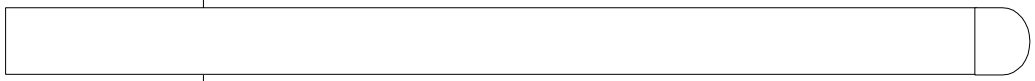
# SQL da programma



# Il problema

- Come abbiamo visto, SQL permette in modo dichiarativo di accedere i dati memorizzati in una base di dati
- il suo potere espressivo è però limitato
- questo significa che non tutte le elaborazioni che possiamo volere applicare ai dati possono essere espresse in SQL

# Il problema - Esempio

- Supponiamo di avere una base di dati avente il seguente schema:

  Impiegati(#Imp,Nome,Stipendio,#Manager)

- supponiamo adesso di volere determinare tutti i manager diretti o indiretti dell'impiegato Rossi
- questa computazione non è esprimibile in SQL

# Il problema

- Nasce quindi l'esigenza di estendere il potere espressivo di SQL
- **idea**: combinare SQL con linguaggi di programmazione
  - SQL: utilizzato per l'accesso ai dati
  - linguaggio di programmazione: utilizzato per manipolare i dati

# Approcci

- Tre possibili approcci:
  - **estensioni imperative di SQL**: si estende SQL con tipici costrutti dei linguaggi di programmazione
  - **interfacce verso il DBMS**: si utilizza un'interfaccia per utilizzare le funzionalità del DBMS da linguaggio di programmazione
  - **embedded SQL**: si utilizza SQL all'interno di un linguaggio di programmazione

# Estensioni imperative di SQL

- Combinazione di statement SQL e costrutti imperativi
- possibilità di definire funzioni e procedure
- ogni procedura viene compilata e può essere direttamente eseguita o chiamata da programmi applicativi, utilizzando SQL embedded o interfacce per DBMS

# Interfacce per DBMS

- Libreria di funzioni del DBMS che possono essere chiamate dai programmi applicativi
- Anche chiamate Call Level Interface (CLI)
- simile alle tipiche librerie C
- Funzioni tipiche:
  - Connessione
  - Invio statement per esecuzione
  - Analisi risultato
  - Disconnessione
- Esempio:
  - la CLI di Oracle si chiama OCI (Oracle Call Interface)
  - librerie standard: ODBC, JDBC

# Embedded SQL

- Possibilità di utilizzare statement SQL in programmi scritti con tipici linguaggi di programmazione (C, Java)
- Gli statement SQL vengono processati da uno speciale precompilatore che invia lo statement al DBMS per l'esecuzione e recupera il risultato
- Necessità di gestire la comunicazione tra programma applicativo e DBMS

# Flusso

- Qualunque sia l'approccio scelto, il programma deve eseguire i seguenti passi fondamentali:
  - connessione alla base di dati
  - dichiarazione variabili di comunicazione con il DBMS
  - esecuzione statement SQL
  - analisi risultato
  - disconnessione dalla base di dati

# Flusso - Connessione

- Le procedure scritte con estensioni imperative di SQL possono venire eseguite direttamente dal DBMS,quindi non richiedono una connessione esplicita
- negli altri due casi è necessario specificare a quale base di dati ci si vuole connettere, con quale utente e quale password

# Flusso - Variabili di comunicazione

- Il linguaggio di programmazione e gli statement SQL possono in genere comunicare utilizzando opportune variabili
- tali variabili possono essere utilizzate per definire gli statement SQL o per inserire valori ottenuti come risultato di interrogazioni SQL

# Flusso - Esecuzione statement SQL

- Tutti e tre gli approcci devono dare la possibilità di eseguire statement SQL
  - interrogazioni: restituiscono in generale un insieme di tuple
  - insert, delete, update: restituiscono in genere il numero di tuple inserite, cancellate, aggiornate
  - statement DDL: restituiscono valore predefinito

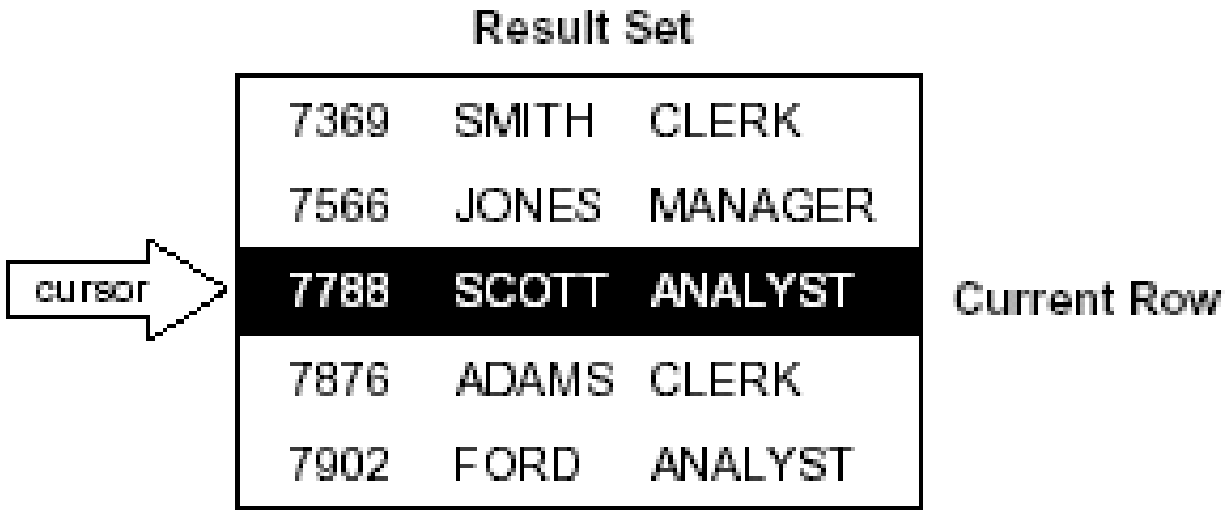
# Flusso - Analisi del risultato

- Problema relativo all'esecuzione di interrogazioni SQL
- se l'interrogazione restituisce solo una tupla:
  - Il numero di informazioni da analizzare è noto a tempo di compilazione e pari al numero degli attributi della relazione
  - è possibile definire variabili di comunicazione da utilizzare per inserire i valori degli attributi della tupla restituita
- se l'interrogazione restituisce più tuple:
  - è necessario un meccanismo che permetta di muoversi sulle tuple del risultato, una ad una, e manipolarle
  - questo è possibile utilizzando un **cursore**

# FLusso - Cursore

- Un cursore si può definire in astratto come un puntatore alle tuple ottenute come risultato di un'interrogazione SQL
- Operazioni sui cursori:
  - **dichiarazione**: associa un cursore ad un'interrogazione
  - **apertura**: esegue l'interrogazione associata al cursore e lo inizializza
  - **avanzamento**: sposta il cursore sulla tupla successiva del risultato
  - **chiusura**: disabilita il cursore

# Flusso - Cursore



# Flusso - Disconnessione

- Chiude la connessione tra  il programma applicativo e la base di dati
- nel caso di procedure scritte con estensioni imperative di SQL, quando vengono eseguite direttamente dal DBMS, questa operazione non è necessaria

# SQL statico e dinamico

- SQL statico:
  - Uso di statement SQL noti a tempo di compilazione
- SQL dinamico:
  - uso di statement SQL noti solo a tempo di esecuzione
  - tipico esempio:
    - clausola WHERE in uno statement SELECT cambia in relazione al fatto che una certa condizione sia vera o falsa
    - se siamo in maggio, determina gli impiegati con stipendio > 1000, altrimenti con stipendio > 2000;

# Estensioni imperative di SQL

SQL da programma

# Estensioni imperative di SQL

- Ogni DBMS in genere supporta un'estensione di SQL che supporta concetti tipici dei linguaggi di programmazione
- Tale linguaggio può essere usato per scrivere programmi che coinvolgano statement SQL
- Tali programmi possono essere:
  - scritti ed eseguiti direttamente da una shell
    - In questo caso chiamiamo batch l'insieme dei comandi da eseguire
  - memorizzati e richiamati quando necessario
    - procedure, package

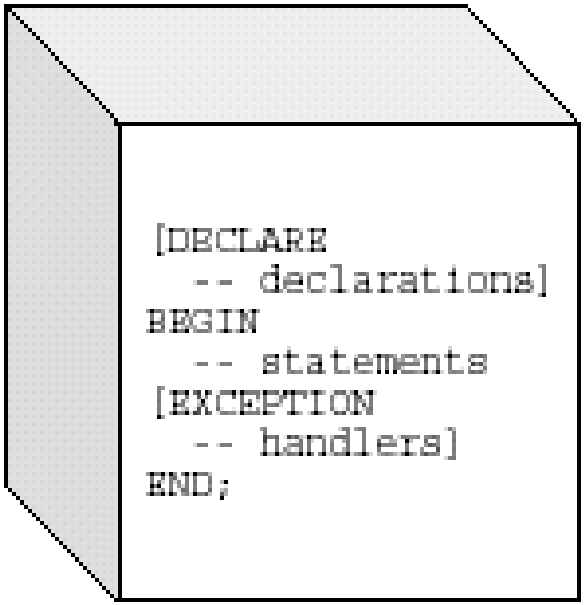
# Estensioni imperative di SQL

- Non esiste uno standard per le estensioni imperative di SQL
  - Oracle: PL/SQL
  - SQL Server: T-SQL
- ogni variante può differire dalle altre per la specifica sintassi
- ogni estensione deve permettere di specificare
  - Dichiarazioni
  - assegnamenti
  - strutture di controllo
  - istruzioni per cursori
  - modularità
  - (transazioni)

# Estensioni imperative di SQL in Oracle: PL /SQL

# PL/SQL

- É un linguaggio con struttura a blocchi
- ogni blocco è composto da tre parti:
  - parte dichiarativa
  - parte di esecuzione
  - parte di gestione delle eccezioni
- nel seguito vedremo solo alcuni aspetti di base di PL/SQL



# PL/SQL

- Nel seguito vedremo:
  - Dichiarazioni
  - statement SQL
  - Istruzioni
  - Cursori
  - Procedure, funzioni
  - Package

# Dichiarazioni

- Le dichiarazioni devono essere precedute dalla parola chiave DECLARE
- è possibile dichiarare sia costanti che variabili ed usarle ovunque possa comparire un'espressione
- le variabili possono avere un qualunque tipo SQL
- è possibile assegnare un valore di default alle variabili al momento della dichiarazione
- è possibile specificare vincoli di NOT NULL

# Dichiarazioni - Esempio

```
DECLARE
    stipendio NUMBER(4) NOT NULL;
    limite_di_credito CONSTANT NUMBER(7) := 3000000;
    data_nascita DATE;
    num_impiegati INTEGER := 0;
    manager BOOLEAN;
```

# Dichiarazioni - Uso di attributi in dichiarazioni

- Possibilità di specificare che il tipo di una variabile coincide con il tipo di un'altra variabile o di un campo di una tabella
  - impiegato  impiegati.nome%TYPE;
    - il tipo della variabile impiegato coincide con il tipo del campo nome della tabella impiegati
  - credit REAL(7,2);

    debit credit%TYPE;
    - il tipo della variabile debit coincide con il tipo della variabile credit

# Dichiarazioni - Uso di attributi in dichiarazioni

- Possibilità di specificare che una variabile rappresenta un record, corrispondente ad una tupla di una tabella
  - dept_rec dept%ROWTYPE;
    - la variabile dept_rec è dichiarata come un record, i cui elementi hanno gli stessi nomi e tipi dei campi della tabella dept

# Statement SQL

- Possibilità di eseguire statement DML
- gli statement DDL devono essere eseguiti come SQL dinamico
  - si veda oltre

# Istruzioni - Assegnamenti

- Assegnamenti di base
  - stipendio := 2000000;
  - manager := TRUE;
  - bonus := stipendio * 0.5;
- assegnamenti con valori estratti dal database
  - si esegue una select che restituisce un'unica tupla
  - si assegnano i campi della tupla ad opportune variabili
- operatori per ogni tipo supportato, come in SQL
- possibilità di effettuare assegnamenti tra variabili record basate sulla stessa tabella

