# Semester 2 Final Review

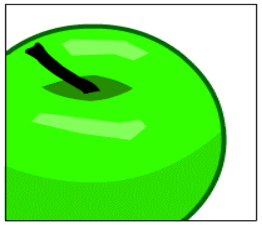
# Semester 2 Final Review

**This slide set contains the following topics:**

- **Creating DDL and Database Event Triggers**
- **Managing Triggers**
- **Using Large Object Data Types**
- **Managing BFILES**
- **User-Defined Records**
- **Index by Tables**
- **Understanding Dependencies**

# Creating DDL and Database Event Triggers

## DDL and Database Event Triggers



DDL Triggers are fired by DDL statements: `CREATE`, `ALTER` or `DROP`.

Database Event triggers are fired by non-SQL events in the database, for example:

- A user connects to, or disconnects from, the database
- The DBA starts up, or shuts down, the database
- A specific exception is raised in a user session.

## Creating Triggers on DDL Statements

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
Timing
[ddl_event1 [OR ddl_event2 OR ...]]
ON {DATABASE|SCHEMA}
trigger_body
```

- `ON DATABASE` will fire the trigger for DDL on all schemas in the database
- `ON SCHEMA` will fire the trigger only for DDL on objects in your own schema

## Example of a DDL Trigger:

We want to write a log record every time a new database object is created in our schema:

```
CREATE OR REPLACE TRIGGER log_create_trigg
AFTER CREATE ON SCHEMA
BEGIN
  INSERT INTO log_table
    VALUES (USER, SYSDATE);
END;
```

The trigger will fire whenever any (type of) object is created.  We cannot create a DDL trigger which refers to a specific database object.

## Examples of Database Event Triggers

1. LOGON and LOGOFF triggers:

```
CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON  ON  SCHEMA
BEGIN
 INSERT INTO log_trig_table(user_id,log_date,action)
 VALUES (USER, SYSDATE, 'Logging on');
END;
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF  ON  SCHEMA
BEGIN
 INSERT INTO log_trig_table(user_id,log_date,action)
 VALUES (USER,  SYSDATE,  'Logging off');
END;
```

## Examples of Database Event Triggers

2. A SERVERERROR Trigger:

We want to keep a log of any ORA-00942 errors which occur in our sessions:

```
CREATE OR REPLACE TRIGGER servererror_trig
AFTER SERVERERROR ON SCHEMA
BEGIN
  IF (IS_SERVERERROR (942)) THEN
    INSERT INTO error_log_table ...
  END IF;
END;
```

## CALL Statements

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing
event1 [OR event2 OR event3]
ON table_name
[REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
[WHEN condition]
CALL procedure_name
```

```
CREATE OR REPLACE TRIGGER log_employee
BEFORE INSERT ON EMPLOYEES
CALL log_execution
```

Note: There is no `END;` statement, and no semicolon at the end of the `CALL` statement.

## Mutating Tables and Row Triggers

A mutating table is a table which is currently being modified by a DML statement.

A row trigger cannot `SELECT` from a mutating table, because it would see an inconsistent set of data (the data in the table would be changing while the trigger was trying to read it).

However, a row trigger can SELECT from a different table if needed.

This restriction does not apply to DML statement triggers, only to DML row triggers.

## Mutating Table: Example

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE INSERT OR UPDATE OF salary, job_id ON employees
  FOR EACH ROW
DECLARE
  v_minsalary employees.salary%TYPE;
  v_maxsalary employees.salary%TYPE;
BEGIN
  SELECT MIN(salary), MAX(salary)
   INTO  v_minsalary, v_maxsalary
   FROM  employees
   WHERE job_id = :NEW.job_id;
  IF :NEW.salary < v_minsalary OR
     :NEW.salary > v_maxsalary THEN
     RAISE_APPLICATION_ERROR(-20505,'Out of range');
  END IF;
END;
```

A runtime error will occur because the trigger is SELECTing from the same table as the triggering DML statement is referencing.

## Mutating Table: Example

```
UPDATE employees
  SET salary = 3400
  WHERE last_name = 'Davies';
```

```
ORA-04091: table USVA_TEST_SQL01_T01.EMPLOYEES is mutating, trigger/function may not
see it
ORA-06512: at “USVA_TEST_SQL01_T01.CHECK_SALARY”, line 5
ORA-04088: error during execution of trigger 'USVA_TEST_SQL01_T01.CHECK_SALARY'
3. WHERE last_name = 'Davies';
```

# Managing Triggers

## Privileges Needed for Triggers

To create a trigger in your own schema, you need:

- `CREATE TRIGGER` system privilege
- Normal object privileges (`SELECT, UPDATE, EXECUTE` etc.) on any objects in other schemas that are referenced in your trigger body
- `ALTER` privilege on the table or view associated with the trigger.

Statements in the trigger body use the privileges of the trigger owner (definer's rights), not the privileges of the user executing the operation that fires the trigger.

## Privileges Needed for Triggers: Example

User MOE needs to create the following trigger:

```
CREATE OR REPLACE TRIGGER upd_tom_emp
AFTER UPDATE ON tom.employees
BEGIN
  INSERT INTO mary.log_table VALUES(USER,SYSDATE);
  sharon.calledproc;
END;
```

Moe will need:

- CREATE TRIGGER
- ALTER on TOM.EMPLOYEES
- INSERT on MARY.LOG_TABLE
- EXECUTE on SHARON.CALLEDPROC.

## Viewing Trigger Information

You can view the following trigger information:

- `USER_OBJECTS` data dictionary view: object information
- `USER_TRIGGERS` data dictionary view: text of the trigger
- `USER_ERRORS` data dictionary view: PL/SQL syntax errors (compilation errors) of the trigger

## Using USER_TRIGGERS

| Column | Column Description |
|---|---|
| TRIGGER_NAME | Name of the trigger |
| TRIGGER_TYPE | The type is BEFORE, AFTER, INSTEAD OF |
| TRIGGERING_EVENT | The DML operation firing the trigger |
| TABLE_NAME | Name of the database table |
| REFERENCING_NAMES | Name used for :OLD and :NEW |
| WHEN_CLAUSE | The when_clause used |
| STATUS | The status of the trigger |
| TRIGGER_BODY | The action to take |

**Not all columns are shown here**

## Listing the Code of Triggers

```
SELECT trigger_name, trigger_type, triggering_event,
       table_name, status, trigger_body
  FROM   USER_TRIGGERS
  WHERE  trigger_name = 'RESTRICT_SALARY';
```

| TRIGGER_NAME | TRIGGER_TYPE | TRIGGERING_EVENT | TABLE_NAME | STATUS | TRIGGER_BODY |
|---|---|---|---|---|---|
| RESTRICT_SALARY | BEFORE EACH ROW | INSERT OR UPDATE | EMPLOYEES | ENABLED | BEGIN IF NOT (:NEW job_id IN ('AD_PRES', 'AD_VP')) AND :NEW.salary > 15000 THEN RAISE_APPLICATION_ERROR (-20202, 'Employee cannot earn more than $15,000'); END IF; END; |

## Managing Triggers

- Disable or enable a database trigger:

```
ALTER TRIGGER trigger_name DISABLE | ENABLE;
```

- Disable or enable all triggers for a table:

```
ALTER TABLE table_name DISABLE | ENABLE
  ALL TRIGGERS;
```

- Recompile a trigger for a table:

```
ALTER TRIGGER trigger_name COMPILE;
```

## Removing Triggers

To remove a trigger from the database, use the `DROP TRIGGER` statement:

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER secure_emp;
```

Note: All triggers on a table are removed when the table is removed.

