ifis
Institut für Informationssysteme
Technische Universität Braunschweig

Relational Database Systems I

Wolf-Tilo Balke,
Jan-Christoph Kalo, Florian Plötzky,
Janus Wawrzinek and Denis Nagel
Institut für Informationssysteme
Technische Universität Braunschweig
www.ifis.cs.tu-bs.de

12 Application Programming 2

- JDBC
  - **Prepared Statements**
  - **Transactions**
  - **SQL Injection**
- Active Databases
  - Integrity constraints
  - Triggers
  - Complex Database Programs
    - User-Defined Functions
    - Stored Procedures
- Basic security
  - Access control

12.1 JDBC

Summary

- Basic steps when working with JDBC
  1. **Load** the driver
  2. **Define** a connection URL
  3. **Establish** a connection
  4. **Create** a statement(s)
  5. **Execute** a statement(s)
  6. **Process** the result(s)
  7. **Close** the connection

12.1 JDBC: Prepared Statements

- When performing a simple statement, roughly the following happens
  - the statement is composed in your app using **String manipulation**
  - the SQL String is **wrapped** and send to the **database** via the JDBC driver
  - the DBMS **parses** and **checks** the statement
  - the DBMS **compiles** the statement
  - the DBMS **optimizes** the statement and tries to find the best **access path**
  - the statement is **executed**
- When you execute the same/similar statement multiple times, all those steps are performed for each single statement

12.1 JDBC: Prepared Statements

- To avoid unnecessary overhead, **prepared statements** may be used
- Prepared statements use **parameterized SQL**
  - use **?** as markers for parameters
  - example:
    - `SELECT * FROM heroes WHERE id = ?`
    - generic SQL query for retrieving an hero by it's ID
  - Prepared Statements may either be used for queries or for updates / DDL operations

12.1 JDBC: Prepared Statements

- Prepared Statements use the following workflow
  - when creating a (parameterized) prepared statement, it is wrapped, sent to the DBMS, parsed, checked, and optimized
    - **only once for any number of execution**
  - each time it is executed, the values for the parameters are transferred to the DBMS and the statement is executed
  - **performance may be significantly higher** compared to using dynamic statements

12.1 JDBC: Prepared Statements

- To supply values for the placeholders, use **`setX`**`(number, value)` methods
  - like for the get and update methods, there are set methods for any data type
    - placeholders are referenced by the position in the SQL string starting with 1
  - After all placeholders are filled, you may call
    - **`executeQuery`**`()` for queries returning a **`ResultSet`**
    - **`executeUpdate`**`()` for update/DDL statements return the number of affected rows

```
PreparedStatement moviesInYears = conn.prepareStatement(
  "SELECT * FROM movies WHERE releaseDate=>? AND releaseDate=<?"
);
for(int i=0; i<10; i++) {
  moviesInYears.setInt(1, 1990+i*2);
  moviesInYears.setInt(2, 1991+i*2);
  ResultSet rs = moviesInYears.executeQuery();
  // … do something
}
```

12.2 JDBC: Transactions

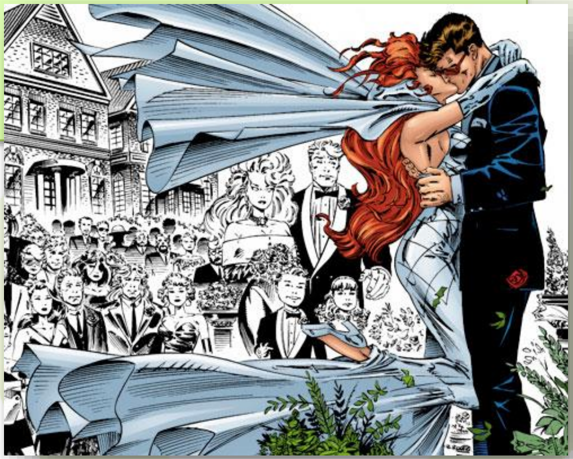
- Of course, you can use **transactions** within JDBC
  - transactions are normally disabled by default (depending on the DBMS)
    - "auto-commit"-mode is normally active
  - use **`setAutoCommit`**`(boolean switch)` to change transactional behavior
    - **`true`**`:` Every statement is executed immediately
    - **`false`**`:` Statement execution is held back until `COMMIT` is called

12.2 JDBC: Transactions

- When transactions are enabled, any number of statements is considered as one transaction until it is **committed** or **canceled**
  - to **commit** a transaction use
    - `conn.`**`commit`**`()`
  - you may also create **save points**
    - `conn.`**`setSavepoint`**`(String savepointName)`
  - to **roll back** use
    - `conn.`**`rollback`**`()`
    - or `conn.`**`rollback`**`(String savepointName)` to return to a given safe point

12.2 JDBC: Transactions

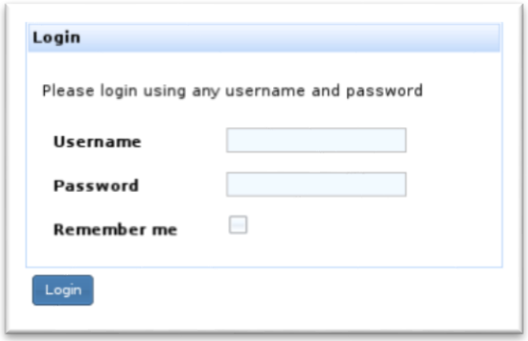
```
conn.setAutoCommit(false);

PreparedStatement changeNameStmt = conn.prepareStatement(
  "UPDATE hero SET name=? WHERE name=?"
);

changeNameStmt.setString(1, "Jean Grey-Summers");
changeNameStmt.setString(2, "Jean Grey");
changeNameStmt.executeUpdate();

changeNameStmt.setString(1, "Scott Grey-Summers");
changeNameStmt.setString (2, "Scott Summers");
changeNameStmt.executeUpdate();

conn.commit();
```



12.3 SQL Injection

- Wrong usage of JDBC can cause security problems, e.g. SQL injection
- **SQL injection** is a **security vulnerability** of an application using an SQL database
- Characteristic
  - **user input** is directly **embedded** into an SQL statement without further checking
  - user is able to **extend** the SQL statement or even **inject** completely new ones
  - thus, data may be **corrupted, deleted**, or **stolen**

12.3 SQL Injection



- Example scenario
  - *A web interface asking for a **username** and a **password**.*
  - following **statement** is used to authenticate the user:

```
String s = "SELECT * FROM users " +
   "WHERE username = '" + user + "'" +
   "AND password = '" + passwd + "';"
```

  - the application **simply inserts the user input** into the SQL string (using string concatenation)
  - if there is the given username/password combination, the application proceeds to the protected member area

12.3 SQL Injection

- Possible attacks
  - authenticate as admin
    - username = admin
    - password = ' OR 1=1 '

```
SELECT * FROM users WHERE username = 'admin'
AND password = '' OR 1=1'';
```

  - drop the user table
    - username = admin
    - password = '; DROP TABLE users; --

