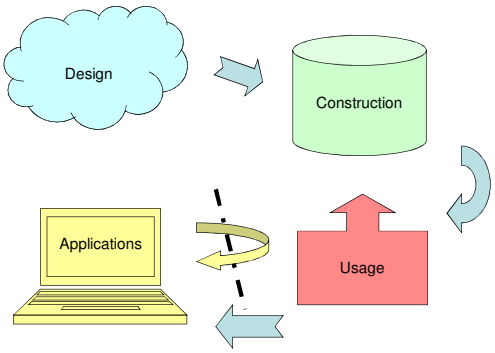
Database Applications

SQL/PSM
Embedded SQL
JDBC

Course Objectives



Course Objectives – Interfacing

When the course is through, you should

- Know how to connect to and use a database from external applications.
  - … using JDBC

External applications

- Normally, databases are not manipulated through a generic SQL interpreter (like iSQL*Plus). Rather they are used as a building block in larger applications.
- SQL is not well suited for building full-scale applications – we need more computational power!
  - Control structures, ordinary arithmetic, input/output, etc.

Mixing two worlds

- Mixing SQL with ordinary programming constructs is not immediately straightforward.
  - ”The impedance mismatch problem” – differing data models
    - SQL uses the relational data model.
    - Ordinary imperative languages cannot easily model sets and relations.
- Various approaches to mixing SQL and programming solve this problem more or less gracefully.

Two approaches

- We have SQL for manipulating the database. To be able to write ordinary applications that use SQL, we can either
  - Extend SQL with ”ordinary” programming language constructs.
    - SQL/PSM, PL/SQL
  - Extend an ordinary programming language to support database operations through SQL.
    - Embedded SQL, SQL/CLI (ODBC), JDBC, …

SQL/PSM

- PSM = ”persistent, stored modules”
- Standardized extension of SQL that allows us to store procedures and functions as database schema elements.
- Mixes SQL with conventional statements (if, while, etc.) to let us do things that we couldn’t do in SQL alone.
  - PL/SQL is Oracle-specific, and very similar to PSM (only minor differences).

Basic PSM structure

To create a procedure:

```
CREATE PROCEDURE name (
   parameter list )
local declarations
body;
```

To create a function:

```
CREATE FUNCTION name (
   parameter list )
   RETURNS type
local declarations
body;
```

Example:

```
CREATE PROCEDURE AddDBLecture (
   IN day VARCHAR(9),
   IN hour INT,
   IN room VARCHAR(30)
)
INSERT INTO Lectures
VALUES ('TDA356', 2, day, hour, room);
```

Used like a statement:

```
CALL AddDBLecture('Monday', 13, 'VR');
```

Parameters

- Unlike the usual name-type pairs in languages like C, PSM uses mode-name-type triples, where mode can be:
  - IN: procedure uses the value but does not change it.
  - OUT: procedure changes the value but does not read it.
  - INOUT: procedure both uses and changes the value.

Procedure body

- The body can consist of any PSM statement, including
  - SQL statements (INSERT, UPDATE, DELETE).
  - setting the values of variables (SET).
  - calling other procedures (CALL)
  - …

More complex example:

```
CREATE FUNCTION RateCourse (
   IN c CHAR(6)) RETURNS CHAR(10)
DECLARE totalNrSt INTEGER;
BEGIN
   SET totalNrSt =
      (SELECT SUM(nrStudents)
       FROM GivenCourses
       WHERE course = c);
   IF totalNrSt < 50 THEN RETURN 'unpopular'
    ELSEIF totalNrSt < 150 THEN RETURN 'average'
    ELSE RETURN 'popular'
   END IF;
END;
```

DECLARE used to declare local variables.

Number of students reading course c over the whole year.