# Istruzioni - Esempio

```
DECLARE
emp_id emp.empno%TYPE;
emp_name emp.ename%TYPE;
wages NUMBER(7,2);
BEGIN
...
SELECT ename, sal + comm
INTO emp_name, wages FROM emp
WHERE empno = emp_id;
...
END;
```

# Istruzioni - Strutture di controllo

- Costrutti di scelta:
  - IF THEN, IF THEN ELSE, IF THEN ELSIF
  - come in C

```
IF condition1 THEN
sequence_of_statements1
ELSIF condition2 THEN
sequence_of_statements2
ELSE
sequence_of_statements3
END IF;
```

# Istruzioni - Esempio

```
BEGIN
...
IF sales > 50000 THEN
bonus := 1500;
ELSIF sales > 35000 THEN
bonus := 500;
ELSE
bonus := 100;
END IF;
INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```

# Istruzioni - Strutture di controllo

- cicli
  - **LOOP**: LOOP
    sequence_of_statements
    EXIT WHEN boolean_expression;
    END LOOP;
  - **WHILE LOOP**: WHILE condition LOOP
    sequence_of_statements
    END LOOP;
  - **FOR LOOP**:FOR counter IN [REVERSE]
    lower_bound..higher_bound
    LOOP
    sequence_of_statements
    END LOOP;

# Istruzioni - Esempio

```
WHILE total <= 25000 LOOP
    ...
    SELECT sal INTO salary FROM emp WHERE ...
    total := total + salary;
END LOOP;
```

# Istruzioni - Strutture di controllo

- GOTO
  - è necessario associare label a determinati punti del programma
  - con l'istruzione GOTO è possibile modificare il flusso, portandolo ad eseguire l'istruzione associata alla label specificata

```
BEGIN
...
GOTO insert_row;
...
<<insert_row>>
INSERT INTO emp VALUES ...
END;
```

# Cursori

- Dichiarazione:
  - CURSOR cursor_name [(parameter[, parameter]...)]
    IS select_statement;
- apertura:
  - OPEN cursor_name;
- avanzamento:
  - FETCH cursor_name INTO record_name
  - FETCH cursor_name INTO lista_variabili
  - il cursore viene associato ad una variabile booleana NOT FOUND che diventa vera quando non ci sono più tuple da analizzare
- chiusura:
  - CLOSE cursor_name

# Cursori - Esempio

```
DECLARE
my_sal emp.sal%TYPE;
my_job emp.job%TYPE;
factor INTEGER := 2;
CURSOR c1 IS SELECT factor*sal FROM emp WHERE job = my_job;
BEGIN
...
OPEN c1;
LOOP
    FETCH c1 INTO my_sal;
    EXIT WHEN c1%NOTFOUND;
    ....
END LOOP;
CLOSE c1;
END;
```

definizione

apertura

avanzamento

chiusura

# Cursori - Esempio: uso di parametri

- DECLARE
  emp_name emp.ename%TYPE;
  CURSOR c1 **(name VARCHAR2)** IS
  SELECT * FROM emp WHERE ename = name

- Al momento dell'apertura:
  OPEN c1**("John")**;

# Eccezioni

- L'ultima sezione di un programma PL/SQL permette di gestire gli errori
- le eccezioni possono essere:
  - definite dal sistema
  - definite dall'utente

# Eccezioni di sistema

| Exception | Oracle Error | SQLCODE Value |
|---|---|---|
| ACCESS_INTO_NULL | ORA-06530 | -6530 |
| COLLECTION_IS_NULL | ORA-06531 | -6531 |
| CURSOR_ALREADY_OPEN | ORA-06511 | -6511 |
| DUP_VAL_ON_INDEX | ORA-00001 | -1 |
| INVALID_CURSOR | ORA-01001 | -1001 |
| INVALID_NUMBER | ORA-01722 | -1722 |
| LOGIN_DENIED | ORA-01017 | -1017 |
| NO_DATA_FOUND | ORA-01403 | +100 |
| NOT_LOGGED_ON | ORA-01012 | -1012 |
| PROGRAM_ERROR | ORA-06501 | -6501 |
| ROWTYPE_MISMATCH | ORA-06504 | -6504 |
| SELF_IS_NULL | ORA-30625 | -30625 |
| STORAGE_ERROR | ORA-06500 | -6500 |
| SUBSCRIPT_BEYOND_COUNT | ORA-06533 | -6533 |
| SUBSCRIPT_OUTSIDE_LIMIT | ORA-06532 | -6532 |
| SYS_INVALID_ROWID | ORA-01410 | -1410 |
| TIMEOUT_ON_RESOURCE | ORA-00051 | -51 |
| TOO_MANY_ROWS | ORA-01422 | -1422 |
| VALUE_ERROR | ORA-06502 | -6502 |
| ZERO_DIVIDE | ORA-01476 | -1476 |

# Eccezioni definite dall'utente

- Dichiarazione:

```
DECLARE
  …
  my_exception EXCEPTION;
```

- sollevamento eccezione:

```
RAISE my_exception;
```

- definizione eccezione:

```
EXCEPTION
  WHEN my_exception
  … codice PL/SQL
```

# Eccezioni – Esempio (eccezione di sistema)

```
DECLARE
    pe_ratio NUMBER(3,1);
BEGIN
      SELECT price / earnings INTO pe_ratio FROM stocks
      WHERE symbol = 'XYZ'; -- might cause division-by-zero error
      INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
      COMMIT;
EXCEPTION
      WHEN ZERO_DIVIDE THEN -- handles 'division by zero' error
      INSERT INTO stats (symbol, ratio) VALUES ('XYZ', NULL);
      COMMIT;
      ...
      WHEN OTHERS THEN -- handles all other errors
      ROLLBACK;
END; -- exception handlers and block end here
```

# Eccezioni – Esempio (eccezione definita dall'utente)

```
DECLARE
      ...
    comm_missing EXCEPTION; -- declare exception
BEGIN
      ...
      IF commission IS NULL THEN
      RAISE comm_missing; -- raise exception
      END IF;
      bonus := (salary * 0.10) + (commission * 0.15);
      EXCEPTION
      WHEN comm_missing THEN ... -- process the exception
END;
```

# Modularità

- Il codice PL/SQL può essere organizzato in unità programmative:
  - procedure, funzioni: concetto analogo a quello visto per C e Java
  - package: insieme di dichiarazioni di variabili, costanti, procedure e funzioni che possono essere memorizzate nel database e utilizzate dalle applicazioni

# Procedure e funzioni

- Possono essere create come oggetti di database con construtti DDL
  - CREATE PROCEDURE
  - CREATE FUNCTION
- possono essere dichiarate nella sezione di dichiarazione di un programma PL/SQL
  - tutto uguale a prima
  - si toglie la parola chiave CREATE

# Procedure

- Creazione procedura:
  [CREATE [OR REPLACE]]
  PROCEDURE procedure_name[(parameter[, parameter]...)]{IS|AS}
  [local declarations]
  BEGIN
  executable statements
  [EXCEPTION
  exception handlers]
  END [name];
- definizione parametri:
  parameter_name [IN | OUT | IN OUT ] datatype [{:= | DEFAULT} expression]
- chiamata:
  procedure_name(p1,...,pn);

# Procedure - Esempio

```
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
current_salary REAL;
salary_missing EXCEPTION;
BEGIN
    SELECT sal INTO current_salary FROM emp
    WHERE empno = emp_id;
    IF current_salary IS NULL THEN
    RAISE salary_missing;
    ELSE
    UPDATE emp SET sal = sal + amount
    WHERE empno = emp_id;
    END IF;
```

# Esempio (continua)

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    INSERT INTO emp_audit VALUES (emp_id, 'No such
      number');
    WHEN salary_missing THEN
    INSERT INTO emp_audit VALUES (emp_id, 'Salary is null');
END raise_salary;
```

# Funzioni

[CREATE [OR REPLACE]]
FUNCTION function_name[(parameter[, parameter]...)] RETURN
datatype} {IS|AS}
[local declarations]
BEGIN
executable statements
[EXCEPTION
exception handlers]
END [name];

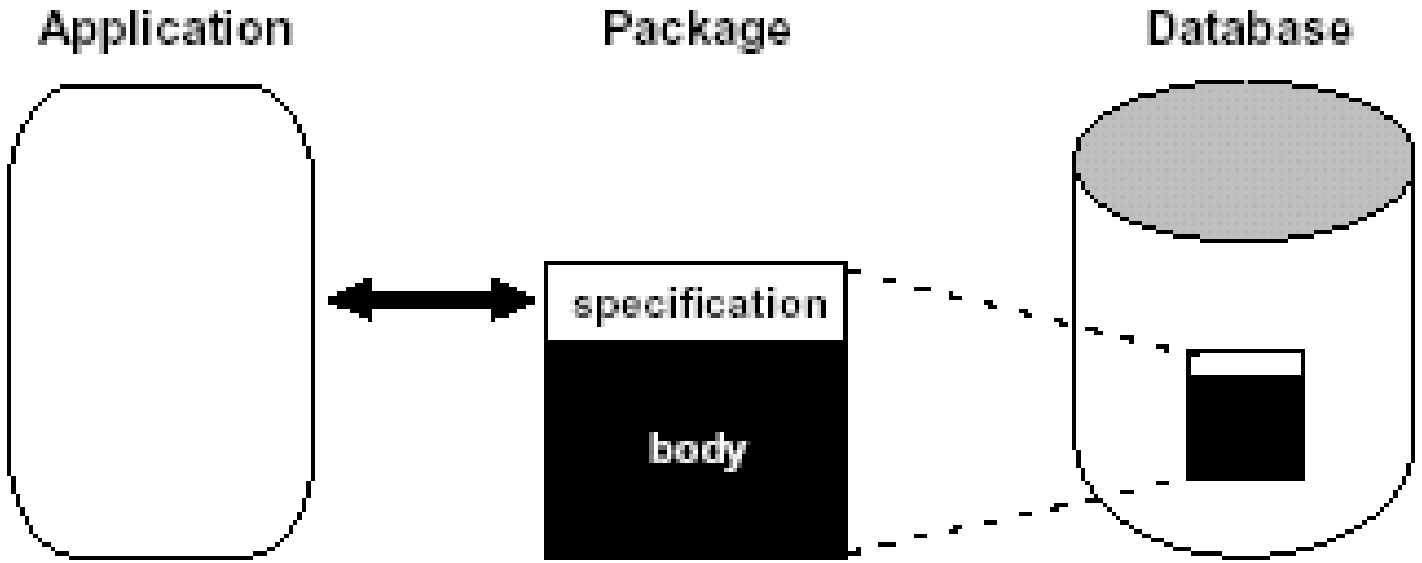
# Funzioni - Esempio

```
FUNCTION sal_ok (salary REAL, title VARCHAR2) RETURN BOOLEAN
    IS
min_sal REAL;
max_sal REAL;
BEGIN
      SELECT losal, hisal INTO min_sal, max_sal FROM sals
      WHERE job = title;
      RETURN (salary >= min_sal) AND (salary <= max_sal);
END sal_ok;
```

# Package

- Un package è un oggetto contenuto nello schema di una base di dati che contiene definizioni di variabili, costanti, procedure, funzioni, eccezioni, ecc.
- L’interfaccia del package è accessibile alle applicazioni

# Package



# Creazione interfaccia package

```
CREATE [OR REPLACE] PACKAGE package_name
{IS | AS}
...
[constant_declaration ...]
[exception_declaration ...]
[record_declaration ...]
[variable_declaration ...]
[cursor_spec ...]
[function_spec ...]
[procedure_spec ...]
[call_spec ...]
END [package_name];
```

# Interfaccia package - Esempio

```
CREATE OR REPLACE PACKAGE emp_actions AS -- spec
TYPE EmpRecTyp IS RECORD (emp_id INT, salary REAL);
CURSOR desc_salary RETURN EmpRecTyp;
PROCEDURE hire_employee (
ename VARCHAR2,
job VARCHAR2,
mgr NUMBER,
sal NUMBER,
comm NUMBER,
deptno NUMBER);
PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;
```

# Creazione body package

[CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
...
[constant_declaration ...]
[exception_declaration ...]
[record_declaration ...]
[variable_declaration ...]
[cursor_body ...]
[function_spec ...]
[procedure_spec ...]
[call_spec ...]
[BEGIN
sequence_of_statements]
END [package_name];]

