Relational Database Systems I

Wolf-Tilo Balke
Hermann Kroll, Janus Wawrzinek, Stephan Mennicke

Institut für Informationssysteme
Technische Universität Braunschweig
www.ifis.cs.tu-bs.de
13 Object Persistence

• Persistence
  – Object Persistence
  – Manual Persistence
  – Persistence Frameworks

• Generating IDs
• Persistence Frameworks
• Object Databases
Within typical software engineering cycles, application data models are developed
- data models are used *internally* in application, following its design and programming paradigms
- nowadays, most application data models are *object-oriented*
- also often called *domain model*

When developing an application using a database there always is one huge problem
- how do you map your domain data model to your database data model?
- impedance mismatch!
13.1 Object Persistence

Application Data Model

- Villain
  - id: int
  - name: string
- Mad Scientist
  - scientific field: varchar
- Super Villain
  - Super power: varchar
- Evil Mad Super Scientist

Database Data Model

- Application
- RDBMS

Villain
id : int
name : string
Mad Scientist
scientific field : varchar
Super Villain
Super power: varchar
Evil Mad Super Scientist
{overlapping}
13.1 Object Persistence

• Model mapping is hard for object-oriented programming languages!
  – object model differs significantly from the relational model

• In most cases, this leads to the fact that developers adapt their domain model to the used database
  – but good software engineering demands that your domain model follows your business needs, not the needs of the underlying storage technology!
13.1 Object Persistence

• The object model
  – objects represent objects in the real world
  – thus, objects have a state and a behavior
  – e.g. This car is blue and has 33 PS, it can drive and honk the horn.

• Besides state and behavior, objects may have complex relationships to other objects
  – usually, all relationships we already know from UML are possible
    • generalization, specialization, aggregation, composition, association, etc.

  – example
    • A car may have 4 wheels and 2 doors.
    • When the car drives, also the wheels are moving and rotating.
13.1 Object Persistence

• Application objects in an application only “live” throughout the lifetime of the application
  – they are transient

• However, you may want to use a database to store the state of an object
  – thus, the object may persist beyond the application’s termination
  – object may thus be retrieved later or shared among programs

• Permanently storing the state of an object is called object persistence
  – which also includes restoring the object
• When your object data is persistent, you may
  – exchange it with other applications
  – inspect it manually
  – continue using it when you restart your application
  – replicate and aggregate it
  – ...

13.1 Object Persistence
13.1 Object Persistence

- There is a problem with object persistence in relational DBMS.
- Each object has a unique identity that must be preserved.
  - Objects have a unique implicit immutable identity independent of their state (values).
    - usually, the identity cannot be accessed. It just “is.”
  - In the relational model, rows are explicitly identified by their values.
    - no duplicates
    - the only safe assumption: all columns taken together form a key
  - To make objects persistent in a RDBMS, an explicit identity (key) needs to be generated.
    - so called unique identifier (UID)
• Objects may have complex relationships (object structure)
  – as a whole, this structure is **hard to capture** in a **RDBMS**
  – remember: you only have tables, columns and rows
  – objects needs to be **disassembled** and stored in **multiple tables** linked by foreign key relationships
  – think of this:
    *Every time you want to park your car in the garage, you have to disassemble it and assemble it when you want to drive to work…*
13.1 Object Persistence

- **Example**: store following information in a RDB
  - a hero called **Ms. Marvel** has the powers to **fly** and **shoot energy bursts** out of her hands
  - another hero also called **Ms. Marvel** has the powers of **super human strength** and **durability**
  - **problem**: how to store this information in tables (in an extensible and flexible way)?

