MONTANA

STATE UNIVERSITY

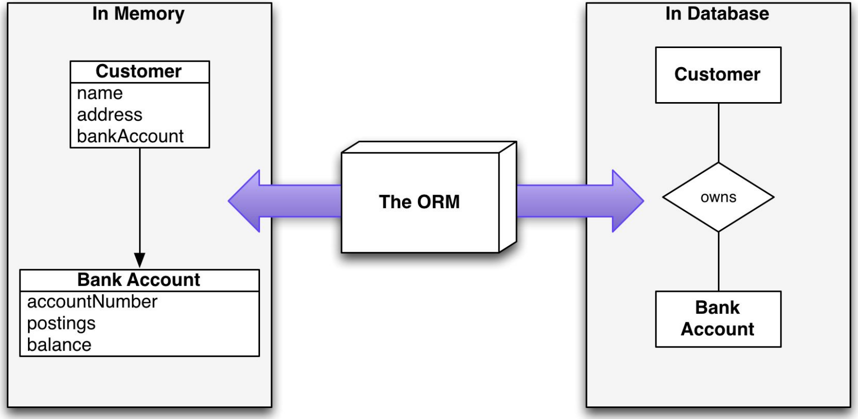
Object Relational Mapping

•••

Tools For Working With Databases

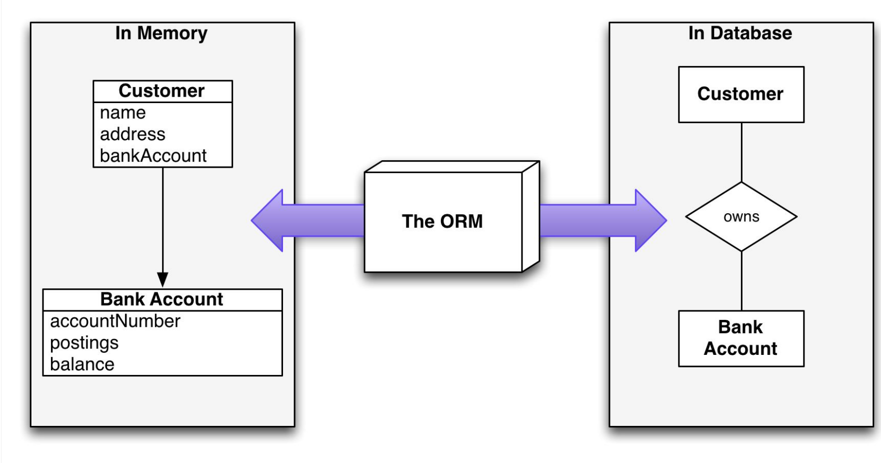
Object Relational Mapping

- The *Object/Relational Mapping* problem is the problem of mapping in memory objects to relations stored in a DBMS
- A system that does this management is called an *Object Relational Mapper (ORM)*



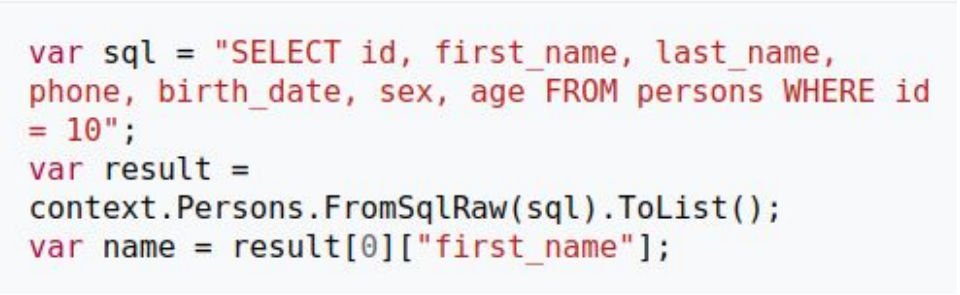
Object Relational Mapping

- Thus far in our projects, we have been building an ORM ourselves, by hand
    - This is done so that you can see how things are working at the SQL Level
- Professionally, you will most likely be working with an ORM of some sort

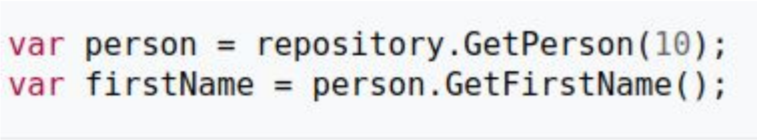


Object Relational Mapping

- Top: raw SQL accessing relational data directly from an Database API
- Below: the same logic, using an ORM



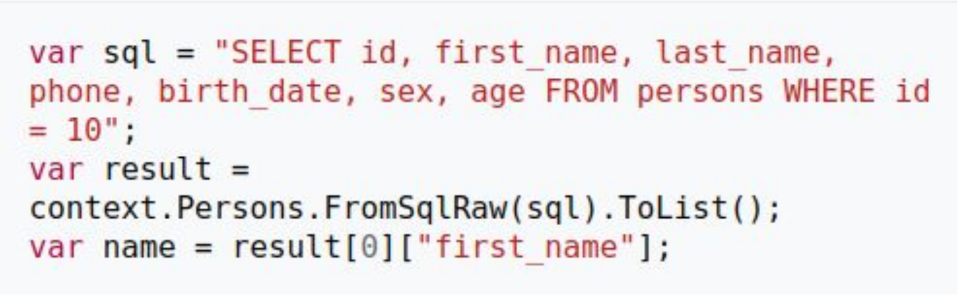
```
var sql = "SELECT id, first_name, last_name,
phone, birth_date, sex, age FROM persons WHERE id
= 10";
var result =
context.Persons.FromSqlRaw(sql).ToList();
var name = result[0]["first_name"];
```



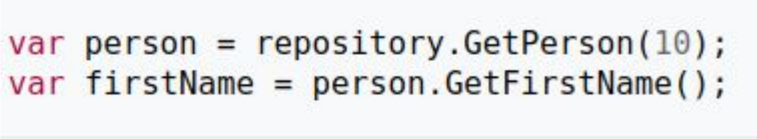
```
var person = repository.GetPerson(10);
var firstName = person.GetFirstName();
```

Object Relational Mapping

- Looks a lot nicer, doesn’t it?
- And it is…
  - Typically less code
  - Often has better code-tooling support (e.g. autoComplete)
- However
  - You are less connected to the underlying database
  - Easy to cause performance issues without realizing it



```
var sql = "SELECT id, first_name, last_name,
phone, birth_date, sex, age FROM persons WHERE id
= 10";
var result =
context.Persons.FromSqlRaw(sql).ToList();
var name = result[0]["first_name"];
```



```
var person = repository.GetPerson(10);
var firstName = person.GetFirstName();
```

Object Relational Mapping

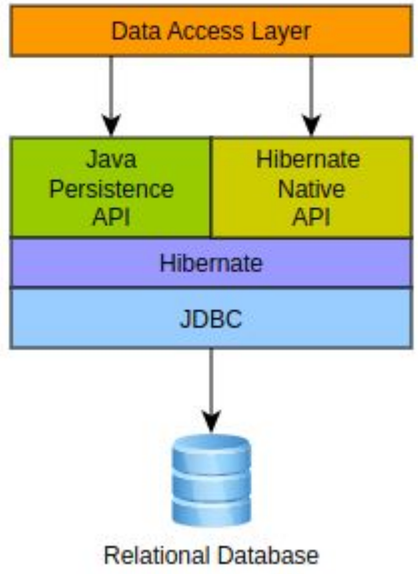
- In the java world, there are several options for ORM frameworks
- The oldest one that I’m aware of is Hibernate
  - First release in 2001
  - Developed by Cirrus Technologies, then JBoss, now Red Hat

Object Relational Mapping

- I don’t particularly care for Hibernate, but it is a standard and demonstrates most things an ORM will do
- Other alternatives
  - Spring
  - jOOQ
  - Apache Cayenne

