SQL language: advanced constructs

SQL for applications

DBMG

SQL for applications

- Introduction
- Cursors
- Updatability
- Embedded SQL
- Call Level Interface (CLI)
- Stored Procedures
- Comparison of alternatives

DBMG

SQL for applications

Introduction

DBMG

Example application

- Banking operations
  - withdrawal operation from an account through an ATM
  - withdrawal operation from an account at a bank counter

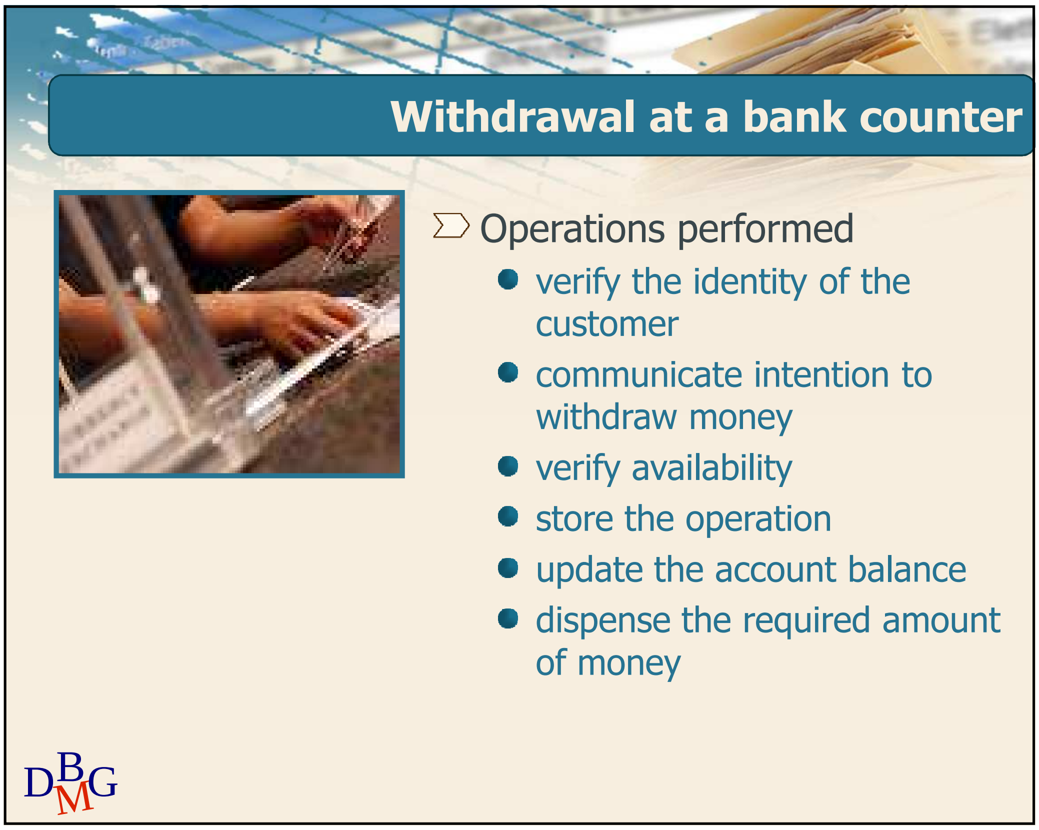
DBMG

Withdrawal from an ATM

- Operations performed
  - check the validity of ATM card and PIN code
  - select withdrawal operation
  - specify the required amount
  - verify availability
  - store the operation
  - update the account balance
  - dispense the required amount of money

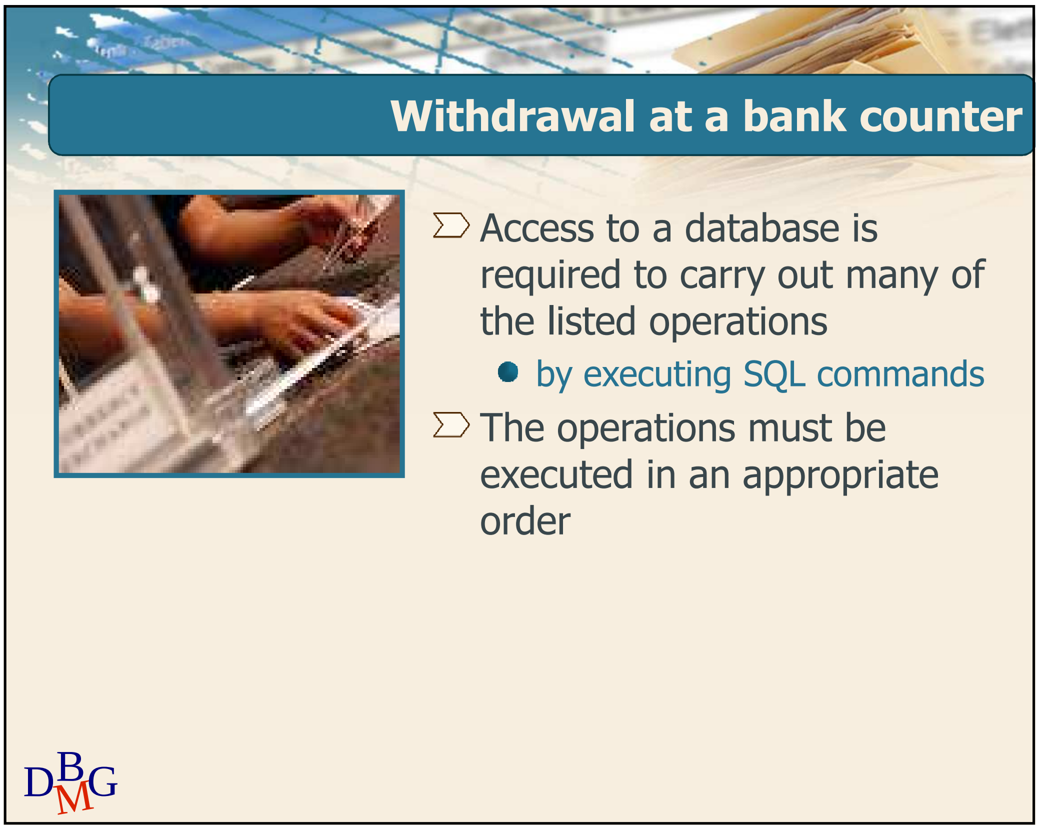
Withdrawal from an ATM

- Access to a database is required to carry out many of the listed operations
  - by executing SQL commands
- The operations must be executed in an appropriate order



Withdrawal at a bank counter

- Operations performed
  - verify the identity of the customer
  - communicate intention to withdraw money
  - verify availability
  - store the operation
  - update the account balance
  - dispense the required amount of money



Withdrawal at a bank counter

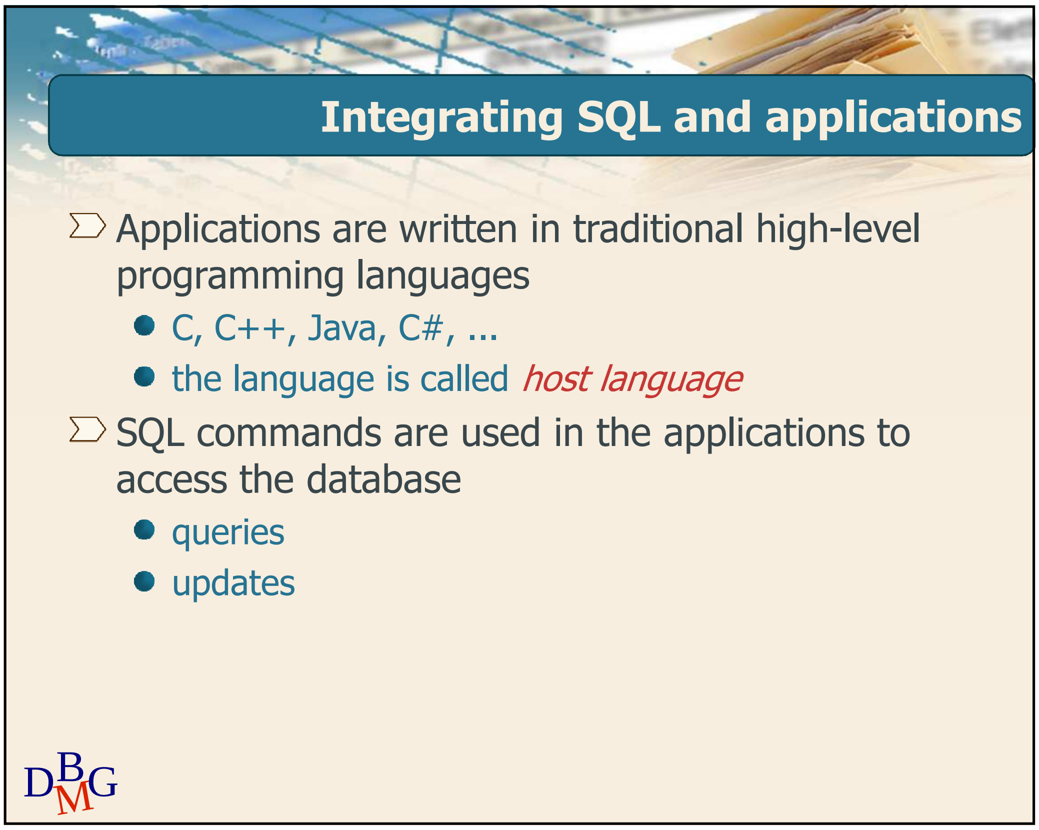
- Access to a database is required to carry out many of the listed operations
  - by executing SQL commands
- The operations must be executed in an appropriate order

Example: banking operations

- Banking operations require accessing the database and modifying its contents
  - execution of SQL commands
    - customers or the bank employees are not directly executing the SQL commands
  - an application hides the execution of the SQL commands
- Correctly managing banking operations requires executing a specific sequence of steps
  - an application allows specifying the correct order of execution for the operations

Applications and SQL

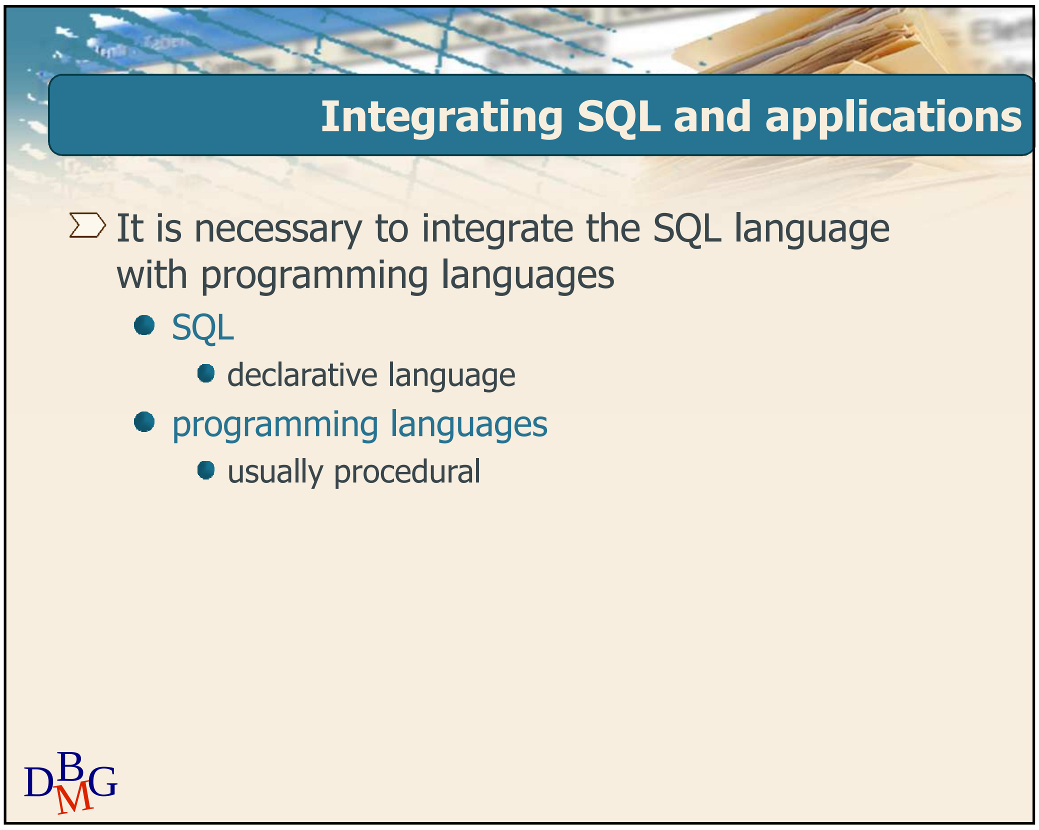
- Real problems can hardly ever be solved by executing single SQL commands
- We need applications to
  - acquire and handle input data
    - user choices, parameters
  - manage the application logic
    - flow of the operations to execute
  - return results to the user using different formats
    - non-relational data representation
      - XML document
    - complex data visualization
      - graphs, reports



Integrating SQL and applications

- Applications are written in traditional high-level programming languages
  - C, C++, Java, C#, ...
  - the language is called *host language*
- SQL commands are used in the applications to access the database
  - queries
  - updates

DBMG



Integrating SQL and applications

- It is necessary to integrate the SQL language with programming languages
  - SQL
    - declarative language
  - programming languages
    - usually procedural

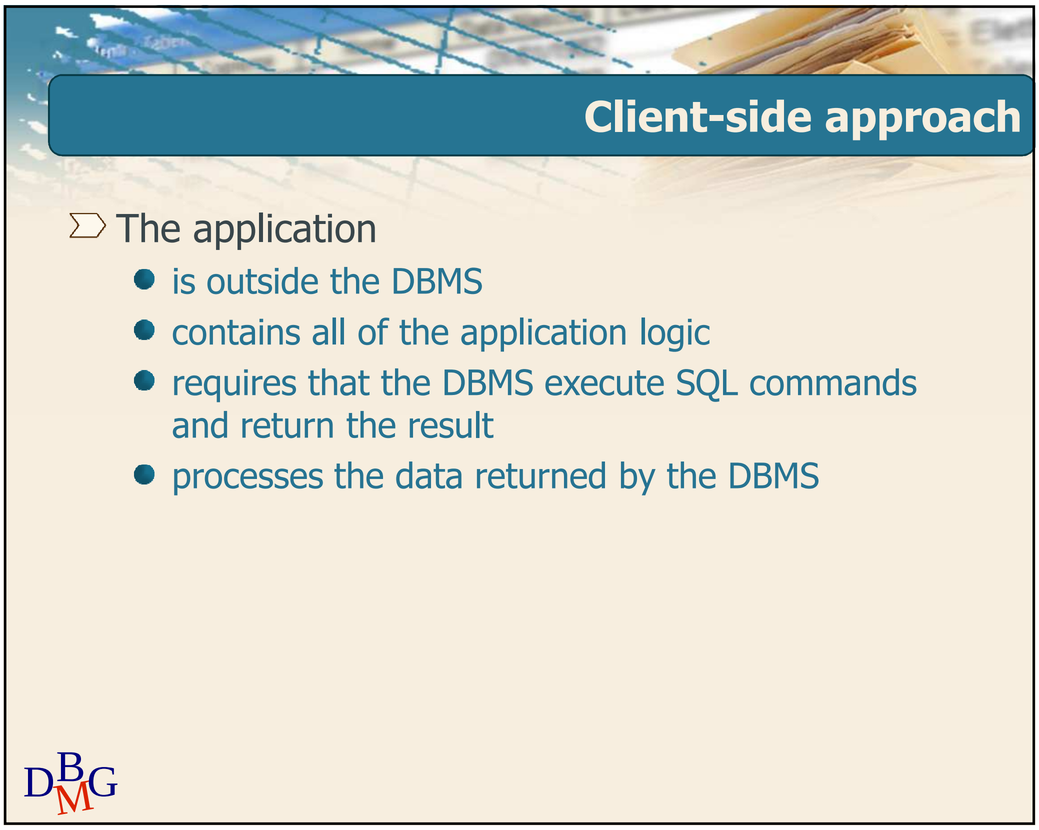
DBMG

Impedance mismatch

- Impedance mismatch
  - SQL queries operate on one or more tables and produce a table as a result
    - set-oriented approach
  - programming languages access tables by reading rows *one by one*
    - tuple-oriented approach
- Possible solutions to solve the conflict
  - use cursors
  - use languages that intrinsically provide data structures storing "sets of rows"

SQL and programming languages

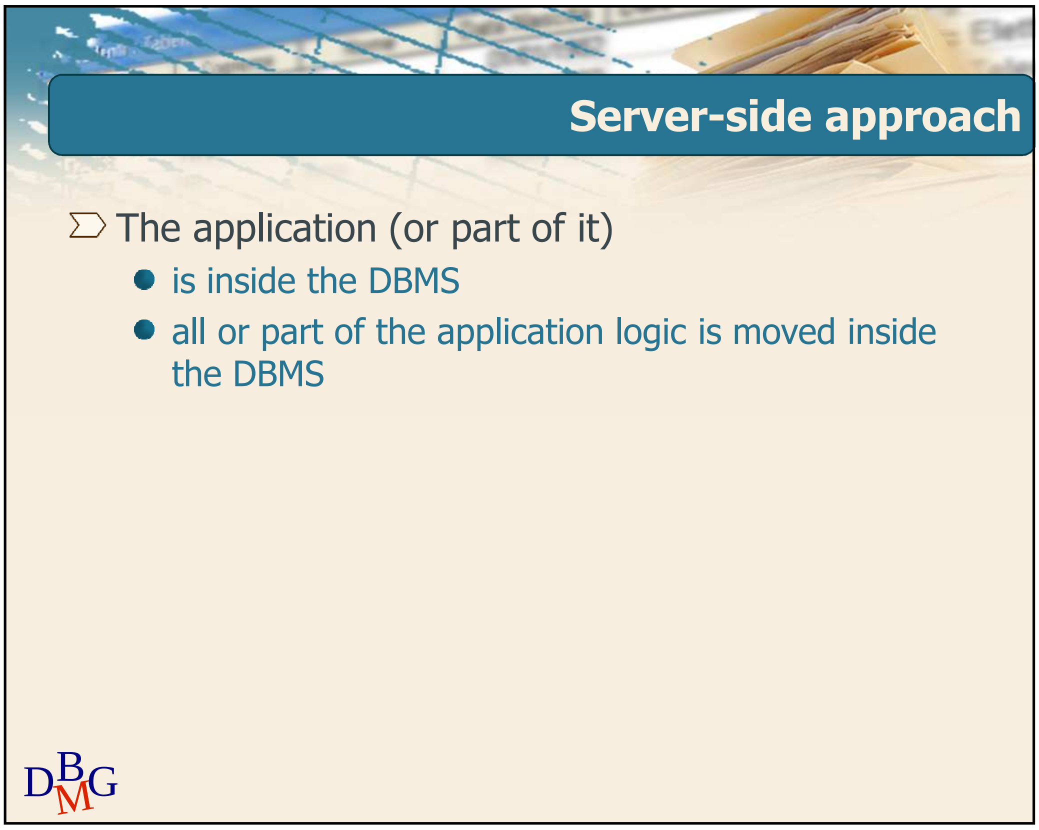
- Main integration techniques
  - Embedded SQL
  - Call Level Interface (CLI)
    - SQL/CLI, ODBC, JDBC, OLE DB, ADO.NET, ..
  - Stored procedures
- Classified as
  - client-side
    - embedded SQL, call level interface
  - server-side
    - stored procedures



Client-side approach

- The application
  - is outside the DBMS
  - contains all of the application logic
  - requires that the DBMS execute SQL commands and return the result
  - processes the data returned by the DBMS

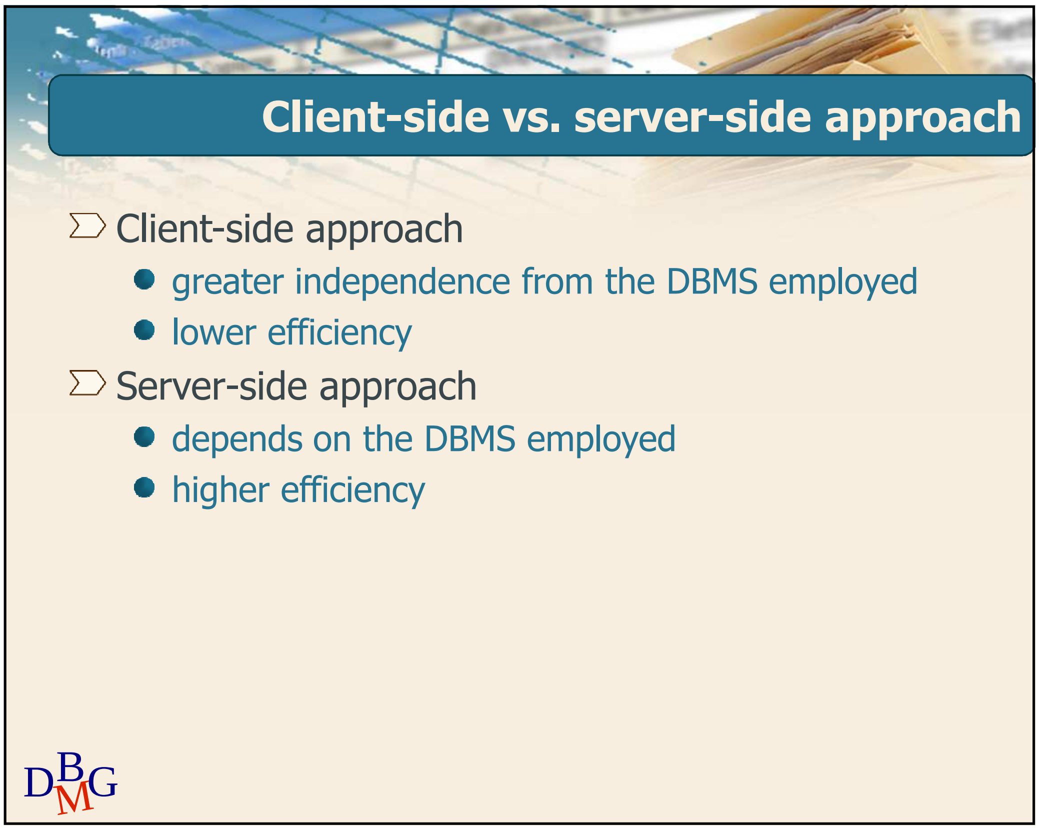
DBMG



Server-side approach

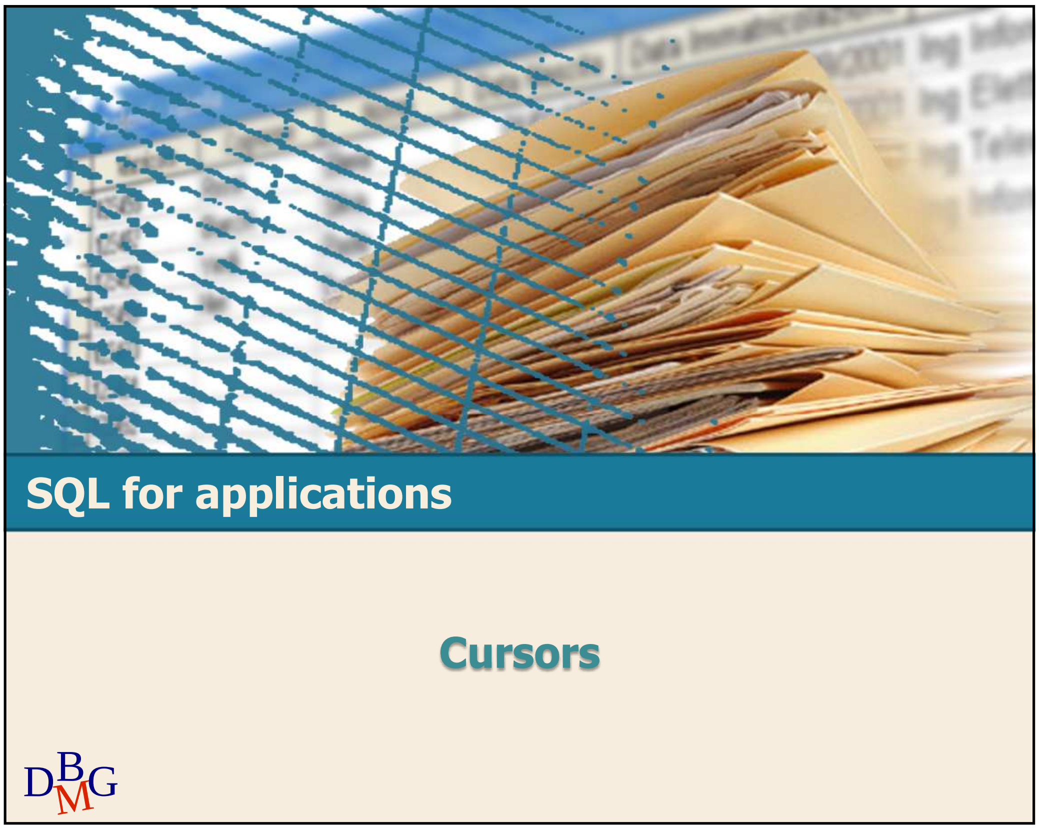
- The application (or part of it)
  - is inside the DBMS
  - all or part of the application logic is moved inside the DBMS

