ifis

Institut für Informationssysteme
Technische Universität Braunschweig

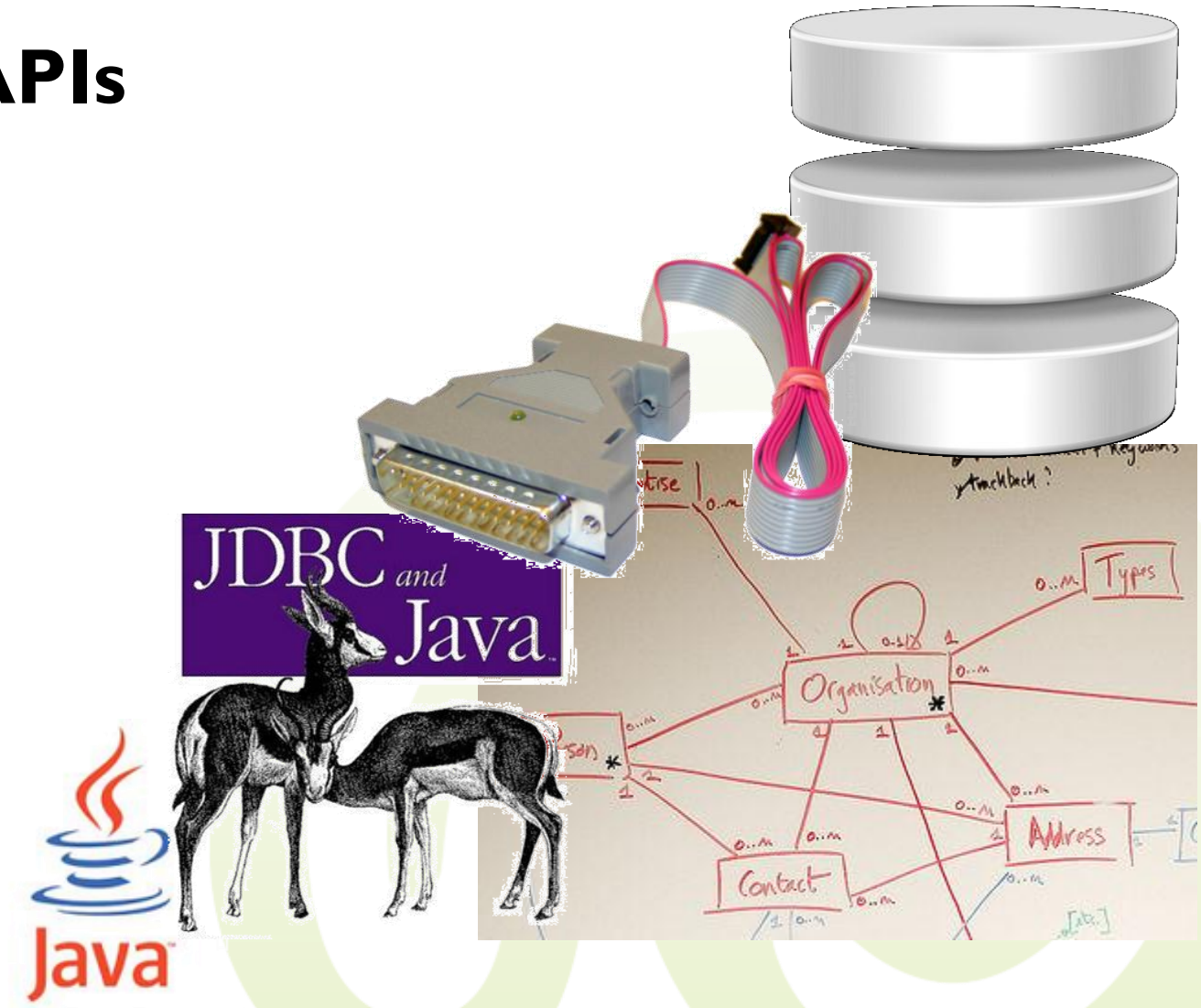
Relational Database Systems I

Wolf-Tilo Balke
Simon Barthel

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

Overview

- **Database APIs**
  - **CLI**
  - **ODBC**
  - **JDBC**

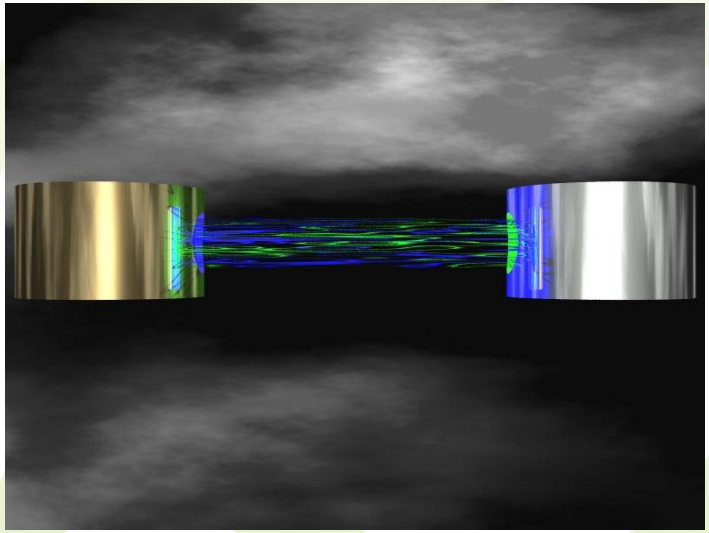


12 Accessing Databases

- Database access using a **library** (application programming interface, **API**)
  - Most popular approach
  - Prominent examples
    - **CLI** (Call level interface)
    - **ODBC** (Open Database Connectivity)
    - **JDBC** (Java Database Connectivity)

12 Accessing Databases

- General steps in using database APIs
  - Set up the **environment**
  - **Define** and **establish** connections to the DBMS
  - **Create** and **execute** statements (strings)
  - **Process** the results (cursor concept)
  - **Close** the connections

