ifis

Institut für Informationssysteme
Technische Universität Braunschweig

Relational Database Systems I

Wolf-Tilo Balke
Simon Barthel

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

13 Object Persistence

- General problem
- Manual persistence
  - Generating UIDs
- Persistence frameworks
  - JPA
- Object databases
  - db4o

Exam Stuff

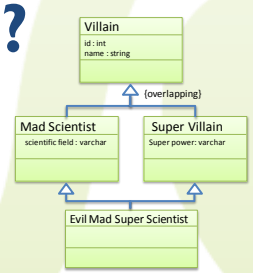
- Written exam on March 1, 2013 at 11:30
  - Duration: 90 minutes
  - BI83, SN22.1, SN23.2
- Remember the 50% homework rule!
  - Who already took the exam doesn't need 50% (at least this time)
  - New Bachelor Prüfungs Ordnung (BPO)
    - Exercises = Studienleistung
  - Old Bachelor Prüfungs Ordnung (BPO)
    - Exercises are required for exam

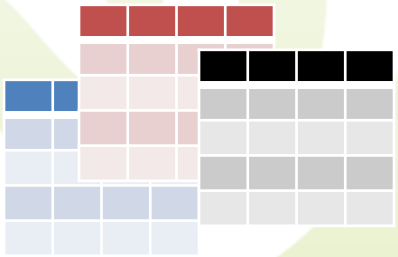
Exam Stuff

- You may bring **two sheets** of A4 paper with you
  - You decide what is written on these sheets
  - You may write on the **front and back** of these sheets
- **Important:**
  The sheets must be **written on by hand**
  - This means: **No printouts, no copies!**
  - Otherwise, we will confiscate the sheets
- The exam will be about:
  - **All topics of the lecture** (excluding detours)
  - **All homework exercises**
  - **Detours** covered in **homework** exercises are **relevant** for the exam!

13.1 Object Persistence

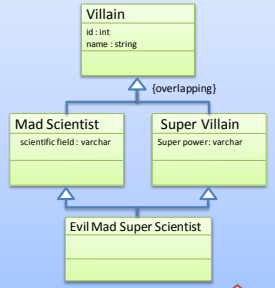
- Within typical software engineering cycles, usually **application data models** are developed
  - Data models are used **internally** in application, following its design and programming paradigms
  - Nowadays, most application data models are **object-oriented**
  - Also often called **domain model**
- When developing an application using a database there always is one huge problem
  - **How do you map your domain data model to you database data model?**
  - **Impedance mismatch!**



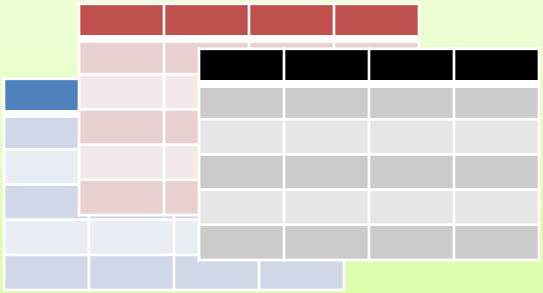


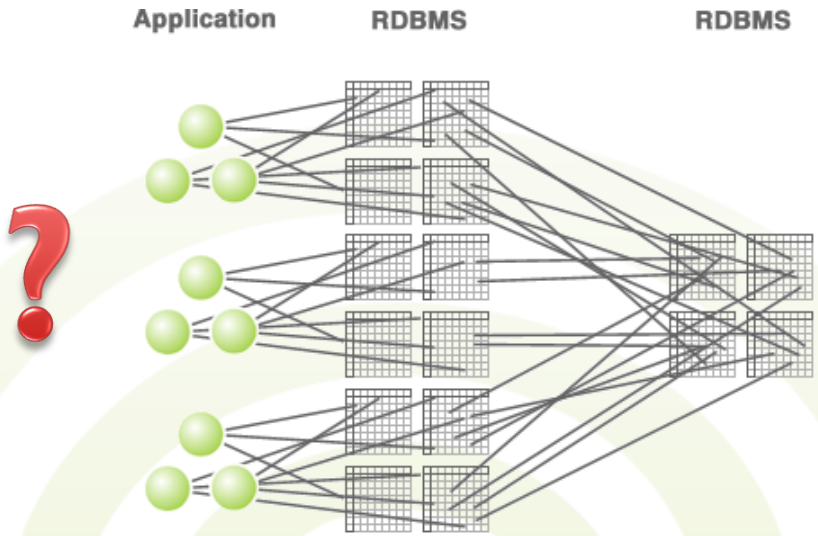
13.1 Object Persistence

Application Data Model



Database Data Model





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**
  - Example: This car is **blue** and has **33 PS,** it can **drive** and **honk the horn**

