Part 17: Application Programming I

References:

- Elmasri/Navathe: Fundamentals of Database Systems, 2nd Edition. Section 10.5, "Programming Oracle Applications"
- R. Sunderraman: Oracle Programming — A Primer, Addison-Wesley, 1999. Chapter 3, "Embedded SQL", Chapter 5, "Oracle JDBC".
- Michael Gertz: Oracle/SQL Tutorial, 1999. [http://www.db.cs.ucdavis.edu/teaching/sqltutorial/]
- Oracle8 Application Developer's Guide, Oracle Corporation, 1997, Part No. A58241-01.
- Pro*C/C++ Precompiler Programmer's Guide, Release 8.0, Oracle Corporation, 1997, Part No. A58233-01.
- Kernighan/Ritchie: The C Programming Language, 2nd Edition, Prentice Hall, 1988.
- Harbison/Steele Jr.: C — A Reference Manual, 4th Ed. Prentice Hall, 1995.
- Chamberlin: A Complete Guide to DB2 Universal Database. Morgan Kaufmann, 1998.
- Microsoft ODBC 3.0, Programmers Reference and SDK Guide, Microsoft Press, 1997.
- Microsoft Open Database Connectivity (Part of MSDN Library Visual Studio 6.0)
- Roger E. Sanders: ODBC 3.5 Developer's Guide. McGraw-Hill, 1999.
- SQL Server Books Online, "Building SQL Server Applications".
- Art Taylor: JDBC Developer's Resource, 2nd Edition. Prentice Hall, 1999.

Objectives

After completing this chapter, you should be able to:

- name a few languages/interfaces/tools which can be used for application program development.
- develop programs that use Embedded SQL.

  You should know the steps to translate a C-program with Embedded SQL to an executable program. You should be able to explain how host variables can be used in embedded SQL statements, and when an indicator variable is needed.
- explain the notion of a cursor and the steps involved in using a cursor.
- write application programs using ODBC and JDBC.

Overview

Introduction (1)

- SQL is a database language, but not a programming language:
  - ◇ Powerful queries and updates can be written as short SQL-commands.

    Writing something similar in C would take much more time.
  - ◇ However, one cannot define user interfaces or complex computations on the data in SQL.
  - ◇ SQL is not computationally complete.

    Not every computable function on the database states can be expressed in SQL. Otherwise termination of query evaluation could not be guaranteed.

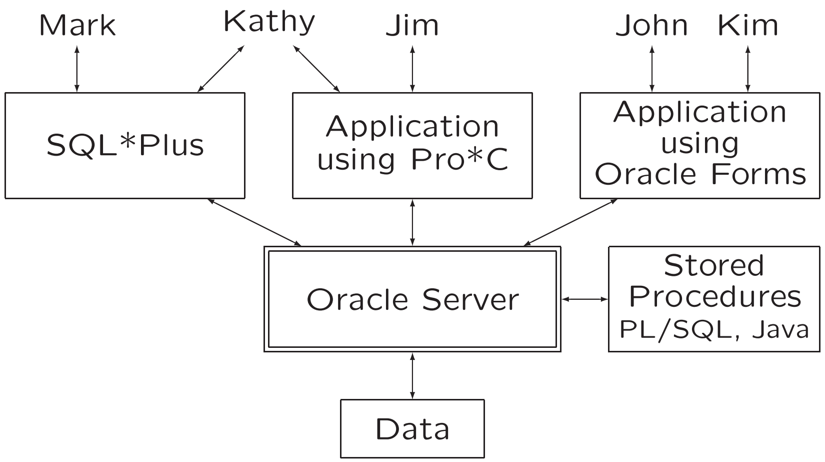
Introduction (2)

- SQL is used directly for ad-hoc queries or one-time updates of the data.
- Repeating tasks have to be supported by application programs written in a programming language.

  These programs do use SQL internally for data access.
- Not all database users know SQL.
- Even if a user knows SQL, it might be easier and faster to use an application program.

  The application program might also display the query result in a better way, or perform additional checks on the input data.

Introduction (3)



Introduction (4)

Some Languages/Tools for Application Programming:

- SQL*Plus Scripts
- C with Embedded SQL (Pro*C)
- C with procedure calls (OCI, ODBC)
- Java with procedure calls (JDBC)
- Form-based applications (Oracle Forms)

  Simple Oracle Forms applications are basically editors for relations. Developed in a graphical enviromentment, without real programming.
- Web Interface: CGI-Program, PL/SQL-Procedures, Java Servlet, HTML with embedded Perl code, . . .

Introduction (5)

- Often one has to work with more than one language (e.g. C and SQL) to develop the application. This leads to several problems:
  - ◇ The interface is often not smooth: E.g. different type systems, "impedance mismatch problem".

    "Impedance mismatch": SQL is a declarative, set-oriented language. Most programming languages are imperative, tuple-oriented.
  - ◇ Only local optimization of single SQL commands.
  - ◇ Query evaluation plans for the SQL statements in the program should be kept between program executions, but programs are external to the DB.

Introduction (6)

- These problems could be avoided with integrated systems consisting of a programming language and a database. Proposed solutions are, e.g.
  - ◇ Persistant programming languages.
  - ◇ Pascal/R: Pascal with the type "relation".
  - ◇ 4GLs: Fourth Generation Languages

    4GL (Fourth Generation Language): GUI + DB + Rules(?)
    Visual development environment, "real code" is written seldom.
  - ◇ Procedures stored in the DB Server.
  - ◇ Object-oriented databases.
  - ◇ Deductive databases.

Making Good Use of SQL

- Quite often, application programs use a relational DBMS only to make objects/records persistent, but do all computation in the programming language.

  I.e. queries retrieve only single rows, and do not perform joins or aggregations.
- Using more powerful SQL commands might
  - ◇ simplify the program, and
  - ◇ significantly improve the performance.

    There is an overhead for executing an SQL statement: It must be sent over the network to the server, and the result sent back. Thus, the fewer SQL statements are needed to solve the task, the better. Also query optimization is little used in this style.

