Movie: getAll

```
@Override
public List<Movie> getAll() {
   JdbcTemplate template = getJdbcTemplate();
   return template.query("select * from MOVIES",
      new RowMapper<Movie>(){
         @Override
         public Movie mapRow(ResultSet rs, int row)
                                          throws SQLException {
            return createMovie(rs);
         }
      }
   );
}
```

- MovieDAO extends JdbcDaoSupport
  - getJdbcTemplate() returns the template

Movie: createMovie

```
private Movie createMovie(ResultSet rs) throws SQLException {
   long priceCategory = rs.getLong("PRICECATEGORY_FK");
   Movie m = new Movie(
         rs.getString("MOVIE_TITLE"),
         rs.getDate("MOVIE_RELEASEDATE"),
         priceCategoryDAO.getById(priceCategory));
   m.setId(rs.getLong("MOVIE_ID"));
   m.setRented(rs.getBoolean("MOVIE_RENTED"));
   return m;
}
```

- Movie constructor does not accept a primary key
  - User code never provides a PK

Movie: getById

```
@Override
public Movie getById(Long id){
   JdbcTemplate template = getJdbcTemplate();
   return template.queryForObject(
      "select * from MOVIES where MOVIE_ID = ?",
      new RowMapper<Movie>(){
         @Override
         public Movie mapRow(ResultSet rs, int row)
                                          throws SQLException {
            return createMovie(rs);
         }
      },
      id
   );
}
```

Create / Update / Delete

- **SaveOrUpdate**
  - Decide which operation is needed depending on the PK
    - getId() == null => not saved in database => INSERT
    - getId() != null => stored in database => UPDATE

```
@Override
public void saveOrUpdate(Movie movie) {
   JdbcTemplate template = getJdbcTemplate();
   if (movie.getId() == null) {
      // insert in DB
      long id = ...;
      movie.setId(id);
   }
   else {
      // update in DB
   }
}
```

Create / Update / Delete

- **Update**

```
...
else {
   // update
   template.update(
      "UPDATE MOVIES SET MOVIE_TITLE = ?, MOVIE_RENTED = ?,
           MOVIE_RELEASEDATE = ?,PRICECATEGORY_FK = ?
           WHERE MOVIE_ID = ?",
      movie.getTitle(),
      movie.isRented(),
      movie.getReleaseDate(),
      movie.getPriceCategory().getId(),
      movie.getId());
}
...
```

Create / Update / Delete (cont)

- **Delete**

```
@Override
public void delete(Movie movie) {
   JdbcTemplate template = getJdbcTemplate();
   template.update(
      "delete from MOVIES where MOVIE_ID = ?", movie.getId());
   movie.setId(null);
}
```

  - Upon deletion instance has to be marked as fresh, otherwise a subsequent call to saveOrUpdate would not succeed
  - For the deletion of a movie one could check whether the rented-flag is false, otherwise a foreign key constraint exception will be thrown

```
if(movie.isRented()) throw new IllegalStateException();
```

    - But such a test is already performed in the service implementation

PrimaryKey Generation

- **Problem with auto-increment**

```
CREATE MEMORY TABLE MOVIES(MOVIE_ID BIGINT
   GENERATED BY DEFAULT AS IDENTITY(START WITH 1)
   NOT NULL PRIMARY KEY,
```

  - PK has to be read before it can be assigned to the instance
  - How to access the instance if PK is not known?
- **Determine new PK first**
  - Sequence
    - Read next PK from a sequence
  - Read next value

```
template.queryForLong("select max(MOVIE_ID) from MOVIES")+1;
```

    - Not thread-safe
  - Use a singleton PK factory
    - May use a table where the PKs are stored

PrimaryKey Generation

- **Solution: JDBC 3.0 getGeneratedKey functionality**
  - Works with auto-increment

```
SimpleJdbcInsert insert = new SimpleJdbcInsert(getDataSource())
   .withTableName("MOVIES")
   .usingGeneratedKeyColumns("MOVIE_ID");

Map<String,Object> parameters = new HashMap<String, Object>();
parameters.put("MOVIE_TITLE", movie.getTitle());
parameters.put("MOVIE_RELEASEDATE", movie.getReleaseDate());
parameters.put("MOVIE_RENTED", movie.isRented());
parameters.put("PRICECATEGORY_FK",
                        movie.getPriceCategory().getId());

Number id = insert.executeAndReturnKey(parameters);
movie.setId((Long)id);
```