BEGIN and END used to group statements (cf. { … } in e.g. Java)

Used whenever we can use a value:

```
SELECT code, RateCourse(code)
FROM Courses;
```

Setting values of variables

- Two ways of assigning values to variables:
  - SET works like ”ordinary” assignment:

```
SET totalNrSt = (SELECT SUM(nrStudents)
                 FROM   GivenCourses
                 WHERE  course = c);
```

  - We can also assign values as part of a query using **SELECT … INTO**:

```
SELECT SUM(nrStudents), AVG(nrStudents)
INTO   totalNrSt, avgNrSt
FROM   GivenCourses
WHERE  course = c;
```

Using **SELECT … INTO** we can assign multiple values at the same time, if the result of the query is a tuple.

Control Flow in SQL/PSM

- SQL/PSM has conditionals and loops like an imperative programming language:
  - Conditional:
    - **IF … THEN … ELSEIF … ELSE … END IF;**
  - Loops:
    - **WHILE … DO … END WHILE;**
    - **REPEAT … UNTIL … END REPEAT;**
    - **LOOP … END LOOP;**

LOOP works like while (true) …

  - Loops can be named by prepending a name and a colon to the loop. Exiting can then be done explicitly:

```
loop1: LOOP
 …
 LEAVE loop1;
 …
END LOOP;
```

Returning values

- The keyword **RETURN** sets the return value of a function.
  - Unlike e.g. Java, **RETURN** *does not* terminate the execution of the function.
  - Example:

```
CREATE FUNCTION ToRating (IN nrSt INT)
 RETURNS CHAR(10)
BEGIN
 IF nrSt < 50 THEN RETURN 'unpopular' END IF;
 RETURN 'popular';
END;
```

Returning the value doesn't happen until END, so the result will always be 'popular'.

”Exceptions” in SQL/PSM

- SQL/PSM defines a magical variable SQLSTATE containing a 5-digit string.
- Each SQL operation returns a status code into this variable, thus indicating if something goes wrong.
  - Example:
    - 00000 = ”OK”
    - 02000 = ”No tuple found”

Handling exceptions

- We can declare condition indicators for when something goes wrong (or right):

```
DECLARE NotFound CONDITION FOR SQLSTATE '02000';
```

- These can be used as ordinary tests, or using exception handlers:

```
IF NotFound RETURN NULL END IF;

DECLARE EXIT HANDLER FOR NotFound RETURN NULL;
```

More complex example, with exceptions:

```
CREATE FUNCTION RateCourse (
   IN c CHAR(6)) RETURNS CHAR(10)
DECLARE totalNrSt INTEGER;
DECLARE NotFound CONDITION FOR SQLSTATE '02000';
BEGIN
  DECLARE EXIT HANDLER FOR NotFound RETURN NULL;
  SET totalNrSt =
     (SELECT SUM(nrStudents)
      FROM GivenCourses
      WHERE course = c);
  IF totalNrSt < 50 THEN RETURN 'unpopular'
   ELSEIF totalNrSt < 150 THEN RETURN 'average'
   ELSE RETURN 'popular'
  END IF;
END;
```

If we should end up in a state where Not_Found is true, i.e. the course c doesn't exist, we return NULL.

Quiz!

We can use queries that return a single value, or a single tuple (using SELECT … INTO), but how use queries that return more than one row?

Key idea is to not return the rows themselves, but rather a pointer that can be moved from one tuple to another (cf. iterators in Java).
SQL/PSM calls these *cursors*.

Cursors

- Declaring
  - **DECLARE *name* CURSOR FOR *query***
- Initializing
  - **OPEN *name***
- Taking values from
  - **FETCH *name* INTO *variables***
- Ending
  - **CLOSE *name***

Example (declaration part):

```
CREATE PROCEDURE FixEarlyLectures (
    IN theCourse CHAR(6),
    IN thePeriod INT)
DECLARE theHour INT;
DECLARE theDay VARCHAR(9);
DECLARE theRoom VARCHAR(30);
DECLARE NotFound CONDITION FOR
   SQLSTATE '02000';
DECLARE c CURSOR FOR
   (SELECT hour, weekday, room
    FROM Lectures
    WHERE course = theCourse
      AND period = thePeriod);
```

Used to hold values fetched by the cursor.

