  - Delete
  - All
- Hibernate adds options
  - Delete All Orphans

Inheritance

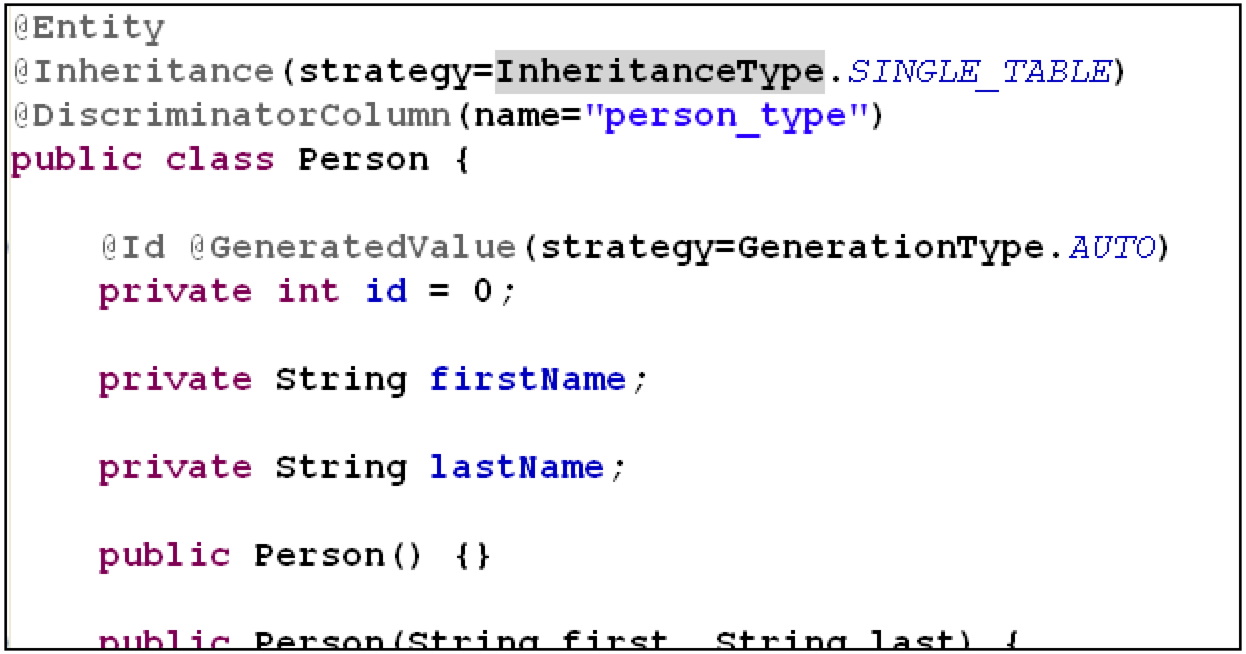
- JPA and Hibernate support mapping Class inheritance to relational tables
- Three approaches provided:
  - Table per class hierarchy
  - Table per sub class
  - Table per concrete class

Table per Class Hierarchy

- All of the data representing the properties of every class in the Hierarchy are stored in one table
- Requires that columns representing fields in subclasses allow null
- Uses discriminator column to determine the type of object represented a row
  - Each subclass must declare the discriminator value for the type