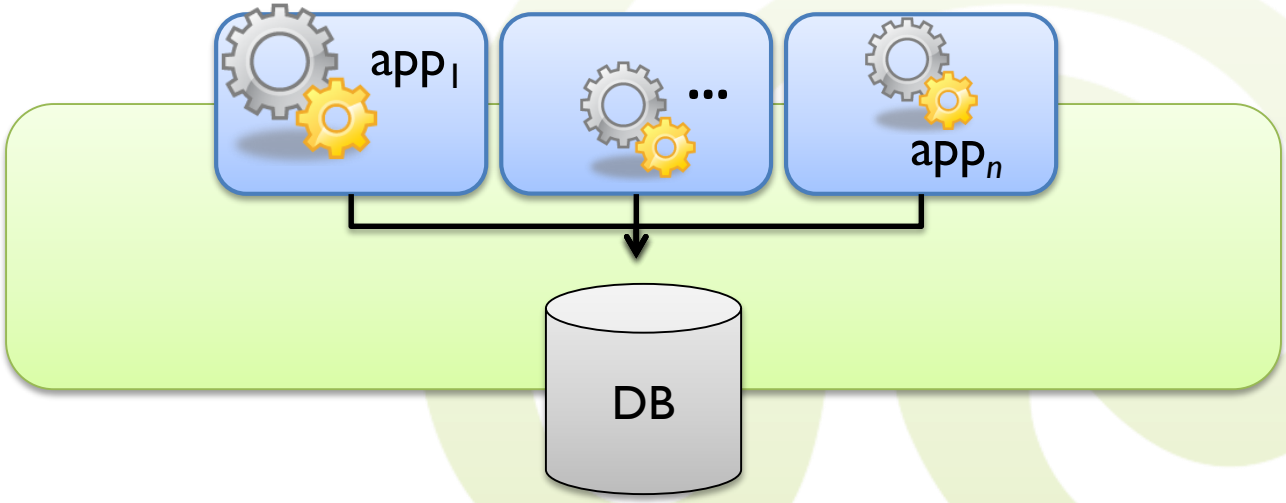
- Besides state and behavior, objects may have **complex relationships** to other objects
  - Usually, all relationships we already know from UML are possible
    - **Generalization, specialization, aggregation, composition, association,** etc.
  - **Example:**
    - A car may have 4 wheels and 2 doors
    - 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 **re**storing the object

13.1 Object Persistence

- 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
  - …



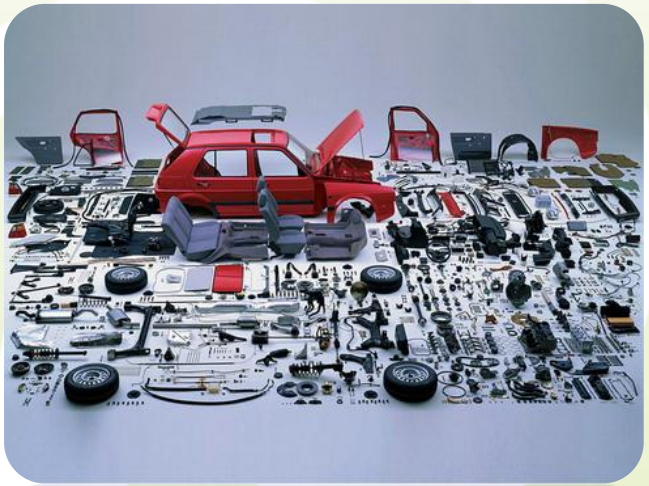
13.1 Object Persistence

- 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)
    - However, 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)**