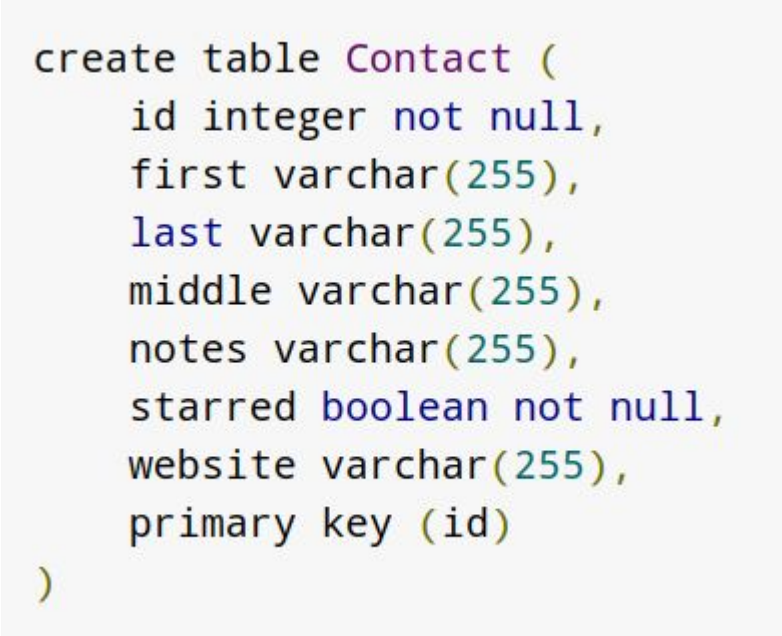
Hibernate: Architecture

- Hibernate Architectural Model
  - Data Access Layer: Java objects
  - JPI/HNI: Database abstraction layer
  - Hibernate: core Hibernate
  - JDBC: The Java Database Connectivity API
    - This is what we are using
  - The Relational Database



Hibernate: Mapping

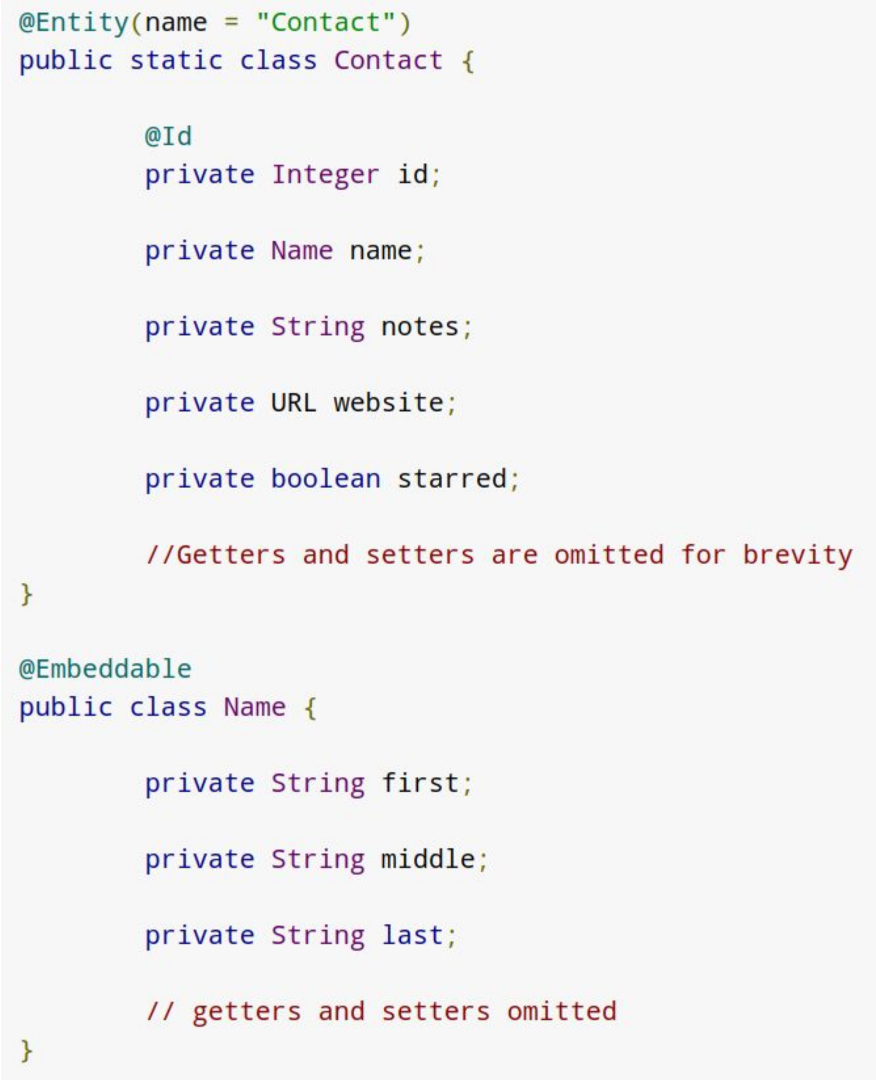
- Consider this Contact table
  - ID field (Primary Key)
  - First, last, middle name
  - Notes
  - Starred
  - Website



```
create table Contact (
    id integer not null,
    first varchar(255),
    last varchar(255),
    middle varchar(255),
    notes varchar(255),
    starred boolean not null,
    website varchar(255),
    primary key (id)
)
```

Hibernate: Mapping

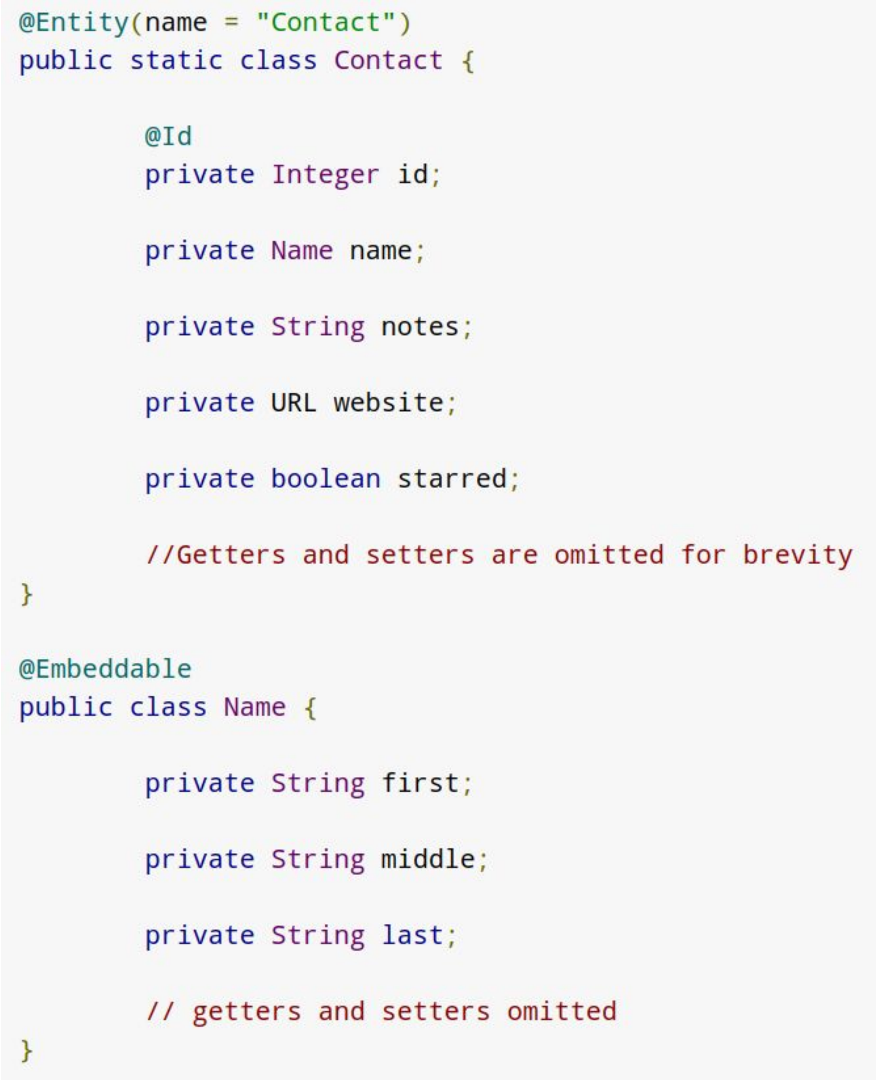
- Hibernate uses java *annotations* to provide metadata
  - This helps hibernate map the fields in the java object to the database
  - Very common approach for metadata in Java frameworks



```
@Entity(name = "Contact")
public static class Contact {

        @Id
        private Integer id;

        private Name name;

        private String notes;

        private URL website;

        private boolean starred;

        //Getters and setters are omitted for brevity
}

@Embeddable
public class Name {

        private String first;

        private String middle;

        private String last;

        // getters and setters omitted
}
```