## Using Large Object Data Types

## Large Object (LOB) Column Datatypes

- The Problem:
  - In SQL, the largest character column is 4096 bytes.
  - There is no specific datatype for MP3, JPEG, EXE etc.
  - What if there is a need to store an object bigger than 4KB?
- The Solution:
  - Large Objects (LOBs) address all of these problems.
  - They can store *ANYTHING* of any type.
  - A single LOB column value in a table can be up to 4GB.

## Two ways to store large objects: the old and the new way

- The Old Way
  - There are two deprecated data types: `LONG` and `LONG RAW`.
  - They should not be used any more.
- The New Way
  - LOBs come in three flavors: `CLOB, BLOB`, and `BFILE`.
  - `CLOB`: Character Large Objects, such as resumes, text articles, source code files.
  - `BLOB`: Binary Large Objects, such as sound (MP3), photos (JPEG, BMP), proprietary formats (PDF, DOC, XLS), and executables (EXE, DDL).
  - `BFILE`: Binary Files, just like `BLOB` but stored outside the database, often on separate media (CD, DVD, HD-DVD).

## The Old Way

- **LONG**
  - Up to 2GB
  - Replaced by CLOB
- **LONG RAW**
  - Up to 2GB
  - Replaced by BLOB

## The New Way

- **CLOB**
  - Up to 4GB
  - Replaces LONG
- **BLOB and BFILE**
  - Up to 4GB
  - Replace LONG RAW
  - Inside or outside the database

A table can contain only one LONG or LONG RAW column, but as many LOB columns as needed.

CLOB and BLOB data are stored in the database ("internal LOBs"). BFILEs are stored outside the database ("external LOBs") in separate files.

## Converting LONG to CLOB

You can convert `LONG` columns to `CLOB`s (and `LONG RAW` columns to `BLOB`s) using `ALTER TABLE`:

```
ALTER TABLE table_name
  MODIFY (long_col_name {CLOB | BLOB});
```

For example, to convert the `RESUMES` column in the `EMPLOYEES` table from `LONG` to `CLOB` you enter:

```
ALTER TABLE employees
  MODIFY (resumes CLOB);
```

## CLOB column

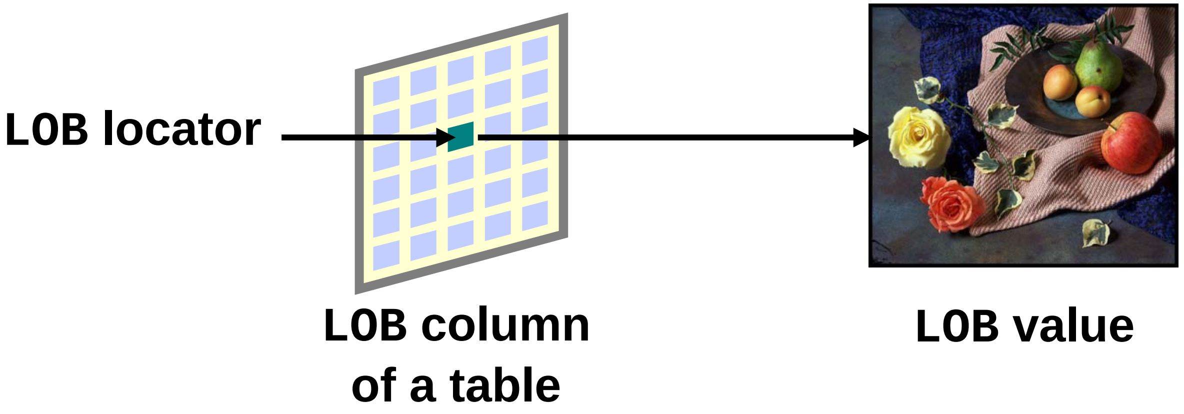
- Text only
- No fonts, no bold, no italic, no formatting, nothing fancy.
- Very useful for storing XML, HTML, DDL, PL/SQL, scripts and other source code for programs
- Can use all the built-in SQL character functions such as `SUBSTR, LENGTH`.

## BLOB column

- Can store absolutely anything (e.g. ZIP, EXE, DLL file types).
- A few file types such as PDF, BMP, GIF, JPEG, MP3, WAV (usually the ones known by web browsers) can be displayed by Oracle Application Express.

## How and where is LOB data stored?

Unlike other data types in which the column value is stored inline as part of the main row data, a LOB column value is stored in a separate area of the database with a pointer to it from the main table row.



We say that the LOB data value is stored **out-of-line**. The pointer from the main row is called a **locator**.

## Adding a LOB column to a table:

```
ALTER TABLE employees ADD (annual_evals CLOB);

ALTER TABLE employees ADD (badge_photo  BLOB);
```

Like any other newly added column, the LOB value is NULL. This means that the column data does not exist, and the locator is set to NULL.

## Adding CLOB data to a column:

Now we can insert the data values using normal DML statements:

```
UPDATE employees
  SET annual_evals = 'Evaluation Date: 14 September
         2005.  Performance Rating: Good ... '
  WHERE employee_id = 100;
```

```
INSERT INTO employees (employee_id, ..., annual_evals)
  VALUES (272, ..., 'New Employee; evaluation details will
                    be added later');
```

Each of these statements creates the out-of-line data values elsewhere in the database, and creates a locator value in the main table row which points to the data value.

## Reading CLOB data from the table:

We can SELECT a CLOB column like any other column:

```
SELECT annual_evals FROM employees
  WHERE employee_id = 100 ;
```

But these values can be very large. Reading the whole of a 4GB LOB value would take a long time and use a lot of memory. And maybe we only want to see part of the value anyway:

```
SELECT SUBSTR(annual_evals,2001,1000)
  FROM employees WHERE employee_id = 100;
```

## Updating CLOB data:

But we cannot use SQL functions like SUBSTR in UPDATE or INSERT statements:

```
UPDATE employees
  SET SUBSTR(annual_evals,2001,8) = 'NEW TEXT' -- ERROR
  WHERE employee_id = 100;
```

We manipulate large LOB values by using the DBMS_LOB PL/SQL package. And we cannot use DBMS_LOB directly in a SQL SELECT or DML statement. We can do it only from inside a PL/SQL block.

## Using DBMS_LOB, Step 1: Initializing the Locator:

When we use DBMS_LOB, the locator must be initialized before the data value can be stored. We do this using the EMPTY_CLOB( ) and EMPTY_BLOB( ) SQL functions:

```
UPDATE employees
  SET annual_evals = EMPTY_CLOB(),
      badge_photo  = EMPTY_BLOB();
```

EMPTY_CLOB( ) and EMPTY_BLOB( ) are built-in SQL functions, just like UPPER,  TO_CHAR and so on, except that they can only be used in DML statements because they modify the table. The functions allocate initial space elsewhere in the database to hold the data value, and update the locator to point to this space.