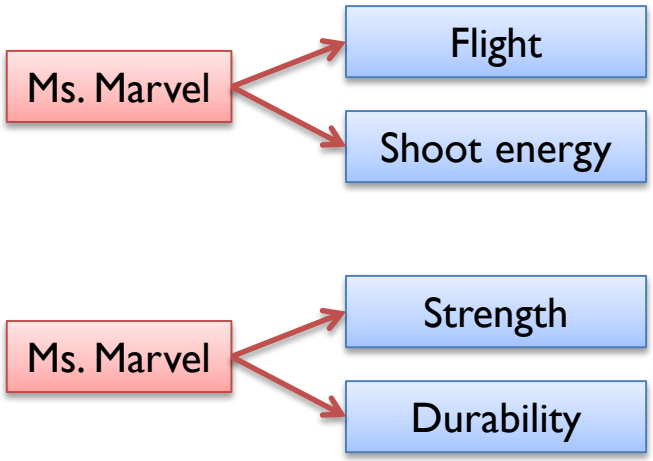
13.1 Object Persistence

- 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)?



Name
Ms. Marvel
Ms. Marvel

Power
Flight
Strength
Durability
Shoot energy

Name	Power 1	Power 2
Ms. Marvel	Flight	Shoot energy
Ms. Marvel	Strength	Durability

13.1 Object Persistence

- 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

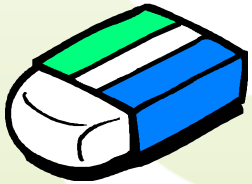
id	Name
1	Ms. Marvel
2	Ms. Marvel

herold	Power
1	Flight
2	Strength
2	Durability
1	Shoot energy

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

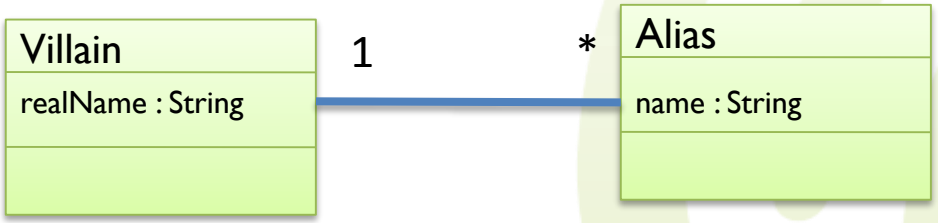


13.2 Manual Persistence

Detour

- Of course, persistence can be achieved **manually** by using tools we already know
  - Example: **SQL** and **Java/JDBC**
  - However, this approach often is cumbersome and takes a lot of effort
- **Example:** Let's make a simple JavaBean persistent



13.2 Manual Persistence

Detour

This is all the application actually needs!

- **Beans**

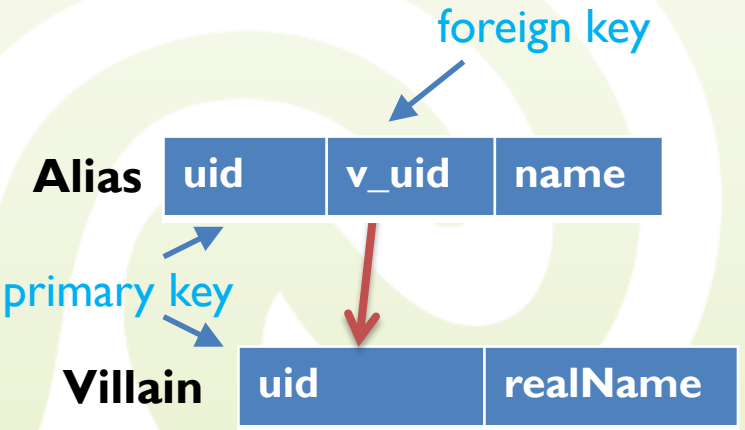
Villain.java

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

Alias.java

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

- **Relational tables**
  - Beans have **no primary keys** (names are not unique)
    - Generate a key somehow…



