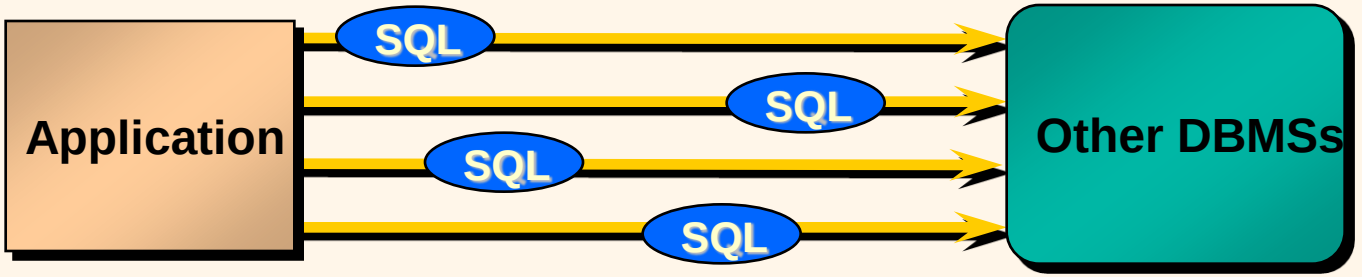
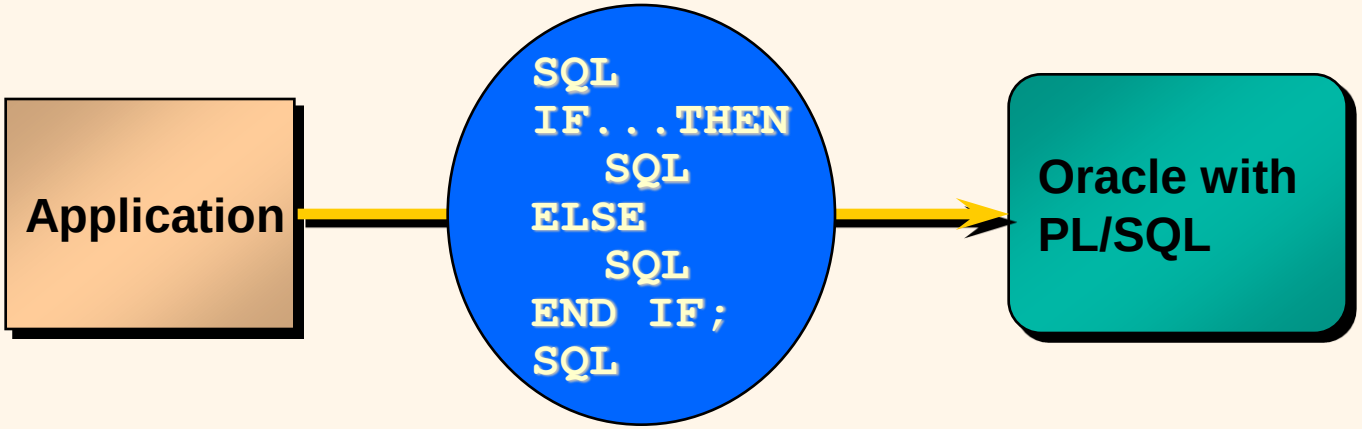
PL/SQL - Einführung

Vorteile von PL/SQL

- Enge Integration mit SQL
- Höhere Performance
- Bessere Sicherheit
- Höhere Produktivität
- Nutzung vordefinierter Packages
- Unterstützung von OO Programmierung





PL/SQL Blockstruktur

- DECLARE – Optional
  - Variable, Cursor, User-defined Exceptions (Ausnahmen)
- BEGIN – Mandatory
  - SQL Statements
  - PL/SQL Statements
- EXCEPTION – Optional
  - Aktionen, die im Fehlerfall auszuführen sind
- END; – Mandatory

```
DECLARE
  v_variable  VARCHAR2(5);
BEGIN
  SELECT     column_name
    INTO     v_variable
    FROM     table_name;
EXCEPTION
  WHEN exception_name THEN
  ...
END;
```