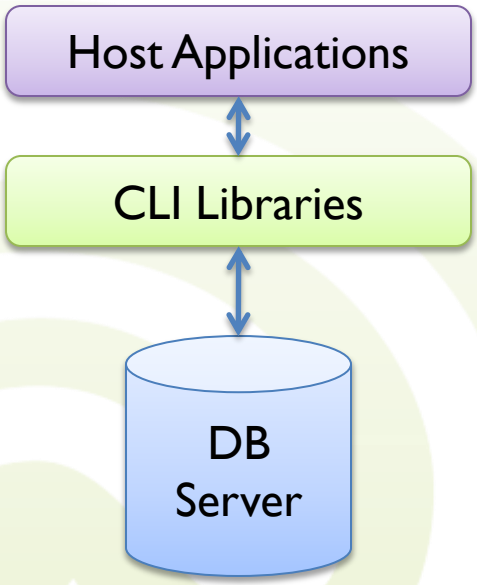


12.1 CLI

- The **Call Level Interface (CLI)** is an ISO software standard developed in the early 1990s
  - Defines how programs send queries to DBMS and how result sets are returned
  - Was originally targeted for **C** and **Cobol**
- Vision: **Common Application Environment**
  - Set of standards and tools to develop open applications
  - Allows to integrate different programming teams and DB vendors

12.1 CLI

- CLI libraries are provided by the **DB vendors**
  - Each library is specific for the respective DBMS and follows the individual DBMS's syntax
  - However, vendor libraries all follow the CLI standard and can be used interchangeably by all applications

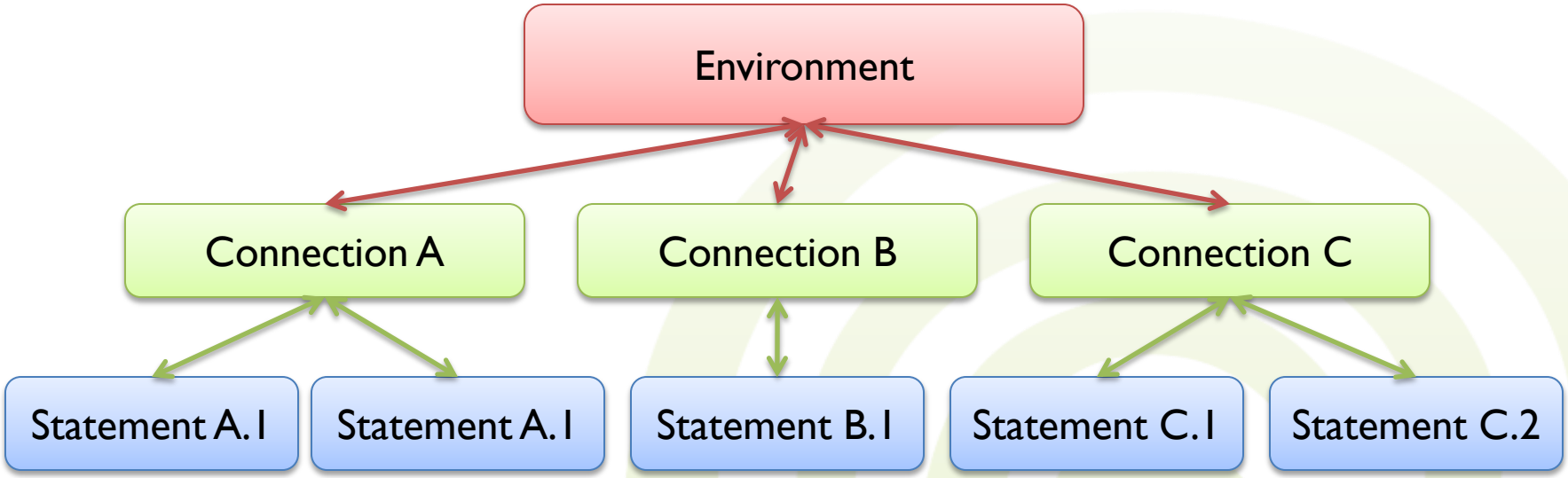


12.1 CLI

- Host language connects and accesses DB using following concepts
  - **Environments:** Represent the DBMS installation
    - Properties and specifics
  - **Connections:** A current session with the DBMS
    - URL, username, password, session context
  - **Statements:** SQL statements to be passed to DBMS via a connection
  - **Descriptions:** Records about tuples from a query or parameters of a statement
    - Number of attributes and respective types
    - Parameters in function calls

12.1 CLI

- An **environment** can host several **connections,** while a **connection** can host several **statements**



12.1 CLI

- When working with CLI, **following steps** have to be performed
  - Include the CLI function libraries and open database connections
  - Metadata about the database, tables, and columns can be retrieved
  - Define variables to contain SQL statement information
  - Execute the query and manipulate the result set in a (implicitly declared) cursor
  - Terminate statements, connections and the environment