Example Database

STUDENTS

SID	FIRST	LAST	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

EXERCISES

CAT	ENO	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

RESULTS

SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

Overview

1. Introduction and Overview
2. Embedded SQL
3. ODBC
4. JDBC

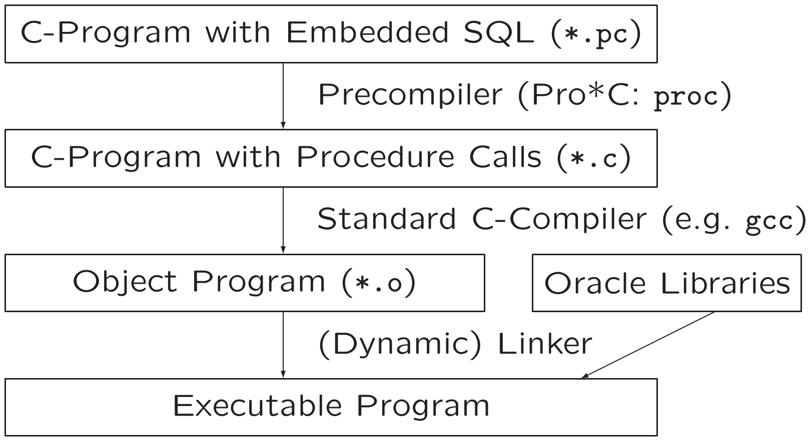
Embedded SQL (1)

- Embedded SQL means that SQL commands are inserted as specially marked statements in programs written in C, C++, Cobol, and other languages.
- Inside SQL, variables of the programming language can be used where SQL allows a constant.
- E.g., insert a new row into the table "`RESULTS`":

```
EXEC SQL INSERT INTO RESULTS(SID,CAT,ENO,POINTS)
         VALUES (:sid, :cat, :eno, :points);
```

- Here `sid` and so on are "C" variables, and these lines can be in between normal "C" statements.

Embedded SQL (2)



A Short "C" Repetition (1)

- The programming language "C" was designed by Dennis Ritchie around 1972 in Bell Labs.

  The first textbook was Kernighan/Ritchie, 1978. C is an ANSI Standard since 1989.

- The traditional first program is:

```
#include <stdio.h>
int main(void)
{
   printf("Hello, world!\n");
      /* Comment: \n means newline */
   return 0;
}
```


A Short "C" Repetition (2)

- The above program declares a procedure "`main`" without parameters. It returns an integer.

  Every program must have a procedure "`main`". This is where the execution starts (there might be some initialization before it, but the user cannot influence that). The return value 0 means that the program terminated successfully. It is not clear that it is really used. If something is wrong, it might be better to call `exit(i)` with a small positive error code `i`.

- The file "`stdio.h`" contains the declaration of the library function "`printf`" used for output.
- The braces "{" and "}" are used as "BEGIN" and "END". In general, C has a very concise syntax.

A Short "C" Repetition (3)

- In C, a variable declaration is written as

  $\langle$Type$\rangle$ $\langle$Variable$\rangle$;

  But e.g. for array types, on the right hand side something like an expression is written, which would give the type on the left.

- E.g., a variable `sid` for integers is declared as:

```
int sid; /* Student ID */
```

- There are integer types of different size, e.g. `short`.

  The type "`short`" (or "`short int`") is typically 16 bit: `-32768..+32767`. The type "`int`" is the natural word size of the machine (today 32 bit is normal, which is more than `NUMERIC(9)`, but earlier, it could be 16 bit). The type "`long`" is at least 32 bit. Integer types can be modified with the prefix "`unsigned`", e.g. "`unsigned short`" has the range `0..65535`.

A Short "C" Repetition (4)

- The type "`char`" is typically used for storing characters, but it is really a byte (small numbers).

  It depends on the hardware whether the number is signed or not. One can use `unsigned char` if this is important (range `0..255`).

- Declaration of an array of 20 characters `a[0]..a[19]`:

```
char a[20];
```

- In C, strings are represented in such character arrays. A null byte is used to mark the string end.

  E.g. `"xyz"` is represented as `a[0]='x'`, `a[1]='y'`, `a[2]='z'`, `a[3]='\0'`. An array of size 20 can contain strings up to only 19 characters.

A Short "C" Repetition (5)

- A value is assigned to a variable in the form:

```
sid = 101;
```

- A conditional statement is written as:

```
if(retcode == 0) /* == means equals */
        printf("Ok!\n");
else
        printf("Error!\n");
```

- If more than one statement is needed in the "if" part or the "else" part, one must use "`{...}`".

- C has no boolean type, but uses `int` instead: `0` is treated as false, everything else counts as true.

A Short "C" Repetition (6)

- One can print an integer e.g. with:

```
printf("The current student ID is: %d\n", sid);
```

  The first argument is the format string. Normal characters in it are literally printed, but "`%d`" is a format element that prints the value of an additional "`int`" argument in decimal notation. Number and type of additional arguments are determined by the format string.

- One can read an integer in decimal notation with

```
ok = scanf("%d", &sid);
```

  "`&sid`" denotes a pointer to `sid`. C has only "call by value", but here one must pass the address of `sid` to `scanf`, because it must set the variable. "`scanf`" returns the number of converted format elements, i.e. here "`1`" would mean "ok". "`scanf`" leaves a line end in the input.

A Short "C" Repetition (7)

- Suppose that `name` is declared as

```
char name[21];
```

- In C, one can assign only single characters with "=", but there is a library function to copy strings:

```
strcpy(name, "Smith");
```

The philosophy is that "=" should correspond to a single machine instruction. One must use `#include <string.h>` to declare this function. The programmer is responsible that there is never a "buffer overflow". If one should assign a string longer than 20 characters, other variables on the stack are overwritten. C as no built-in protection against the violation of array limits. Hackers might be able to execute arbitrary code because of this error.

A Short "C" Repetition (8)

- Strings can be read e.g. with

```
scanf("%s", name);
```

For arrays, the memory address is passed, thus no "&" is needed.

- This command will read only a string consisting of non-blank characters (a word). To read an entire line use "`gets(name);`" or

```
fgets(name, 21, stdin);
name[strlen(name)-1]='\0';
```

- Strings can be printed e.g. with

```
printf("%s", name);
```


Host Variables (1)

- If SQL is embedded in C, then C is the "host language". C-variables which should be used in SQL statements are called "host variables".
- Note that the database has a type system which is quite different from the "C" type system.

  E.g. C has no type "`DATE`", no C type corresponds to "`NUMERIC(30)`".
- In addition, "C" has no notion of null values.
- Even if there is a natural correspondence between an SQL type and a C type, the storage format Oracle uses can differ from the C representation.

Host Variables (2)

- E.g. Oracle stores strings (`VARCHAR`) with the length followed by an array of characters. C uses an array of characters with a null character at the end.

  Also Oracle stores numbers with mantissa and exponent (scientific notation), where the mantissa is stored in a kind of BCD format (4 bits for every digit). But C uses a binary representation.
- So some sort of type / memory format conversion has to take place whenever data values are passed from the database to the program or vice versa.
- The precompiler can do quite a lot, but some work remains for the programmer.