Blocktypen

Anonymous

```
[DECLARE]


BEGIN
  --statements

[EXCEPTION]

END;
```

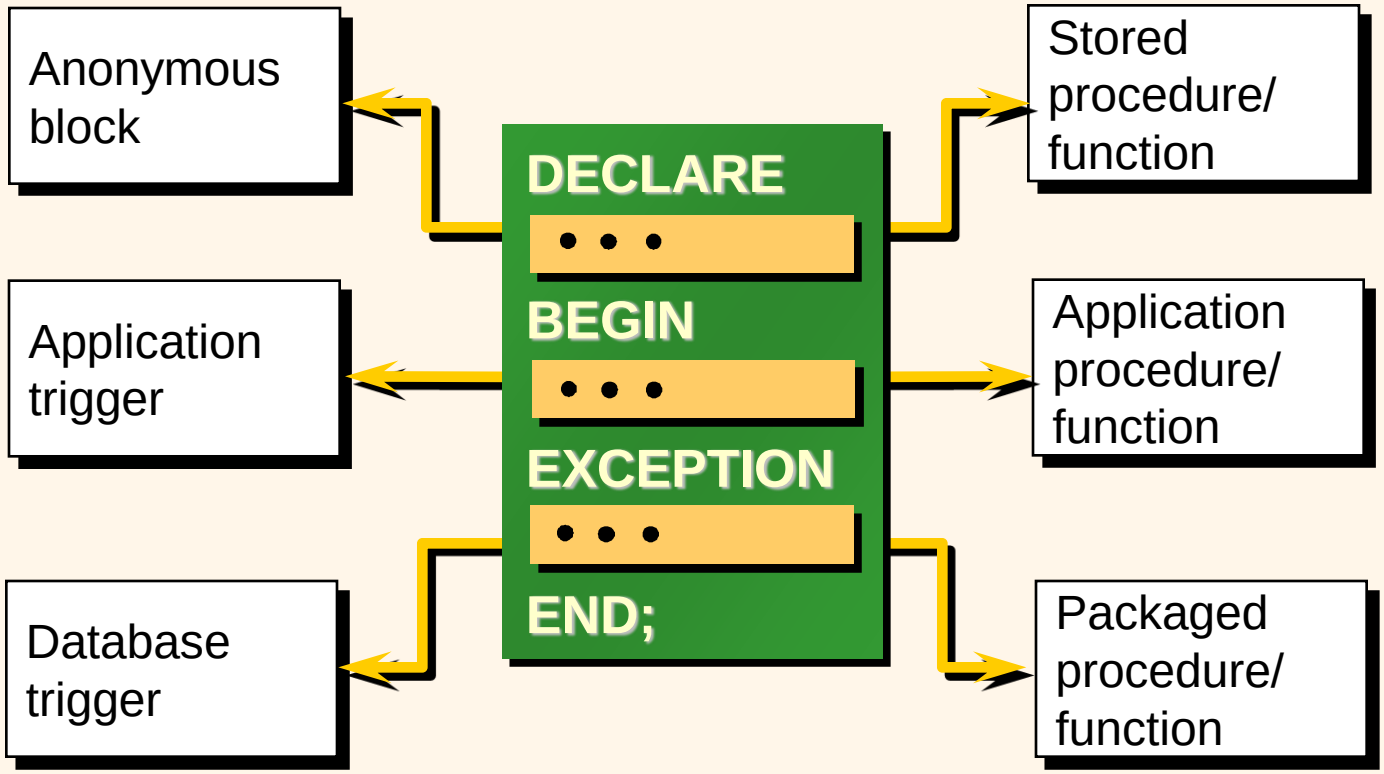
Procedure

```
PROCEDURE name
IS

BEGIN
  --statements

[EXCEPTION]

END;
```

Function

```
FUNCTION name
RETURN datatype
IS
BEGIN
  --statements
  RETURN value;
[EXCEPTION]

END;
```

Programmkonstrukte



Variablen in PL/SQL

- **Behandlung** von Variablen
  - Deklarieren und initialisieren in der Declaration Section.
  - Zuweisung neuer Werte an Variable in der Executable Section.
  - Durchreichen von Werten in PL/SQL Blöcke durch Parameter.
  - Anzeige von Ergebnissen durch Ausgabe-Variable.

- **Variablen-Typen**
  - PL/SQL Variable
    - Skalar
    - Composite (Zusammengesetzt)
    - Reference
    - LOB (large objects)
  - Non-PL/SQL Variables
  - Bind und Host-Variable

Deklaration von PL/SQL-Variablen

Syntax

```
identifier [CONSTANT] datatype [NOT NULL]
     [:= | DEFAULT expr];
```

```
DECLARE
  v_hiredate      DATE;
  v_deptno     NUMBER(2) NOT NULL := 10;
  v_location      VARCHAR2(13) := 'Atlanta';
  c_comm       CONSTANT NUMBER := 1400;
```

Wertzuweisungen an Variable

Syntax

```
identifier := expr;
```

Beispiele:

Setze ein vordefiniertes Einstellungsdatum für neue Angestellte.

```
v_hiredate := '31-DEC-98';
```

Setze den Angestellten-Namen auf “Maduro.”

```
v_ename := 'Maduro';
```

Skalare Basisdatentypen

- VARCHAR2 (*maximum_length*)
- NUMBER [(*precision, scale*)]
- DATE, TIMESTAMP, INTERVAL
- CHAR [(*maximum_length*)]
- LONG (jetzt: CLOB)
- LONG RAW (jetzt: BLOB)
- BOOLEAN
- BINARY_INTEGER / PLS_INTEGER

Skalare Variablendeklarationen (Beispiele):

```
v_job          VARCHAR2(9);
v_count          BINARY_INTEGER := 0;
v_total_sal      NUMBER(9,2) := 0;
v_orderdate      DATE := SYSDATE + 7;
c_tax_rate       CONSTANT NUMBER(3,2) := 8.25;
v_valid          BOOLEAN NOT NULL := TRUE;
```

%TYPE Attribut

- Deklariere eine Variable entsprechend
  - einer Tabellenspalten-Definition
  - einer anderen bereits deklarierten Variablen
- Präfix %TYPE mit:
  - DB-Tabelle und Spalte
  - Name der vorher deklarierten Variablen

Beispiele

```
...
  v_ename                emp.ename%TYPE;
  v_balance             NUMBER(7,2);
  v_min_balance          v_balance%TYPE := 10;
...
```

Referenzieren von Nicht-PL/SQL-Variablen

Speichere das Jahresgehalt in einer SQL*Plus Host-Variablen.