SQL comment

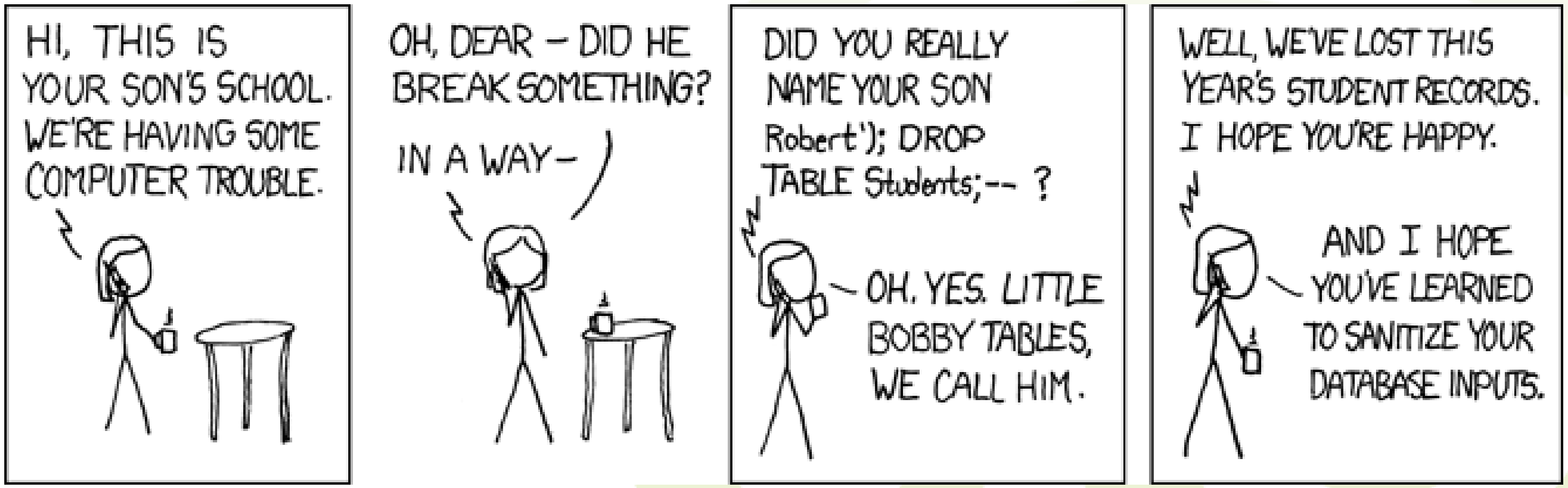
```
SELECT * FROM users WHERE username = 'admin'
AND password = ''; DROP TABLE users; --';
```

12.3 SQL Injection

– even worse – capture the whole system!

- some DBMS systems provide stored procedures to access the underlying operating system itself (e.g. MS SQL)

- `'; EXEC xp_cmdshell 'format c:  /s';`


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

12.3 SQL Injection

- What hackers usually do
  - hackers usually don't know the queries, tables, and inner workings of applications
    - vulnerabilities need to be *discovered*
  - start with entering information containing any SQL **control characters** (e.g. ')
    - if this results into an error,
      the application is potentially prone to injection attacks
  - inject SQL code in order to guess the **structure** of the tables and columns, and also the **security boundaries** of the system
    - observe the **error codes** to validate your guesses
  - as soon as the extend of the vulnerability data schema is known, data can be freely manipulated or stolen

12.3 SQL Injection

- How to **prevent injection attacks?**
- **Sanitize the input!**
  - restrict all user input to only **safe characters** (i.e. remove control characters)
  - will also delete characters which might be needed in the input (e.g. ')
  - won't protect you in case of integer values
    - … `WHERE id = 17 OR 1 = 1`

12.3 SQL Injection

- **Quote and escape the input**
  - escape all control characters
    - this might be quite tricky and often depends on the DBMS
      - e.g. backslash is not a special character in DB2 but in MySQL it is used as default escape character
    - most database APIs provide special functions for quoting and escaping
      - e.g. `mysql_real_escape_string()` in PHP
  - example:

    input: `\'; DROP TABLE users; --`
    escaped:

    ```
    WHERE email = '\\\'; DROP TABLE users; --'
    ```
  - **Notice:** for DB2 this would not work:

    ```
    WHERE email = '\\\'; DROP TABLE users; --'
    ```
    - dedicated escape procedures for each DBMS are needed

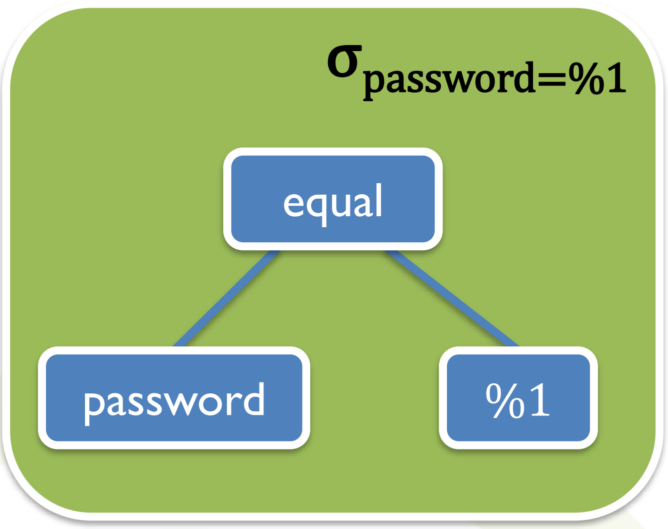
12.3 SQL Injection

- **Use strongly typed parameters**
  - cast/parse each user input to its intended data type
    - prevents e.g. integer input with injected code
    - together with sanitized input or escaping and quoting, typing provides a acceptable minimum amount of protection

```
String s = readUserInput();
try {
  Integer.parseInt(s);
} catch (NumberFormatException ex) {
  // respond to invalid input
}
```

12.3 SQL Injection

- **Use prepared statements**
  - the structure of a prepared statement is fixed
    - user input is *just data* and cannot change the predefined statement structure



  - simplest and most secure way to sanitize your input
  - besides the security benefit, prepared statements may also **increase your query performance**
  - **BEST SOLUTION – USE PREPARED STATEMENTS!**
    - If you do not use prepared statements in an application, have a good reason for that!

12.3 SQL Injection

- **Isolate your Web/DB server**
  - put your servers in a secure DMZ (**De**Militarized **Z**one)
    - even if the attacker is able to completely capture the machine, he/she won't be able to do much harm
- **Restrict your error reporting**
  - many programming frameworks are by default configured into developer mode
  - on failure, they report in detail what went wrong
    - e.g. display the faulty query and excerpts from the call hierarchy or the DB schema
    - this information is very helpful in finding security vulnerabilities, so don't give it to your foes!