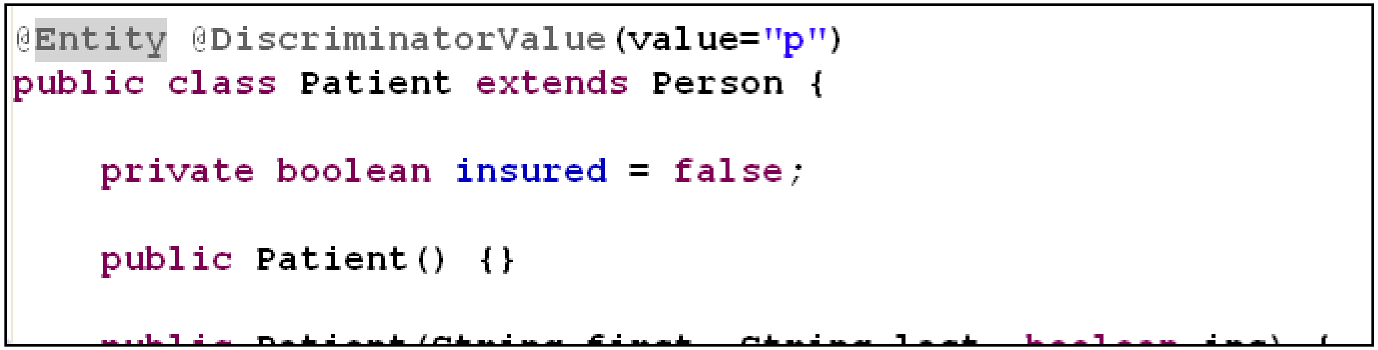
Table per Class Hierarchy Example

- Patient and Physician extend Person

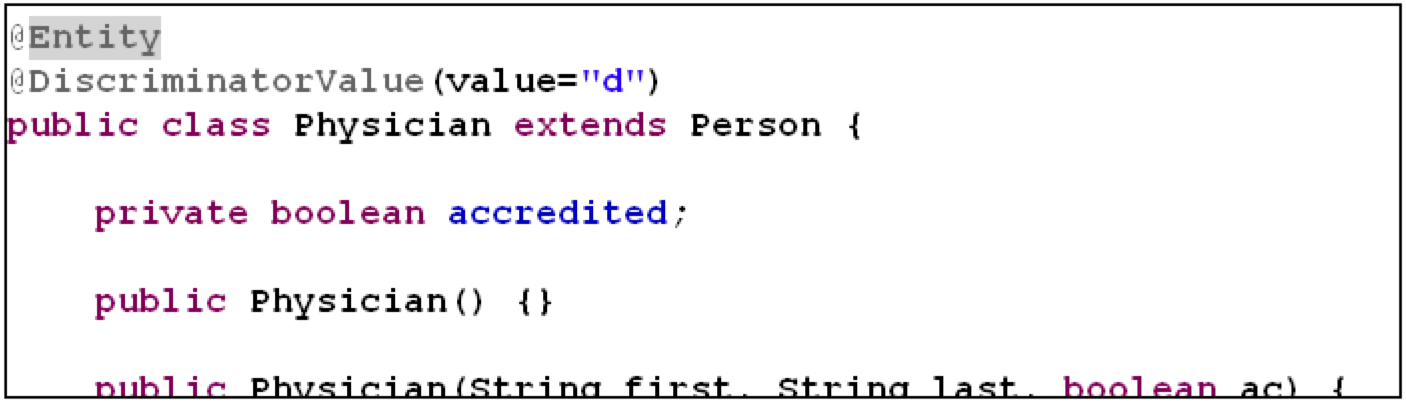


```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="person_type")
public class Person {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id = 0;

    private String firstName;

    private String lastName;

    public Person() {}

    public Person(String first, String last) {
```

Patient and Person

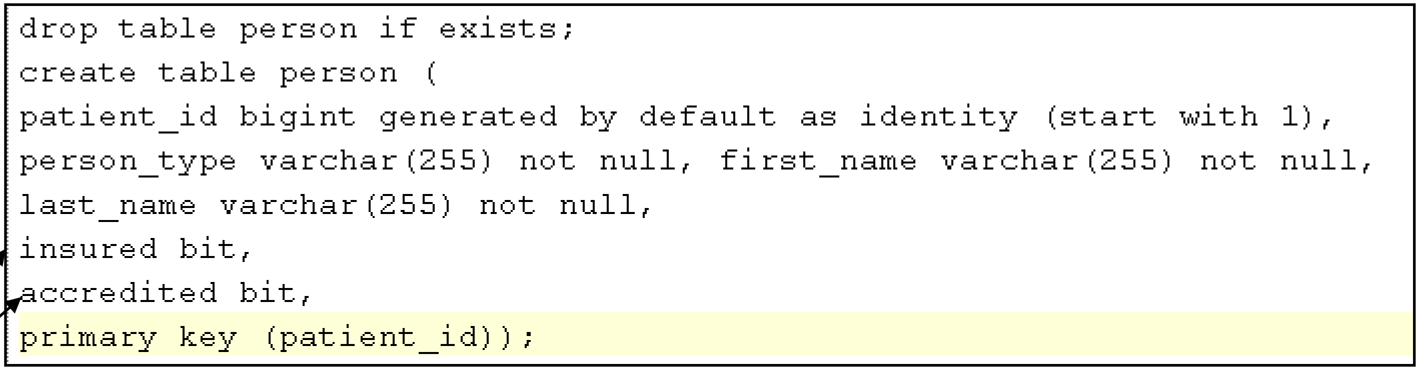


```
@Entity @DiscriminatorValue(value="p")
public class Patient extends Person {

    private boolean insured = false;

    public Patient() {}

    public Patient(String first, String last, boolean ins) {
```



```
@Entity
@DiscriminatorValue(value="d")
public class Physician extends Person {

    private boolean accredited;

    public Physician() {}

    public Physician(String first, String last, boolean ac) {
```

Resultant DDL



```
drop table person if exists;
create table person (
patient_id bigint generated by default as identity (start with 1),
person_type varchar(255) not null, first_name varchar(255) not null,
last_name varchar(255) not null,
insured bit,
accredited bit,
primary key (patient_id));
```