Find all lectures of the given course.

Example continued (logic part):

```
BEGIN
  OPEN c;
  fixLoop: LOOP
    FETCH c INTO theHour, theDay, theRoom;
    IF NotFound THEN LEAVE fixLoop END IF;
    IF theHour < 10 THEN
      IF NOT EXISTS
        (SELECT * FROM Lectures
         WHERE period = thePeriod AND day = theDay
           AND hour = 10 AND room = theRoom)
      THEN
        UPDATE Lectures SET hour = 10
        WHERE course = theCourse AND day = theDay
          AND period = thePeriod AND room = theRoom;
      END IF;
    END IF;
  END LOOP;
  CLOSE c;
END;
```

Check if last **FETCH** failed to find a tuple.

What does it do?

Summary SQL/PSM

- Procedures, functions
  - Parameters, local declarations
  - Returning values
  - Exceptions and handling
  - Calling (**CALL** procedures, use functions as values)
- Assigning to variables
  - **SET**
  - **SELECT … INTO …**
- Cursors
  - Declaring, fetching values

Two approaches

- We have SQL for manipulating the database. To be able to write ordinary applications that use SQL, we can either
  - Extend SQL with ”ordinary” programming language constructs.
    - SQL/PSM, PL/SQL
  - Extend an ordinary programming language to support database operations through SQL.
    - Embedded SQL, SQL/CLI (ODBC), JDBC, …

Yet again two approaches

- Extending a programming language with support for database manipulation can be done in two ways:
  - Embedding SQL within the source code of the host language.
    - Embedded SQL
  - Adding native mechanisms to the host language for interfacing to a database
    - Call-level: SQL/CLI (C), JDBC (Java), many more…
    - High-level: HaskellDB, LINQ (C#)

Embedded SQL

- Key idea: Use a preprocessor to turn SQL statements into procedure calls within the host language.
- All embedded SQL statements begin with EXEC SQL, so the preprocessor can find them easily.
- By the SQL standard, implementations must support one of: ADA, C, Cobol, Fortran, M, Pascal, PL/I.

Shared variables

- To connect the SQL parts with the host language code, some variables must be shared.
  - In SQL, shared variables are preceded by a colon.
  - In the host language they are just like any other variable.
- Declare shared variables between

```
EXEC SQL BEGIN DECLARE SECTION;
 /* declarations go here */
EXEC SQL END DECLARE SECTION;
```

Example (in C):

```
EXEC SQL BEGIN DECLARE SECTION;
  char theCourse[7];
  int  thePeriod;
  char theTeacher[41];
EXEC SQL END DECLARE SECTION;

/* Read in values for theCourse and thePeriod
   from stdin or somewhere else. */

EXEC SQL SELECT teacher INTO :theTeacher
         FROM GivenCourses
         WHERE course = :theCourse
           AND period = :thePeriod;

/* Do something with theTeacher */
```

41-char array to hold 40 chars + end marker.

SELECT … INTO just like in SQL/PSM

Embedded queries

- Same limitations as in SQL/PSM:
  - **SELECT … INTO** for a query guaranteed to return a single tuple.
  - Otherwise, use cursors.
    - Small syntactic difference between SQL/PSM and Embedded SQL cursors, but the key ideas are identical.

Example (in C again):

```
EXEC SQL BEGIN DECLARE SECTION;
  char theCourse[7];
  char theName[51];
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE c CURSOR FOR
  (SELECT * FROM Courses);

EXEC SQL OPEN CURSOR c;
while(1){
  EXEC SQL FETCH c INTO :theCourse, :theName;
  if (NOT FOUND) break;
  /* Print theCourse and
     theName to stdout */
}
EXEC SQL CLOSE CURSOR c;
```

Only difference from SQL/PSM is the word CURSOR when opening and closing a cursor.