Recommended DB Software

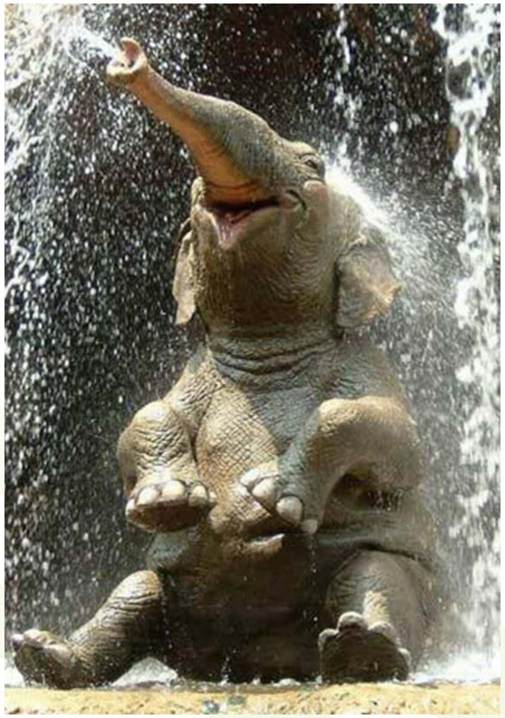
Detour

- MySQL and PostgreSQL
  - very good open source RDBMS
    - server-client architecture
  - also good for practicing
    - with a little bit more administrative overhead
  - recommended if you need a fully featured RDBMS for an application
  - MySQL
    - comes with a set of storage engines
      - **MyISAM**: no ACID, no fail recovery, no foreign keys but **fast**!
      - **InnoDB:** ACID compliant, referential integrity, etc. but slower.
      - **NDB Cluster:** in-memory DB

Recommended DB Software

Detour

- MySQL and PostgreSQL
  - PostgreSQL
    - more serious implementing the SQL standard
    - architecture closer to database theory
    - used to be slower than MySQL, but performance and scalability increased dramatically in the last years



Recommended DB Software

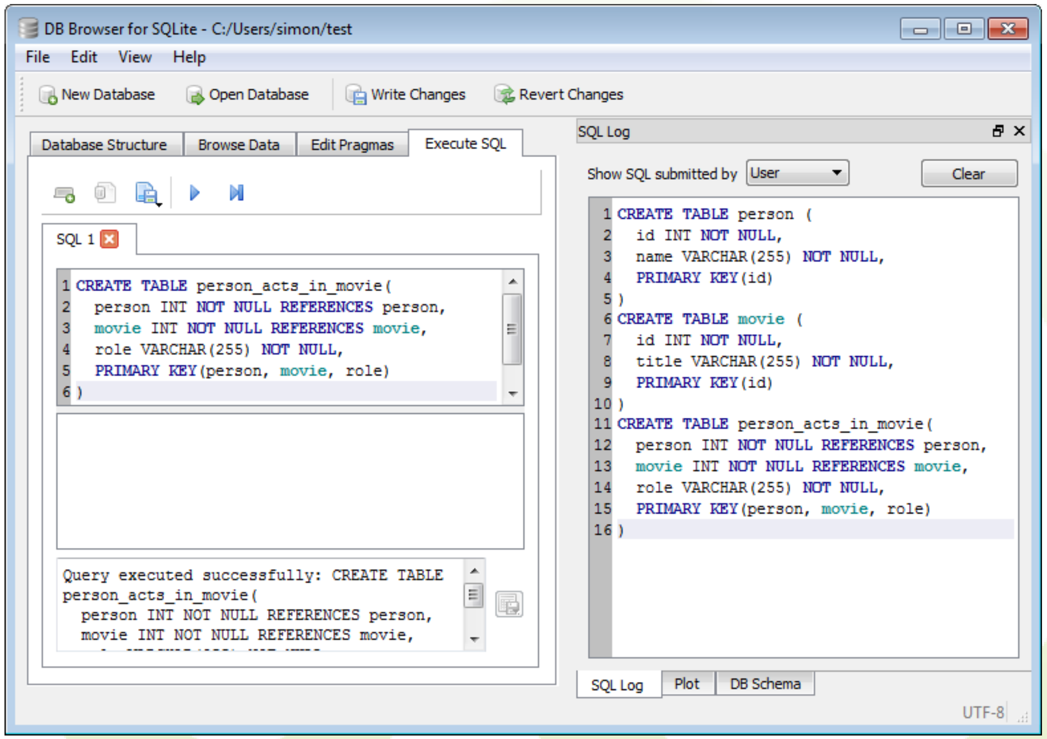
Detour

- H2 and SQLite
  - light weight and fast RDBMS
    - recommended, if a server-client architecture is not needed
      - no shared data among applications
      - no remote/distributed data access needed
  - H2
    - pure Java (and only available for Java)
    - also capable of in-memory storage
  - SQLite
    - Integrated in Android!
    - available for a large number of programming languages
      - currently 37 languages, covering C, C#, C++, Haskell, Java, JavaScript, Lua, Perl, PHP, Python, R, Ruby, Visual Basic and many more...

Recommended DB Software

Detour

- SQLite Browser
  - ideal for practicing SQL
  - download from http://sqlitebrowser.org/
  - create or load a database file and start using SQL!



```sql
CREATE TABLE person_acts_in_movie(
  person INT NOT NULL REFERENCES person,
  movie INT NOT NULL REFERENCES movie,
  role VARCHAR(255) NOT NULL,
  PRIMARY KEY(person, movie, role)
)
```

```
Query executed successfully: CREATE TABLE
person_acts_in_movie(
  person INT NOT NULL REFERENCES person,
  movie INT NOT NULL REFERENCES movie,
```

```sql
CREATE TABLE person (
  id INT NOT NULL,
  name VARCHAR(255) NOT NULL,
  PRIMARY KEY(id)
)
CREATE TABLE movie (
  id INT NOT NULL,
  title VARCHAR(255) NOT NULL,
  PRIMARY KEY(id)
)
CREATE TABLE person_acts_in_movie(
  person INT NOT NULL REFERENCES person,
  movie INT NOT NULL REFERENCES movie,
  role VARCHAR(255) NOT NULL,
  PRIMARY KEY(person, movie, role)
)
```

12 Application Programming 2

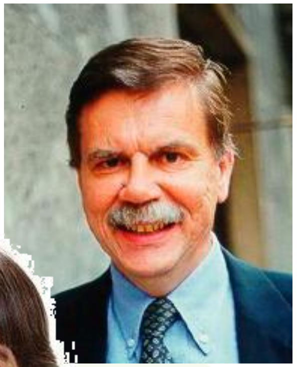
- JDBC
  - Prepared Statements
  - Transactions
  - SQL Injection
- **Active Databases**
  - **Integrity constraints**
  - Triggers
  - Complex Database Programs
    - User-Defined Functions
    - Stored Procedures
- Basic security
  - Access control

12.5 Active Databases

- There is a growing need for databases that can **actively manipulate their data**
  - in particular, the first wave of object databases encouraged many RDBMS vendors to provide active functionalities within their systems

12.5 Active Databases

- **Active databases** are RDBMS that can
  - **recognize predefined situations** and
  - **respond** to those situations with individual **predefined actions.**
- Initially proposed by S. Ceri and J. Widom in 1990
  - *Deriving Production Rules for Constraint Maintenance.* In 16th International Conference on Very Large Data Bases, Brisbane, Australia, 1990.