13.2 Manual Persistence

CRUD create operation

Take care of identity

Villain.java

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

Take care of structure

Alias.java

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

13.2 Manual Persistence

Detour

- **Note:** The generated **UID** is usually not related to the "real" object identity (i.e. internal Java ID)
  - However, it serves the same purpose of **identifying** and connecting related **rows** in the RDBMS
  - Although the **UIDs** do not convey information that is needed within the application's data model, it usually **must be carried along** to make the object persistence work

13.2 Manual Persistence

Detour

- **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

13.2 Manual Persistence

CRUD read operation (lazy)

You need the UID, somehow…

Villain.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);
    rs.next();
    villain.uid = uid;
    villain.realName = rs.getString("realName");
    villain.aliases = null;
    return villain;
}
```

Note:
Additional method besides the “normal” JavaBean methods

13.2 Manual Persistence

CRUD read operation (lazy)

Note: This one is embedded in a "normal" JavaBean accessor method!

Alias.java

```
public Set<Alias> getAliases() throws SQLException {
    if (aliases == null) {
        aliases = new HashSet<Alias>();
        Statement stmt = ... // get statement from somewehere
        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;
}
```

13.2 Manual Persistence

- **CRUD update:**
  - For updates, there are also various implementation approaches with different performance and safety properties
  - **Immediate updates:**
    - **Directly persist updates after they occur**
    - Augment each Bean's set method with additional JDBC calls that directly write the new value into the DBMS
    - Database immediately updated
    - **Performance might be very low**
    - Only few problems with transactional consistency

13.2 Manual Persistence

- **Explicit updates:**
  - **Persist updates on manual command**
  - Just leave set methods as they are, but mark each object as **"dirty"** as long as it is used
  - Provide an additional update method writing the whole Bean to the DB if it is dirty
    - **Traverse** the whole **object structure recursively** and also write all dirty, related objects
  - Worst case:
    Severe **problems with transactional consistency**

13.2 Manual Persistence

- **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?**

13.2 Manual Persistence

Detour

- Another major drawback is **code perturbation**
  - The code of the application data model is littered with code for dealing with persistence
  - Application models are **"dirty"** and bound to the chosen implementation
  - In this example, the initial Villain.java model grew from **26 lines** of code (class definition, attributes, getters & setters) to over **180 lines** of code (with simple (!) methods for persistence added)!

13.3 Persistence Frameworks

- **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!

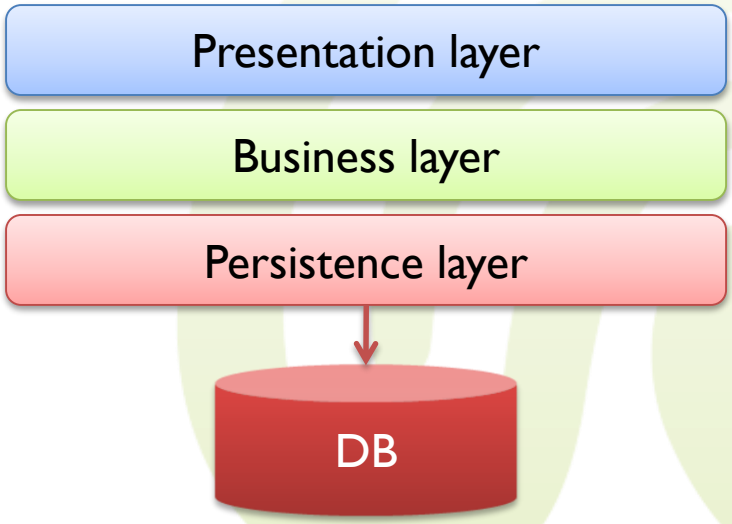
    - Provide automated support to problems of persistence
  - Persistence handling should be **transparent**
    - Just model your application data the way **you** need it, and not the way your DBMS needs it
    - **Keep your models clean!**

13.3 Persistence Frameworks

- 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
    - Example: Web interface vs. Web service interface
    - Example: Persistence using a RDBMS vs. persistence using XML

13.3 Persistence Frameworks

- 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



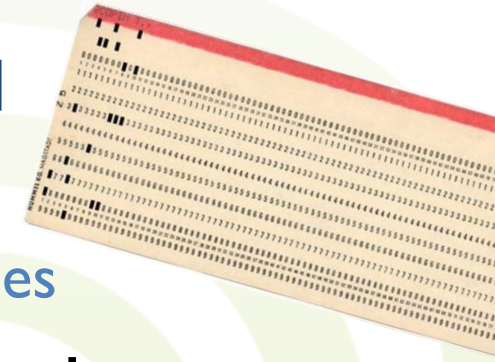
13.3 Persistence Frameworks

- **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**

13.4 Generating IDs

Detour