Hibernate: Mapping

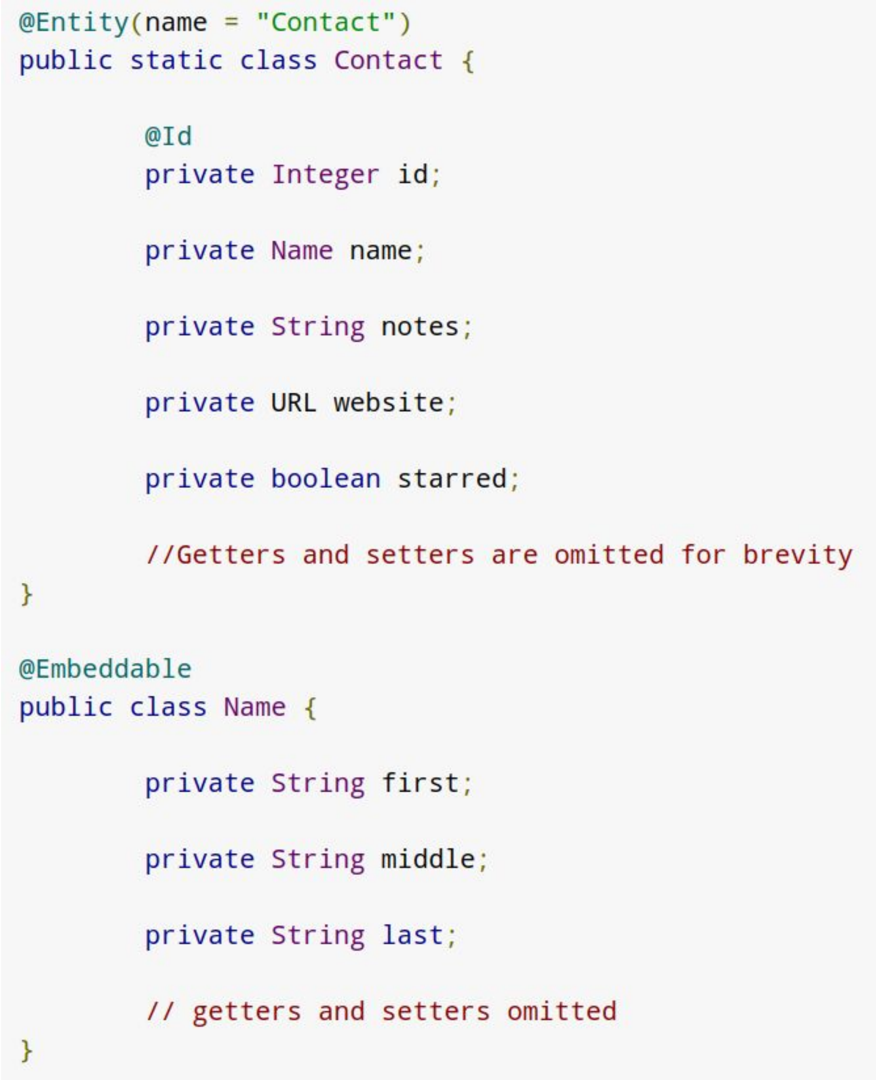
- @Entity - this class is an entity that maps to the *Contact* table
- @Id - this the primary key for this table
- @GeneratedValue - this value is automatically generated by the database
  - Shown in a few slides



```
@Entity(name = "Contact")
public static class Contact {

        @Id
        private Integer id;

        private Name name;

        private String notes;

        private URL website;

        private boolean starred;

        //Getters and setters are omitted for brevity
}

@Embeddable
public class Name {

        private String first;

        private String middle;

        private String last;

        // getters and setters omitted
}
```

Hibernate: Mapping

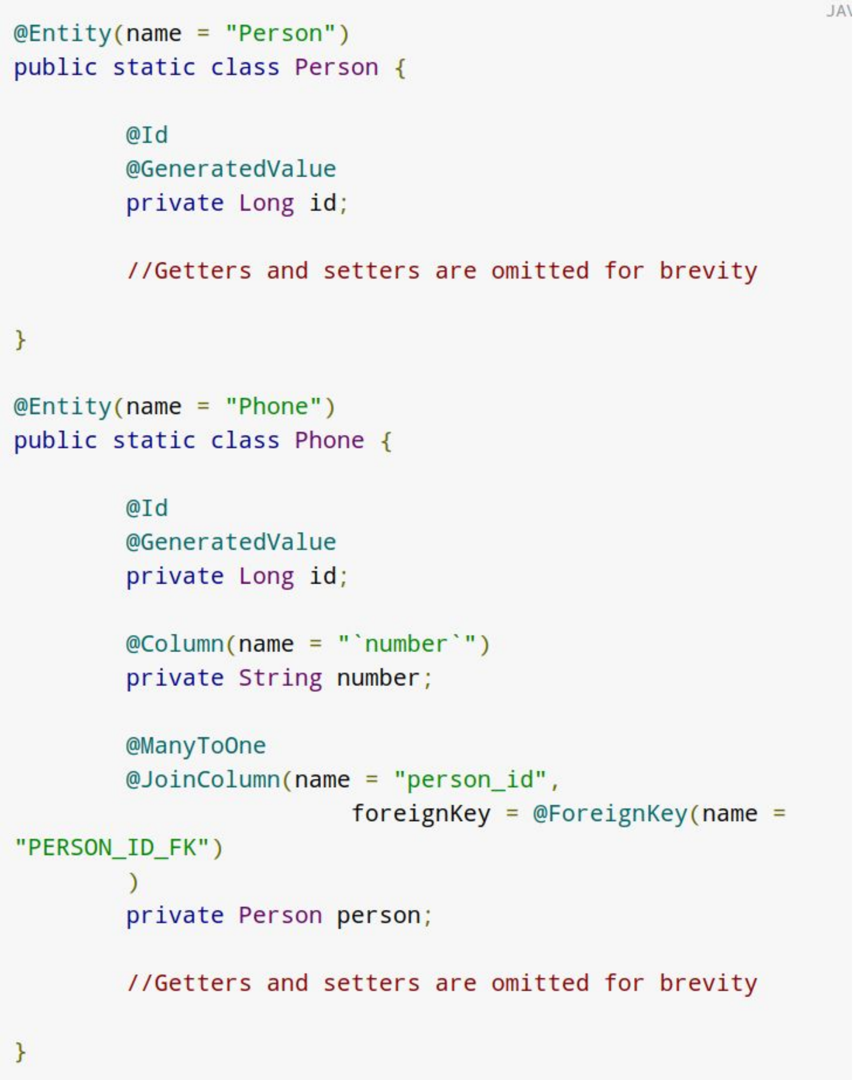
- Note that the Name class has been pulled out and annotated as @Embeddable
    - The object oriented concept *does not match* the database implementation
    - Columns are directly in the Contact table, but are modeled in a more OO approach here
    - More on this next lecture



```
@Entity(name = "Contact")
public static class Contact {

        @Id
        private Integer id;

        private Name name;

        private String notes;

        private URL website;

        private boolean starred;

        //Getters and setters are omitted for brevity
}

@Embeddable
public class Name {

        private String first;

        private String middle;

        private String last;

        // getters and setters omitted
}
```