DBMG



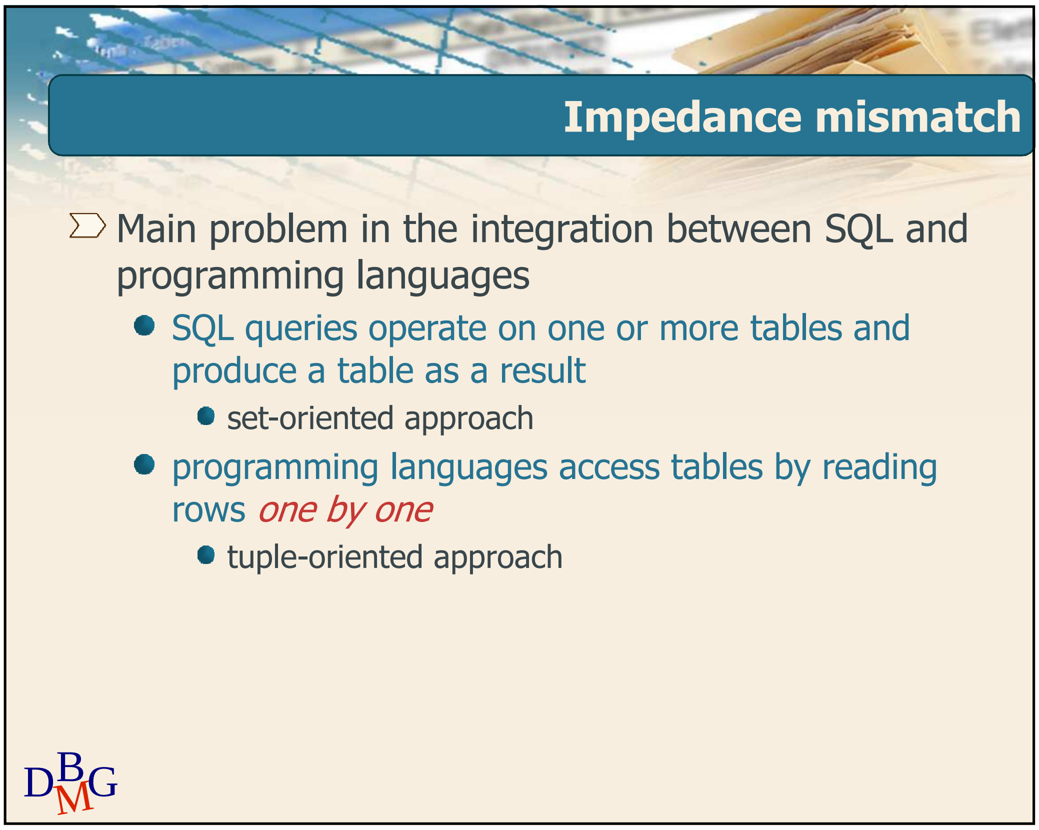
Client-side vs. server-side approach

- Client-side approach
  - greater independence from the DBMS employed
  - lower efficiency
- Server-side approach
  - depends on the DBMS employed
  - higher efficiency



SQL for applications

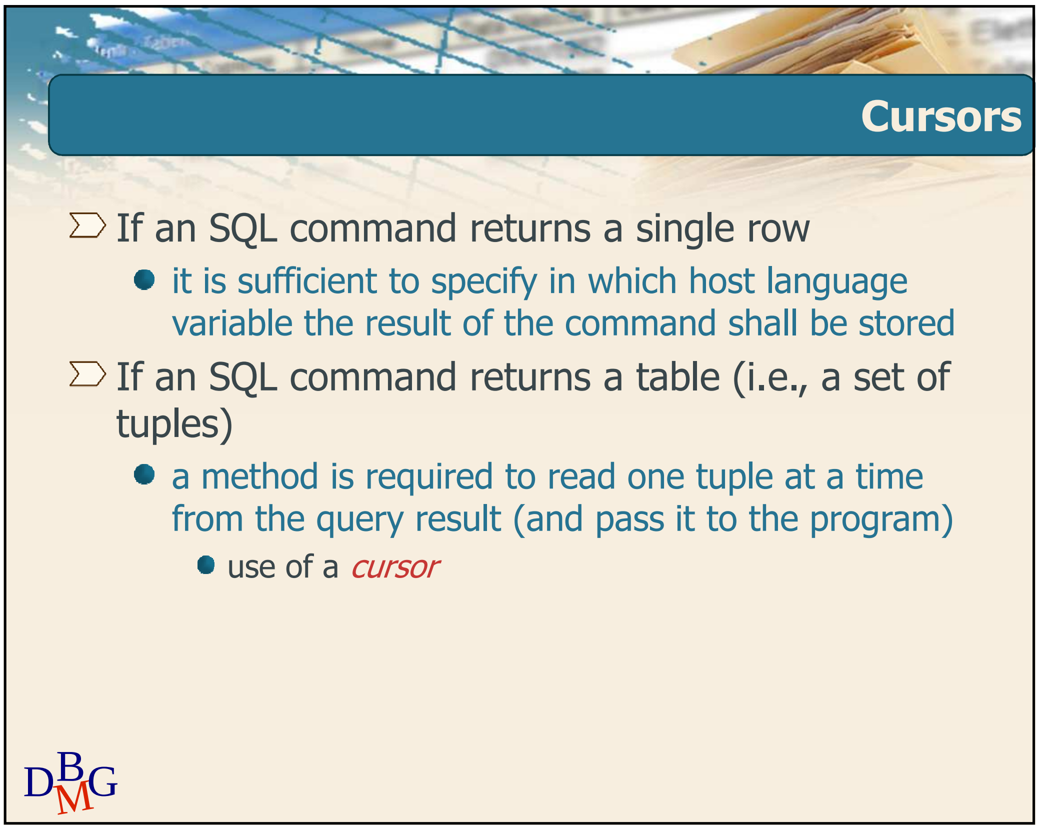
Cursors



Impedance mismatch

- Main problem in the integration between SQL and programming languages
  - SQL queries operate on one or more tables and produce a table as a result
    - set-oriented approach
  - programming languages access tables by reading rows *one by one*
    - tuple-oriented approach

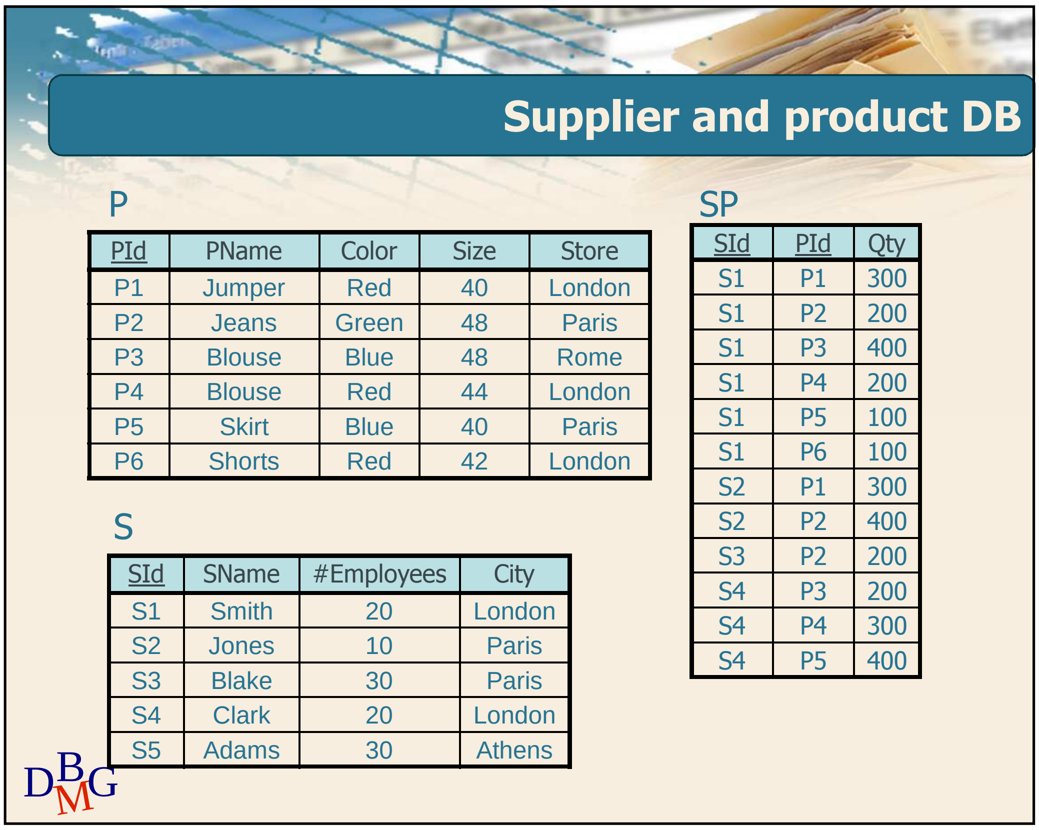
DBMG



Cursors

- If an SQL command returns a single row
  - it is sufficient to specify in which host language variable the result of the command shall be stored
- If an SQL command returns a table (i.e., a set of tuples)
  - a method is required to read one tuple at a time from the query result (and pass it to the program)
    - use of a *cursor*

DBMG



Supplier and product DB

P

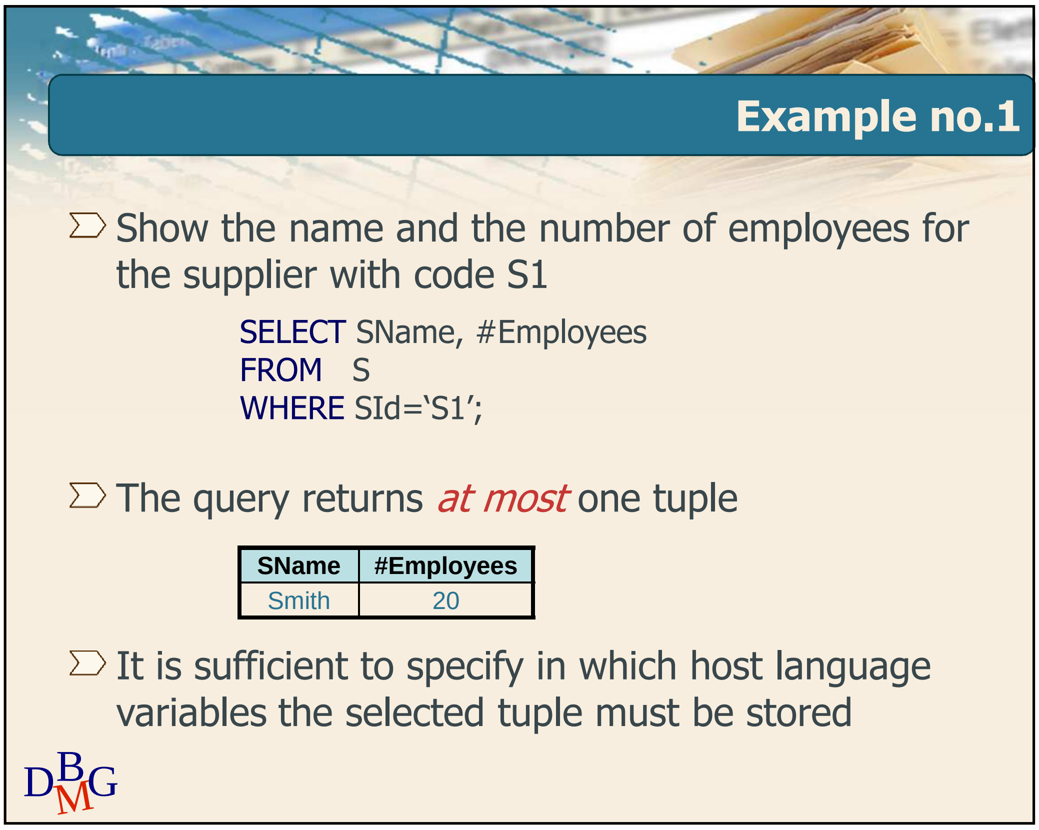
<u>PId</u>	PName	Color	Size	Store
P1	Jumper	Red	40	London
P2	Jeans	Green	48	Paris
P3	Blouse	Blue	48	Rome
P4	Blouse	Red	44	London
P5	Skirt	Blue	40	Paris
P6	Shorts	Red	42	London

S

<u>SId</u>	SName	#Employees	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

<u>SId</u>	<u>PId</u>	Qty
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400



Example no.1

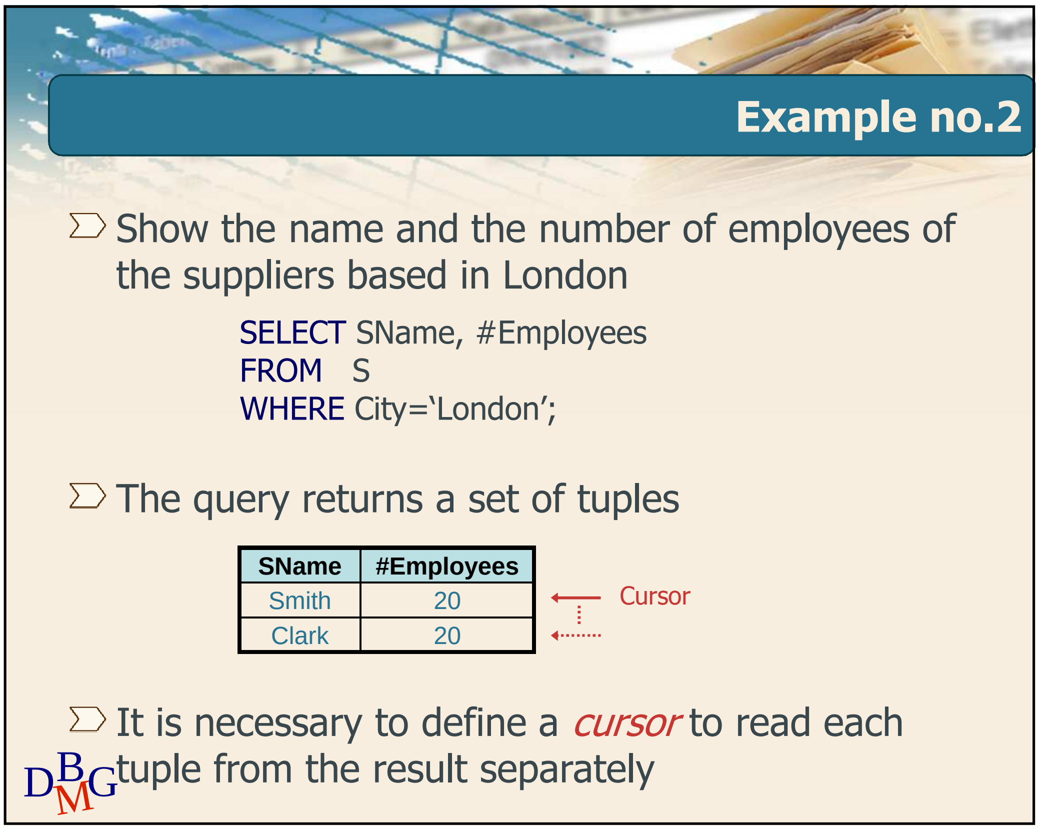
- Show the name and the number of employees for the supplier with code S1

```
SELECT SName, #Employees
FROM   S
WHERE SId='S1';
```

- The query returns *at most* one tuple

SName	**#Employees**
Smith	20

- It is sufficient to specify in which host language variables the selected tuple must be stored



Example no.2

- Show the name and the number of employees of the suppliers based in London

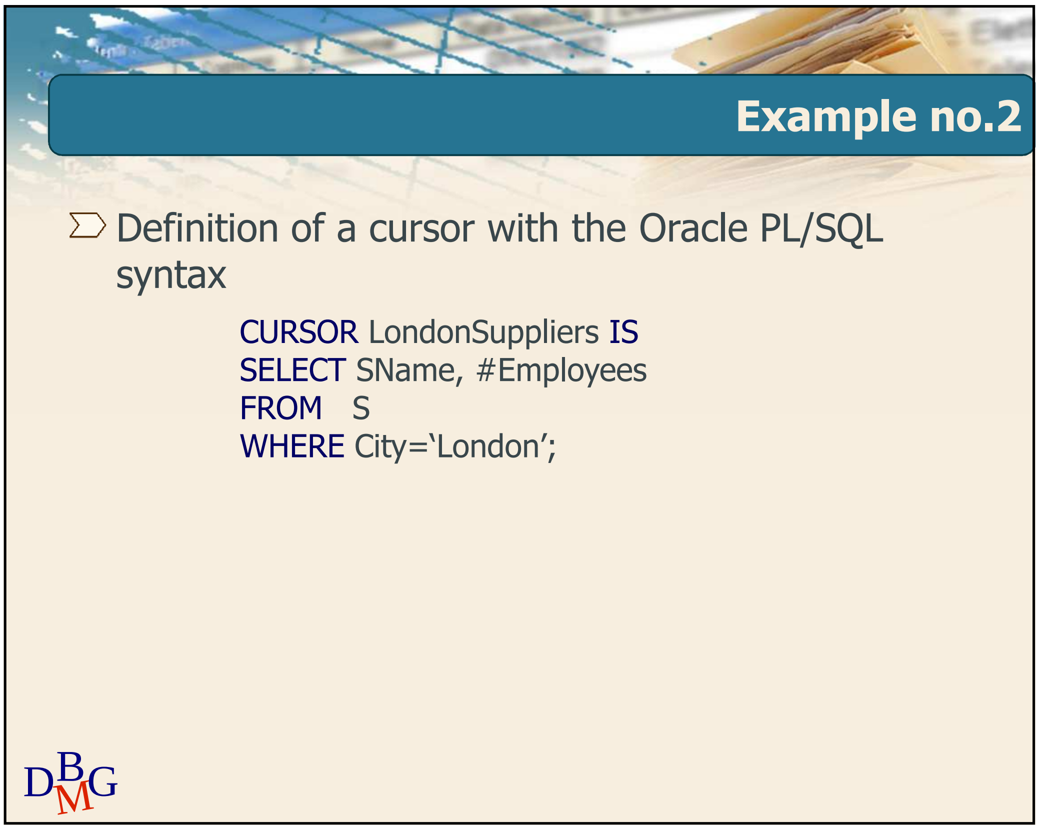
```
SELECT SName, #Employees
FROM   S
WHERE City=‘London’;
```

- The query returns a set of tuples

SName	#Employees
Smith	20
Clark	20

Cursor

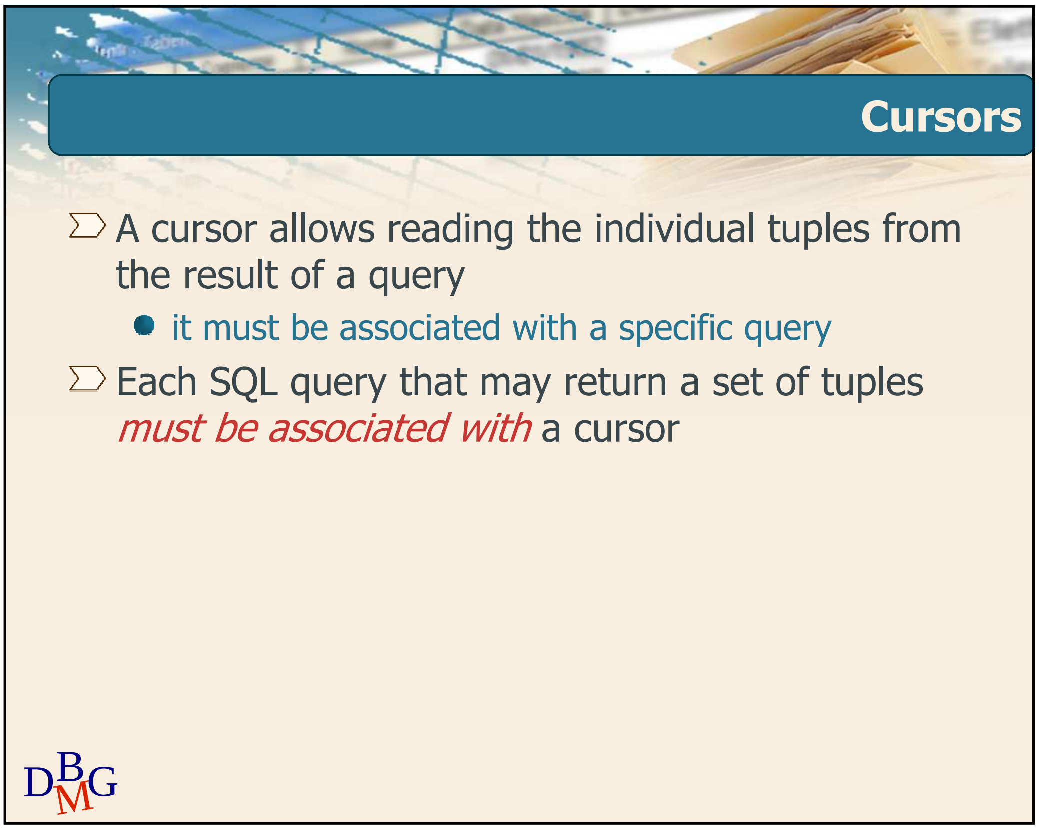
- It is necessary to define a *cursor* to read each tuple from the result separately



Example no.2

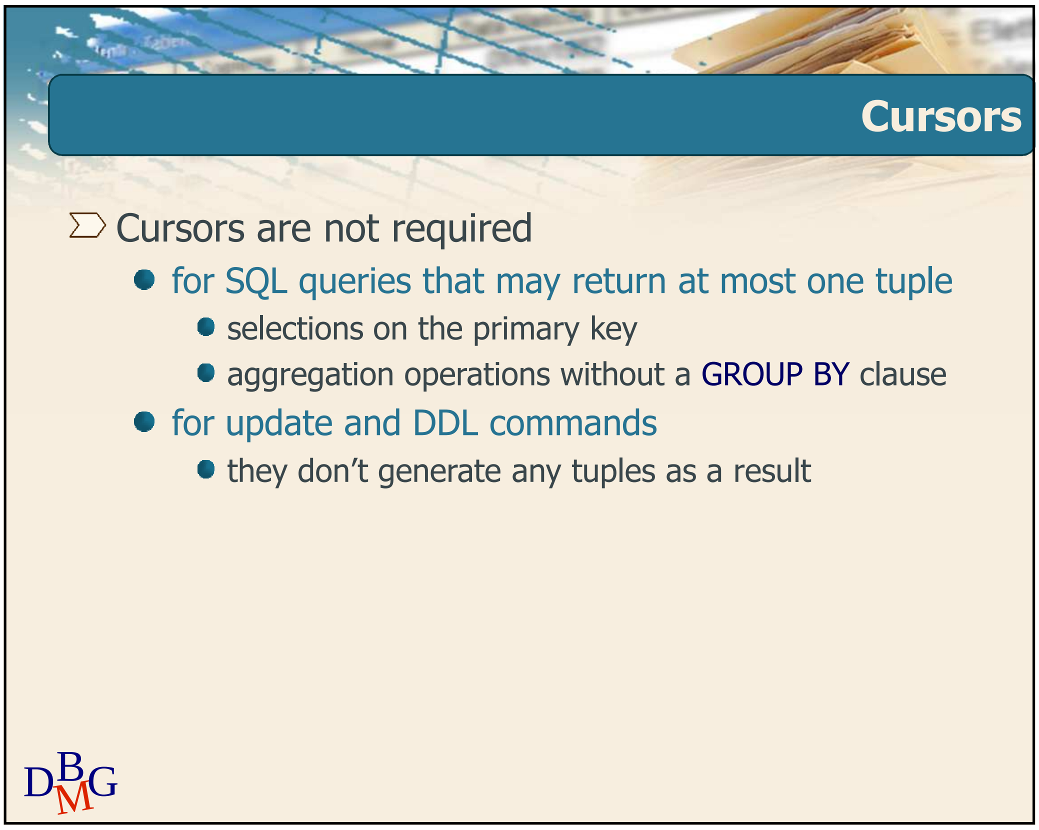
- Definition of a cursor with the Oracle PL/SQL syntax

```
CURSOR LondonSuppliers IS
SELECT SName, #Employees
FROM   S
WHERE City=‘London’;
```



Cursors

- A cursor allows reading the individual tuples from the result of a query
  - it must be associated with a specific query
- Each SQL query that may return a set of tuples *must be associated with* a cursor



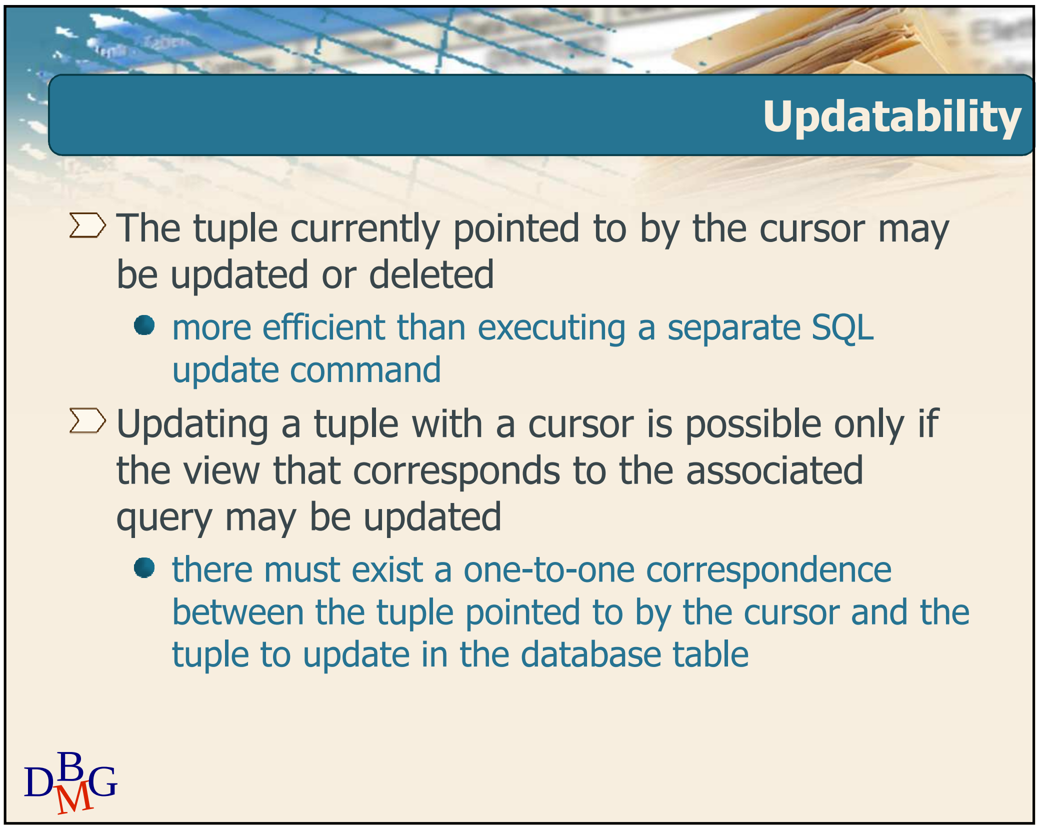
Cursors

- Cursors are not required
  - for SQL queries that may return at most one tuple
    - selections on the primary key
    - aggregation operations without a GROUP BY clause
  - for update and DDL commands
    - they don't generate any tuples as a result

