// The parameters of the query are set.
stmt.setInt(1, 20);
stmt.setInt(2, 10);

// The query is executed.
ResultSet rs = stmt.executeQuery();

- advantages
  - If queries are executed several times in a similar way, time is saved for the repeated translation process.
  - high optimization costs only once due to one translation

Dynamic SQL in Java

- JDBC permits to pose queries dynamically, since an arbitrary object of class String is expected as input for the execution of an SQL statement.

- example:
  
  String str;
  ...
  ResultSet rs = con.createStatement().executeQuery(str);

- problem: type of the result is unknown at run time
In order to provide such type information at run time, the class `ResultSetMetaData` is used. This class offers operations to query for metadata like the number of attributes and the database types of the result. An object of the class is then created by

```java
ResultSetMetaData rsmd = rs.getMetaData();
```

Afterwards the number of attributes of the result relation can be determined, for example, with the statement

```java
int count = rsmd.getColumnCount();
```

and with

```java
for (int i = 0; i < count; i++) {
    int sqlType = rsmd.getColumnType(i);
    ...
}
```

an integer is returned in each loop, which yields the type of the ith attribute. For each type the corresponding `get` function can then be called, for example.
8.3 Embedded SQL in Java (eSQL, SQLJ)

Basic principles

- use of a preprocessor
- static determination of database operations at translation time
- type checking between AP and database through the preprocessor
- simple transmission of data from the database into the AP
- use of the **cursor principle** for traversing relations
Syntactical tagging of database operations in Java APs

- syntax: `#sql{<SQL statement>}`

- An SQL statement relates to database objects. An exception are the so-called **host variables** that are used for the data transfer between the database and the AP.

- A host variable can be declared and used like a usual variable in Java.

- A host variable can be used in an SQL statement by preceding the variable name with a `:`.

- The purpose of host variables is to receive the results of a query. Only one result may be assigned at a time.

- extension of the **select** clause by the keyword **into** followed by the host variable

- examples:
  - `#sql{select A, B from R where B > :x}`
    The value of the variable `x` is inserted into the SQL command.
  - `#sql{select A, B into :a, :b from R where Id = 7}`
    The result is bound to the host variables `a` and `b` (assumption: `Id` is key candidate).
Creation of a connection to a database

- SQLJ needs a reference (a context) to an existing database:

  ```java
  #sql context connect
  ```

- Afterwards `connect` can be used like a class which especially contains the following constructor:

  ```java
  connect connectionObject =
  new connect("jdbc:oracle:thin:@venus.mathematik.uni-marburg.de:1521:Init_DB", "scott", "tiger");
  ```

- This context object is optionally part of the SQL statement:

  ```sql
  (connectionObject){select A, B from R where B > :x}
  ```

- During the translation of an SQLJ program, checks are feasible which can be performed with JDBC only later at run time.

Query formulation with iterators

- For SQL statements that yield more than one answer, *iterators (cursors)* can be defined.

- distinction between position-related and name-related iterators
Position-related iterators (example)
- declaration of an iterator type Pos with two components
  \[
  \textit{\#sql} \texttt{public iterator} \texttt{Pos(string, int);} \\
  \]
- declaration of a variable of that type
  \[
  \texttt{Pos x;}
  \]
- binding of an SQL command to that variable
  \[
  \textit{\#sql} \texttt{x = \{select A, B from R where B > 10\};}
  \]
- The access to the result set is then performed in a loop:
  \[
  \texttt{while (not x.endFetch()) {}
  \textit{\#sql} \{\texttt{fetch :x into :a, :b}\};
  \texttt{System.out.println(a + \" earns \" + b + \" Dollars.\);} \\
  \}
  \]

Name-related iterators (example)
- declaration
  \[
  \textit{\#sql} \texttt{public iterator} \texttt{Name(string A, int B);} \\
  \]
- declaration of a variable of that type
  \[
  \texttt{Name y;}
  \]
binding to an SQL command

```sql
y = {select A, B from R where B > 10};
```

access to the result set

```java
while (y.next()) System.out.println(y.A() + " earns " + y.B() + " Dollars.");
```

access to the values is done by calling methods where the name of the method corresponds to the name of the attribute.

Method `next` accesses the next tuple.

Set-valued operations for change and deletion

Such operations also employ iterators. The data set to be changed or deleted is bound to the iterator. Then changes can be executed.

```sql
public iterator Name(String A, int B);
Name y;
...

y = {select A, B from R where B > 10};
...

while (y.next())

```sql
{update R set B = B + 10 where current of :y};
```

The currently addressed data record is changed.
8.4 PL/SQL

Introduction

- PL/SQL is Oracle’s procedural/imperative language extension to SQL.
- Syntax is very similar to the programming language ADA.
- PL/SQL offers software engineering features like data capsuling, information hiding, overloading, and exception handling.
- PL/SQL is a block-structured language. Basic units like procedures, functions and anonymous blocks are logical blocks that can contain a number of nested subblocks. A block or subblock groups declarations and statements that logically belong together. Declarations are valid only locally to the block and do not exist any more, if the block has been executed.
- advantages with respect to a host language
  - homogeneous connection of the imperative concepts to SQL
  - type conversions are not needed
  - platform independent execution
- disadvantage: imperative concepts are not sufficient for a complete development of APs
PL/SQL block

- PL/SQL block consists of three parts
  - an optional declaration part where variables and objects can be declared,
  - an executable part where variables are manipulated,
  - an optional exception handling part where exception and errors can be dealt with that arise during execution.

- definition of an PL/SQL block

  [DECLARE <declarations>]
  BEGIN
      <statements>
      [EXCEPTION <exceptions>]
  END

Declaration part

- type declarations
  - Variables can be of an SQL data type or of an additional PL/SQL data type (e.g. boolean).
  - Variables can be assigned values.
PL/SQL also supports the definition of records