- 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**

13.4 Generating IDs

Detour

- 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

13.4 Generating IDs

Detour

- 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!

13.4 Generating IDs

Detour

- 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

13.4 Sequences

Detour

- **Full sequenced keys**
  - A **central authority** (usually the DBMS) provides all keys
  - The keys follow a strict sequence
    - Example: 1, 2, 3, 4, 5, ….
    - Usually, there is **one sequence per table**

  - **Pros**
    - Keys are short and easy to debug
      - Example: … 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**

13.4 Sequences

Detour

- 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

13.4 Sequences

Detour

- **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 AS IDENTITY

- **Example DB2:**
  - **CREATE TABLE** hero (id **INTEGER NOT NULL GENERATED 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!**

13.4 Hi-Low Keys

Detour

- **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

13.4 Hi-Low Keys

Detour

– **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

13.4 UUIDs

Detour

- **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

13.4 UUIDs

Detour

- 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**)

13.4 UUIDs

Detour

- **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

13.4 UUIDs

Detour

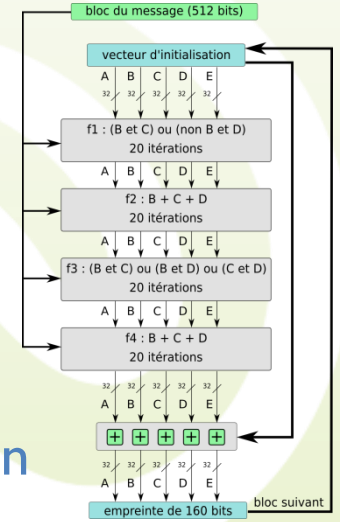
– The Melissa worm

- 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 **arrested**

13.4 UUIDs

Detour

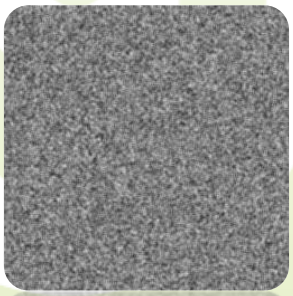
- **Type-2 UUID**:
  - Like Type-1, but parts of the timestamp are replaced by the POSIX UID domain and user UID
    - Privacy issues even more severe
- **Type-3 UUID:**
  - Everything but the type digit is generated by a MD5 hash function
  - Input is usually a URL, object identifier, etc…
    - Problems 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



13.4 UUIDs

Detour

- **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%"*



13.4 UUIDs

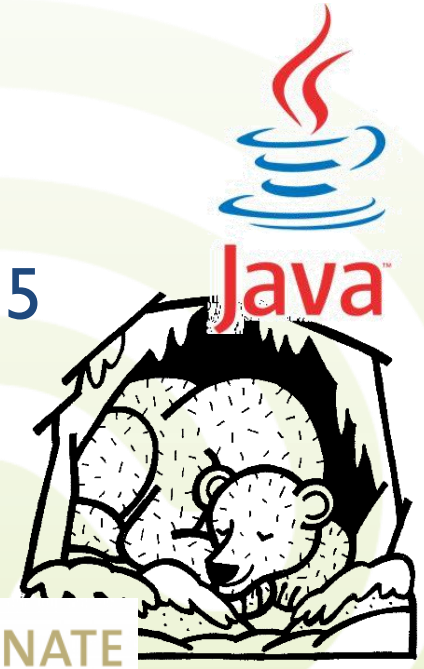
Detour

- 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

13.5 Persistence Frameworks

- 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



13.5 JPA

- 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
  - …

Castor

XORM

JPOX
Java Persistent Objects

- Each of these frameworks had their own **proprietary API** and **metadata format**

13.5 JPA

- JPA provides a common interface for all relational persistence frameworks written in Java
  - Released in **May 2006** by Sun Microsystems
  - 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)
  - Also known as **JSR 220**
- Consist of three **components**
  - The actual **API** in **javax.persistence**
  - The **JPQL query language**
  - Facilities for handling **metadata**

13.5 JPA

- JPA is a **standardized API**
  - Sun was able to convince most persistence framework vendors to **adopt the API**
  - If you want to use it, you have to get a compliant framework
    - The current reference implementation is Oracle TopLink Essential
    - Now open source and supported by Eclipse, called **EclipseLink** http://www.eclipse.org/projects/project_summary.php?projectid=rt.eclipselink
- **Resources**
  - Overview:
    - http://java.sun.com/javaee/technologies/persistence.jsp
  - API doc:
    - http://java.sun.com/javaee/5/docs/api/javax/persistence/package-summary.html

13.5 JPA

- **Main idea:**
  Use of **POJO**s 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

13.5 JPA

- Using **annotations**,
  the development **workflow** looks like this:
  1. Annotate all **persistent entities**
     - That is: 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

13.5 JPA

Detour

POJOs

- **Beans**

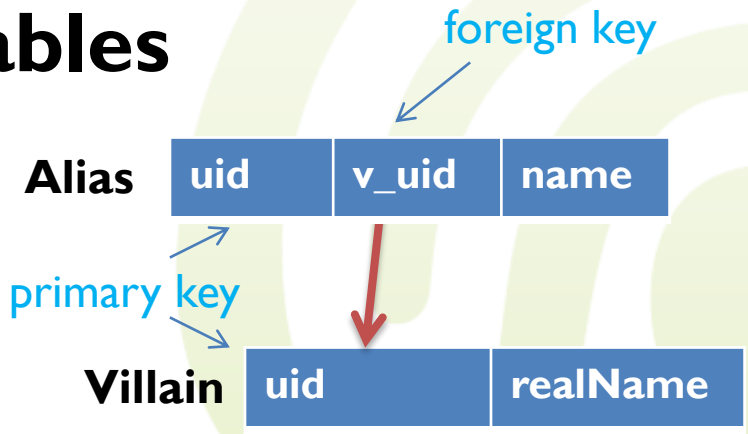
Villain.java