12.5 Active Databases

- Active databases allow programmers and admins to **enhance the functionality of the DBMS** by defining
  - **constraints**
  - **triggers**
  - **user-defined data types (UDTs)**
  - **user-defined functions (UDFs)**
  - **stored procedures**

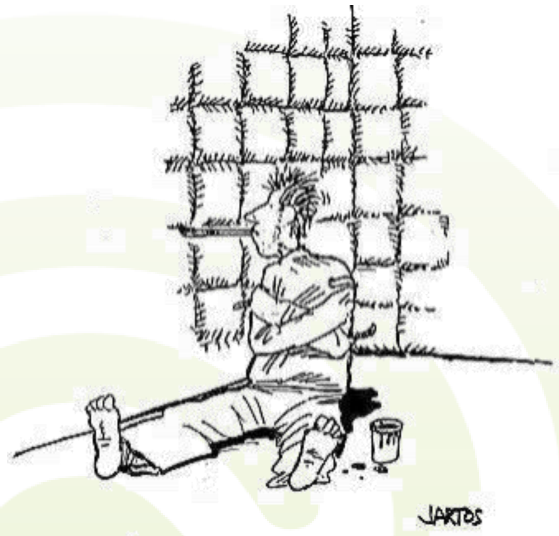
12.5 Active Databases

- Most of these active extensions started as **proprietary technologies**
  - the exact syntax strongly differs among database vendors
- Some of them have been **standardized** in SQL
  - constraints and assertions
  - triggers
  - procedural statements

ACTIVE O2 TWO fitness
ACTIVE O2 TWO fitness Peach-White Tea
15 TIMES MORE OXYGEN
ACTIVE O2 TWO fitness Apfel Kiwi
OHNE

14 Active Databases

- JDBC
  - Prepared Statements
  - Transactions
  - SQL Injection
- **Active Databases**
  - **Integrity constraints**
  - Triggers
  - Complex Database Programs
    - User-Defined Functions
    - Stored Procedures
- Basic security
  - Access control



12.5 Integrity Constraints

- The original aim of active components in database systems was to respond to attempted **violations of integrity constraints**
  - integrity constraints describe
    - what is a **valid database state**
    - how to make **valid transitions** between database states
  - **examples**
    - primary/foreign key constraints
    - data types and domains
    - *CHECK conditions* in SQL

12.5 Integrity Constraints

- **Types of constraints** include
  - **static** integrity constraints
    - bound to a single DB state (e.g. data types, key constraints)
  - **dynamic** integrity constraints
    - **transitional integrity constraints** are bound to a change of the DB state (e.g. update, insert, delete)
    - **temporal integrity constraints** are bound to a sequence of DB states (e.g. transactions, periodical checks)
- Some constraints may be difficult to evaluate and require **predicate logic** for specification
  - Master Course:
    Knowledge-Based Systems and Deductive Databases

12.5 Integrity Constraints

- An integrity constraint is called
  - **local,** if it only concerns a single relation
    - e.g. value domains, data types of attributes
  - **global,** if more than one relation is concerned
    - e.g. foreign keys

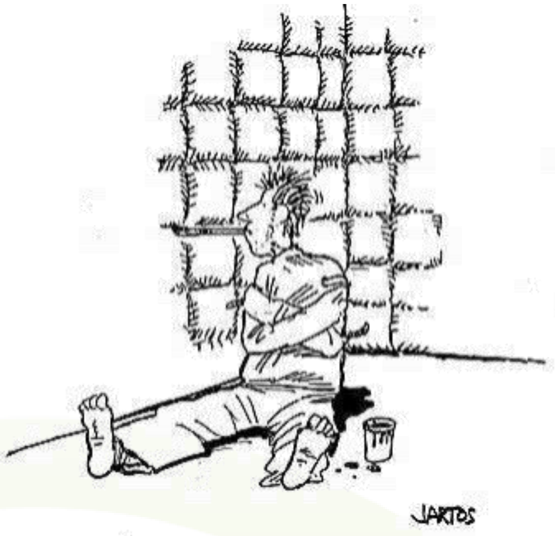
  - **implicit,** if it is a consequence of the data model
    - e.g. data types of attributes
  - **explicit,** if it is not implicit, but can be expressed in DDL
    - e.g. primary key
  - **external,** if it is neither implicit, nor explicit
    - e.g. semantic check clauses

12.5 Defining Constraints

- **Constraints** (or **assertions**) are conditions which have to be true for all data in the database instance
  - we already introduced constraints briefly (SQL)
- Constraints may be defined
  - **explicitly** by the CREATE CONSTRAINT statement
  - **implicitly** within the DDL table/column definition (CREATE TABLE … CHECK …)
- An SQL statement is executed only if it does not result in a constraint violation
  - usually critical: insert, delete, and update operations

12.5 Defining Constraints

- Summary of constraint types:
  - **data type constraint,**
    NOT NULL constraint,
    UNIQUE constraint
    - usually within column definition
  - **primary key constraint (key integrity)**
    - usually within table or column definition
  - **foreign key constraints (referential integrity)**
    - usually within table or column definition
  - **check constraints**
    - **support any arbitrary complex condition expressible in SQL**
    - usually defined explicitly or within a table definition
  - **informational constraints**
    - this type of constraint is **not enforced**
    - used by the query optimizer to better understand the data



12.5 Defining Constraints

- **Example:**
  Aliases of superheroes
  - data types, primary key, foreign key, check clause

```
CREATE  TABLE has_alias (
  hero_id INTEGER REFERENCES hero
                    ON DELETE CASCADE ON UPDATE CASCADE,
  alias VARCHAR(100) NOT NULL,
  PRIMARY KEY (hero_id, alias),
  CONSTRAINT no_silly_alias CHECK (alias <> ’Stupid Man’)
)
```

12.5 Constraint Definitions

- Constraints are used to enforce valid DB states by **rejecting** all operations resulting in invalid DB states
  - **simple and robust** tool for enforcing some basic (static) constraints
- But invalid DB operations cannot be ***repaired*** depending on the type of constraint violation
  - example: *If a tuple in some insert statement refers to a non-existing foreign key, why not simply add the respective foreign key before the insert is committed?*

12 Application Programming 2

- JDBC
  - Prepared Statements
  - Transactions
  - SQL Injection
- Active Databases
  - Integrity constraints
  - **Triggers**
  - Complex Database Programs
    - User-Defined Functions
    - Stored Procedures
- Basic security
  - Access control

12.5 Server-Side Code

- We will cover three main technologies for executing code on server side
  - **triggers**
    - a trigger is automatically executed by the DBMS when a predefined event occurs
  - **UDFs (user-defined functions)**
    - a UDF can used in any SQL statement as a function (similar to MIN, MAX, and COUNT)
  - **stored procedures**
    - a stored procedure can be executed using SQL's CALL statement (also, parameters may be specified)

12.6 Triggers

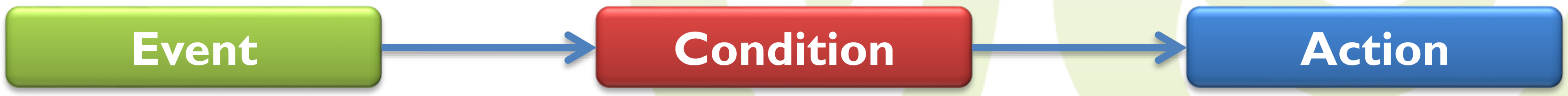
- **Triggers** link user-defined actions to standard database operations
  - whenever a certain DB operation is performed, the trigger ***fires***
  - very helpful to implement **dynamic** integrity constraints
  - each operation can have assigned **several triggers**
    - sequence of execution is usually non-deterministic
  - several triggers can fire within a transaction
  - again, different vendors use **different syntax**…

12.6 Triggers

- Standardized in **SQL:1999**
- Some DBMS offer **native extensions** to SQL for specifying the triggers
  - examples:
    PL/SQL (Oracle), Transact SQL (MS SQL Server)
- Some DBMS allow the use of **general purpose programming languages**
  - examples: Java (Oracle, DB2), C#/VB (MS SQL Server)
- Some DBMS use an **extended trigger concept**
  - example: triggers on views (Oracle)