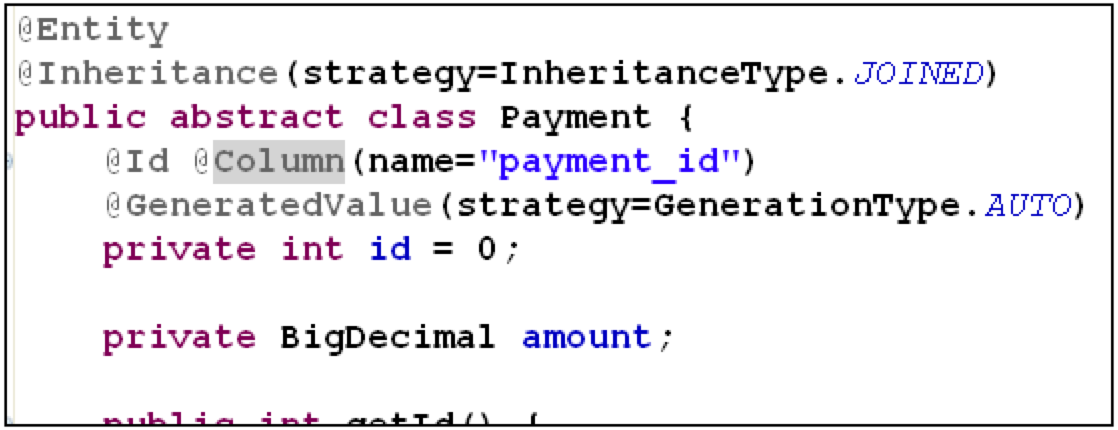
Has fields for both Patient and Physician

Table per Subclass

- Each subclass in the hierarchy gets its own table
- Parent class fields are in one table
- Primary key in subclass tables refer back to the parent class table
- Use joined-subclass configuration

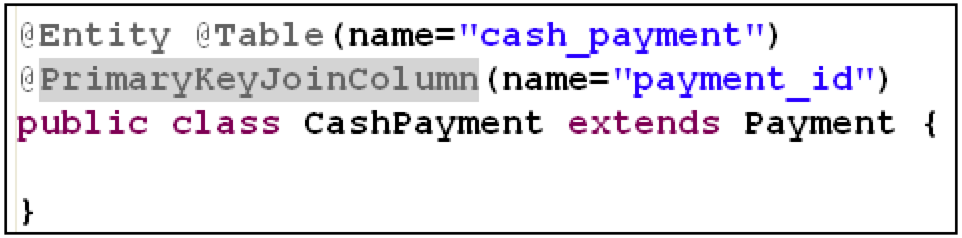
Table per Subclass Example

- CreditCardPayment, CashPayment, and CheckPayment all extend Payment

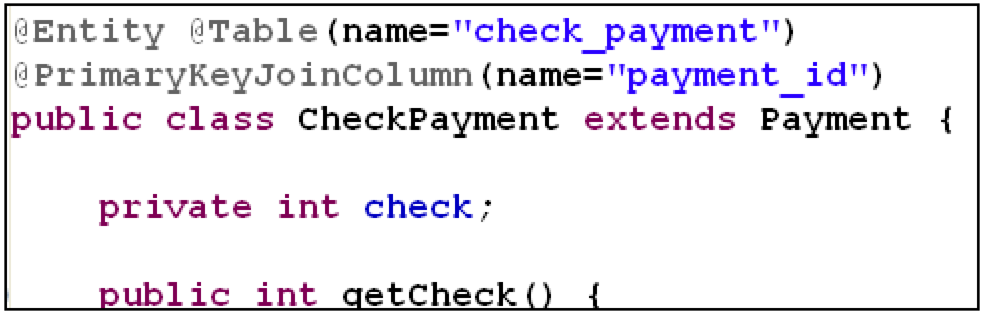


```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Payment {
    @Id @Column(name="payment_id")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id = 0;

    private BigDecimal amount;

    public int getId() {
```

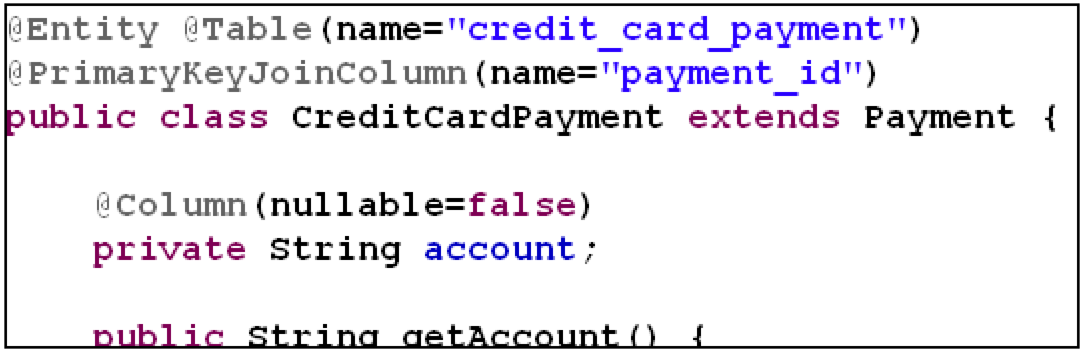
Payment Subclasses



```
@Entity @Table(name="cash_payment")
@PrimaryKeyJoinColumn(name="payment_id")
public class CashPayment extends Payment {

}
```