Host Variables (3)

- Oracle has a list of "internal types" (Oracle types) and "external types" (types of C and other host languages) and possible conversions between them.
- Many conversions are done automatically, e.g. from `NUMERIC(`p`)`, p not big, into the C-type "`int`".

  Also `NUMERIC(`p`,`s`)` can be converted to and from the C-type "`float`", although precision may be lost.
- But for `VARCHAR`, one must either declare variables with an unusual C-type corresponding to Oracle's representation, or explicitly state that one wants to use the C convention for marking the string end.

Host Variables (4)

- The precompiler must be able to understand the declaration of the host variables.
- The precompiler does not necessarily understand the full "C" syntax.

  In general it can do its job by looking only at the statements specially marked with the prefix "`EXEC SQL`". It is also not really desirable that the precompiler checks the complete C syntax. Then one would have to write "C" which is acceptable to two compilers.
- Thus, variable declarations relevant to the precompiler must be enclosed in "`EXEC SQL BEGIN DECLARE SECTION`" and "`EXEC SQL END DECLARE SECTION`".

Host Variables (5)

- E.g. the declare section can look as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    int     sid;       /* Student ID */
    varchar first[20]; /* Student first name */
    char    last[21];  /* Student last name */
    EXEC SQL VAR last IS STRING(21);
EXEC SQL END DECLARE SECTION;
```

- "`sid`" is a standard C integer variable, and Oracle can convert it automatically to and from `NUMERIC(`p`)`.
- "`last`" is a standard C string variable.

  Here an explicit type-declaration for the precompiler is required (or the option `CHAR_MAP=STRING`). Note that the max. string length is 20.

Host Variables (6)

- "`varchar first[20]`" is not a standard C data type.
- The precompiler translates it into

```
struct {
    unsigned short len;
    unsigned char arr[20];
} first;
```

- One possibility to read this is by

```
scanf("%s", first.arr);
first.len = strlen(first.arr);
```

  Only strings up to 19 characters can be read since `scanf` places the null byte at the end. If the input is longer, other variables are overwritten! In real projects one needs protection against such "buffer overflows".

Host Variables (7)

- In the same way, Oracle strings can be printed by transforming them first into C strings:

```
first.arr[first.len] = '\0';
printf("%s", first.arr);
```

- A string can be assigned/copied in this way:

```
strcpy(first.arr, "Ann");
first.len = strlen(first.arr);
```

- Space for the additional null byte is needed in each of these operations.

  One can declare the host variable longer than the corresponding database column. E.g. it would be better to declare "`varchar first[21];`".

Host Variables (8)

- The variables in the `DECLARE SECTION` can be global or local.
- The types of these variables must be such that the precompiler can understand them.
- Especially, no non-standard types (defined by the programmer with `typedef`) are allowed.
- Host variables can have the same name as database columns since they are marked in the SQL commands with a ":".

Error Checking (1)

- After every execution of an SQL command, one must check whether an error happened.
- One possibilities for this is to declare a variable

```
char SQLSTATE[6];
```

- As required by the SQL-92 standard, if such a variable is declared, Oracle stores a "Return Code" in it after executing an SQL command.

  The SQL State has five characters, `SQLSTATE[5]` should be set to `0`. The first two characters contain a "class code", e.g. `'00'` means "ok" and `'02'` means that there are no more tuples to be returned by the query. The following three characters can contain a "subclass code".

Error Checking (2)

- Another possibility, only retained for compatibility with the SQL-86 standard, is to declare a variable

```
long SQLCODE;
```

  Here 0 means successful execution, 100 means that there are no further tuples, and negative values denote errors.

- Oracle (but not the standard) also offers a data structure called "SQL Communications Area", containing various information including a return code. This structure "`sqlca`" is declared with

```
EXEC SQL INCLUDE SQLCA;
```

Error Checking (3)

- The component "`sqlca.sqlcode`" contains the return code: E.g. `0`: ok, `1403`: no further tuples.
- The component "`sqlca.sqlerrm.sqlerrmc`" contains the text of an error message (not null terminated), "`sqlca.sqlerrm.sqlerrml`" is its length:

```
printf("%.*s\n", sqlca.sqlerrm.sqlerrml,
    sqlca.sqlerrm.sqlerrmc);
```

- The error message can also be obtained with the procedure `sqlglm`.

Error Checking (4)

- One can tell the precompiler to automatically insert an error check after every "`EXEC SQL`" command:

```
EXEC SQL WHENEVER SQLERROR GOTO ⟨Label⟩;
```

- The `DO` clause allows to execute any C-statement:

```
EXEC SQL WHENEVER SQLERROR DO error();
```

This is an Oracle extension, otherwise `WHENEVER` is part of SQL-86/92.

- One can cancel previous `WHENEVER`-declarations with

```
EXEC SQL WHENEVER SQLERROR CONTINUE;
```

A `WHENEVER` declaration applies to all `EXEC SQL` statements which follow in in the program text, until the next `WHENEVER` declaration.

Example (1)

```
/* Program to enter a new exercise */

#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR user[128];
   VARCHAR pw[32]; /* Password */
   VARCHAR cat[1];
   int     eno;
   int     points;
   VARCHAR topic[42]; /* column: size 40 */
EXEC SQL END DECLARE SECTION;

```


Example (2)

```
/* Procedure called in case of errors: */
void error(const char msg[])
{
   /* Print error message: */
   fprintf(stderr, "Error: %s\n", msg);

   /* Close database connection: */
   EXEC SQL ROLLBACK WORK RELEASE;

   /* Terminate Program: */
   exit(1);
}

```

Example (3)

```
int main(void)
{
    char line[80];

    /* Catch errors: */
    EXEC SQL WHENEVER SQLERROR GOTO error;

    /* Log into Oracle: */
    strcpy(user.arr, "SCOTT");
    user.len = strlen(user.arr);
    strcpy(pw.arr, "TIGER");
    pw.len = strlen(pw.arr);
    EXEC SQL CONNECT :user IDENTIFIED BY :pw;
```


Example (4)

```
    /* Read CAT, ENO of new exercise: */
    printf("Enter data of new exercise:\n");
    printf("Category (H,M,F) and number: ");
    fgets(line, 80, stdin);
    if(line[0]!='H' && line[0]!='M' &&
           line[0]!='F')
        error("Invalid Category");
    cat.arr[0] = line[0];
    cat.len    = 1;
    if(sscanf(line+1, "%d", &eno) != 1)
        error("Invalid number");
```


Example (5)

```
    /* Read topic of new exercise: */
    printf("Topic of the exercise: ");
    fgets((char*) topic.arr, 42, stdin);
    topic.len = strlen(topic.arr) - 1;
           /* -1 removes line end '\n' */