```
public class Villain {
    String  realName;
    Set<Alias>  aliases;
    //...put additional getter/setters here
}
```

Alias.java

```
public  class  Alias {
    String name;
    //...put getter/setters here
}
```

- **Relational tables**



13.5 JPA

Detour

Annotate persistent entities

Villain.java

```
@Entity
public class Villain {
      String  realName;
      Set<Alias>  aliases;
      //...put additional getter/setters here
}
```

Alias.java

```
@Entity
public  class  Alias {
      String name;
      //... put  getter/setters here
}
```

Add and annotate UID columns

Villain.java

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

Alias.java

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

Detour

Annotate attributes and relationships

Villain.java

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

"Use lazy loading"

Alias.java

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

"Ignore this attribute"

- All simple attributes are **persisted by default** unless annotated with @Transient
- All collections need an annotation defining the relationship (**one-to-one, 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**

13.5 JPA

Detour

- Usually, the JPA implementation **automatically designs the relational schema** as it likes
- That is fine, if **only one application** uses the persistent entities or all applications use the same JPA meta data
- However, if you have to **use an existing relational schema** or other existing applications have to use the tables, it might become tricky
  - Detailed control of **object-relational mapping** necessary
  - Use **advanced annotations** for this task

13.5 JPA

Detour

- **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.

13.5 JPA

Detour

- Also, all this **mapping meta data** can be provided by an **XML** file instead of annotations
- Beside mapping meta data, you need to define your **DBMS connection**
  - Use persistence.xml file in the META-INF folder

```
<persistence>
    <persistence-unit name="villains">
        <provider>YOUR JPA IMPLEMENTATION NAME GOES HERE</provider>
        <class>Villains</class>
        <class>Alias</class>
        <properties>
            <!-- Properties for your JDBC driver, username, password, URL, etc. --!>
        </properties>
    </persistence-unit name="villains">
</persistence>
```

13.5 JPA

Detour

- After persistence properties are defined, the **JPQL query** language may be used to retrieve objects
  - Similar to SQL, but uses persistent entities and their attributes instead of tables
- You can query and load/store/change objects using the JPA Entity Manager
  - Entity Manager is created using a persistence unit definition

13.5 JPA

Detour