Predefined C macro:
#define NOT FOUND = strcmp(SQLSTATE, "02000")

Need for dynamic SQL

- Queries and statements with EXEC SQL can be compiled into calls for some library in the host language.
- However, we may not always know at compile time in what ways we want to manipulate the database. What to do then?

Dynamic SQL

- We can prepare queries at compile time that will be instantiated at run time:
  - Preparing:

    ```
    EXEC SQL PREPARE name FROM query-string;
    ```

  - Executing:

    ```
    EXEC SQL EXECUTE name;
    ```

  - Prepare once, execute many times.

Example: A generic query interface

```
EXEC SQL BEGIN DECLARE SECTION;
  char theQuery[MAX_LENGTH];
EXEC SQL END DECLARE SECTION;
while(1){
  /* issue SQL prompt */
  /* read user query into array theQuery */
  EXEC SQL PREPARE q FROM :theQuery;
  EXEC SQL EXECUTE q;
}
```

Summary: Embedded SQL

- Write SQL inline in host language code.
  - Prepend SQL with **`EXEC SQL`**
- Shared variables.
  - Prepend with colon in SQL code.
- No inherent control structures!
  - Uses control structures of the host language.
- Compiled into procedure calls of the host language.

JDBC

- JDBC = Java DataBase Connectivity
- JDBC is Java's *call-level interface* to SQL DBMS's.
  - A library with operations that give full access to relational databases, including:
    - Creating, dropping or altering tables, views, etc.
    - Modifying data in tables
    - Querying tables for information
    - …

JDBC Objects

- JDBC is a library that provides a set of classes and methods for the user:
  - DriverManager
    - Handles connections to different DBMS. Implementation specific.
  - Connection
    - Represents a connection to a specific database.
  - Statement, PreparedStatement
    - Represents an SQL statement or query.
  - ResultSet
    - Manages the result of an SQL query.

Registering a driver

- The **`DriverManager`** is a global class with static functions for loading JDBC drivers and creating new connections.
- Load the Oracle JDBC driver:

```
DriverManager.registerDriver(
  new oracle.jdbc.driver.OracleDriver());
```

  - Will be done for you on the lab.

Getting connected

- A **`Connection`** object represents a connection to a specific database:

```
private static final String HOST =
  "delphi.medic.chalmers.se";
private static final String PORT = "1521";
private static final String SID = "medic1";
private static final String USER = username;
private static final String PWD = password;

Connection myCon =
  DriverManager.getConnection("jdbc:oracle:thin:@" +
    HOST + ":" + PORT + ":" + SID, USER, PWD);
```

  - Will also be done for you on the lab, except username and password.

Statements

- A **`Statement`** object represents an SQL statement or query, including schema-altering statements.
- A **`Statement`** object represents one statement at a time, but may be reused.

```
Statement myStmt = myCon.createStatement();
```

A statement is associated with a particular connection

Using statements

- **`Statement`** objects have two fundamental methods:
  - **`ResultSet executeQuery(String query)`**
    - Given a string, which must be a query, run that query against the database and return the resulting set of rows.
  - **`int executeUpdate(String update)`**
    - Given a string, which must be a non-query, run that update against the database.
    - Note that a JDBC update is not an SQL update, but rather an SQL modification (which could be an update).

Example:

```
String code, name;
/* Get values for code and name from user */

String myInsertion =
  "INSERT INTO Courses VALUES ('" +
  code + "', '" + name + "')";

Statement myStmt = myCon.createStatement();

myStmt.executeUpdate(myInsertion);
```

Note the inserted single quotes.

Has return type int (the number of rows that were changed)

Quiz!

What's wrong with the program below?

```
String code;
/* Get value for code from user */

String myQuery =
  "SELECT name" +
  "FROM Courses" +
  "WHERE code = " + code;

Statement myStmt = myCon.createStatement();
ResultSet rs = myStmt.executeQuery(myQuery);
/* Do something with result. */
```

No spaces! "`SELECT nameFROM CoursesWHERE code = …`"

No single-quotes either.