<table>
<thead>
<tr>
<th>Name</th>
<th>Power 1</th>
<th>Power 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ms. Marvel</td>
<td>Flight</td>
<td>Shoot energy</td>
</tr>
<tr>
<td>Ms. Marvel</td>
<td>Strength</td>
<td>Durability</td>
</tr>
</tbody>
</table>
This problem is commonly known as the **multi-valued data types problem**

- a more popular version of this is the **bill-of-material problem**
- most solutions rely on introducing new surrogate keys

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>heroid</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ms. Marvel</td>
<td>1</td>
<td>Flight</td>
</tr>
<tr>
<td>2</td>
<td>Ms. Marvel</td>
<td>2</td>
<td>Strength</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Durability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Shoot energy</td>
</tr>
</tbody>
</table>
13.1 Object Persistence

• Providing object persistence is a complex task
  – provide means to create, read, update, and delete persistent objects
    • called the CRUD operations
  – for each of these operations, respect the object’s identity and structure
    • create explicit identities if necessary
    • break object structure into relationships among entities
Of course, persistence can be achieved manually by using tools we already know—e.g. SQL and Java/JDBC—but this approach often is cumbersome and takes a lot of effort.

**Example:** let’s make a simple JavaBean persistent:

<table>
<thead>
<tr>
<th>Villain</th>
<th>1</th>
<th>*</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>realName : String</td>
<td></td>
<td></td>
<td>name : String</td>
</tr>
</tbody>
</table>
13.1 Manual Persistence

• **Beans**

Villain.java

```java
public class Villain {
    String realName;
    Set<Alias> aliases;
    // some getter/setter methods
}
```

Alias.java

```java
public class Alias {
    String name;
    // some getter/setter
}
```

This is all the application actually needs!

• **Relational tables**

  – beans have **no primary keys**
  (names are not unique)
  
  • generate a key somehow…

Villain

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
<td></td>
</tr>
<tr>
<td>v_uid</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
</tbody>
</table>

Alias

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
</tbody>
</table>

foreign key

primary key
13.1 Manual Persistence

CRUD create operation

Villain.java

```java
int uid;
public void createPersistent(Connection conn) throws SQLException {
    uid = magicallyCreateAnUID(this);
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("INSERT INTO villains (" + uid + "," + realName + ")");
    for (Alias alias : aliases) {
        alias.createPersistent(this, conn);
    }
}
```

Alias.java

```java
int uid;
public void createPersistent(Villain villain, Connection conn) throws conn {
    uid = magicallyCreateAnUID(this);
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("INSERT INTO alias (" + uid + "," + villain.uid + ", " + name + ")");
}
```

Take care of structure

Take care of identity
13.1 Manual Persistence

• Note: The generated **UID** is not related to the *real* object identity in Java (i.e. the *reference ID*)
  
  – the **reference ID** is important e.g. to build data structures at runtime but has no semantical meaning otherwise
  
  – the **generated UID** is important to identify and connect rows in the DBMS and also has no semantical meaning
  
  – although the **UIDs** does not contain relevant information, the application **has to maintain** it
• CRUD read operation
  – for reading an object, you need to know its UID
  – for accessing the full object structure, you have two options
    • eager loading:
      loads the whole object structure as soon as the base object is requested
      – good performance if you usually need the whole structure
    • lazy loading:
      loads only the base object when requested, loads remaining parts of the object’s structure when they are needed
      – good performance if you usually only need parts of the structure
13.1 Manual Persistence

**CRUD read operation (lazy)**

Villain.java

```java
public static Villain readPersistent(int uid, Connection conn) throws SQLException {
    Statement stmt = conn.createStatement();
    Villain villain = new Villain();
    ResultSet rs = stmt.executeQuery(
        "SELECT * FROM villains WHERE uid=" + uid);
    if (rs.next()) {
        villain.uid = uid;
        villain.realName = rs.getString("realName");
        villain.aliases = null;
    } else {
        // handle case when uid does not exist
    }
    return villain;
}
```

You need the UID, somehow…
13.1 Manual Persistence

CRUD read operation (lazy)

Villain.java

```java
public Set<Alias> getAliases() throws SQLException {
    if (aliases == null) {
        aliases = new HashSet<Alias>();
        Statement stmt = ... // create statement somehow
        ResultSet rs = stmt.executeQuery("SELECT * FROM alias WHERE v_uid= " + uid);

        while (rs.next()) {
            Alias alias = new Alias();
            alias.uid = rs.getInt("uid");
            alias.name = rs.getString("name");
            aliases.add(alias);
        }
    }
    return aliases;
}
```
13.1 Manual Persistence