- Referenziere Nicht-PL/SQL Variable als Host-Variable.
- Referenzen erhalten als Präfix Doppelpunkt (:).

```
:g_monthly_sal := v_sal * 12;
```

Nutzung von Bind Variablen bei Dynamic SQL Statements

- In WHERE und VALUES-Klauseln dynamischer SQL-Befehle
- USING-Klausel definiert korrespondierende PL/SQL-Variable
- Anwendung des Prinzips bei Entwicklungstools wie APEX

```
'DELETE FROM employees WHERE employee_id = :id'
 USING emp_id;
```

Datentyp-Konvertierung

- Konvertiere Daten in vergleichbare Datentypen
- Gemischte Datentypen können zu Fehlern führen
- Vermeide implizite Konvertierung (langsamer, evtl. anderes Ergebnis in künftigen Releases)
- Explizite Konvertieurng mittels Konvertierungs-funktionen:
  - TO_CHAR
  - TO_DATE
  - TO_NUMBER

```
DECLARE
   v_date VARCHAR2(15);
BEGIN
   SELECT TO_CHAR(hiredate,
                  'MON. DD, YYYY')
   INTO   v_date
   FROM   emp
   WHERE  empno = 7839;
END;
```

PL/SQL Collections

- assoziative Arrays (index-by tables)
  - PL/SQL TABLE Typ
  - Elementzugriff über Position oder Suche nach Strings über Indexwert
  - Vergleichbar mit Hash-Tabellen in anderen Programmiersprachen
- Nested Tables
  - Beliebige Anzahl von Elementen
  - Nutzung der äquivalenten SQL-Typen, somit Speicherung in Datenbanktabellen und Zugriff mittels SQL
- Varrays (variable-size arrays)
  - feste Anzahl von Elementen (Obergrenze kann erhöht werden)
  - Nutzung der äquivalenten SQL-Typen, somit Speicherung in DB-Tabellen und Verarbeitung mittels SQL (weniger flexibel als nested tables)
- Schachtelung von Collections möglich (somit auch mehrdimensionale Arrays)

PL/SQL Records

- enthalten eine oder mehrere Komponenten: Skalar, RECORD, oder PL/SQL TABLE Datentyp (genannt: Felder)
- Sammlung von Feldern als logische Einheit behandelt
- ähnlich wie Records in Programmiersprachen (z.B. Pascal)
- nicht das gleiche wie Zeilen einer DB-Tabelle
- bequem zum Lesen einer Datenzeile aus Tabelle zur Verarbeitung

Erzeugen eines PL/SQL-Record

Syntax

```
TYPE type_name IS RECORD
       (field_declaration[, field_declaration]…);
identifier    type_name;
```

- *field_declaration* ist definiert als:

```
field_name {field_type | variable%TYPE
    | table.column%TYPE | table%ROWTYPE}
    [[NOT NULL] {:= | DEFAULT} expr]
```

Beispiel: Deklariere Variable, um Name, Job und Gehalt eines neuen Angestellten zu speichern

```
...
  TYPE emp_record_type IS RECORD
    (ename    VARCHAR2(10),
     job          VARCHAR2(9),
     sal          NUMBER(7,2));
  emp_record      emp_record_type;
```

%ROWTYPE Attribut

- Deklariere Variable entsprechend einer Menge von Spalten einer DB-Tabelle oder Sicht.
- Präfix %ROWTYPE mit DB-Tabelle.
- Felder im Record übernehmen Namen und Datentyp von den Spalten aus Tabelle oder Sicht
- Vorteile: Nummer und Datentyp der Spalten können sich zur Laufzeit ändern
- Nützlich für Lesen einer Zeile mit SELECT

Beispiel: Deklariere Variable, die die gleiche Information über Abteilungen und Angestellte speichert wie die zugehörige DEPT und EMP Tabelle.

```
dept_record dept%ROWTYPE;
emp_record  emp%ROWTYPE;
```

Erzeugen einer PL/SQL Table

Syntax

```
TYPE type_name IS TABLE OF
  {column_type | variable%TYPE
  | table.column%TYPE} [NOT NULL]
  [INDEX BY BINARY_INTEGER];
identifier    type_name;
```

Deklariere eine PL/SQL Variable, um einen Namen zu speichern.