# Body package - Esempio

```
CREATE OR REPLACE PACKAGE BODY emp_actions AS -- body
CURSOR desc_salary RETURN EmpRecTyp IS
SELECT empno, sal FROM emp ORDER BY sal DESC;
PROCEDURE hire_employee (
ename VARCHAR2,
job VARCHAR2,
mgr NUMBER,
sal NUMBER,
comm NUMBER,
deptno NUMBER) IS
BEGIN
INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job,
mgr, SYSDATE, sal, comm, deptno);
END hire_employee;
```

# Body package - Esempio (continua)

```
PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
DELETE FROM emp WHERE empno = emp_id;
END fire_employee;
END emp_actions;
```

# SQL dinamico

- Statement SQL non noti a tempo di compilazione possono essere eseguiti con il comando EXECUTE IMMEDIATE

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable[, define_variable]... | record}]
[USING [IN | OUT | IN OUT] bind_argument
[, [IN | OUT | IN OUT] bind_argument]...]
[{RETURNING | RETURN} INTO bind_argument[,
    bind_argument]...];
```

# SQL dinamico - EXECUTE IMMEDIATE

- Il comando è composto da tre parti principali:
  - stringa costruita dinamicamente (statement SQL o blocco PL/SQL)
  - INTO: variabili nelle quali inserire valori tupla risultato (se viene restituita una singola tupla)
  - USING: parametri da utilizzare nella stringa dinamica
  - RETURNING: variabili nelle quali inserire alcuni valori di risultato

# SQL dinamico - Esempi

```
DECLARE
sql_stmt VARCHAR2(200);
plsql_block VARCHAR2(500);
emp_id NUMBER(4) := 7566;
salary NUMBER(7,2);
dept_id NUMBER(2) := 50;
dept_name VARCHAR2(14) := 'PERSONNEL';
location VARCHAR2(13) := 'DALLAS';
emp_rec emp%ROWTYPE;
```

# SQL dinamico - Esempio (continua)

```
BEGIN
EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt
    NUMBER)';

sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
EXECUTE IMMEDIATE sql_stmt USING dept_id, dept_name, location;

sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING emp_id;

plsql_block := 'BEGIN emp_pkg.raise_salary(:id, :amt); END;';
EXECUTE IMMEDIATE plsql_block USING 7788, 500;
```

# SQL dinamico - Esempio (continua)

```
sql_stmt := 'UPDATE emp SET sal = 2000 WHERE empno = :1
RETURNING sal INTO :2';
EXECUTE IMMEDIATE sql_stmt USING emp_id RETURNING INTO salary;

EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :num'
USING dept_id;
END;
```

# SQL dinamico - Esempio

```
CREATE PROCEDURE delete_rows (
table_name IN VARCHAR2,
condition IN VARCHAR2 DEFAULT NULL) AS
where_clause VARCHAR2(100) := ' WHERE ' || condition;
BEGIN
IF condition IS NULL THEN where_clause := NULL; END IF;
EXECUTE IMMEDIATE 'DELETE FROM ' || table_name ||
    where_clause;
EXCEPTION
...
END;
```

# SQL dinamico - Cursori

- Se un'interrogazione SQL restituisce più di una tupla, è necessario analizzare il risultato utilizzando un cursore
- per i cursori, vale tutto quanto detto per i cursori per SQL statico
- unica differenza:
  - dichiarazione e apertura vengono unificate in un'unica operazione

# SQL dinamico - Cursori

OPEN {cursor_variable | :host_cursor_variable} FOR
dynamic_string
[USING bind_argument[, bind_argument]...];

Esempio:

OPEN emp_cv FOR -- open cursor variable
’SELECT ename, sal FROM emp WHERE sal > :s’
USING my_sal;

# Estensioni imperative di SQL in SQL Server: T-SQL

# T-SQL

- T-SQL è un'estensione imperativa di SQL supportata da Microsoft SQL Server
- Ogni programma T-SQL è composto da una parte di definizione e una parte di esecuzione
- Nel seguito vedremo:
  - Dichiarazioni
  - Istruzioni
  - Cursori
  - Procedure, funzioni

# Dichiarazioni

- In T-SQL è possibile dichiarare variabili per ogni tipo supportato da T-SQL
- Le variabili vengono dichiarate con la clausola DECLARE
- I nomi di variabili devono essere preceduti da @
- Esempio:

```
DECLARE @ImpID int
```

# Istruzioni - Assegnazioni

- Alle variabili è possibile assegnare valori con il comando SET

```
SET @ImpID = 1234
```

# Istruzioni - Strutture di controllo

- Classici costrutti imperativi per alterare il flusso sequenziale di esecuzione degli statement specificati
  - BEGIN END
    - stesso ruolo {} in Java
  - IF  ELSE
    - classico costrutto di scelta
  - WAITFOR
    - WAITFOR DELAY  ‘00:00:02’  aspetta due secondi
    - WAITFOR TIME ‘22:00’  riparte alle 22
  - WHILE
    - come in Java
  - CASE

# Istruzioni - Esempio

```
SELECT name, CASE state
                    WHEN 'CA' THEN 'California'
                    WHEN 'KS' THEN 'Kansas'
                    WHEN 'TN' THEN 'Tennessee'
                    WHEN 'OR' THEN 'Oregon'
                    WHEN 'MI' THEN 'Michigan'
                    WHEN 'IN' THEN 'Indiana'
                     WHEN 'MD' THEN 'Maryland'
                    WHEN 'UT' THEN 'Utah'
                END AS StateName
  FROM Authors
```

# Cursori

- Gli statement SQL restituiscono un insieme di tuple
- può capitare di dovere analizzare le tuple una per una
- in questo caso è necessario associare al risultato un cursore, cioè un puntatore che permette di muoversi all'interno di un insieme di tuple risultato
- un cursore deve:
  - essere dichiarato
  - aperto
  - utilizzato per muoversi sulle tuple
  - chiuso
- è possibile dichiarare una variabile di tipo cursore

# Cursori

- **Dichiarazione**

  DECLARE <nome cursore> CURSOR FOR
  <select statement>
- **Apertura**

  OPEN <nome cursore>
- **Avanzamento**

  FETCH NEXT FROM <nome cursore> INTO <lista variabili>
- **Chiusura**

  CLOSE <nome cursore>
- **Deallocazione**

  DEALLOCATE <nome cursore>

# Cursori

- @@FETCH_STATUS:
  - Variabile di sistema
  - è uguale a 0 se la tupla è stata letta
  - è < 0 se si è verificato qualche problema (ad esempio la tupla non esiste, siamo arrivati alla fine del result set)

# Cursori - Esempio

```
DECLARE ImpCursor CURSOR FOR   SELECT Nome FROM
    Impiegati
OPEN ImpCursor
DECLARE @NomeImp VARCHAR(10)
FETCH NEXT FROM ImpCursor INTO @NomeImp
WHILE (@@FETCH_STATUS = 0)
    BEGIN
        PRINT @NomeImp
        FETCH NEXT FROM ImpCursor INTO @NomeImp
    END
CLOSE ImpCursor
DEALLOCATE ImpCursor
```

# Organizzazione codice T-SQL

- I programmi scritti in T-SQL possono essere organizzati ed eseguiti in tre modi distinti:
  - inviati all'SQL engine interattivamente (batch)
  - organizzati in procedure (stored procedure)
  - script: sequenza di statement T-SQL memorizzati in un file e quindi eseguiti, utilizzando una funzionalità particolare di SQL Server
    - possono essere eseguiti dalla shell del DOS mediante il comando: osql
    - osql -U <nome utente> -i <nome file input> -o <nome file risultato>

# Batch

- Gruppi di statement  T-SQL inviati interattivamente all'SQL Engine ed eseguiti contemporanemante
- vengono compilati in un singolo piano di esecuzione
  - se si verifica errore di compilazione il piano non viene generato
- se si verifica un errore in esecuzione
  - gli statement seguenti non vengono eseguiti
- per speficare un batch da SQL Query Analyzer:
  - GO
- le variabili dichiarate in un batch sono locali a tale batch

# Batch - Esempio

```
CREATE TABLE Impiegati
(Imp# numeric(4) PRIMARY KEY,
Nome VarChar(20),
Mansione VarChar(20),
Data_A Datetime,
Stipendio Numeric(7,2),
Premio_P Numeric(7,2),
Dip# Numeric(2));

SELECT Nome, Dip# FROM Impiegati
WHERE Stipendio>2000 AND
Mansione = 'ingegnere';
GO
```

# Procedure (stored procedures)

- Le stored procedure sono simili al concetto di procedura (o funzione) presente nei linguaggi di programmazione
- come vedremo le applicazioni possono poi utilizzare le stored procedure per interagire con il DBMS
- funzionamento di base:
  - accettano parametri di input
  - usano un'estensione di SQL per elaborare I dati contenuti nel DB
  - settano parametri di output
  - restituiscono valori di stato per indicare se l'esecuzione ha avuto successo

# Procedure

- Le stored procedure sono un particolare tipo di oggetto per la base di dati, quindi il linguaggio dovrà permettere di:
  - Crearle, tramite il DDL
  - Eseguirle, tramite comandi specifici
- La creazione di una procedura deve essere l'unica operazione contenuta in un batch

# Procedure - Creazione ed esecuzione

CREATE PROCEDURE <nome>

<parametri>

AS

<codice>

EXECUTE <nome>

# Procedure - Esempio

```
CREATE PROCEDURE Stipendi
AS
SELECT Stipendio
FROM Impiegati;

EXECUTE Stipendi;
EXEC Stipendi;
```

# Procedure - Parametri

- I parametri sono utilizzati per scambiare valori tra la procedura e l'applicazioni o il tool che la richiama
- Tipi di parametri:
  - input
  - output
  - valore di ritorno (se non specificato si assume 0)

# Procedure - Parametri di input

```
CREATE PROCEDURE ImpSelect  @ImpID INT
AS
SELECT *
FROM Impiegati
Where Imp# = @ImpID;
```

# Procedure -Parametri di input

- Al momento dell'esecuzione della procedura, vengono specificati i parametri attuali:
  - EXEC ImpSelect @ImpID = 1234
- è inoltre possibile specificare un valore di default
- in questo caso, non sarà necessario passare un valore per il parametro

# Procedure - Esempio

```
CREATE PROCEDURE ImpSelect
@ImpID INT = 1234
AS
SELECT *
FROM Impiegati
Where Imp# = @ImpID;
GO
EXEC ImpSelect
GO
```

# Procedure - Parametri di Output

- Per restituire valori all'ambiente chiamante, è possibile utilizzare parametri di output

```
CREATE PROCEDURE AvgSal @Dip int, @Avg int OUTPUT
AS
SELECT @Avg = avg(stipendio)
FROM Impiegati
WHERE Dip# = @Dip;
GO
DECLARE @AVGex int
EXEC AvgSal @Dip = 1, @Avg = @AVGex OUTPUT
PRINT @AVGex
GO
```

# Procedure - Valori di ritorno

- L'istruzione RETURN permette di restituire un valore all'ambiente chiamanete
- Per default
  - 0 indica che l'esecuzione è andata a buon fine
  - 1 indica che l'esecuzione ha generato errori

# Procedure - Esempio

```
CREATE PROCEDURE AvgSal
@Dip int, @Avg int OUTPUT
AS
DECLARE @ErrorSave INT
SET @ErrorSave = 0
SELECT avg(stipendio)
FROM Impiegati
WHERE Dip# = @Dip;
IF (@@ERROR <>0)
    SET @ErrorSave = @@ERROR
RETURN @ErrorSave
GO
```

```
DECLARE @AVGex int
DECLARE @ReturnStatus INT
EXEC @ReturnStatus = AvgSal
          @Dip = 1, @Avg = @AVGex OUTPUT
PRINT 'Return Status= '  +
  CAST(@ReturnStatus AS CHAR(10))
PRINT @AVGex
GO
```

# SQL dinamico

- Statement dinamici possono essere eseguiti utilizzando una particolare stored procedure, definita dal sistema: sp_executesql
- parametri:
  - stringa dinamico, che può contenere parametri (variabili non dichiarate)
  - stringa che rappresenta la dichiarazione dei tipi dei parametri
  - una assegnazione per ogni parametro

# SQL dinamico - Esempio

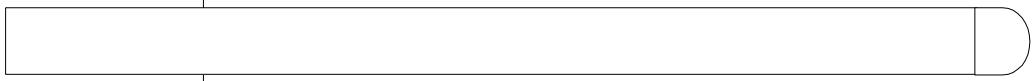
```
DECLARE @IntVariable INT
DECLARE @SQLString  NVARCHAR(500)
DECLARE @ParmDefinition NVARCHAR(500)