12.1 CLI: Handle Concept

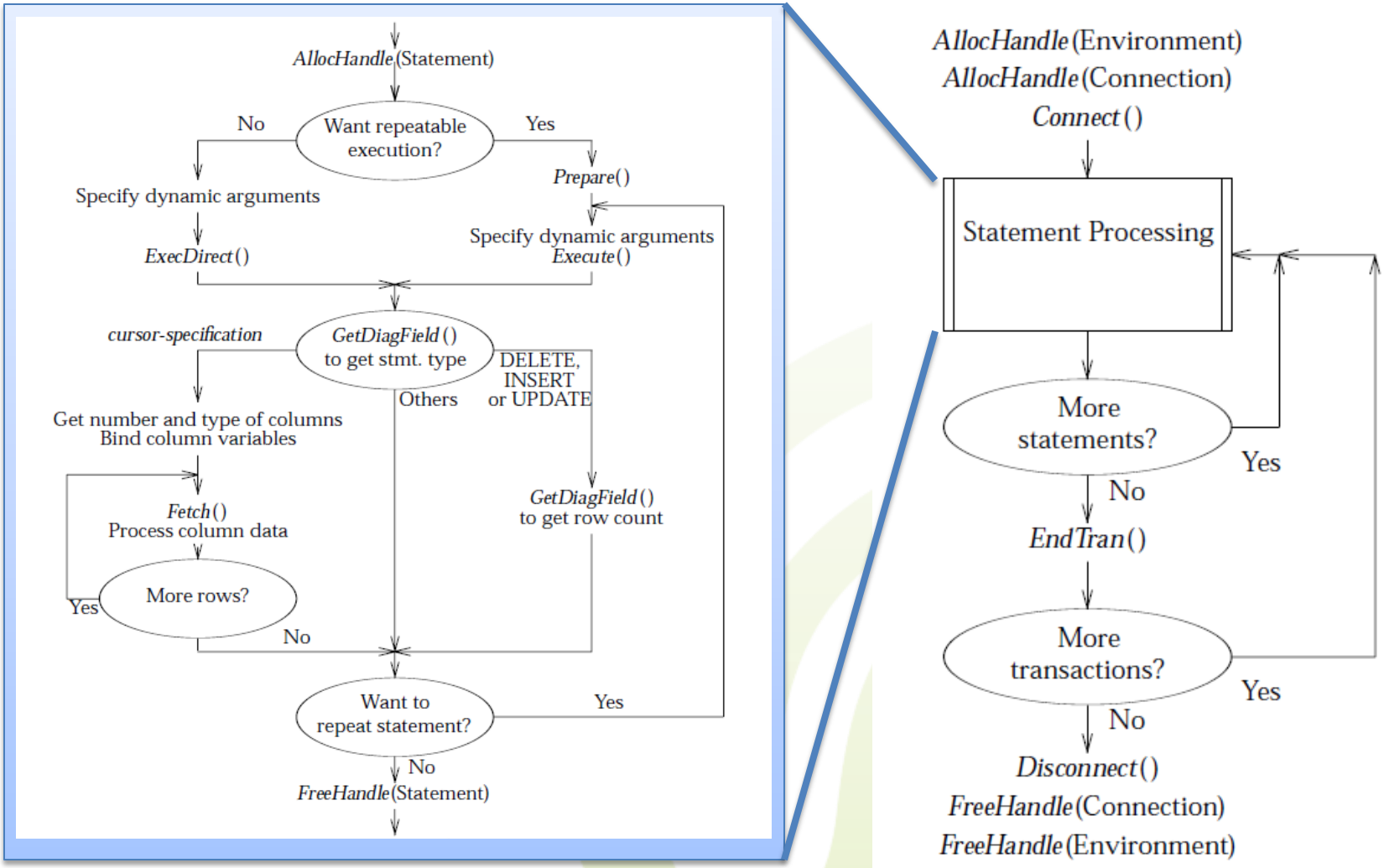
- Function **SQLAllocHandle(T, I, O)** is used to create data structures (variables), which are called environment, connection, and statement handles
  - **T: Handle type**, e.g., an environment, a connection, or a statement
  - **I: Input handle**, container structure at next higher level (statement < connection < environment)
  - **O: Output handle** (pointer to new handle)
- Example for handling statements:
  - SQLAllocHandle(SQL_HANDLE_STMT, myCon, myStat);
    - myCon is a previously created connection handle.
    - myStat is the name of the statement handle that will be created.

12.1 CLI: Handle Concept

- For details please read the manual… ☺
  - **Example in C:**

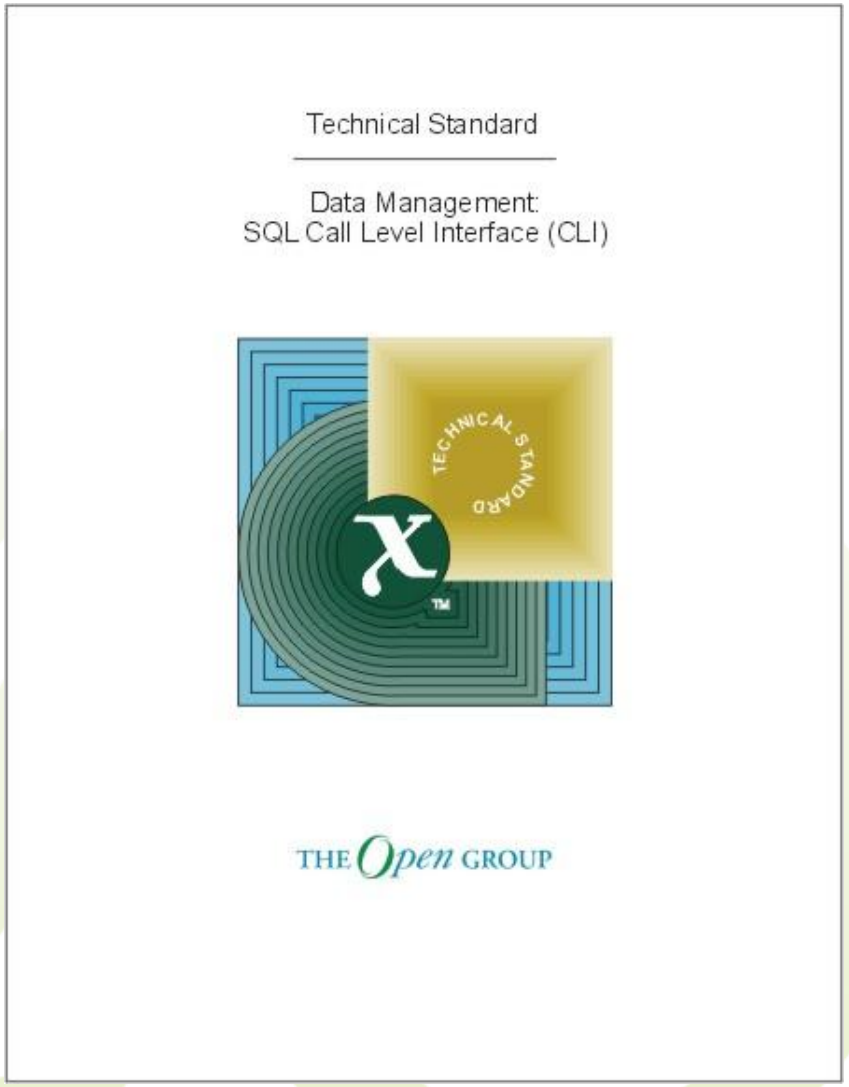
```
#include <sqlcli.h>
SQLRETURN ReturnCodeEnv;
SQLHENV EnvironmentHandle;
...
ReturnCodeEnv =
SQLAllocHandle(SQL_HANDLE_ENV,
SQL_NULL_HANDLE, &EnvironmentHandle);
```