Beispiel

```
...
TYPE ename_table_type IS TABLE OF emp.ename%TYPE
  INDEX BY BINARY_INTEGER;
ename_table ename_table_type;
...
```

PL/SQL Block - Syntax

- **Identifier**
  - können bis zu 30 Zeichen enthalten
  - dürfen keine Schlüsselwörter enthalten
  - müssen mit Buchstaben beginnen
  - sollten nicht gleichen Namen wie DB-Tabelle haben
- **Literale**
  - Zeichen und Datums-Literale müssen in einfachen Hochkommas eingeschlossen sein
  - auch numerische Literale möglich
- **Kommentare**
  - Einzelzeile: **--**
  - mehrere Zeilen: Kommentar zwischen /* und */

Geschachtelte Blöcke und Variablen-Scope

Beispiel

```
...
  x  BINARY_INTEGER;
BEGIN
  ...
  DECLARE
    y  NUMBER;
  BEGIN
    ...
  END;
  ...
END;
```

Scope von *x*

Scope von *y*

SQL-Befehle in PL/SQL

- Extrahiere eine Datenzeile aus der Datenbank mittels SELECT Befehl. Nur eine einfache Menge von Werten kann zurückgeliefert werden.
- Mache Änderungen an Zeilen in der Datenbank mittels DML-Befehlen.
- Kontrolliere eine Transaktion mittels COMMIT, ROLLBACK, oder SAVEPOINT Kommando.
- Bestimme DML Ergebnis mit impliziten Cursorn.

SELECT Befehle in PL/SQL

Lesen von Daten aus der Datenbank mit SELECT

```
SELECT select_list
INTO    {variable_name[, variable_name]...
        | record_name}
FROM    table
WHERE   condition;
```

Beispiel: Lese Bestell- und Lieferdatum für eine bestimmte Bestellung (INTO)

```
DECLARE
  v_orderdate   ord.orderdate%TYPE;
  v_shipdate    ord.shipdate%TYPE;
BEGIN
  SELECT   orderdate, shipdate
  INTO     v_orderdate, v_shipdate
  FROM     ord
  WHERE    id = 620;
       ...
END;
```

Ändern von Daten

Beispiel:

Erhöhe das Gehalt aller Angestellten in der Tabelle EMP, die Analysten sind.

```
DECLARE
  v_sal_increase   emp.sal%TYPE := 2000;
BEGIN
  UPDATE      emp
  SET         sal = sal + v_sal_increase
  WHERE       job = 'ANALYST';
END;
```

INSERT und DELETE funktionieren analog

Transaktionsverarbeitung in PL/SQL

- Beginne eine Transaktion mit dem ersten DML Befehl, der einem COMMIT oder ROLLBACK Befehl folgt (implizites BOT)
- Blockstruktur (BEGIN-END) ≠ Transaktionsstruktur
- COMMIT: beendet explizit Transaktion, macht Änderungen dauerhaft und nach außen sichtbar
- ROLLBACK: beendet explizit Transaktion, macht alle Änderungen rückgängig
- SAVEPOINT: Markierung innerhalb einer Transaktion zum teilweisen Zurückrollen der Transaktion: **`ROLLBACK TO Savepoint`**

SQL-Attribute

Mit SQL-Attributen kann das Ergebnis eines SQL-Befehls getestet werden.

SQL%ROWCOUNT	**Anzahl Zeilen, die vom letzten SQL-Befehl betroffen ist**
SQL%FOUND	**Boolesches Attribut, ist gleich TRUE, wenn der letzte SQL-Befehl mehr als eine Zeile betrifft**
SQL%NOTFOUND	**Boolesches Attribut, ist gleich TRUE, wenn der letzte SQL-Befehl überhaupt keine Zeile betrifft**
SQL%ISOPEN	**FALSE, weil PL/SQL implizite Cursor unmittelbar nach Ausführung schließt**

IF-Anweisungen

Syntax

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Beispiel:
Setze die Manager-ID auf 22, wenn der Angestellten-Name Osborne ist.

```
IF v_ename = 'OSBORNE' THEN
  v_mgr := 22;
END IF;
```

IF-THEN-ELSIF Anweisungen

Für einen gegebenen Wert, berechne einen Prozentanteil dieses Wertes basierend auf einer Bedingung.

Beispiel:

```
. . .
IF v_start > 100 THEN
  v_start := 2 * v_start;
ELSIF v_start >= 50 THEN
  v_start := .5 * v_start;
ELSE
  v_start := .1 * v_start;
END IF;
. . .
```

Einfache Schleife

Syntax

```
LOOP                                        -- delimiter
  statement1;                               -- statements
  . . .
  EXIT [WHEN condition];                    -- EXIT statement
END LOOP;                                   -- delimiter
```

Condition: Boolean variable | expression (TRUE, FALSE, NULL)

Beispiel

```
DECLARE
  v_ordid     item.ordid%TYPE := 601;
  v_counter   NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```

FOR-Schleife

Syntax

```
FOR counter in [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- FOR Schleife als abgekürzter Test einer bestimmten Anzahl von Iterationen
- Index wird bereits implizit deklariert

FOR-Schleife (Beispiel)

Beispiel:

Einfügen der ersten 10 Gegenstände für Bestell-Nr 601.

```
DECLARE
  v_ordid    item.ordid%TYPE := 601;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, i);
  END LOOP;
END;
```

WHILE-Schleife

Syntax

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Condition wird geprüft am Beginn jeder Iteration.