    /* Read maximal number of points: */
    printf("Maximal number of points: ");
    fgets(line, 80, stdin);
    if(sscanf(line, "%d", &points) != 1)
        error("Invalid number");
```


Example (6)

```
    /* Show exercise data: */
    printf("%c %d [%s]: %d points\n"
        cat.arr[0], eno, title.arr, maxpt);

    /* Execute INSERT statement: */
    EXEC SQL INSERT INTO
        EXERCISES(CAT, ENO, TOPIC, MAXPT)
        VALUES(:cat, :eno, :topic, :points);

    /* End transaction, Log off: */
    EXEC SQL COMMIT WORK RELEASE;
```

Example (7)

```
      /* Terminate Program: */
      return(0);

      /* In case of errors: */
      error:
         EXEC SQL WHENEVER SQLERROR CONTINUE;
         printf("Oracle Error: %.*s\n",
            sqlca.sqlerrm.sqlerrml,
            sqlca.sqlerrm.sqlerrmc);
         EXEC SQL ROLLBACK WORK RELEASE;
         exit(2);
   }
```


Practical Hints (1)

- Example programs are available in

  `$ORACLE_HOME/precomp/demo/proc`

- There is also a `makefile` "`demo_proc.mk`" which calls the precompiler, the compiler, and the linker.

  The number of libraries which must be linked to the program is large.

- When the final program is executed, the environment variables need to be correctly set.

  If one executes the program directly, this should be no problem: When e.g. SQL*Plus works, also one's own Oracle application should work. But when it is a cgi-program started by the web server, one must ensure that the environment variables (e.g. `LD_LIBRARY_PATH`) are set.

Practical Hints (2)

- The result of the precompilation is a standard C program with the `EXEC SQL` replaced by initializations of data structures and procedure calls.

  The original `EXEC SQL` statement is preserved in comments.

- Error messages of the C-compiler contain the line number in the "`*.c`" file, not in the "`*.pc`" file.

  That is bad, because the error must be corrected in the "`*.pc`-file. C has a mechanism for avoiding this (`#line`), but Pro*C doesn't use it.

- If the program becomes larger and is split into several modules, one probably wants only one "`*.pc`-module, and otherwise standard C modules.

Simple Queries (1)

- The above example shows how to pass values from the program into the database (e.g. for `INSERT`).
- Now the task is to get values out of the database into program variables.
- If it is known that a query can return at most one tuple, the following format can be used:

```
EXEC SQL SELECT FIRST, LAST
         INTO   :first, :last
         FROM   STUDENTS
         WHERE  SID = :sid;
```

Simple Queries (2)

- It is an error if the `SELECT` yields more than one row.
- E.g. the following query is only safe if `FIRST, LAST` is declared as an alternative key of `STUDENTS`:

```
EXEC SQL SELECT SID
         INTO   :sid
         FROM   STUDENTS
         WHERE  FIRST = :first
         AND    LAST = :last;
```

If it is not declared as key, Oracle will still execute the query without warning as long as there is at most one SID returned. However, when the query should ever return two SIDs, the program will brake (terminate with an Oracle error).

Simple Queries (3)

- After the `SELECT`-query, one must check whether a row was found. It is no error if the query result is empty, but then the `INTO` variables are not set.
- One possibility is to check the status code:

```
if(sqlca.sqlcode == 0)
    ... process the returned data ...
```

- Alternative (conforms to SQL-86/92 standard):

```
EXEC SQL WHENEVER NOT FOUND GOTO empty;
EXEC SQL SELECT ... INTO ...;
... process the returned data ...
empty: ...
```


General Queries (1)

- If the query can yield more than one tuple, the tuples must be read one after the other in a loop.
- In order to do this, one must first declare a "cursor" for the query:

```
EXEC SQL DECLARE c1 CURSOR FOR
         SELECT  CAT, ENO, POINTS
         FROM    RESULTS
         WHERE   SID = :sid;
```

- At this point, the query is not yet executed, and the value of "`:sid`" is not important.

General Queries (2)

- The second step is to open the cursor:

```
EXEC SQL OPEN c1;
```

- At the `OPEN`, the query is evaluated, and the then current value of the query parameter "`:sid`" is used.
- It is possible to close the cursor and open it again with a different value of "`:sid`".

General Queries (3)

- The third step is to read the query result a tuple at a time into the host variables.
- This is done by using "FETCH" in a loop:

```
EXEC SQL WHENEVER NOT FOUND GOTO done;
while(1) { /* 1 is true, i.e. while(forever) */
    EXEC SQL FETCH c1 INTO :cat, :eno, :points;
    ... Process result tuple ...
}
done: ...
```


General Queries (4)

- The GOTO can be avoided by using the DO clause. However, this is part of the standard:

```
EXEC SQL WHENEVER NOT FOUND DO break;
```

The C statement "break" terminates the nearest enclosing loop.

- One can also watch the return code directly, but then the FETCH-statement must be duplicated:

```
EXEC SQL FETCH c1 INTO :cat, :eno, :points;
while(sqlca.sqlcode == 0) {
    ... Process result tuple ...
    EXEC SQL FETCH c1 INTO :cat, :eno, :points;
}
```


General Queries (5)

- "sqlca.sqlcode == 0" works only in Oracle.
- In SQL-86, one would write "SQLCODE == 0".
- In SQL-92, one should use "SQLSTATE":

```
while(SQLSTATE[0] == '0' and SQLSTATE[1] == '0')
```

- The last (fourth) step is to close the cursor:

```
EXEC SQL CLOSE c1;
```

- Open cursors need memory and might retain locks on the data, therefore one should not forget to close the cursor.

Positioned Updates/Deletes

- One can refer to the tuple last FETCHed in UPDATE and DELETE commands:

```
EXEC SQL UPDATE RESULTS SET POINTS = :points
         WHERE CURRENT OF c1;
```

- This is helpful if one has to ask the user or if the new attribute value is computed in C (not in SQL).
- In this case one should add the "FOR UPDATE" clause to the query (this locks the current tuple).

  Of course, this works only with queries where every result tuple is derived from a single source tuple. One cannot use positioned updates with join or aggregation queries.

Null Values (1)

- If a query can yield a null value, one must use two variables: One for the data value, and one for indicating whether the value is null.
- Such variables are called "Indicator-Variables".

  They must be declared as "`short`" in Oracle.

- The indicator variable will be set to `-1` if a null value was returned, and `0` if a normal value was returned.

  Oracle uses other codes besides these two, but this does not agree with the standard.

Null Values (2)

- E.g. consider a cursor to fetch student data:

```
EXEC SQL DECLARE stud CURSOR FOR
         SELECT  FIRST, LAST, EMAIL
         FROM    STUDENTS;
```

- An indicator-variable can be attached to any variable in an SQL statement, e.g.