- **CRUD update**
  - for updates, there are also various implementation approaches

- **Immediate updates**
  - directly persist updates after they occur
    - embed JDBC calls directly in each beans set methods
  - performance might be very low
  - only few problems with transactional consistency
13.1 Manual Persistence

- **explicit deferred updates**
  - collect updates and apply them in batches
    - every $x$ seconds, after $n$ updates or at the end of a bigger procedure
  - JDBC provides dedicated batch update/insert methods

- **implicit deferred updates**
  - mark object as *dirty*, when its content has been changed
  - provide an additional method for updating the Beans content in the DB if it is dirty
    - identify and update *dirty objects* periodically
    - traverse the whole *object structure recursively* and also write all dirty, related objects

- deferred updates perform better, but they may cause problems with transactional consistency
• **Summary**
  
  – a considerable amount of **work is necessary**
  
  – there a **many fetching and update schemes** with different properties and performance impacts
    
    • lazy load vs. eager load
    
    • immediate update vs. explicit update
    
    • ...
  
  – there are several problems involved
    
    • generating **IDs** and propagating them
    
    • dealing with the **object structure**
    
    • keeping **transactional consistency** in multi-user scenarios
    
    • providing **sufficient performance**
    
    • how to perform **complex querying**?
Another major drawback is **boilerplate code**

- the code for the data model is littered with code dealing with persistence
  - most of it is trivial and could be generated
  - makes the data model **dirty** and **inflexible**

- in this example, the initial Villain.java model grew from **26 lines** of code to over **180 lines** of code
  - even though the implementation is very simple
• **Persistence frameworks** come to rescue…

• Main idea

  – providing **persistence** for an application should be **as easy as possible**
    
    • programmer should not spend too much time and code on these issues
      
      – concentrate on more important things!
    
    • provide automated support to problems of persistence

  – persistence handling should be **transparent**
    
    • just model your application data the way **you** need it, and not the way your DBMS needs it
    
    • keep your models clean!
During software development and maintenance, complexity is the archenemy.

A popular approach for reducing software complexity are layered architectures:
- each layer has a defined responsibility and communicates with other layers using clearly defined interfaces.
- usually, no code from one layer should spill into another.
- implementations of a layer may change without affecting the others.
  - e.g. web interface vs. web service interface
  - E.g. persistence using a RDBMS vs. persistence using XML
A layered architecture for applications using persistent object might look like this:

- **presentation layer:** present data to the user and interact with him/her
- **business layer:** contains the domain data model and the business logic
- **persistence layer:** stores and retrieves objects of the domain data model
13.1 Persistence Frameworks

- **Persistence frameworks** aim at providing the complete persistence layer
  - in the best case, there are only very few things an application programmer needs to do
  - domain model stays clean
  - usually, information needed by the persistence layer is provided using **meta data**
  - however, you buy these features with a **performance penalty**
13 Object Persistence

- Persistence
  - Object Persistence
  - Manual Persistence
  - Persistence Frameworks

- Generating IDs
- Persistence Frameworks
- Object Databases
As you have seen, identifying data is important.

In the good old days of **punch cards** and **magnetic tapes**, identifying data was easy...

- each punchcard had a number that sequentially increased from card to card
  - card No. 1, Card No. 2, ...
  - the same works for tapes and similar devices

Using sequential identifiers also worked great within the **hierarchical data model** and **network data model**.
13.2 Generating IDs

- The **relational data model** changed the view upon data
  - data is organized as a **set of tuples**
  - each tuple is identified by its **value**
    - explicit identification is not mandatory!
  - a small subset of attributes is selected as **primary key** for easier **identification** and **reference**

- The model was designed to be used with **natural keys**
  - the key values are part of the data domain
    - each student already has a unique matriculation number, so just use it as key
    - imagine, you have a set of weather stations. Each reading can be uniquely identified by its time, type, and station
• But what happens if your data does not provide a useful natural key?
  – create a synthetic key yourself (so-called **surrogate keys**)

