ISYS 365 -
Built-in Functions
Agenda

- Review
  - Stored Procedures
  - Functions
  - Parameters
  - Calling Stored Procedures & Functions
- Built-in Functions
- Examples
Anonymous PL/SQL Block

DECLARE

-- variable declaration goes here

BEGIN

-- required executable goes here

EXCEPTION

-- exception handling goes here

END;
/

* Can return more than one value to the calling program

CREATE OR REPLACE PROCEDURE X

[(formal_parameters)] AS[IS]

BEGIN

-- variable declaration goes here

-- required executable goes here

EXCEPTION

-- exception handling goes here

END X;
/

* Stored Procedure
Stored Procedures vs. Functions

**Stored Procedure**

```sql
CREATE OR REPLACE PROCEDURE X
  [(parameters)] AS

  -- variable declaration goes here

BEGIN

  -- required executable goes here

EXCEPTION

  -- exception handling goes here

END X;
/
```

**Function**

```sql
CREATE OR REPLACE FUNCTION X
  [(formal_parameters)] RETURN
return_type IS[AS]

  -- variable declaration goes here

BEGIN

  -- required executable goes here

  /* required RETURN statement to
  return one value */

  RETURN Z;

EXCEPTION

  -- exception handling goes here

END X;
/
```
Parameters

- MUST be given a data type, but must NOT be given a size

- 3 Modes
  - IN
  - OUT
  - IN OUT

- Can pass parameters by reference or by value

- NOCOPY pass parameters by reference

- Formal parameters with default values must appear as the last items in the parameter list

- When calling a stored procedure or function, the actual arguments can be passed by positional or named notation
Calling Stored Procedures & Functions

With Parameters
- Stored Procedure from SQL> prompt
  - CALL X(v_Variable1, ..., v_VariableN);
  - OR CALL X(p_Parameter1 => v_Variable1,...);
  - EXEC X(v_Variable1,...,v_VariableN);
- Stored Procedure from within PL/SQL block
  - EXECUTE IMMEDIATE ‘CALL X(……..)’;     OR
  - X(v_Variable1, ..., v_VariableN);
- Function
  - Used in an expression in SELECT and WHERE clauses
    - SELECT ElapsedDays(‘01-JAN-1999’) FROM dual;

Without Parameters
- If the stored procedure (or function) does not have parameters, then do not use parentheses to define or call
  the stored procedure (or function)
Built-in Functions

**SUBSTR(string, start, count)**

SELECT DISTINCT dname, SUBSTR(dname, 1, 4) FROM dept;

<table>
<thead>
<tr>
<th>DNAME</th>
<th>SUBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNTING</td>
<td>ACCO</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>OPER</td>
</tr>
<tr>
<td>RESEARCH</td>
<td>RESE</td>
</tr>
<tr>
<td>SALES</td>
<td>SALE</td>
</tr>
</tbody>
</table>

**SOUNDEX(string)**

SELECT  workerID, lname FROM worker
WHERE  SOUNDEX(lname) = SOUNDEX('HENASEE');

<table>
<thead>
<tr>
<th>WORKERID</th>
<th>LNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>HENNESSEY</td>
</tr>
</tbody>
</table>
Built-in Functions

**DECODE(fieldname, if1, then1, if2, then2,..., else)**

```sql
SELECT branchID, bdname, DECODE(branchID, 122, 'NY', 123, 'TX', 124, 'IL', 167, 'MA', NULL) AS STATE FROM branch;
```

<table>
<thead>
<tr>
<th>BRANCHID</th>
<th>NAME</th>
<th>STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>122</td>
<td>NEW YORK</td>
<td>NY</td>
</tr>
<tr>
<td>123</td>
<td>DALLAS</td>
<td>TX</td>
</tr>
<tr>
<td>124</td>
<td>CHICAGO</td>
<td>IL</td>
</tr>
<tr>
<td>167</td>
<td>BOSTON</td>
<td>MA</td>
</tr>
</tbody>
</table>
**Built-in Functions**

**LENGTH(string)**

```sql
SELECT dname, LENGTH(dname) AS len FROM dept;
```

<table>
<thead>
<tr>
<th>DNAME</th>
<th>LEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNTING</td>
<td>10</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>10</td>
</tr>
<tr>
<td>RESEARCH</td>
<td>8</td>
</tr>
<tr>
<td>SALES</td>
<td>5</td>
</tr>
</tbody>
</table>

Note this assumes dname was declared as VARCHAR2 type. When CHAR(X) is used, LENGTH function always returns X.
Built-in Functions

**TRIM(string)**
Removes leading and trailing blanks

**TO_CHAR(date, ‘format’)**
See Table 5-4 for a list of valid formats

The date field in the reservation table has been populated, but the weekday field is NULL.