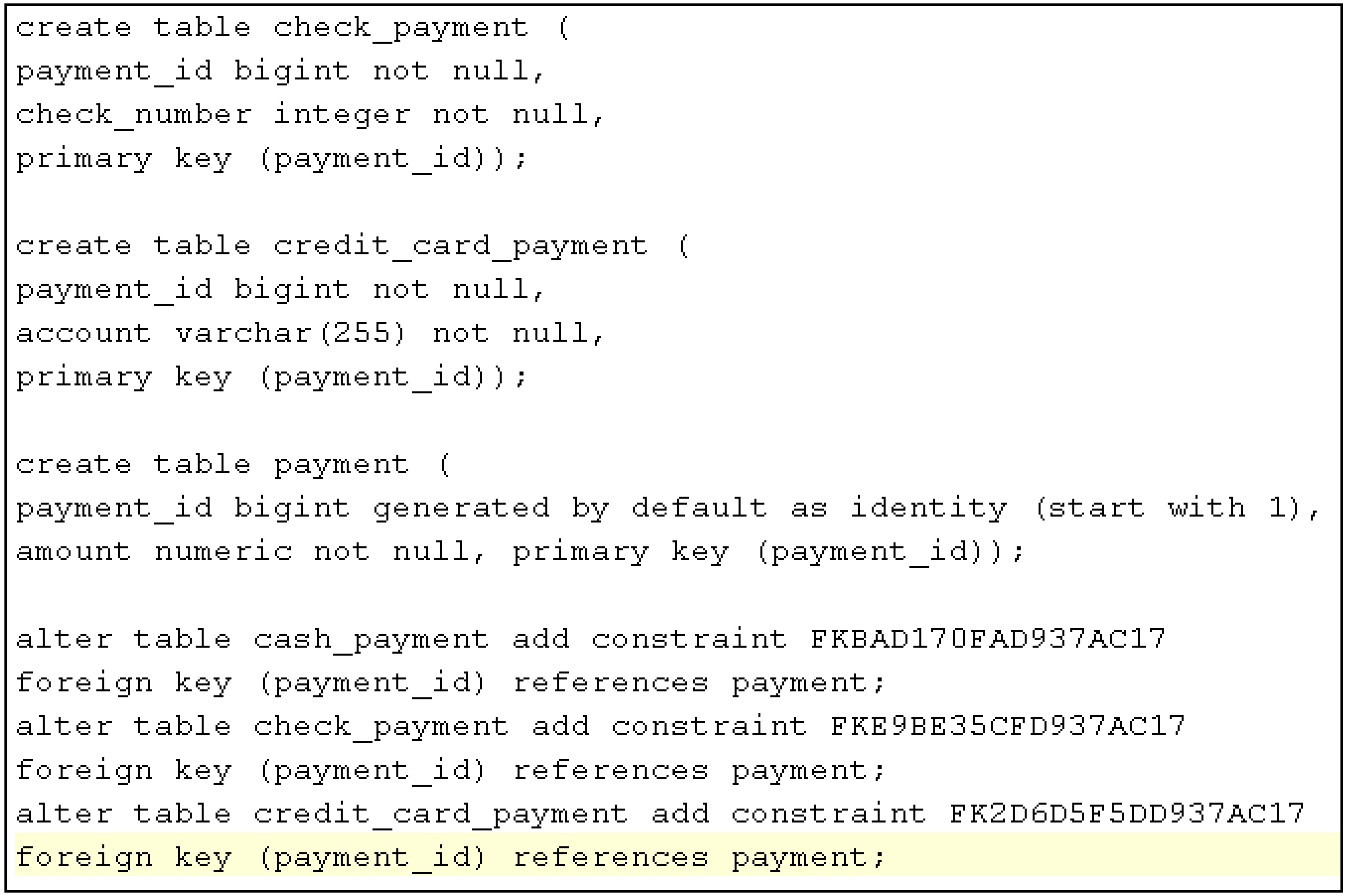


```
@Entity @Table(name="check_payment")
@PrimaryKeyJoinColumn(name="payment_id")
public class CheckPayment extends Payment {

    private int check;

    public int getCheck() {
```



```
@Entity @Table(name="credit_card_payment")
@PrimaryKeyJoinColumn(name="payment_id")
public class CreditCardPayment extends Payment {

    @Column(nullable=false)
    private String account;

    public String getAccount() {
```