• When performing **object-relational mapping**, this problem frequently arises since each object always needs a **unique ID**
  – **real ID** usually hidden within the OS or VM
  – **surrogate Keys** are needed!
13.2 Generating IDs

• How to create surrogate keys?
There are several approaches:

  – sequence keys
    • full sequences keys
    • hi-low sequences

  – UUIDs
    • time & device-based UUIDs
    • hash-based UUIDs
    • random UUIDs
13.2 Sequences

- **Full sequenced keys**
  - A **central authority** (usually the DBMS) provides all keys
  - The keys follow a strict sequence
    - E.g. 1, 2, 3, 4, 5, …
    - Usually, there is one sequence per table
- **Pros**
  - Keys are short and easy to debug
    - E.g. ... WHERE id = 12
  - Easy to handle and storage efficient, indexes can grow with less need for restructuring (ordered inserts!)
- **Cons**
  - Key is provided centrally by the DBMS → **Bottleneck**!
  - For each single key, you need to connect to the DB to retrieve it → **Network traffic**
13.2 Sequences

• Most databases supporting this feature use sequence tables
  – for each sequence to be generated there is a single-column/single-row table containing an integer that is continuously incremented

• Define a table using an auto-incremented key
  – Oracle: the sequence needs to be specified explicitly
    • use the CREATE SEQUENCE statement
    • when you want to insert a new row with a surrogate key, call the sequence using `<seq_name>.nextval` within the INSERT statement
13.2 Sequences

- **MySQL**: declare it in the column definition of the `CREATE TABLE` statement
  - `colName dataType NOT NULL AUTO_INCREMENT`

- **DB2**: like MySQL
  - `colName dataType NOT NULL GENERATED ALWAYS AS IDENTITY`

