# Part 17: Application Programming II

Elmasri/Navathe:

- Section 10.5, "Programming Oracle Applications"

I have also used the following sources:

- R. Sunderraman: Oracle Programming — A Primer, Addison-Wesley, 1999. Chapter 4, "PL/SQL"
- 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.
- PL/SQL User's Guide and Reference, Oracle Corporation, 1997, Part No. A58236-01.





## Objectives

- You should know some advantages of using stored procedures.
- You should have some basic impression of the Oracle PL/SQL syntax.
- You should be able to explain what triggers are.
- You should know some applications of triggers.
- You should have some basic impression of the trigger syntax in Oracle.





## Overview

1. PL/SQL
2. Triggers in Oracle





## Introduction (1)

- PL/SQL is a procedural/imperative programming language with variables and usual control structures (loops, if, etc.).
- PL/SQL has an especially good interface to SQL, e.g. the same type system as the Oracle database.
- Procedures written in PL/SQL can be stored and executed on the database server.
- PL/SQL is not intended for creating user interfaces, although it is used in other Oracle user interface products.

  All input/output must be done with parameters or database tables. E.g., if a web page is created, it is first stored in the DB. It is a "server-side" programming language.

## Introduction (2)

Advantages:

- Instead of sending many SQL commands to the server over the network, you send only a single PL/SQL procedure, or call only a single procedure (already stored on the server).
- So the network load is reduced as the data intensive computation is done on the DB server and only the final result is sent to the client where it is displayed.
- SQL statements in PL/SQL procedures are stored in preprocessed form on the server (quicker execution).
- PL/SQL procedures with their dependencies are recorded in the data dictionary.





## Introduction (3)

Natural Development:

- DBMS were introduced to put the data of a company under central control:
  - to avoid redundancies and inconsistencies,
  - to share the data,
  - to simplify the development of new applications,
  - to simplify administration (data stored in a single place).
- Future DBMS should manage also procedures/programs, not only data.





## Introduction (4)

A Subprogram-Library is not the Same:

- If a procedure in the library was changed, all application programs must be relinked with the new version.
- If the DB schema is changed, problems are only discovered when the corresponding application program is executed.
- Application programs and used tables not in data dictionary.
- Such procedures exist only outside the DBMS, they cannot be used in queries and triggers.
- Application programs run on the client.
- Different client architectures → different versions.





## Example (1)

```
CREATE PROCEDURE
   Withdraw(acc number, amt number)
IS
   b Account.Balance%TYPE; -- Variable Decl.
   -- b has the same type as this column
BEGIN
   SELECT Balance INTO b FROM Account
      WHERE No=acc FOR UPDATE OF Balance;
   IF b < amt THEN
      INSERT INTO Rejected VALUES
         (SYSDATE, acc, amt);
```

## Example (2)

```
    ELSE -- Current Balance is sufficient.
       b := b - amt;
       UPDATE Account SET Balance = b
          WHERE No = acc;
       INSERT INTO Done
          VALUES (SYSDATE, acc, amt);
    END IF;
    COMMIT;
END;
```





## Lexical Syntax

- The lexical syntax of PL/SQL (defining tokens/words) is intentionally very similar to SQL.
- PL/SQL is case-insensitive, except in string literals.
  And in delimited/quoted identifiers.
- Comments are written “`-- ...` ⟨`Line End`⟩” if they span only a single line, or “`/* ... */`” for multi-line comments.
- PL/SQL identifiers can be up to 30 characters long.
- PL/SQL has 218 reserved words.
  This includes SQL reserved words, plus about 100 new ones.





## Data Types (1)

- The Oracle SQL data types are also PL/SQL types.
  A `VARCHAR2`-variable can be declared with a length of up to 32767 characters in PL/SQL, whereas a database column of `VARCHAR2`-type can have only 4000 bytes.
- PL/SQL has a type `BOOLEAN` with values `TRUE` and `FALSE`.
  It can be used for variables, but not stored in the DB.
- PL/SQL also has a type `BINARY_INTEGER`.
  `BINARY_INTEGER` has values $-(2^{31}-1)$ to $+2^{31}-1$. It has e.g. the subtypes `NATURAL` for non-negative numbers and `POSITIVE` for positive numbers. `PLS_INTEGER` is the same as `BINARY_INTEGER`, except that an overflow creates an exception instead of an element of the larger `NUMBER` type.