Hibernate: Associations

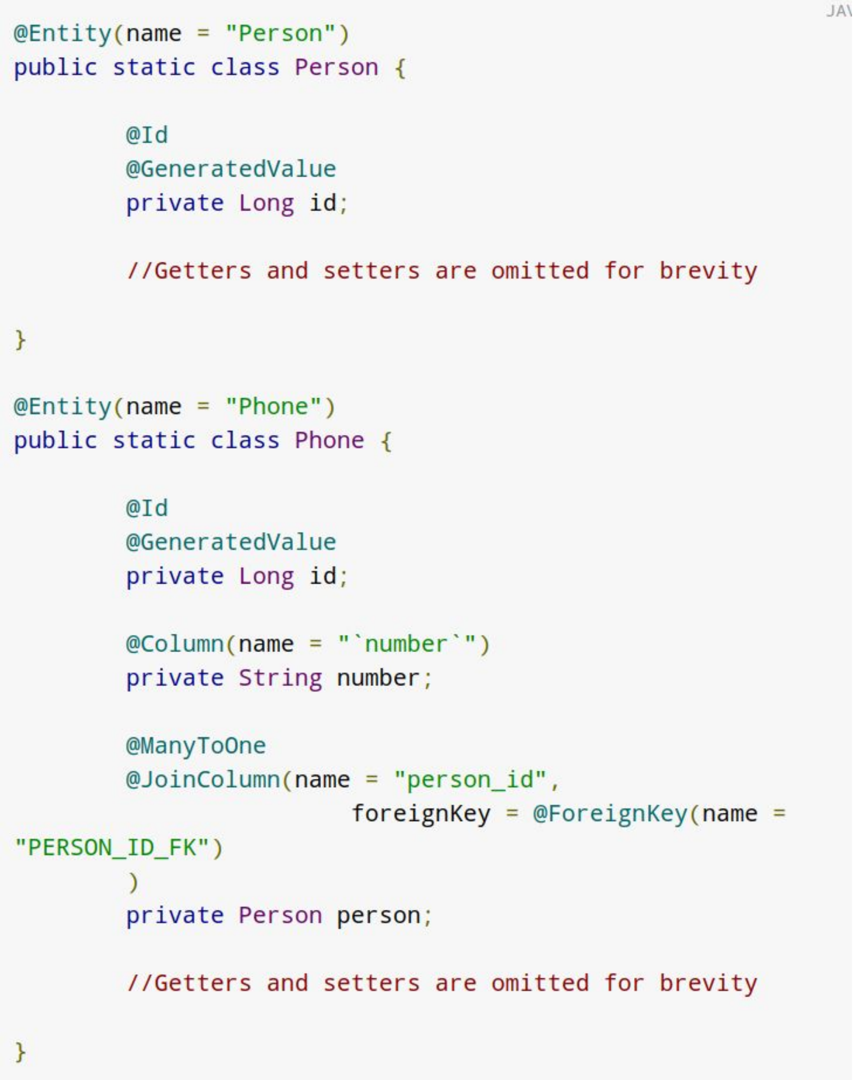
- *Associations* describe how two or more entities form a relationship by providing join semantics for the relationship
- Here we have a phone, which has a Many-to-one relationship with the Person class using the *@ManyToOne* annotation



```
@Entity(name = "Person")
public static class Person {

        @Id
        @GeneratedValue
        private Long id;

        //Getters and setters are omitted for brevity

}

@Entity(name = "Phone")
public static class Phone {

        @Id
        @GeneratedValue
        private Long id;

        @Column(name = "`number`")
        private String number;

        @ManyToOne
        @JoinColumn(name = "person_id",
                        foreignKey = @ForeignKey(name =
"PERSON_ID_FK")
        )
        private Person person;

        //Getters and setters are omitted for brevity

}
```

Hibernate: Associations

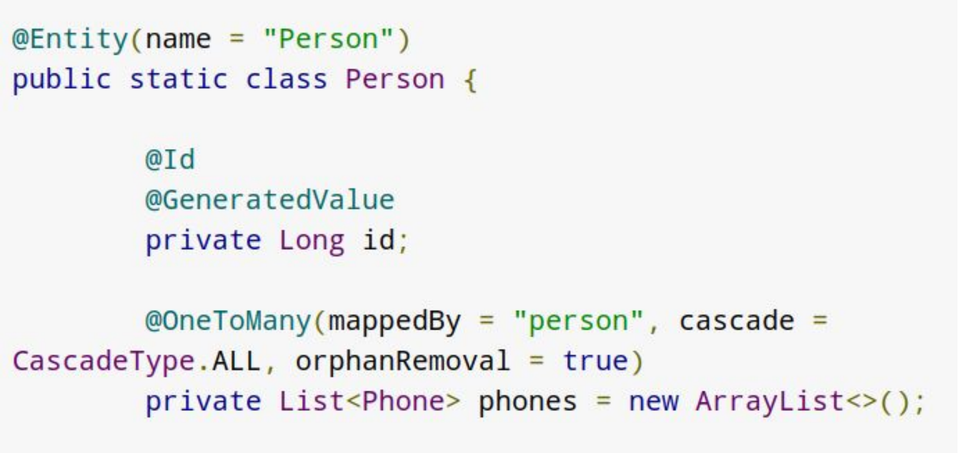
- Note the use of the @JoinColum annotation to describe the foriegn key to be used in the relationship



```
@Entity(name = "Person")
public static class Person {

        @Id
        @GeneratedValue
        private Long id;

        //Getters and setters are omitted for brevity

}

@Entity(name = "Phone")
public static class Phone {

        @Id
        @GeneratedValue
        private Long id;

        @Column(name = "`number`")
        private String number;

        @ManyToOne
        @JoinColumn(name = "person_id",
                        foreignKey = @ForeignKey(name =
"PERSON_ID_FK")
        )
        private Person person;

        //Getters and setters are omitted for brevity

}
```

Hibernate: Associations

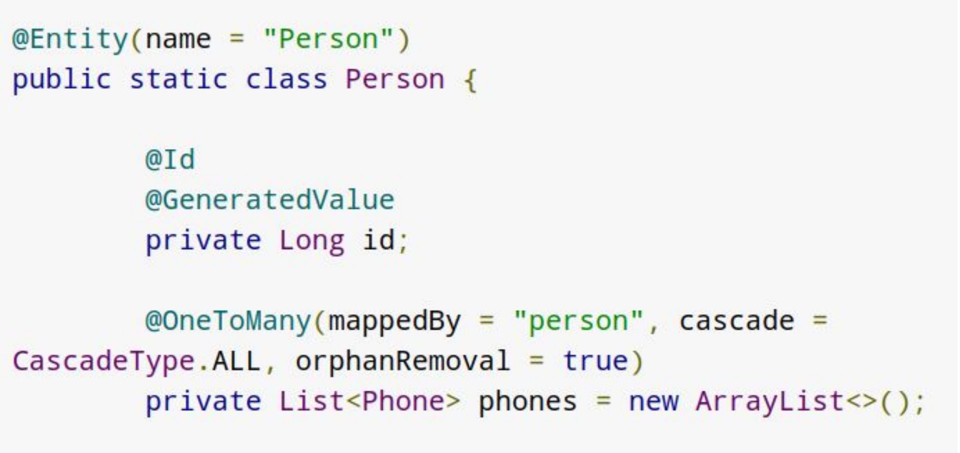
- *List Associations* are defined using the @OneToMany annotation
    - Hibernate supports join tables or direct references
    - Here it is a direct reference
        - Note that it will programmatically cascade deletes
        - And will also remove any “orphans” - any Phones that have a null Person



```
@Entity(name = "Person")
public static class Person {

        @Id
        @GeneratedValue
        private Long id;

        @OneToMany(mappedBy = "person", cascade =
CascadeType.ALL, orphanRemoval = true)
        private List<Phone> phones = new ArrayList<>();
```