- **Example DB2**
  ```sql
  CREATE TABLE hero(
    id INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY,
    name VARCHAR(255)
  )

  INSERT INTO hero(name) VALUES ('The Hulk')
  
  - Note that you do not know what key the DB assigned to your new row!
  - Use the JDBC method `getGeneratedKeys` for returning your key
13.2 Hi-Low Keys

• Hi-low keys
  – central authority (DBMS) and application share responsibility for key creation
  – idea: key is made up of two parts
    • high: provided by the DBMS
    • low: provided by the application
  – every time an application connects to the DBMS, it receives a unique high value for that session
    • usually derived using some sequence within the DBMS
  – the application creates its own sequence for that session and increments it for each needed low value
  – key is created by concatenating the high and low part
13.2 Hi-Low Keys

– pros

• just one DBMS access per user/application session
  → reduces network traffic, rarely any bottlenecks
• keys are still of manageable size
• full key is immediately known to the application
  – very important for OR-mapping!

– cons

• application is responsible for finally creating the key
  (no plain easy auto-incrementing columns)
• still communication with the DBMS necessary
13.2 UUIDs

- **Universally unique identifiers (UUID)**
  - standard provided by the Open Software Foundation (OSF) for unique surrogate keys and IDs
    - later: IETF RFC 4122 and ISO/IEC 9834-8:2005
  - **base idea:**
    generate unique keys **without any central control**
  - most popular implementations
    - Microsoft Globally Unique Identifiers (GUID)
    - Ext2/Ext3 file system identifiers
13.2 UUIDs

• A UUID is a 128-bit number
• Usually represented by 32 hex digits in 5 groups of lengths 8, 4, 4, 4, and 12
  – e.g. 4e84890a-5f12-42fd-b1fe-0d32afb1d9d8
• There are 5 defined types of UUID algorithms
  – you can identify the used algorithm by inspecting the first digit of the third hex block (in red)
13.2 UUIDs

• Type-1 UUIDs

  – UUID is a concatenation of the MAC address of the generator host and the number of 100 ns intervals since February 24, 1582 (introduction of the Gregorian calendar) with some additional bit shifting

  • you can identify when and by whom the key has been generated

  • you can catch evil, but stupid people by their Type-1 UUIDs
• The Melissa Virus
  – self-replicating macro virus using vulnerabilities in Outlook
  – melissa virus shut down large parts of the internet in March 1999
  – however, the code contained some UUIDs left by its author…
  – by reversing the UUID, its creator David L. Smith could be back-traced and was sentenced to 10 years
13.2 UUIDs

• **Type-2 UUID**
  – like Type-1, but parts of the timestamp are replaced by the POSIX UID domain and user ID
    • privacy issues even more severe

• **Type-3 UUID**
  – everything but the type digit is generated by a MD5 hash function
  – input is usually a URL, object identifier, etc…
    • problems in generating truly unique IDs

• **Type-5 UUID**
  – like type 3, but uses SHA-1 hashing
    • SHA-1 is a stronger and more efficient hash function
• Type-4 UUID
  – all digits but the type digit are randomly generated
    • a cryptographically secure random number generator is needed
  – collisions may occur, but are highly improbable
    • for $2^{36}$ UUIDs (~68 trillion), collision probability is $4 \cdot 10^{-16}$
    • for $2^{46}$ UUIDs (~70 quadrillion), probability is $4 \cdot 10^{-10}$
    • When you generate 1 billion UUIDs every second over the next 100 years, then the probability for at least one collision is 50%.
13.2 UUIDs

• Is using UUIDs a good idea?
  – depends…
  – **good**
    • UUIDs can be generated very easily without central control and without support from the DBMS
    • when using UUIDS, you can easily integrate data from different data sources
      – no key collisions
    • performance of queries not affected if used correctly
  – **bad**
    • UUIDs are **horrible to debug**
      – `SELECT ... FROM ... WHERE id = 8ac7fb3d4f4047419c7f7d22d1802fe3`
    • usually, more storage space is needed
13 Object Persistence

• Persistence
  – Object Persistence
  – Manual Persistence
  – Persistence Frameworks

• Generating IDs

• Persistence Frameworks

• Object Databases
Currently, there are many products available

We will focus on

- **JPA (Java Persistence API):** default API provided by SUN JAVA EE 5
- **Hibernate:** most popular Java persistence framework
Before the introduction of JPA (Java Persistence API), there were a multitude of persistence frameworks:

- Hibernate
- Apache OJB
- Apache Castor
- JPOX
- XORM
- Persist
- Oracle Toplink
- ...

Each of these frameworks had their own proprietary API and metadata format.
13.3 JPA

• JPA provides a common interface for relational persistence frameworks written in Java
  – released in **May 2006** by Sun Microsystems
    • latest update (version 2.1) in 2013
  – part of **EJB 3.0 specification** of Java EE 5.0 standard
    • unofficial replacement/alternative for EJB CMP (Enterprise Java Beans Container Managed Persistence – a really complex and tedious persistence mechanism)

• **Consist of three components**
  – the actual **API** in **javax.persistence**
  – the **JPQL query language**
  – facilities for handling **metadata**
Sun was able to convince most persistence framework providers to adopt the API

- JPA implementations are called persistence providers
- Most important persistence providers:
  - EclipseLink (JPA 2.0 reference implementation)
  - Hibernate
  - Apache OpenJPA
Main idea:
Use of POJOs as the application’s domain data model
  – POJO: Plain old Java objects, that is, JavaBeans without any complex stuff in them

Metadata describes how the POJOs are mapped to the relational DB
  – metadata as annotations: The POJO is enriched using JSR 175 annotations (Java 5 and beyond)
    • very easy, but domain model needs to be directly annotated
  – metadata as XML: In addition to the POJO domain model, there are XML files describing the OR mapping
    • more complex, more effort needed
    • strong separation of business and persistence layer
Using annotations, the development workflow looks like this:

1. annotate all persistent entities
   - i.e. those classes that are supposed to be stored in an own table
2. annotate either all attributes or all getter/setters of your persistent entities
   - define what attributes contain UIDs
   - define relationships
   - define special persistence behavior
3. provide an XML document describing your persistence units
   - which entities should end up in which database?
4. use the JPA EntityManager to create/read/update/delete your persistent objects
### Beans

Villain.java