## Data Types (2)

- Procedure parameters can have only unconstrained data types, e.g. “`NUMBER`”, but not “`NUMBER(4)`”.
- PL/SQL has also record types, so you can store an entire database row into a single variable.
- PL/SQL also has collection types `VARRAY` (array of varying size) and `TABLE` (something between a set and an array).
- PL/SQL allows you to define your own data types.
- Instead of explicitly specifying a type, you can also refer to the type of a DB column (`TABLE.COLUMN%TYPE`), to the record-type corresponding to a DB row (`TABLE%ROWTYPE`) and to the type of another variable (`VARIABLE%TYPE`).

## Declarations

- A simple declaration has the form "⟨Variable⟩ ⟨Type⟩;", e.g.

```
current_balance NUMBER(8,2);
```

- You can specify an intial value, otherwise `NULL` is used:

```
emp_count PLS_INTEGER := 0;
```

- You can forbid the assignment of null values:

```
zip_code CHAR(8) NOT NULL := 'PA 15260';
```

- You can forbid any future assignments:

```
credit_limit CONSTANT NUMBER := 100.00;
```

- You can declare a variable with a database type:

```
emp_no EMP.EMPNO%TYPE;
```





## Expressions and Assignments

- Boolean expressions are similar to `WHERE`-conditions, e.g. you can use =, <>, <, >, <=, >=, `LIKE`, `BETWEEN`, `IS NULL`, `AND`, `OR`, `NOT`.
- For other data types, expressions are also very similar to SQL, e.g. what you could write in the `SELECT`-list.

  E.g. for numbers you can use +, -, *, /, for strings `||` (concatenation), and for date values - as well as + (number of days). In addition, the data type functions of Oracle SQL are also available in PL/SQL (`ABS`, `ACOS`, ...).
- An assignment has the form

  ⟨Variable⟩ := ⟨Expression⟩;





## Procedure Calls (1)

- A procedure call has the form

  ⟨Procedure⟩(⟨Parameter⟩,..., ⟨Parameter⟩);
- Procedures without parameters are called without "(...)":

  ⟨Procedure⟩;
- Procedures contained in packages (library modules) must be called in the form

  ⟨Package⟩.⟨Procedure⟩(...);
- E.g. in order to print something, one needs to call a procedure of Oracle's `DBMS_OUTPUT` package (see below):

```
DBMS_OUTPUT.PUT_LINE('Wrong Employee name');
```





## Procedure Calls (2)

- PL/SQL allows to define default values for the parameters:

```
CREATE PROCEDURE
    Withdraw(acc number,
             amt number DEFAULT 60.00)
IS ...
```

- In this case, you do not have to specify values for these parameters, e.g. `Withdraw(123456)` becomes legal.

  If no default value was specified, a parameter value must be given in the call.
- If the declared default value is "`NULL`", the procedure can detect whether a value was actually specified.

# Procedure Calls (3)

- Oracle uses default values e.g. in the HTP package for creating HTML output.
- E.g. the procedure HTP.BR is declared with two parameters, cclear and cattributes, and will print

```
<BR CLEAR="cclear" cattributes>
```

- However, you can also call it “HTP.BR;”, then both parameters default to null, and only “<BR>” will be printed.
- PL/SQL allows besides the usual “positional” notation for parameter values also a named notation, e.g.

```
Withdraw(amt => 100.00, acc => 123456);
```





# Procedure Calls (4)

- PL/SQL has not only input parameters, but also output parameters and input/output parameters:

```
CREATE PROCEDURE
    Withdraw(acc IN NUMBER, amt IN NUMBER,
                 ok OUT BOOLEAN)
```

- IN is the default, so you do not have to specify this. There is also a third mode, “IN OUT”.
- IN parameters can be called with any expression as actual parameter, OUT and IN OUT parameters need a variable.
- Inside the procedure, IN parameters are write protected, and OUT parameters are read protected.





# Conditional Statements

- The most general form of the IF-statement in PL/SQL is:

```
IF ⟨Condition⟩ THEN
    ⟨Sequence of Statements⟩
ELSIF ⟨Condition⟩ THEN
    ⟨Sequence of Statements⟩
  ⋮
ELSE
    ⟨Sequence of Statements⟩
END IF;
```

- ELSIF and ELSE can be missing.