12.1 CLI Basic Control Flow



12.1 CLI

- The complete technical standard is available freely from the Open Group
- Specification C451
- Over 300 pages…
- http://www.opengroup.org/products/publications/catalog/c451.htm



12.2 ODBC

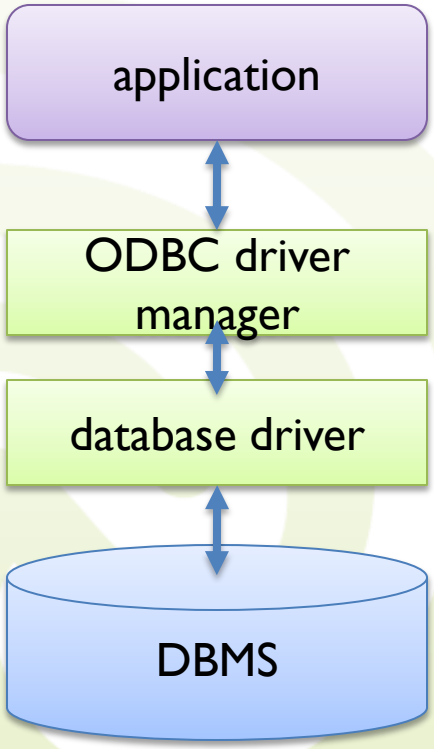
- The **Open Database Connectivity (ODBC)** provides a standardized application programming interface to DBMS using the CLI standard
  - Development driven by **Microsoft** in 1992, later versions aligned with X/Open and ISO/IEC
  - Builds on **several CLI specifications**, but does not implement the full SQL features
  - Central for the design was **independence** of programming language, operating system, and DBMS
  - Implements the standardized **middleware** concept

12.2 ODBC

- Basic idea: **The DBMS is virtualized**
  - The person with **specialized knowledge** to make the application logic work with the database is the driver developer and not the application programmer
  - Application developers write to a generic DBMS interface and **loadable drivers** map that logic to vendor-specific commands

12.2 ODBC

- Being a **middleware solution** a basic implementation of ODBC always contains…
  - A generic **ODBC "driver manager" library** to interpret the applications' commands
    - Defines standard types and features
  - And a set of **database drivers** to provide the DBMS-specific details
    - Each database vendors can write an individual driver to map ODBC commands



12.2 ODBC

- ODBC supports different **numbers of tiers** that have to be passed to access the databases
  - **Tier 1**
    - **Direct access** to database files by the database drivers (usually only used for desktop databases)
  - **Tier 2**
    - The database drivers prepare the requests and **pass them on to the DBMS** for execution (which is the normal case)
  - **Tier 3**
    - The database drivers prepare the requests and pass them to a specific **ODBC gateway** that manages the communication to the DBMS (e.g., via a low level interface) for execution

12.2 ODBC

- ODBC development has driven by the need of **easy application programming**
  - Originally in Microsoft's Visual Basic
  - But quickly adapted for use in C, C++ and other languages
- The ODBC architecture also has certain **drawbacks**
  - **Large client networks** may need a variety of drivers increasing the system-administration overhead
  - **Multi-tier ODBC drivers** can ease this problem

12.2 ODBC

- ODBC uses **standard CLI calls**…

- The concept of **handles** is used to set up the environment and connections
  - First, applications have to allocate an **environment handle** by calling SQLAllocEnv
  - Then, a handle for a **database connection** (SQLAllocConnect) has to be allocated before calling **connection functions** like SQLConnect

12.2 ODBC

- To process SQL statements…
  - an application must first acquire a **statement handle** by calling SQLAllocStmt
  - There is a function for **direct execution** of a SQL statement (SQLExecDirect) and functions to **prepare** and **execute statements** (SQLPrepare and SQLExecute)
  - An application can use **named cursors** by getting and setting the cursor name for a statement (SQLGetCursorName and SQLSetCursorName)
  - An application **retrieves a row** in a result set by calling SQLFetch
  - …

12.2 ODBC

- As part of ODBC's termination logic…
  - Every application should free statement handles using SQLFreeStmtclose
  - Close the connection and free the connection and environment handles by calling SQLDisconnect, SQLFreeConnect, and SQLFreeEnv
  - We won't go into more implementation details here, but consider the exact use for the case of JDBC

12.3 JDBC

- **JDBC** provides a standard **Java** library for **accessing tabular data**
  - Tabular data usually means a **relational DBMS**
  - API provides standard way to **connect** to a DB
  - API allows to perform **dynamic queries**
  - Method to create stored (**parameterized**) **queries**
  - Provides data types for Java/DB **impedance mismatch**
    - **Result sets** with rows and columns
    - Methods for accessing table **meta data**
  - Provides functionality **independent** of chosen DBMS

12.3 JDBC

- JDBC does **not standardize SQL**
  - SQL statements are treated as **Java strings**
  - In case of full dynamic SQL, sometimes excessive **string manipulation** is necessary
  - If DBMS uses different/extended SQL syntax, this has to be considered by the programmer
- JDBC is not an acronym, but a registered product **trademark** by Sun Microsystems
  - However, usually, it is assumed that it stands for **J**ava **D**atabase **C**onnectivity

12.3 JDBC

- Why not just use ODBC?
  - ODBC is based on **binary libraries** (usually written in C)
    - Native calls necessary
    - Not platform-independent which is one of Java's goals
  - 1:1 translation from ODBC to Java does not work as ODBC heavily **relies on pointers**
  - ODBC API is more **complex** and **littered** (and thus harder to learn and use)
    - For example, programmer needs to worry about byte alignment and advanced connection properties explicitly
  - Intention was to create a **"pure" Java** and fully portable API
    - No installation required, JDBC can easily be bundled into the application archive