PrimaryKey Generation

- **Solution: call identity() [HSQL specific]**

```
Connection conn = getDataSource().getConnection();

PreparedStatement pst;
pst = conn.prepareStatement("INSERT INTO MOVIES"
 + "(MOVIE_TITLE, MOVIE_RELEASEDATE, MOVIE_RENTED, "
 + " PRICECATEGORY_FK) VALUES (?,?,?,?)");
pst.setString(1, movie.getTitle());
pst.setDate(2, new Date(movie.getReleaseDate().getTime()));
pst.setBoolean(3, movie.isRented());
pst.setLong(4, movie.getPriceCategory().getId());
pst.execute();

Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("call identity()");
rs.next();
movie.setId(res.getLong(1));
```

equals / hashCode

- **Comparison**
  - Same entity may exist in several instances, not found with contains
    => equals / hashCode must be overridden
- **Implementation of equals & hashCode**
  - Primary Key is unique, but set after creation
    - Entity must not be added to a collection before it is saved or
    - Primary key must not be used in implementation of equals/hashCode
  - hashCode
    - Use final & immutable attributes only (hashCode must not change)
  - equals
    - Compare PKs if both are available (!= null), compare fields otherwise

=> equals/hashCode can be generated by Eclipse

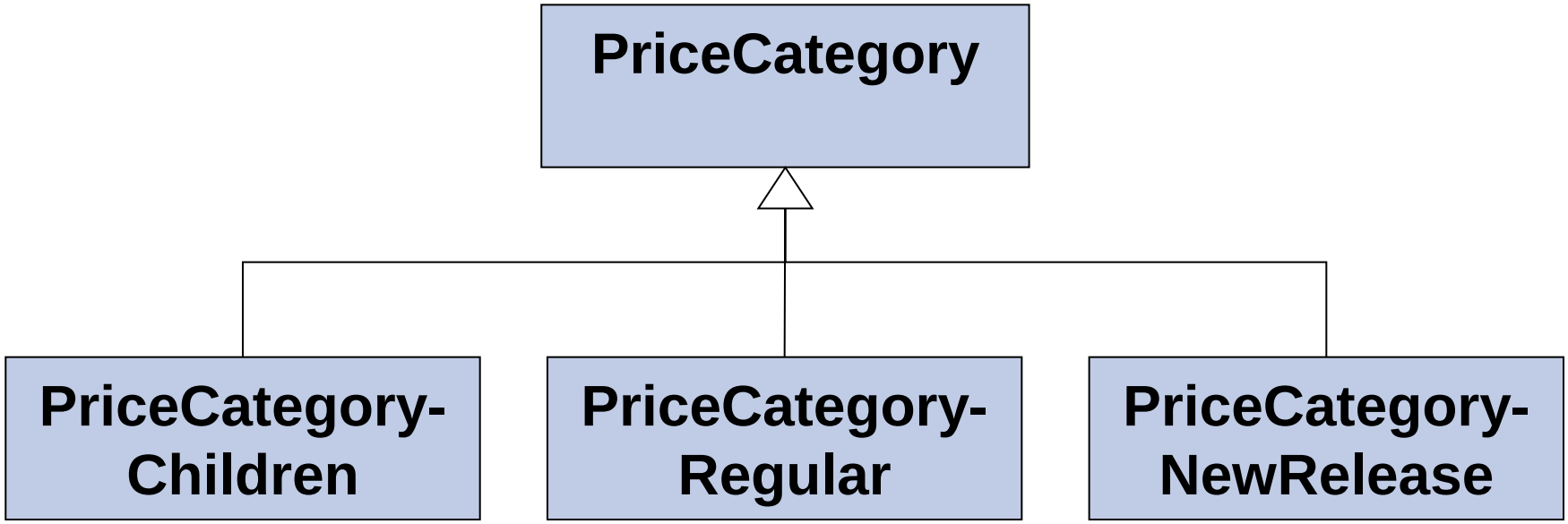
hashCode

```
// Rental.hashCode
public int hashCode() {
   int result = 1;
   …
   result = 31*result + ((user == null) ? 0 : user.hashCode());
   return result;
}
```

```
// User.hashCode
public int hashCode() {
   int result = 1;
   ...
   result = 31*result + ((rentals==null) ? 0 :rentals.hashCode());
   return result;
}
```