WHILE-Schleife zur Wiederholung von Befehlen solange eine Bedingung erfüllt ist, d.h.TRUE.

WHILE-Schleife (Beispiel)

Beispiel:

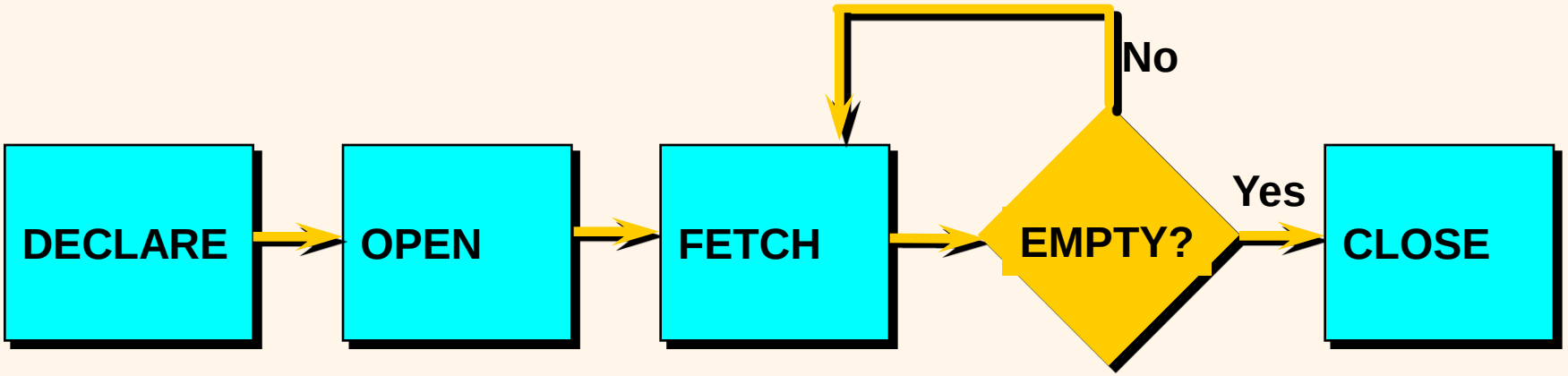
Einfügen von n Gegenständen (**p_items**) zu einer selbstgewählten Bestell-Nr. (**p_new_order**)

```
ACCEPT p_new_order PROMPT 'Enter the order number: '
ACCEPT p_items -
  PROMPT 'Enter the number of items in this order: '
DECLARE
v_count     NUMBER(2) := 1;
BEGIN
  WHILE v_count <= &p_items LOOP
    INSERT INTO item (ordid, itemid)
    VALUES (&p_new_order, v_count);
    v_count := v_count + 1;
  END LOOP;
  COMMIT;
END;
/
```