SQL for applications

Updatability

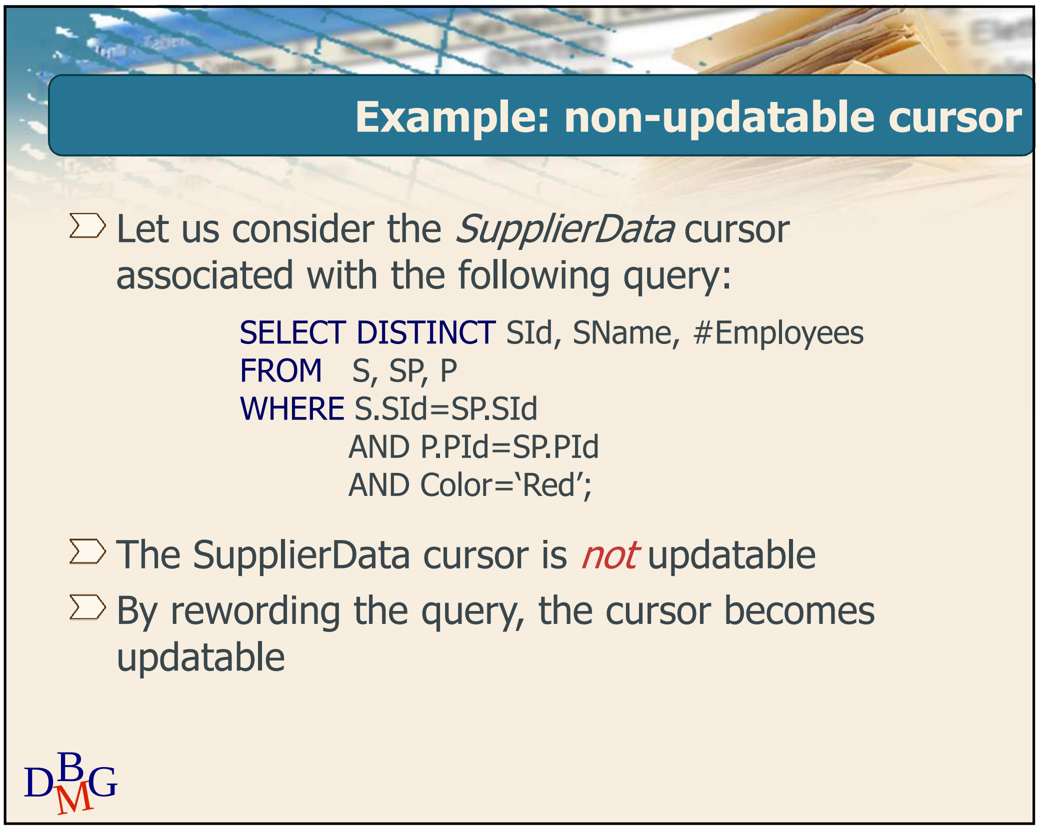
DBMG



Updatability

- The tuple currently pointed to by the cursor may be updated or deleted
  - more efficient than executing a separate SQL update command
- Updating a tuple with a cursor is possible only if the view that corresponds to the associated query may be updated
  - there must exist a one-to-one correspondence between the tuple pointed to by the cursor and the tuple to update in the database table

DBMG

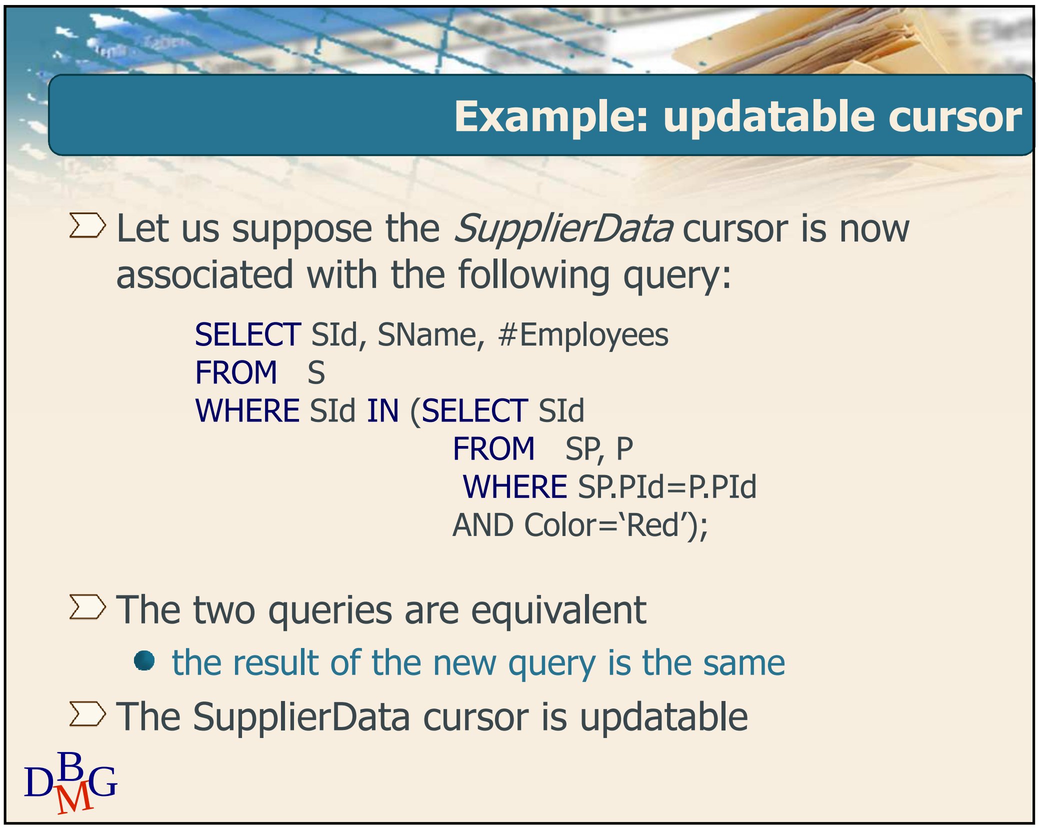


Example: non-updatable cursor

- Let us consider the *SupplierData* cursor associated with the following query:

```
SELECT DISTINCT SId, SName, #Employees
FROM   S, SP, P
WHERE S.SId=SP.SId
         AND P.PId=SP.PId
         AND Color='Red';
```

- The SupplierData cursor is *not* updatable
- By rewording the query, the cursor becomes updatable



Example: updatable cursor

- Let us suppose the *SupplierData* cursor is now associated with the following query:

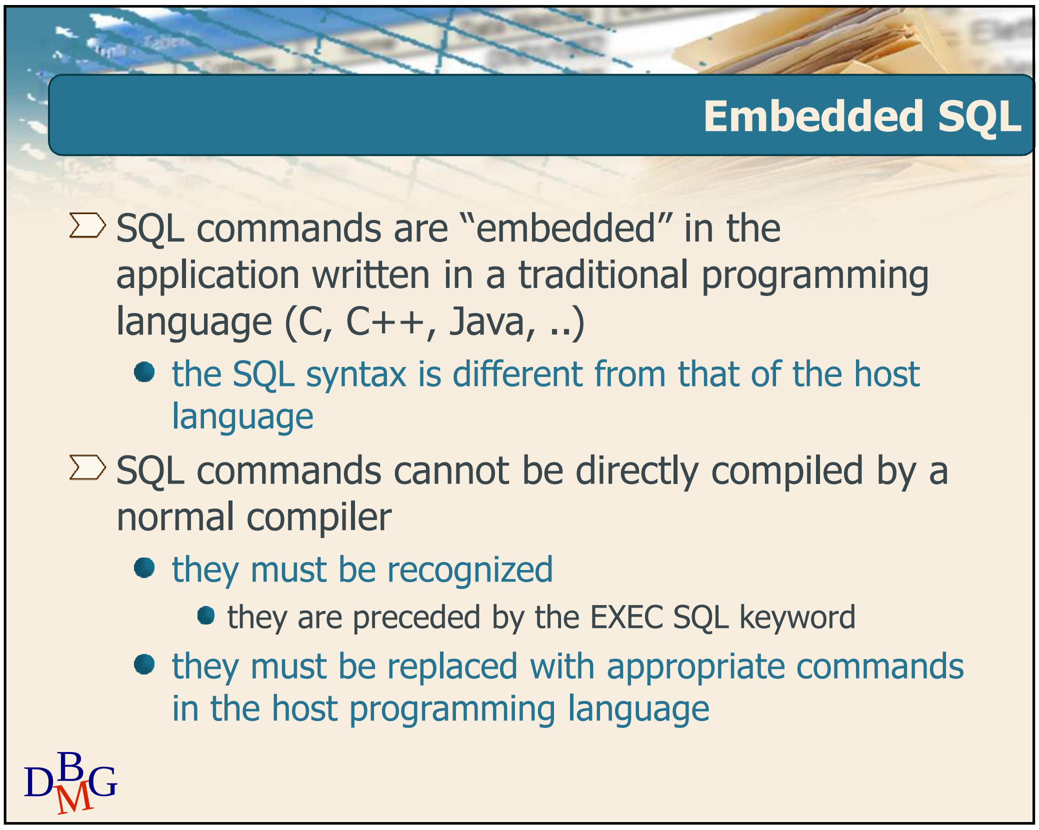
```
SELECT SId, SName, #Employees
FROM   S
WHERE SId IN (SELECT SId
              FROM   SP, P
               WHERE SP.PId=P.PId
              AND Color='Red');
```

- The two queries are equivalent
  - the result of the new query is the same
- The SupplierData cursor is updatable

SQL for applications

Embedded SQL

DBMG



Embedded SQL

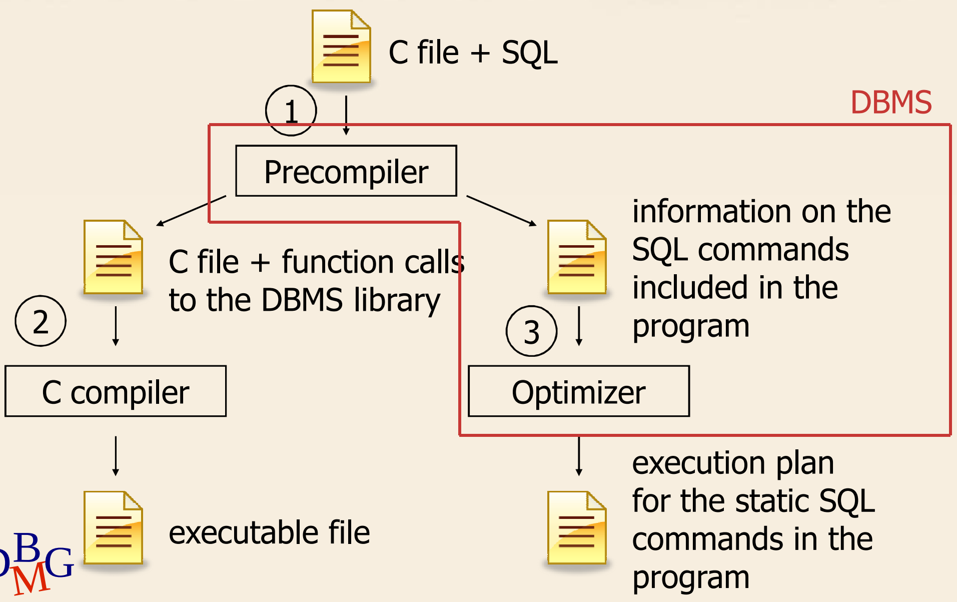
- SQL commands are "embedded" in the application written in a traditional programming language (C, C++, Java, ..)
  - the SQL syntax is different from that of the host language
- SQL commands cannot be directly compiled by a normal compiler
  - they must be recognized
    - they are preceded by the EXEC SQL keyword
  - they must be replaced with appropriate commands in the host programming language

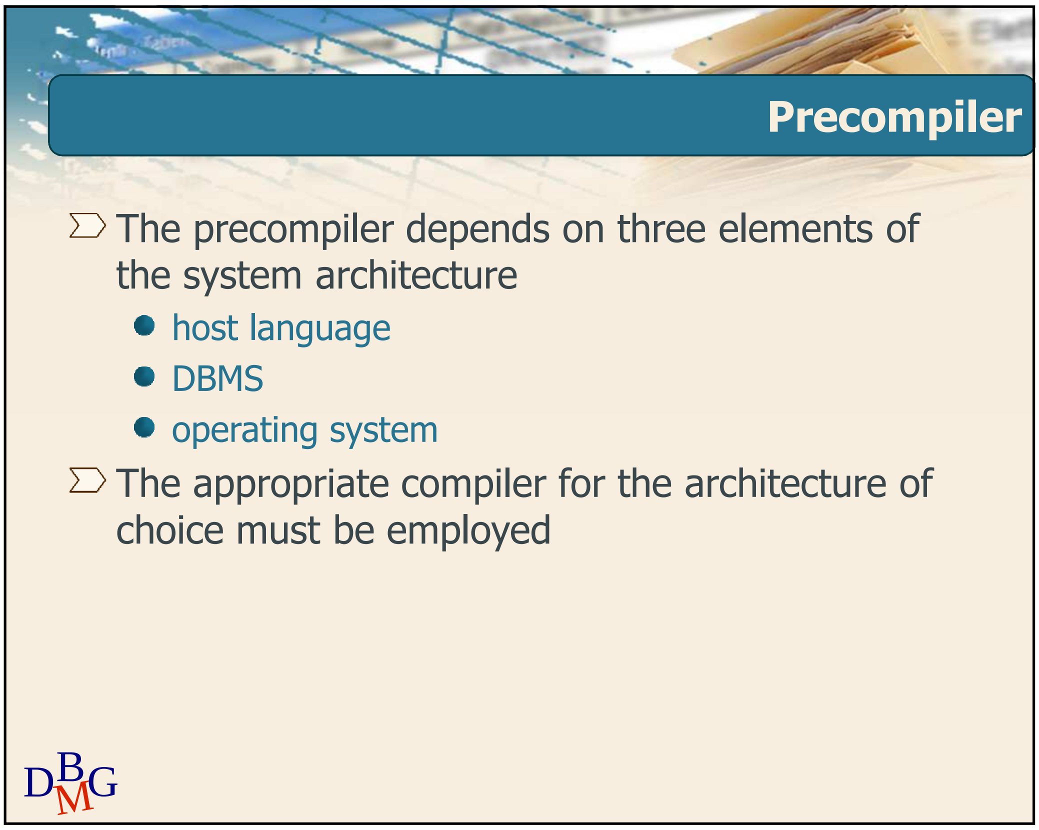
DBMG

Precompilation

- The precompiler
  - identifies SQL commands embedded in the code
    - parts preceded by EXEC SQL
  - replaces the SQL commands with function calls to specific APIs of the chosen DBMS
    - such functions are written in the host programming language
  - it optionally sends the static SQL commands to the DBMS for compilation and optimization
- The precompiler is tied to a specific DBMS

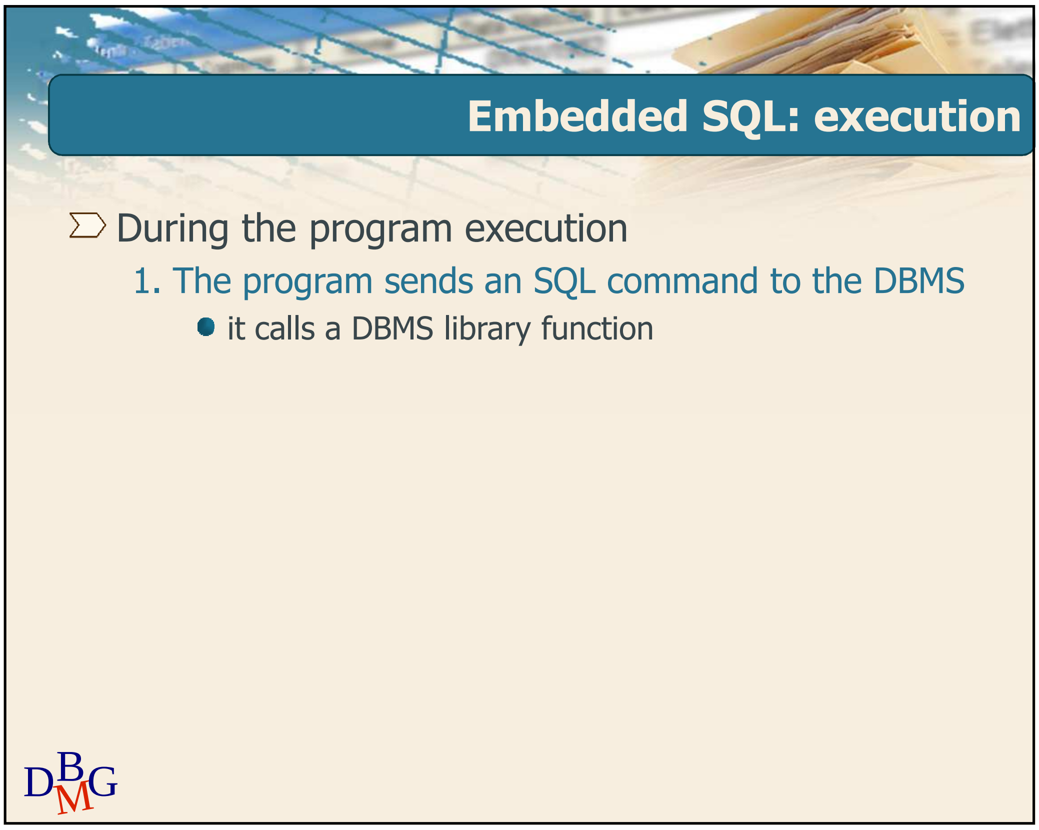
Embedded SQL: compilation


C file + SQL
1
DBMS
Precompiler
C file + function calls to the DBMS library
information on the SQL commands included in the program
2
3
C compiler
Optimizer
execution plan for the static SQL commands in the program
executable file



Precompiler

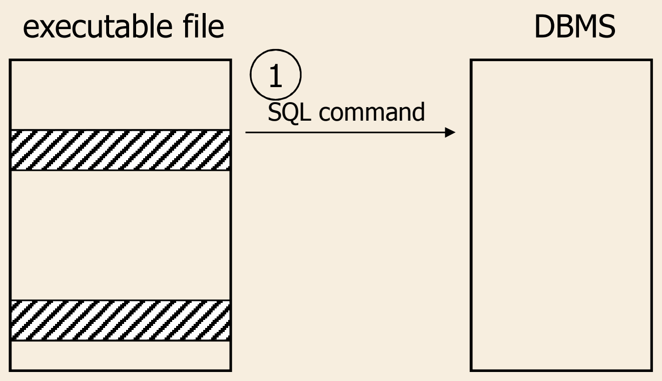
- The precompiler depends on three elements of the system architecture
  - host language
  - DBMS
  - operating system
- The appropriate compiler for the architecture of choice must be employed



Embedded SQL: execution

- During the program execution
  1. The program sends an SQL command to the DBMS
     - it calls a DBMS library function

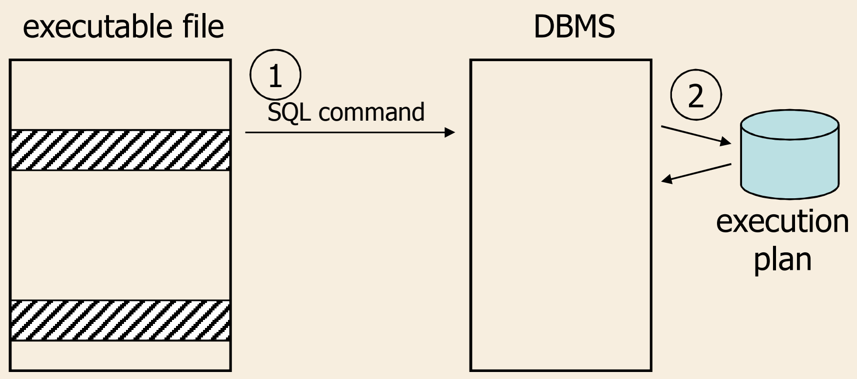
Embedded SQL: execution



Embedded SQL: execution

- During the program execution
  1. The program sends an SQL command to the DBMS
     - it calls a DBMS library function
  2. The DBMS generates the execution plan for the command
     - if one has already been defined, it will be retrieved

Embedded SQL: execution



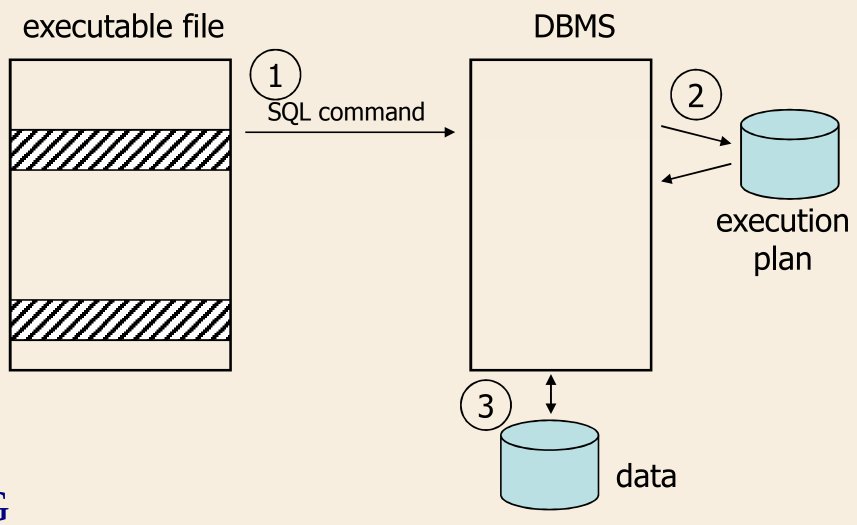
DBG
M

Embedded SQL: execution

- During the program execution
  1. The program sends an SQL command to the DBMS
     - it calls a DBMS library function
  2. The DBMS generates the execution plan for the command
     - if one has already been defined, it will be retrieved
  3. The DBMS executes the SQL command

DBG
M

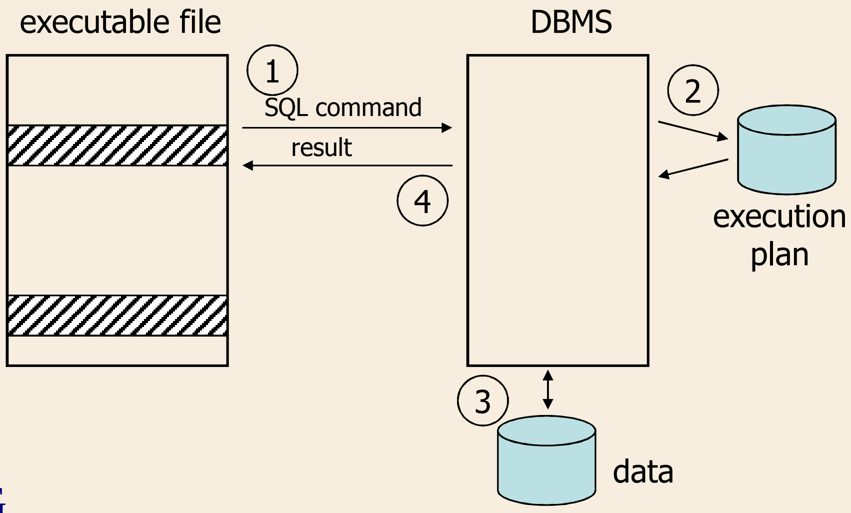
Embedded SQL: execution



Embedded SQL: execution

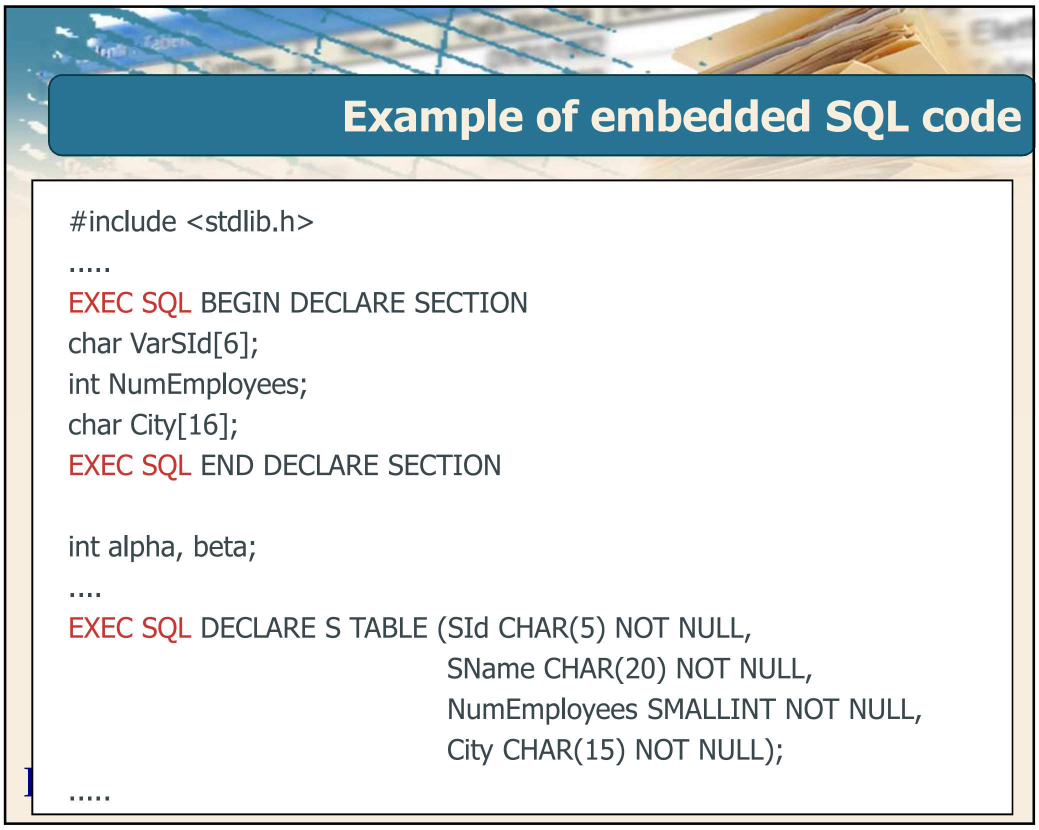
- During the program execution
  1. The program sends an SQL command to the DBMS
     - it calls a DBMS library function
  2. The DBMS generates the execution plan for the command
     - if one has already been defined, it will be retrieved
  3. The DBMS executes the SQL command
  4. The DBMS returns the result of the SQL command
     - a transit area is used as temporary data storage