12.6 Triggers

- Triggers implement the **event-condition-action** model
  - triggers are **active rules**
    - typical syntax: ON **<event>** IF **<condition>** DO **<action>**
  - **events** activate a rule
    - usually, triggers are fired upon data modifications
    - in general, it may be any external event
  - the **condition** determines whether the action is executed
    - optional; contains a Boolean expression
  - the **action** is executed for every event satisfying the condition
    - usually, this is done as a series of SQL (update) statements within the same transaction as the triggering event
    - but an action may also be the call of an external program



12.6 Triggers

- **Types of events** include
  - timed events
    - absolute, relative, or periodic
  - database events
    - begin/end of some insert, delete, or update statement
  - DBMS events
    - DDL commands
    - transaction events: begin, commit, or abort
    - changes in user accounts, or access control
- Today's commercial databases typically support triggers **only for database events**

12.6 Triggers

- **What to use triggers for?**
  - **auditing table operations**
    - write a protocol of each data access
  - **tracking record value changes**
    - write a modification log and archive all previous data
  - **preserving a database’s referential integrity**
    - retaining referential integrity by actively changing all affected records

12.6 Triggers

- **maintenance of semantic integrity**
  - Example: *When a super villain is caught, all henchmen should become unemployed.*
- **storing derived data**
  - customized update of materialized views
  - computing complex aggregations that cannot be expressed easily using pure SQL
- **access control**
  - checking user privileges when accessing sensitive information

12.6 Triggers

- When **creating a trigger,** the following information needs to be specified
  - **trigger name**
    - triggers use qualified names within a given schema
  - **trigger event**
    - trigger events may either monitor row updates (ON INSERT/ ON DELETE) or column updates (ON UPDATE)
    - a trigger gets **attached** to the table mentioned in the event
  - **activation time**
    - the trigger can be activated either **before** or **after** the event occurred
    - BEFORE or AFTER keywords

12.6 Triggers

- **granularity**
  - a trigger's **actions** may be executed **per statement** (statement trigger) or **per row** (row trigger)
  - per statement
    - default
    - the whole body is executed once **per event**
    - FOR EACH STATEMENT keyword
  - per row
    - the body is executed once **per affected row**
    - FOR EACH ROW keyword

12.6 Triggers

- **transition variables**
  - optional
  - often triggers need **access** to the updated (new and old values), deleted, or added data
  - REFERENCING clause
  - there are four types of transition variables:
    - old row (OLD):
      references the modified row before the triggering event
    - new row (NEW):
      references the modified row after the triggering event
    - old table (OLD_TABLE):
      references the table as it was before the triggering event (read-only)
    - new table (NEW_TABLE):
      references the table as it is after the triggering event

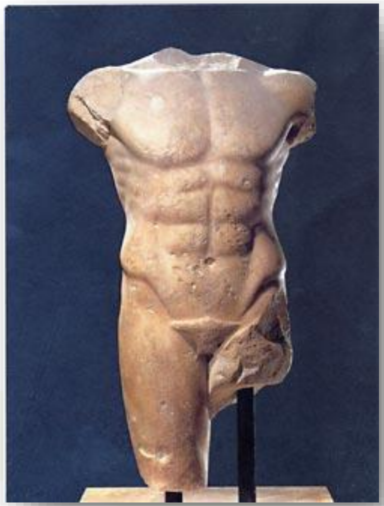
12.6 Triggers

- Not all **combinations** of trigger **events,** activation **times, granularities,** and **transition variables** are possible

Event and time	ROW TRIGGER	STATEMENT TRIGGER
BEFORE INSERT	NEW	
BEFORE UPDATE	OLD, NEW	
BEFORE DELETE	OLD	
AFTER INSERT	NEW, NEW_TABLE	NEW_TABLE
AFTER UPDATE	OLD, NEW, OLD_TABLE, NEW_TABLE	OLD_TABLE , NEW_TABLE
AFTER DELETE	OLD, OLD_TABLE	OLD_TABLE

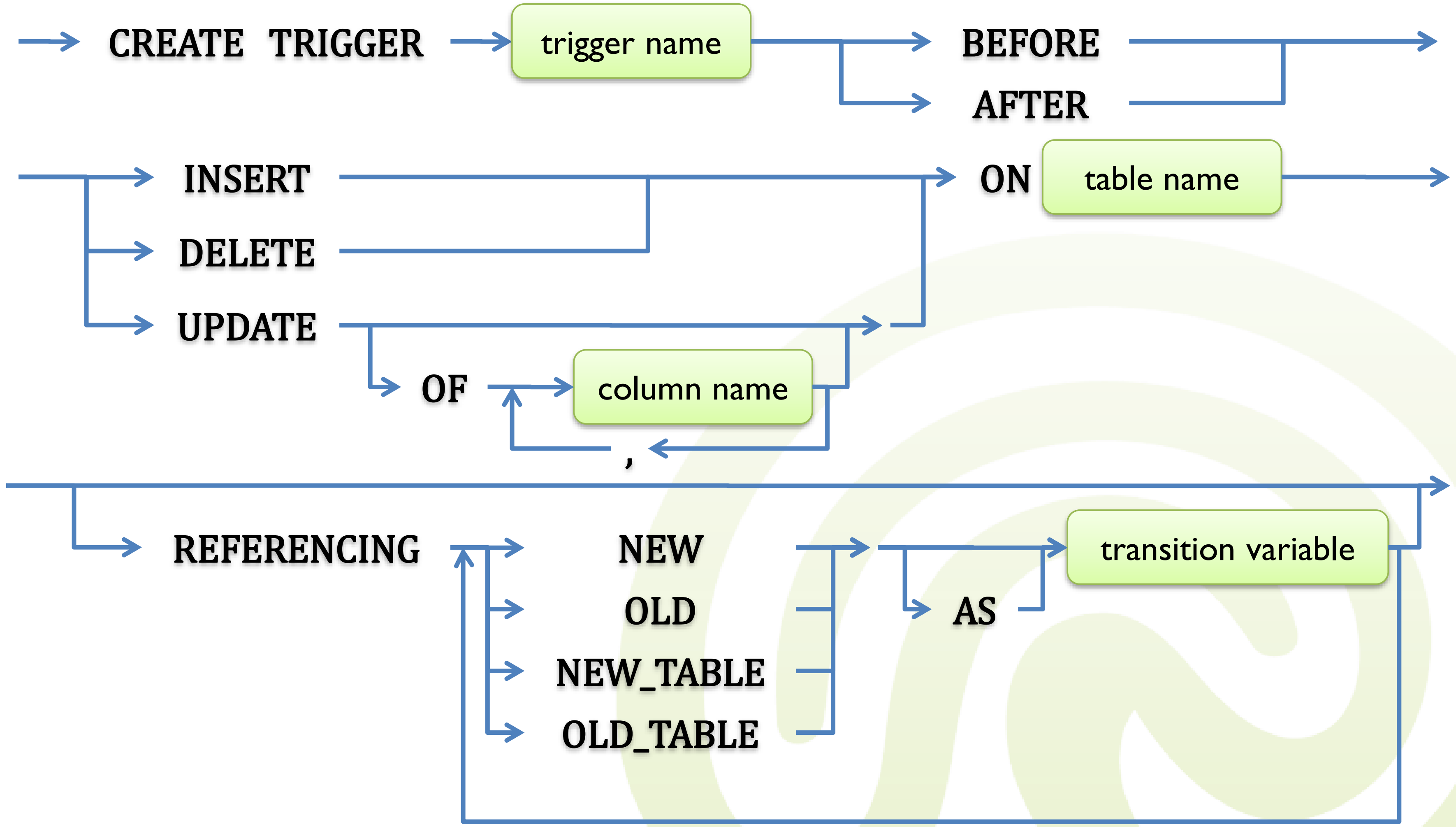
12.6 Triggers

- **trigger condition**
  - optional
  - WHEN clause
  - use any Boolean expression
    (as in SQL's WHERE clause)
- **trigger body**
  - can be any number of SQL statements,
    separated by semicolon
  - embedded into a BEGIN-END block
  - some DBMS also allow calling code
    written in other languages or even binary programs