Table Per Subclass DDL



```
create table check_payment (
payment_id bigint not null,
check_number integer not null,
primary key (payment_id));

create table credit_card_payment (
payment_id bigint not null,
account varchar(255) not null,
primary key (payment_id));

create table payment (
payment_id bigint generated by default as identity (start with 1),
amount numeric not null, primary key (payment_id));

alter table cash_payment add constraint FKBAD170FAD937AC17
foreign key (payment_id) references payment;
alter table check_payment add constraint FKE9BE35CFD937AC17
foreign key (payment_id) references payment;
alter table credit_card_payment add constraint FK2D6D5F5DD937AC17
foreign key (payment_id) references payment;
```

Discriminator w/Table Per Subclass

- Table per subclass also supports the use of a discriminator
- Not required

Table per Concrete Class

- Each class in the hierarchy gets its own table
- The parent class fields are duplicated in each table
- Two approaches
  - Union subclass
  - Implicit polymorphism

Union Subclass

- Map the Parent class as normal
  - No discriminator required
  - Specify parent class properties
  - @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
- Map each of the subclasses
  - Specify a table name
  - Specify subclass properties

Implicit Polymorphism

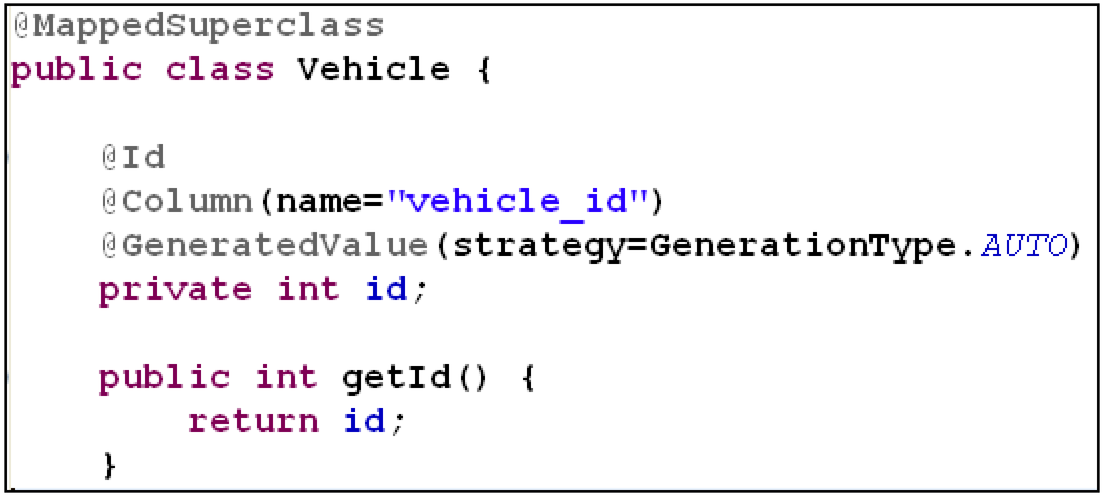
- Parent class is not mapped using hibernate
  - Annotate with @MappedSuperclass
- Each of the subclasses is mapped as a normal class
  - Map all of the properties, including the parent properties

Implicit Polymorphism Example

- Car and Truck extend Vehicle
- SUV extends Truck

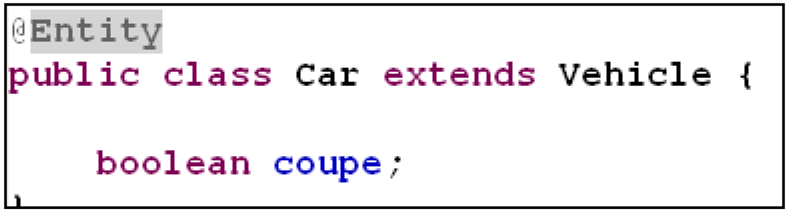
Vehicle Parent Class

- Class not mapped, but properties are

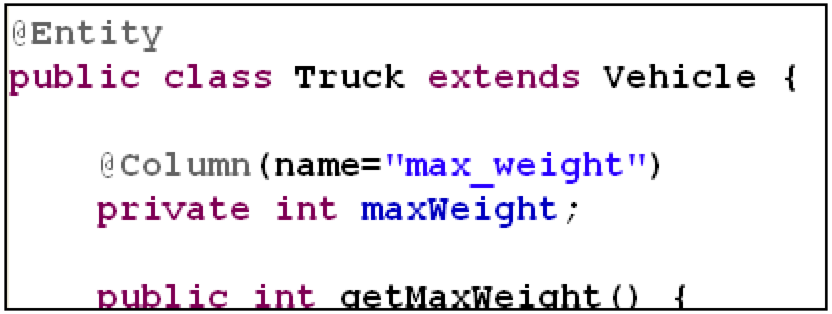


```
@MappedSuperclass
public class Vehicle {

    @Id
    @Column(name="vehicle_id")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    public int getId() {
        return id;
    }
```