Embedded SQL: execution



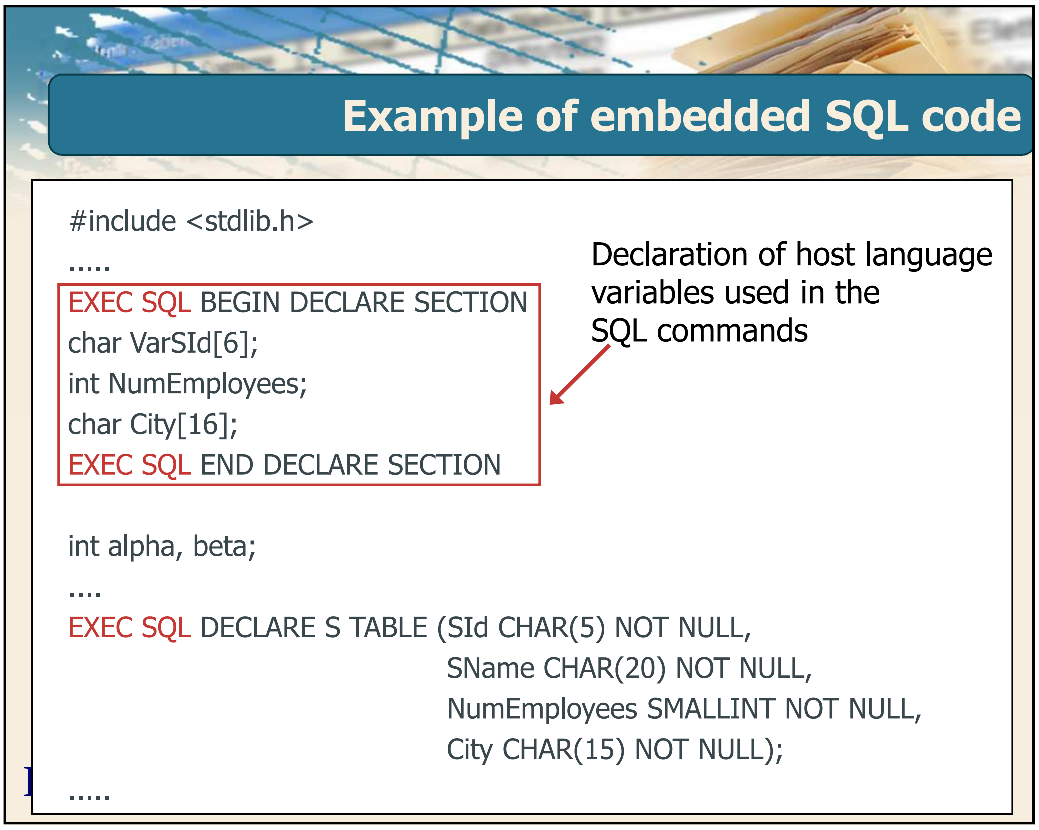
Embedded SQL: execution

- During the program execution
  1. The program sends an SQL command to the DBMS
     - it calls a DBMS library function
  2. The DBMS generates the execution plan for the command
     - if one has already been defined, it will be retrieved
  3. The DBMS executes the SQL command
  4. The DBMS returns the result of the SQL command
     - a transit area is used as temporary data storage
  5. The program processes the result



Example of embedded SQL code

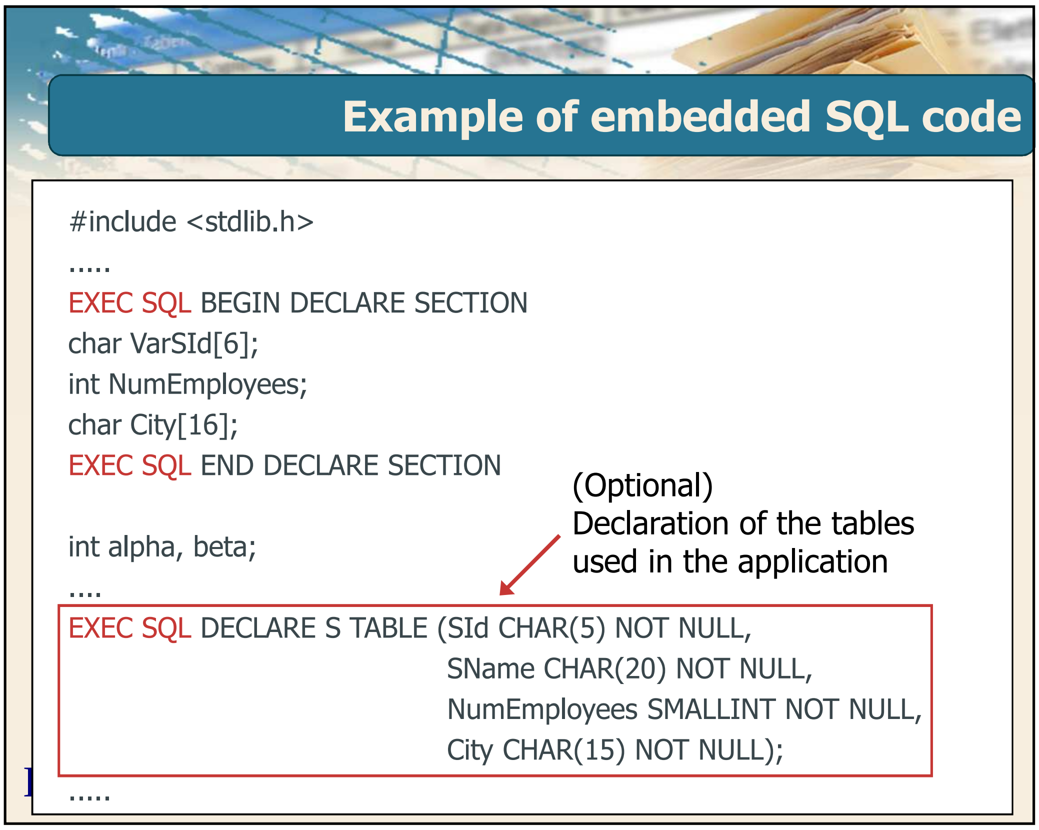
```
#include <stdlib.h>
.....
EXEC SQL BEGIN DECLARE SECTION
char VarSId[6];
int NumEmployees;
char City[16];
EXEC SQL END DECLARE SECTION

int alpha, beta;
....
EXEC SQL DECLARE S TABLE (SId CHAR(5) NOT NULL,
                          SName CHAR(20) NOT NULL,
                          NumEmployees SMALLINT NOT NULL,
                          City CHAR(15) NOT NULL);
.....
```



Example of embedded SQL code

```
#include <stdlib.h>
.....
EXEC SQL BEGIN DECLARE SECTION
char VarSId[6];
int NumEmployees;
char City[16];
EXEC SQL END DECLARE SECTION

int alpha, beta;
....
EXEC SQL DECLARE S TABLE (SId CHAR(5) NOT NULL,
                          SName CHAR(20) NOT NULL,
                          NumEmployees SMALLINT NOT NULL,
                          City CHAR(15) NOT NULL);
.....
```

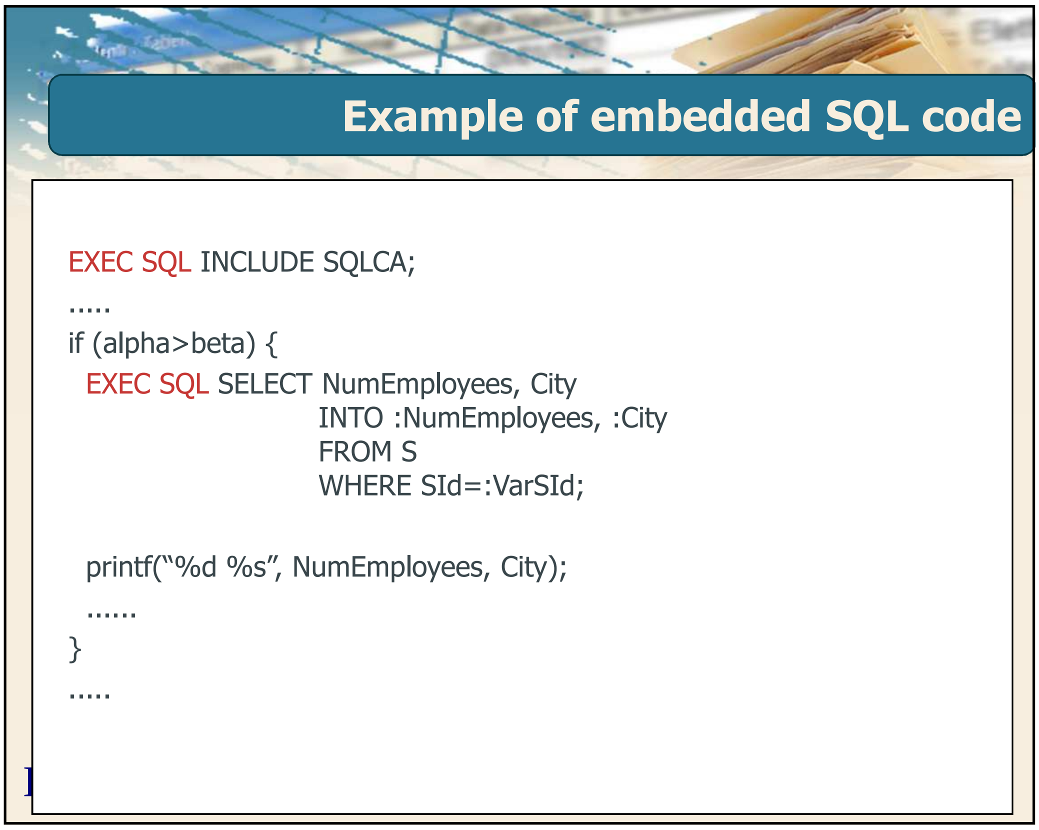
Declaration of host language variables used in the SQL commands



Example of embedded SQL code

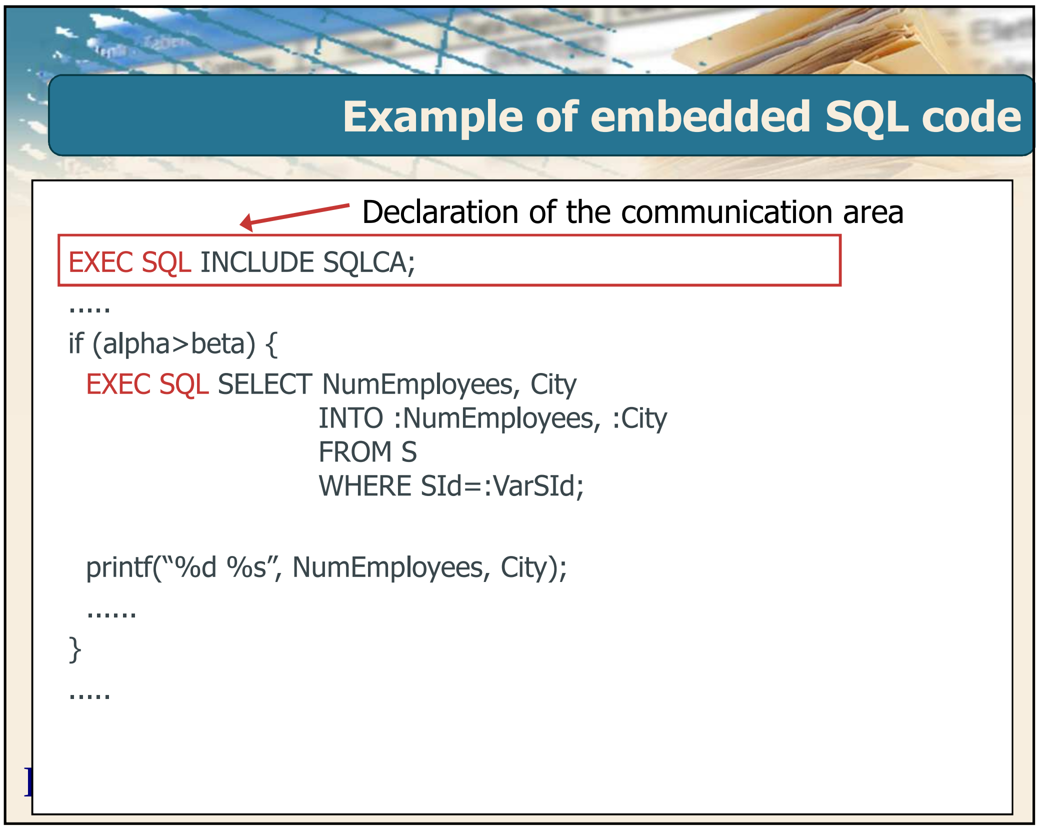
```
#include <stdlib.h>
.....
EXEC SQL BEGIN DECLARE SECTION
char VarSId[6];
int NumEmployees;
char City[16];
EXEC SQL END DECLARE SECTION

int alpha, beta;
....
EXEC SQL DECLARE S TABLE (SId CHAR(5) NOT NULL,
                          SName CHAR(20) NOT NULL,
                          NumEmployees SMALLINT NOT NULL,
                          City CHAR(15) NOT NULL);
.....
```

(Optional)
Declaration of the tables used in the application



Example of embedded SQL code

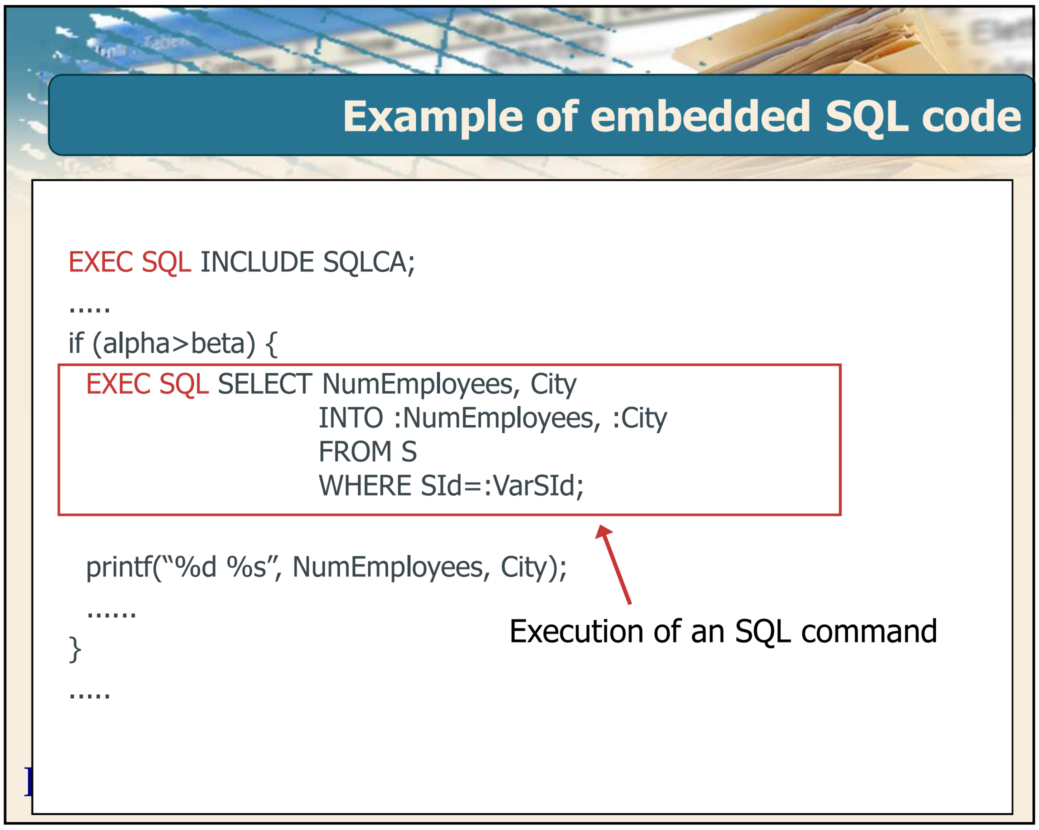
```
EXEC SQL INCLUDE SQLCA;
.....
if (alpha>beta) {
  EXEC SQL SELECT NumEmployees, City
                INTO :NumEmployees, :City
                FROM S
                WHERE SId=:VarSId;

  printf("%d %s", NumEmployees, City);
  ......
}
.....
```



Example of embedded SQL code

Declaration of the communication area

```
EXEC SQL INCLUDE SQLCA;
.....
if (alpha>beta) {
  EXEC SQL SELECT NumEmployees, City
                    INTO :NumEmployees, :City
                    FROM S
                    WHERE SId=:VarSId;

  printf("%d %s", NumEmployees, City);
  ......
}
.....
```



Example of embedded SQL code

```
EXEC SQL INCLUDE SQLCA;
.....
if (alpha>beta) {
  EXEC SQL SELECT NumEmployees, City
                    INTO :NumEmployees, :City
                    FROM S
                    WHERE SId=:VarSId;

  printf("%d %s", NumEmployees, City);
  ......
}
.....
```

Execution of an SQL command

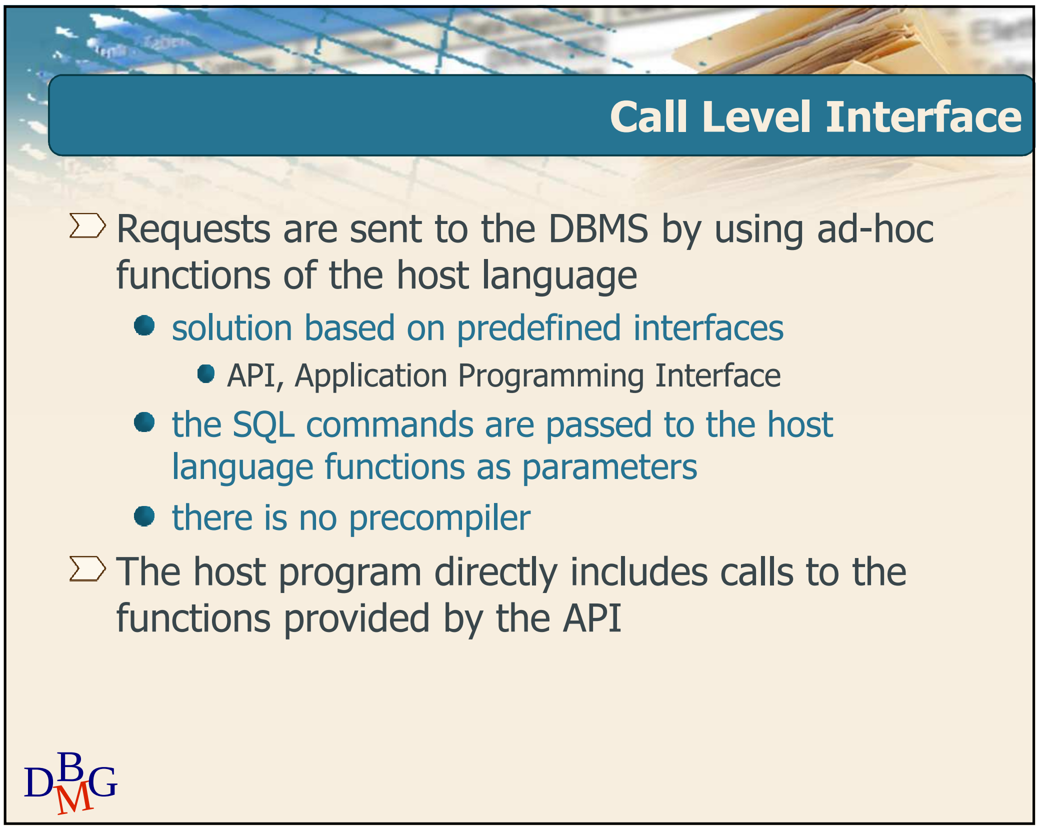
Example of embedded SQL code

Host language variables

```
EXEC SQL INCLUDE SQLCA;
.....
if (alpha>beta) {
  EXEC SQL SELECT NumEmployees, City
                         INTO :NumEmployees, :City
                         FROM S
                         WHERE SId=:VarSId;

  printf("%d %s", NumEmployees, City);
  ......
}
.....
```

SQL for applications

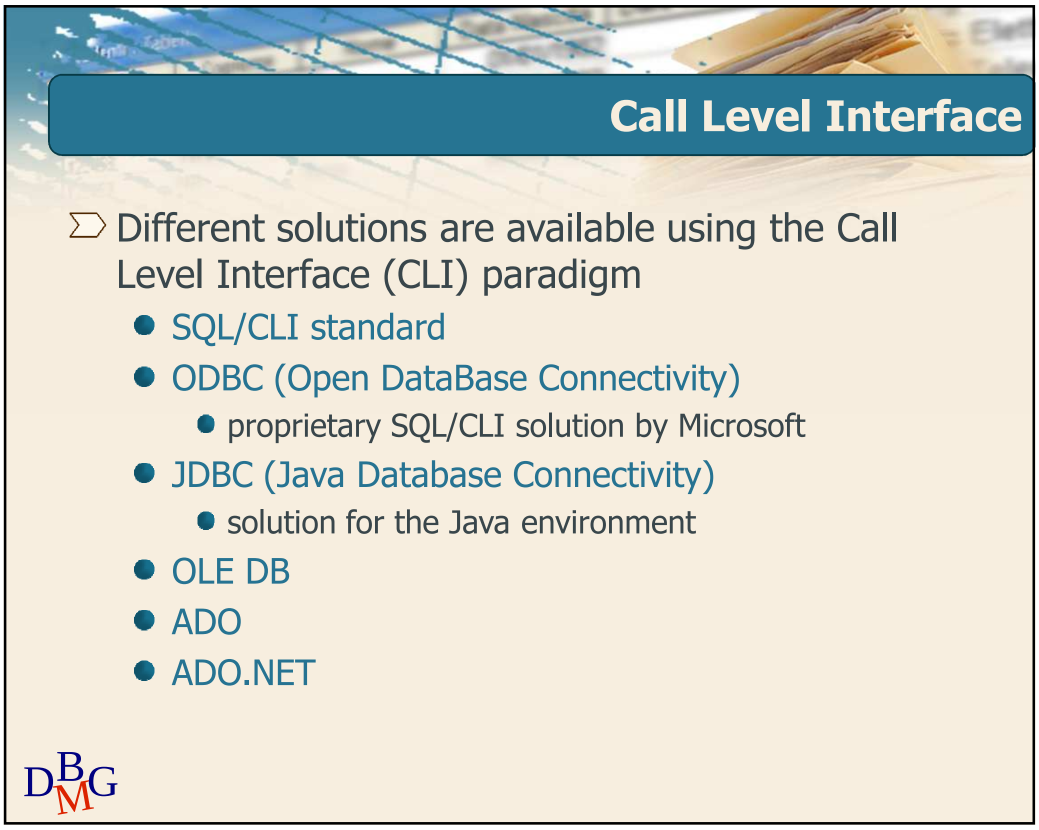
Call Level Interface (CLI)



Call Level Interface

- Requests are sent to the DBMS by using ad-hoc functions of the host language
  - solution based on predefined interfaces
    - API, Application Programming Interface
  - the SQL commands are passed to the host language functions as parameters
  - there is no precompiler
- The host program directly includes calls to the functions provided by the API

DBMG



Call Level Interface

- Different solutions are available using the Call Level Interface (CLI) paradigm
  - SQL/CLI standard
  - ODBC (Open DataBase Connectivity)
    - proprietary SQL/CLI solution by Microsoft
  - JDBC (Java Database Connectivity)
    - solution for the Java environment
  - OLE DB
  - ADO
  - ADO.NET

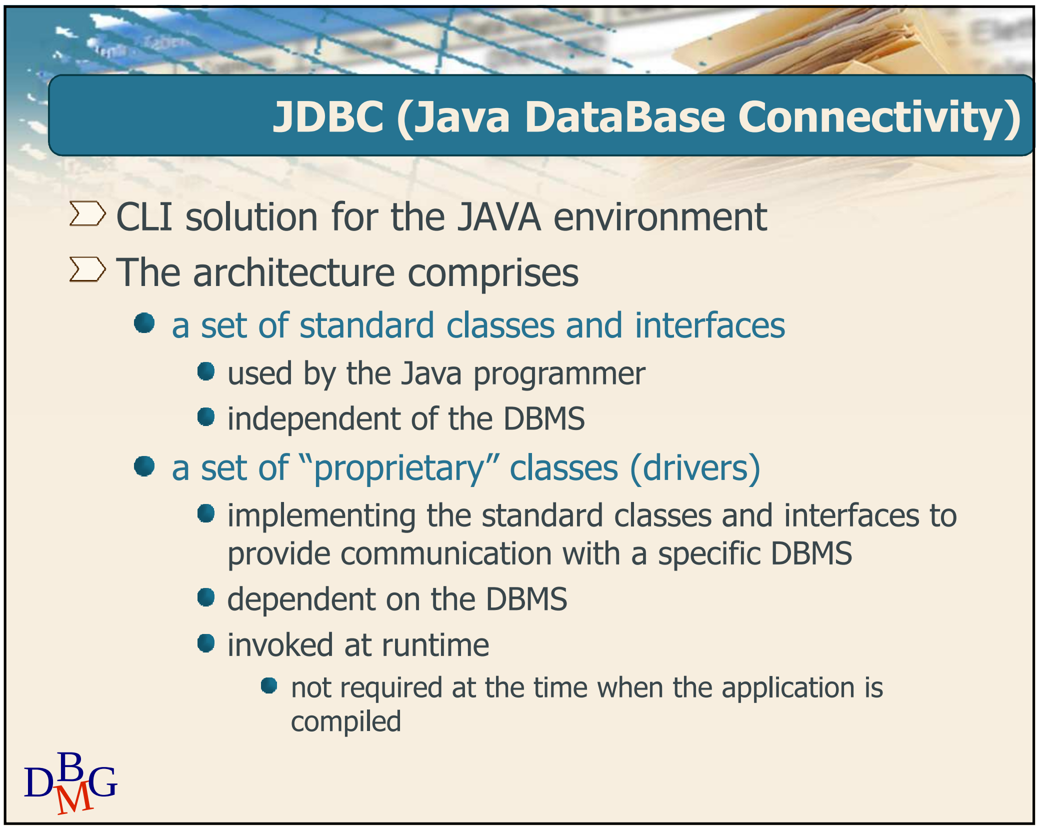
DBMG

Usage pattern

- Regardless of the specific CLI solution adopted, the interaction with the DBMS has a common structure
  - open a connection to the DBMS
  - execute SQL commands
  - close the connection