12.3 JDBC

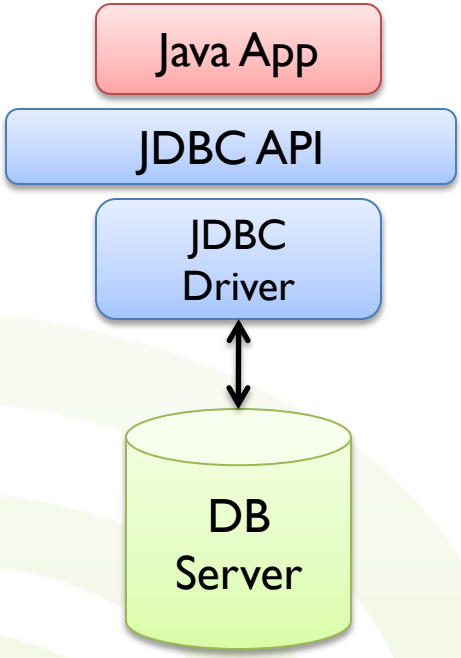
- JDBC is composed of **two primary components**
- **JDBC API:** An **programming interface** for database connectivity.
  - Written in **100% pure Java**
  - Completely **independent** of platform, vendor, and DBMS
  - Provided by the Sun in its **Java SDK by default**
    - Usually to be found in java.sql and javax.sql

12.3 JDBC

- **JDBC driver**
  - Implementation of the respective API interface, responsible for **communication** with the **database**
  - Interface implementation in Java, but may depend on any amount of binary libraries, middleware, or other tools
  - Heavily dependent on the used DBMS
  - Usually provided by the **DB vendor**

12.3 JDBC

- General Architecture
  - Java application uses API
  - API uses driver
  - Driver communicates with DB
- If you change the DBMS, you need to
  - Provide a new driver
  - Change configuration of driver
  - Assuming the SQL syntax is compatible, you are done
    - If not, you are in trouble…



12.3 JDBC: Versions

- There are several versions of JDBC, each with improved functionality

Version	Year	Java Version
JDBC 4.0	2006	Java 6
JDBC 3.0	**2001**	**Java 1.4 & Java 5**
JDBC 2.1	1999	Java 1.3
JDBC 1.2	1997	Java 1.1

  - JDBC drivers are written for a specific JDBC version
    - Driver should match the JDBC version
    - However, most features also work with outdated drivers
  - JDBC 3.0 is most common
  - However, JDBC 4 is gaining momentum as it contains many useful features and improvements

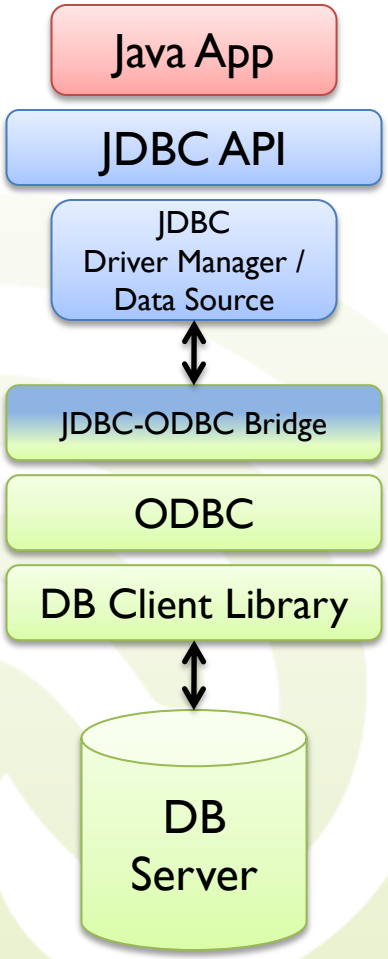
12.3 JDBC: Levels

- Beside versions, there are JDBC levels
  - Comparable to ODBC tiers
  - For each level, there are different drivers
    - Be careful when picking your driver! You need the right version and correct level!
  - All levels offer the same functionality (i.e., API is the same), but use different means of driver implementation and communication with the DBMS
    - Different performance and portability properties

12.3 JDBC: Levels

- **Level 1: JDBC/ODBC bridge**
  - JDBC driver just translates requests to ODBC calls
    - **Performance overhead** due to translation
  - Needs correctly installed ODBC drivers on every client machine
    - Distribution difficult
    - ODBC drivers are not platform-independent

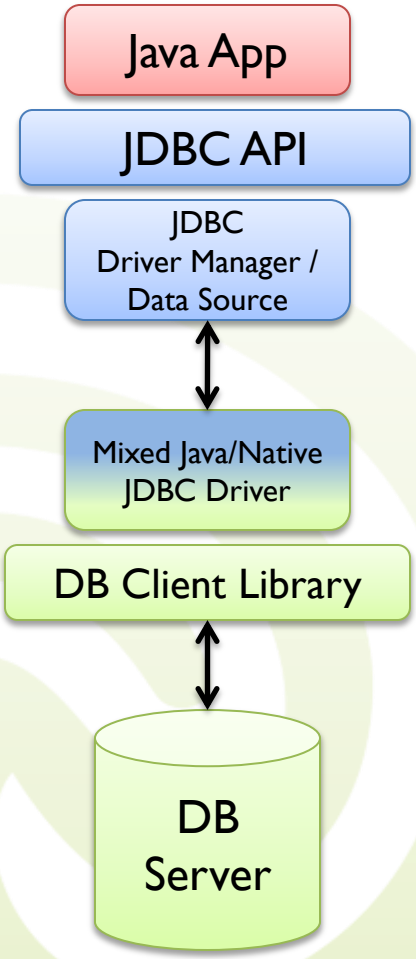
JDBC Level 1 (ODBC Bridge)



12.3 JDBC: Levels

- **Level 2: Native API**
  - JDBC driver uses native calls to connect to a proprietary client software which handles DB connections
    - e.g. ORACLE client (which is a 1.7 GB installation)
  - Difficult to port and with deployment problems
  - Often used as cheap-and-dirty solution for older systems
    - Also, may be a good idea if application is running on the same machine as the DBMS