Exceptions in JDBC

- Just about anything can go wrong!
  - Syntactic errors in SQL code.
  - Trying to run a non-query using **`executeQuery`**.
  - Permission errors.
  - …
- Catch your exceptions!

```
try {
     // database stuff goes in here
} catch (SQLException e) { … }
```

Executing queries

- The method **`executeQuery`** will run a query against the database, producing a set of rows as its result.
- A ResultSet object represents an interface to this resulting set of rows.
  - Note that the **`ResultSet`** object is not the set of rows itself – it just allows us to access the set of rows that is the result of a query on some **`Statement`** object.

ResultSet

- A ResultSet is very similar to a cursor in SQL/PSM or Embedded SQL.
  - **`boolean next()`**
    - Advances the ”cursor” to the next row in the set, returning false if no such rows exists, true otherwise.
  - **`X getX(i)`**
    - **`X`** is some type, and **`i`** is a column number (index from 1).
    - Example: `rs.getInt(1)`

      returns the integer value of the first column of the current row in the result set **`rs`**.

ResultSet is not a result set!

- Remember a **`ResultSet`** is more like a cursor than an actual set – it is an interface to the rows in the actual result set.
- A **`Statement`** object can have one result at a time. If the same **`Statement`** is used again for a new query, any previous **`ResultSet`** for that **`Statement`** will no longer work!

Courses

code	*name*
TDA357	Databases
TIN090	Algorithms

GivenCourses

course	*per*	*teacher*
TDA357	2	Niklas Broberg
TDA357	4	Rogardt Heldal
TIN090	1	Devdatt Dubhashi

TDA357 Databases
  Period 2: Niklas Broberg
  Period 4: Rogardt Heldal
TIN090 Algorithms
  Period 1: Devdatt Dubhashi

Quiz!

What will the result be?

```
Statement myStmt = myCon.createStatement();
ResultSet rs =
  myStmt.executeQuery(”SELECT * FROM Courses”);
while (rs.next()) {
  String code = rs.getString(1);
  String name = rs.getString(2);
  System.out.println(name + ” (” + code + ”)”);
  ResultSet rs2 = myStmt.executeQuery(
    ”SELECT teacher FROM GivenCourses ” +
    ”WHERE course = ’” + code + ”’”);
  while (rs2.next())
    System.out.println(” ” + rs2.getString(1));
}
```

Due to overuse of the same **`Statement`**, only the first course will be printed, with teachers. After the second query is executed, **`rs.next()`** will return false.

Two approaches

- If we need information from more than one table, there are two different programming patterns for doing so:
  - Joining tables in SQL
    - Join all the tables that we want the information from in a single query (like we would in SQL), get one large result set back, and use a ResultSet to iterate through this data.
  - Use nested queries in Java
    - Do a simple query on a single table, iterate through the result, and for each resulting row issue a new query to the database (like in the example on the previous page, but without the error).

Example: Joining in SQL

```
Statement myStmt = myCon.createStatement();
ResultSet rs =
  myStmt.executeQuery(
    "SELECT code, name, period, teacher " +
    "FROM Courses, GivenCourses " +
    "WHERE code = course " +
    "ORDER BY code, period");

String currentCourse, course;
while (rs.next()) {
  course = rs.getString(1);
  if (!course.equals(currentCourse))
    System.out.println(rs.getString(2));
  System.out.println("  Period " + rs.getInt(3) +
                     ": " + rs.getString(4));
  currentCourse = course;
}
```

Compare with previous row to see if this is a new course. If it is, print its name.