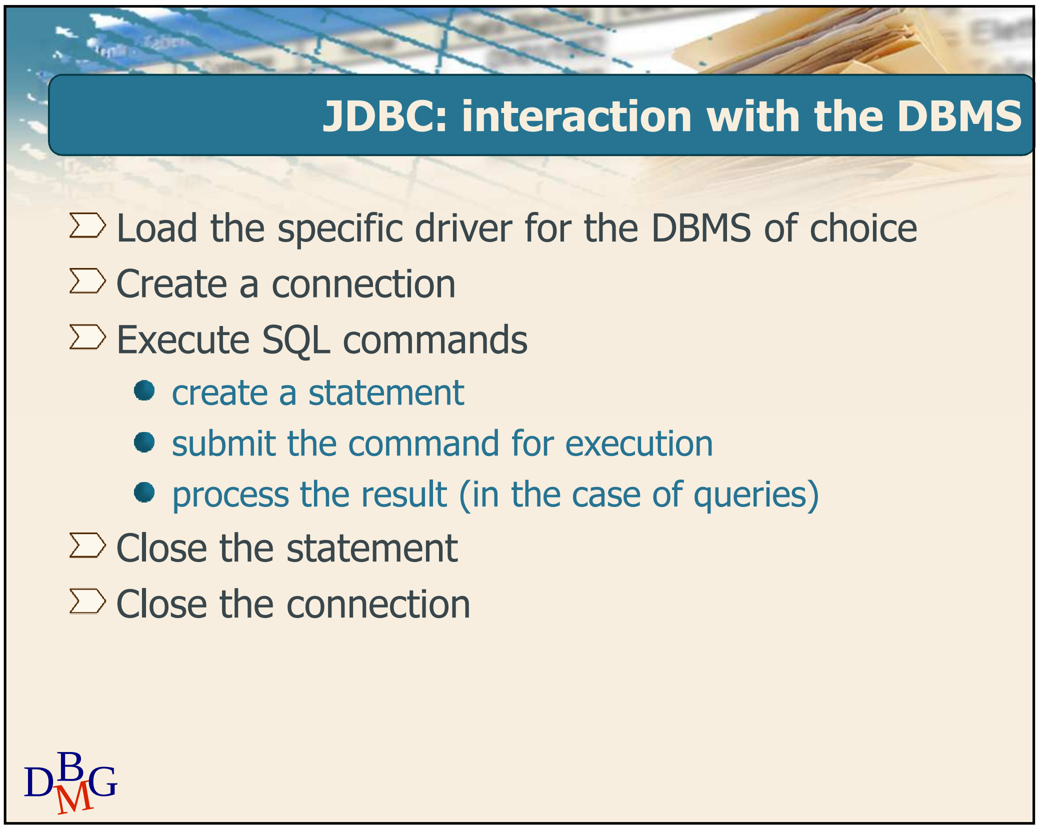
Interaction with the DBMS

1. Call an API primitive to create a connection to the DBMS
2. Send an SQL command across the connection
3. Receive a result in response to the command
   - i.e., a set of tuples, in the case of a SELECT command
4. Process the result obtained from the DBMS
   - ad-hoc primitives allow reading the result
5. Close the connection at the end of the working session



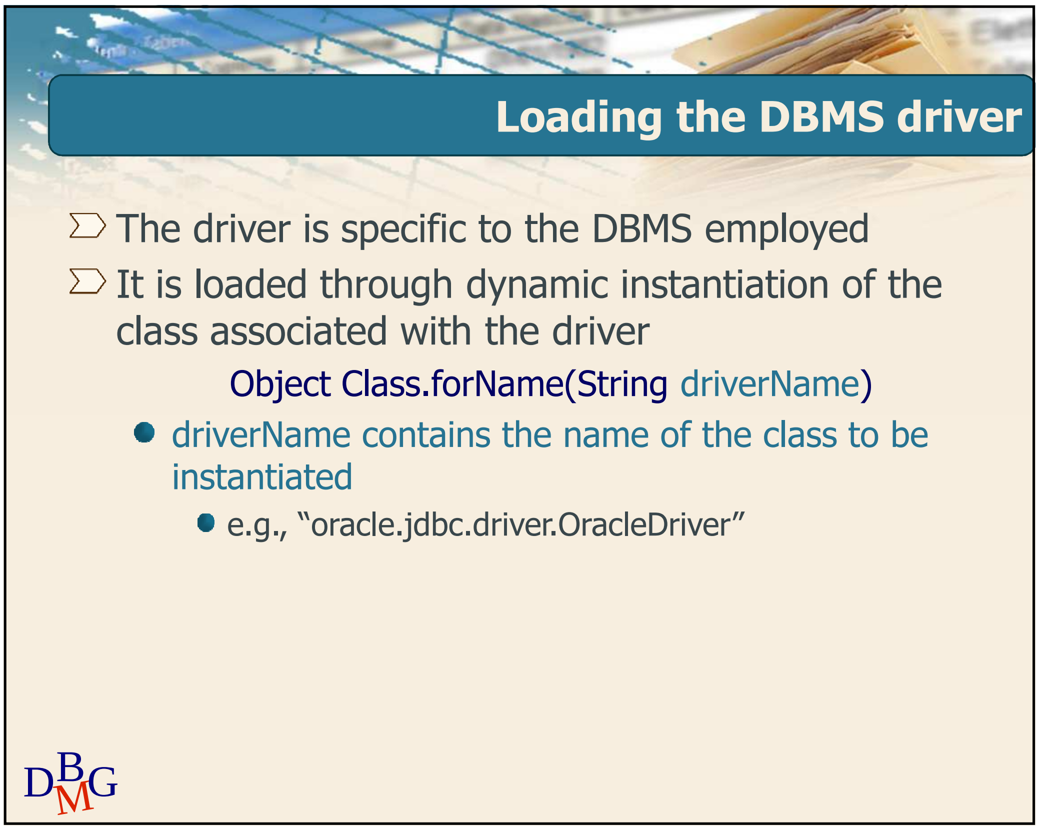
JDBC (Java DataBase Connectivity)

- CLI solution for the JAVA environment
- The architecture comprises
  - a set of standard classes and interfaces
    - used by the Java programmer
    - independent of the DBMS
  - a set of "proprietary" classes (drivers)
    - implementing the standard classes and interfaces to provide communication with a specific DBMS
    - dependent on the DBMS
    - invoked at runtime
      - not required at the time when the application is compiled



JDBC: interaction with the DBMS

- Load the specific driver for the DBMS of choice
- Create a connection
- Execute SQL commands
  - create a statement
  - submit the command for execution
  - process the result (in the case of queries)
- Close the statement
- Close the connection



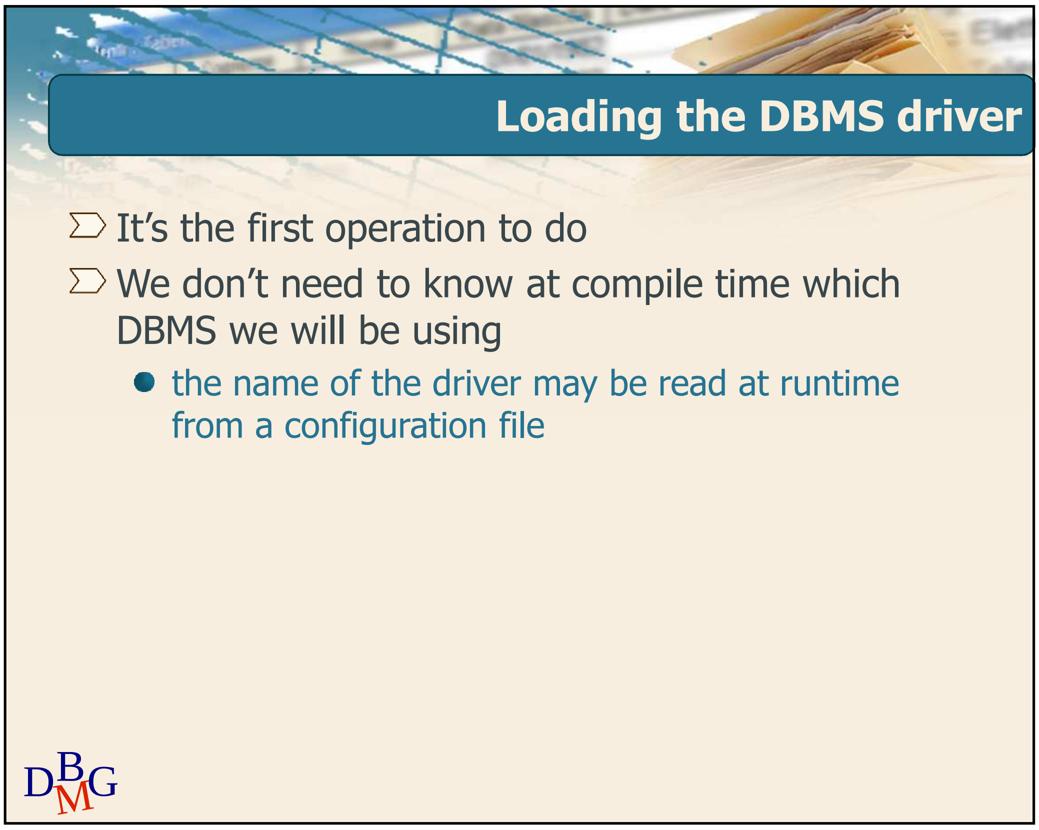
Loading the DBMS driver

- The driver is specific to the DBMS employed
- It is loaded through dynamic instantiation of the class associated with the driver

```
Object Class.forName(String driverName)
```

  - driverName contains the name of the class to be instantiated
    - e.g., "oracle.jdbc.driver.OracleDriver"

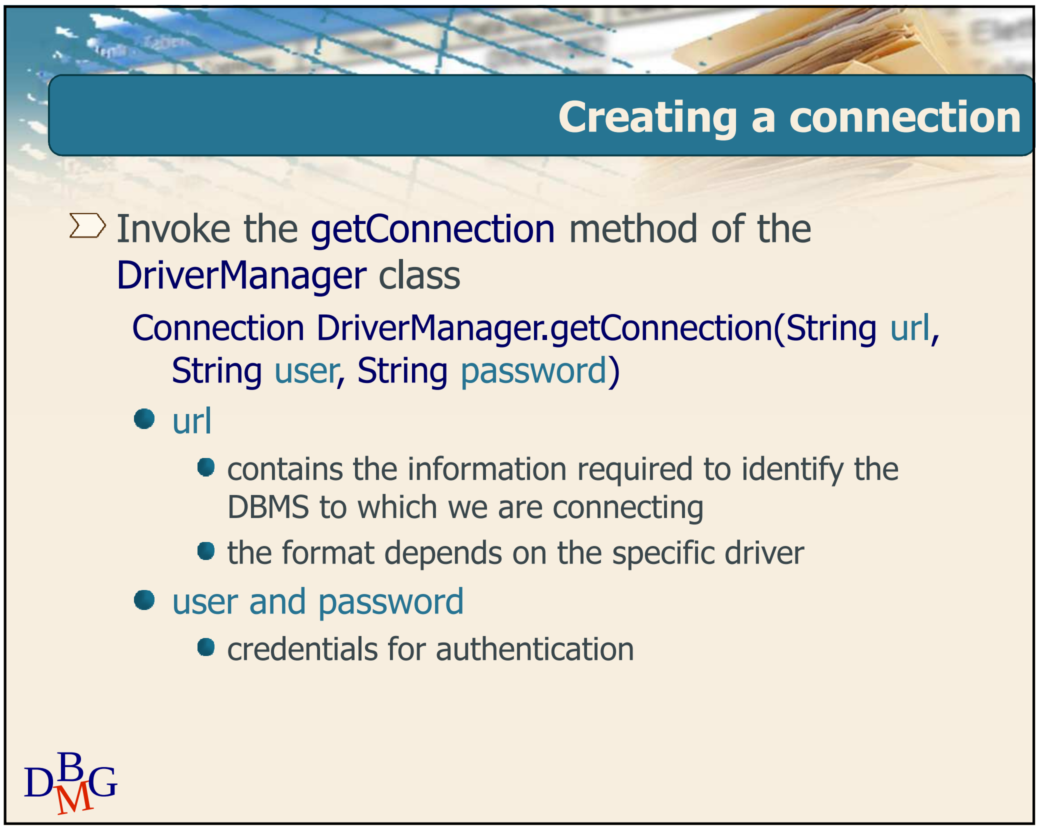
DBMG



Loading the DBMS driver

- It's the first operation to do
- We don't need to know at compile time which DBMS we will be using
  - the name of the driver may be read at runtime from a configuration file

DBMG



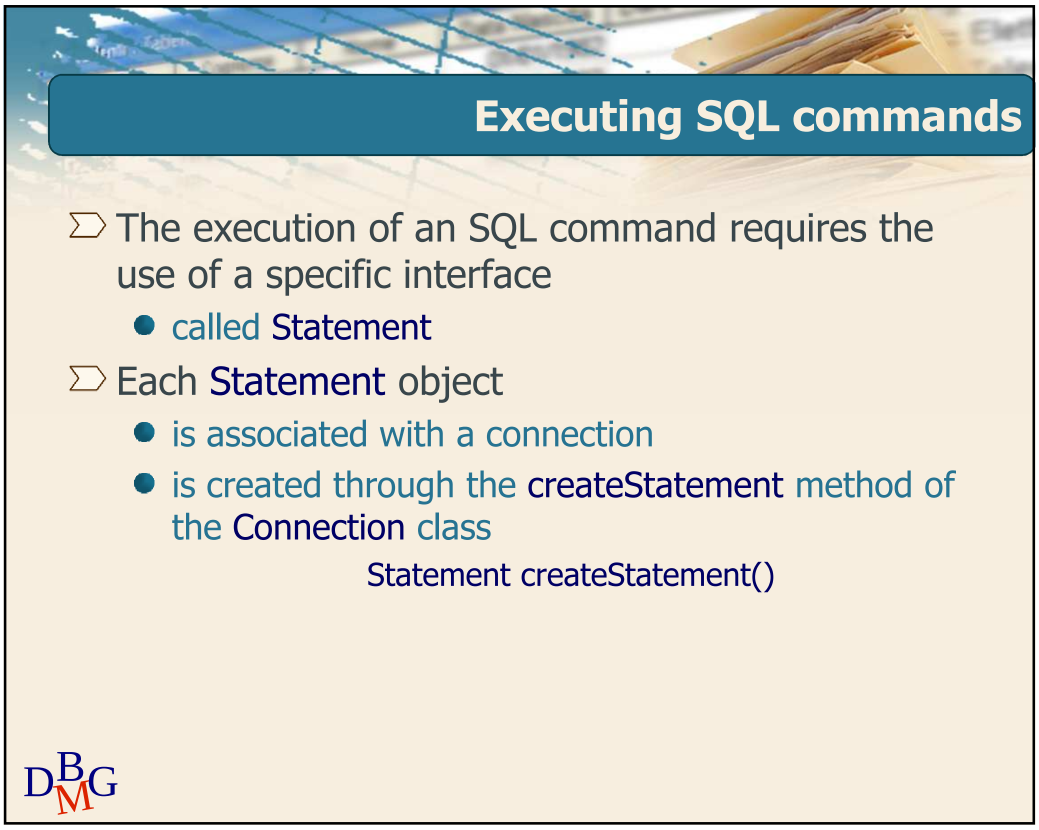
Creating a connection

- Invoke the getConnection method of the DriverManager class

  Connection DriverManager.getConnection(String url, String user, String password)

  - url
    - contains the information required to identify the DBMS to which we are connecting
    - the format depends on the specific driver
  - user and password
    - credentials for authentication

DBMG



Executing SQL commands

- The execution of an SQL command requires the use of a specific interface
  - called Statement
- Each Statement object
  - is associated with a connection
  - is created through the createStatement method of the Connection class

    Statement createStatement()

DBMG

Update and DDL commands

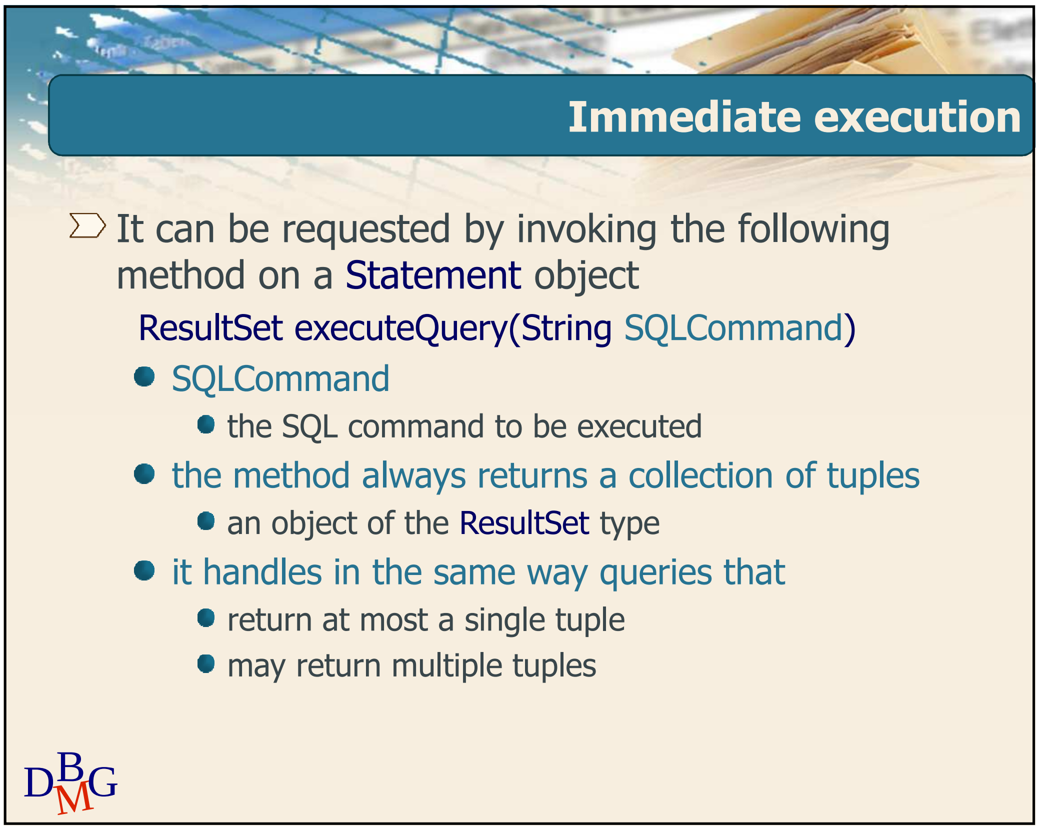
- The execution of the command requires invoking the following method on a Statement object

  int executeUpdate(String SQLCommand)

  - SQLCommand
    - the SQL command to be executed
  - the method returns
    - the number of processed (i.e., inserted, modified, deleted) tuples
    - a value of 0 for DDL commands

Queries

- Immediate query execution
  - the server compiles and immediately executes the SQL command received
- "Prepared" query execution
  - useful when the same SQL command must be executed multiple times in the same working session
    - only the values of parameters may change
  - the SQL command
    - is compiled (prepared) only once and its execution plan is stored by the DBMS
    - is executed several times throughout the session



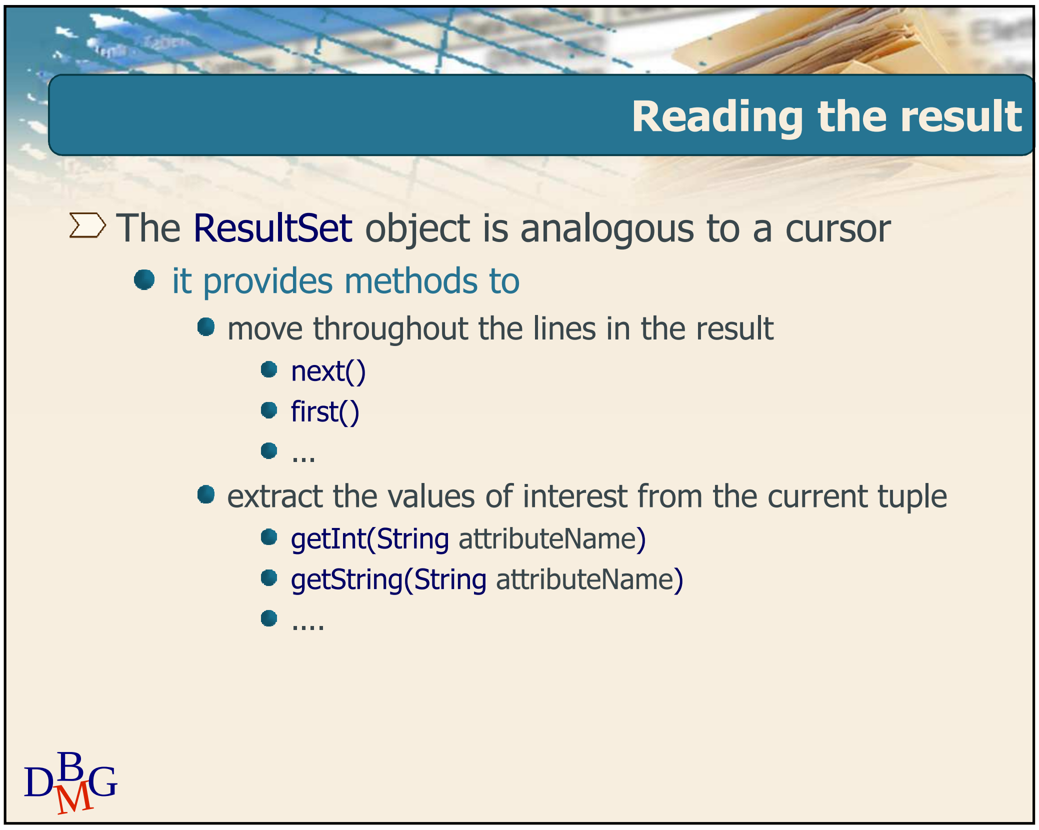
Immediate execution

- It can be requested by invoking the following method on a Statement object

  ResultSet executeQuery(String SQLCommand)

  - SQLCommand
    - the SQL command to be executed
  - the method always returns a collection of tuples
    - an object of the ResultSet type
  - it handles in the same way queries that
    - return at most a single tuple
    - may return multiple tuples

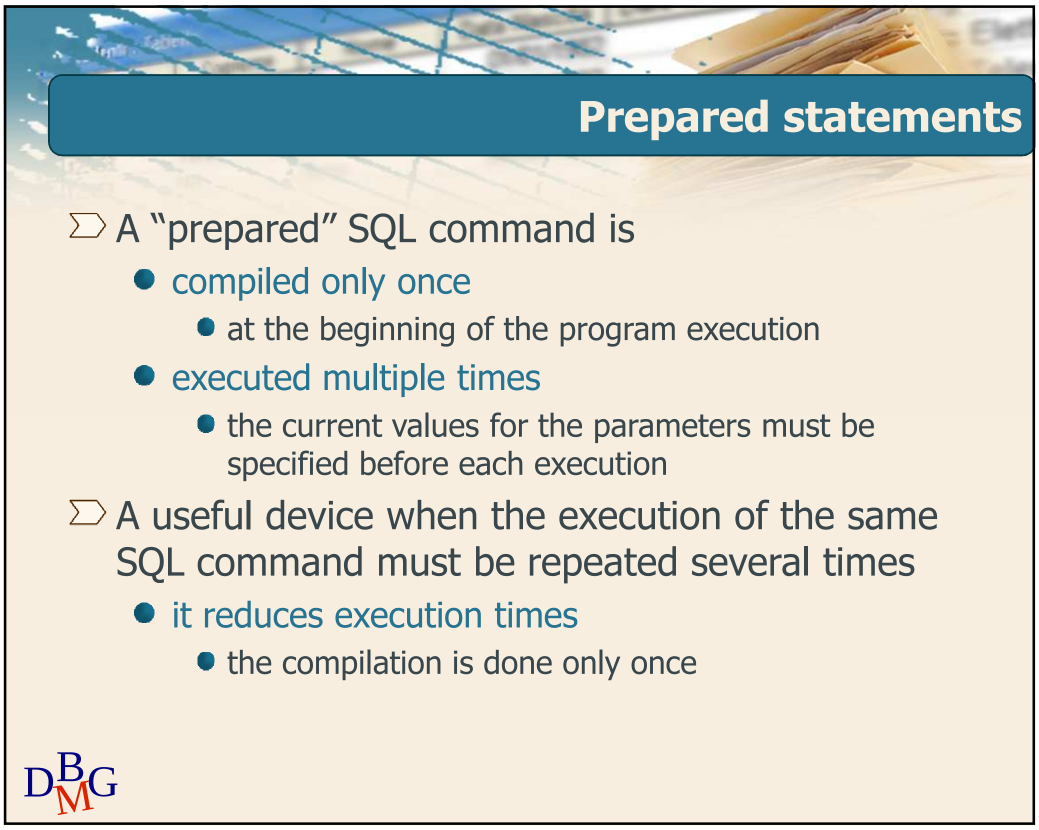
DBMG



Reading the result

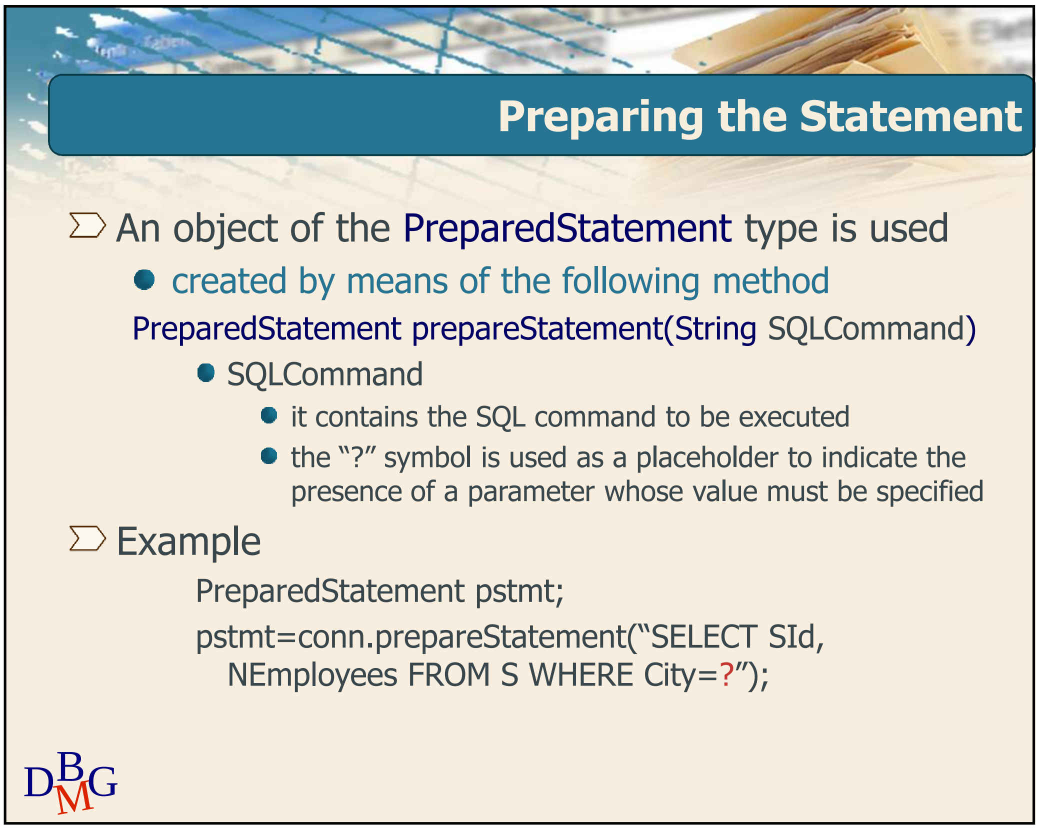
- The ResultSet object is analogous to a cursor
  - it provides methods to
    - move throughout the lines in the result
      - next()
      - first()
      - ...
    - extract the values of interest from the current tuple
      - getInt(String attributeName)
      - getString(String attributeName)
      - ....