SET @SQLString =
N'SELECT * FROM pubs.dbo.employee WHERE job_lvl = @level'
SET @ParmDefinition = N'@level int'

SET @IntVariable = 35
EXECUTE sp_executesql @SQLString, @ParmDefinition, @level = @IntVariable

SET @IntVariable = 32
EXECUTE sp_executesql @SQLString, @ParmDefinition,@level = @IntVariable
```

# Interfacce per DBMS



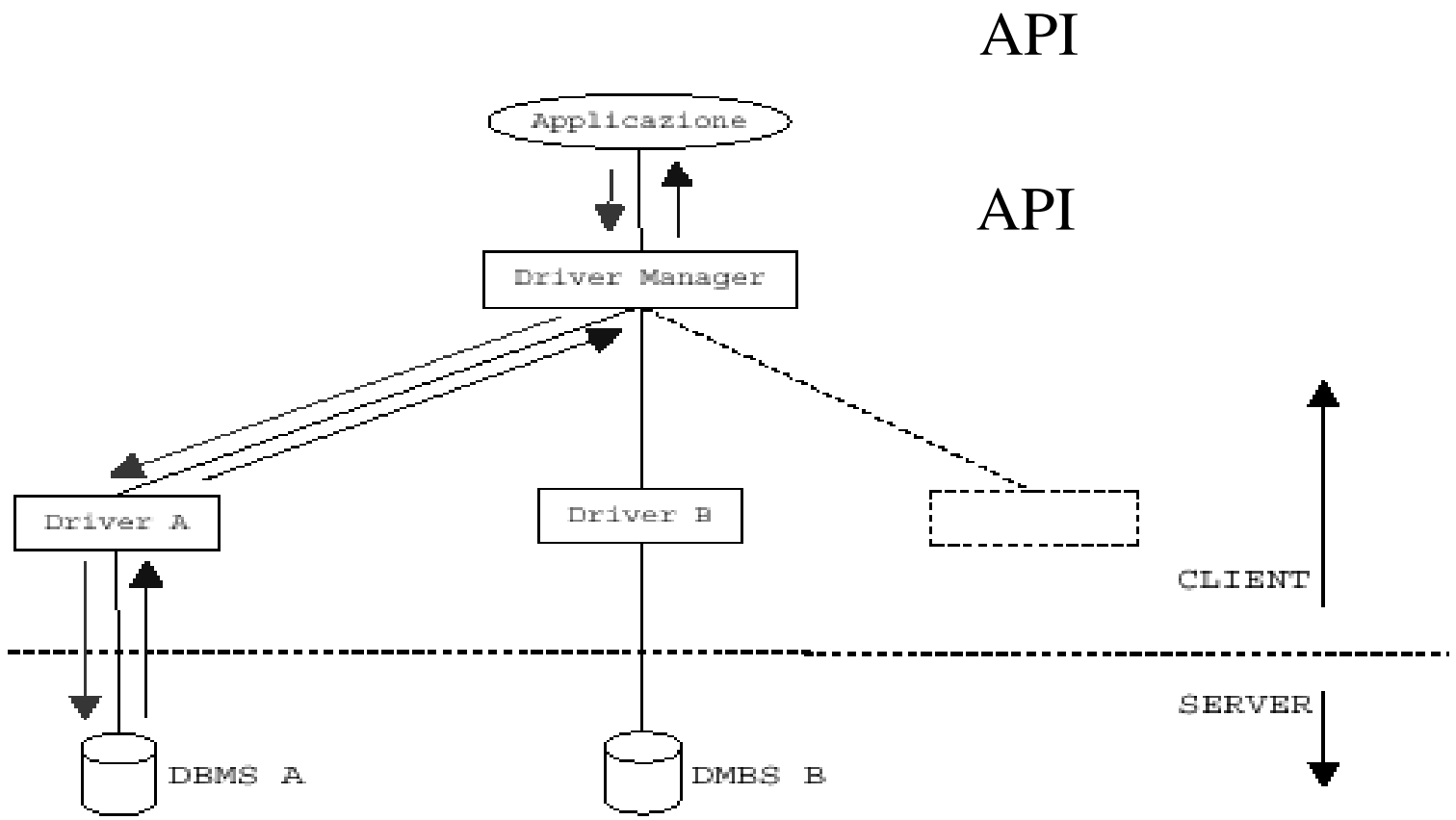
# Interfacce per DBMS

- L'idea è quella di standardizzare l'accesso ad una base di dati definendo una libreria di funzioni che possono essere utilizzate dai programmi applicativi per interagire con la base di dati
- esistono librerie proprietarie per i vari DBMS
  - Oracle: OCI (Oracle Call Interface)
- recentemente, si è cercato di standardizzare queste librerie in modo da rendere le applicazioni indipendenti dal DBMS al quale l'applicazione ha accesso
- questo permette di non dovere riscrivere completamente l'applicazione nel caso in cui il DBMS cambi
  - interoperabilità

# Interfacce per DBMS

- Sono state quindi definite interfacce standard:
  - ODBC: Scritta in C
  - JDBC: scritta in Java
- le operazioni supportate sono sostanzialmente le stesse, anche se in JDBC la sintassi è più semplice

# Architettura di riferimento



# Applicazione

- Un’applicazione è un programma che chiama specifiche funzioni API per accedere ai dati gestiti da un DBMS
- Flusso tipico:
  - selezione sorgente dati (DBMS e specifico database) e connessione
  - sottomissione statement SQL per l’esecuzione
  - recupero risultati e processamento errori
  - disconnessione

# Driver Manager

- È una libreria che gestisce la comunicazione tra applicazione e driver
- risolve problematiche comuni a tutte le applicazioni
  - quale driver caricare, basandosi sulle informazione fornite dall'applicazione
  - caricamento driver
  - chiamate alle funzioni dei driver
- l'applicazione interagisce solo con il driver manager

# Driver

- Sono librerie dinamicamente connesse alle applicazioni che implementano le funzioni API
- ciascuna libreria è specifica per un particolare DBMS
  - driver Oracle è diverso dal driver Informix
- traducono le varie funzioni API nel dialetto SQL utilizzato dal DBMS considerato (o nell'API supportata dal DBMS)
- il driver maschera le differenze di interazione dovute al DBMS usato, il sistema operativo e il protocollo di rete

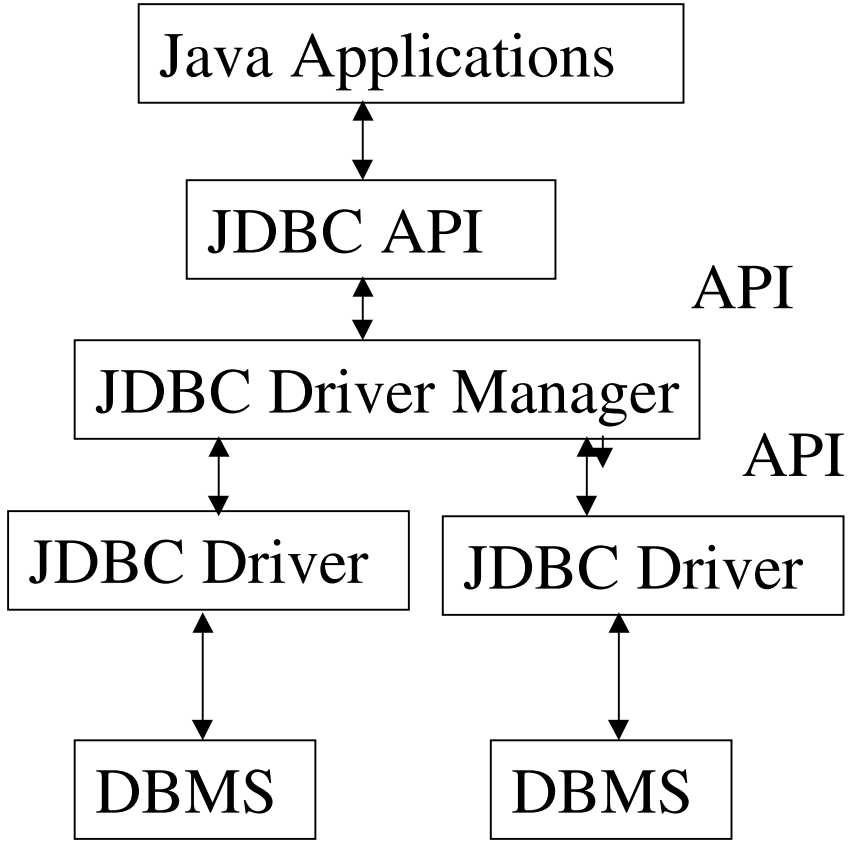
# DBMS

- Il DBMS sostanzialmente rimane inalterato nel suo funzionamento
- riceve sempre e solo richieste nel linguaggio supportato
- esegue lo statement SQL ricevuto dal driver e invia i risultati

# JDBC (Java Database Connectivity)

- JDBC (che non è solo un acronimo ma un trademark della SUN) è stato sviluppato nel 1996 dalla Sun per superare questi problemi
- Rappresenta una API standard per interagire con basi di dati da Java
- cerca di essere il più semplice possibile rimanendo al contempo flessibile
- permette di ottenere una soluzione “pure Java” per l’interazione con DBMS
  - indipendenza dalla piattaforma

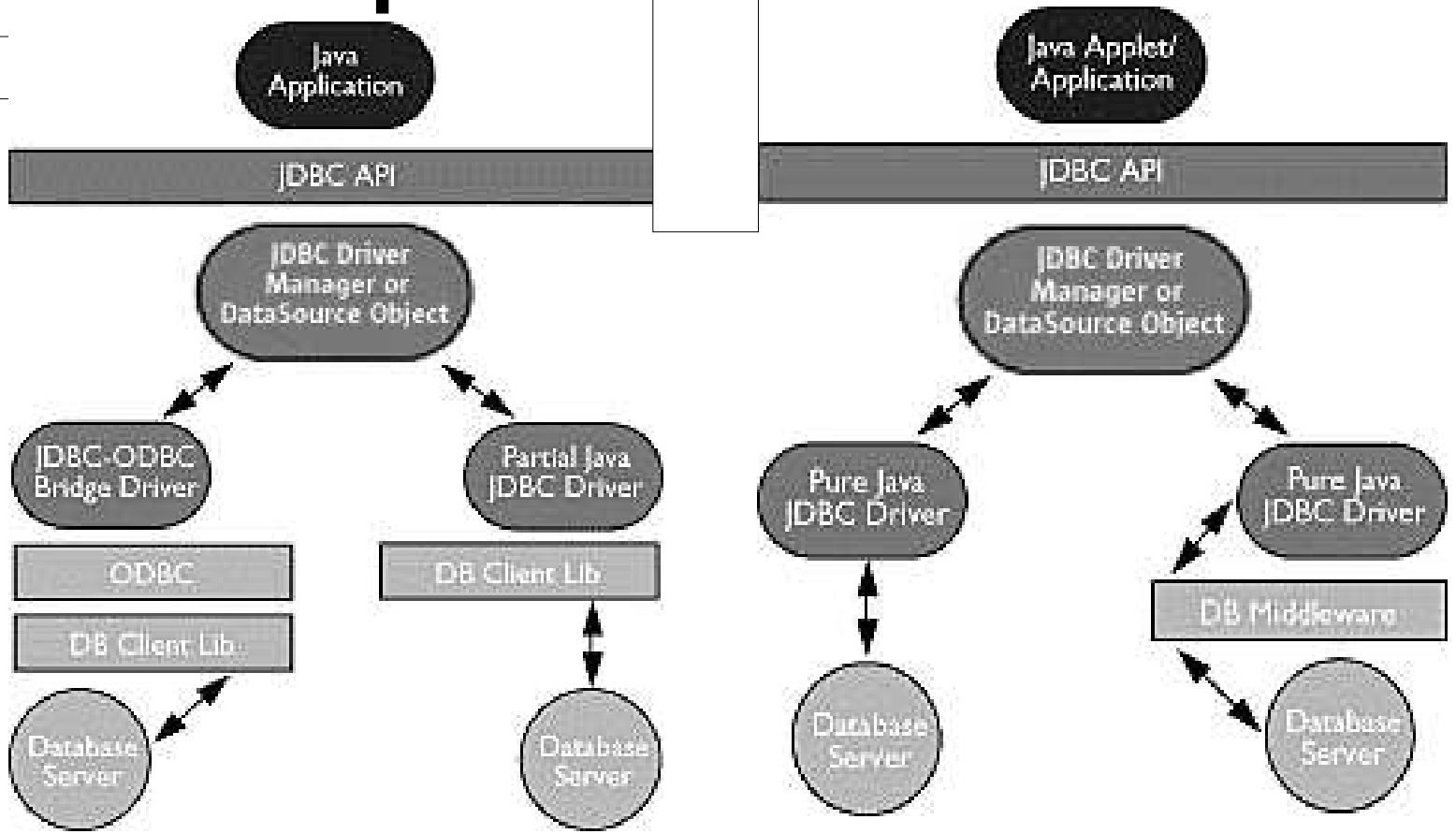
# JDBC - Architettura generale



# JDBC - Tipi di driver

- Esistono quattro tipi di driver:
  - JDBC-ODBC Bridge + ODBC Driver
  - A native-API partly Java technology-enabled driver
  - Pure Java Driver for Database Middleware
  - Direct-to-Database Pure Java Driver.

# JDBC - Tipi di driver




1 JDBC-ODBC Bridge plus ODBC Driver

2 A native-API partly Java technology-enabled driver

4 Direct-to-Database Pure Java Driver

3 Pure Java Driver for Database Middleware

# JDBC - Driver di tipo 1

- Accesso a JDBC tramite un driver ODBC
- Uso di JDBC-ODBC bridge
- il codice ODBC binario e il codice del DBMS client, deve essere caricato su ogni macchina client che usa JDBC-ODBC bridge
- si consiglia di utilizzare questa strategia quando non esistono soluzioni alternative …
- è necessario installare le librerie sul client perché non sono trasportabili su HTTP
- compromette “write once, run anywhere”

# JDBC - Driver di tipo 2

- Le chiamate a JDBC vengono tradotte in chiamate all'API del DBMS prescelto (es. OCI Oracle)
- Il driver contiene codice Java che chiama metodi scritti in C/C++
- Problemi per connessione via Web
- anche in questo caso, codice binario deve essere caricato sul client
- compromette "write once, run anywhere"

# JDBC - Driver di tipo 3

- Basato completamente su Java
- converte le chiamate JDBC nel protocollo di una applicazione middleware  che traduce quindi la richiesta del client nel protocollo proprietario del DBMS
- l'applicazione middleware può garantire l'accesso a diversi DBMS
- il protocollo middleware dipende dal DBMS sottostante

# JDBC - Driver di tipo 4

- Basato completamente su Java
- converte le chiamate JDBC nel protocollo di rete usato direttamente dal DBMS
- permette quindi un accesso diretto dalla macchina client alla macchina server
- è la soluzione più pura in termini Java

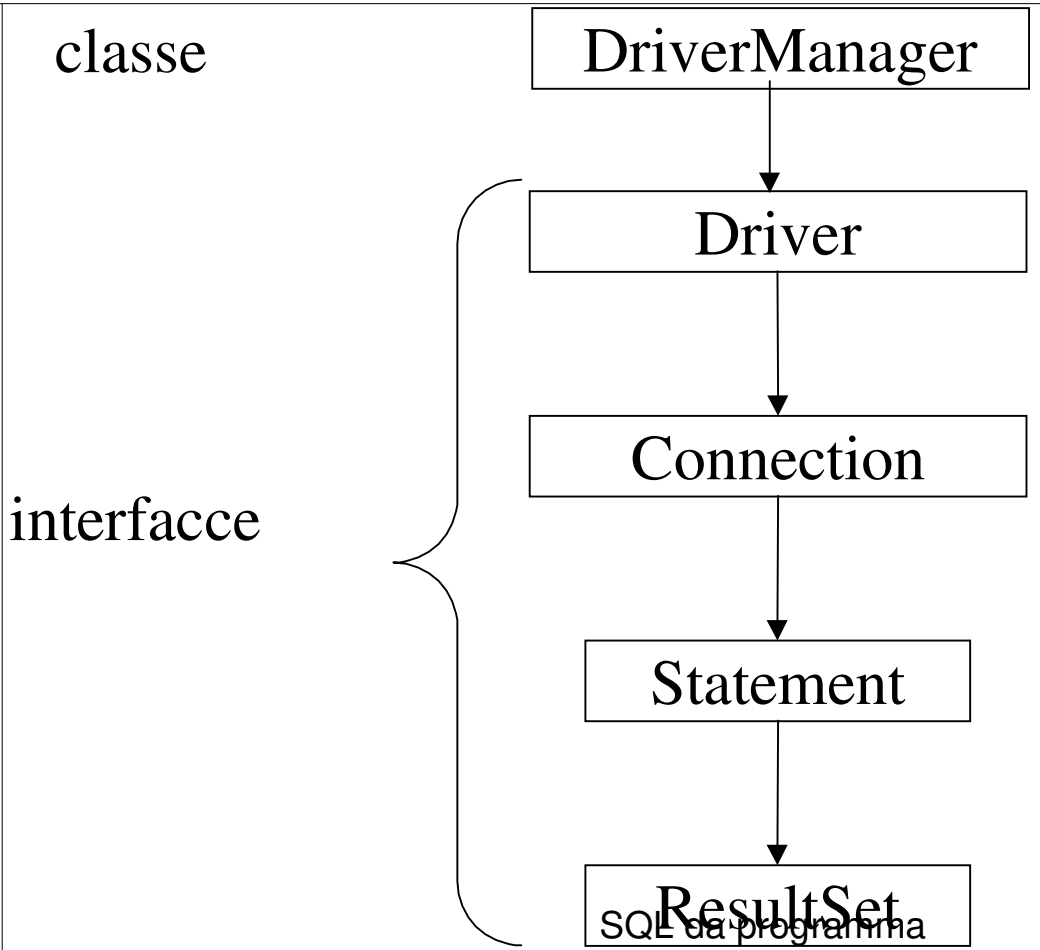
# JDBC - Tipi di dato

- JDBC definisce un insieme di tipi SQL, che vengono poi mappati in tipi Java
- Gli identificatori sono definiti nella classe java.sql.Types

# JDBC - Tipi di dato

| JDBC Types Mapped to Java Types | |
|---|---|
| **JDBC Type** | **Java Type** |
| **CHAR** | String |
| **VARCHAR** | String |
| **LONGVARCHAR** | String |
| **NUMERIC** | java.math.BigDecimal |
| **DECIMAL** | java.math.BigDecimal |
| **BIT** | boolean |
| **TINYINT** | byte |
| **SMALLINT** | short |
| **INTEGER** | int |
| **BIGINT** | long |
| **REAL** | float |
| **FLOAT** | double |
| **DOUBLE** | double |
| **BINARY** | byte[] |
| **VARBINARY** | byte[] |
| **LONGVARBINARY** | byte[] |
| **DATE** | java.sql.Date |
| **TIME** | java.sql.Time |
| **TIMESTAMP** | java.sql.Timestamp |

# JDBC - Interfaccia (semplificata)




Java.sql.*

# JDBC - Flusso applicativo

- Caricamento driver
- Connessione
- Esecuzione statement
- Disconnessione

- Nel seguito, per semplicità di notazione, negli esempi non sempre inseriremo la gestione delle eccezioni
- La gestione delle eccezioni è però necessaria (si veda avanti)

# JDBC - Passo 1: caricamento driver

- Il primo passo in un'applicazione JDBC consiste nel caricare il driver che si intende utilizzare

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

- Il nome della classe da usare viene fornito con la documentazione relativa al driver

# JDBC - Caricamento driver: esempio