```
EXEC SQL FETCH stud INTO :first, :last,
                        :email INDICATOR :ind;
```

- In Oracle, the keyword "`INDICATOR`" is not needed.

Null Values (3)

- It is an error if a column should be null and one tries to `FETCH` it into a variable without indicator.
- In the above example, the columns `FIRST` and `LAST` are declared `NOT NULL`, therefore no indicator variable is needed when they are fetched.
- Do not forget that aggregation functions (except `COUNT`) can return a null value if their input is empty!
- Indicator variables can e.g. also be used in an `INSERT` statement to set columns to null.

Using Arrays (1)

- It is not very efficient to exchange a large number of tuples one by one between database and application program.
- Oracle allows to use arrays as host variables for exchanging larger chunks of data in one step.
- However, this is not part of the SQL-92 standard.
- The number of tuples returned can be found in `sqlca.sqlerrd[2]`.

  This counts tuples fetched until now after cursor was opened.

Using Arrays (2)

- When inserting an array of values into a relation, one must specify the number of values as

  `FOR :n INSERT INTO ...`

- In one application, using arrays did significantly improve the performance of the program.
- Oracle also supports records as host variables.

Dynamic SQL (1)

- Above, table and column names were already known when the program was written, only some constants (data values) were not known until runtime.
- In this case, the precompiler can check the existence of the tables and columns.

  Oracle Pro*C has an option for this. It is not the default. One must specify a username and password already during the precompiler run. However, at least the SQL syntax is checked.

- In some systems (e.g. DB2), the queries are already optimized and a "query evaluation plan" (program for executing the query) is stored in the database.

Dynamic SQL (2)

- It is possible to compose SQL commands at runtime in string variables and then to submit the string to the database for execution.

  E.g. SQL interpreters like SQL*Plus work this way, because tables and columns depend on the input from the user. Also a database interface library like ODBC is typically implemented with dynamic SQL.

- If the SQL command is not a query, it can be executed in this way:

  `EXEC SQL EXECUTE IMMEDIATE :sql_cmd;`

  In this form, the SQL command cannot contain host variables, but that is no restriction, since one can drectly insert their values into the command string.

Dynamic SQL (3)

- One of the problems of Dynamic SQL is that the command is compiled (into a QEP) every time it is executed. Query optimization needs runtime.

  Normally, it is compiled at precompilation time or only once when it is executed for the first time. Oracle does not compute QEPs at precompilation time, but it cashes them for the most recently executed queries. This reduces the problems, but only when one uses host variables in the strings. If one simply copies data values in the SQL command strings, the SQL commands are always different.

- If an SQL statement is executed several times with different parameter values, one can compile it with `PREPARE` and then use "`EXECUTE...USING` ⟨`Variables`⟩".

Dynamic SQL (4)

- Dynamic queries are quite complicated, because the result columns are not known until runtime.
- In this case, an "SQL Descriptor Area" (SQLDA) is used to get information about the result columns.

  Oracle uses the function `sqlald` followed by several calls to `malloc`, the SQL-92 standard contains a command `ALLOCATE DESCRIPTOR`.
- The `DESCRIBE` statement stores the number, names, and datatypes of the result columns of a dynamic query into the SQLDA.

  The SQLDA also contains place for pointers to variables which will contain the retrieved data elements (the "`FETCH`" host variables).

Dynamic SQL (5)

- The sequence of steps is:
  - ◇ An SQLDA is allocated (with `sqlald/malloc` in Oracle and `ALLOCATE DESCRIPTOR` in SQL-92).
  - ◇ The string with the query is composed.
  - ◇ `PREPARE` is used to compile the SQL query.
  - ◇ `OPEN` is used to execute the query and open a cursor for the query result.
  - ◇ `DESCRIBE` is used to fill the `SQLDA`.
  - ◇ Variables for the query result are allocated.
  - ◇ `FETCH` is called in a loop to get the result tuples.

Portability Issues (1)

- The SQL standard is mainly for Embedded SQL, so that application programs become portable.

  For ad-hoc queries, a standard is not so important.
- The CASCADE project at the university of Pittsburgh was ported from Oracle Pro*C to Microsoft ESQL (running on MS SQL Server).

  ESQL is no longer supported by Microsoft. Microsoft wants everbody to use ODBC, ADO, or OLE DB. CASCADE is a collaborative authoring system developed by Michael Spring and his research group.
- In general, the port was more work than expected.

Portability Issues (2)

- One problem was that the SQL commands contained non-standard constructs, only supported by Oracle, e.g.
  - ◇ the strange outer join syntax,
  - ◇ the "`START WITH ... CONNECT BY ...`" construct.

    This is intended for processing tree-structured data. In particular, Oracle supports the transitive closure in this way.
- If one anticipates that a port will be necessary, one can in part avoid this. But the price is: (1) less work done in SQL, more in C, (2) more complicated SQL statements, (3) non-optimal performance.

Portability Issues (3)

- Another problem was that the type conversion with the host variables was different:
  - ◇ MS ESQL does not have the `VARCHAR` type for host variables (with length and character array).
  - ◇ It might be more portable to use `char x[`n`]`, but then `x` will be filled with spaces until the length n.

    One can have standard C strings in MS ESQL, but only if the host variable is a character pointer (pointing to a memory buffer).
  - ◇ Elements of the `DATE` type were converted very differently into character arrays.

    It was already lucky that the `DATE` type is supported.

Portability Issues (4)

- Using the return codes for executed SQL commands also caused problems:
  - ◇ One must use `SQLCODE` in MS ESQL (Oracle's `sqlca` is not contained in the Standard).
  - ◇ "`WHENEVER NOT FOUND DO break;`" does not work in MS ESQL (it is not contained in SQL-92).

    MS ESQL allows "`WHENEVER ... CALL` ⟨`Procedure`⟩", but also that is not contained in the standard (and is not supported in Oracle).
- Also the data dictionary is structured differently in Oracle and MS SQL Server.

Overview

1. Introduction and Overview
2. Embedded SQL
3. ODBC
4. JDBC

Introduction (1)

- ODBC (Open Database Connectivity) is an interface for application programs (API) to send SQL commands to a DBMS and get back the results.
- ODBC was developed by Microsoft.

  Microsoft ODBC website: [http://www.microsoft.com/data/odbc/]
- An SQL Call Level Interface (CLI), which is a subset of ODBC 3.x, was standardized by X/OPEN and ISO/IEC in 1995.
- Since only procedure calls are used, no precompiler is needed.

Introduction (2)

- The goal of ODBC is to make application programs portable: ODBC drivers are available for many different DBMS.

  Like printer drivers implement a common interface for printers, so that word processing software can print on any printer, an ODBC-application can work with any DBMS which has an ODBC driver.

- ODBC drivers are also available for non-relational data sources, e.g. spreadsheets and flat files.

  These drivers must implement some SQL themselves.

Introduction (3)

- ODBC does not make the differences between systems completely invisible, but it
  - ◇ guarantees a minimal set of SQL constructs,

    Only few constructs are guaranteed, e.g. no subqueries, no aggregations, no explicit tuple variables, and no `LIKE`. But the minimal syntax does contain joins and `ORDER BY`.

  - ◇ has many functions for checking which features a DBMS supports,
  - ◇ defines escape sequences in SQL commands that the driver replaces by a DBMS-specific syntax.

    E.g. for date etc. constants, for the `ESCAPE` clause of the `LIKE` predicate, for outer joins, and for procedure calls.

Introduction (4)

- ODBC uses dynamic SQL: SQL queries are passed as strings to the DBMS. Thus, queries can be constructed at runtime.

  When not using a precompiler, there is basically no alternative to this. It might a bit slower than static embedded SQL, where already the precompiler can generate a query evaluation plan (in DB2). Also, a precompiler can check SQL syntax for static SQL, whereas here errors are only detected when the faulty statements are executed.

- It is possible with ODBC to optimize a statement only once and then execute it for different parameter values (improves the performance).

Introduction (5)

- ODBC is naturally very common on Windows platforms: There Oracle and DB2 come also with an ODBC driver.

  Of course, SQL Server has an ODBC driver.

- But e.g. ODBC drivers for Oracle on Solaris are available from a number of third-party vendors.

  E.g. Merant, Intersolv. Microsoft states that it works together with Visigenic Software to port its SDK to the Apple Macintosh and a variety of UNIX platforms. In general, if one has a precompiler for embedded SQL that supports dynamic SQL, one could write an ODBC interface oneself. But depending on how much of ODBC one really uses, that could be a lot of work.

Other APIs

- Microsoft has developed newer object-oriented DB interfaces [http://www.microsoft.com/data/]:
  - ◇ ADO: Microsoft AcitiveX Data Objects: Easier, language-neutral interface to OLE DB.
  - ◇ OLE DB: COM API for accessing data.
- Although both are intended to be open interfaces and support other data sources than Microsoft SQL Server, there is no independent standard for them (whereas ODBC is compatible with SQL CLI).

  However, there is an "OLE DB Provider" which can use an ODBC driver to access data. Thus, it is upward compatible to ODBC.

Using ODBC (1)

- In order to use ODBC, the following is needed:
  - ◇ An ODBC driver for the DBMS.

    Most DBMS for Windows come with such a driver.
  - ◇ Header files (`.h`) that define constants and types needed for calling the ODBC procedures.

    These are part of the Microsoft ODBC Software Development Kit (SDK) and should come e.g. with Microsoft Visual C++.
  - ◇ The driver manager, a library (DLL), to which the ODBC application is linked.

    It is responsible for loading the correct driver and forwarding the ODBC function calls to the driver. It should already come with the Windows operating system.

Using ODBC (2)

- Under Windows, an ODBC data source for the DB should be configured in the control panel.

  This is simply a name for the database connection information. One must specify e.g. what ODBC driver should be used and to which database it should connect. Username and password can be defined either here or in the application program. There are also many options that can be set, e.g. whether there should be an automatic `COMMIT` after every update.
  In this way, it is possible to distribute application programs in binary form that can work e.g. with an Oracle database or an Access database without any change.
- Alternatively, the database connection information can be hardcoded in the application program.

ODBC: Handles (1)

- "Handles" are opaque pointers to data structures in the driver. The data structures are only accessed via ODBC functions. There are 4 types of handles:
  - ◇ Environment Handles (e.g. ODBC version).
  - ◇ DB Connection Handles (filled by logging into a DBMS, necessary to submit SQL commands)
  - ◇ Statement Handles (for SQL statements in execution, contains e.g. a cursor)
  - ◇ Descriptor Handles (e.g. for data about result columns)

ODBC: Handles (2)

- The data structures are allocated with the function

  `SQLAllocHandle(HandleType, HandleIn, HandleOut)`

- `HandleType` is one of the constants `SQL_HANDLE_ENV`, `SQL_HANDLE_DBC`, `SQL_HANDLE_STMT`, `SQL_HANDLE_DESC`.
- `HandleIn` (of type `SQLHANDLE`) defines a context:
  - ◇ For environment handles: `SQL_NULL_HANDLE`.
  - ◇ For connection handles: an environment handle.
  - ◇ For statement handles: a connection handle.

    Descriptor handles are seldom explicitly allocated.
- `HandleOut` must point to a variable of type `SQLHANDLE`.

ODBC: String Parameters

- Strings are passed to ODBC functions with two arguments: a character array and the length.
- The special constant `SQL_NTS` for the length means that the character array is a null-terminated string.
- E.g. for opening a database connection, one must specify a connection handle, the data source name, user and password (if not set in the data source):