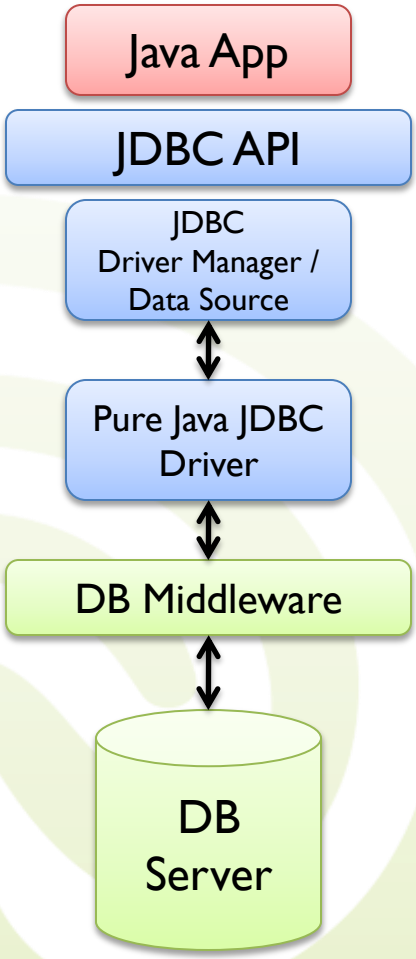
JDBC Level 2 (Native API)



12.3 JDBC: Levels

- **Level 3: Middleware**
  - JDBC driver communicates with a **middleware** software instead of the DBMS
  - Often used for **large-scale enterprise applications** in a multi-tier-environment
  - Vendor specific translation may happen at the middleware
    - Just one client driver for any used DBMS
  - Middleware encapsulates the actual DBMS
    - Useful for advanced clustering, extended security, logging, caching, pooling, etc..

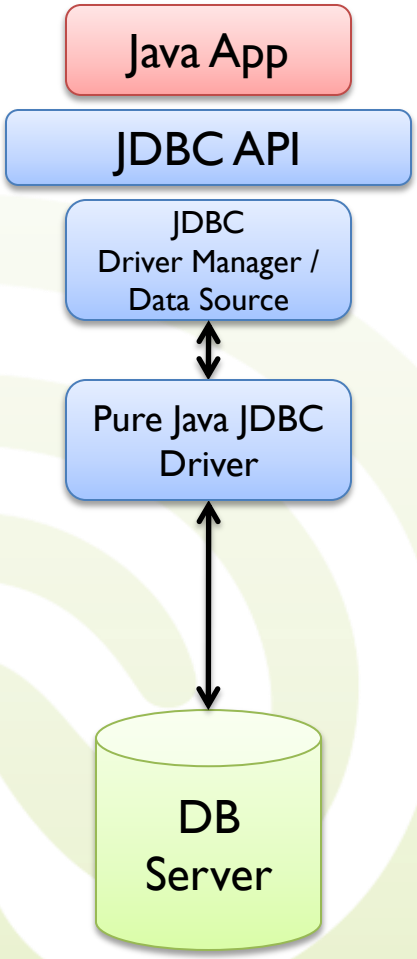
JDBC Level 3 (Middleware)



12.3 JDBC - Levels

- **Level 4: Direct pure Java**
  - Driver purely written in Java
    - No call translation
    - No installation, no deployment problems
    - Full portability due to platform-independence
  - Driver connects directly to the DBMS
    - You need a different driver for each different DBMS
    - Superior performance in remote scenarios
    - For access to a local DBMS, Level 1 might be better

JDBC Level 4 (Pure Java)



12.3 JDBC

- 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

JDBC: Create a Connection

Detour

- A connection creates a query session for a given **user** within a specific **DBMS**
  - Each connection may have specific **properties**
  - The DBMS server is specified using an **URL**
  - Common URL format is
    - jdbc:[driverAlias]:[driverParameters]
    - DB2 Level 4:
      - **jdbc:db2://[server][:port]/[db-name]**
      - Example: jdbc:db2://dblab.ifis.cs.tu-bs.de:50000/DBLAB
    - MySQL Level 4:
      - **jdbc:mysql://[server][:port]/[database]**
      - Example: jdbc:mysql://dblab.ifis.cs.tu-bs.de:3306/rdb1
    - Further URL formats for most DBMS:
      - http://www.redmountainsw.com/wordpress/archives/jdbc-connection-urls

JDBC: Create a Connection

Detour

- To create a connection, the **DriverManager** is used
  - Returns a **Connection** object

```
public Connection getConnection(Properties connectionProps)
    throws SQLException {
  return DriverManager.getConnection(
     "jdbc:db2://" +
     connectionProps.getProperty("server") + ":" +
       connectionProps.getProperty("port") + "/" +
       connectionProps.getProperty("database"),
     connectionProps.getProperty("user"),
     connectionProps.getProperty("password")
  );
}
```

JDBC: Create a Connection

- The Connection object coordinates the whole database-application interaction
  - Contains DBMS instance metadata
  - Responsible for providing query statements
- Driver matching the driver alias in the URL must be contained in the classpath

```
Exception in thread "main" java.sql.SQLException: No
suitable driver found for jdbc:db2://dblab.ifis.cs.tu-
bs.de:50000/dblab
```

JDBC: Create a Connection

Detour

- Using the connection object, the DB **metadata** can be accessed
  - DBMS **name**, DBMS **version**, DBMS installation **properties**, available **schemas**, available **tables**, **columns** for a table, **primary keys** for a given table, and many **many more**

```
DatabaseMetaData metaData =
    connection.getMetaData();
String dbmsName =
    metaData.getDatabaseProductName();
ResultSet schemas =
    metaData.getSchemas();
```

12.3 JDBC: Statements

Detour

- To actually execute a SQL statement, JDBC provides Statement objects
  - Any kind of statement is created by the **Connection** object
  - When a query is executed, a **ResultSet** is returned
    - **ResultSet** encapsulates SQL result tables
    - Result is stored on the server and transferred to the client **row by row**

12.3 JDBC: Statements

Detour

- There are three Statements facilities provided by JDBC
  - **Statement**
    - Used for simple or very rare statements
    - A real **dynamic fashion** of executing SQL
    - The whole SQL statements is provided as a **String**
  - **PreparedStatement**
    - Used for frequent statements
    - **Semi-dynamic** statements
    - Statement is provided as **parameterized String**
    - For each execution, **parameters are replaced by values**
    - Usually, **performance is much higher** due to lower overhead and better query plan optimization
  - **CallableStatement**
    - Used to execute server-side stored procedure (UDF)