# Loop Statements (1)

- The WHILE-statement executes a sequence of statements (the loop body) iteratively as long as a condition is true:

```
WHILE ⟨Condition⟩ LOOP
    ⟨Sequence of Statements⟩
END LOOP;
```

- E.g., we can compute the factorial $1 * 2 * 3 * \cdots * 20$:

```
factorial := 1;
i := 1;
WHILE i <= 20 LOOP
      factorial := factorial * i;
      i := i + 1;
END LOOP;
```

## Loop Statements (2)

- The FOR-statement executes a sequence of statements (the loop body) once for each possible value of a control variable.

```
FOR ⟨Variable⟩ IN [REVERSE] ⟨Lower⟩..⟨Upper⟩ LOOP
    ⟨Sequence of Statements⟩
END LOOP;
```

- E.g., we can compute the factorial $1 * 2 * 3 * \cdots * 20$:

```
factorial := 1;
FOR i IN 1..20 LOOP
    factorial := factorial * i;
END LOOP;
```

- There is a FOR-loop for working with cursors (see below).





## Loop Statements (3)

- The LOOP-statement executes a sequence of statements (the loop body) repeatedly until an EXIT statement is executed.

```
LOOP
    ⟨Sequence of Statements⟩
END LOOP;
```

- "EXIT;" ends the innermost enclosing loop.
- Instead of

```
IF ⟨Condition⟩ THEN EXIT; END IF;
```

you can also write

```
EXIT WHEN ⟨Condition⟩;
```





## Loop Statements (4)

- Computation of the factorial with LOOP statement:

```
factorial := 1;
i := 1;
LOOP
    factorial := factorial * i;
    i := i + 1;
    EXIT WHEN i > 20;
END LOOP;
```

- Of course, the FOR-Loop is the right way to compute the factorial, since you know the needed number of iterations.
- You can label loops "<<Name>> LOOP ..." and then use "EXIT Name" to finish an outer loop.





## SQL Statements (1)

- Every SQL command (SELECT, INSERT, UPDATE, DELETE, COMMIT, etc.) except DDL commands (CREATE TABLE etc.) is also a valid PL/SQL statement.
- You can use PL/SQL variables and parameters in SQL commands whereever a constant would be allowed in SQL.
- Beware of name conflicts! PL/SQL variables are not specially marked in SQL statements (unlike host variables in Embedded SQL, which are marked with ":").

  If a variable has the same name as a DB table, PL/SQL assumes that the variable is meant.

  If a variable has the same name as a DB column, PL/SQL assumes that the column is meant.

## SQL Statements (2)

- Example (Give a raise to a specific employee):

```
CREATE PROCEDURE
    raise_sal(no NUMBER, pct NUMBER) IS
BEGIN
    UPDATE EMP
        SET Sal := Sal * (1 + pct/100)
        WHERE EmpNo = no;
    INSERT INTO Sal_Development
        SELECT SYSDATE, no, Sal
        FROM Emp WHERE EmpNo = no;
END;
```





## SQL Statements (3)

Single Row SELECT:

- You can use SELECT statements which return a single row to set variables to values retrieved from the DB:

```
SELECT Sal INTO s FROM Emp WHERE EmpNo = no;
```

- This is like an assignment to the variable "s".
- Of course, you can use SELECT INTO also with multiple columns and corresponding variables:

```
SELECT Sal, Comm INTO s, c FROM ...
```

- A runtime error (exception) is generated if the SELECT statement should return no row or more than one row.





## Cursors (1)

- If a SELECT statement can return more than one row, you need a cursor (or a special kind of FOR-Loop).
- The cursor must be declared in the declaration section.
  - The declaration section is the part before the BEGIN.
  - You specify an SQL query when you declare the cursor.
- Then you OPEN the cursor (executes query).
- Next, you use a FETCH-statement in a loop to get the column values for each output row.
- Finally, you CLOSE the cursor.
- Example (see following pages): Create a web page containg the names of all employees (using Oracle's HTP package).





## Cursors (2)

```
CREATE PROCEDURE printEmployees IS
    name Emp.EName%TYPE;
    CURSOR c IS SELECT EName FROM Emp;
BEGIN
    htp.htmlOpen;
    htp.headOpen;
    htp.title('Employee List');
    htp.headClose;
    htp.bodyOpen;
    htp.heading(1, 'Employees');
    htp.ulistOpen;
```

## Cursors (3)

```
    OPEN c;
    LOOP
       FETCH c INTO name;
       EXIT WHEN c%NOTFOUND;
       htp.listItem(name);
    END LOOP;
    CLOSE c;
    htp.print('Generated:  ' || SYSDATE);
    htp.bodyClose;
    htp.htmlClose;
END;
```





## Cursors (4)

- Alternative (with WHILE-loop):

```
OPEN c;
FETCH c INTO name;
WHILE c%FOUND LOOP
   htp.listItem(name);
   FETCH c INTO name;
END LOOP;
CLOSE c;
```