## Using DBMS_LOB, Step 2: Populating a CLOB column:

```
DECLARE
  v_lobloc   CLOB;   -- this will store the LOB locator
  v_text     VARCHAR2(32767);
  v_length   INTEGER;
  v_offset   INTEGER;
BEGIN
  SELECT annual_evals INTO v_lobloc FROM employees
  WHERE employee_id = 100 FOR UPDATE;
  FOR i IN 1..3 LOOP
    v_text   := 'The next piece of text number ' || i;
    v_offset := DBMS_LOB.GETLENGTH(v_lobloc)+ 2;
    v_length := LENGTH(v_text);
    DBMS_LOB.WRITE(v_lobloc,v_length,v_offset,v_text);
  END LOOP;
END;
```

## Reading BLOB column data using DBMS_LOB:

BLOB data cannot be displayed in Oracle Application Express, but we can see that the data exists by finding and displaying its length:

```
DECLARE
  CURSOR country_curs IS
    SELECT country_id, country_name, flag
      FROM wf_countries WHERE country_name LIKE 'A%';
  v_length      NUMBER;
BEGIN
  FOR country_rec IN country_curs LOOP
    v_length := DBMS_LOB.GETLENGTH(country_rec.flag);
    DBMS_OUTPUT.PUT_LINE(country_rec.country_id ||'  '
               ||country_rec.country_name||' '||v_length);
  END LOOP;
END;
```

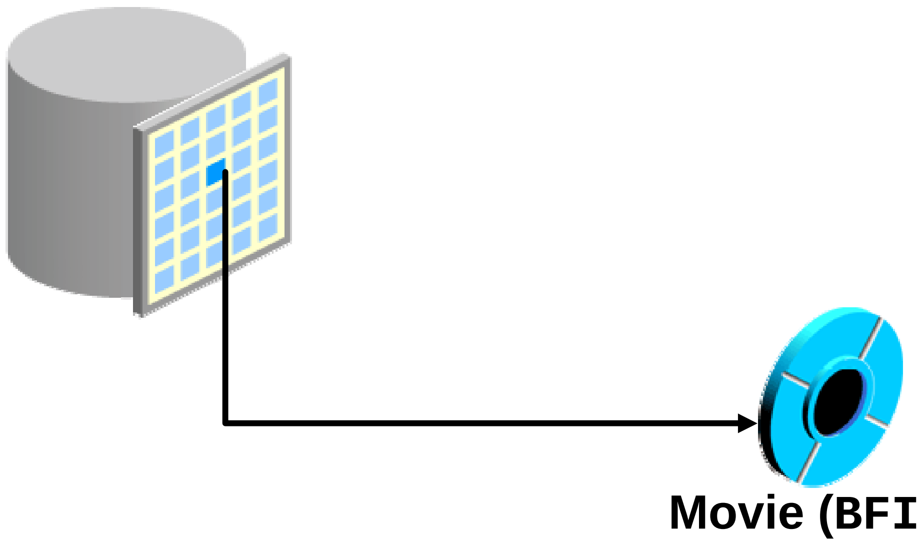
## Managing BFILES

## What Is a BFILE?

A BFILE is like a CLOB or BLOB, except that its value is stored outside the database in a separate file. The database holds a pointer to the external file.

The data can be text (like a CLOB) or multimedia (like a BLOB).

The external file can be on a normal computer disk, or on a CD or DVD etc. as long as it is accessible from the database.



## How is a BFILE different from CLOBs and BLOBs?

**Because the BFILE data is stored outside the database:**

- **It can be read but not modified, therefore it must be created outside Oracle**
- **It cannot have normal database privileges granted on it**
- **Normal SQL statements cannot be used on it.**
- **All data access is through the `DBMS_LOB` package**.

## When to use a BFILE?

- If the media already exists (e.g. CD or DVD), then why bother re-copying it into the database (into a BLOB column)?
- If the media is already read-only (e.g. CD or DVD), then the BFILE being read-only is not an issue.

## When *NOT* to use a BFILE?

- If it doesn’t fit! Oracle LOB values cannot be more than 4GB.
- How big is a DVD? Up to 4.7 GB.
- How big is a Blu-ray or HD-DVD? From 15 to 50 GB.

## A New Database Object: DIRECTORY

We need a way of specifying which operating system directories (folders) contain our BFILEs, and also controlling which Oracle users are allowed to read the BFILE data. To do this, we create a `DIRECTORY`.

A `DIRECTORY` is a pointer from the database to an operating system directory (folder) where BFILEs are stored.

## Creating and Managing Directories

Create a directory to point to an external location where BFILEs are stored, then allow everyone to use it:

```
CREATE DIRECTORY movie_dir   AS 'c:\mymovies';

GRANT READ ON DIRECTORY movie_dir  TO PUBLIC;
```

Later we move our movie files to a different location.  We must update the directory pointer:

```
ALTER DIRECTORY movie_dir   AS 'c:\latermovies';
```

ORACLE Academy

## Viewing Directories in the Data Dictionary

Unlike most database objects, directories do not belong to any schema, so there is no `USER_DIRECTORIES` dictionary view. Instead, we use `ALL_DIRECTORIES`:

```
SELECT directory_name, directory_path
  FROM all_directories;
```

| DIRECTORY_NAME | DIRECTORY_PATH |
|---|---|
| WF_FLAGS | /u02/webapps/oa1bprd_dir |

## Adding and Populating a BFILE column

We add the column like any other column:

```
ALTER TABLE employees ADD (movie BFILE);
```

Then we populate it with a locator value which points to a specific file in a specific directory, using the `DIRECTORY` pointer we created earlier and the inbuilt `BFILENAME` function, within a PL/SQL block.

## Adding and Populating a BFILE column continued

```
DECLARE
  v_locator       BFILE;
BEGIN
  v_locator := BFILENAME('MOVIE_DIR','titanic.avi');
  IF DBMS_LOB.FILEEXISTS(v_locator) = 1 THEN
    UPDATE employees SET movie = v_locator
      WHERE employee_id = 100;
  ELSE
    RAISE_APPLICATION_ERROR
      (-20210,'This BFILE does not exist');
  END IF;
END;
```

ORACLE Academy

## Reading BFILE Locator Values

We can read the locator value using `DBMS_LOB.FILEGETNAME:`

```
DECLARE
  v_locator        BFILE;
  v_directory      VARCHAR2(30);
  v_filename       VARCHAR2(50);
BEGIN
  SELECT movie INTO v_locator
    FROM employees WHERE employee_id = 100;
  DBMS_LOB.FILEGETNAME(v_locator,v_directory,v_filename);
  DBMS_OUTPUT.PUT_LINE(v_directory|| ' ' ||v_filename);
END;
```

## Reading BFILE Data Values

```
DECLARE
  v_locator       BFILE;
  v_raw_data      RAW(32767);
  v_offset        INTEGER := 1;
BEGIN
  SELECT movie INTO v_locator
    FROM employees WHERE employee_id = 100;
  DBMS_LOB.FILEOPEN(v_locator);
  FOR i IN 1..100 LOOP
    DBMS_LOB.READ(v_locator,32000,v_offset,v_raw_data);
    v_offset := v_offset + 32000;
  END LOOP;
  DBMS_LOB.FILECLOSE(v_locator);
END;
```

We cannot display the BFILE data in Application Express.