12.3 JDBC: Statements

Detour

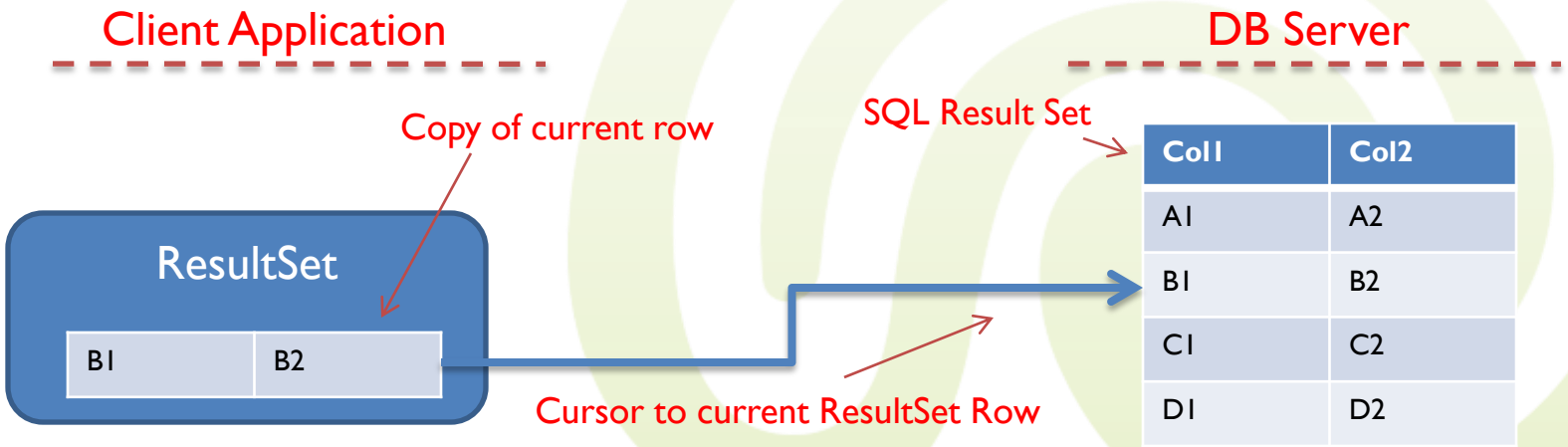
- Using **simple statements**
  - Create a **statement object** with the connection
  - Call one of the three execution methods
  - **executeQuery**(String query):
    - Use for **SELECT** queries
    - Returns a ResultSet
  - **executeUpdate**(String query):
    - Use for any DDL or data changing statements (**INSERT, UPDATE, DELETE**)
    - Returns an integer with number of affected rows
  - **execute**(String query):
    - Advanced feature for multi-ResultSet queries

```
Statement stmt =
    connection.createStatement();
ResultSet rs = stmt.executeQuery(
    "SELECT count(*) FROM IMDB.title");
```

12.3 JDBC: Result Sets

Detour

- **ResultSet** encapsulate a query result table
  - Rows are retrieved one after another from the server
    - Inspired by (but not compatible to) the Iterator-Interface
    - A **cursor** is held pointing the actual row in the server-side result set
      - like row cursor EmbeddedSQL
  - At first, the result set points **before the first row** (i.e. to no row at all)
  - For each row, the column values may be retrieved individually by **column getter methods**



12.3 JDBC: Statements

- After all result set rows have been read, the statement is marked complete
  - Statements and Results are usually **garbage collected** by Java
  - However, it is highly recommended to **manually close** (using the close()-method) statements (and thus result sets) to save system resources in the mean time

12.3 JDBC: Result Sets

Detour

- Default ResultSet only allows moving the cursor forward and is read only
  - TYPE_FORWARD_ONLY and CONCUR_READ_ONLY
- Movement operations for TYPE_FORWARD_ONLY
  - next(): Moves the cursor to the next row
    - Returns true if successful, false if there is no next row
  - afterLast(): Move the cursor after the last row

```
// iterate over all result set rows
while (rs.next()) {
   // do something with the current row
}
```

12.3 JDBC: Result Sets

Detour

- There are multiple getter methods for the column values
  - Columns may be either accessed by **name** or by **number (starting at 1)**
  - There are getters for each **variable type**
    - Each SQL data type is mapped to a Java data type
    - All getters are named get*TYPE*() (e.g. getInt(), getDouble(), …)
- Example: Return all IDs and realNames of heroes

```
ResultSet rs = stmt.executeQuery(
  "SELECT  id, realName FROM heroes");
while(rs.next()) {
    int id = rs.getInt(1); // or: rs.getInt("id")
    String realName = rs.getString("realName");
    System.out.println(id + ":" + realName);
}
```

12.3 JDBC: Data Types

Detour

Example: Direct JDBC data types for DB2

Java Data Type	SQL Data Type
short, boolean, byte	**SMALLINT**
int, java.lang.Integer	**INTEGER**
long, java.lang.Long	DECIMAL(19,0)
long, java.lang.Long	BIGINT
float, java.lang.Float	REAL
double, java.lang.Double	**DOUBLE**
java.math.BigDecimal	DECIMAL(p,s)
java.lang.String	CHAR(n)
java.lang.String	GRAPHIC(m)
java.lang.String	**VARCHAR(n)**
java.lang.String	VARGRAPHIC(m)
java.lang.String	CLOB(n)

12.3 JDBC: Data Types

Detour

Java Data Type	SQL Data Type
byte[]	CHAR(n) FOR BIT DATA
byte[]	VARCHAR(n) FOR BIT DATA
byte[]	BLOB(n)
byte[]	ROWID
java.sql.Blob	BLOB(n)
java.sql.Clob	CLOB(n)
java.sql.Clob	DBCLOB(m)
java.sql.Date	**DATE**
java.sql.Time	**TIME**
java.sql.Timestamp	**TIMESTAMP**
java.io.ByteArrayInputStream	BLOB(n)
java.io.StringReader	CLOB(n)
java.io.ByteArrayInputStream	CLOB(n)
com.ibm.db2.jcc.DB2RowID	ROWID
java.net.URL	DATALINK

12.3 JDBC: Result Sets

Detour

- Different ResultSet types may be used changing **navigation** and **update options**
  - It depends on the DBMS and the JDBC drivers which options / combinations are possible
- **Navigation** options
  - TYPE_FORWARD_ONLY
    - Only forward movement is possible
    - DBMS may materialize result set incrementally, it is not exactly clear how up-to-date the row is