```
SQLConnect(conn,
           (SQLCHAR*) "MyOracleDB", SQL_NTS,
           (SQLCHAR*) "SCOTT", SQL_NTS,
           (SQLCHAR*) "TIGER", SQL_NTS);
```


ODBC: Return Codes (1)

- Every ODBC function returns a result code (of type `SQLRETURN`). It should normally be `SQL_SUCCESS`.
- The value `SQL_SUCCESS_WITH_INFO` means that the request was executed, but a warning was generated.

  It is good programming practice to check these result codes. This was not done in most program examples shown here for space reasons.
- If an ODBC call fails, one can get more information with `SQLGetDiagRec`. E.g. if the `SQLConnect` above fails, one could execute the code on the next page.

  With every handle, a sequence of zero or more diagnostic records is associated (error message stack of the last command for this handle).

ODBC: Return Codes (2)

```
SQLCHAR sqlstate[6], msg[256];
SQLINTEGER native_err;
SQLSMALLINT i = 1, len;
while(SQLGetDiagRec(SQL_HANDLE_DBC, conn,
              i, sqlstate, &native_err,
              msg, (SQLSMALLINT)256, &len)
        == SQL_SUCCESS) {
    printf("Error %d ", (int)native_err);
    sqlstate[5] = '\0';
    printf("(SQLSTATE = %s):\n", sqlstate);
    printf("    %.*s\n", (int)len, msg);
    i = i + 1;
}
```

ODBC: Executing SQL (1)

- The easiest way to execute an SQL statement is to pass the string to `SQLExecDirect`:

```
SQLExecDirect(stmt,
    (SQLCHAR*) "SELECT EMAIL, SID FROM STUDENTS",
    SQL_NTS)
```

- Here, `stmt` is a statement handle that was allocated with `SQLAllocHandle`.

  In particular, it contains a link to the connection handle that was specified in the call to `SQLAllocHandle`.

- `SQLExecDirect` creates a cursor in the statement handle that is positioned before the first result row.

ODBC: Executing SQL (2)

- In order to fetch result rows, one must bind C variables to columns. ODBC can do a type conversion, one only has to specify the type of the C variable.