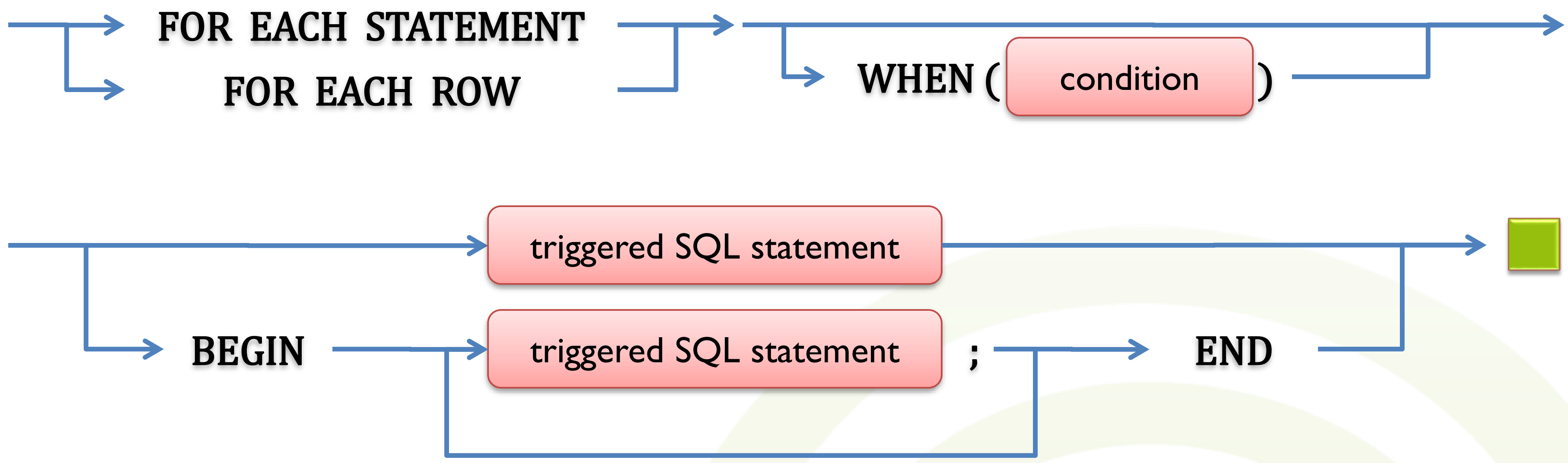


12.6 Triggers

CREATE TRIGGER STATEMENT



12.6 Triggers



12.6 Triggers

- **Example**
  - a DB storing the current location of things and heroes
  - trigger: *As soon as Superman comes near kryptonite,* ***delete him!***

```
CREATE TRIGGER kill_superman
    AFTER UPDATE OF location ON heroes
    REFERENCING NEW AS hn
    FOR EACH ROW
    WHEN hn.name = 'Superman'
            AND EXISTS(SELECT * FROM stuff s
                                    WHERE s.name = 'Kryptonite'
                                        AND s.location = hn.location)
    BEGIN
        DELETE FROM heroes h WHERE h.id = hn.id;
    END
```

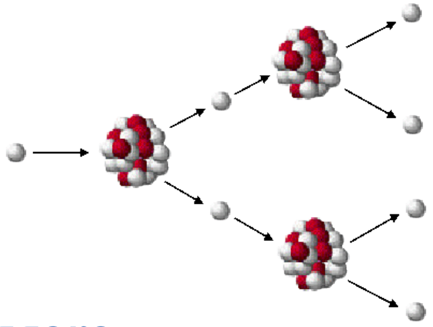
12.6 Triggers

- The previous example is **standard SQL:1999**
  - it won't necessarily work on all DBMS
  - example **DB2:**
    - replace BEGIN by BEGIN ATOMIC
      - or just don't use BEGIN-END at all
    - add MODE DB2SQL before WHEN
  - read the technical documentation of your DBMS!
- There are some **prototype implementations** for **active databases** based on ECA rules, thus also supporting a larger group of events

12.6 Triggers

- **Trigger execution order**
  1. Execute all BEFORE STATEMENT triggers
  2. Temporarily disable all integrity constraints defined on each affected table
  3. Loop for each row in the table
     1. Execute all BEFORE ROW triggers
     2. Execute the SQL statement against the row and perform integrity constraint checks of the data
     3. Execute all AFTER ROW triggers
  4. Complete deferred integrity constraint checks on the table
  5. Execute all AFTER STATEMENT triggers

12.6 Triggers



- **Trigger chaining**
  - when using triggers, the actions of one trigger might **activate another trigger**
    - that trigger may actually activate even more triggers
  - it is very easy to lose track of what happens…
  - you need to be **very careful here!**
- **Recursive triggers**
  - special case of chaining: A trigger **activates itself** again
  - it is easy to create **infinite loops**
  - even if you do not create infinite loops, most **DBMS don't handle this too well**
    - Example: *DB2 aborts with a TOO COMPLEX error, if a trigger activates itself more than once.*

12 Application Programming 2

- JDBC
  - Prepared Statements
  - Transactions
  - SQL Injection
- **Active Databases**
  - Integrity constraints
  - Triggers
  - **Complex Database Programs**
    - **User-Defined Functions**
    - **Stored Procedures**
- Basic security
  - Access control

12.7 Complex Database Programs

- Next, we will introduce two flavors of complex **database programs**
  - **Stored Procedures**
  - **User-Defined Functions (UDFs)**
- Both reside **within the DBMS** and may be called explicitly
  - exchange of information possible via input and/or output **parameters** and **result sets**
  - application programmers and users may define and create those database programs

12.7 Complex Database Programs

- **Stored procedures** are **called directly** by the application or by other procedures
  - CALL removeInactiveHeroes(00200000)
- **UDFs** can be used within any SQL statement as a **functional expression**

  -
    ```
    SELECT *
       FROM villains v
    WHERE notoriety(v.id) > 100
    ```

12.7 Complex Database Programs

- What are possible **advantages?**
  - **move** parts of program **logic** (code!) to the server
  - **improve** application **performance** by reducing client/server communication
    - database program is executed in the DBMS
  - **control access** to database objects
    - database programs can be used instead of queries, thus enabling fine-grained access control
  - integrate some ***non-database functionality*** into the DBMS
  - **readability** and **reliability** of common, complex queries can be increased by encapsulation of some functionality

12.7 Complex Database Programs

- What **problems** can you encounter?
  - database server may end up being a **performance bottleneck**
  - writing database programs ***disturbs*** your usual application development and **deployment process**
    - they are usually written in a different language
    - they have to be installed and registered with the DBMS
  - database programs can be **tricky to debug**
    - it can be cumbersome to get debug information from DBMS
    - your normal debugging environment may not work
    - there may be complex dependencies among DB programs
  - you can easily **lose track** of your database programs and versions
    - they reside outside your normal source control programs

12.7 Stored Procedures in DB2

Detour

- DB2 offers three kinds of stored procedures
  - **SQL stored procedures**
    - directly written in procedural SQL as defined by SQL:1999
  - **external stored procedures**
    - written in one of the many higher programming languages supported by DB2
      - e.g. C, CL, RPG, Cobol, ...
  - **Java stored procedures**
    - actually, they are also external stored procedures
    - due to the different implementation and deployment mechanics, they are treated as an extra case

12.7 Stored Procedures in DB2

Detour

- Stored procedures
  - defined by the **CREATE PROCEDURE** statement:

```
CREATE PROCEDURE name
(list of input and output variables)
Procedure properties
Procedure body
```

interface variables

e.g. programming language,
IO properties, ...

e.g. list commands in SQL or an
external method written in Java

12.7 Stored Procedures in DB2

Detour

- Example: Modify data using SQL
  - create a new table `numbers`
    - containing all numbers between 0 and *x*

```
CREATE PROCEDURE create_numbers
   (IN x INTEGER)
  LANGUAGE SQL
  MODIFIES SQL DATA
BEGIN
  DECLARE v_counter INTEGER DEFAULT 0;
  CREATE TABLE numbers (num INTEGER);
  WHILE v_counter < x DO
    INSERT INTO numbers VALUES (v_counter);
    SET v_counter = v_counter + 1;
  END WHILE;
END
```

12.7 Stored Procedures in DB2

Detour

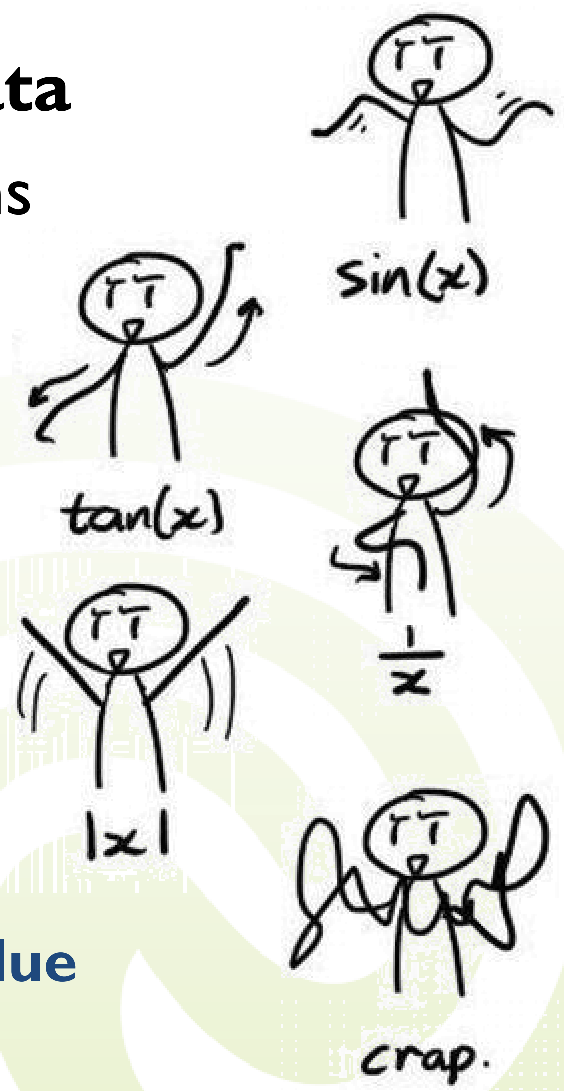
- DB2 also allows to create **stored procedures** written in **Java**
  - DB2 comes with its own Java Virtual Machine
  - class files containing the procedure can be uploaded and bound to the DBMS
  - a single Java class can define multiple stored procedures
  - classes have to inherit from **`StoredProc`**
    - provided by DB2's JDK

```
CREATE PROCEDURE get_random_number
  (OUT number double)
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  EXTERNAL NAME 'ifis.SomeJavaStoredProcedures!getRandomNumber';
```

12.7 User-Defined Functions in DB2

Detour

- SPs are usually used to **manipulate data**
- **User Defined Functions** are functions that can be used in SQL statements
  - implement a grouping function for standard deviations, medians, etc.
  - create a function returning the number of days passed since your birthday
  - return the response of a web service, the parsed content of a text file, etc.
- There are two types of UDFs
  - **scalar** functions returning just a **single value**
  - **table** functions returning a whole **table**



12.7 User-Defined Functions in DB2

- UDFs
  - Defined by the **CREATE FUNCTION** statement

```
CREATE FUNCTION name
(list of input parameters)
Returns
Function properties
Routine body
```

name and type of input variables

type of output

programming language, optional interpreter hints, ...

SQL commands or external call

12.7 User-Defined Functions in DB2

Detour

- **Example:** Simple function with scalar return value

```
CREATE FUNCTION display_name
  (firstname VARCHAR(50), lastname VARCHAR(50))
  RETURNS VARCHAR(100)
  LANGUAGE SQL
  SPECIFIC displayName01
  DETERMINISTIC CONTAINS SQL
RETURN firstname || '  ' || lastname;
```

- **Example:** Simple function with tabular return value