## User-Defined Records

## A Problem Scenario

The EMPLOYEES table contains eleven columns. We need to code a single-row SELECT in our PL/SQL subprogram.

```
CREATE OR REPLACE PROCEDURE query_one_emp
      (p_emp_id IN employees.employee_id%TYPE)  IS
  v_employee_id   employees.employee_id%TYPE;
  v_first_name    employees.first_name%TYPE;
  ... -- seven more scalar variables here !!
  v_manager_id    employees.manager_id%TYPE;
  v_department_id employees.department_id%TYPE;
BEGIN
  SELECT employee_id, first_name, ..., department_id
    INTO v_employee_id, v_first_name, ..., v_department_id
    FROM employees
    WHERE employee_id = p_employee_id;
EXCEPTION
  WHEN no_data_found THEN ...;
END;
```

## And how can we return the results to the calling environment?

```
CREATE OR REPLACE PROCEDURE query_one_emp
      (p_emp_id         IN  employees.employee_id%TYPE,
       p_first_name     OUT employees.first_name%TYPE,
       ... – seven more OUT parameters here !!
       p_manager_id     OUT employees.manager_id%TYPE,
       p_department_id  OUT employees.department_id%TYPE)
IS
  v_employee_id   employees.employee_id%TYPE;
  v_first_name    employees.first_name%TYPE;
  ...
```

Fortunately we don’t have to do all this.  Instead, we declare and use a PL/SQL record.

## Using a PL/SQL Record

```
CREATE OR REPLACE PROCEDURE query_one_emp
      (p_emp_id     IN  employees.employee_id%TYPE,
       p_emp_record OUT employees%ROWTYPE)
IS
BEGIN
  SELECT * INTO p_emp_record
    FROM employees
      WHERE employee_id = p_emp_id;
EXCEPTION
  WHEN no_data_found THEN ...;
END;
```

We can use %ROWTYPE with records just as we can with cursors.

## PL/SQL Records

A PL/SQL record is a **composite** data type consisting of a group of related data items stored as fields, each with its own name and data type. We can refer to the whole record by its name, and/or to individual fields by their names.

Using %ROWTYPE implicitly declares a record whose fields match the corresponding columns by name and data type.  We can reference individual fields by prefixing the field-name with the record-name:

```
...
   IF p_emp_record.salary > 25000 THEN
      RAISE_APPLICATION_ERROR
        (-20104,'This employee earns too much!');
    END IF;
...
```

## Defining Our Own Records

But what if our example procedure `SELECT`s from a join of several tables?

We can declare our own record structures containing any fields we like.  A PL/SQL record:

- must contain one or more components (fields) of any scalar or composite type
- is not necessarily  the same as rows in a database table
- can be assigned initial default values and can be constrained as `NOT  NULL`
- can be a component of another record (nested records)

## Creating a User-Defined PL/SQL Record

A record structure is a composite data type, just as DATE, VARCHAR2, NUMBER and so on are Oracle-defined scalar data types. We declare the type and then declare one or more variables of that type.

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

identifier   type_name;
```

*field_declaration* can be of any PL/SQL data type, including %TYPE, %ROWTYPE and RECORD.

## User-Defined PL/SQL Records: Example

```
DECLARE
  TYPE person_type IS RECORD
     (first_name  employees.first_name%TYPE,
      last_name   employees.last_name%TYPE,
      gender      VARCHAR2(6));
  TYPE employee_type IS RECORD
     (job_id      VARCHAR2(10),
      salary      NUMBER(8,2),
      person_data person_type);

  person_rec      person_type;
  employee_rec    employee_type;
BEGIN
  IF person_rec.last_name ... END IF;
  employee_rec.person_data.last_name := ...;
```

## Reusing PL/SQL Types for Multiple Records continued

```
DECLARE
  TYPE person_type IS RECORD
     (first_name  employees.first_name%TYPE,
      last_name   employees.last_name%TYPE,
      gender      VARCHAR2(6));

  employee_rec    person_type;
  vendor_rec      person_type;
  customer_rec    person_type;
  ...
```

Even though the type is based loosely on employees, there could be other records that are of the same type from different tables.

## Where Can Types and Records be Declared and Used?

They are composite variables, and can be declared anywhere that scalar variables can be declared: in anonymous blocks, procedures, functions, package specifications (global), package bodies (local), triggers, and so on.

Their scope and visibility follows the same rules as for scalar variables. For example, a type could be declared in a package specification. Records based on that type could be declared and used anywhere within the package, and also in the calling environment.

ORACLE Academy

## Where Can Types and Records be Declared and Used Cont

```
CREATE OR REPLACE PACKAGE pers_pack IS
  TYPE person_type IS RECORD
     (first_name  employees.first_name%TYPE,
      last_name   employees.last_name%TYPE,
      gender      VARCHAR2(6));
  PROCEDURE pers_proc (p_pers_rec OUT person_type);
END pers_pack;
CREATE OR REPLACE PACKAGE BODY pers_pack IS
  PROCEDURE pers_proc (p_pers_rec OUT person_type)
    IS
      v_pers_rec person_type;
    BEGIN
      SELECT first_name, last_name, 'Female'
        INTO v_pers_rec
        FROM employees WHERE employee_id = 101;
    p_pers_rec := v_pers_rec;
    END pers_proc;
END pers_pack;
```

## Visibility and Scope of Records: Example

Now invoke the package procedure from another PL/SQL block (it could be a Java, C or other language application):

```
DECLARE
  a_pers_rec       pers_pack.person_type;
BEGIN
  pers_pack.pers_proc(a_pers_rec);
  DBMS_OUTPUT.PUT_LINE
     (a_pers_rec.first_name || a_pers_rec.gender);
END;
```

## Index by Tables

## What is a Collection?

A **collection** is a set of occurrences of the same kind of data. For example, the set of all employees' last names. Or, the set of all department rows.

In PL/SQL, a collection is a type of composite variable, just like user-defined records and `%ROWTYPE`. There are two kinds of collection:

- an **INDEX BY TABLE**, which is based on a single field or column, for example on the last_name column of EMPLOYEES
- an **INDEX BY TABLE OF RECORDS**, which is based on a composite record type, for example on the whole `DEPARTMENTS` row.

Because collections are PL/SQL variables, their data is stored in a private memory area like any other PL/SQL variable.

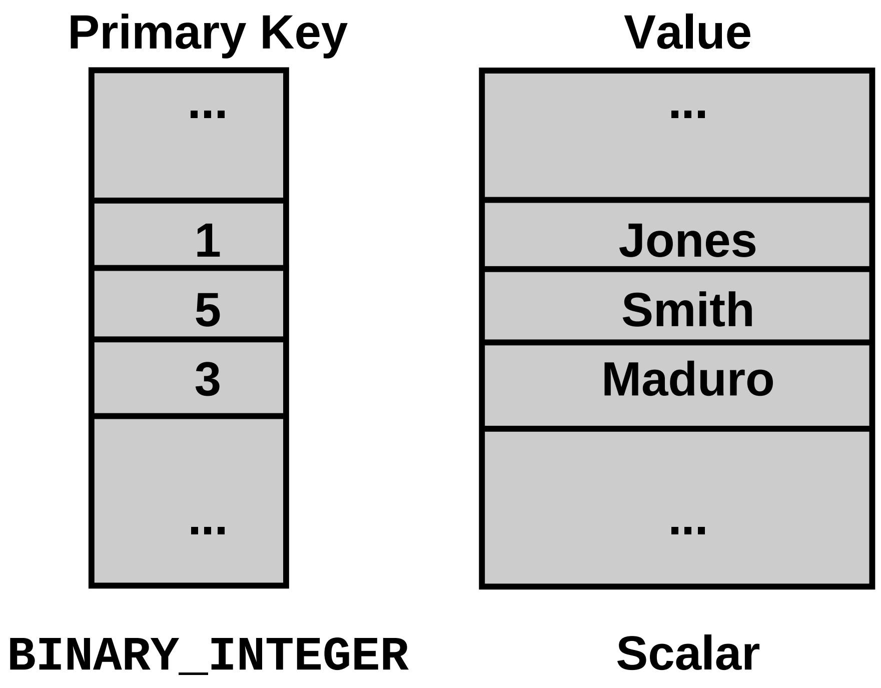
## An INDEX BY Table Has a Primary Key

We will need to able to distinguish between individual values in the table, so that we can reference them individually.  Therefore, every `INDEX  BY` table automatically has a numeric primary key, which serves as an index into the table.

The primary key must be of datatype `BINARY_INTEGER` (the default) or `PLS_INTEGER`.   The primary key can be negative as well as positive.

Since it is only an integer, some numeric data such as phone number may exceed the allowable precision.  For example 8005551212 won’t fit.

## INDEX BY Table Structure



The primary key must be numeric, but could be meaningful business data, for example an employee id.

## Declaring an INDEX BY Table