  - TYPE_SCROLL_INSENSITY
    - Usually only works if DBMS supports rownums
    - Cursor may forward, backward, or directly to a particular row
    - While the result set is open, changes in the underlying data are not visible

12.3 JDBC: Result Sets

Detour

- Some additional navigation methods for TYPE_SCROLL_INSENSITY
  - **Previous()** Moves cursor to the previous row
  - **beforeFirst()**: Moves cursor before the first row
  - **relative(int x)**: Moves cursor x rows forward (or backward if x is negative)
  - **absolute(int x)**: Moves cursor to the given absolute row number

– TYPE_SCROLL_SENSITIVE

- Same as TYPE_SCROLL_INSENSITY
- BUT changes to the underlying data are reflected in the result (i.e. rows are always up-to-date)
- May have bad performance

12.3 JDBC: Result Sets

Detour

- **Update** options
  - CONCUR_READ_ONLY
    - Results can only be read
    - Unlimited concurrency

  - CONCUR_UPDATABLE
    - Result set may be **updated** (i.e. row/column values changed, rows inserted, etc.; similar to updateable view)
    - May **degenerate performance** in massively concurrent applications due to lock contention thrashing (see RDB2)

12.3 JDBC: Result Sets

Detour

- Different result set types are declared during statement initialization
  - Connection.**createStatement** (int resultSetTyp, int resultSetConcurrency)
    - int values declared as constants in ResultSet

```
Statement stmt =
   connection.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY);
```

12.3 JDBC: Result Sets

Detour

- If a result set is updatable, you may use a set of **update*TYPE*(column, value)** methods
  - Similar to get methods
  - Column may either be references by **name or number**
  - All update commands relate to the current row
  - Updates are only performed after you call **updateRow()**
    - There may be multiple update commands followed by one updateRow()
    - If you want to cancel the updates, call **cancelRowUpdates()**
    - If you move to another row without updateRow(), nothing happens

```
Statement stmt = connection. createStatement(ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_UPDATABLE);
ResultSet rs= stmt.executeQuery
    ("SELECT  id, realName FROM heroes");
while (rs.next()) {
    rs.updateString("realName", rs.getString("realName").toUpperCase());
    rs.updateRow();
}
```

12.3 JDBC: Result Sets

- Furthermore, new rows may be **inserted**
  - There is a special row for inserting ("**insert row**")
  - To start an insertion, move cursor to the insert row by using **moveToInsertRow()**
  - While being in the insert row, you may only use get and update methods
  - You must update all columns
  - When all columns are updated, call **insertRow()** to commit the insert
    - Cursor will move to old position
    - There is no way to know where the row is inserted nor if it will appear in the current result set
  - Of course, this only works with updatable result sets

12.3 JDBC: Result Sets

Detour

- To delete the current row, call **deleteRow()**
  - Cannot be performed when being in the insert row

```
Statement stmt = connection. createStatement
    (ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
ResultSet rs= stmt.executeQuery
    ("SELECT  id, realName FROM heroes");
//
rs.moveToInsertRow();
rs.updateInt(1, 999);
rs.updateString(2, "Peter Parker")
rs.insertRow();
//
while (rs.next()) {
    rs.deleteRow();
}
```

12.3 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

12.3 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

12.3 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** than when using simple statements

12.3 JDBC: Prepared Statements

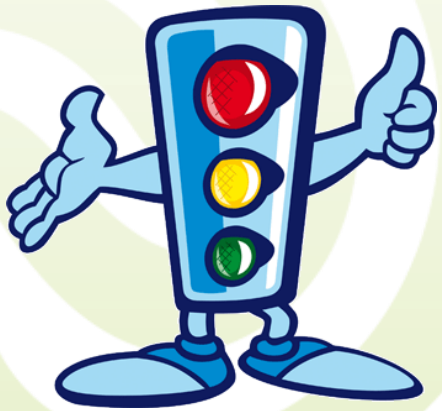
- To supply values for the placeholders, use **set***TYPE*(number, value) methods
  - Like for the get and update methods, there are set methods for any data type
    - Placeholders are referenced by the position in the SQL string starting with 1

```
PreparedStatement moviesInYears = connection. 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
}
```

12.3 JDBC: Transactions

Detour

- Of course, you may use **transactions** within JDBC
  - Depending on the DBMS, transactions are either enabled or disabled by default
    - Transactions disabled means that auto-commit mode is enabled and vice versa
  - Use **setAutoCommit**(boolean switch) to change transactional behavior
    - **true**: Use no transactions (every statement is an auto transaction)
    - **false**: Use transactions



12.3 JDBC: Transactions

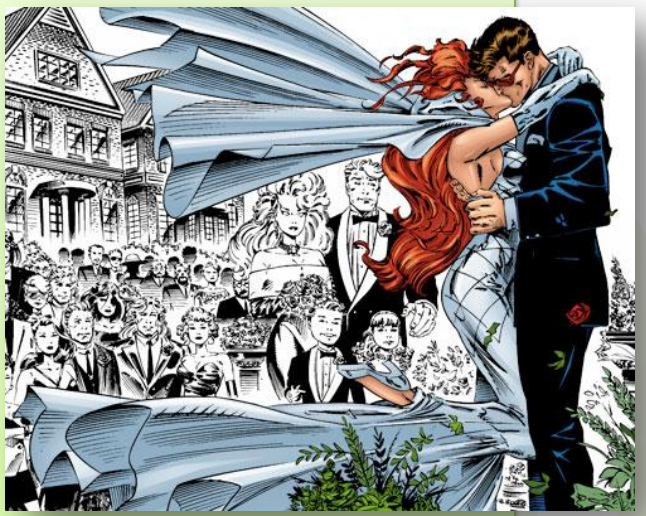
Detour

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

12.3 JDBC: Transactions

Detour

```
connection.setAutoCommit(false);
PreparedStatement changeName= connection. prepareStatement
      ("UPDATE  hero SET name=?  WHERE name=?");
//
changeName.setString(1, "Jean Grey-Summers");
changeName.setString (2, "Jean Grey");
changeName.executeUpdate();
//
changeName.setString(1, "Scott Grey-Summers");
changeName.setString (2, "Scott Summers");
changeName.executeUpdate();
//
connection.commit();
```



Next Lecture

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