```
import java.sql.*;

class JdbcTest
{
   public static void main (String args []) {
      Class.forName ("oracle.jdbc.OracleDriver");
   }
}
```

# JDBC - Passo 2: Connessione

- Per realizzare la connessione vengono utilizzate le seguenti classi ed interfaccie:
  - classe java.sql.DriverManager: gestisce la registrazione dei driver
  - interfaccia java.sql.Driver: non viene esplicitamente utilizzata a livello applicativo
  - interfaccia java.sql.Connection: permette di inviare una serie di richieste SQL al DBMS
- È possibile connettersi a qualunque database, locale e remoto, specificandone l'URL

# JDBC – Connessione

- In JDBC, l'URL è formato da tre parti:

  **jdbc: <subprotocol>: <subname>**

  - **<subprotocol>** identifica il driver o il meccanismo di connessione al database
  - **<subname>** dipende da subprotocol ed identifica lo specifico database

# JDBC - Connessione: esempi

- jdbc:oracle:thin:@everest:1521:GEN:
  - subprotocol: Oracle
  - subname:
    - thin specifica che deve essere utilizzato Oracle ODBC Thin driver
    - Everest specifica il nome della macchina
    - 1521: numero porta
    - GEN: nome database Oracle
- jdbc:mysql://cannings.org:3306/test
  - subprotocol: MySQL
  - subname:
    - cannings.org specifica il nome della macchina
    - 3306 : numero porta
    - test : nome database MySQL
- se si usa JDBC-ODBC driver: jdbc:odbc:subname

# JDBC - Connessione

- La connessione avviene chiamando il metodo getConnection della classe DriverManager, che restituisce un oggetto di tipo Connection

```
Connection con =
DriverManager.getConnection(“jdbc:mysql://cannings.org:3306
/test”, "myLogin", "myPassword");
```

- Se uno dei driver caricati riconosce l’URL fornito dal metodo, il driver stabilisce la connessione

# JDBC – Connessione: esempio

```
import java.sql.*;
class JdbcTest
{
   static String ARS_URL = "jdbc:oracle:@PutDatabaseNameHere";

   public static void main (String args [])
    {
     Class.forName ("oracle.jdbc.OracleDriver");
     Connection ARS;
     ARS =DriverManager.getConnection(ARS_URL,
                    'whitney",
                   "secret");
   } }
```

# JDBC - Passo 3: creazione ed esecuzione statement

- Creazione:
  - **Statement non preparati**: il piano di accesso viene generato immediatamente prima dell'esecuzione
  - **Statement preparati**: la generazione del piano di accesso e l'esecuzione sono due fasi distinte
    - Il piano può essere generato una sola volta ed eseguito più volte
    - Utile quando la stessa operazione deve essere eseguita con parametri diversi
- Esecuzione:
  - Interrogazioni: restituisce un insieme di tuple (Result Set)
  - Aggiornamenti: restituisce il numero di tuple modificate
  - statement DDL: restituisce 0

# JDBC - Creazione statement non preparati

- Un oggetto di tipo Statement viene creato a partire da un oggetto di tipo Connection e permette di inviare comandi SQL al DBMS:

```
Connection con;
…
Statement stmt = con.createStatement();
```

- Si noti che l'oggetto statement non è ancora associato all'istruzione SQL da eseguire
  - Tale istruzione verrà specificata al momento dell'esecuzione

# JDBC - Creazione prepared Statement

- Un oggetto di tipo PreparedStatement viene creato a partire da un oggetto di tipo Connection e permette di inviare comandi SQL al DBMS:

```
PreparedStatement queryImp = con.prepareStatement(”SELECT
    * FROM IMPIEGATI");
```

- La creazione di uno statement preparato richiede la specifica dello statement che dovrà poi essere eseguito

# JDBC - Esecuzione statement non preparati

- É necessario distinguere tra statement che rappresentano query e statement di aggiornamento
- Per eseguire una query:

```
stmt.executeQuery(”SELECT * FROM IMPIEGATI");
```

- Per eseguire una operazione di aggiornamento, inclusi gli statement DDL:

```
stmt.executeUpdate(”INSERT INTO IMPIEGATI VALUES
  ‘AB34’,‘Gianni’, ‘Rossi’,’GT67’,1500");
stmt.executeUpdate("CREATE TABLE PROVA (CAMPO1
        NUMBER)”);
```

# JDBC - Esecuzione statement preparati

- É necessario distinguere tra statement che rappresentano query e statement di aggiornamento
- Per eseguire una query:

  queryImp.executeQuery();

- Per eseguire una operazione di aggiornamento, inclusi gli statement DDL:

  queryImp.executeUpdate();

# JDBC - Esecuzione statement

- Il terminatore dello statement (es. ‘;’) viene inserito direttamente dal driver prima di sottomettere lo statement al DBMS per l’esecuzione

# JDBC - Uso di parametri in statement preparati

- È possibile specificare che la stringa che rappresenta lo statement SQL da preparare verrà completata con parametri al momento dell'esecuzione
- I parametri sono identificati da '?'