SQL Cursor

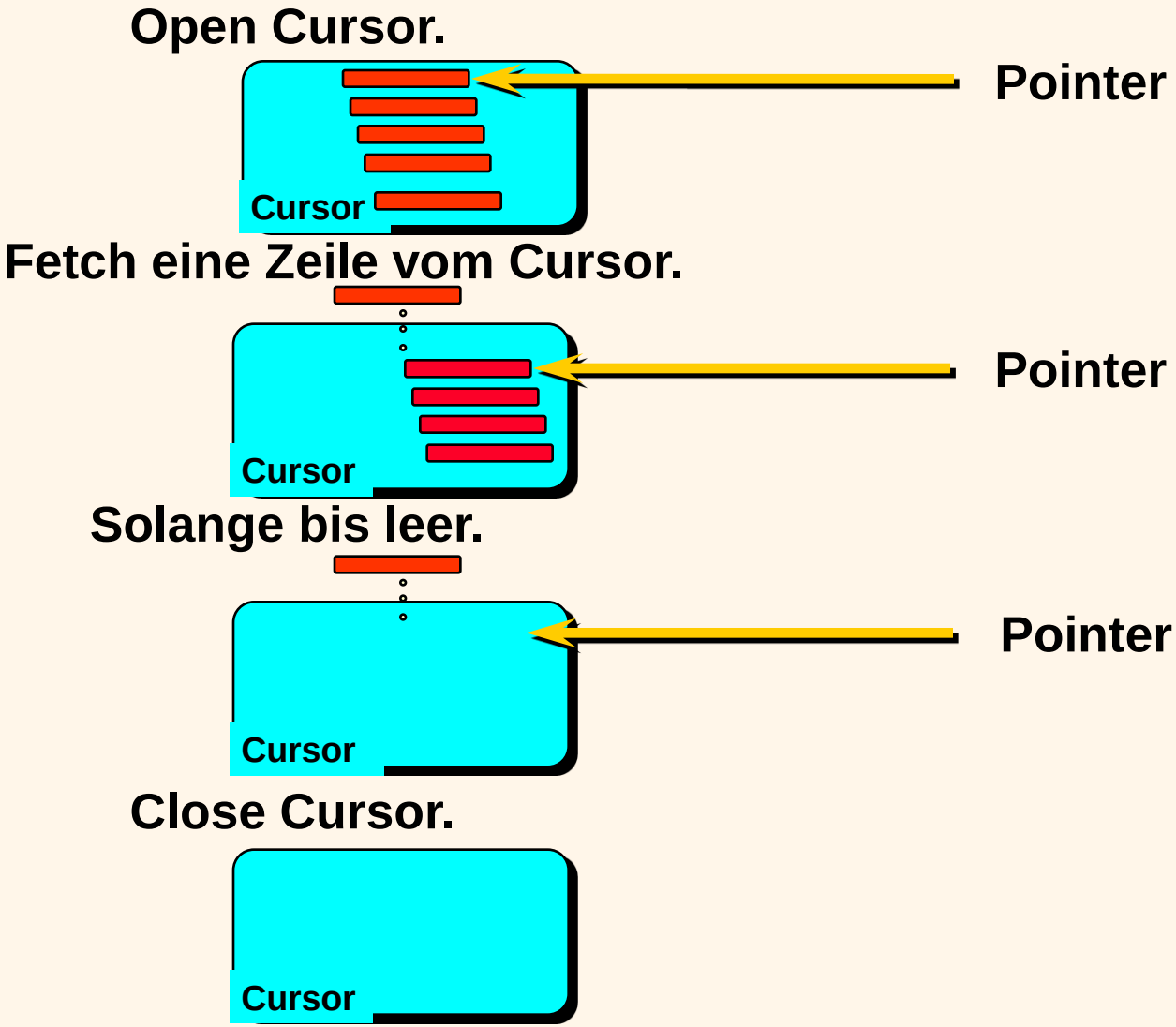
- Jeder SQL-Befehl, der vom Oracle-Server ausgeführt wird, ist mit einem individuellen Cursor assoziiert.
- Ein Cursor ist ein privater SQL Arbeitsbereich.
- Es gibt zwei Typen von Cursorn:
  - Implizite Cursor
  - Explizite Cursor
- Oracle Server nutzt implizite Cursor, um SQL-Befehle (SELECT, PL/SQL DML) zu übersetzen und auszuführen.
- Explizite Cursor werden explizit durch den Programmierer benannt und deklariert.

Kontrolle expliziter Cursor



- Erzeuge benannte SQL Area
- Identifiziere die aktive Menge
- Lade die aktuelle Zeile in Variablen
- Teste auf vorhandene Zeilen
- Freigabe der aktiven Menge

Kontrolle expliziter Cursor



Deklaration eines Cursors

Syntax

```
CURSOR cursor_name IS
      select_statement;
```

- keine INTO Klausel in der Cursor-Deklaration.
- Wenn Verarbeitung der Zeilen in bestimmter Reihenfolge erforderlich, ORDER BY Klausel verwenden

Beispiel

```
DECLARE
  CURSOR emp_cursor IS
    SELECT empno, ename
    FROM   emp;

  CURSOR dept_cursor IS
    SELECT *
    FROM   dept
    WHERE  deptno = 10;
BEGIN
  ...
```

Öffnen eines Cursors

Syntax

```
OPEN  cursor_name;
```

- Öffnen des Cursors, um die Query auszuführen und die aktive Menge zu identifizieren.
- Wenn die Query keine Zeilen zurückliefert, wird eine Exception erzeugt.
- Nutze Cursor-Attribute, um das Ergebnis eines Fetch zu testen.

Daten lesen vom Cursor

Syntax

```
FETCH cursor_name INTO  [variable1, variable2, ...]
                        | record_name];
```

- Lese die aktuellen Zeilenwerte in Ausgabevariablen („Fetch“)
- Gleiche Anzahl von Variablen wie Spalten
- Jede Variable muss mit den Spalten von der Position her übereinstimmen
- Test, ob der Cursor Zeilen enthält

Daten lesen vom Cursor (Beispiel)

Beispiele

```
FETCH emp_cursor INTO v_empno, v_ename;
```

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;
```

Schließen eines Cursors

Syntax

```
CLOSE    cursor_name;
```

- Schließe den Cursor nach beendeter Verarbeitung der Zeilen
- Re-Open des Cursors wenn erforderlich
- nach Schließen eines Cursors können Daten nicht mehr gefetcht werden.

Explizite Cursor-Attribute

Einholen von Status-Informationen über einen Cursor

Attribut	Typ	Beschreibung
%ISOPEN	Boolean	Ist gleich TRUE wenn der Cursor geöffnet ist
%NOTFOUND	Boolean	Ist gleich TRUE wenn der letzte Fetch keine Zeile geliefert hat
%FOUND	Boolean	Ist gleich TRUE wenn der letzte Fetch eine Zeile geliefert hat; Komplement zu %NOTFOUND
%ROWCOUNT	Number	Gesamtanzahl Zeilen, die bisher zurückgegeben wurden

%ISOPEN Attribut

- Fetch von Zeilen nur möglich, wenn Cursor geöffnet ist
- Nutze %ISOPEN Cursor-Attribut vor Ausführung eines Fetch, um zu testen, ob der Cursor geöffnet ist

Beispiel:

```
IF NOT emp_cursor%ISOPEN THEN
    OPEN emp_cursor;
END IF;
LOOP
  FETCH emp_cursor...
```

Cursor und Records

Verarbeitung der Zeilen einer aktiven Menge durch Lesen der Werte (Fetch) in einen PL/SQL RECORD.

Beispiel

```
DECLARE
  CURSOR emp_cursor IS
    SELECT  empno, ename
    FROM    emp;
  emp_record   emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
  ...
```

FOR Schleifen mit Cursor

- Syntax

```
FOR record_name IN cursor_name LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

  - FOR Schleife mit Cursor ist eine Abkürzung, um explizite Cursor zu verarbeiten
  - Implizit: Open, Fetch und Close des Cursor
  - Record ist implizit deklariert

FOR-Schleifen mit Cursor (Beispiel)

Lese Angestellte nacheinander, bis keiner mehr übrig ist.

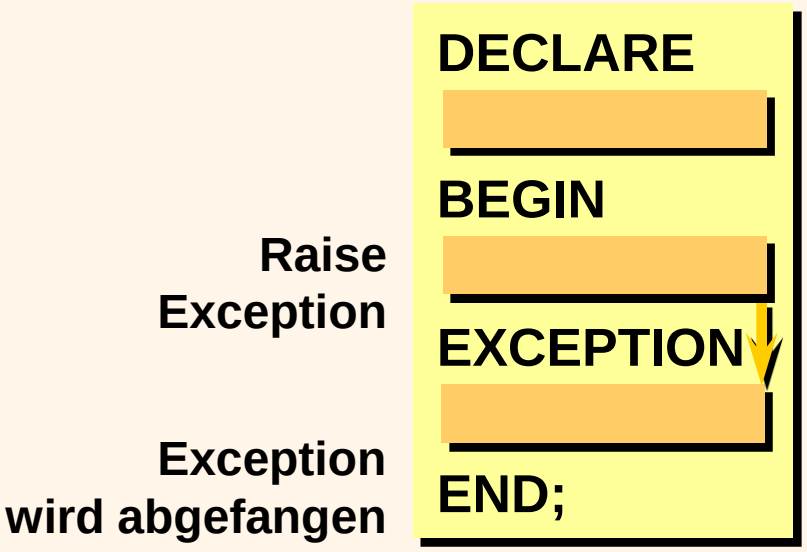
```
DECLARE
  CURSOR emp_cursor IS
    SELECT ename, deptno
    FROM   emp;
BEGIN
  FOR emp_record IN emp_cursor LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.deptno = 30 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

Ausnahmebehandlung in PL/SQL

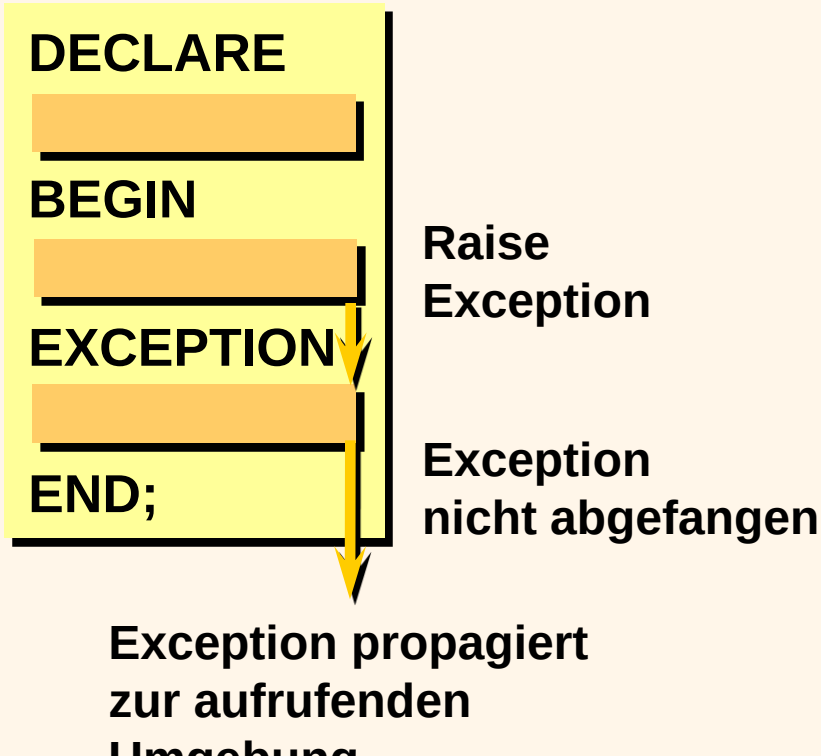
- Was ist eine Exception?
  - Identifier in PL/SQL, der erzeugt wird während der Ausführung
- Wie wird sie erzeugt?
  - Ein Oracle-Fehler tritt auf
  - Explizit erzeugen (benutzerdefinierte Ausnahme)
- Wie behandeln?
  - Abfangen mit einem Handler
  - Propagieren zur Aufrufumgebung

Behandlung von Ausnahmen

Exception abfangen



Exception propagieren



Abfangen von Exceptions

Syntax

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

Abfangen von Oracle-Fehlern

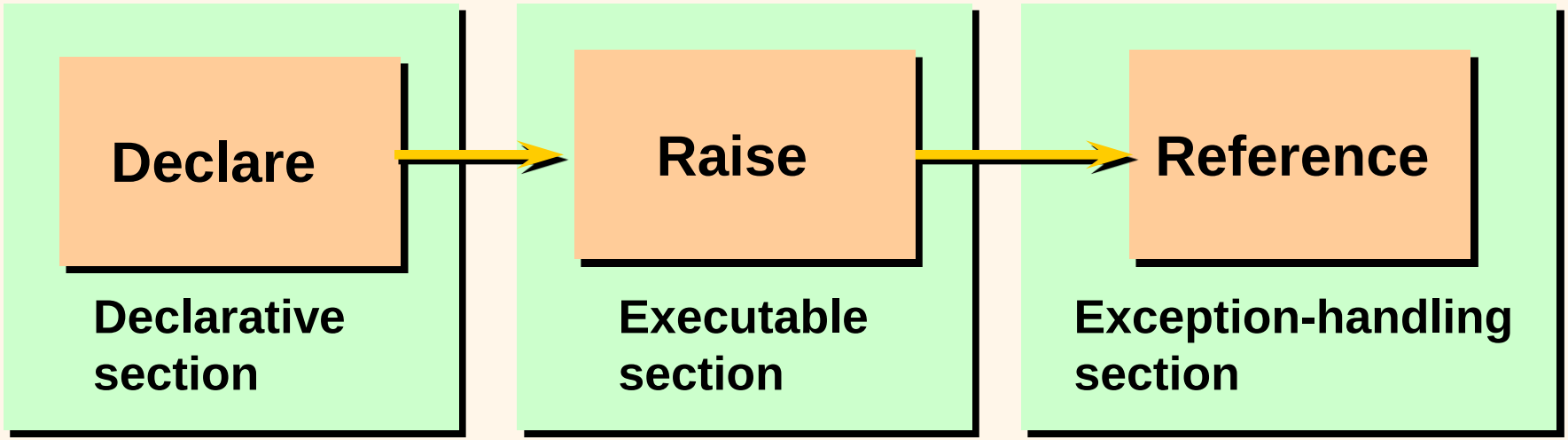
- Standardnamen in Exception-Handling Routine verwenden
- Beispiele vordefinierter Exceptions:
  - NO_DATA_FOUND: SELECT liefert keine Daten
  - TOO_MANY_ROWS: Liefert mehrere Zeilen, nur eine erlaubt
  - INVALID_CURSOR: Illegale Cursor-Operation
  - ZERO_DIVIDE: Division durch 0
  - DUP_VAL_ON_INDEX: Versuch, ein Duplikat einzufügen

Vordefinierte Exception

Syntax