```java
public class Villain {
    String realName;
    Set<Alias> aliases;
    // getters/setters...
}
```

Villain.java

```java
public class Alias {
    String name;
    // getters/setters...
}
```

### Relational tables

<table>
<thead>
<tr>
<th>Alias</th>
<th>uid</th>
<th>v_uid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Villain</td>
<td>uid</td>
<td></td>
<td>realName</td>
</tr>
</tbody>
</table>

- **Primary key**: `uid`
- **Foreign key**: `v_uid`
### 13.3 JPA

#### Annotate persistent entities

**Villain.java**

```java
@Entity
public class Villain {
    String realName;
    Set<Alias> aliases;
    // getters/setters
}
```

**Alias.java**

```java
@Entity
public class Alias {
    String name;
    // getters/setters
}
```

#### Add and annotate UID columns

**Villain.java**

```java
@Entity
public class Villain {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    int uid;
    String realName;
    Set<Alias> aliases;
}
```

**Alias.java**

```java
@Entity
public class Alias {
    @Id
    int uid;
    String name;
}
```
Annotate attributes and relationships

Villain.java

```java
@Entity
public class Villain {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    int uid;
    String realName;
    @OneToMany(fetch = FetchType.LAZY)
    Set<Alias> aliases;
}
```

Alias.java

```java
@Entity
public class Alias {
    @Id
    int uid;
    String name;
    @Transient
    String nonsense;
}
```

- all simple attributes are **persisted by default** unless annotated with `@Transient`
- all collections need an annotation defining the relationship (one-to-many, many-to-one, many-to-many)
  - provide the desired name of the **foreign key column**
  - optionally, also provide **loading** and **updating strategies**

"Use lazy loading"

"Ignore this attribute"
• Usually, the persistence provider automatically generates the relational schema
• That is fine, when persistence is added to an application
  – However, you often have existing data (and thus an existing schema) to adapt to
    • How to get the persistence provider to automatically generate the exact schema that you already have?
    • use advanced annotations for this task
    • tends to be very tedious
Advanced annotations cover the following:

- indexing
- structural design
  - naming, embedding vs. linking, splitting and merging of object structures, etc.
- IDs
  - natural ids, UUIDS, sequence-based ids, etc.
- custom data types
- update and delete behavior
  - cascading, Restricting, etc.
- constraints
- computed values
- sorting
- locking
- caching
- filtering
- etc.
• To actually store and retrieve entities you need an **EntityManger**
  
  – use the **Java way** to get it:
  
  • create an XML File, defining (and naming) a **persistence-unit**
  
  • use the **Persistence-class** to create an **EntityManagerFactory** (using name of the persistence-unit)
  
  • use the **EntityManagerFactory** to create the **EntityManager**
13.3 JPA

- The XML file (persistence.xml) might look like this:

```xml
<persistence>
  <persistence-unit name="heroes">
    <!-- Define a persistence provider e.g. Hibernate -->
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <!-- Define classes to be mapped -->
    <class>de.ifis.heroes.Villain</class>
    <class>de.ifis.heroes.Alias</class>

    <!-- Properties for your JDBC driver and additional setup -->
    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.ibm.db2.jcc.DB2Driver" />
      <property name="javax.persistence.jdbc.url" value="jdbc:db2://dmlab.ifis.cs.tu-bs.de:50000/DBLAB" />
      <property name="javax.persistence.jdbc.user" value="simon" />
      <property name="javax.persistence.jdbc.password" value="123" />
    </properties>
  </persistence-unit>
</persistence>
```
factory = Persistence.createEntityManagerFactory("heroes");
EntityManager em = factory.createEntityManager();

em.getTransaction().begin();

//create a new villain
Villain newVillain = new Villain();
newVillain.setRealName("Gala");
newVillain.addAlias(new Alias("Galactus"));
newVillain.addAlias(new Alias("Eater-of-Worlds"));
em.persist(newVillain);

//update Galactus
newVillain.setRealName("Galan");
em.persist(newVillain);

em.getTransaction().commit();
em.close();
13.3 JPA

• Using the **EntityManager**, you may also define **JPQL queries** to retrieve objects
  – similar to SQL, but uses persistent entities and their attributes instead of tables