Example: Using nested queries in Java

```
Statement cStmt = myCon.createStatement();
Statement gcStmt = myCon.createStatement();
ResultSet courses = cStmt.executeQuery(
    "SELECT code, name " +
    "FROM Courses " +
    "ORDER BY code");

while (courses.next()) {
  String course = courses.getString(1);
  System.out.println(courses.getString(2));
  ResultSet gcourses = gcStmt.executeQuery(
    "SELECT period, teacher " +
    "FROM GivenCourses
    "WHERE course = '" + course + "' " +
    "ORDER BY period");
  while (gcourses.next()) {
    System.out.println("  Period " + gcourses.getInt(1) +
                       ": " + gcourses.getString(2));
  }
}
```

Find the given courses for each course separately with an inner query.

Comparison

- Joining in SQL
  - Requires only a single query.
  - Everything done in the DBMS, which is good at optimising.
- Nested queries
  - Many queries to send to the DBMS
    - communications/network overhead
    - compile and optimise many similar queries
  - Logic done in Java, which means optimisations must be done by hand.
  - Limits what can be done by the DBMS optimiser.

`PreparedStatement`

- Some operations on the database are run multiple times, with the same or only slightly different data.
  - Example: asking for information from the same table, perhaps with different tests, or with a different ordering.
- We can create a specialized **`PreparedStatement`** with a particular associated query or modification.

```
PreparedStatement myPstmt =
  myCon.prepareStatement("SELECT * FROM Courses");
```

Parametrized prepared statements

- We can parametrize data in a statement.
  - Data that could differ is replaced with **`?`** in the statement text.
  - **`?`** parameters can be instantiated using functions **`setX(int index, X value)`**.

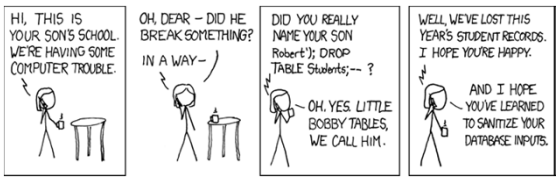
```
PreparedStatement myPstmt =
 myCon.prepareStatement(
   "INSERT INTO Courses VALUES (?,?)");

myPstmt.setString(1, "TDA356");
myPstmt.setString(2, "Databases");
```

Summary JDBC

- **`DriverManager`**
  - Register drivers, create connections.
- **`Connection`**
  - Create statements or prepared statements.
  - Close when finished.
- **`Statement`**
  - Execute queries or modifications.
- **`PreparedStatement`**
  - Execute a particular query or modification, possibly parametrized.
- **`ResultSet`**
  - Iterate through the result set of a query.

SQL Injection



http://xkcd.com/327/

http://www.val.se/val/val2010/handskrivna/handskrivna.skv

K;13;Hallands län;80;Halmstad;01;Halmstads västra valkrets;904;Söndrum 4;pwn DROP TABLE VALJ;1

Quiz!

Write a query in SQL that returns the value 1.

Assuming we have a table T, we could write it as

```
SELECT 1 FROM
(SELECT COUNT(*) FROM T);
```

We must use an aggregation, otherwise we cannot ensure that we get only one value as the result.

Oracle recognizes this problem and supplies the table Dual. This table has one column dummy, and one row with the value 'X'. Since it is guaranteed to have only one row in it, we can write the above as

```
SELECT 1 FROM Dual;
```

Lab Part IV - Interfacing

- Write a Java application that uses JDBC to connect to and use the database that you created in part II.
- Your application should make use of the views and triggers that you created in parts II and III.
- Start from a stub application.

Lab Part IV - Interfacing

- Hand in (yes, hand in!):
  - Your Java source code
- The fourth part of the lab will be accepted or rejected at the lab supervision sessions!
  - You will show us your running application, we will stress test it, and ask to see some parts of the source code.
  - You should still hand in your source code!
- Deadline: Friday 13 December (at the supervision session)

Final deadline!

- The final, hard deadline for all parts of the lab is Friday 13 December, at the supervision session.
  - If your lab is not accepted by the deadline, you will be asked to come back to finish it in period 3.