Write a query to populate the weekday field with the name of the day that corresponds to the date specified in the date field.

```
UPDATE reservation SET weekday = TRIM(TO_CHAR(date, ‘DAY’));
```

**NOTE:** The ‘DAY’ format returns the name of the day with blanks padded on the right such that the length is 9 characters.
Other Built-in Date Functions

- **SYSDATE**
- **ADD_MONTHS**(date, count)
- **MONTHS_BETWEEN**(date2, date1)
- **NEXT_DAY**(date, ‘day’)
- **LAST_DAY**(date)
- **GREATEST**(date1, date2, date3)
- **LEAST**(date1, date2, date3)
- ...
- **AND MANY, MANY MORE…**
# Date Arithmetic

For allowable Date & Datetime arithmetic, See Table 5.3 in the PL/SQL Programming book

<table>
<thead>
<tr>
<th>Operation</th>
<th>Return Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1 - d2</td>
<td>NUMBER</td>
<td># of days between d1 &amp; d2</td>
</tr>
<tr>
<td>d1 + n</td>
<td>DATE</td>
<td>Adds n days to d1, resulting in a new date</td>
</tr>
<tr>
<td>d1 - n</td>
<td>DATE</td>
<td>Subtracts n days from d1, resulting in a new date</td>
</tr>
</tbody>
</table>
Function Exercise 1: ElapsedDays

- Write a function called **ElapsedDays** that calculates the number of days that have elapsed between a given date and the system date…
  - The function has one date parameter that cannot be changed
  - The return data type is NUMBER
Function Exercise2: Elapsed2Dates

- Create a function that returns the number of days that has elapsed between two dates...
  - The function takes two read-only date parameters: the due date and the current date, which has a default value of the system date.
  - If the difference between the two dates is negative then raise an exception.
  - Handle any other type of exceptions that can occur as follows:
    - Display the error code and the first 200 characters of the error message.
Function Exercise 3: Decode

Rewrite the following function that calculates the next business date with DECODE function

- The function takes one read-only date parameter
- Business days are Mondays through Fridays. Saturdays and Sundays are not business days.
  - If the day that corresponds to the date parameter is a Friday, then the next business date is 3 days later (i.e. Monday).
  - If the day that corresponds to the date parameter is Saturday, then the next business date is 2 days later (i.e. Monday).
  - Otherwise, the next business date is 1 day later.
CREATE OR REPLACE FUNCTION NextBusinessDate1 (p_Date DATE) RETURN DATE IS

-- Variable that will contain the day that corresponds to the date parameter
v_CurrentDay VARCHAR2(9);

-- Variable that will contain the computed date of the next business day
v_NextDate DATE;

BEGIN

/* First, determine the corresponding name of the day for the date parameter. It will be used later to determine the number of days by which the date should be incremented.*/

v_CurrentDay := UPPER(TRIM(TO_CHAR(p_Date, 'DAY')));

/* Based upon the name of the day and the business rule, calculate the next business date*/

IF v_CurrentDay = 'FRIDAY' THEN
    v_NextDate := p_Date + 3;
ELSIF v_CurrentDay = 'SATURDAY' THEN
    v_NextDate := p_Date + 2;
ELSE
    v_NextDate := p_Date + 1;
END IF;

-- Now, return the computed next business date to the calling program
RETURN v_NextDate;

END NextBusinessDate1;
Suppose we have the following table:

PRICES (Product_id, list_price, end_date)

Write a stored procedure that updates the list_price in the prices table where the end_date is NULL.

The increased percent is passed by a parameter

Uses a WHILE loop to fetch data into an implicit record

Display the product id, old list price and new list price to the output screen
Stored Procedure Example #1 with Exceptions  
(see ProcedureExample01_Exceptions.sql)

To the previous example, add the following:

- Exceptions
  - Raise an exception if the increase is a negative number
  - Raise an exception if the new price is more than 50% above the old price
- Display the number of rows affected
  - Display the product_ID and updated price
Stored Procedure Example #2

- Write a stored procedure that...