Hibernate: Associations

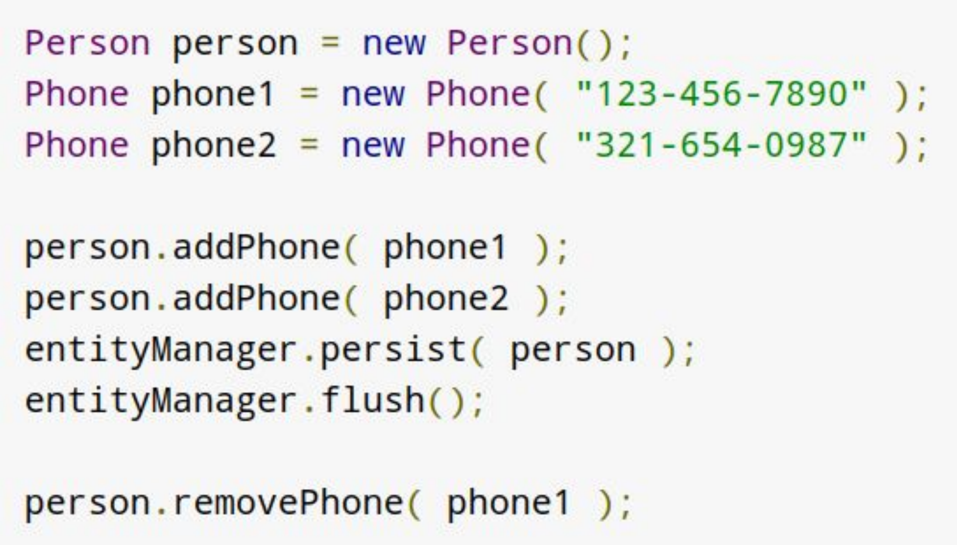
- Note that this association does not specify any join attributes
- Rather it defers to the *person* property on the phone, on the other side of the 1-to-Many relationship
    - Subtle, and a little annoying



```
@Entity(name = "Person")
public static class Person {

        @Id
        @GeneratedValue
        private Long id;

        @OneToMany(mappedBy = "person", cascade =
CascadeType.ALL, orphanRemoval = true)
        private List<Phone> phones = new ArrayList<>();
```

Hibernate: Associations

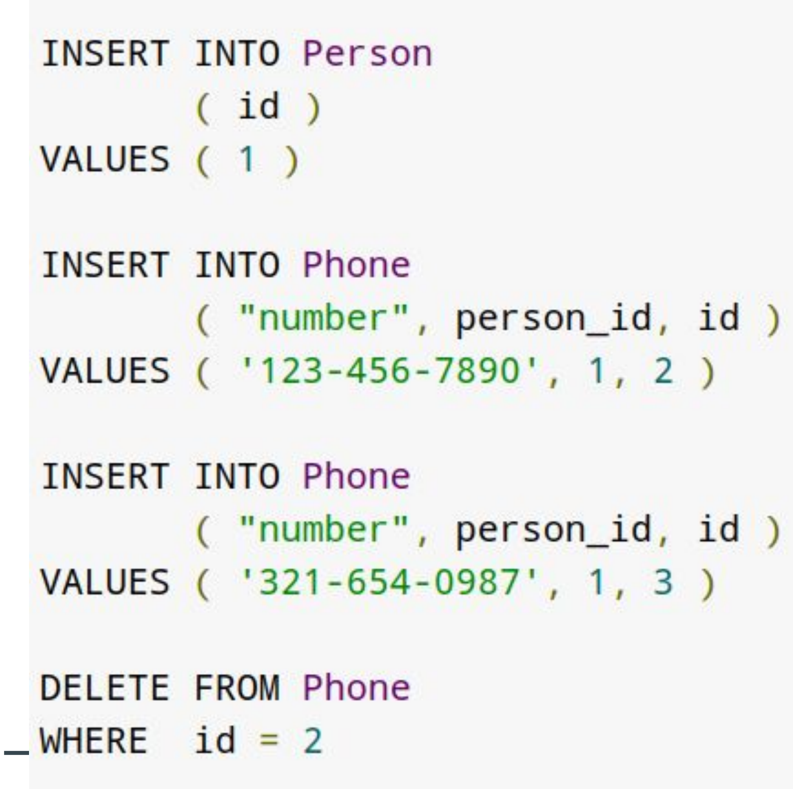
- Given this java code...



```java
Person person = new Person();
Phone phone1 = new Phone( "123-456-7890" );
Phone phone2 = new Phone( "321-654-0987" );

person.addPhone( phone1 );
person.addPhone( phone2 );
entityManager.persist( person );
entityManager.flush();

person.removePhone( phone1 );
```

Hibernate: Associations

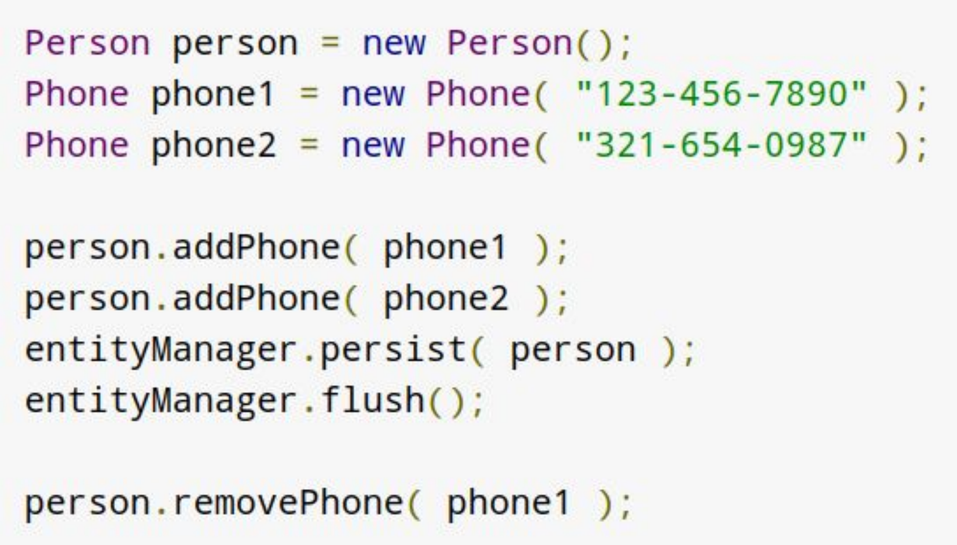
- Given this java code…
- This SQL will be executed



```sql
INSERT INTO Person
       ( id )
VALUES ( 1 )

INSERT INTO Phone
       ( "number", person_id, id )
VALUES ( '123-456-7890', 1, 2 )

INSERT INTO Phone
       ( "number", person_id, id )
VALUES ( '321-654-0987', 1, 3 )

DELETE FROM Phone
WHERE  id = 2
```