```
PreparedStatement queryImp = con.prepareStatement(
  "SELECT * FROM IMPIEGATI WHERE Nome = ?");
```

- I parametri possono poi essere associati allo statement preparato quando diventano noti
- È possibile associare valori ai parametri usando il metodo setXXX, dove XXX rappresenta un tipo Java

```
queryImp.setString(1, 'Rossi');
queryImp.executeQuery():
```

# JDBC - Passo 4: Elaborazione risultato

- JDBC restituisce i risultati di esecuzione di una query in un result set

```
String query = " SELECT * FROM IMPIEGATI ";
ResultSet rs = stmt.executeQuery(query);
```

- Il result set è costruito solo per query e non per statement di aggiornamento
- In questo caso viene restituito un intero, che rappresenta il numero di tuple modificate (0 in caso di statement DDL)

# JDBC - Elaborazione risultato

- Il metodo next() permette di spostarsi nel result set (cursore):
  - while (rs.next()) {/* get current row */}
- inizialmente il cursore è posizionato prima della prima tupla
- il metodo diventa falso quando non ci sono più tuple da analizzare

# JDBC - Metodi per accedere i valori associati agli attributi

- Il metodo  getXXX, di un ResultSet, permette di recuperare il valore associato ad un certo attributo, puntato correntemente dal cursore
- XXX è il tipo Java nel quale il valore deve essere convertito

```
String s = rs.getString(”Cognome");
```

- Gli attributi possono anche essere acceduti tramite la notazione posizionali:
  - String s = rs.getString(2);
  - int n = rs.getInt(5);
- Usare getInt per valori numerici, getString per char, varchar

# JDBC - Esempio esecuzione diretta

```
...
    Statement selImp = ARS.createStatement ();
    String stmt =  "SELECT * FROM Impiegati WHERE Cognome
                                                =‘Rossi’";
    ResultSet  impRossi = selImp.executeQuery (stmt);

    while ( impRossi.next() )
    {
      System.out.println (impRossi.getString (”Stipendio"));
    }
  ...
```

# JDBC - Esempio prepared statement

```
...
      String stmt =
                    "SELECT * FROM Impiegati WHERE Cognome =
    ‘Rossi’";
     PreparedStatement prepStmt = ARS.preparedStatement (stmt);
     ResultSet impRossi = prepStmt.executeQuery ();

       while ( impRossi.next() )
     {
       System.out.println (impRossi.getString (”Stipendio"));
     }
...
```

# JDBC - Passo 5: Disconnessione

- Per risparmiare risorse, può essere utile chiudere gli oggetti di classe Connection, Statement, ResultSet quando non vengono più utilizzati
- metodo close()
- la chiusura di un oggetto di tipo Connection chiude tutti gli Statement associati mentre la chiusura di uno Statement chiude ResultSet associati

# JDBC - Eccezioni

- La classe java.sql.SQLException estende la classe java.lang.Exception in modo da fornire informazioni ulteriori in caso di errore di accesso al database, tra cui:
  - la stringa SQLState che rappresenta la codifica dell'errore in base allo standard X/Open
    - getSQLState()
  - il codice di errore specifico al DBMS
    - getErrorCode()
  - una descrizione dell'errore
    - getMessage()

```
import java.sql.*;
import java.io.*;
class JdbcTest
{
   static String ARS_URL = "jdbc:oracle:@PutDatabaseNameHere";

   public static void main (String args [])
{   try{
      Class.forName ("oracle.jdbc.OracleDriver");
      Connection ARS;
      ARS =DriverManager.getConnection(ARS_URL,
                  "whitney", "secret");
   }
      catch (SQLException e) {
        while( e!=null){
                        System.out.println("SQLState: " + e.getSQLState());
                        System.out.println("    Code: " + e.getErrorCode());
                        System.out.println(" Message: " + e.getMessage());
                        e = e.getNextException();
}}}}
```

# JDBC - Warning

- Sottoclasse di SQLException
- la classe java.sql.SQLWarning fornisce informazioni su warning relativi all'accesso al database
  - i warning vengono collegati agli oggetti i cui metodi hanno generato l'eccezione
    - connection
    - statement
    - resultset
  - recuperabii con il metodo getWarnings()
- non bloccano l'esecuzione del programma

# JDBc – Warning: esempio

```
Statement selImp = ARS.createStatement ();
String stmt =  "SELECT * FROM Impiegati WHERE Cognome ='Rossi'";
ResultSet  impRossi = selImp.executeQuery (stmt);
while ( impRossi.next() )
      {
         System.out.println (impRossi.getString ("Stipendio"));
          SQLWarning warning_stmt = selImp.getWarnings();
          while (warning_stmt != null)
                   {
                        System.out.println("Message: " +
                                       warning_stmt.getMessage());
                        System.out.println("SQLState: " +
                                       warning_stmt.getSQLState());
                        System.out.println("Vendor error code: " +
                                       warning_stmt.getErrorCode());
   warning_stmt = warning_stmt.getNextWarning();
                   }
      }
```

# JDBC - SQL dinamico

- I metodi executeQuery() e prepareStatement() vogliono una stringa come argomento
- questa stringa non necessariamente deve essere nota a tempo di compilazione
- questo permette di eseguire facilmente statement SQL dinamici

# JDBC - SQL dinamico: esempio

- Supponiamo che la condizione WHERE di uno statement SQL sia noto solo a tempo di esecuzione

```
String query = “SELECT nome FROM Impiegati”;
if (condition)
        query += “WHERE stipendio > 1000”;
else
        query += “WHERE stipendio > 2000”;
...
```

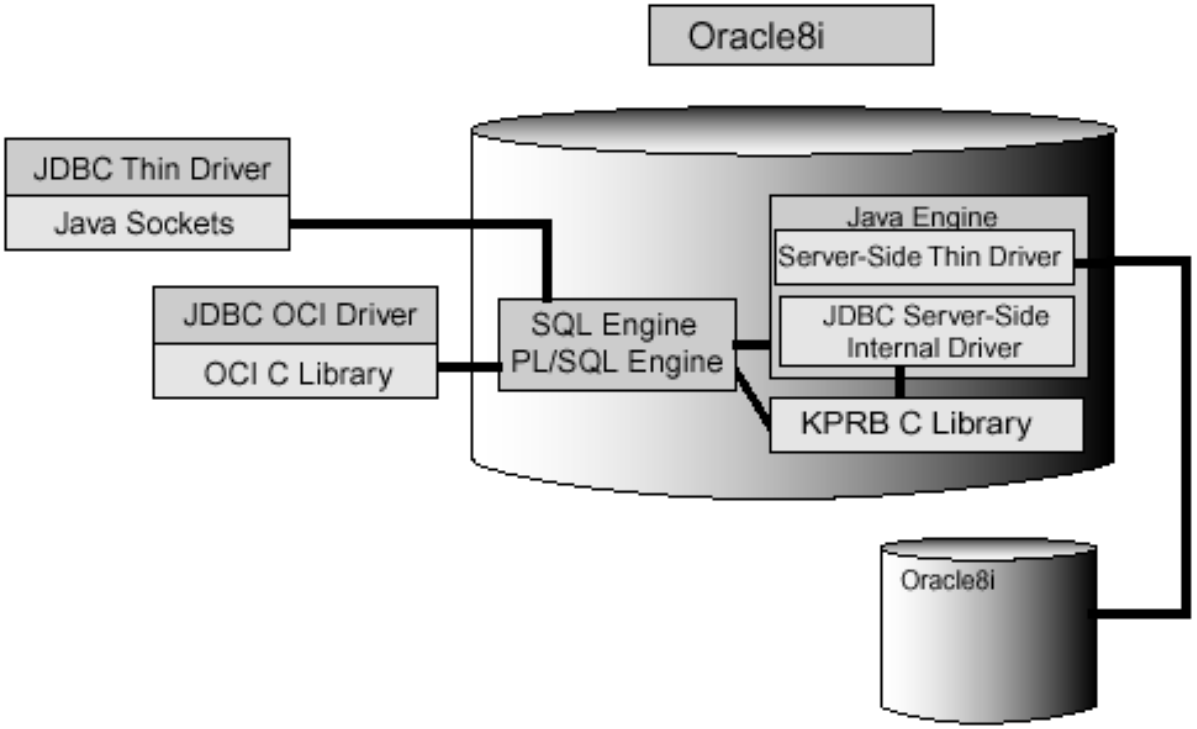
# JDBC in Oracle

- Quattro tipi di JDBC drivers:
  - uguali dal punto di vista dell'API supportata
  - diversi dal punto di vista della connessione al database
- JDBC Thin driver:
  - 100% pure Java
  - driver di tipo IV
  - usato in applet
  - non richiede software Oracle addizionale sul client
- JDBC OCI driver:
  - driver di tipo II
  - Oracle client deve essere installato sul client
  - non adatti ad applet perché dipendono dalla piattaforma Oracle
  - combinazione di Java e C
  - convertono chiamate JDBC in chiamate OCI

# JDBC in Oracle

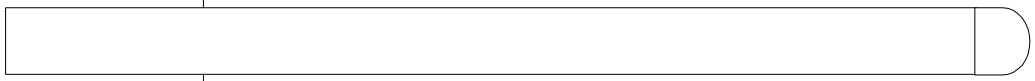
- JDBC Server-side Thin Driver:
  - stesse funzionalità di JDBC Thin driver ma viene eseguito in un database Oracle e può accedere un database remoto
  - utile per accedere un server Oracle remoto da un server Oracle
- JDBC Server-side Internal Driver:
  - comunicazione diretta con il motore SQL del server sul quale è installato
  - no accesso remoto

# JDBC in Oracle



# Embedded SQL

SQL da programma



# Concetti di base

- Il linguaggio SQL può essere usato come linguaggio "ospitato" in un linguaggio di programmazione (detto linguaggio ospite)
- per poter usare SQL da un linguaggio di programmazione sono necessarie alcune funzionalità aggiuntive ad SQL,ad esempio per trasferire risultati di interrogazioni in variabili di programma
- questa variante di SQL e' detta SQL ospitato (embedded SQL)

# Concetti di base

- Fasi nella compilazione di un programma P scritto in SQL ospitato e linguaggio ospite:
  - precompilazione di P:
    - il programma viene passato ad un precompilatore (specifico del linguaggio ospite)
    - il precompilatore elimina gli statements di SQL sostituendoli con chiamate di procedura (al DBMS) espresse nella sintassi del linguaggio ospite
    - Il risultato e' un programma scritto completamente nel linguaggio ospite
  - compilazione del programma risultato della fase precedente

# Concetti di base

- Ogni istruzione di SQL ospitato deve essere preceduta da una stringa che permette al precompilatore di distinguere un'istruzione SQL dalle istruzioni del linguaggio ospite
  - EXEC SQL: ad esempio in C
  - #sql: in SQLJ
- un'istruzione SQL eseguibile può comparire dovunque possa comparire uno statement del linguaggio ospite

# SQLJ

- Permette di utilizzare statement SQL statici all'interno di codice Java
- linguaggio semplice e conciso per inserire statement SQL in programmi Java
- standard per assemblare componenti binarie prodotte da tool differenti
- standard per garantire portabilità di codice binario tra diversi DBMS

# SQLJ - La tecnologia

- Part 0: SQLJ Embedded SQL (lo vediamo)
  - quasi completamente implementata
  - Integrata con JDBC API
- Part 1: SQLJ Stored Procedures and UDFs (cenni)
  - uso di metodi statici Java come SQL stored procedures & funzioni
- Part 2: SQLJ Data Types (non lo vediamo)
  - classi Java come SQL ADTs
  - Alternativa ad SQL99 Abstract Data Types

# SQLJ e JDBC

- **SQLJ**