```
DECLARE
 TYPE t_names IS  TABLE OF VARCHAR2(50)
                  INDEX BY BINARY_INTEGER;

 last_names_tab   t_names;
 first_names_tab  t_names;
```

Like user-defined records, we must first declare a type and then declare “real” variables of that type.

This example declares two INDEX BY tables of the same type.

## Populating an INDEX BY Table:

```
DECLARE
  TYPE t_names IS  TABLE OF VARCHAR2(50)
                  INDEX BY BINARY_INTEGER;
  last_names_tab   t_names;
BEGIN
  FOR emp_rec IN (SELECT employee_id, last_name
                   FROM employees) LOOP
    last_names_tab(emp_rec.employee_id) := emp_rec.last_name;
  END LOOP;
END;
```

This example populates the INDEX BY table with employees' last names, using employee_id as the primary key.

## Using INDEX BY Table Methods

We can use built-in procedures and functions (called methods) to reference single elements of the table, or to read successive elements. The available methods are:

- EXISTS
- COUNT
- FIRST and LAST
- PRIOR
- NEXT
- DELETE
- TRIM

We use these methods by dot-prefixing the method-name with the table-name.

## Important INDEX BY Table Methods

| | |
|---|---|
| COUNT | returns the number of elements in the table: |

```
v_count := last_names_tab.COUNT;
```

| | |
|---|---|
| EXISTS(N) | returns TRUE if the Nth element exists |

```
IF last_names_tab.EXISTS(27) THEN ...
```

| | |
|---|---|
| FIRST | returns the smallest index number in the table |
| LAST | returns the largest index number in the table |

```
v_lowest_number  := last_names_tab.FIRST;
v_highest_number := last_names_tab.LAST;
```

## Using INDEX BY Table Methods

```
DECLARE
  TYPE t_names IS  TABLE OF VARCHAR2(50)
                   INDEX BY BINARY_INTEGER;
  last_names_tab   t_names;
  v_count          INTEGER;
BEGIN
 -- populate the INDEX BY table with employee data as before
  v_count := last_names_tab.COUNT;                     --1
  FOR i IN last_names_tab.FIRST .. last_names_tab.LAST --2
    LOOP
      IF last_names_tab.EXISTS(i) THEN                 --3
        DBMS_OUTPUT.PUT_LINE(last_names_tab(i));
      END IF;
    END LOOP;
END;
```

## INDEX BY TABLE OF RECORDS

Even though an index by table can only have one data field, that field can a composite data type such as a RECORD.

The record can be %ROWTYPE or a user-defined record.

This example declares an INDEX BY table to store complete employee rows:

```
DECLARE
 TYPE t_emprec IS  TABLE OF employees%ROWTYPE
                  INDEX BY BINARY_INTEGER;
 employees_tab   t_emprec;
```

## Using an INDEX BY Table of Records