Hibernate: Associations

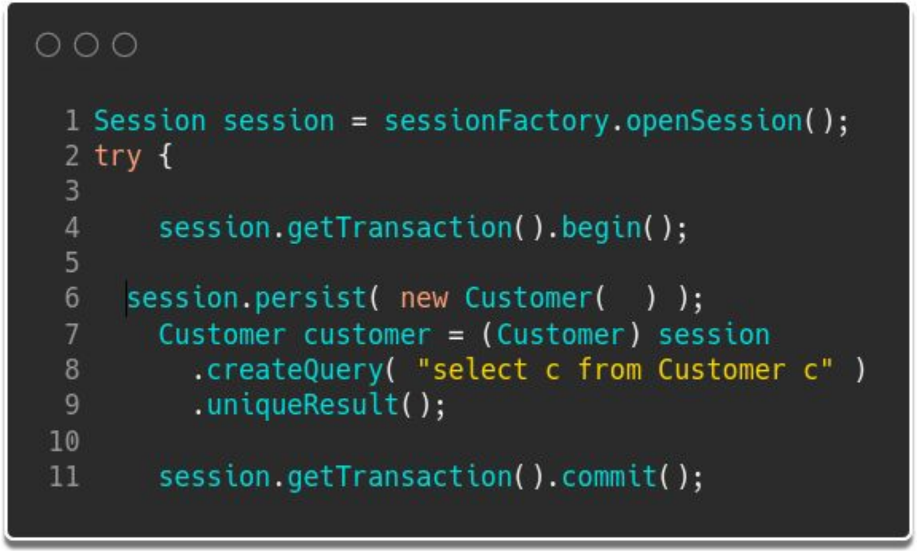
- What’s the difference between *flush()* and *persist()*?
- *persist()* - Hibernate, here is some data for you to save
- *flush()* - Hibernate, make sure that all the changes you have has been synchronized with the database



```
Person person = new Person();
Phone phone1 = new Phone( "123-456-7890" );
Phone phone2 = new Phone( "321-654-0987" );

person.addPhone( phone1 );
person.addPhone( phone2 );
entityManager.persist( person );
entityManager.flush();

person.removePhone( phone1 );
```

Hibernate: Transactions

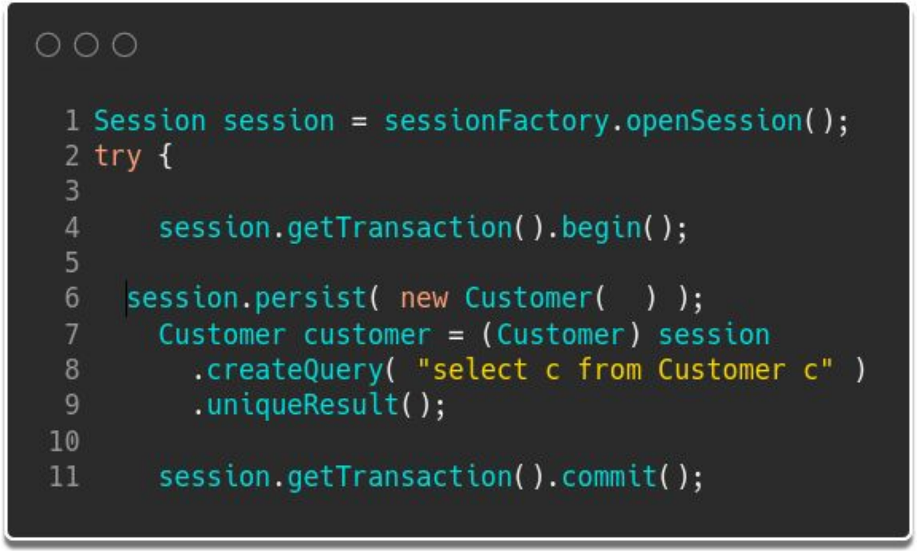
- Hibernate supports transactions
  - The API is pretty terrible, I omitted a bunch of code...
  - The begin() and commit() methods start and commit the transaction



```
Session session = sessionFactory.openSession();
try {

    session.getTransaction().begin();

  session.persist( new Customer(  ) );
    Customer customer = (Customer) session
      .createQuery( "select c from Customer c" )
      .uniqueResult();

    session.getTransaction().commit();
```

Hibernate: Transactions

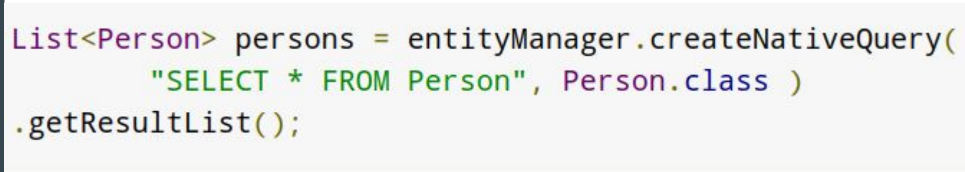
- Side Rant: This is the problem with the Java community
  - API designers just can’t get out of their own way and build an API without a ton of builders, factories and so forth
  - A legacy of the J2EE era
  - Too bad, java is a pretty good language and the JVM is awesome



```
Session session = sessionFactory.openSession();
try {

    session.getTransaction().begin();

  session.persist( new Customer(   ) );
    Customer customer = (Customer) session
      .createQuery( "select c from Customer c" )
      .uniqueResult();

    session.getTransaction().commit();
```

Hibernate: Querying

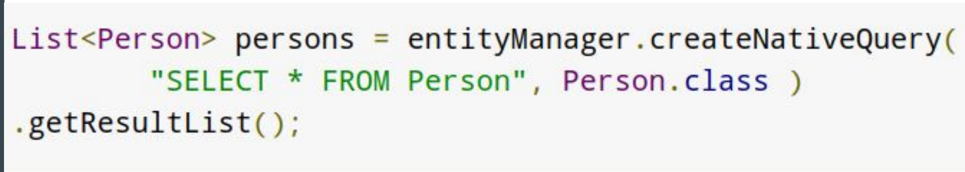
- Hibernate offers a bunch of different ways to query data
- We will focus on two
  - Native SQL
  - HQL



```
List<Person> persons = entityManager.createNativeQuery(
        "SELECT * FROM Person", Person.class )
.getResultList();
```

Hibernate: Querying

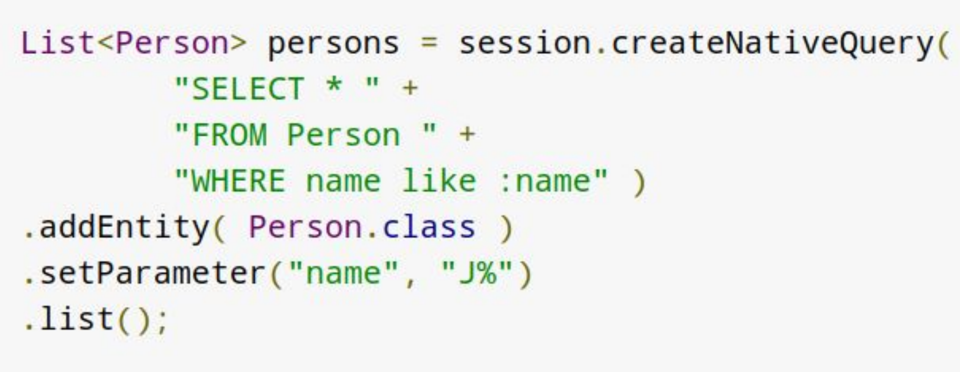
- Native querying uses the native query syntax of the backing database
- Produces a List of the type given as an argument



```
List<Person> persons = entityManager.createNativeQuery(
        "SELECT * FROM Person", Person.class )
.getResultList();
```