## Cursors (5)

- Alternative (with FOR-loop):

```
FOR name IN c LOOP
   htp.listItem(name);
END LOOP;
```

- This also does OPEN and CLOSE automatically.
- Alternative (without explicit cursor):

```
FOR name IN (SELECT Ename FROM Emp)
LOOP
   htp.listItem(name);
END LOOP;
```





## Cursors (6)

- When you do updates in loops, it is helpful that you can refer to “CURRENT OF ⟨Cursor⟩” in WHERE-conditions.
- Then you must use a “SELECT ... FOR UPDATE” query.
- Suppose we want to assign a unique number to every instructor. We have used ALTER TABLE to add a column No to the table Instructor, but it first contains null values.
- The following program also uses PL/SQL’s record-types (which allow to store entire rows in a single variable).

# Cursors (7)

```
CREATE PROCEDURE Number_Instructors IS
    inst Instructor%ROWTYPE;
    i NUMBER(4) := 1;
    CURSOR c IS SELECT * FROM Instructor
                    FOR UPDATE;
BEGIN
    FOR inst IN c LOOP
        UPDATE Instructor SET no = i
            WHERE CURRENT OF c;
        i := i + 1;
    END LOOP;
END;
```





# Cursors (8)

- A cursor can have parameters, e.g.

```
(3)   CURSOR c(dno NUMBER) IS
(4)       SELECT EName FROM Emp
(5)       WHERE DeptNo = dno;
```

- Values for these parameters are specified in the OPEN-statement (when the query is actually executed):

```
(12)   OPEN c(20);
```

- The FOR-loop needs no explicit OPEN, so it also allows to specify the parameter value:

```
(12)   FOR name IN c(20) LOOP ... END LOOP;
```





# Error Handling (1)

- Exceptions are run-time errors, like division by zero.
- The processing of SQL statements can also cause exceptions, e.g.

```
SELECT Sal INTO s FROM Emp WHERE EmpNo = no;
```

  must return exactly one row.
- If there is no employee with the given number, Oracle will raise the exception “NO_DATA_FOUND”. If there is more than one row, the exception “TOO_MANY_ROWS” will be raised.
- Normally, such exceptions will terminate your program.





# Error Handling (2)

- PL/SQL allows you to react on such errors by writing exception handlers, i.e. sequences of statements which are executed if such an error happens.

  Exception handlers are defined at the end of the block.
- You can write handlers for specific exceptions or write a default handler (“WHEN OTHERS”).
- After the exception is handled, control returns to the enclosing block, e.g. a procedure would normally return.

  Statements after the one causing the error are not executed.
- If an exception is not handled, it is propagated to the enclosing block. It may be handled there.

## Error Handling (3)

```
CREATE PROCEDURE printSalary(no NUMBER);
   s NUMBER(6,2);
BEGIN
   SELECT Sal INTO s WHERE EmpNo = no;
   DBMS_OUTPUT.PUT_LINE('Sal = ' || s);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Wrong EmpNo');
END;
```





## Error Handling (4)

- You can declare your own exceptions: You define them like a variable with type "`EXCEPTION`".
- Then you use the statement "`RAISE` ⟨`Exception`⟩" to cause this exception.
- This allows, e.g., to have non-local jumps if you have many nested procedures and want to get back to some outer level.
- You can also use the procedure

  `raise_application_error(error_number, message);`

  to create a normal Oracle error message.

  The error number must be between -20000 and -20999.





## Anonymous PL/SQL Blocks

- "Anonymous PL/SQL blocks" are pieces of PL/SQL code which are not part of a procedure.

  E.g. you can directly enter such blocks into SQL*Plus or use them inside Embedded SQL (see below).

```
DECLARE -- Instead of procedure head
   i number;
   facorial number;
BEGIN
   ...  -- Computation of 20!, see above
   DBMS_OUTPUT.PUT_LINE(factorial);
END;
```





## Functions (1)

- Functions are similar to procedures, but return a value:

```
CREATE FUNCTION
   factorial(n INTEGER) RETURN NUMBER
IS
   i INTEGER;
   f NUMBER := 1;
BEGIN
   FOR i IN 1..n LOOP f := f * i;
                     END LOOP;
   RETURN f;
END;
```

# Functions (2)

- Functions can be used in PL/SQL expressions, e.g.

```
x := factorial(20) / 1000;
```

- Functions can also be used in SQL queries (!):

```
SELECT n, factorial(n)
FROM   test_inputs
```

- Functions are not allowed to have side effects.
- Functions must execute a `RETURN` statement, or the exception "`PROGRAM_ERROR`" is raised.
- Procedures may also contain a `RETURN` statement (without argument) to transfer the control back to the caller.