DBMG



Prepared statements

- A "prepared" SQL command is
  - compiled only once
    - at the beginning of the program execution
  - executed multiple times
    - the current values for the parameters must be specified before each execution
- A useful device when the execution of the same SQL command must be repeated several times
  - it reduces execution times
    - the compilation is done only once

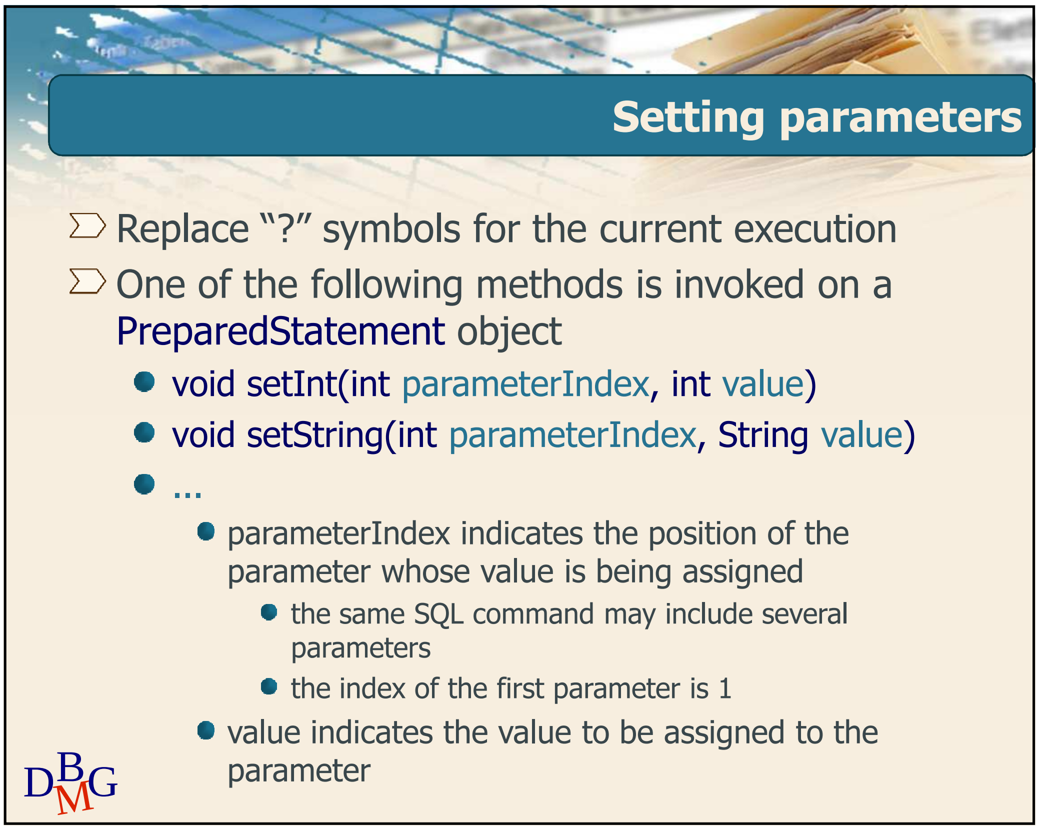


Preparing the Statement

- An object of the PreparedStatement type is used
  - created by means of the following method

  PreparedStatement prepareStatement(String SQLCommand)
    - SQLCommand
      - it contains the SQL command to be executed
      - the "?" symbol is used as a placeholder to indicate the presence of a parameter whose value must be specified
- Example

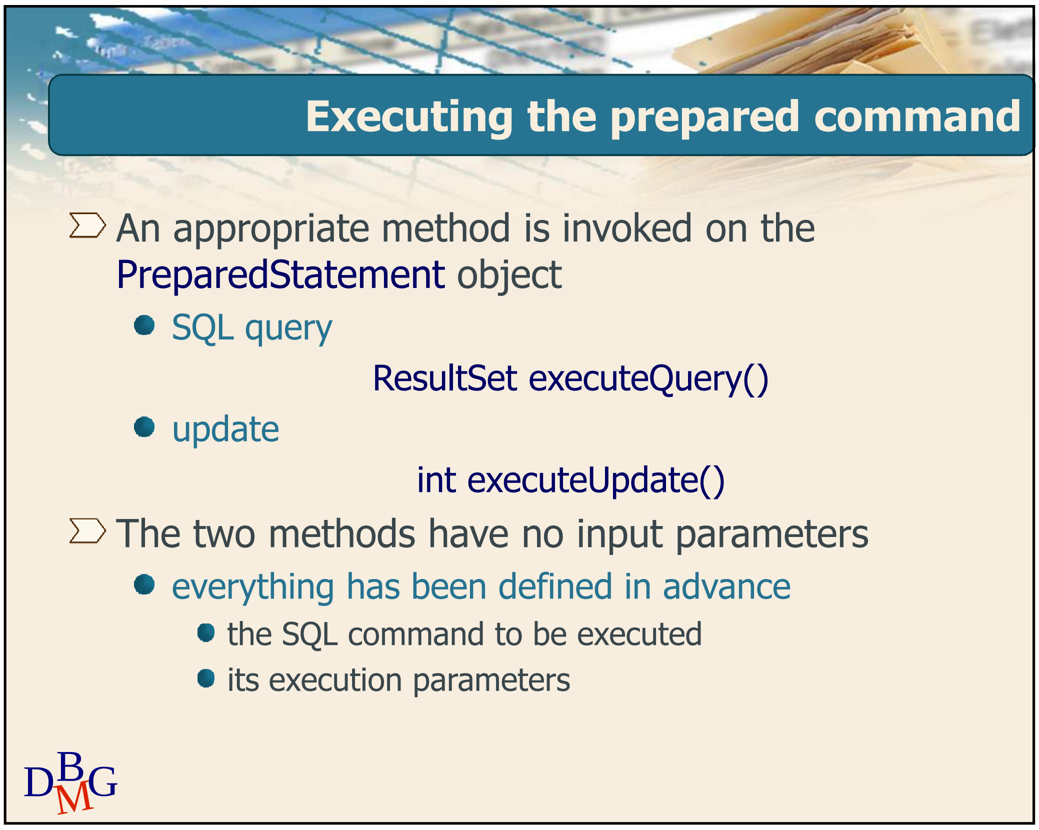
```
PreparedStatement pstmt;
pstmt=conn.prepareStatement("SELECT SId,
   NEmployees FROM S WHERE City=?");
```



Setting parameters

- Replace "?" symbols for the current execution
- One of the following methods is invoked on a PreparedStatement object
  - void setInt(int parameterIndex, int value)
  - void setString(int parameterIndex, String value)
  - …
    - parameterIndex indicates the position of the parameter whose value is being assigned
      - the same SQL command may include several parameters
      - the index of the first parameter is 1
    - value indicates the value to be assigned to the parameter

DBMG



Executing the prepared command

- An appropriate method is invoked on the PreparedStatement object
  - SQL query

    ResultSet executeQuery()
  - update

    int executeUpdate()
- The two methods have no input parameters
  - everything has been defined in advance
    - the SQL command to be executed
    - its execution parameters

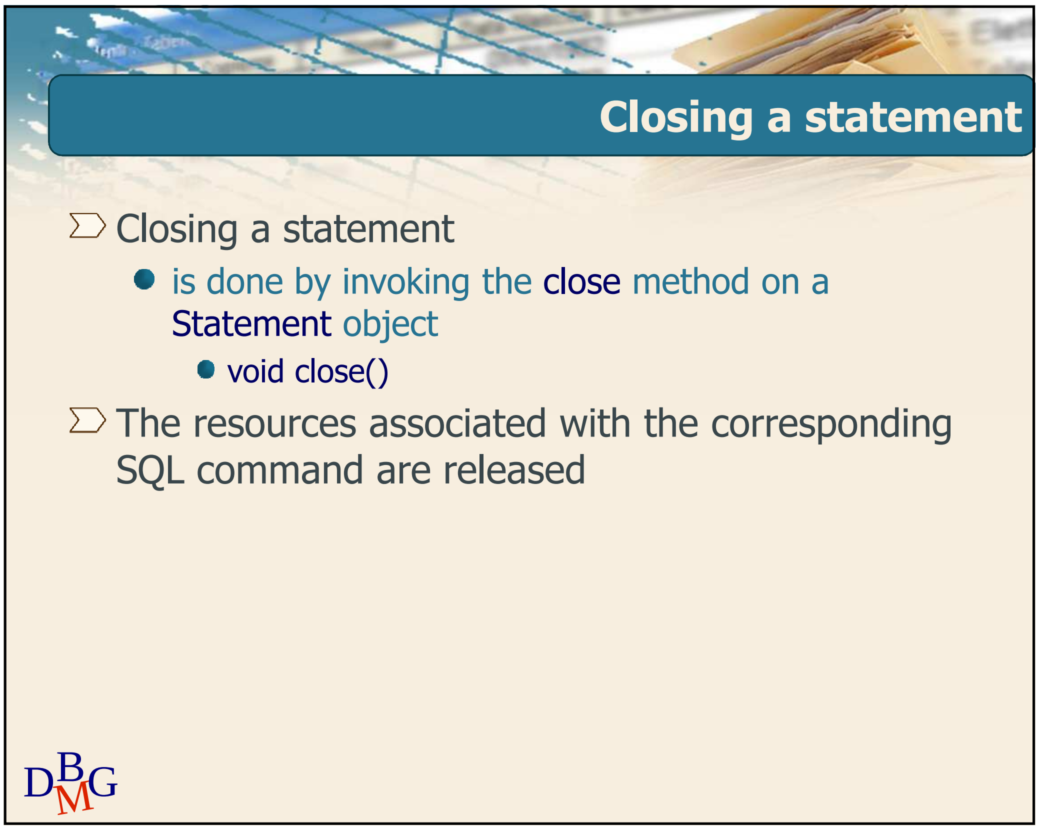
DBMG

Example: prepared statements

```
.....
PreparedStatement pstmt=conn.prepareStatement("UPDATE P
    SET Color=? WHERE PId=?");

/* Assign color Crimson to product P1 */
pstmt.setString(1, "Crimson");
pstmt.setString(2, "P1");
pstmt.executeUpdate();

/* Assign color SteelBlue to product P5 */
pstmt.setString(1, "SteelBlue");
pstmt.setString(2, "P5");
pstmt.executeUpdate();
```

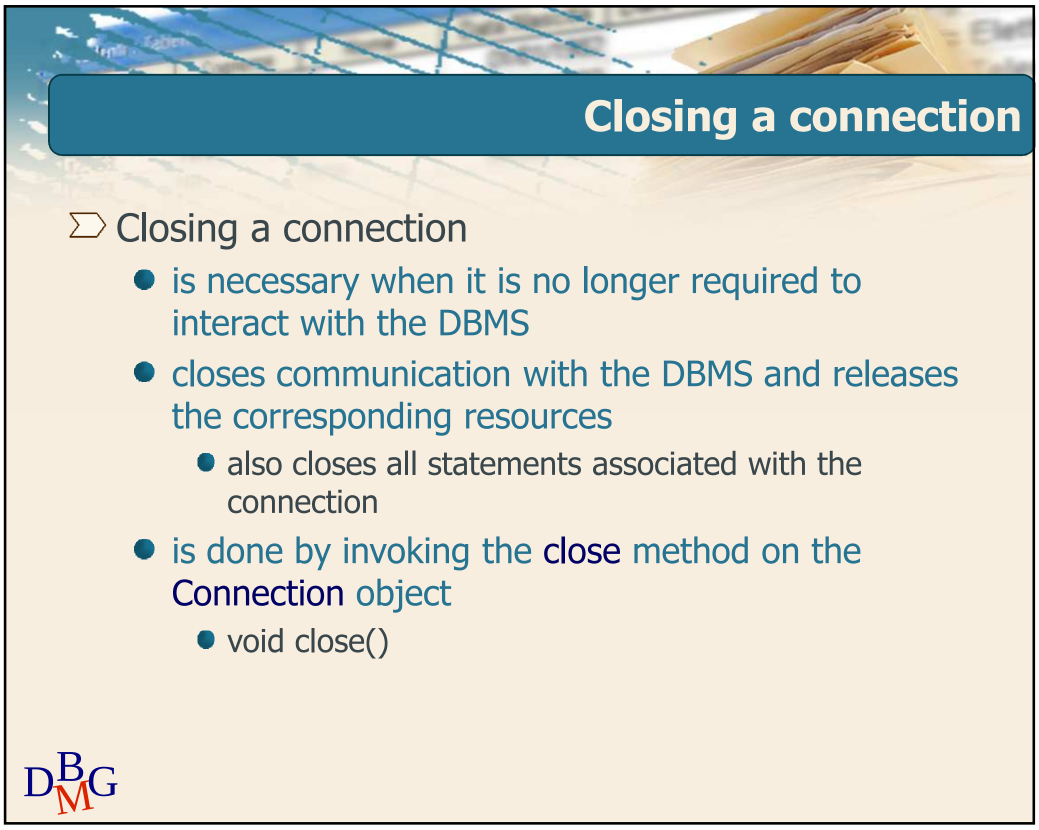
Closing statement and connection

- As soon as a statement or a connection are no longer needed
  - they must be immediately closed
- Resources previously allocated to the statement or the connection can be released
  - by the application
  - by the DBMS



Closing a statement

- Closing a statement
  - is done by invoking the close method on a Statement object
    - void close()
- The resources associated with the corresponding SQL command are released



Closing a connection

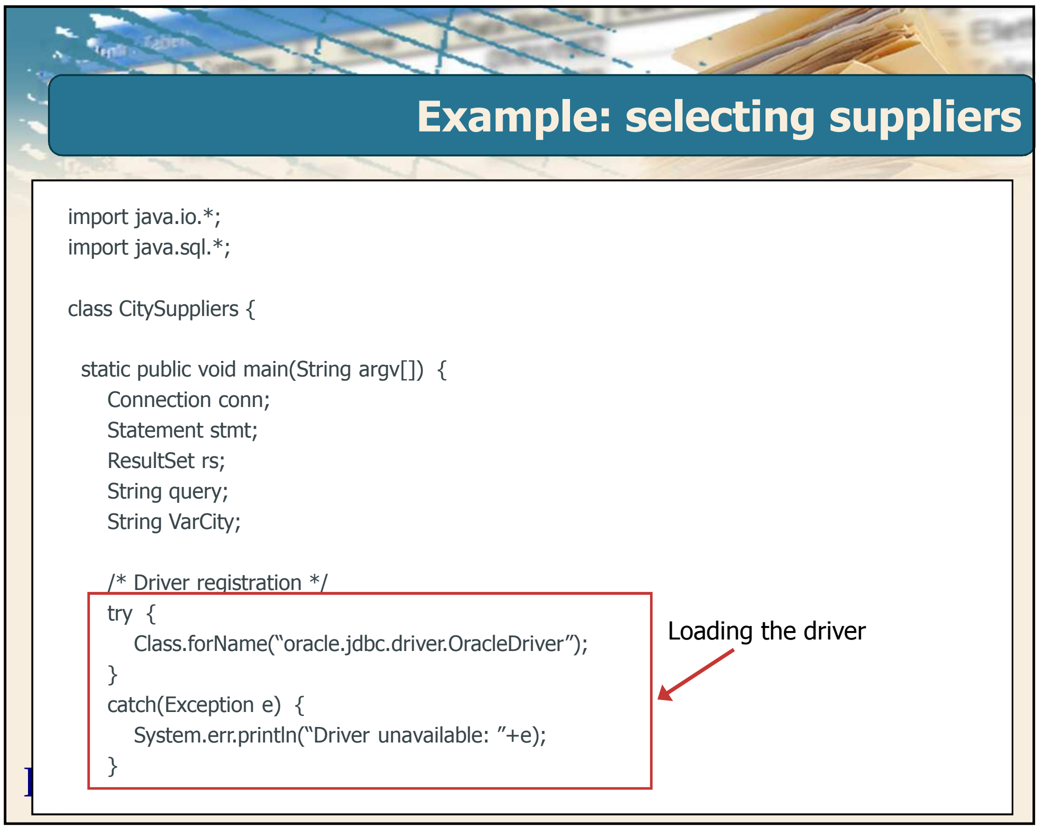
- Closing a connection
  - is necessary when it is no longer required to interact with the DBMS
  - closes communication with the DBMS and releases the corresponding resources
    - also closes all statements associated with the connection
  - is done by invoking the close method on the Connection object
    - void close()

Example: selecting suppliers

- Print the codes and the number of employees of the suppliers whose city is stored in host variable *VarCity*
  - the value of *VarCity* is provided by the user as a parameter of the application

Example: selecting suppliers

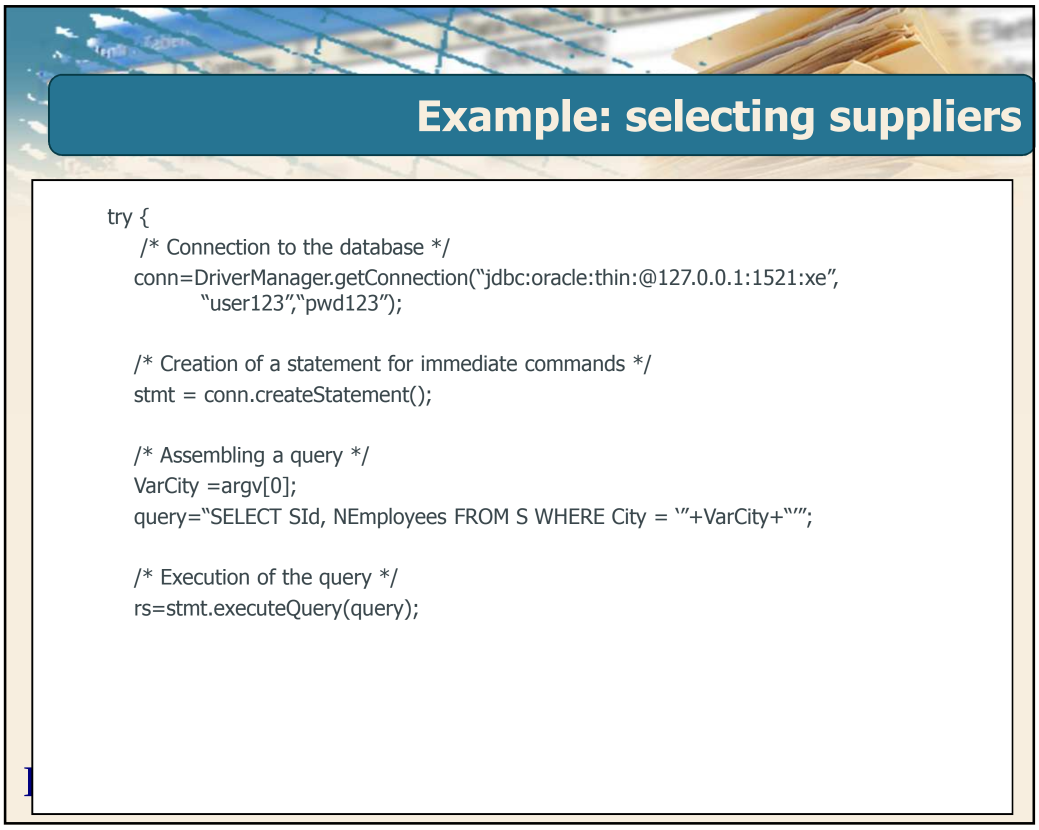
```
import java.io.*;
import java.sql.*;

class CitySuppliers {

  static public void main(String argv[]) {
     Connection conn;
     Statement stmt;
     ResultSet rs;
     String query;
     String VarCity;

     /* Driver registration */
     try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
     }
     catch(Exception e) {
        System.err.println("Driver unavailable: "+e);
     }
```



Example: selecting suppliers

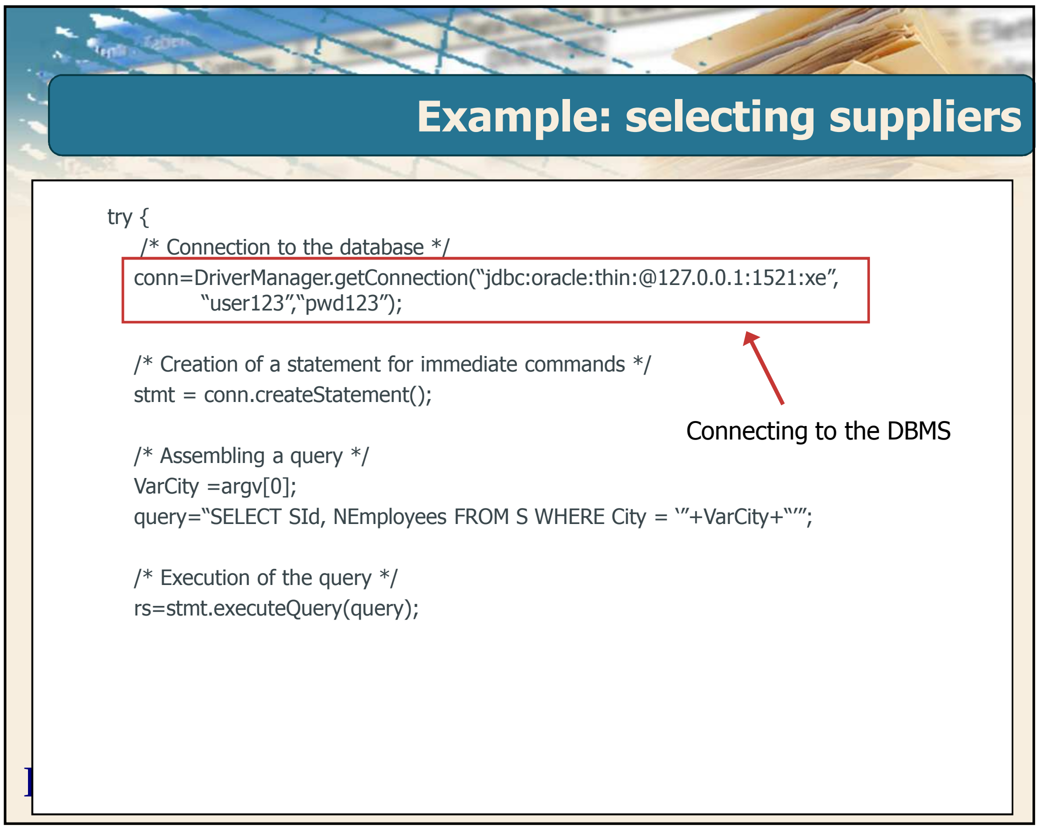
```
import java.io.*;
import java.sql.*;

class CitySuppliers {

  static public void main(String argv[]) {
      Connection conn;
      Statement stmt;
      ResultSet rs;
      String query;
      String VarCity;

      /* Driver registration */
      try {
          Class.forName(“oracle.jdbc.driver.OracleDriver”);
      }
      catch(Exception e) {
          System.err.println(“Driver unavailable: ”+e);
      }
```

Loading the driver



Example: selecting suppliers

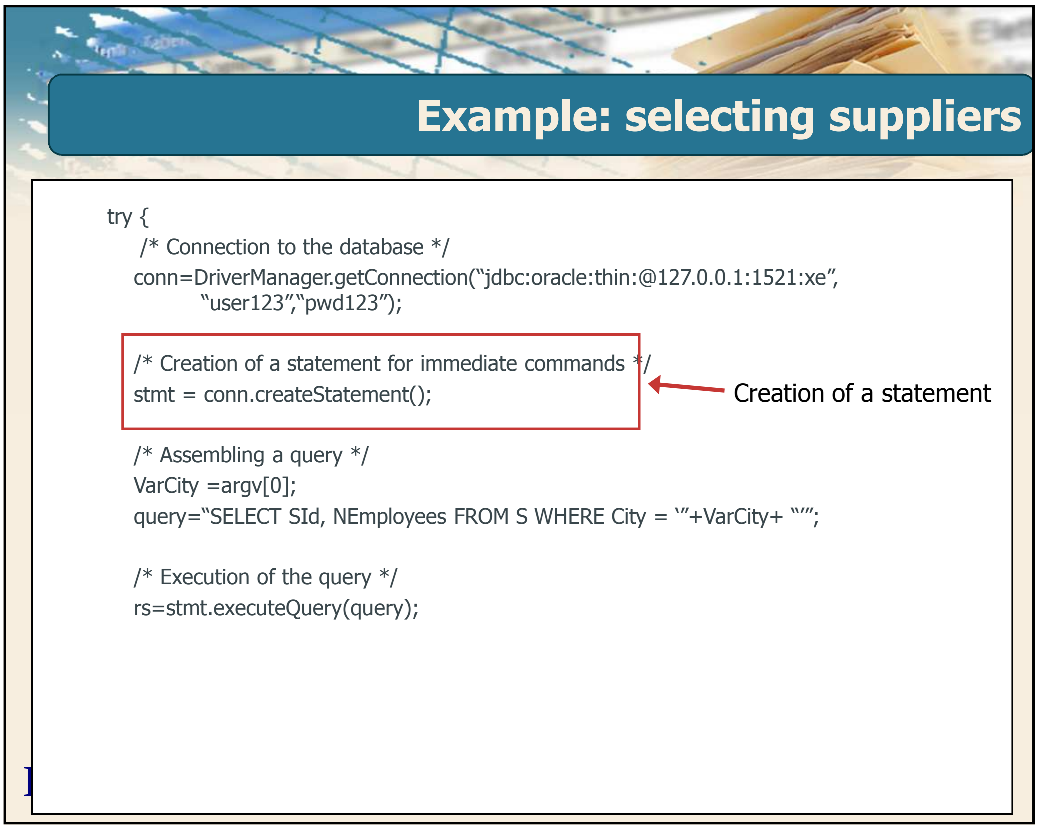
```
      try {
           /* Connection to the database */
          conn=DriverManager.getConnection(“jdbc:oracle:thin:@127.0.0.1:1521:xe”,
                     “user123”,“pwd123”);

          /* Creation of a statement for immediate commands */
          stmt = conn.createStatement();

          /* Assembling a query */
          VarCity =argv[0];
          query=“SELECT SId, NEmployees FROM S WHERE City = '”+VarCity+“'”;

          /* Execution of the query */
          rs=stmt.executeQuery(query);
```