```
DECLARE
  TYPE t_emprec IS  TABLE OF employees%ROWTYPE
                    INDEX BY BINARY_INTEGER;
 employees_tab   t_emprec;
BEGIN
  FOR emp_rec IN (SELECT * FROM employees) LOOP
   employees_tab(emp_rec.employee_id) := emp_rec;
  END LOOP;
 FOR i IN employees_tab.FIRST .. employees_tab.LAST
   LOOP
    IF employees_tab.EXISTS(i) THEN
       DBMS_OUTPUT.PUT_LINE(employees_tab(i).first_name);
    END IF;
   END LOOP;
END;
```

## Understanding Dependencies

## Understanding Dependencies

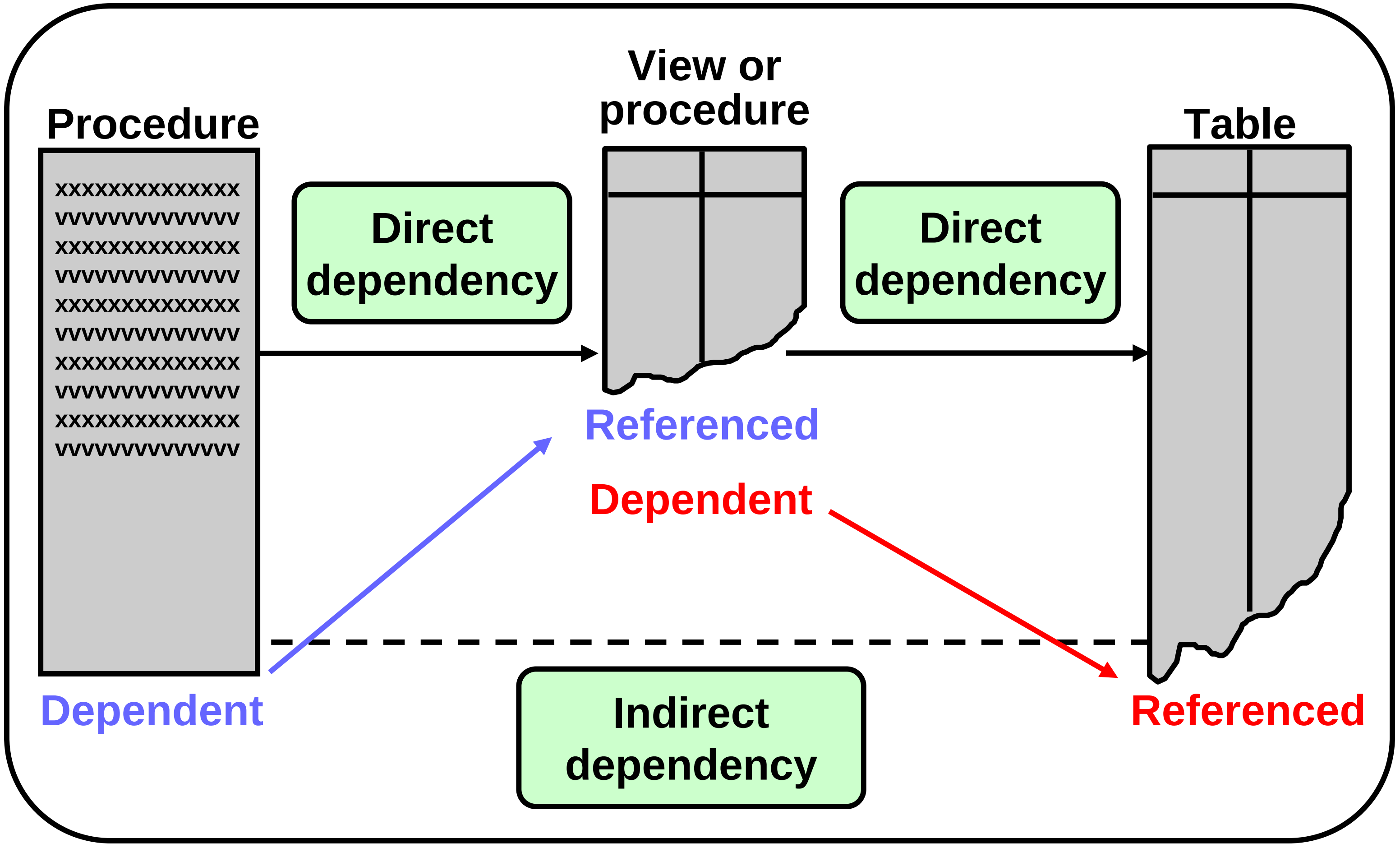
### Dependent and Referenced Objects

- Some objects reference other objects as part of their definitions. For example, a stored procedure could contain a `SELECT` statement that selects columns from a table. For this reason, the stored procedure is called a dependent object, whereas the table is called a referenced object.

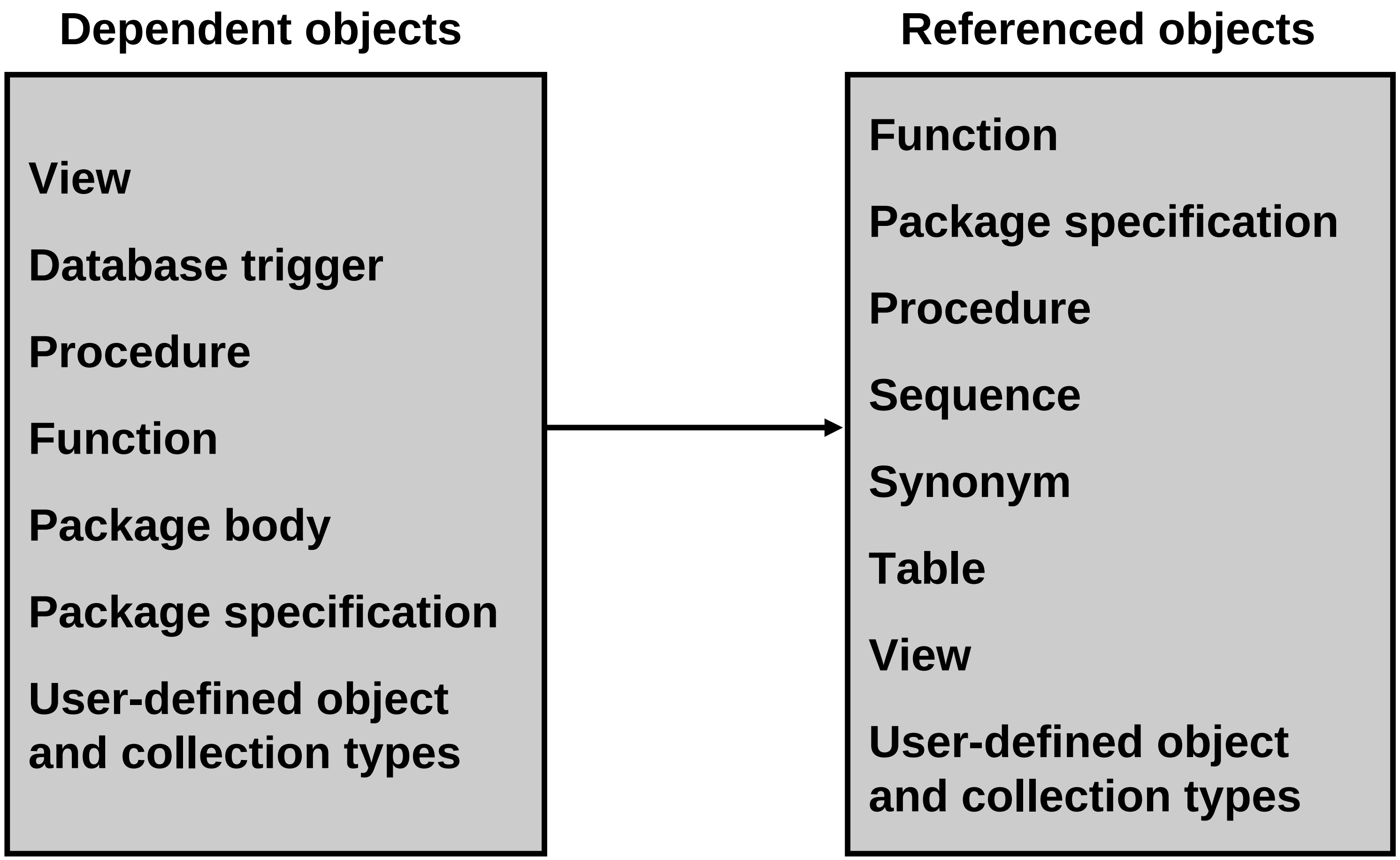
### Dependency Issues

- If you alter the definition of a referenced object, dependent objects may or may not continue to work properly. For example, if the table definition is changed, the procedure may or may not continue to work without error.
- The Oracle server automatically records dependencies among objects. To manage dependencies, all schema objects have a status (valid or invalid) that is recorded in the data dictionary, and you can view the status in the `USER_OBJECTS` data dictionary view.