Hibernate: Parameters

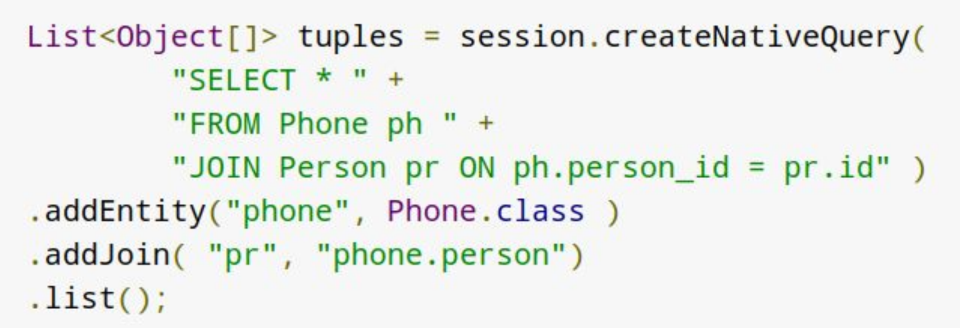
- Adding parameters is simple as well
- Note that parameter is name-based rather than index based, as in raw JDBC



```
List<Person> persons = session.createNativeQuery(
        "SELECT * " +
        "FROM Person " +
        "WHERE name like :name" )
.addEntity( Person.class )
.setParameter("name", "J%")
.list();
```

Hibernate: Eager Loading

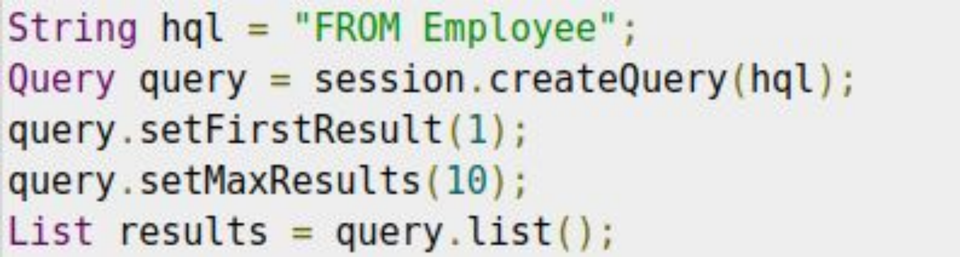
- A person has multiple phones
- We may wish to avoid multiple queries to display this information
- Hibernate can eagerly load collections
  - One query for all the info



```
List<Object[]> tuples = session.createNativeQuery(
        "SELECT * " +
        "FROM Phone ph " +
        "JOIN Person pr ON ph.person_id = pr.id" )
.addEntity("phone", Phone.class )
.addJoin( "pr", "phone.person")
.list();
```

Hibernate: HQL

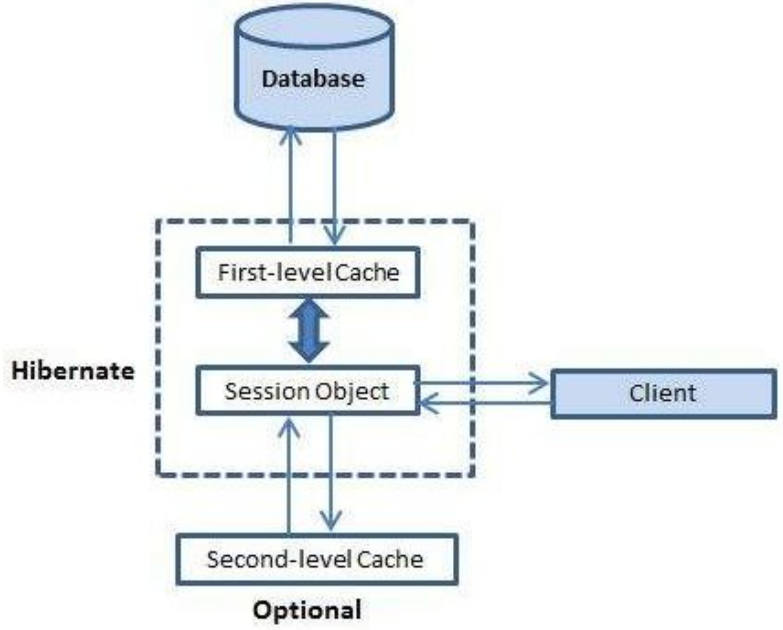
- Hibernate has implemented its own query language, *HQL*
  - Similar to SQL
  - Object oriented
- Supports niceties such as
  - omitting some unnecessary syntax
  - allows you to use polymorphic information in your queries



```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
query.setFirstResult(1);
query.setMaxResults(10);
List results = query.list();
```

Hibernate: Caching

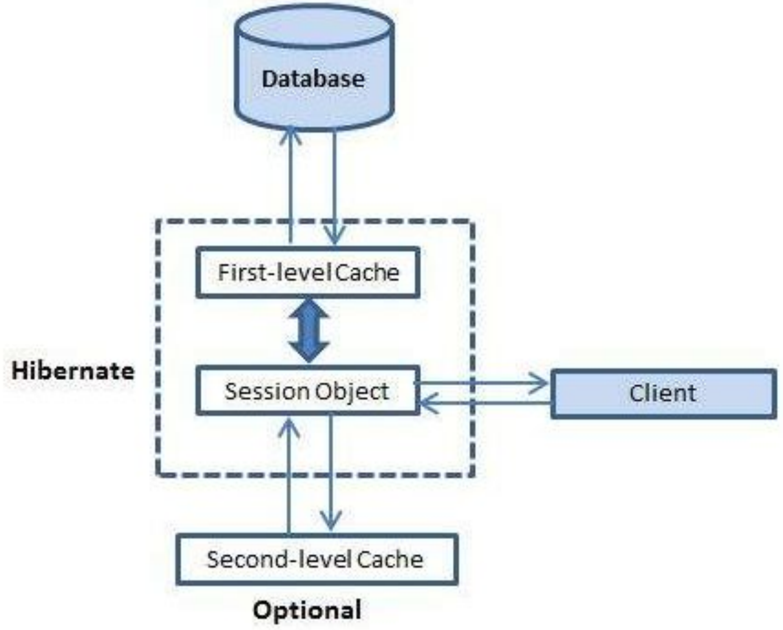
- Hibernate has multi-level cache infrastructure
- First-level cache
  - Multiple updates to an object will be kept until an update (flush()) occurs
- Second-level cache
  - An optional, pluggable cache layer
- Query-level cache
  - An optional layer that caches query results



Hibernate: Caching

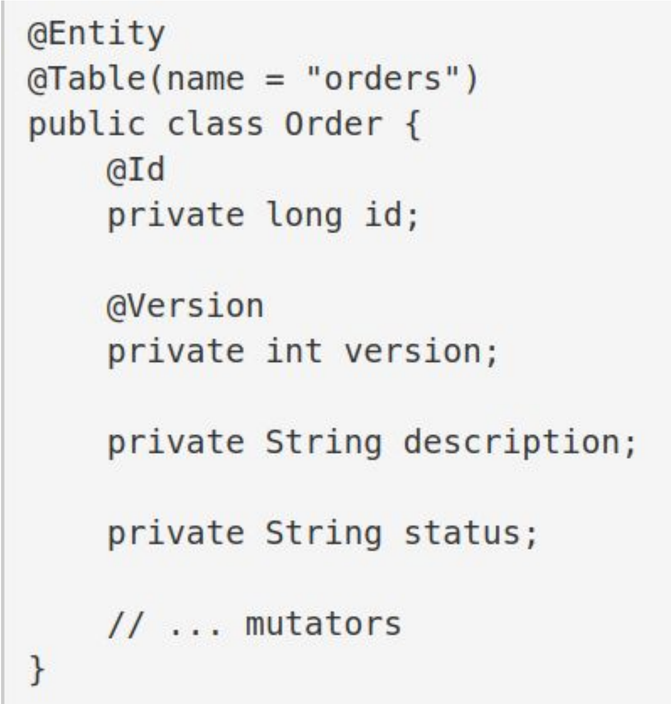
- Caching can be an extremely important aspect of system performance
- But remember:

  There are only two hard things in Computer Science: cache invalidation and naming things.
  -- Phil Karlton