```
SQLCHAR email[80]; /* unsigned char */
SQLINTEGER ind;    /* long int */
SQLBindCol(stmt, 1, SQL_C_CHAR,
    (SQLPOINTER) email, (SQLINTEGER) 80, &ind);
```

  The parameters of `SQLBindCol` are (1) a statement handle, (2) the output column number, (3) an identification of the C type of the variable to be bound, (4) a pointer to the output variable, (5) its size if it is an array, (6) a pointer to a length and indicator variable.

ODBC: Executing SQL (3)

- An integer variable can be bound to the numeric column `SID` (the second result column) as follows:

```
SQLINTEGER sid;   /* (signed) long int */
SQLBindCol(stmt, 2, SQL_C_SLONG,
    (SQLPOINTER) &sid, (SQLINTEGER) 0, &ind);
```

- The buffer length is not needed, since the variable is not an array. Thus "(`SQLINTEGER`) 0" is passed.

  `SID` can never be null, thus one could also use "(`SQLPOINTER`) 0" for the length/indicator variable.

- The ODBC reference contains a list of C types (including structures for `DATE` values) and their codes.

ODBC: Executing SQL (4)

- The following call retrieves the next result row and stores it in the C variables bound to the columns:

```
SQLFetch(stmt);
```

- `SQL_NO_DATA` is returned at the end.
- The value stored in the array `email` will be null-terminated. The variable `ind` is set to the string length of the result or `SQL_NULL_DATA` for a null value.

  If the buffer `email` should be too small, `ind` still contains the length of the complete result string, although the value in `email` is truncated. Even in this case, `email` is null-terminated. The return code is then `SQL_SUCCESS_WITH_INFO`. For long object types, the total length is difficult to determine, in this case `SQL_NO_TOTAL` is stored in `ind`.

ODBC: Parameters (1)

- The parameter marker "?" can be used to introduce placeholders in an SQL statement, for which later values of program variables will be inserted:

```
SQLPrepare(stmt, (SQLCHAR*)
  "SELECT FIRST,LAST FROM STUDENTS WHERE SID = ?",
  SQL_NTS)
```

- This corresponds to using ":⟨Variable⟩" in Embedded SQL.
- Parameters are especially useful if an SQL statement must be executed several times with different parameter values (it is optimized only once).

ODBC: Parameters (2)

- However, even for a one-time execution, parameters can be useful: One can avoid in this way to construct an SQL statement at runtime.

  E.g. if one is not careful, the generated SQL statement will give a syntax error when a string variable contains a quotation mark '.

- Of course, an SQL statement can have more than one parameter "?". The parameters are identified by position (1st, 2nd, etc.).

  Parameters are not permitted in the SELECT-list. It is also not possible that both sides of a comparison operator are a parameter: Even when the parameter is not yet bound to a variable, it must be known whether the comparison is numeric or lexicographic.

ODBC: Parameters (3)

- SQLBindParameter is used to bind C variables to parameters in SQL statements.
- One must specify the type of the C variable and the SQL type to which is should be converted:

```
SQLBindParameter(stmt, 1, SQL_PARAM_INPUT,
    SQL_C_SLONG, SQL_NUMERIC,3,0,/*NUMERIC(3,0)*/
    &sid, 0, NULL)
```

- All parameters in normal SQL statements are input parameters (SQL_PARAM_INPUT).

  However, when stored procedures (executed in the DBMS server) are called, output and input/output parameters are supported.

ODBC: Parameters (4)

- E.g. if the statement had a second parameter of type VARCHAR(20) one could bind this as follows:

```
SQLCHAR name[20]; /* unsigned char */
SQLINTEGER len = SQL_NTS;
SQLBindParameter(stmt, 2, SQL_PARAM_INPUT,
    SQL_C_CHAR, SQL_VARCHAR,20,0,/*VARCHAR(20)*/
    name, 0, &len)
```

- The prepared statement is executed with

```
SQLExecute(stmt);
```

  Only at this point, the parameter variables must contain values.

ODBC Example (1)

```
#include <stdio.h>
#include <windows.h>
#include <Sql.h>
#include <Sqlext.h>
#include <Sqltypes.h>

int main(void)
{
    SQLRETURN rc;  /* Return Code */
    SQLHENV env;   /* Environment Handle */
    SQLHDBC conn;  /* DB Connection Handle*/
    SQLHSTMT stmt; /* Statement Handle */
```


ODBC Example (2)

```
    SQLINTEGER prod_id;
    SQLINTEGER id_null;
    SQLCHAR prod_name[50];
    SQLINTEGER name_null;

    /* Allocate Environment Handle: */
    SQLAllocHandle(SQL_HANDLE_ENV,
          SQL_NULL_HANDLE, &env);
    SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION,
          (SQLPOINTER) SQL_OV_ODBC3,
          (SQLINTEGER) 0);
```


ODBC Example (3)

```
    /* Allocate Connection Handle: */
    SQLAllocHandle(SQL_HANDLE_DBC,
          env, &conn);

    /* Connect to Database: */
    rc = SQLConnect(conn,
          (SQLCHAR*) "Northwind", SQL_NTS,
          (SQLCHAR*) "", SQL_NTS, /* User */
          (SQLCHAR*) "", SQL_NTS); /* Pwd */
    if(rc != SQL_SUCCESS
          && rc != SQL_SUCCESS_WITH_INFO)
          ...
```


ODBC Example (4)

```
    /* Allocate Statement Handle: */
    SQLAllocHandle(SQL_HANDLE_STMT,
          conn, &stmt);

    /* Execute a Query: */
    rc = SQLExecDirect(stmt, (SQLCHAR*)
          "SELECT ProductID, ProductName "
          "FROM Products ORDER BY 1",
                SQL_NTS);
    if(rc != SQL_SUCCESS)
          ...
```

ODBC Example (5)

```
    /* Bind Variables to Output Columns: */
    SQLBindCol(stmt, 1, SQL_C_SLONG,
          (SQLPOINTER) &prod_id,0,
          &id_null);
    SQLBindCol(stmt, 2, SQL_C_CHAR,
          (SQLPOINTER) &prod_name,
          (SQLINTEGER) sizeof(prod_name),
          &name_null);