```
BEGIN  SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```

Abfangen benutzerdefinierter Exceptions



- Benenne Exception
- Erzeuge die Exception explizit mit RAISE-Statement
- Behandele die erzeugte Exception

Benutzerdefinierte Exceptions (Beispiel)

```
DECLARE
   e_invalid_product  EXCEPTION;
BEGIN
   UPDATE      product
   SET         descrip = '&product_description'
   WHERE       prodid = &product_number;
   IF SQL%NOTFOUND THEN
   RAISE e_invalid_product;
   END IF;
   COMMIT;
EXCEPTION
   WHEN  e_invalid_product  THEN
     DBMS_OUTPUT.PUT_LINE('Invalid product number.');
END;
```

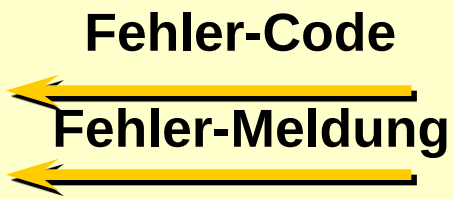
1

2

3

Funktionen zum Abfangen von Exceptions (Beispiel)

```
DECLARE
  v_error_code       NUMBER;
  v_error_message    VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQLCODE ;
    v_error_message := SQLERRM ;
    INSERT INTO errors VALUES(v_error_code,
                              v_error_message);
END;
```



RAISE_APPLICATION_ERROR

Syntax

```
raise_application_error (error_number,
                         message[, {TRUE | FALSE}]);
```

Prozedur zur Ausgabe benutzerdefinierter Fehlermeldungen aus gespeicherten Subprogrammen

- genutzt an 2 Stellen: Executable Section, Exception Section
- liefert Fehlermeldungen, die wie Oracle-Fehler aussehen