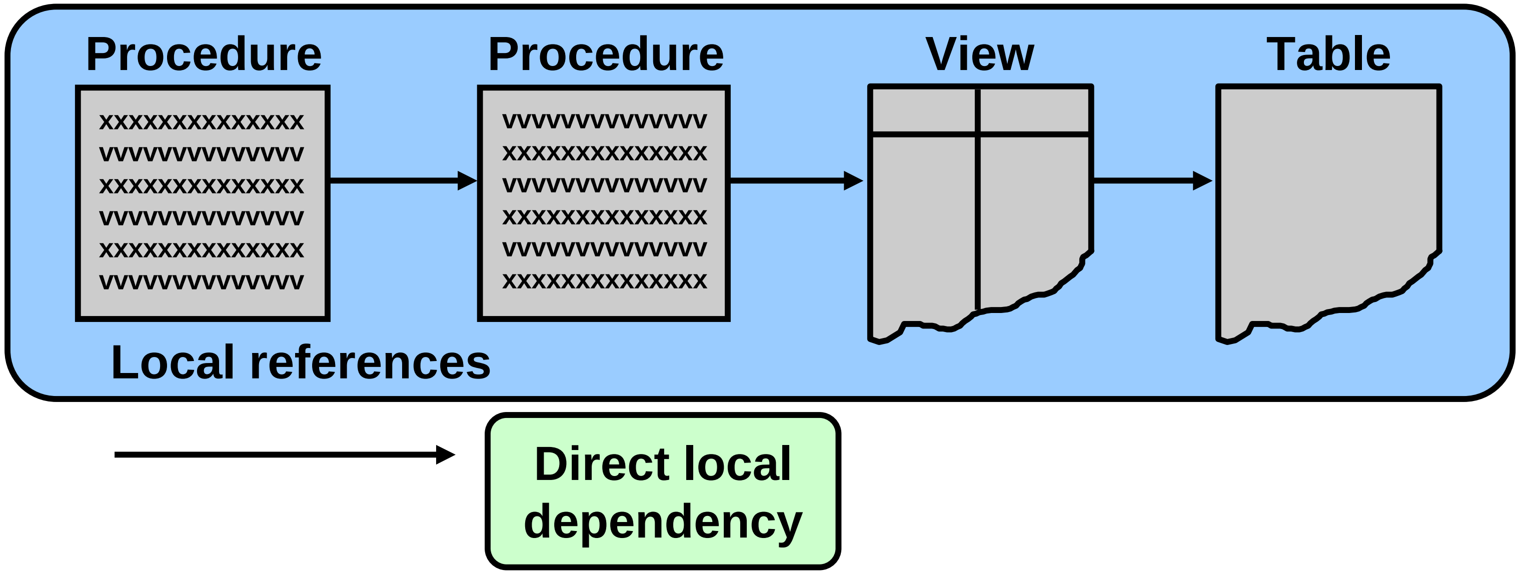
## Dependencies



## Dependencies Summarized

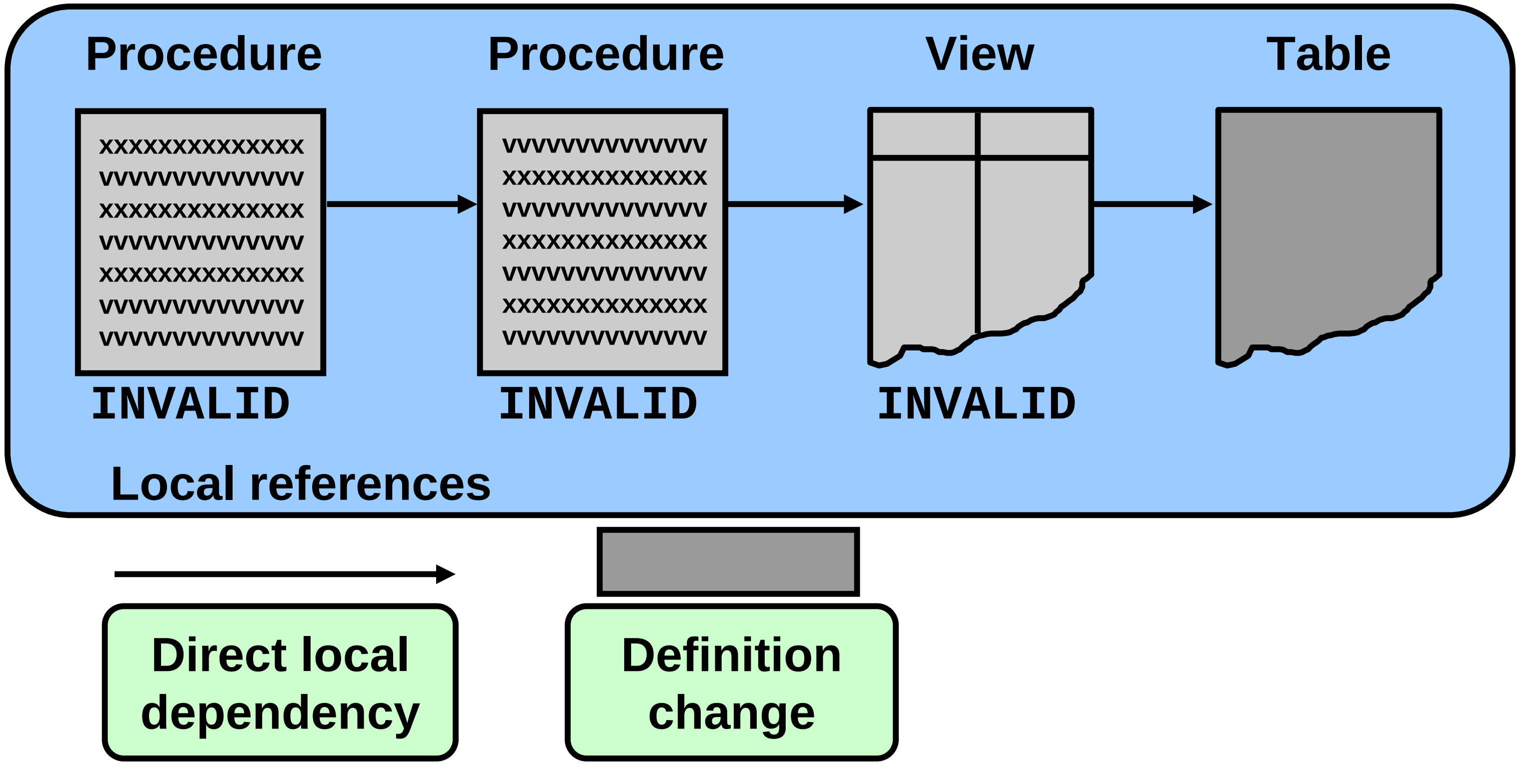


## Local Dependencies



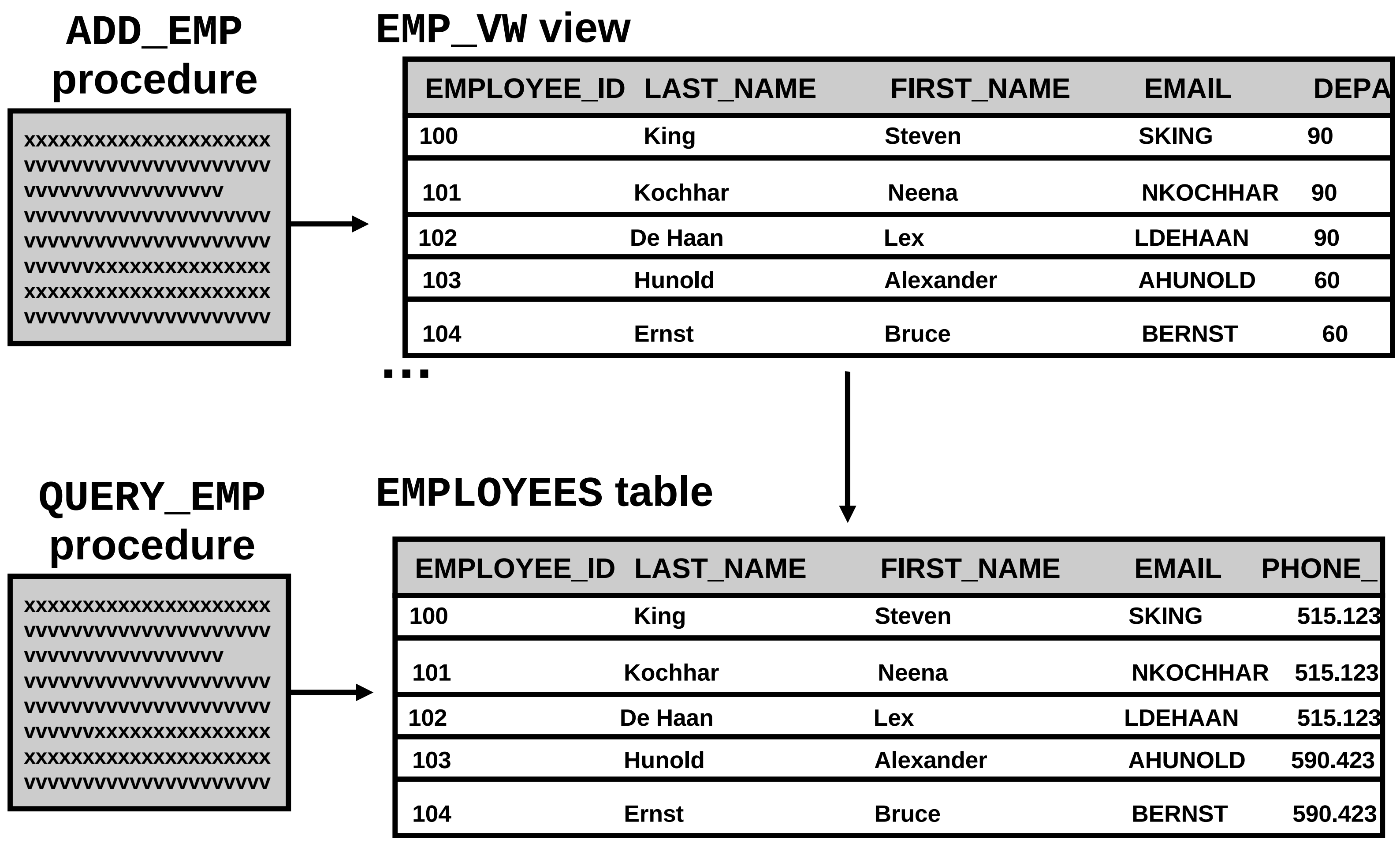
In the case of local dependencies, the objects are on the same node in the same database. The Oracle server automatically manages all local dependencies, using the database's internal "depends-on" table. When a referenced object is modified, the dependent objects are invalidated.

## Local Dependencies continued



The Oracle server implicitly attempts to recompile any `INVALID` object when the object is next called.

## A Scenario of Local Dependencies



| EMPLOYEE_ID | LAST_NAME | FIRST_NAME | EMAIL | DEPA |
|---|---|---|---|---|
| 100 | King | Steven | SKING | 90 |
| 101 | Kochhar | Neena | NKOCHHAR | 90 |
| 102 | De Haan | Lex | LDEHAAN | 90 |
| 103 | Hunold | Alexander | AHUNOLD | 60 |
| 104 | Ernst | Bruce | BERNST | 60 |

| EMPLOYEE_ID | LAST_NAME | FIRST_NAME | EMAIL | PHONE_ |
|---|---|---|---|---|
| 100 | King | Steven | SKING | 515.123 |
| 101 | Kochhar | Neena | NKOCHHAR | 515.123 |
| 102 | De Haan | Lex | LDEHAAN | 515.123 |
| 103 | Hunold | Alexander | AHUNOLD | 590.423 |
| 104 | Ernst | Bruce | BERNST | 590.423 |

## Displaying Direct Dependencies by Using USER_DEPENDENCIES