    /* Print Query Result: */
    rc = SQLFetch(stmt); /* Get first row */
    if(rc == SQL_NO_DATA)
          printf("Query result is empty\n");
```


ODBC Example (6)

```
    else {
       while(rc == SQL_SUCCESS ||
             rc == SQL_SUCCESS_WITH_INFO) {
          printf("%6d ", prod_id);
          if(name_null == SQL_NULL_DATA)
             printf("(null)\n");
          else
             printf("%s\n", prod_name);
          rc = SQLFetch(stmt);
       }
    }
```


ODBC Example (7)

```
    /* Free Handles, Close DB Connection: */
    SQLFreeHandle(SQL_HANDLE_STMT, stmt);
    SQLDisconnect(conn);
    SQLFreeHandle(SQL_HANDLE_DBC, conn);
    SQLFreeHandle(SQL_HANDLE_ENV, env);

    /* We are done: */
    printf("Query successfully executed\n");
    return(0);
}
```


Overview

1. Introduction and Overview
2. Embedded SQL
3. ODBC
4. JDBC

Introduction (1)

- JDBC is an API for accessing databases from Java Programs.
- It based on the SQL CLI (ODBC), but adds a nice object-oriented interface.
- See [http://java.sun.com/products/jdbc/].
- An alternative is SQLJ (Embedded SQL in Java).
- Thanks to Kevin Ho for the example and to Jan Grau for many suggestions for improvements.

  I slightly modified the example and may have introduced errors.

Introduction (2)

- There are currently four different ways to access a database via JDBC from a Java program:
  - ◇ JDBC-ODBC Bridge: An existing ODBC driver is used.

    Problem: The ODBC driver must be installed on every client. Also not very efficient.
  - ◇ Native-API Driver: JDBC calls are translated to the native API on the client machine.

    Then the DBMS client software must be installed on every client. The network connection to the server is now done entirely by the existing DBMS software. The Oracle-OCI JDBC driver is of this type.

Introduction (3)

- JDBC driver types, continued:
  - ◇ JDBC-Net Driver: JDBC calls are converted to a DBMS-independent network protocol.

    A program on the server machine (middleware server) translates this to calls of the DBMS.
  - ◇ Native Protocol Java Driver: Converts JDBC calls to the network protocol of the chosen DBMS.

    This type usually gives fast access to the database from Java programs. The Oracle Thin JDBC driver is of this type. The Oracle JDBC drivers (and example programs) are in `$ORACLE_HOME/jdbc`. OTN members (membership is free) can also download the drivers from [http://technet.oracle.com/software/].

Introduction (4)

- Important classes are:
  - ◇ `DriverManager`: Provides access to JDBC.
  - ◇ `Connection`: Result of logging into a DB.
  - ◇ `Statement`: SQL Statement (e.g. query, update).

    Subclass `PreparedStatement`: Statements with parameters.
  - ◇ `ResultSet`: Result of a query.
- When one has a `Connection` object, one can call its method `createStatement()` to create a `Statement` object. This is needed for executing SQL statements.

Introduction (5)

- The `Statement` object then has the methods
  - ◇ `executeQuery(String)` to execute an SQL query. It returns a `ResultSet` object.
  - ◇ `executeUpdate(String)` to execute an SQL statement that is not a query.

    It returns the number of rows affected by the update.
- For a `ResultSet` object, the method `next` moves the cursor to the next row.

  It must be called also before the first row. It returns `false` if there are no more rows. A `ResultSet` object must be closed (with method `close()` before it can be used for another query.

Introduction (6)

- Once the cursor is positioned on a row with `next`, the `get...`-methods of the `ResultSet` object can be used to retrieve a column value.
  - ◇ They take the column number (1, 2, ...) or name as argument.
  - ◇ One uses the method `getString`, `getInt`, etc. depending on the result type.

    These methods automatically perform type conversions if needed. E.g. one can use `getString` even when the result column is numeric.

Introduction (7)

- The class `ResultSet` has a method `getMetaData()` that returns an object of the type `ResultSetMetaData`.
- This has e.g. the following methods:
  - ◇ `getColumnCount()`: Number of columns in answer.
  - ◇ `getColumnName(int col)`:
    Name of column at position `col` (1, 2, ...).
  - ◇ `getColumnDisplaySize(int col)`:
    Normal maximum width in characters.
  - ◇ `getColumnType(int col)`: Column datatype.

    See list in java.sql.Types. Alternative `getColumnTypeName`.

Introduction (8)

- If a statement must be executed several times with different values, one uses the method

  `prepareStatement(String)`

  of the class `Connection` to create an object of the class `PreparedStatement`.
- This class has methods `setString`, `setInt`, etc. for setting the parameters.
- Then one can call `executeQuery` or `executeUpdate` (without parameter).

Introduction (9)

- Common steps to establish a database connection:
  - ◇ Load a driver with `Class.forName(⟨drivername⟩)`.

    This driver has to be installed in the `ext` directory of the Java installation first. `Class.forName` throws a `ClassNotFoundException` if the driver was not properly installed.
  - ◇ Get a `Connection`-object by calling `DriverManager.getConnection(...)`

    There are three different versions of this method:
    (1) `getConnection(⟨url⟩)`,
    (2) `getConnection(⟨url⟩, ⟨info⟩)`,
    (3) `getConnection(⟨url⟩, ⟨user⟩, ⟨password⟩)`.
    `⟨url⟩`) is a database URL of the type `jdbc:subprotocol:subname`, where `subprotocol` is e.g. `oracle`, and `subname` is driver-specific. `⟨info⟩`) is a string of attribute/value pairs (e.g. user, password).

JDBC Example (1)

```
import java.sql.*;
import java.net.URL;

class JdbcExample
{

public static void main (String args [])
{
   String driver =
      "oracle.jdbc.driver.OracleDriver";
   String url = "jdbc:oracle:thin:" +
      "@vsam.sis.pitt.edu:1521:sis";
```


JDBC Example (2)

```
   String username = "scott";
   String password = "tiger";
   String query = "SELECT ENAME FROM EMP";
   String ename;

   try {
      // Load driver:
      Class.forName(driver);

      // Get connection:
      Connection con =
         DriverManager.getConnection
            (url, username, password);
```


JDBC Example (3)

```
      // Execute statement:
      Statement stm = con.createStatement();
      ResultSet rs =
         stm.executeQuery(query);

      // Print results:
      while (rs.next()) {
         ename = rs.getString(1);
         System.out.println(ename);
      }
```

JDBC Example (4)

```
            rs.close();
            stm.close();
            con.close();
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```