Example: selecting suppliers

```
try {
    /* Connection to the database */
  conn=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",
          "user123","pwd123");

  /* Creation of a statement for immediate commands */
  stmt = conn.createStatement();

  /* Assembling a query */
  VarCity =argv[0];
  query="SELECT SId, NEmployees FROM S WHERE City = '"+VarCity+"'";

  /* Execution of the query */
  rs=stmt.executeQuery(query);
```

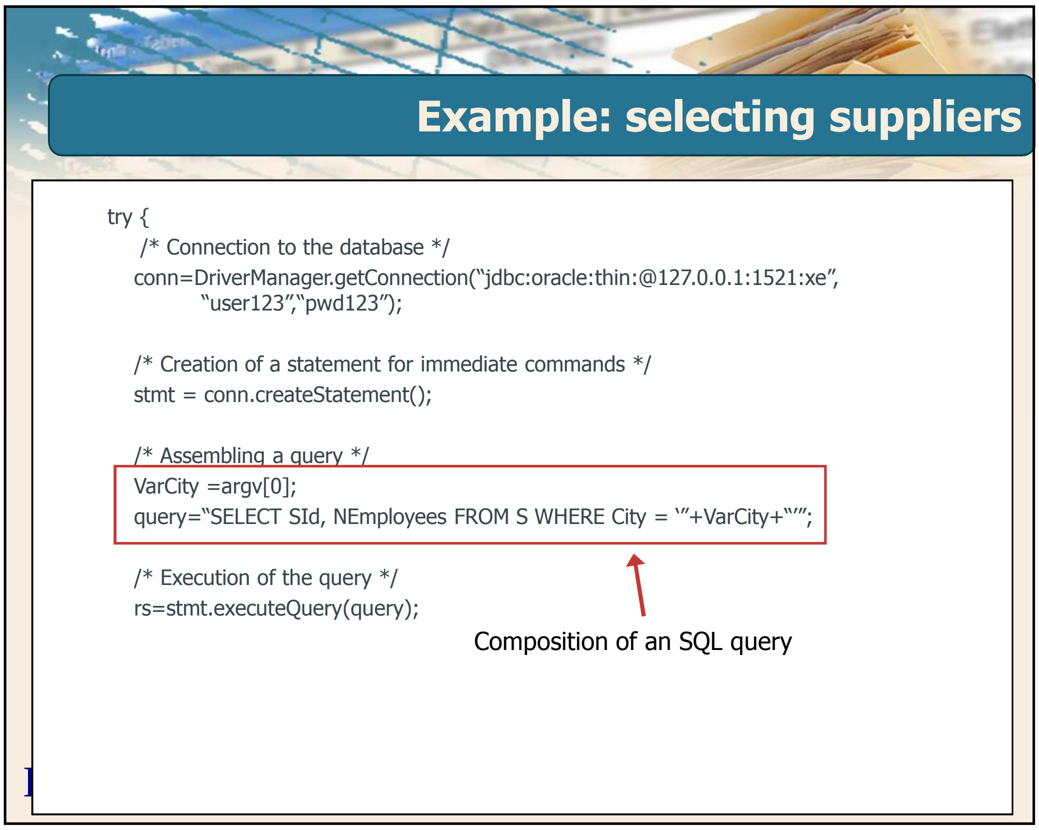
Connecting to the DBMS



Example: selecting suppliers

```
try {
    /* Connection to the database */
  conn=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",
          "user123","pwd123");

  /* Creation of a statement for immediate commands */
  stmt = conn.createStatement();

  /* Assembling a query */
  VarCity =argv[0];
  query="SELECT SId, NEmployees FROM S WHERE City = '"+VarCity+ "'";

  /* Execution of the query */
  rs=stmt.executeQuery(query);
```

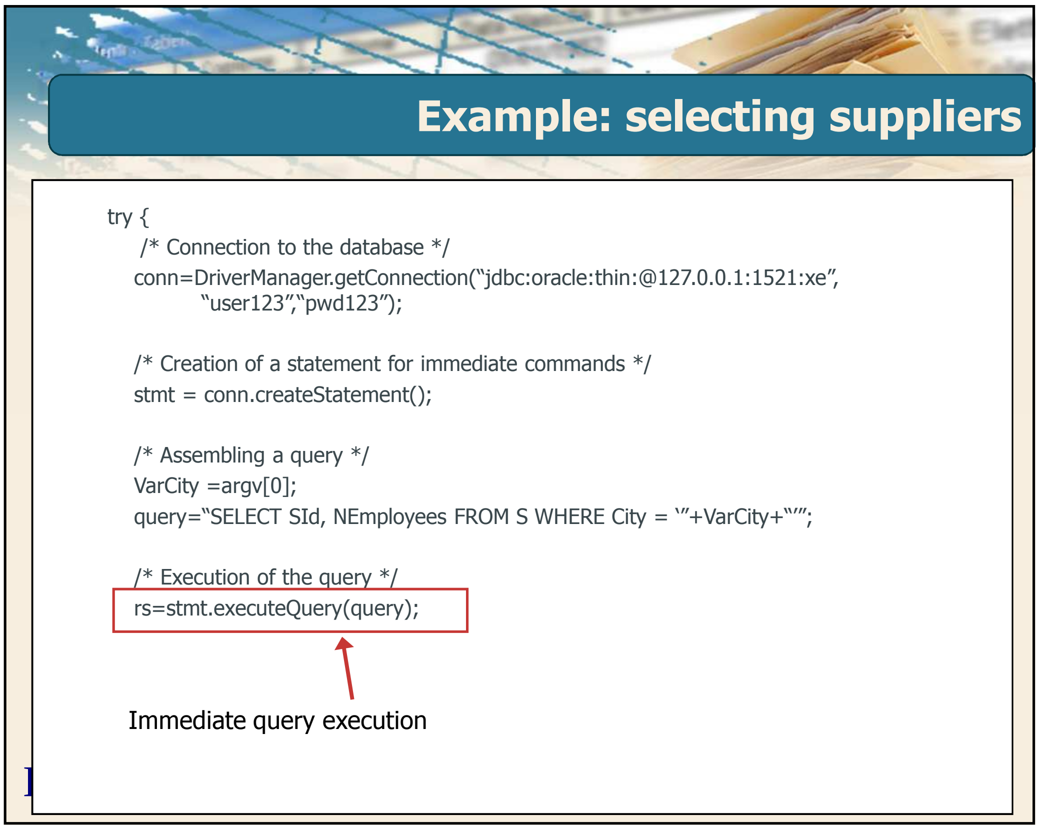
Creation of a statement



Example: selecting suppliers

```
try {
   /* Connection to the database */
  conn=DriverManager.getConnection(“jdbc:oracle:thin:@127.0.0.1:1521:xe”,
          “user123”,“pwd123”);

  /* Creation of a statement for immediate commands */
  stmt = conn.createStatement();

  /* Assembling a query */
  VarCity =argv[0];
  query=“SELECT SId, NEmployees FROM S WHERE City = ‘”+VarCity+“’”;

  /* Execution of the query */
  rs=stmt.executeQuery(query);
```

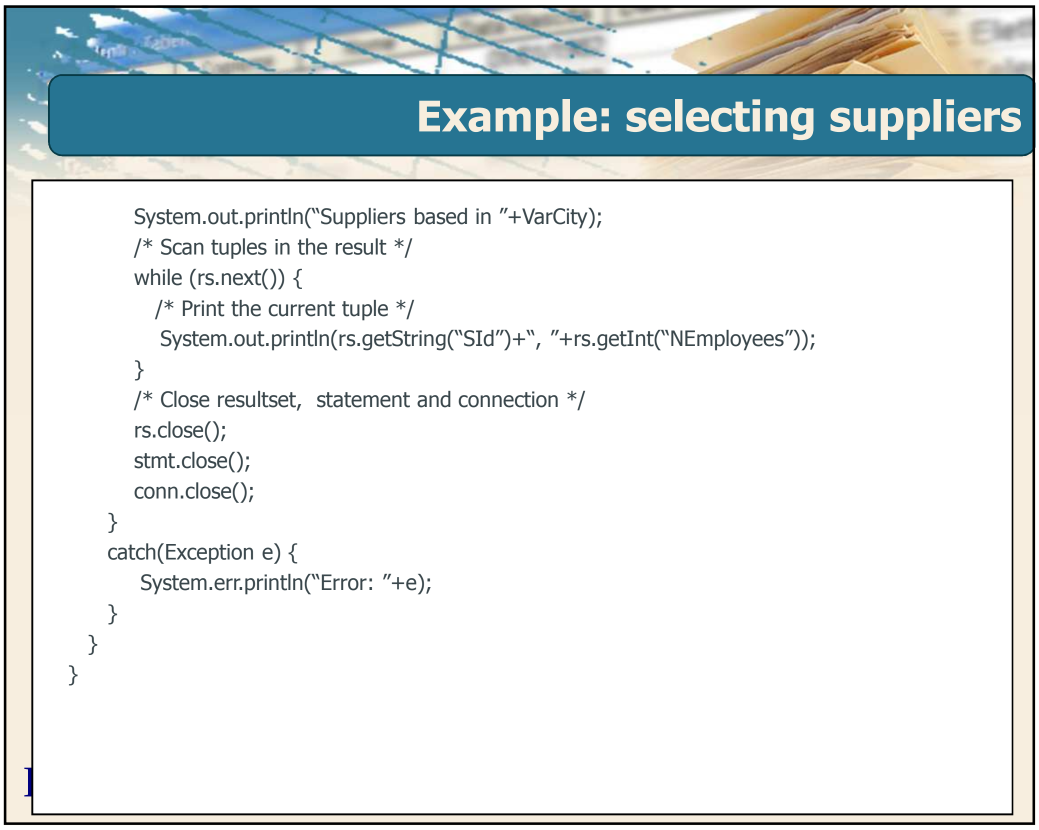
Composition of an SQL query



Example: selecting suppliers

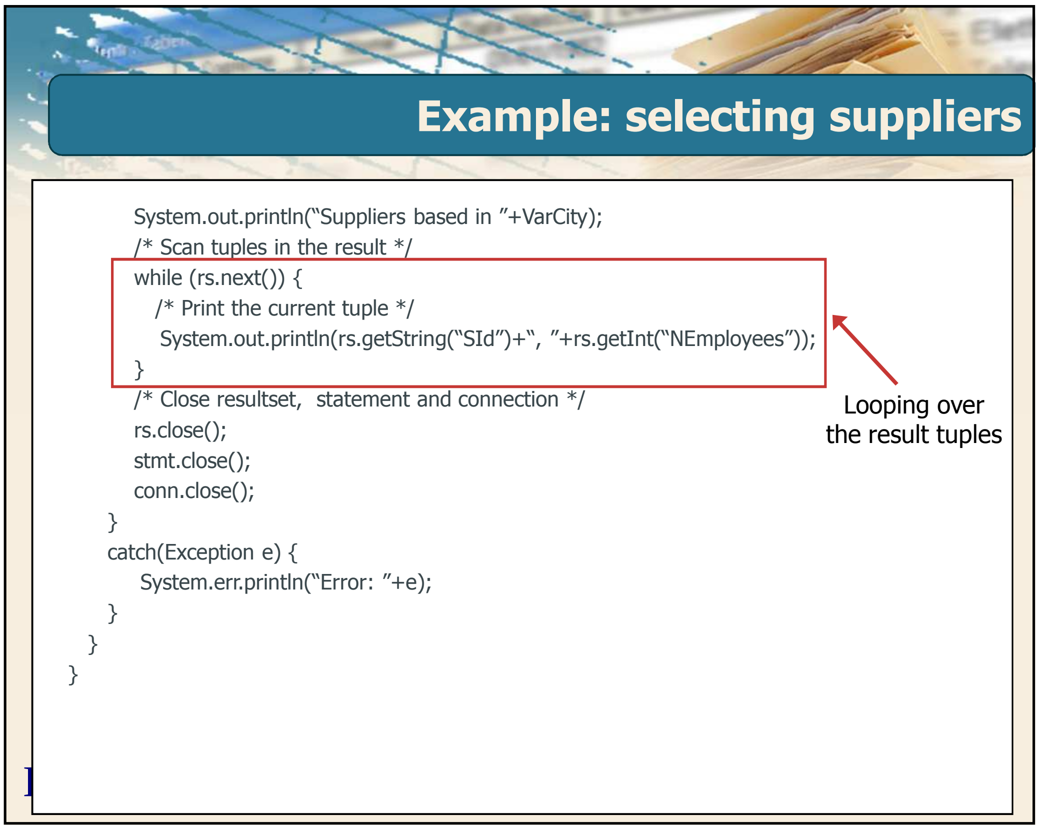
```
try {
   /* Connection to the database */
  conn=DriverManager.getConnection(“jdbc:oracle:thin:@127.0.0.1:1521:xe”,
          “user123”,“pwd123”);

  /* Creation of a statement for immediate commands */
  stmt = conn.createStatement();

  /* Assembling a query */
  VarCity =argv[0];
  query=“SELECT SId, NEmployees FROM S WHERE City = ‘”+VarCity+“’”;

  /* Execution of the query */
  rs=stmt.executeQuery(query);
```

Immediate query execution



Example: selecting suppliers

```
          System.out.println("Suppliers based in "+VarCity);
          /* Scan tuples in the result */
          while (rs.next()) {
            /* Print the current tuple */
             System.out.println(rs.getString("SId")+", "+rs.getInt("NEmployees"));
          }
          /* Close resultset,  statement and connection */
          rs.close();
          stmt.close();
          conn.close();
       }
       catch(Exception e) {
           System.err.println("Error: "+e);
       }
    }
 }
```



Example: selecting suppliers

```
          System.out.println("Suppliers based in "+VarCity);
          /* Scan tuples in the result */
          while (rs.next()) {
            /* Print the current tuple */
             System.out.println(rs.getString("SId")+", "+rs.getInt("NEmployees"));
          }
          /* Close resultset,  statement and connection */
          rs.close();
          stmt.close();
          conn.close();
       }
       catch(Exception e) {
           System.err.println("Error: "+e);
       }
    }
 }
```

Looping over the result tuples

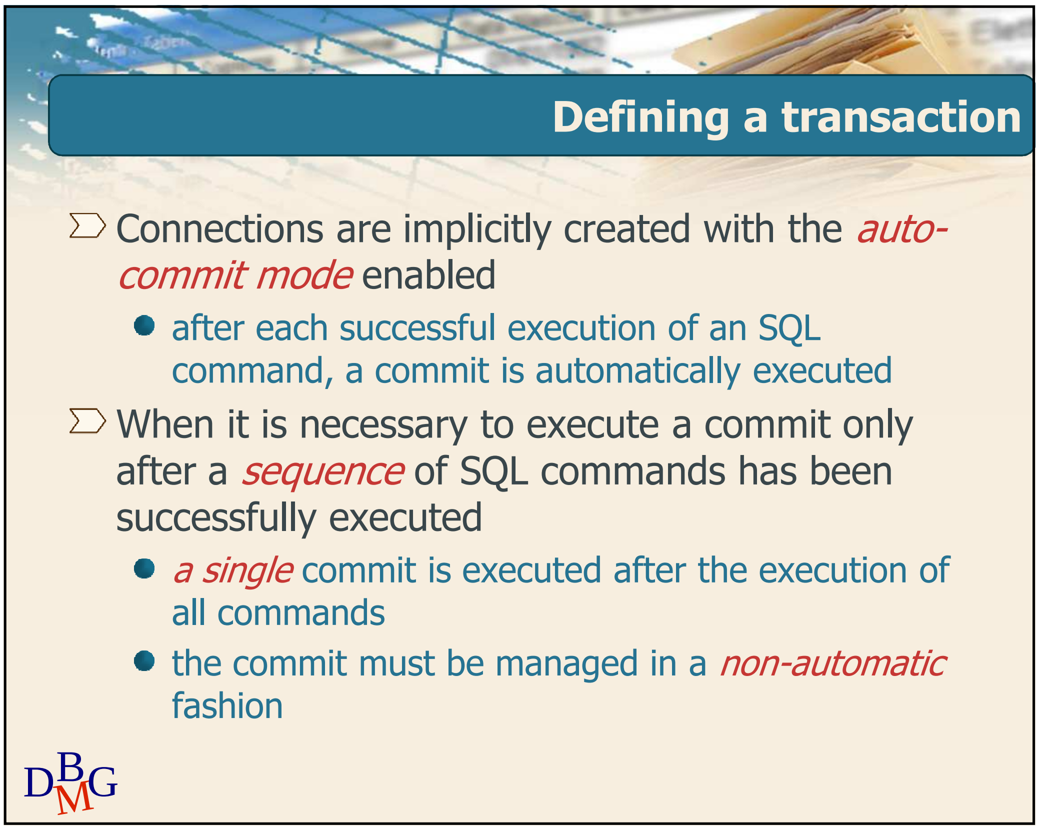
Example: selecting suppliers

```
            System.out.println(“Suppliers based in ”+VarCity);
            /* Scan tuples in the result */
            while (rs.next()) {
               /* Print the current tuple */
                System.out.println(rs.getString(“SId”)+“, ”+rs.getInt(“NEmployees”));
            }
            /* Close resultset,  statement and connection */
            rs.close();
            stmt.close();
            conn.close();
        }
        catch(Exception e) {
             System.err.println(“Error: ”+e);
        }
      }
    }
```

Closing resultset, statement and connection

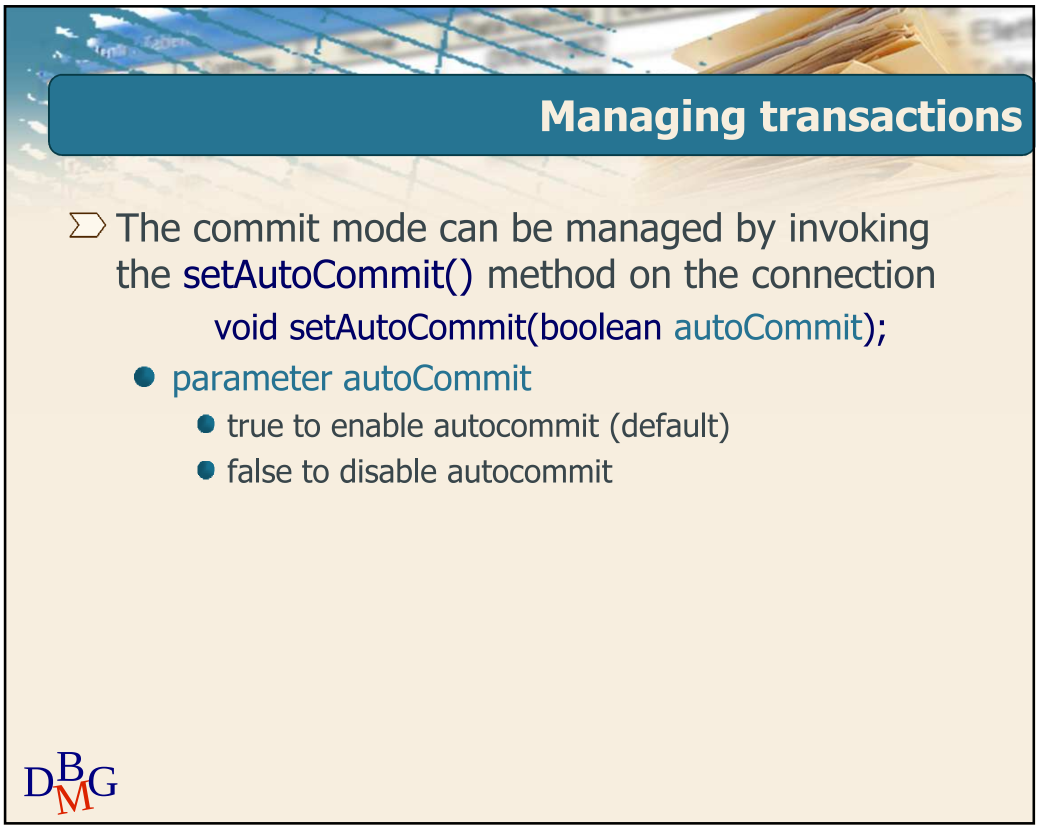
Updatable ResultSets

- It is possible to create an updatable ResultSet
  - the execution of updates on the database is more efficient
  - it is similar to an updatable cursor
    - there must be a one-to-one correspondence between the tuples in the result set and the tuples in the database tables



Defining a transaction

- Connections are implicitly created with the *auto-commit mode* enabled
  - after each successful execution of an SQL command, a commit is automatically executed
- When it is necessary to execute a commit only after a *sequence* of SQL commands has been successfully executed
  - *a single* commit is executed after the execution of all commands
  - the commit must be managed in a *non-automatic* fashion

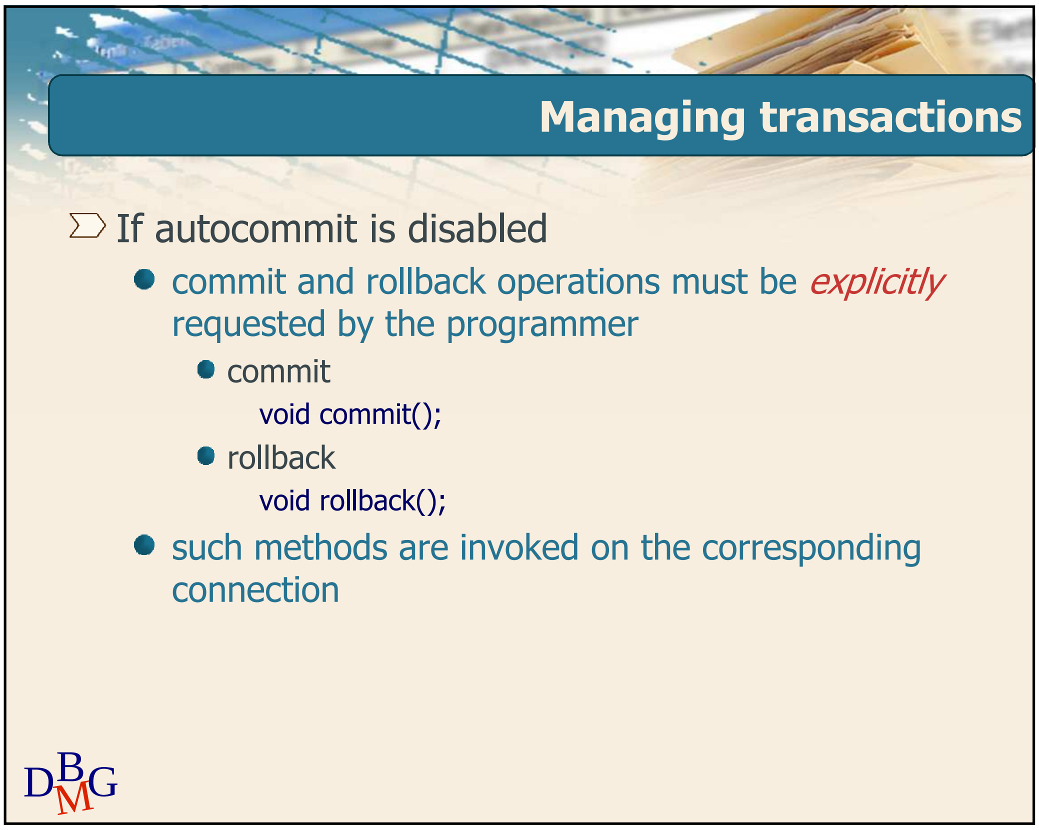


Managing transactions

- The commit mode can be managed by invoking the setAutoCommit() method on the connection

```
void setAutoCommit(boolean autoCommit);
```

  - parameter autoCommit
    - true to enable autocommit (default)
    - false to disable autocommit



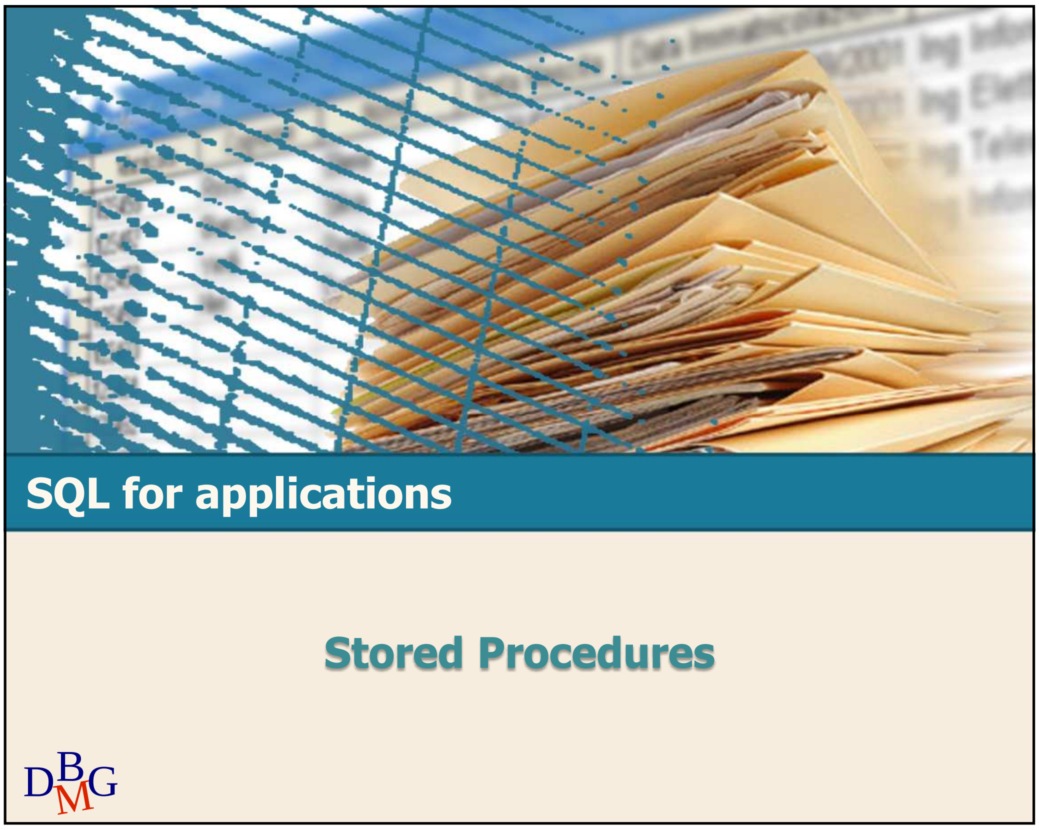
Managing transactions

- If autocommit is disabled
  - commit and rollback operations must be *explicitly* requested by the programmer
    - commit