```plsql
type person_type is record (name varchar(50), salary int);
```

### variable declarations
- specialty: data types of the relations can be used for the declaration of variables
  - example: myBook books%rowtype
  - example: yourBook myBook%type

Program variables can but need not be identical to the corresponding attribute names. The %type notation in each variable declaration means that this variable is of the same type as the corresponding attribute in the relation. That is, a variable of the type of the variable myBook is declared.

### cursor declarations
- The introduction of a cursor (logical pointer to a tuple within a relation) allows the sequential processing of all tuples that form the result of a query
- constant cursor
  ```plsql
cursor current-book is select * from books;
```
- parameterized cursor
  ```plsql
cursor average-earner(from int, to int) is
  select * from persons where salary > from and salary < to;
  ```
Control structures

- imperative flow control
  - conditional statement
    \[
    \text{if} \ <\text{condition}> \ \text{then} \ <\text{PL/SQL statement}> \ \text{else} \ <\text{PL/SQL statement}> \ \text{end if};
    \]
  - loops
    \[
    \text{for} \ <\text{index variable}> \ \text{in} \ <\text{range}> \ \text{loop} \ <\text{PL/SQL statement}> \ \text{end loop};
    \]
  - while-loop, exit-when

- processing of a cursor
  - opening a cursor
    \[
    \text{open} \ \text{current-book};
    \]
    \[
    \text{open} \ \text{average-earner}(1000, 2000);
    \]
  - processing of a result set
    special loop construct:
    \[
    \text{for} \ \text{myBook} \ \text{in} \ \text{current-book} \ \text{loop} \ <\text{PL/SQL statement}> \ \text{end loop};
    \]
Procedures and functions

- In PL/SQL it is also possible to declare procedures and functions.
- A procedure is a block provided with a name and a parameter list.
- A function always yields a result with the aid of the command `return`.
- Example

  ```plsql
  function totalSalary(int from, int to) return int is
  begin
    declare p Persons%rowtype;
    int total;
    open average-earner(from, to);
    ...
    return total;
  end;
  ```

- The parameters of procedures and functions can be provided with one of the following three options: `in`, `out`, `in out`

  ```plsql
  procedure work(par1 in type1, par2 out type2, par3 in out type3) is
  <PL/SQL statements with assignments to the `in out` and `out` parameters>
  ```
Stored Procedures

- With the command **create**, functions and procedures can be stored in the DBMS in a translated form and called on request.

- advantage: no anew translation of the query necessary

- Declaration is done according to the aforementioned pattern.

- **cursor variables**
  - Frequently, it is favorable to transmit the results of a stored procedure through cursor variables to the calling PL/SQL program.

- A cursor is a reference to a list of data records.

- **two types of cursor variables**
  - **strong type**
    - `type personCursorType is ref cursor persons%rowtype;`
  - **weak type**
    - `type allCursorType is ref cursor;`

- variable declaration as usual

- At the time of its declaration the cursor variable does not have a relationship to a query.
Bindung of a cursor variable to queries

When opening a cursor, the variable is bound to a query.

```
open personCursor for select * from persons where salary > 2000
```

usual application

- opening of a cursor variable in the stored procedure/function
- handing over of the cursor to the AP, which processes the records

Despite many advantages of cursor variables, there are still many limitations:

- A cursor variable may not be opened in the update mode.
- Type `ref cursor` is only known in PL/SQL but not in SQL.

stored functions in SQL

- Stored functions can be declared and called in SQL with the following limitations:
  - The functions do not contain grouping operations.
  - All data types of the input and of the output must be known in the DBMS.
- example for the declaration of a stored function

```
create function simple (x in int) return int as begin return x/101; end simple;
```

- example of an SQL query using this function

```
select Name, simple(salary) from Persons;
```
example: program segment which yields the information about the employee with the highest salary

```sql
declare
    lname employee.lastname%type;
    fname employee.firstname%type;
    addr employee.address%type;
    esalary employee.salary%type;
begin
    select lastname, firstname, address, salary
    into lname, fname, addr, esalary
    from employee
    where salary = (select max(salary) from employee);
    dbms_output.put_line(lname, fname, addr, esalary);
exception
    when others then
        dbms_output.put_line(“Error detected!”);
end;
```
example: program segment which increases the salary of employees, whose salary is below the average salary, by 10% and which outputs the average salary, if it exceeds 30000 Dollar after the previous update.

```sql
declare
    avg-salary number;
begin
    select avg(salary) into avg-salary from employee;
    update employee
        set salary = salary * 1.1
        where salary < avg-salary;
    select avg(salary) into avg-salary from employee;
    if avg-salary > 30000 then
        dbms_output.put_line("Average salary is ", avg-salary);
    end if;
    commit;
exception
    when others then dbms_output.put_line("Update error!"); rollback;
end;
```
example: Calculate the salary increases depending on the current salaries of employees.

```
declare
cursor EmpCursor is select salary from employee for update of salary;
EmpSal employee.salary%type;
begin
    open EmpCursor;
    fetch EmpCursor into EmpSal;
    while not EmpCursor%notfound
        if EmpSal > 60000 then
            update employee set salary = salary * 1.1
            where current of EmpCursor;
        elsif EmpSal > 50000 then
            update employee set salary = salary * 1.15
            where current of EmpCursor;
        else
            update employee set salary = salary * 1.20
            where current of EmpCursor;
        end if;
        fetch EmpCursor into EmpSal;
    end loop;
end;
```