# Packages (1)

- PL/SQL has a module mechanism.
- You can define "Packages" which can contain declarations of types, procedures, functions, global variables, constants, cursors, and exceptions.
- The interface (public part) and the implementation (private part) are defined separately (information hiding).
- Global variables persist for the duration of a session.

  Across transactions. But remote calls are restricted.
- When you refer to a packaged procedure etc. from outside a package, you must write ⟨`Package`⟩`.`⟨`Name`⟩.





# Packages (2)

```
CREATE PACKAGE Com_Fun AS
   FUNCTION fac(n INTEGER) RETURN NUMBER;
   FUNCTION bino(n INTEGER) RETURN NUMBER;
END Com_Fun;
CREATE PACKAGE BODY Com_Fun AS
   FUNCTION fac(n INTEGER) RETURN NUMBER IS
      i INTEGER; f NUMBER;
   BEGIN
      ...
   END;
   ...
END Com_Fun;
```





# PL/SQL in the Oracle Server (1)

Defining Stored Procedures in SQL*Plus:

- Write the procedure definition into a file ⟨`Filename`⟩`.sql`, and then execute the file with `@`⟨`Filename`⟩.

  There must be a "/" on a line by itself after the procedure definition. After this, more commands can follow.
- If there are compilation errors, use the SQL*Plus command `SHOW ERRORS` to display the error messages.

  If you write `CREATE OR REPLACE PROCEDURE ...`, you can easily re-execute the source file after changing it.
- You delete a stored procedure with

  `DROP PROCEDURE` ⟨`Name`⟩`;`

# PL/SQL in the Oracle Server (2)

Executing PL/SQL in SQL*Plus:

- You call a procedure in SQL*Plus with

  EXECUTE ⟨Procedure⟩(⟨Parameters⟩);

  E.g.: `EXECUTE withdraw(123456, 100.00);`

  Any statement can be `EXECUTE`d, not only procedure calls.
- The `DBMS_OUTPUT` procedures write their output into a DB table. If you want that SQL*Plus displays the output, you have to "`set serveroutput on`".
- You can also directly enter anonymous PL/SQL Blocks.
- PL/SQL functions can be used in SQL queries.





# PL/SQL in the Oracle Server (3)

Bind Variables in SQL*Plus:

- If you want to call a procedure with `OUT` parameters from SQL*Plus, you must first declare a "bind variable":

  VARIABLE ⟨Name⟩ ⟨Type⟩
- Then you can use this variable, marked with "`:`", in the procedure call:

  `EXECUTE p('Input', :x);`
- You can display the current value of bind variables with

  `PRINT x`

  `set autoprint on` automatically prints bind variables.





# PL/SQL in the Oracle Server (4)

Calling PL/SQL from Pro*C:

- You can use PL/SQL also in Embedded SQL programs:

```
(1)  EXEC SQL EXECUTE
(2)      BEGIN raise_sal(:emp_no, 5); END;
(3)  END-EXEC;
```

- You can also use an anonymous PL/SQL block inside C for computations which can be done simpler in PL/SQL.

  But this is a portability problem: Only Oracle has PL/SQL.
- You can use host variables (C variables) in PL/SQL whereever a PL/SQL variable would be allowed.

  Host variables must be marked with a colon.





# PL/SQL in the Oracle Server (5)

Data Dictionary Tables:

- `USER_ARGUMENTS` contains information about the arguments (parameters) of your procedures.
- `USER_SOURCE(NAME, TYPE, LINE, TEXT)` contains the source code of your procedures.

  `TYPE` can be, e.g., `FUNCTION`, `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, `TYPE`.
- Procedures are registered in `OBJ` (alias `USER_OBJECTS`). They have `OBJECT_TYPE='PROCEDURE'`.

## Overview

1. PL/SQL
2. Triggers in Oracle





## Triggers in Oracle

- The condition part of a trigger in Oracle is quite restricted.

  You can use it only in “row triggers” (see below) and refer only to the current tuple in the condition.
- However, the action part is any PL/SQL block, so you can test arbitrary conditions there.
- Oracle allows to define wether the trigger is called `BEFORE` or `AFTER` the triggering statement.
- Oracle also distinguishes between “Row Triggers” which are called once for each row affected by the triggering statement, and “Statement Triggers” which are called only once.