```
int n;
#sql { INSERT INTO emp VALUES (:n)};
```

- **JDBC**

```
int n;
Statement stmt = conn.prepareStatement
            (“INSERT INTO emp VALUES (?)”);
stmt.setInt(1,n);
stmt.execute ();
stmt.close();
```

# SQLJ - Componenti

- SQLJ translator:
  - precompilatore
  - prende in input un file .sqlj (Java + chiamate SQL) e restituisce un file .java e uno o più profili, che contengono informazioni circa le operazioni SQL da eseguire
- SQLJ customizer:
  - specializza i profili in relazione al DBMS prescelto
- SQLJ runtime
  - viene invocato ogni volta che viene eseguito il programma

# SQLJ - Translator: passi

- Un file SQLJ è un file Java che contiene dichiarazioni e istruzioni SQL (si veda in seguito), con estensione .sqlj (es. MyClass.sqlj)
- il comando “sqlj MyClass.sqlj” invoca la JVM, che invoca a sua volta SQLJ Translator
- SQLJ Translator parserizza il codice, verificando la sintassi SQLJ
- SQLJ Translator invoca il semantic checker, per verificare la correttezza semantica (si connette al database per verificare, ad esempio, se le tabelle considerate esistono) e per effettuare il controllo dei tipi tra tipi di oggetti SQL e variabili Java
- SQLJ Translator converte le operazioni SQL in chiamate a SQLJ runtime (ad esempio in Oracle, le operazioni SQL vengono convertite in chiamate a JDBC)

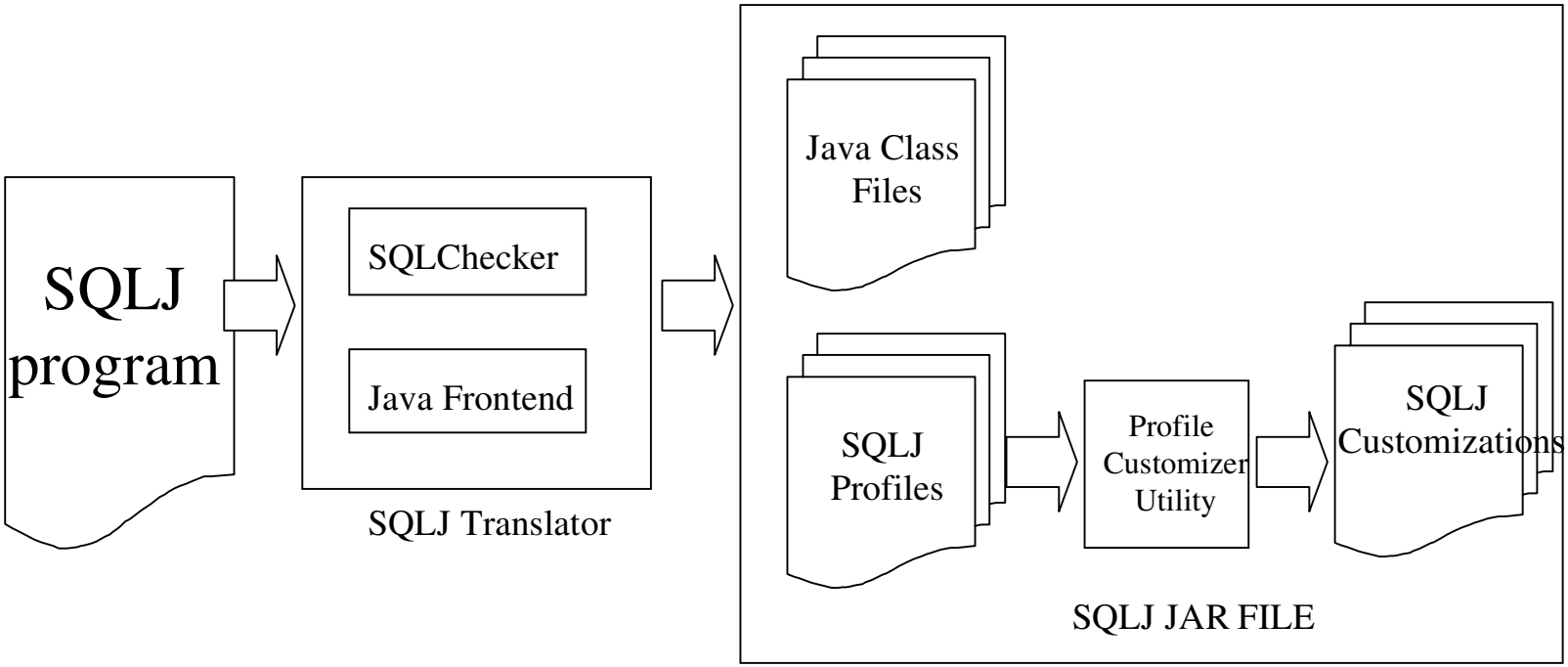
# SQLJ - Translator: passi

- SQLJ translator genera un file MyClass.java e uno o più profili
  - un profilo per ogni connessione (MyClass_SJProfile0.ser, MyClass_SJProfile1.ser)
  - ogni profilo contiene informazioni circa gli statement SQL contenuti nel codice (datatype, tabelle da accedere, ecc)
  - i profili vengono utilizzati da SQLJ runtime per recuperare le operazioni SQL ed eseguirle, utilizzando un driver JDBC
  - ogni profilo: un file *.ser
- JVM invoca il compilatore Java per compilare il file MyClass.java ottenuto, generando un insieme di file *.class
- JVM invoca l'SQLJ customizer per il sistema considerato, che specializza i profili generati al contento del DBMS che si intende utilizzare

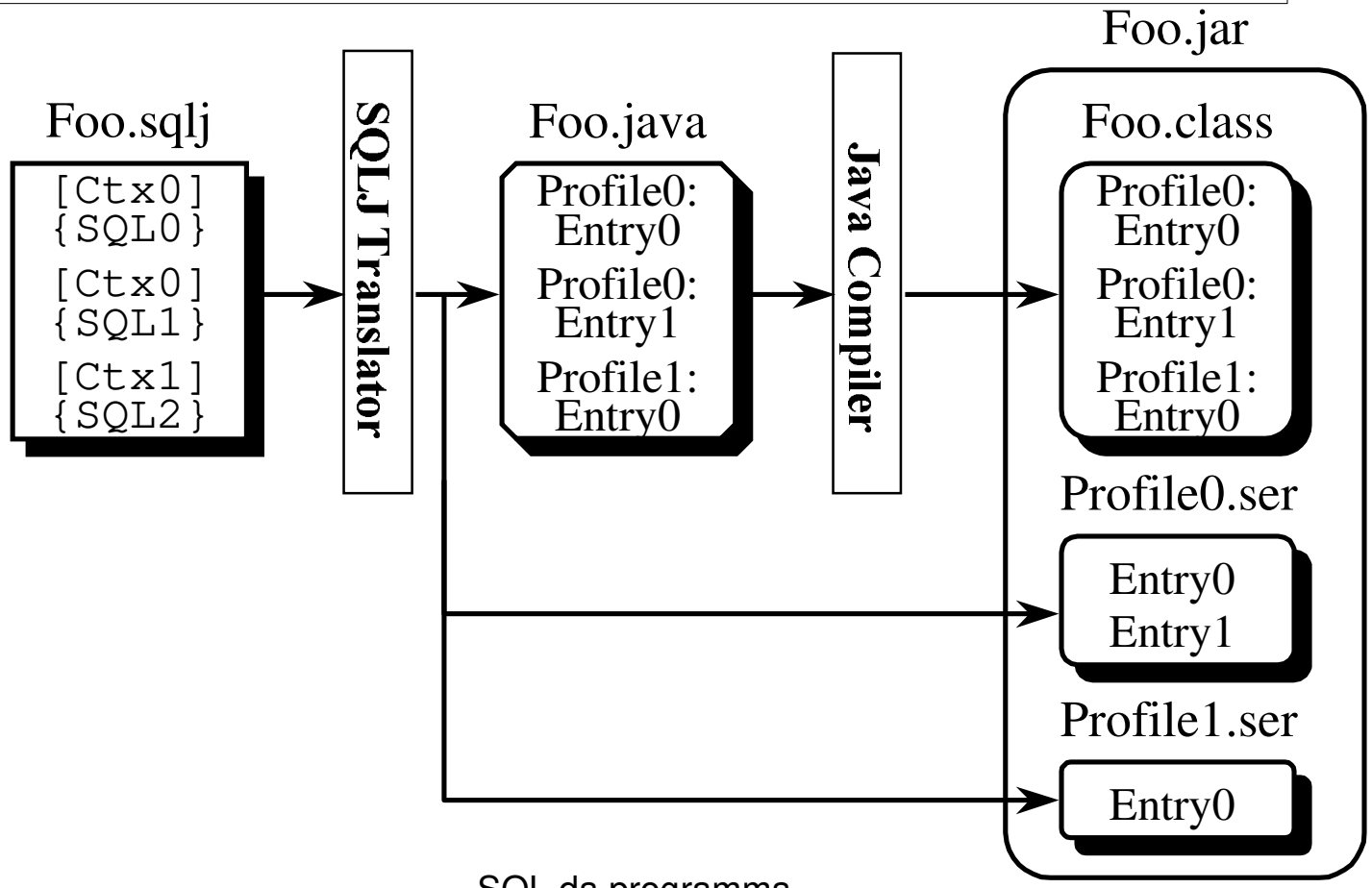
# SQLJ - Runtime processing

- Quando si lancia l'applicazione, SQLJ runtime legge i profili e crea i "profili connessi", che tengono conto della connessione alla base di dati
- durante l'esecuzione, ogni operazione SQL nel codice generato rappresenta una chiamata a SQLJ runtime che legge l'operazione da eseguire dal profilo corrispondente
- SQLJ runtime esegue l'operazione SQL tramite un driver JDBC

# SQLJ - Traduzione: schema



# SQLJ - Traduzione: schema



# SQLJ - inclusioni

- Un programma SQLJ deve includere:
  - libreria JDBC
  - librerie per SQLJ runtime

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
```

# SQLJ - Connessione

- Il primo passo in un programma SQLJ è la connessione al DBMS
- è possibile gestire nella stessa applicazioni connessioni diverse, anche a basi di dati distinte
- ogni connessione viene chiamata contesto
- non vedremo questo aspetto ma ipotizzeremo di eseguire una singola connessione
  - contesto di default

# SQLJ - Connessione

- Due passi:
  - si genera una connessione JDBC
  - si usa la connessione per definire il contesto di default

```
Class.forName(driver_name);
Connection con = DriverManager.getConnection(url,user,pwd);
DefaultContext.setDefaultContext(new DefaultContext(con);
```

# SQLJ - Costrutti di base

- Dichiarazioni
  - permettono di dichiarare particolari tipi nell'applicazione
    - iterator: cursori tipati
    - connection context: permettono di specificare connessioni che agiscono su un particolare insieme di entità (tabelle, viste, stored procedures, ecc.)
      - non li vediamo
- istruzioni (statement SQL eseguibili)
  - permettono di eseguire particolari operazioni SQL e manipolarne il risultato

# SQLJ - Statement eseguibili

- Statement eseguibili iniziano con “#sql” e terminano con “;”
- il testo SQL è racchiuso tra “{..}”
- può essere inserito ovunque possa essere inserito uno statement Java
- Due possibilità:
  - statement: #sql {sql operation};
  - assegnamento: #sql result = {sql operation};
    - se sql operation ritorna un singolo risultato: result è una variabile semplice
    - se sql operation restituisce un insieme di tuple: result deve essere un iterator (si veda oltre)

# SQLJ - esempio di statement

- #sql { INSERT INTO emp (ename, sal) VALUES (’Joe’, 43000) };
- #sql {

```
DECLARE n NUMBER;
BEGIN
    n := 1;
    WHILE n <= 100 LOOP
    INSERT INTO emp (empno) VALUES(2000 + n);
    n := n + 1;
    END LOOP;
END
};
```

# SQLJ - espressioni Java

- Esistono tre tipi di espressioni Java che possono essere utilizzate nel codice SQLJ:
  - **espressioni host**: permettono di passare argomenti tra il codice Java e le operazioni SQLJ
  - **espressioni context**: specificano il contesto nel quale deve essere eseguito un certo statement (non le vediamo)
  - **espressioni result**: variabile di output per il risultato di una query o di una chiamata di funzione