- Problem: Cyclic dependency => StackOverflowException

Inheritance



- **Mapping to database?**

Inheritance

```
private PriceCategory createPriceCategory(ResultSet rs)
                                                throws SQLException {
   String type = rs.getString("PRICECATEGORY_TYPE");
   PriceCategory c = null;
   if("Regular".equals(type)) {
      c = new PriceCategoryRegular();
   }
   else if("Children".equals(type)) {
      c = new PriceCategoryChildren();
   }
   else if("NewRelease".equals(type)) {
      c = new PriceCategoryNewRelease();
   }
   c.setId(rs.getLong("PRICECATEGORY_ID"));
   return c;
}
```

- Java7: case statement on type (but type must not be null)

Inheritance

```
private PriceCategory createPriceCategory(ResultSet rs)
                                              throws SQLException {
   String type = rs.getString("PRICECATEGORY_TYPE");
   PriceCategory c = null;
   switch(type) {
   case "Regular" : c = new PriceCategoryRegular();
   case "Children" :  c = new PriceCategoryChildren();
   case "NewRelease" : c = new PriceCategoryNewRelease();
   }
   c.setId(rs.getLong("PRICECATEGORY_ID"));
   return c;
}
```

Cascade Delete

- **Upon deletion of a user, all associated rentals should be deleted**

```
@Override
public void delete(User user) {
   JdbcTemplate template = getJdbcTemplate();
   for(Rental r : user.getRentals()){
      rentalDAO.delete(r);
   }
   template.update("delete from USERS where USER_ID = ?",
                        user.getId());
   user.setId(null);
}
```

  - Do we have to remove the rentals from the deleted instance as well?
    => user.setRentals(new LinkedList<Rentals>());
  - Design decision:
    does such cascading delete belong to the DAO or to the UserService?

Cascade Delete

- **Variant**

```
@Override
public void delete(User user) {
   JdbcTemplate template = getJdbcTemplate();
   template.update("delete from RENTALS WHERE USER_ID=?",
                        user.getId());
   template.update("delete from USERS WHERE USER_ID=?",
                        user.getId());
}
```

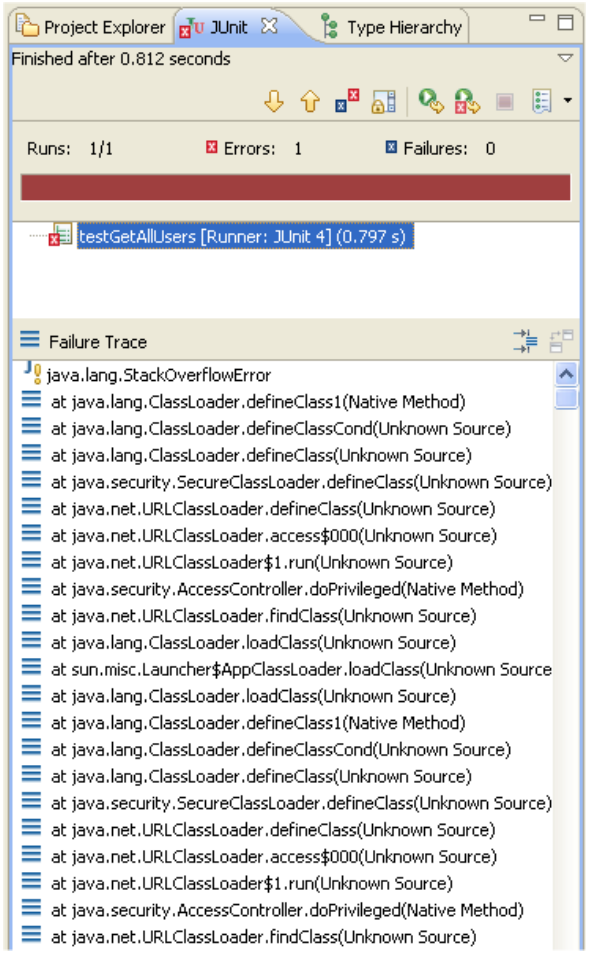
  - This way, UserDAO also accesses RENTALS table