  You can access the affected rows only in row triggers.





## Example (1)

- Suppose we have a table

| Inventory | | | | |
|---|---|---|---|---|
| ItemNo | ItemName | Stock | MinStock | Reordered |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

- Our application programs change this table by

```
UPDATE Inventory SET Stock = Stock - :taken
WHERE  ItemNo = :no
```

- If the `Stock` becomes smaller than `MinStock`, we want to automatically reorder the part.





## Example (2)

```
CREATE TRIGGER Reorder
AFTER UPDATE OF Stock ON Inventory
FOR EACH ROW
WHEN (new.Stock < new.MinStock
      AND new.Reordered IS NULL)
BEGIN
   INSERT INTO Order
      VALUES (SYSDATE, :new.ItemNo);
   UPDATE Inventory SET Reordered = SYSDATE
      WHERE ItemNo = :new.ItemNo;
END;
```

## Syntax (1)

Event:

- The triggering event is specified by

```
(2)   AFTER UPDATE OF Stock ON Inventory
(3)   FOR EACH ROW
```

- So this trigger will be executed after a command

```
UPDATE Inventory SET Stock = ...
```

- It is executed once for each row changed by the UPDATE.
- Triggers can fire on different events:

```
(2)   AFTER UPDATE OF Stock, MinStock OR INSERT
(3)      ON Inventory
```





## Syntax (2)

Condition:

- When the triggering event occurs, the condition is checked:

```
(4)   WHEN (new.Stock < new.MinStock
(5)         AND new.Reordered IS NULL)
```

- The condition is any WHERE condition without subqueries.
- The tuple variable “new” refers to the tuple after the update or insertion, “old” is the version before the update/deletion.
- The condition can alternatively be tested in the action.
  See next slide. This is probably slightly less efficient, but allows more general conditions to be tested.
- “WHEN” conditions can only be used in row triggers.





## Syntax (3)

```
 (1)   CREATE TRIGGER Reorder
 (2)   AFTER UPDATE OF Stock ON Inventory
 (3)   FOR EACH ROW
 (4)   BEGIN
 (5)      IF :new.Stock < :new.MinStock
 (6)         AND :new.Reordered IS NULL THEN
 (7)         INSERT INTO Order
 (8)            VALUES (SYSDATE, :new.ItemNo);
 (9)         UPDATE Inventory SET Reordered = SYSDATE
(10)            WHERE ItemNo = :new.ItemNo;
(11)      END IF;
(12)   END;
```





## Syntax (4)

Action:

- The action part is any PL/SQL block.
- A PL/SQL block has the general form

```
(1)   DECLARE
(2)      ...  -- Declarations
(3)   BEGIN
(4)      ...  -- PL/SQL Statements
(5)   EXCEPTION
(6)      ...  -- Exception handlers
(7)   END;
```

- The DECLARE and EXCEPTION parts are optional.

# Syntax (5)

- A COMMIT or ROLLBACK is not allowed in the PL/SQL block, but you can call the procedure raise_application_error to abort the SQL command which fired the trigger.
- In row triggers, you can use the variables :new and :old, which contain the tuple before and after the command.

  The conditions INSERTING, DELETING, UPDATING, UPDATING($A$) can be used to react on the type of update. You cannot access the affected tuples in statement triggers.
- In row triggers, you are not allowed to access other tuples (besides the changed one) from the updated table.

  Such triggers are executed in the middle of changing this table, so the state of the table is undefined.





# Second Example

```
(1)  CREATE TRIGGER Course_Change_Restriction
(2)  -- Changes are allowed only on weekdays.
(3)  BEFORE INSERT OR UPDATE OR DELETE ON Course
(4)  DECLARE weekday VARCHAR(3);
(5)  BEGIN
(6)     weekday := TO_CHAR(SYSDATE, 'DY');
(7)     IF weekday='SAT' OR weekday='SUN' THEN
(8)        raise_application_error(-20000,
(9)           'No changes allowed today.');
(10)    END IF;
(11) END;
```





# Termination and Confluence

- If the action part of a trigger does database changes, this might fire other triggers.
- It is possible that the trigger execution does not terminate.
- Also, if more than one trigger has to be executed, the system chooses an arbitrary sequence.
- We say that the system triggers is not confluent if the final database state depends on this execution sequence.
- In Oracle, the statement before trigger is fired first. Then for each affected row, a row before trigger is fired followed by the row after trigger. Then the statement AFTER trigger is fired.