```
factory = Persistence.createEntityManagerFactory("villains");
EntityManager em = factory.createEntityManager();
// query
Query q = em.createQuery("SELECT v FROM villains AS v WHERE
    v.realName='Galan' ");
List<Villain> villain = (List<Villain>) q.getResultList();
//create a new villain
em.getTransaction().begin();
Villain newVillain = new Villain();
newVillain.setRealName("Gala");
newVillain.addAlias(new Alias("Galactus"));
newVillain.addAlias(new Alias("Eater-of-Worlds"));
em.persist(newVillain);
// change Galactus
newVillain.setRealName("Galan");
em.persist(newVillain);
//
em.getTransaction().commit();          em.close();
```

13.6 Hibernate

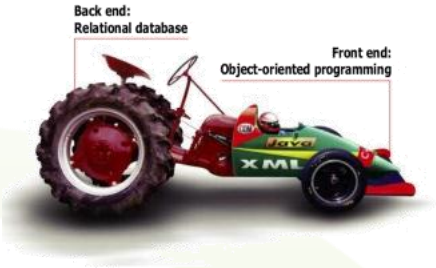
- Hibernate is the **most popular** Java persistence framework
  - Initially developed by **Gavin King**
  - **Open source**
  - Became really popular in 2003
    - However, it is not really known when the first version has been released…
  - Like most early frameworks, Hibernate provided proprietary mapping **meta data definitions,** a **query language** (HQL), and an **entity manager**
  - Supports **JPA** since version 3.0
  - Acquired by **JBoss Inc.**

13.7 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 **impendence 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

13.7 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
  - …



InterSystems CACHÉ

GEMSTONE

Objectivity.

13.7 Object Databases

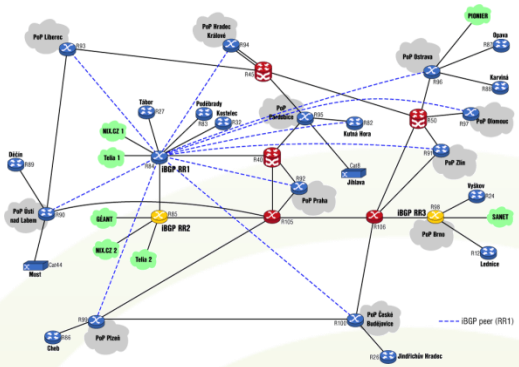
- Unfortunately, most ODBMS **spectacularly failed**
  - Products unfinished and unpolished
  - Crappy performance due to misuse
  - Obscure and highly proprietary APIs, standards, and query languages
  - Integration with legacy systems was very hard
  - Most vendors went out of business…
  - ODBMS closed in 2001

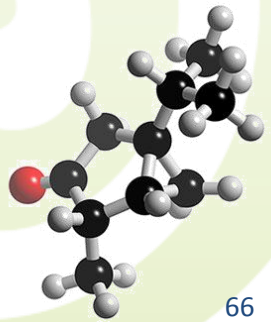
13.7 Object Databases

- 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 to much time** in development and production
  - Newer ODBMS mainly focus on persisting the object state and don't include the behavioral aspects

13.7 Object Databases

- 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)





13.7 Object Databases

- Objects databases are **not a great choice** when
  - Your data needs to **be accessed by** other applications which are not specially devolved for ODBMS
    - Today's ODBMS are still quite proprietary
    - Data is just stored "somehow" **(encapsulation)**
  - 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
  - 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. Nothing else.

13.7 Object Databases

> "… 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

- This might or might not be true, **but** object databases are still a very 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 a ODBMS, you can achieve spectacular performance paired with easy development and management

13.7 db4objects

Detour

- **db4objects** is native Java object database
  - Developed since 2000 as open source sponsored by Veritas and Sun
  - http://www.db4o.com
  - Sold to Versant in 2008
    - Planned target marked: Mobile user data management

- What does "native" mean?
  - Written in **Java,** only using Java and nothing else
  - **Object data model** (no mapping or whatsoever)
  - **No embedded query languages**
    - Queries are also performed via Java objects

13.7 db4objects

Detour

- Handling of Java POJOs is extremely easy thanks to the native support
  - No modifications or meta data necessary
  - No unique identifiers necessary

A real POJO!

Villain.java