Cyclic Dependencies

- **userDao.getById**
  - u.setRentals(rentalDao.getRentalsByUser(u))
- **rentalDao.getRentalsByUser**
  - createRental (for each rental object)
    - movieDao.getById(rs.getLong("MOVIE_ID"))
    - userDao.getById(rs.getLong("USER_ID"))
- **...**

```
java.lang.StackOverflowError
```

=> general problem of bidirectional associations



Solution 1: Do not load known user

```
@Override
public List<Rental> getRentalsByUser(final User user) {
   JdbcTemplate template = getJdbcTemplate();
   return template.query(
      "select * from RENTALS where USER_ID = ?",
      new RowMapper<Rental>() {
         @Override
         public Rental mapRow(ResultSet rs, int row)
                                              throws SQLException {
            return createRental(rs, user);
         }
      }, user.getId());
}
```

- Disadvantages
  - Has to be implemented individually for each case
  - If a rental is loaded, rental may be added several times to rentals of user

Solution 1: Do not load known user

```
private Rental createRental(ResultSet rs) throws SQLException {
   Long id = rs.getLong("RENTAL_ID");
   User user = userDAO.getById(rs.getLong("USER_ID"));

   //return createRental(rs, user); // do not call this!
   for(Rental r : user.getRentals()){
      if(r.getId().equals(id)) return r;
   }
   throw new RuntimeException("inconsistent user");
}
```

Solution 2: Unification

- **Cyclic Dependencies can be solved with unification**

```
public class ObjectUnifier<T> {
   private Map<Long, WeakReference<T>> cache =
      new HashMap<Long, WeakReference<T>>();
   public T getObject(Long id) {
      if (cache.get(id) != null){
         return cache.get(id).get(); // may be null
      } else {
         return null;
      }
   }
   public void putObject(Long id, T obj) {
      cache.put(id, new WeakReference<T>(obj));
   }
   public void remove(Long id) { cache.remove(id); }
   public void clear() { cache.clear(); }
}
```

Solution 2: Unification

- **createUser**
  - Calls rentalDAO.getRentalsByUser

```
private User createUser(ResultSet rs) throws SQLException {
   Long id = rs.getLong("USER_ID");
   User u = cache.getObject(id);
   if (u == null) {
      u = new User(rs.getString("USER_NAME"),
                   rs.getString("USER_FIRSTNAME"));
      u.setId(id);
      u.setEmail(rs.getString("USER_EMAIL"));
      cache.putObject(id, u);
      u.setRentals(rentalDAO.getRentalsByUser(u));
   }
   return u;
}
```

Solution 2: Unification

- **createRental**
  - Calls userDAO.getById

```
private Rental createRental(ResultSet rs) throws SQLException {
   Long id = rs.getLong("RENTAL_ID");
   Rental r = cache.getObject(id);
   if (r == null) {
      r = new Rental();
      r.setId(id);
      cache.putObject(id, r);
      r.setMovie(movieDAO.getById(rs.getLong("MOVIE_ID")));
      r.setUser(userDAO.getById(rs.getLong("USER_ID")));
      r.setRentalDays(rs.getInt("RENTAL_RENTALDAYS"));
   }
   return r;
}
```

Solution 2: Unification

- **Consequences**
  - equals / hashCode must not necessarily be overwritten
  - Cache has to be cleared, e.g. with an aspect

```
@Aspect
public class CacheAspect {
   private int level = 0;

   @Before("execution(* *..*Service.*(..))")
   public void enterService(){ level++; }

   @After("execution(* *..*Service.*(..))")
   public void exitService() { level--;
      if (level == 0) { cleanup(); }
   }

   private void cleanup(){ ... }
}
```

Summary: Problems which had to be solved

- **Primary Key Generation**
- **Mapping of inheritance**
- **Cyclic Structures**
- **Equals & hashCode**
- **Dependent objects**
  - Deletion of dependent objects
  - Update of dependent objects ???
  - Insert of dependent objects ???

=> JPA addresses all these aspects