  - Defines a cursor that uses a bind variable that points to a record set that contains the sailors’ IDs, sailors’ names, reservation day and boat id where the boat color is red
  - Uses a WHILE loop to fetch data into an implicit record
  - Use an implicit cursor to select the rating of the sailor
  - Displays each last name, rating, reservation date and boat id for each record to the output screen
  - Uses exception handles

<table>
<thead>
<tr>
<th>Boat</th>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td></td>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sailor</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Brutus</td>
<td>1</td>
<td>33.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Andy</td>
<td>8</td>
<td>25.5</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>10</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>Horatio</td>
<td>7</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>Zorba</td>
<td>10</td>
<td>16.0</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>Horatio</td>
<td>9</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>Art</td>
<td>3</td>
<td>25.5</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reserve</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/98</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>102</td>
<td>10/10/98</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>103</td>
<td>10/8/98</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>104</td>
<td>10/7/98</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>11/10/98</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>11/6/98</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>104</td>
<td>11/12/98</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>101</td>
<td>9/5/98</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>102</td>
<td>9/8/98</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>103</td>
<td>9/8/98</td>
<td></td>
</tr>
</tbody>
</table>

19
Example #3: Stored Procedure

- Create a stored procedure that adds records to a reservation table, including the due date for returning the rental...
  
  - The stored procedure takes the following read-only parameters: customerID, productID and reservation date.
  
  - Rentals are due back the following business day (M-F).
  
  - If the customerID is valid, then add the reservation (i.e. customerID, productID, reservation date and due date) to the reservation table.
  
  - Raise exceptions when appropriate using the RAISE_APPLICATION_ERROR function.
Agenda

- Triggers
  - Kinds of triggers
  - Firing triggers
  - Characteristics of triggers
  - Order of firing triggers
  - Syntax
  - Trigger Predicates
  - Naming Convention
  - Example
Triggers

- A stored procedure w/o parameters
- Automatically fired with DML commands
- Why use triggers?
  - Another way to implement Business Rules
- Three kinds of triggers…
  - DML (INSERT, UPDATE and/or DELETE)
  - Instead-of triggers
    - Used with views only
    - Will execute instead of the DML statement
  - System triggers
    - Fires when a system event (action) occurs
Firing Triggers

- Firing the trigger
  - The act of executing a trigger

- They are implicitly executed whenever the trigger event happens (e.g. INSERT, UPDATE or DELETE)
Characteristics of Triggers

Three characteristics:

- **Timing**
  - Refers to whether the trigger fires BEFORE or AFTER the triggering action

- **Statement (type of action/type of event)**
  - INSERT, UPDATE and/or DELETE

- **Level**
  - Refers to statement-level (default) or row-level trigger
Characteristics of triggers

**Statement-level triggers**
- Fire once per SQL statement
- Cannot read the values of the columns

**Row-level triggers**
- Fire many times, once for each row affected
- Can evaluate the values of each column for that row (using correlation values)
Triggers: Order of Trigger Firing

- Execute the before statement-level triggers
- For each row affected by the statement:
  - Execute the before row-level triggers
  - Execute the statement itself
  - Execute the after row-level triggers
- Execute the after statement-level triggers
Triggers: Syntax

CREATE OR REPLACE TRIGGER trigger_Name

{BEFORE|AFTER} {INSERT|UPDATE| DELETE}

[OF column_name] ON table_name

[FOR EACH ROW]

[WHEN trigger_condition]

DECLARE

    -- variable declaration goes here

BEGIN

    -- statements goes here

END [trigger_Name];
Triggers: Predicates

CREATE OR REPLACE TRIGGER trigger_Name
  {BEFORE|AFTER} {INSERT|UPDATE| DELETE}
  [OF column_name] ON table_name
  [FOR EACH ROW]
  [WHEN trigger_condition]
DECLARE
  -- variable declaration goes here
BEGIN
  IF INSERTING THEN
    -- statements here
  ELSIF UPDATING THEN
    -- statements here
  ELSIF DELETING THEN
    -- statements here
  END IF;
END [trigger_Name];
Triggers: Naming Convention

- **Table_Name_[A|B][I|U|D][S|R]**
  - **[A|B]**    AFTER or BEFORE
  - **[I|U|D]** INSERT, UPDATE, DELETE
  - **[S|R]** statement-level or row-level
# Naming Conventions