      ```
      void commit();
      ```
    - rollback

      ```
      void rollback();
      ```
  - such methods are invoked on the corresponding connection

DBMG



SQL for applications

Stored Procedures

DBMG

Stored procedures

- A stored procedure is a function or a procedure defined inside the DBMS
  - it is stored in the data dictionary
    - it is part of the database schema
- It may be used like a predefined SQL command
  - it may have execution parameters
- It contains both application code and SQL commands
  - application code and SQL commands are tightly coupled to each other

DBMG

Stored procedures: language

- The language used to define a stored procedure
  - is a procedural extension of the SQL language
  - depends on the DBMS
    - different products may offer different languages
    - the expressiveness of the language may vary according to the product

DBMG

Stored procedures: execution

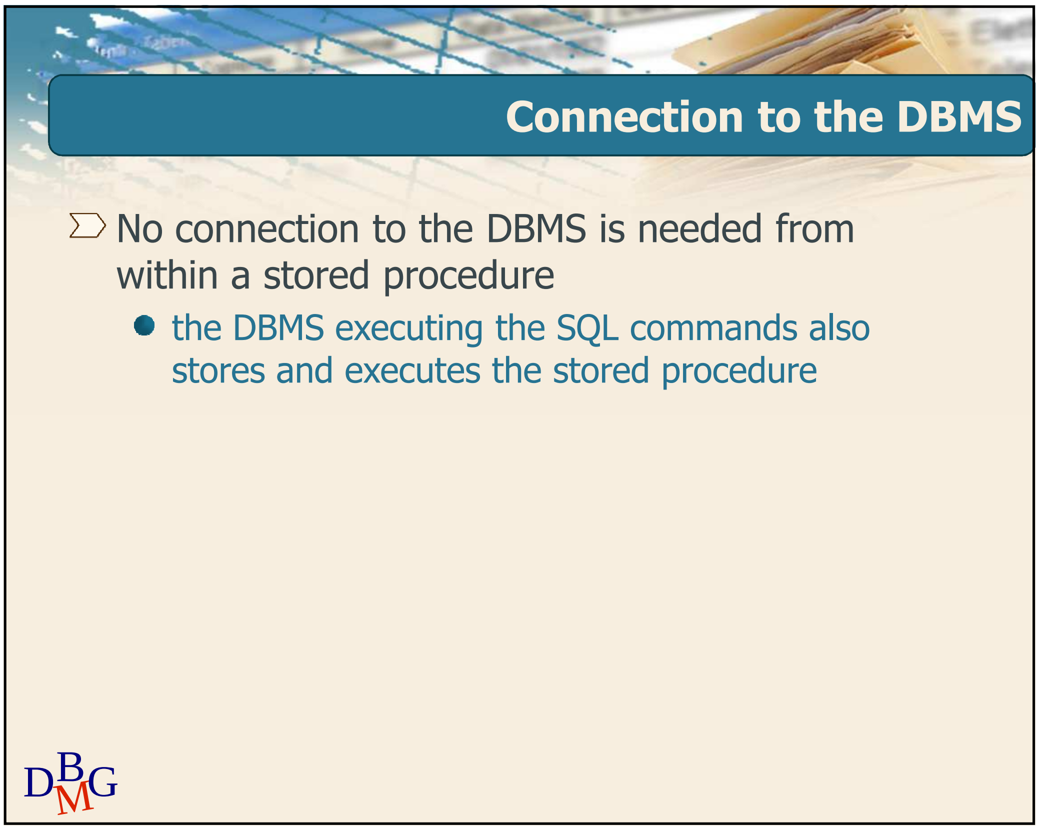
- Stored procedures are integrated in the DBMS
  - server-side approach
- Performance is better compared to embedded SQL and CLI
  - each stored procedure is compiled and optimized *only once*
    - immediately after its definition
    - or when it is invoked for the first time

DBMG

Languages for stored procedures

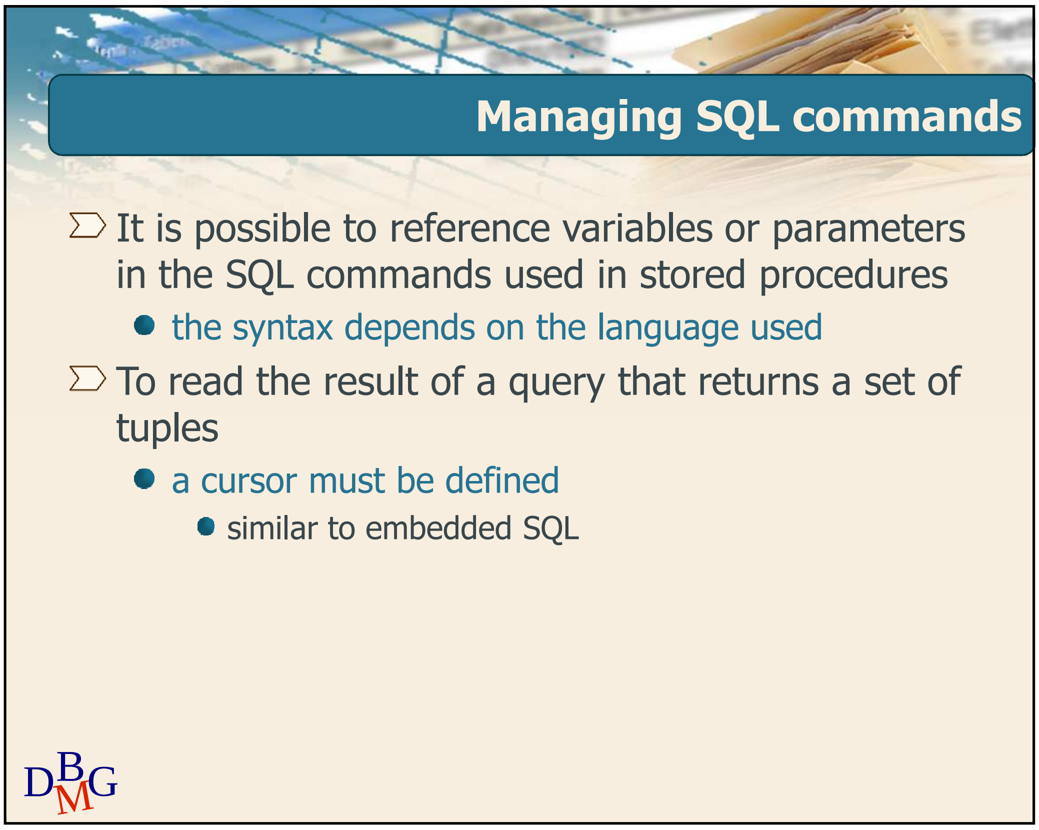
- Different languages are available to define stored procedures
  - PL/SQL
    - Oracle
  - SQL/PL
    - DB2
  - Transact-SQL
    - Microsoft SQL Server
  - PL/pgSQL
    - PostgreSQL

DBMG



Connection to the DBMS

- No connection to the DBMS is needed from within a stored procedure
  - the DBMS executing the SQL commands also stores and executes the stored procedure



Managing SQL commands

- It is possible to reference variables or parameters in the SQL commands used in stored procedures
  - the syntax depends on the language used
- To read the result of a query that returns a set of tuples
  - a cursor must be defined
    - similar to embedded SQL

Stored procedures in Oracle

- Creation of a stored procedure in Oracle

```
CREATE [OR REPLACE] PROCEDURE StoredProcedureName
[(ParameterList)]
IS (SQLCommand | PL/SQL code);
```

- A stored procedure may be associated with
  - a single SQL command
  - a block of code written in PL/SQL

DBMG

SQL for applications

Comparison of alternatives

DBMG

Embedded SQL, CLI and Stored procedures

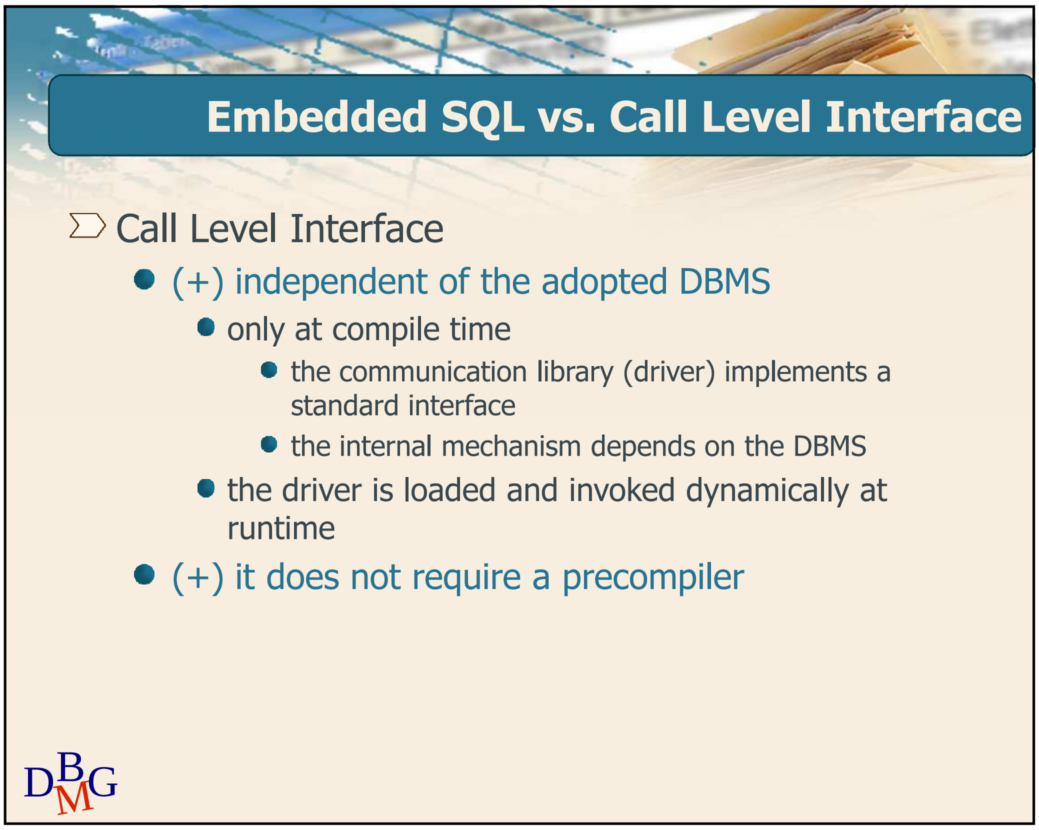
- The techniques proposed for the integration of the SQL language with applications have different features
- There is no winner: no one approach is always better than the others
  - it depends on the type of application
  - it depends on the characteristics of the databases
    - distributed, heterogeneous
- Mixed solutions may be adopted
  - invoking a stored procedure through CLI or embedded SQL

DBMG

Embedded SQL vs. Call Level Interface

- Embedded SQL
  - (+) it precompiles static SQL queries
    - more efficient
  - (-) it depends on the adopted DBMS and operating system
    - due to the presence of a compiler
  - (-) it normally does not allow access to multiple databases at the same time
    - or it is a complex operation

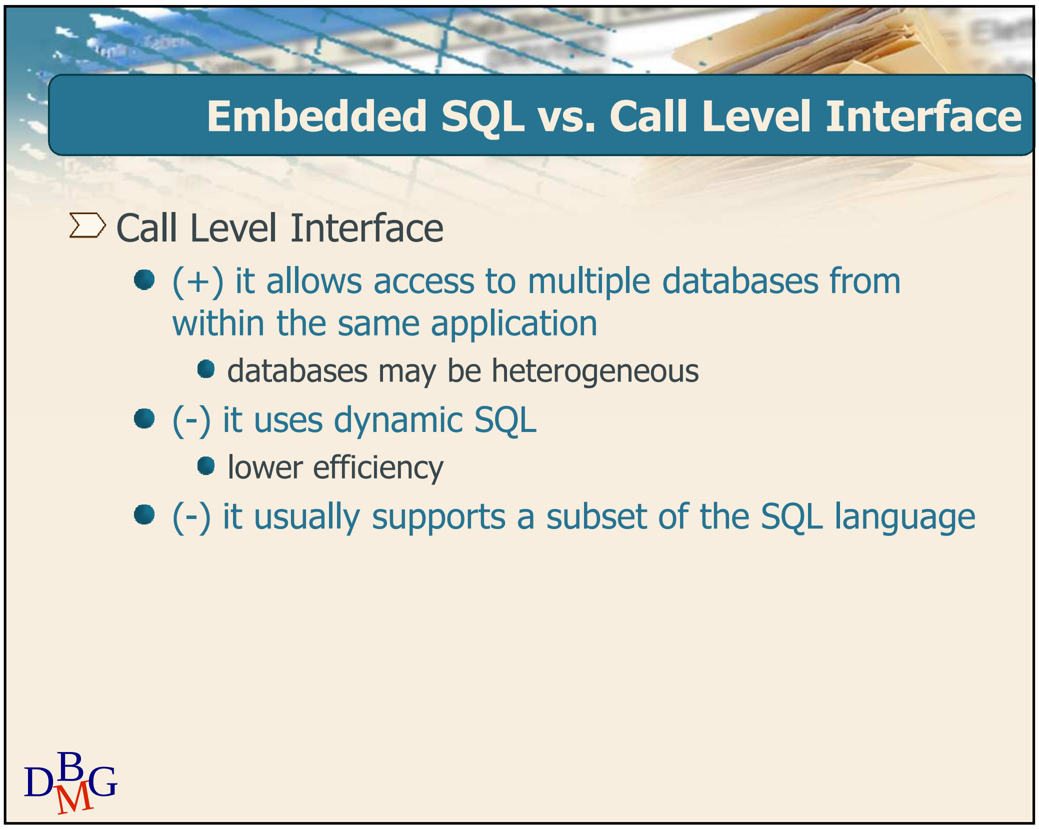
DBMG



Embedded SQL vs. Call Level Interface

- Call Level Interface
  - (+) independent of the adopted DBMS
    - only at compile time
      - the communication library (driver) implements a standard interface
      - the internal mechanism depends on the DBMS
    - the driver is loaded and invoked dynamically at runtime
  - (+) it does not require a precompiler

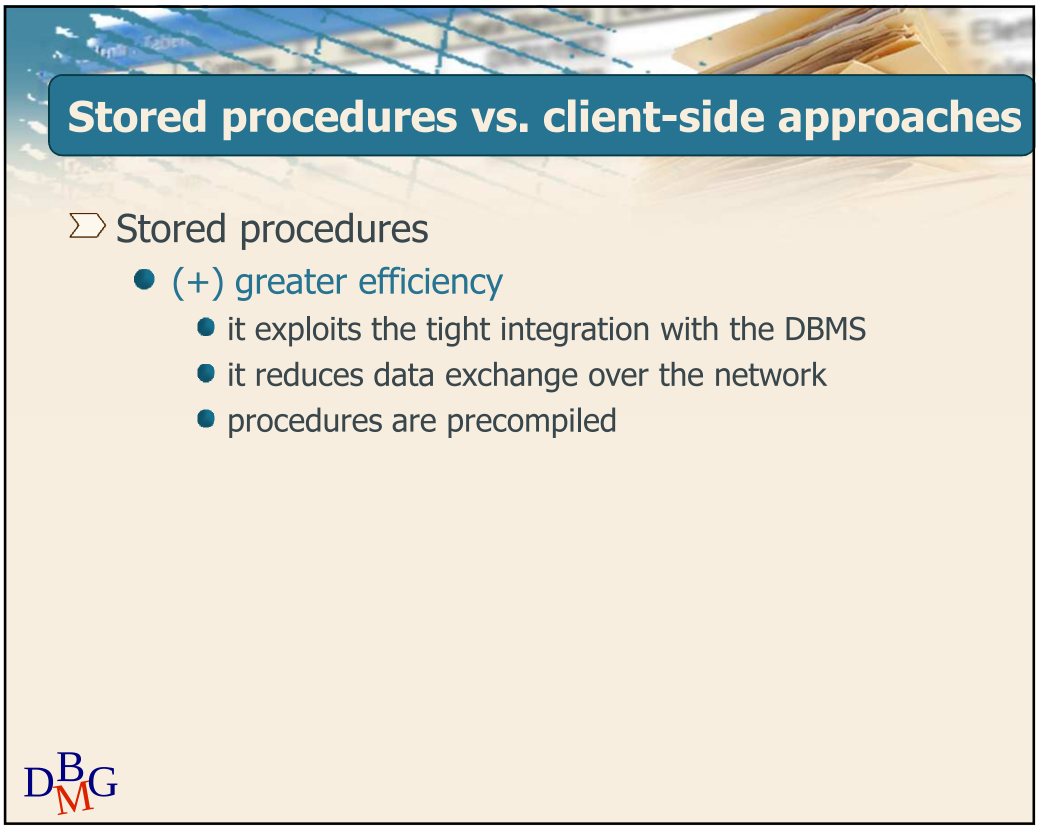
DBMG



Embedded SQL vs. Call Level Interface

- Call Level Interface
  - (+) it allows access to multiple databases from within the same application
    - databases may be heterogeneous
  - (-) it uses dynamic SQL
    - lower efficiency
  - (-) it usually supports a subset of the SQL language

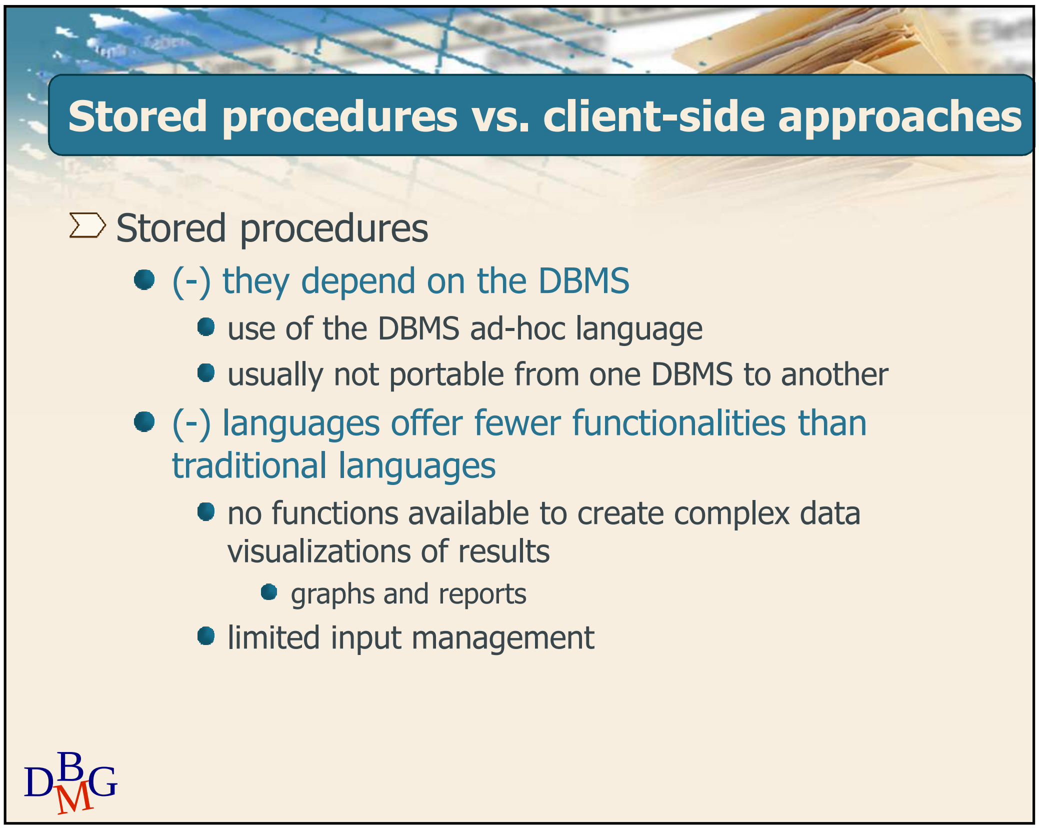
DBMG



Stored procedures vs. client-side approaches

- Stored procedures
  - (+) greater efficiency
    - it exploits the tight integration with the DBMS
    - it reduces data exchange over the network
    - procedures are precompiled

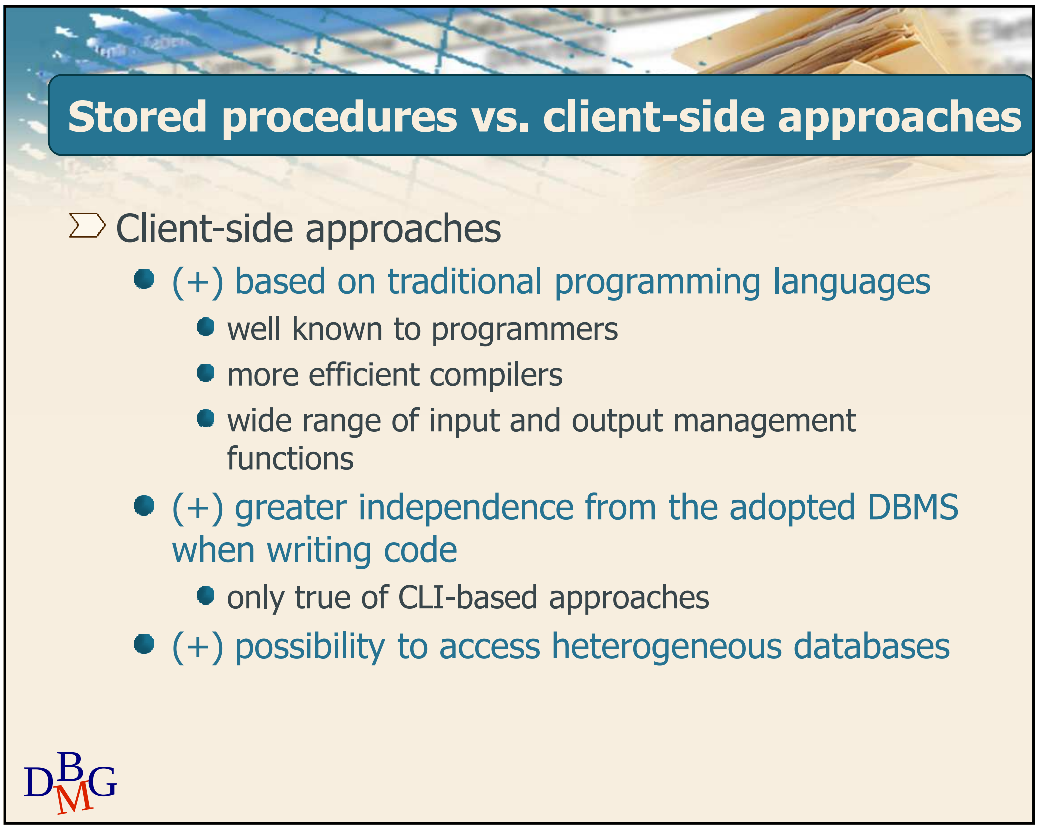
DBMG



Stored procedures vs. client-side approaches

- Stored procedures
  - (-) they depend on the DBMS
    - use of the DBMS ad-hoc language
    - usually not portable from one DBMS to another
  - (-) languages offer fewer functionalities than traditional languages
    - no functions available to create complex data visualizations of results
      - graphs and reports
    - limited input management

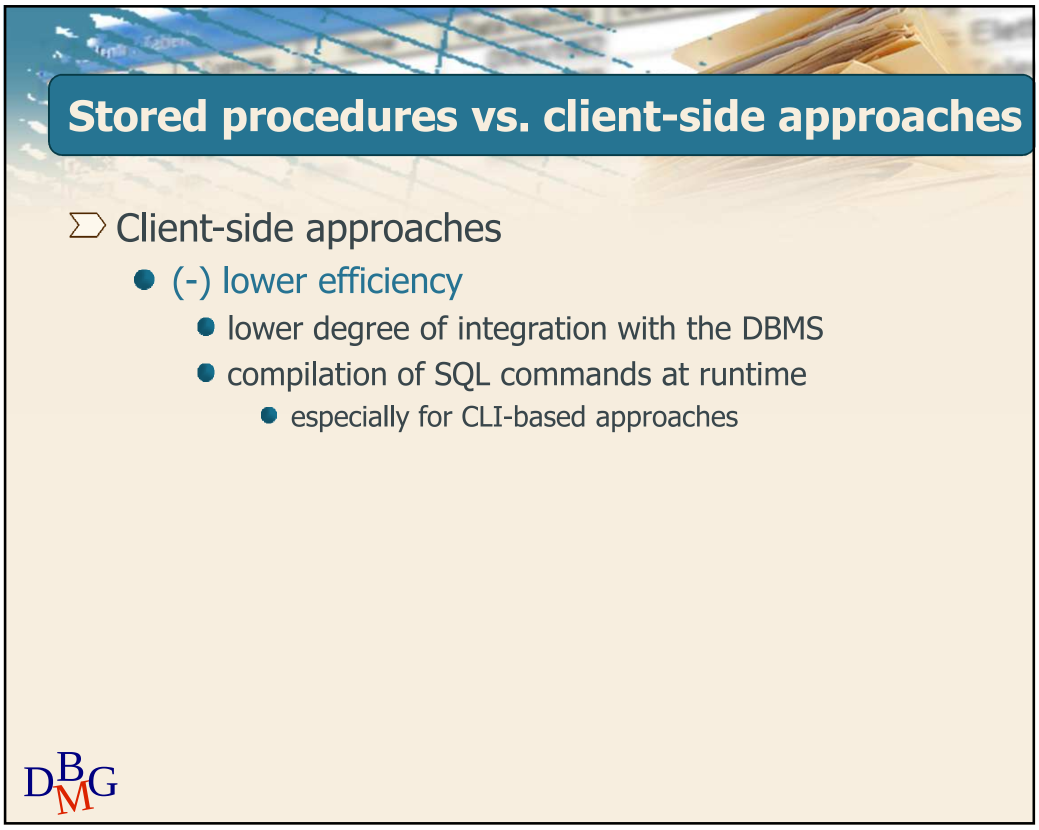
DBMG



Stored procedures vs. client-side approaches

- Client-side approaches
  - (+) based on traditional programming languages
    - well known to programmers
    - more efficient compilers
    - wide range of input and output management functions
  - (+) greater independence from the adopted DBMS when writing code
    - only true of CLI-based approaches
  - (+) possibility to access heterogeneous databases

DBMG



Stored procedures vs. client-side approaches

- Client-side approaches
  - (-) lower efficiency
    - lower degree of integration with the DBMS
    - compilation of SQL commands at runtime
      - especially for CLI-based approaches

DBMG