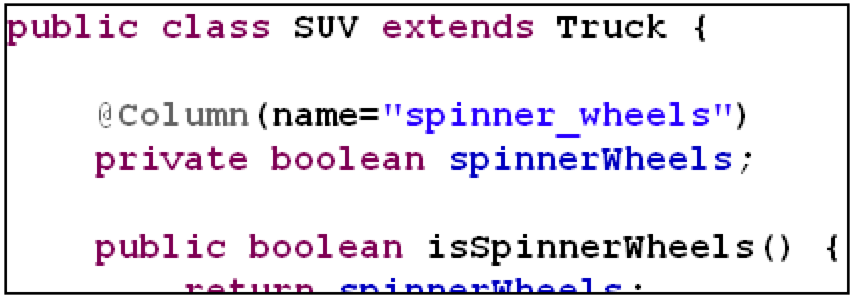
Vehicle Subclasses



```
@Entity
public class Car extends Vehicle {

    boolean coupe;
}
```

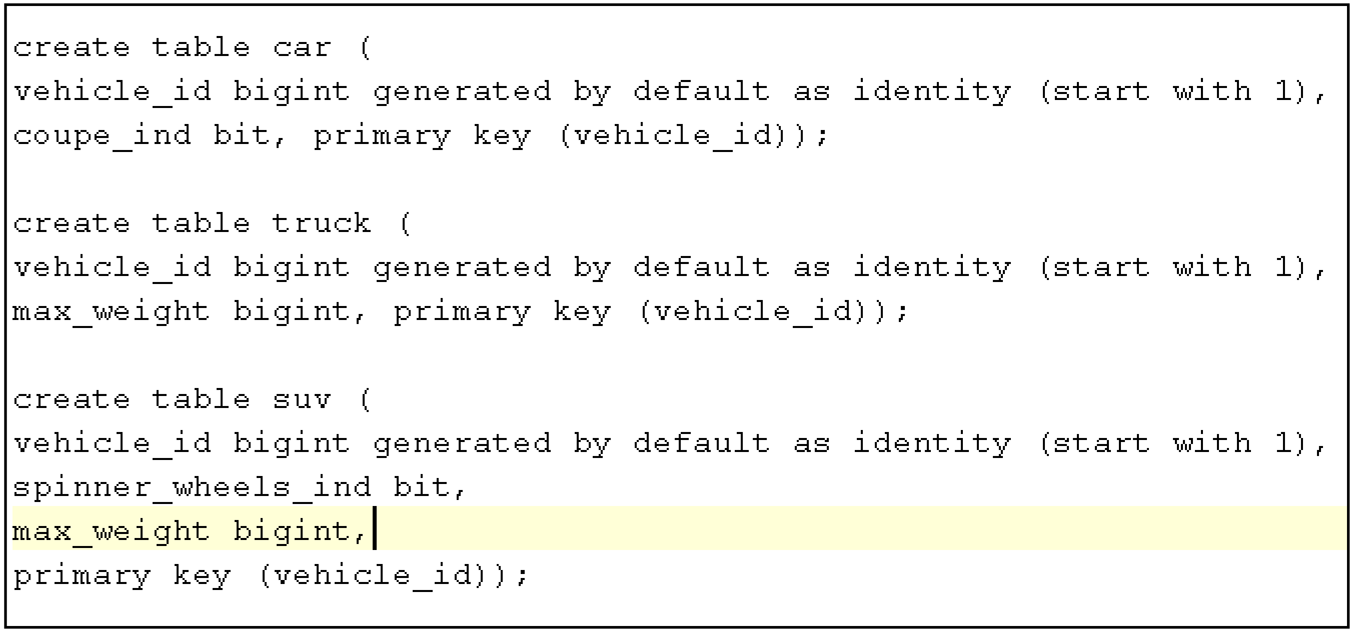


```
@Entity
public class Truck extends Vehicle {

    @Column(name="max_weight")
    private int maxWeight;

    public int getMaxWeight() {
```



```
public class SUV extends Truck {

    @Column(name="spinner_wheels")
    private boolean spinnerWheels;

    public boolean isSpinnerWheels() {
        return spinnerWheels;
```

Implicit Polymorphism DDL



```
create table car (
vehicle_id bigint generated by default as identity (start with 1),
coupe_ind bit, primary key (vehicle_id));

create table truck (
vehicle_id bigint generated by default as identity (start with 1),
max_weight bigint, primary key (vehicle_id));

create table suv (
vehicle_id bigint generated by default as identity (start with 1),
spinner_wheels_ind bit,
max_weight bigint,
primary key (vehicle_id));
```

Persistence Configuration Files

- Used to define application persistence properties
  - Database connection information
  - Dialect (database-specific language)
  - Logging
  - Schema creation parameters
- Default location is in CLASSPATH
- JPA
  - META-INF/persistence.xml
- Hibernate
  - hibernate.cfg.xml