```
public class Villain {
    String  realName;
    Set<Alias>  aliases;
    //...put additional getter/setters here
}
```

Creating and persisting a villain

```
ObjectContainer db =
    Db4o.openFile(dbFile);
Villain v = new  Villain("Norrin Radd");
v.addAlias(new Alias("Silversurfer"));
v.addAlias(new Alias("The Herald"));
db.store(v);
```

13.7 db4objects

Detour

- Provides two query "languages"
- **QBE** (Query by example)
  - Just provide a template resembling the type of object you like to have
- **SODA**
  - Simple Object Database Access
  - Query by criteria
    - Query is an object structure of different query objects

13.7 db4objects

Detour

- **Example:**
  Query for all villains having the alias “Silversurfer”
  - **QBE**
    - Create a prototypical object matching the query description
    - All uninteresting attributes/relationships are left out or set to null

```
Villain proto = new  Villain(null);
proto.addAlias(new Alias(“Silversurfer”));

List<Villain>  result= (List<Villain>)
        db.queryByExample(proto);
```

13.7 db4objects

Detour

- **Example:**
  Query for all villains having the alias "Galactus"
  - **SODA**
    - Create a query expression by composing query objects
    - Provide all conditions that have to be fulfilled by result objects

```
Query q = db.query();
q.constrain(Villain.class);
q.descend("aliases").descend("name").
    constrain("Galactus").equal();
result = (List<Villain>) q.execute();
```

13.7 db4objects

Detour

- Benchmarks
  - Uses PolePosition open source DB benchmark
    - http://polepos.sourceforge.net/
    - Note that **you are not allowed to publish benchmarks for most commercial DBMS…**
    - However, they do not perform well compared to the very strong MySQL
  - Barcelona: **Transactions** (100 selects, 30000 objects with 5 levels)

Barcelona Benchmarks	read	write	query	delete
Native/db4o 6.4	1.0	1.0	1.0	1.0
Hibernate/hsqldb	15.8	3.7	2,583.1	4.3
Hibernate/mysql	48.0	26.1	14.4	26.9
JDBC/MySQL	40.8	19.5	9.3	15.8
JDBC/JavaDB	27,843.7	20.5	47,993.1	17.7
JDBC/HSQLDB*	1.9	1.1	2,554.4	0.5
JDBC/SQLite	8.8	519.1	1.1	362.1

fastest

slowest

* JDBC/HSQLDB not ACID transaction safe

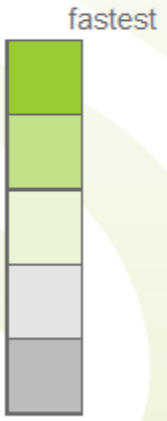
13.7 db4objects

Detour

- Benchmark

  - Melbourne - writes 100,000 unstructured flat objects of one kind in bulk mode
  - Bahrain - writes 30,000 simple flat objects individually
  - Sepang - writes an object tree of depth 14
  - Imola - retrieves 300,000 objects by native ID

Other Benchmarks	Melbourne	Bahrain	Sepang	Imola
Native/db4o 6.4	1.0	1.0	1.0	1.0
Hibernate/hsqldb	3.1	1.3	2.7	5.9
Hibernate/mysql	7.4	4.1	10.9	32.2
JDBC/MySQL	3.8	2.6	7.2	18.3
JDBC/JavaDB	1.8	2.1	407.0	5.7
JDBC/HSQLDB*	0.2	0.2	1.0	0.4
JDBC/SQLite	100.3	62.5	154.8	5.8



* JDBC/HSQLDB not ACID transaction safe

Next Lecture

- Active databases
  - Integrity constraints
  - Triggers
  - UDFs and stored procedures
- Basic security
  - Access control
  - SQL injection