```
CREATE FUNCTION alias_of(heroname VARCHAR(50))
  RETURNS TABLE(alias VARCHAR(50))
  LANGUAGE SQL
  SPECIFIC alias_of_01
  READS SQL DATA
RETURN
  SELECT alias FROM aliases a, heroes h
  WHERE a.hero_id = h.id AND h.name = heroname
```

- External UDFs (in Java, C, Cobol, …) are also possible

12 Application Programming 2

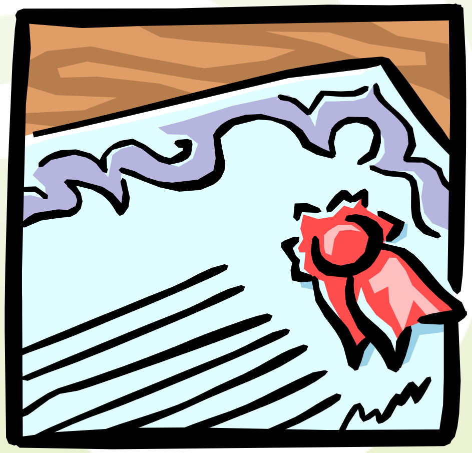
- JDBC
  - Prepared Statements
  - Transactions
  - SQL Injection
- Active Databases
  - Integrity constraints
  - Triggers
  - Complex Database Programs
    - User-Defined Functions
    - Stored Procedures
- **Basic Security**
  - **Access control**

12.8 Basic Access Control

- A major concern in databases is **data security**
  - remember: **views** can be used for restricting the data access of some application
    - e.g. *Salaries of employees are not shown in staff listing.*
    - of course, this works only if the **original table** cannot be accessed by the application
  - a basic mechanism to enforce **access rights** to data is so-called **discretionary access control**
    - grants privileges to users, including the capability to access specific data files, records, or fields in a specific mode (r/w)

12.8 Discretionary Access Control

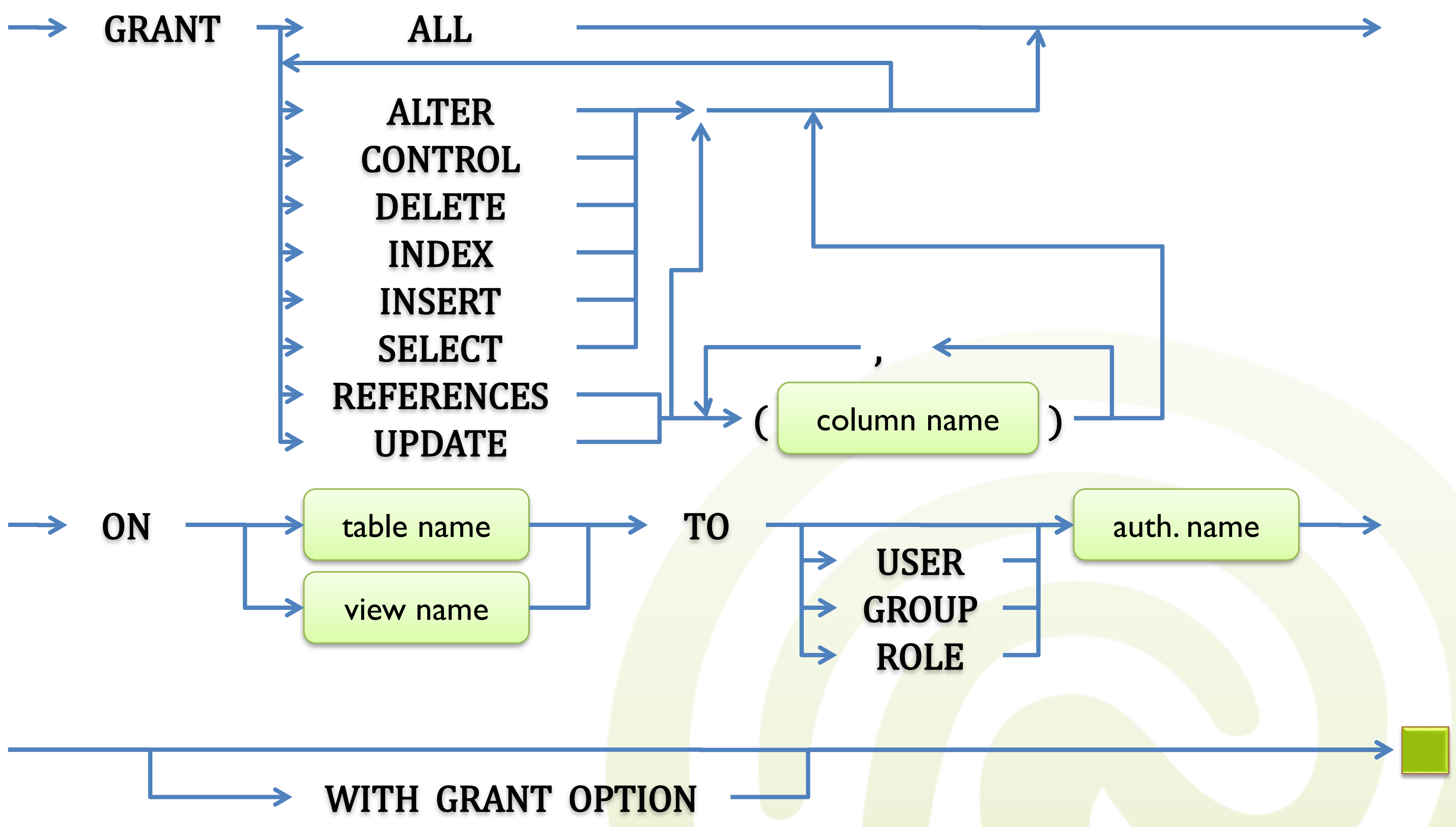
- Discretionary policies require that, **for each user,** authorization rules specify the **privileges granted** on the database objects
  - access requests are checked against the **granted privileges**
  - discretionary means that users may **grant/revoke** permissions (usually based on ownership)
  - by grants, access privileges can be **propagated** through the system



12.8 Discretionary Access Control

- The **SQL GRANT/REVOKE** statement can be used to grant privileges to users
  - GRANT privileges ON table(s)/column(s) TO grantees [WITH GRANT OPTION]
  - REVOKE privileges ON table(s)/column(s) FROM grantees
- **Possible privileges** are:
  - SELECT: user can retrieve data
  - UPDATE: user can modify existing data
  - DELETE: user can remove data
  - INSERT: user can insert new data
  - REFERENCES: user can define foreign keys to the table

12.8 Discretionary Access Control

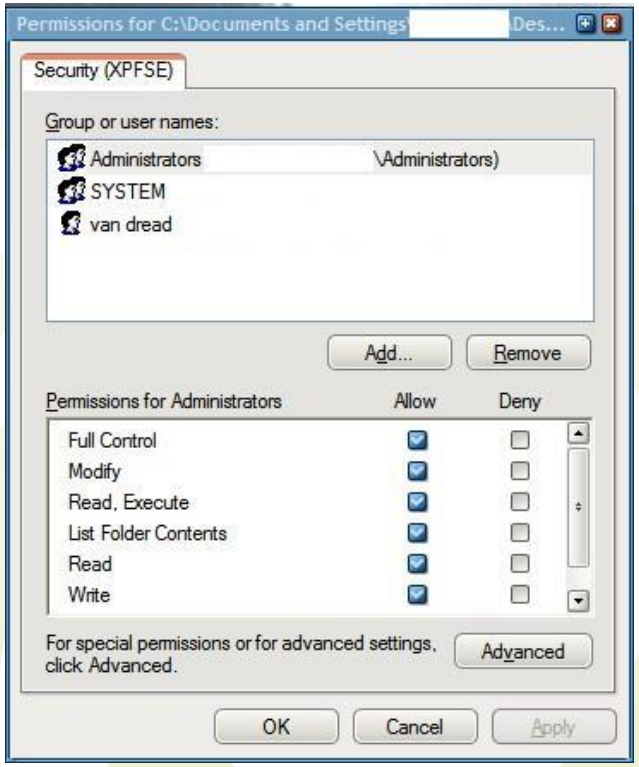


12.8 Discretionary Access Control

- The **WITH GRANT OPTION** option permits the propagation of grant permissions to other users
  - allows other users to define permissions for certain tables
- The **list of grantees** does not need not be (a set of) usernames
  - it is permitted to specify PUBLIC, which means that the privileges are granted to **everyone**
    - **be very careful with that!**

12.8 Discretionary Access Control

- Checking discretionary access control is often implemented by an **authorization matrix**
  - the **rows** represent users
  - the **columns** represent the database objects
  - the **fields** contain the respective privileges
- Similar concept in Windows file security



Permissions for C:\Documents and Settings\ Des...

Security (XPFSE)

Group or user names:

- Administrators \Administrators)
- SYSTEM
- van dread

Permissions for Administrators	Allow	Deny
Full Control	✓	☐
Modify	✓	☐
Read, Execute	✓	☐
List Folder Contents	✓	☐
Read	✓	☐
Write	✓	☐

For special permissions or for advanced settings, click Advanced.

12.8 Discretionary Access Control

- Granting or revoking permissions of users manually **for every possible access** is a very time-consuming task
  - more refined concepts of database security exist, for example role-based access control
- But data **security needs** more than simple access control
  - **authentication:**
    Is the user really who he/she claims to be?
  - concepts are discussed in detail in master course **Relational Databases 2**

13 Next Lecture

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