Example persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
    <persistence-unit name=“persistence-unit">
            <properties>
            <property name="dialect" value="org.hibernate.dialect.DerbyDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.connection.driver_class"
value="org.apache.derby.jdbc.EmbeddedDriver"/>
<property name="hibernate.connection.url“ value="jdbc:derby:db;create=true"/>
            <property name="hibernate.connection.autocommit" value="false" />
        </properties>
    </persistence-unit>
</persistence>
```

Persistence Core Classes

- Hibernate
  - Configuration -> SessionFactory -> Session
  - Session is gateway to persistence functions
- JPA
  - Persistence -> EntityManagerFactory -> EntityManager
  - EntityManager is gateway to persistence functions

EntityManager Functionality

- persist(Obect o)
  - Saves or updates the specified object tree
- remove(Object o)
  - Deletes the specified object
- find(Class type, Serializable id)
  - Retrieves the item of the specified type by id
- merge(Object o)
  - Attaches a detached instance to the manager (required for update on item retrieved from a different manager)
- getTransaction()
  - Provides a transaction object to perform commit and rollback functions

Querying with JPA

- JPAQL/HQL
- Named Queries
- Criteria
- By Example
- Native SQL

JPAQL and HQL

- Query languages that are similar to SQL but are more object-centric
- Supports
  - Selection of object instances from persistent types
  - Selection of properties (rather than columns)
  - Polymorphic selection
  - Automatic joining for nested properties
- Should be very comfortable for those familiar with SQL
- JPAQL is a subset of HQL
  - Hibernate implementation will support both

Example JPAQL/HQL

“from Item i where i.name = ‘foobar’”

“from Item i where i.bidder.name = ‘Mike’” <<< Implicit join

“from Car c where c.coupe = true”

“from Item i where i.bids is not empty”

“from Item i where size(i.bids) > 3” <<< Implicit join

“from Item i order by i.name asc, i.entryDate asc”

“from Item i join i.bids b where b.amount > 100” <<< will return rows with an array

“select distinct(i) from Item i join i.bids b where b.amount > 100” <<< returns Items

“select i.name, i.description from Item i where entryDate > ?”

More JPAQL/HQL Features

- String functions
  - upper, lower, length, substring, concat, trim
- Aggregation functions
  - count, min, max, sum, avg
  - Can require “group by” clause
  - Also supports “having” on “group by”
- Subselects

Query Parameters

- Work just like JDBC Parameters
  - ? Will act as a placeholder for an indexed parameter
  - :name will designate a parameter with the name of “name”
- Examples
  - select i from Item where name like ?
  - select i from Item where name like :name

Running a JPAQL/HQL Query

- Must be created from the EntityManager or Session
  - EntityManger.createQuery(String jpaql)
  - Session.createQuery(String hql)
- Query objects can be configured
  - Maximum number of results
  - Parameters set
- Query objects can then be used to list out the results of the query
  - list() returns a java.util.List

Named Queries