<table>
<thead>
<tr>
<th>Item</th>
<th>Naming Convention</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary keys</td>
<td>*_pk</td>
<td>*= tablename</td>
</tr>
<tr>
<td></td>
<td>*_fk1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*_fk2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*_fk#</td>
<td>*= tablename; # = a sequential number</td>
</tr>
<tr>
<td>foreign keys</td>
<td>*_u1</td>
<td>*= tablename</td>
</tr>
<tr>
<td></td>
<td>*_u2</td>
<td>*= tablename</td>
</tr>
<tr>
<td></td>
<td>*_u#</td>
<td># represents a sequential number</td>
</tr>
<tr>
<td>unique keys</td>
<td>*_ck1</td>
<td>*= tablename</td>
</tr>
<tr>
<td></td>
<td>*_ck2</td>
<td>*= tablename</td>
</tr>
<tr>
<td></td>
<td>*_ck#</td>
<td># represents a sequential number</td>
</tr>
<tr>
<td>checks</td>
<td>*_sequence</td>
<td>*= field name</td>
</tr>
<tr>
<td>sequences</td>
<td>*_sequence</td>
<td>*= field name</td>
</tr>
<tr>
<td>script files</td>
<td>*.sql</td>
<td>*= can be any name you choose</td>
</tr>
<tr>
<td>spooled files</td>
<td>*.lst</td>
<td>*= can be any name you choose (e.g. TEST.LST)</td>
</tr>
<tr>
<td>cursors</td>
<td>c_</td>
<td></td>
</tr>
<tr>
<td>exceptions</td>
<td>e_</td>
<td></td>
</tr>
<tr>
<td>records</td>
<td>t_</td>
<td>Explicit records</td>
</tr>
<tr>
<td>variables</td>
<td>v_</td>
<td></td>
</tr>
<tr>
<td>parameters</td>
<td>p_</td>
<td></td>
</tr>
<tr>
<td>triggers</td>
<td>*__{B</td>
<td>A}{I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l=Insert, u=Update, d=Delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S=Statement-level, R=row-level</td>
</tr>
</tbody>
</table>
Example

/* Log new employees(inserts) or updates to employee salary */
CREATE OR REPLACE TRIGGER employee_BIUR
BEFORE INSERT OR UPDATE OF sal ON employee
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO New_Employee
    VALUES (SYSDATE,:NEW.empID,:NEW.sal);
  ELSE  -- updating
    INSERT INTO Employee_Sal_Change
    VALUES (SYSDATE,:OLD.empID,:OLD.sal,:NEW.sal);
  END employee_BIUR;

Example

CREATE OR REPLACE TRIGGER Update_Dept_Stat_BUIDR
BEFORE INSERT OR UPDATE OR DELETE ON EMPLOYEE
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    UPDATE Dept_Stat SET Total_emps = Total_emps + 1
    WHERE DNAME = v_new_deptname;
  END IF;
  IF DELETING THEN
    UPDATE Dept_Stat SET Total_emps = Total_emps - 1
    WHERE DNAME = v_old_deptname;
  END IF;
  IF UPDATING THEN
    IF :OLD.DNO <> :NEW.DNO THEN
      UPDATE Dept_Stat SET Total_emps = (Total_emps - 1)
      WHERE DNAME = v_old_deptname;
      UPDATE Dept_Stat SET Total_emps = (Total_emps + 1)
      WHERE DNAME = v_new_deptname;
    END IF;
  END IF;
END Update_Dept_Stat_BUIDR;
CREATE OR REPLACE TRIGGER student_AIUDS
    AFTER INSERT OR UPDATE OR DELETE ON student
DECLARE
    CURSOR c_Statistics IS
        SELECT major, COUNT(*) total_students, SUM(current_credits)
        total_credits
        FROM students
        GROUP BY major;
BEGIN
    DELETE * FROM major_stats;
    FOR v_StatsRecord in c_Statistics LOOP
        INSERT INTO major_stats VALUES(v_StatsRecord.major,
        v_StatsRecord.total_credits, v_StatsRecord.total_students);
    END LOOP;
END;
Show Errors

- Retrieving compile errors from the DD view USER_ERRORS

SHOW ERRORS TRIGGER emp_bir