# SQLJ - Espressioni host

- Ogni espressione Java può essere utilizzata come espressione host
- vengono precedute da “:” e opzionalmente seguite dal modo di comunicazione (IN, OUT, INOUT)
  - default: OUT se l’espressione rappresenta un risultato altrimenti IN
- Esempi:
  - :hostvar
  - :INOUT hostvar
  - : IN (hostvar1 + hostvar2)

# SQLJ: espressioni host

- Le espressioni host possono essere usate in istruzioni SQL in accordo a quanto segue:
  - nella clausola INTO del comando di SELECT (in questo caso le variabili conterranno il risultato della query)
  - nella clausola SELECT
  - nella clausola WHERE dei comandi di SELECT, UPDATE, DELETE (in questo caso le espressioni rappresentano costanti da confrontare con gli attributi delle tuple)
  - nella clausola SET del comando di UPDATE (in questo caso le espressioni rappresentano i valori da assegnare alle tuple modificate)
  - nella clausola VALUES del comando di INSERT (in questo caso le espressioni rappresentano i valori da assegnare alle nuove tuple)
  - nelle espressioni aritmetiche, di stringa e temporali

# SQLJ - Espressioni host: esempi

```
String empname = "SMITH";
double salary = 25000.0;
...
#sql { UPDATE emp SET sal = :salary WHERE ename = :empname };
```

equivalente a:

```
#sql { UPDATE emp SET sal = :IN salary WHERE ename = :IN
    empname };
```

# SQLJ - Espressioni host: esempi

```
String empname;
...
#sql { SELECT ename INTO :empname FROM emp WHERE empno
    = 28959 };
```

equivalente a:

```
#sql { SELECT ename INTO :OUT empname FROM emp WHERE
    empno = 28959 };
```

# SQLJ - Espressioni host: esempi

```
float balance = 12500.0;
float minPmtRatio = 0.05;
...
#sql { UPDATE creditacct SET minPayment = :(balance *
    minPmtRatio)  WHERE acctnum = 537845 };
```

equivalente a

```
#sql { UPDATE creditacct SET minPayment = :IN (balance *
    minPmtRatio)   WHERE acctnum = 537845 };
```

# SQLJ - Espressioni host: esempi

```
String empname = "SMITH";
double raise = 0.1;
...
#sql {UPDATE emp SET sal = :(getNewSal(raise, empname))
WHERE ename = :empname};
```

# SQLJ - Analisi del risultato

- É necessario distinguere tra query che restituiscono solo una tupla e query che ne restituiscono un insieme

# SQLJ - Analisi: una singola tupla

- Ogni campo del risultato della query viene inserito in una variabile host
- SELECT expr1,…exprn INTO :host1,..,:hostn FROM T1,…,Tn

  WHERE cond
- le espressioni host possono anche comparire in expr1,…,exprn

# SQLJ - Analisi: una singola tupla

```
String empname;
#sql { SELECT ename INTO :enpname FROM emp WHERE
    empno=28959 };

String empname;
Date hdate;
#sql { SELECT ename, hiredate INTO :empname, :hdate FROM emp
WHERE empno=28959 };

#sql { SELECT sal + :raise INTO :newsal FROM emp WHERE
    empno=28959 };
```

# SQLJ - Analisi: più tuple

- Se la query restituisce un insieme di tuple, le stesse possono essere analizzate utilizzando un **iterator**
- l'iterator corrisponde al concetto di cursore ma è anche tipato
  - si specifica il tipo delle tuple sulle quali deve essere utilizzato
- un iterator è un'istanza di una classe iterator che deve essere definita dall'applicazione, specificando la struttura delle tuple che dovranno essere analizzate (tipo dell'iterator)

# SQLJ - Analisi: più tuple

- Dichiarazione iterator:
  - #sql <modifiers> iterator <iterator_name> (type declarations)
  - modifiers: public, static, ecc.
- Due tipi di iterator:
  - iterator nominali: #sql public iterator ByPos (String, int);
  - iterator posizionali: #sql public iterator ByName (int year, String name);
- quando i dati SQL vengono inseriti in un iterator, i tipi SQL vengono convertiti nei corrispondenti tipi Java specificati nella dichiarazione dell'iterator

# SQLJ - Passi generali per l'utilizzo di un iterator

- Dichiaro l'iterator class
- dichiaro una variable come istanza dell'iterator class
- popolo la variabile iterator con il risultato di una query SQL, usando uno statement SELECT
- accedo il risultato

# SQLJ - iterator nominali

- Dichiarazione:

  #sql iterator SalNamedIter (int empno, String ename, float raise);

Codice

```
class MyClass {
void func() throws SQLException {
        SalNamedIter niter = null;
        #sql niter = { SELECT ename, empno, raise
                FROM empsal };
        ... process niter ...
                                        }
                        }
```

# SQLJ - iterator posizionali

- Dichiarazione:

  #sql iterator SalNamedIter (int, String, float);

Codice

```
class MyClass {
void func() throws SQLException {
        SalNamedIter piter = null;
        #sql piter = { SELECT ename, empno, raise
                FROM empsal };
        ... process piter ...
                                        }
                        }
```

# SQLJ - dichiarazioni: differenze

- La notazione nominale è più flessibile
  - l'ordine con cui vengono selezionati gli elementi nella clausola SELECT è ininfluente in quanto l'assegnazione ai campi dell'iterator viene fatta tramite il nome
  - accesso più semplice
- la notazione posizionale richiede maggiore attenzione
  - l'ordine con cui vengono selezionati gli elementi nella clausola SELECT deve corrispondente all'ordine con cui i campi corrispondenti sono stati dichiarati nell'iterator class
  - accesso meno conciso

# SQLJ - Accesso nominale

- Per gli iterator nominali è sufficiente usare il metodo next(), predefinito per gli iterator
- tale metodo permette di muoversi dalla tupla correntemente analizzata alla successiva
- next() restituisce true se la tupla esiste, false altrimenti
- ogni iterator con campi c1,…,cn supporta i metodi c1(),…,cn() per recuperare il valore associato ai campi c1,…,cn della tupla correntemente considerata

# SQLJ - Accesso nominale: esempio

```
// Declare the iterator variable
ProjIter projsIter = null;
// Instantiate and populate iterator; order of SELECT doesn't matter
#sql projsIter = { SELECT start_date + duration AS deadline, projname, id
FROM projects WHERE start_date + duration >= sysdate };
// Process the results
while (projsIter.next()) {
System.out.println("Project name is " + projsIter.projname());
System.out.println("Project ID is " + projsIter.id());
System.out.println("Project deadline is " + projsIter.deadline());
}
// Close the iterator
projsIter.close();
```

# SQLJ - Accesso posizionale

- Per gli iterator posizionali, è necessario utilizzare l'istruzione fetch per posizionarsi sulla tupla da analizzare e memorizzare i valori associati ai campi della tupla in variabili Java
- il metodo endFetch() restituisce vero quando sono state analizzate tutte le tuple

# SQLJ - Accesso posizionale: esempio

```
// Declare and initialize host variables
int empnum=0;
String empname=null;
float salary=0.0f;
// Declare an iterator instance
EmpIter empsIter;
#sql empsIter = { SELECT ename, empno, sal FROM emp };
```

# SQLJ - Accesso posizionale: esempio

```
while (true) {
#sql { FETCH :empsIter INTO :empnum, :empname, :salary };
if (empsIter.endFetch()) break; // This test must be AFTER fetch,
// but before results are processed.
System.out.println("Name is " + empname);
System.out.println("Employee number is " + empnum);
System.out.println("Salary is " + salary);
}
// Close the iterator
empsIter.close();
...
```

# SQLJ - Procedure e funzioni

- É possibile richiamare procedure attraverso l'istruzione:
  - #sql {CALL nome_proc(parametri)};
- é possibile richiamare funzioni attraverso l'istruzione:
  - #sql  result = {VALUES nome_proc(parametri)};

# SQLJ - Esempio

```
// Import SQLJ classes:
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import oracle.sqlj.runtime.*;

// Import standard java.sql package:
import java.sql.*;
```

# SQLJ - Esempio (continua)

```
// Declare a SQLJ iterator.
// Use object types (Integer, Float) for mgr, sal, And comm rather
// than primitive types to allow for possible null selection.
#sql iterator EmpRecs(
int empno, // This column cannot be null, so int is OK.
// (If null is possible, Integer is required.)
String ename,
String job,
Integer mgr,
Date hiredate,
Float sal,
Float comm,
int deptno);
```

# SQLJ - Esempio (continua)

```
// This is the application class.
public class EmpDemo1App {
public EmpDemo1App() throws SQLException {
Class.forName ("oracle.jdbc.OracleDriver");
Connection ARS;
ARS =DriverManager.getConnection(ARS_URL,"whitney", "secret");
}
public static void main(String[] args) {
try {
EmpDemo1App app = new EmpDemo1App();
app.runExample();}
catch( SQLException exception ) {
System.err.println( "Error running the example: " + exception );}}
finally
{try { Oracle.close(); } catch(SQLException ex) {...}}
```

# SQLJ - Esempio (continua)

```
void runExample() throws SQLException {
System.out.println("\nRunning the example.\n" );
// The query creates a new instance of the iterator and stores it in
// the variable 'employees' of type 'EmpRecs'. SQLJ translator has
// automatically declared the iterator so that it has methods for
// accessing the rows and columns of the result set.
EmpRecs employees;
#sql employees = { SELECT empno, ename, job, mgr, hiredate,
sal, comm, deptno FROM emp };
```

# SQLJ - Esempio (continua)

```
// Print the result using the iterator.
while (employees.next()) {
System.out.println( "Name: " + employees.ename() );
System.out.println( "EMPNO: " + employees.empno() );
System.out.println( "Job: " + employees.job() );
System.out.println( "Manager: " + employees.mgr() );
System.out.println( "Date hired: " + employees.hiredate() );
System.out.println( "Salary: " + employees.sal() );
System.out.println( "Commission: " + employees.comm() );
System.out.println( "Department: " + employees.deptno() );
System.out.println();}
// You must close the iterator when it's no longer needed.
employees.close() ;}}
```

# SQLJ - SQL dinamico

- Gli statement eseguibili SQLJ permettono di specificare solo SQL statico
- è tuttavia possibile utilizzare SQLJ e JDBC nella stessa applicazione
- SQL dinamico è quindi supportato in SQLJ solo tramite JDBC

# SQLJ - Part 1

- Possibilità di usare metodi Java statici come procedure e funzioni SQL
  - vantaggio: possibilità di usare direttamente librerie Java esistenti
- il corpo delle procedure e funzioni contiene computazioni Java e può usare JDBC e/o SQLJ per accedere DBMSto access SQL

# SQLJ - Part1: esempio

```
public class Routines1  {
    //The region method
    //An Integer method that will be called as a function
    public static Integer region(String s) throws SQLException {
                if (s == "MN" || s == "VT" || s == "NH" ) return 1;
                else if (s == "FL" || s == "GA" || s == "AL" ) return 2;
                else if (s == "CA" || s == "AZ" || s == "NV") return 3;
                else return 4;
                }
    //The correctStates method
    //A void method that will be called as a stored procedure
    public static void correctStates (String oldSpelling, String newSpelling) throws SQLException {
                Connection conn = DriverManager.getConnection ("JDBC:DEFAULT:CONNECTION");
    PreparedStatement stmt = conn.prepareStatement ("UPDATE emps SET state = ? WHERE state
    = ?");
    stmt.setString(1, newSpelling);
    stmt.setString(2, oldSpelling);
    stmt.executeUpdate();
    return;
    }
}
```

SQL da programma

# SQLJ - Part 1: esempio

- Associazione nome SQL alla funzione/procedura

```
create procedure correct_states(old char(20), new char(20))
    modifies sql data
    external name 'routines1_jar:Routines1.correctStates'
    language java parameter style java;

create function region_of(state char(20)) returns integer
    no sql
    external name 'routines1_jar:Routines1.region'
    language java parameter style java;
```

- uso delle funzioni/procedure in statement SQL

```
select name, region_of(state) as region
    from emps
    where region_of(state) = 3

call correct_states ('CAL', 'CA');
```

# SQLJ Part 2: classi Java come tipi SQL

- Possibilità di usare classi Java come tipi di dato SQL per:
  - definire il tipo delle tabelle SQL e delle viste
  - definire il tipo dei parametri di procedure/funzioni SQL