- Predefined queries that can be executed by name
  - Standard QL used for query
- Defined using @NamedQuery
  - Typically above the persistent class the query is for
  - Defined with name and query values
- Running the query
  - EntityManager.createNamedQuery(name)
  - Session.getNamedQuery(name)

Criteria and Example Queries

- Hibernate only feature
- Full API for programmatically defining the query structure using objects
- Created by Session
- Criteria maps all logical restrictions to a query as objects and methods
- Example query uses an example object as the basis to find other objects
  - Based on Criteria API

Native SQL Queries

- Both JPA and Hibernate support running raw SQL
  - Session.createSQLQuery(sql)
  - EntityManager.createNativeQuery(sql)
- Supports specifying the entity type to convert the results to

Embeddable Objects

- Some classes contain persistent data, but are not true entities
  - Think Data Objects in DDD
- JPA supports this with @Embeddable
  - Annotate your embeddable class
- When your entity needs to store the fields of an embeddable class, include an instance of that type as a property and annotate it with @Embedded

Custom Types

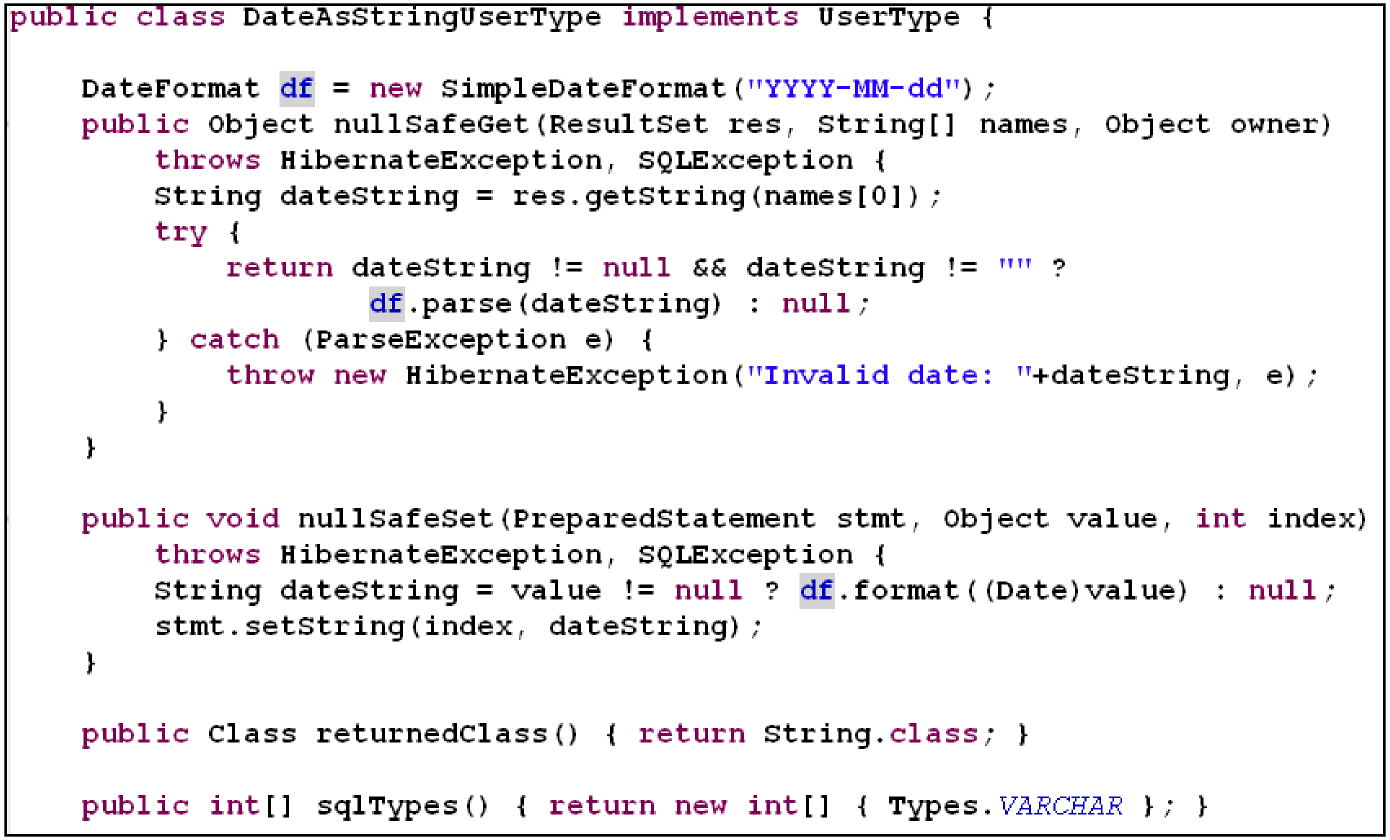
- Sometimes you don’t want the default mapping Hibernateassociates with your property
- Reasons
  - Combine two fields into one column
  - Numbers and dates stored as strings
  - Data encryption

Creating a UserType

- Implement the org.hibernate.usertype.UserType interface
- returnedClass()
  - Tells Hibernate what type of value to expect for your Type
- sqlTypes()
  - Tells Hibernate what JDBC types you are mapping to
- Key conversion methods
  - nullSafeGet
  - nullSafeSet
  - Have access to
    - JDBC resultSet
    - Column names for property
    - Owning object

Example UserType

- DateAsStringUserType converts dates to Strings formatted as yyyy-MM-dd



```
public class DateAsStringUserType implements UserType {

    DateFormat df = new SimpleDateFormat("YYYY-MM-dd");
    public Object nullSafeGet(ResultSet res, String[] names, Object owner)
        throws HibernateException, SQLException {
        String dateString = res.getString(names[0]);
        try {
            return dateString != null && dateString != "" ?
                    df.parse(dateString) : null;
        } catch (ParseException e) {
            throw new HibernateException("Invalid date: "+dateString, e);
        }
    }

    public void nullSafeSet(PreparedStatement stmt, Object value, int index)
        throws HibernateException, SQLException {
        String dateString = value != null ? df.format((Date)value) : null;
        stmt.setString(index, dateString);
    }

    public Class returnedClass() { return String.class; }

    public int[] sqlTypes() { return new int[] { Types.VARCHAR }; }
```

Other UserType Methods

- equals and hashcode
  - Make sure you check for nulls
- isMutable
  - Are objects of the returned type mutable (Strings are not)
- assemble and disassemble
  - Prepare for/extract from caching
  - Strings can be cached as is (simply return)
- replace
  - Handles a merge
  - Return the first parameter on immutable items