```java
Query q = em.createQuery(
    "SELECT v FROM villains AS v WHERE v.realName='Galan'"
);
List<Villain> villain = (List<Villain>) q.getResultList();
```
• Ups and Downs of Java Persistence
  – Does it work?
    • YES!
  – But:
    • It costs some performance
    • Still, it's not really “easy”
      – Likely easier than doing everything manually
    • Sometimes, it can be confusing and hard to debug
      – Persistence tends to produce some obscure error messages
13 Object Persistence

- Persistence
  - Object Persistence
  - Manual Persistence
  - Persistence Frameworks
- Generating IDs
- Persistence Frameworks
- Object Databases
13.4 Object Databases

- **Objects databases (ODBMS)** came up for the first time around the mid-80s
  - driven by the increasing popularity of **object-oriented programming languages**
  - **promise**: get rid of the annoying object-relational impedance mismatch
    - store objects in all their complexity, do not match them to tables, etc.
      - in theory, extremely high performance possible
    - most programmers loved that idea
13.4 Object Databases

- Object databases directly interacted with the programming language, thus developing applications should become very easy.

- The first wave of commercial products in the mid-90s was extremely hyped
  - Gemstone, O2, Versant, Jasmine, Matisse, Objectivity, ObjectStore, Caché, etc.

- Standard committees proposed various ODBMS standards
  - Object Database Management Group (ODMG)
  - Object-extensions in SQL-99
  - ...
• Unfortunately, most ODBMS spectacularly failed
  – products unfinished and unpolished
  – crappy performance due to misuse
  – obscure and highly proprietary APIs, standards, and query languages
  – integration with legacy systems was very hard
  – most vendors went out of business…
  – ODMG closed in 2001
• However, in the last few years, ODBMS gained momentum again
  – software is more and more developed by smaller companies in **smaller projects** within shorter time
  – **agile programming** on the rise
  – object-relational mapping becomes increasingly expensive
    • consumes too much time in development and production
  – newer ODBMS mainly focus on persisting the object state and don’t include the behavioral aspects
• New generation of ODBMS does not aim to replace RDBMS, but provide alternatives for certain areas
  – embedded databases
  – mobile databases
  – real-time systems
    • telecommunications
  – scientific databases (and all other databases storing highly complex data structures)
    • databases in Physics, biology, chemistry, etc.
    • currently, the world’s largest database is a ODBMS (several petabyte, produced by the Stanford Linear Accelerator Experiment)
Objects databases are not a great choice

- when your data needs to be accessed by other applications which are not specially developed for ODBMS
  - today’s ODBMS are still quite proprietary
  - data is just stored somehow (encapsulation)

- when your data has tabular nature
  - true for most accounting data
  - in short: everything you could (in theory) easily do in a huge Excel sheet does not belong into an ODBMS

- when you demand that your DBMS computes statistics or produces complicated aggregations
  - ODBMS are all-or-nothing: You will get only those objects you put into it earlier.
    - Still, some specialized products will try to provide data analytics like Versant Object DB
This be true, but object databases are still an interesting and maybe even a very good alternative to consider. It depends on your task and your data if they are a good idea. If your use case is suitable for an ODBMS, you can achieve spectacular performance paired with easy development and management.

“…we’ll simply observe that object databases haven’t been widely adopted and that it doesn’t appear likely that they will be in the near future.”

Gavin King, 2007
13.4 Object Databases

• Notable Example:
  – ObjectDB, Versant Object Database
  – Usually accessed using JPA or JDO standards
    • …or additionally using some older access techniques / query languages
Object databases are falling out of fashion, being cannibalized by “NoSQL” systems

NoSQL and NewSQL Databases!

- Why the hype?
- What is in there for us?
- Does the hype actually make sense?

Example:

- CouchDB
- MongoDB