Optimistic Concurrency

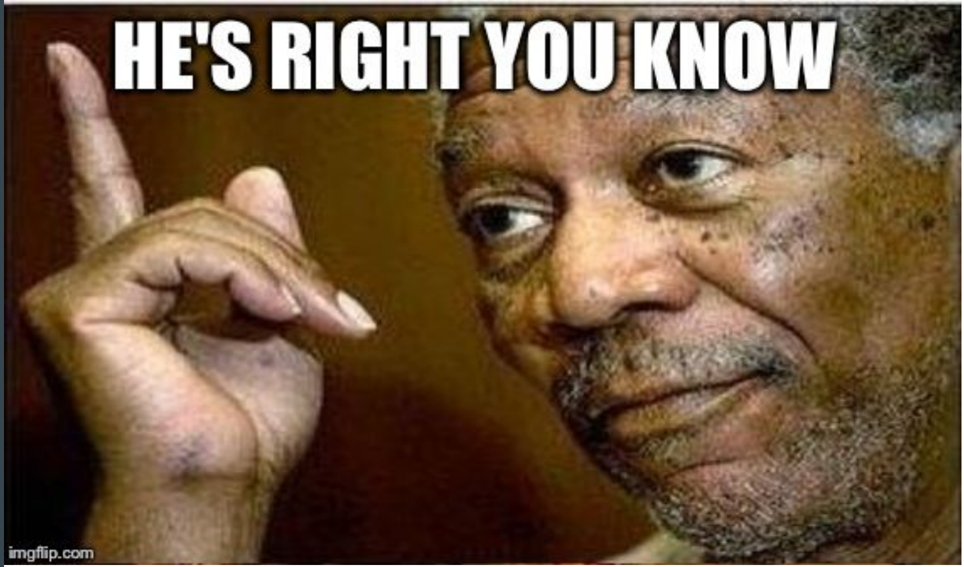
- *“The only approach that is consistent with high concurrency and high scalability, is optimistic concurrency control with versioning.” --Hibernate Docs*



```
@Entity
@Table(name = "orders")
public class Order {
    @Id
    private long id;

    @Version
    private int version;

    private String description;

    private String status;

    // ... mutators
}
```

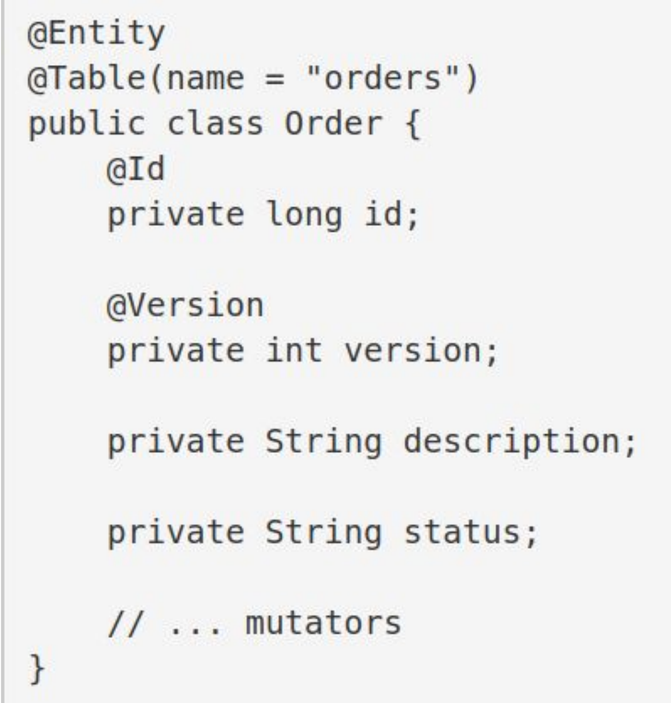
Optimistic Concurrency

- *“The only approach that is consistent with high concurrency and high scalability, is optimistic concurrency control with versioning.” --Hibernate Docs*



Optimistic Concurrency

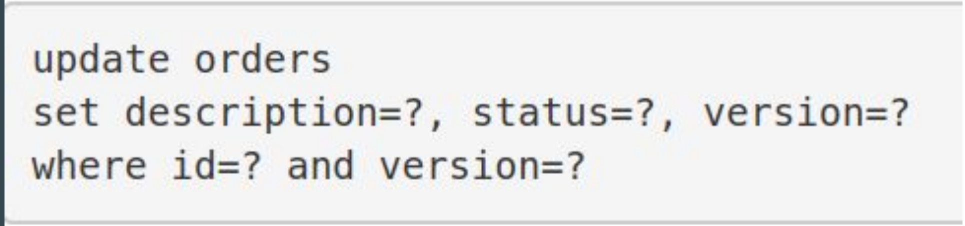
- Note the use of the *@Version* annotation
- This alerts Hibernate to use optimistic concurrency in updates



```
@Entity
@Table(name = "orders")
public class Order {
    @Id
    private long id;

    @Version
    private int version;

    private String description;

    private String status;

    // ... mutators
}
```

Optimistic Concurrency

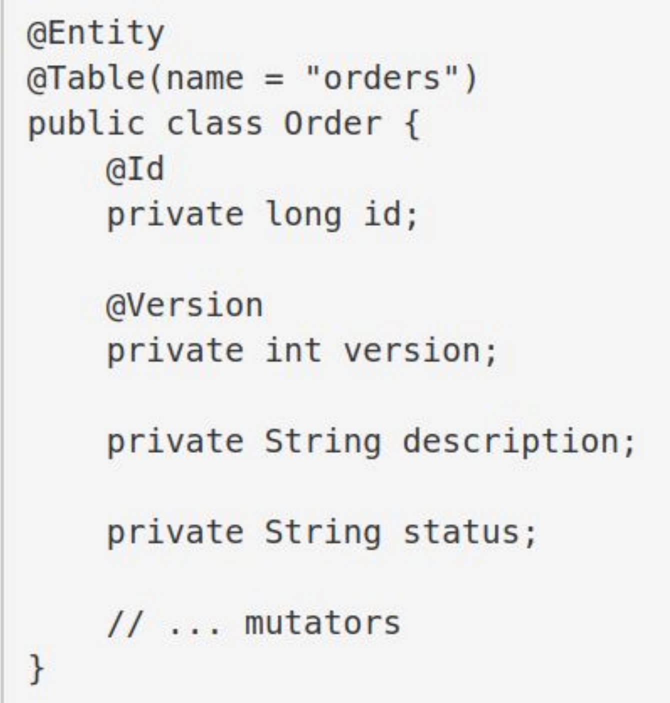
- Hibernate will now emit SQL that looks like this when updating Order objects



```
update orders
set description=?, status=?, version=?
where id=? and version=?
```

Optimistic Concurrency

- You may have noticed slightly different annotations here
- This is using the JPA annotations rather than the native Hibernate annotations
  - Welcome to java!



```
@Entity
@Table(name = "orders")
public class Order {
    @Id
    private long id;

    @Version
    private int version;

    private String description;

    private String status;

    // ... mutators
}
```

Object Relational Mapping

- Today we discussed what ORM systems are
  - A tool for managing objects in memory and *mapping* them down to relations in a database
- We took a look at Hibernate, a popular OR framework for Java
  - Defining entities
  - Working with entities in code
  - Querying entities
- Next time we will discuss problems with ORMs
  - The Object-Relational Impedance Mismatch

Test Relevance

- In the project we are not using an ORM
- In fact, we are *building* a rudimentary ORM
- I do not expect you to know anything about Hibernate in particular for the final
    - I am showing you this to prepare you for a future job, the O/R tool you will end up using will most likely *not* be Hibernate
- I do expect you to understand what an ORM is
- Next lecture, I will discuss many of the problems with ORMs and I will expect you to understand those problems

MONTANA
STATE UNIVERSITY