```
SELECT name, type, referenced_name, referenced_type
FROM   USER_DEPENDENCIES
WHERE  referenced_name IN ('EMPLOYEES','EMP_VW' );
```

| NAME | TYPE | REFERENCED_NAME | REFERENCED_TYPE |
|---|---|---|---|
| ADD_EMP | PROCEDURE | EMP_VW | VIEW |
| EMP_VW | VIEW | EMPLOYEES | TABLE |
| QUERY_EMP | PROCEDURE | EMPLOYEES | TABLE |

Direct Dependencies can also be viewed in Application Express: SQL Workshop -> Object Browser -> choose an object, then click the Dependencies tab.

## Displaying Direct and Indirect Dependencies

Run the script `utldtree.sql` that creates the objects that enable you to display the direct and indirect dependencies.

This script creates four objects:

- A table `deptree_temptab` to hold dependency data
- A procedure `deptree_fill` to populate the table
- Two views `deptree` and `ideptree` to select and format dependency data from the populated table.

For each object whose dependencies you want to see:

1. Execute the `DEPTREE_FILL` procedure.

```
BEGIN  deptree_fill('TABLE','SCOTT','EMPLOYEES');  END;
```

## Displaying Dependencies: the DEPTREE view

2. Display the dependency data using the DEPTREE view

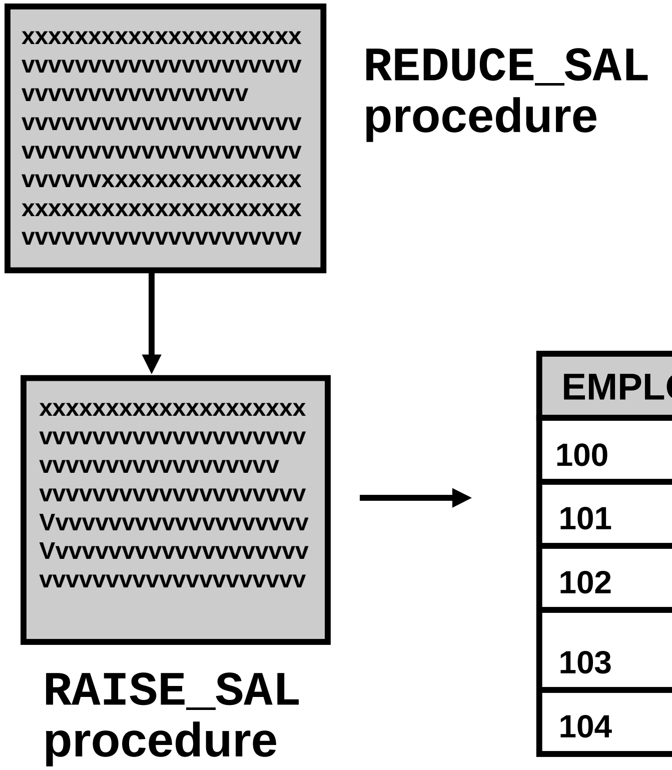
```
SELECT   nested_level, type, name
FROM     deptree
ORDER BY seq#;
```

| NESTED_LEVEL | TYPE | NAME |
|---|---|---|
| 0 | TABLE | EMPLOYEES |
| 1 | VIEW | EMP_VW |
| 2 | PROCEDURE | ADD_EMP |
| 1 | PROCEDURE | QUERY_EMP |

In this example, ADD_EMP is directly dependent on EMP_VW, which in turn is directly dependent on EMPLOYEES (look at the NESTED_LEVEL column).

## Another Scenario of Local Dependencies

If the EMPLOYEES table is altered, both procedures will be marked invalid in the data dictionary. However, if the RAISE_SAL procedure is altered, only the REDUCE_SAL procedure will be marked invalid.



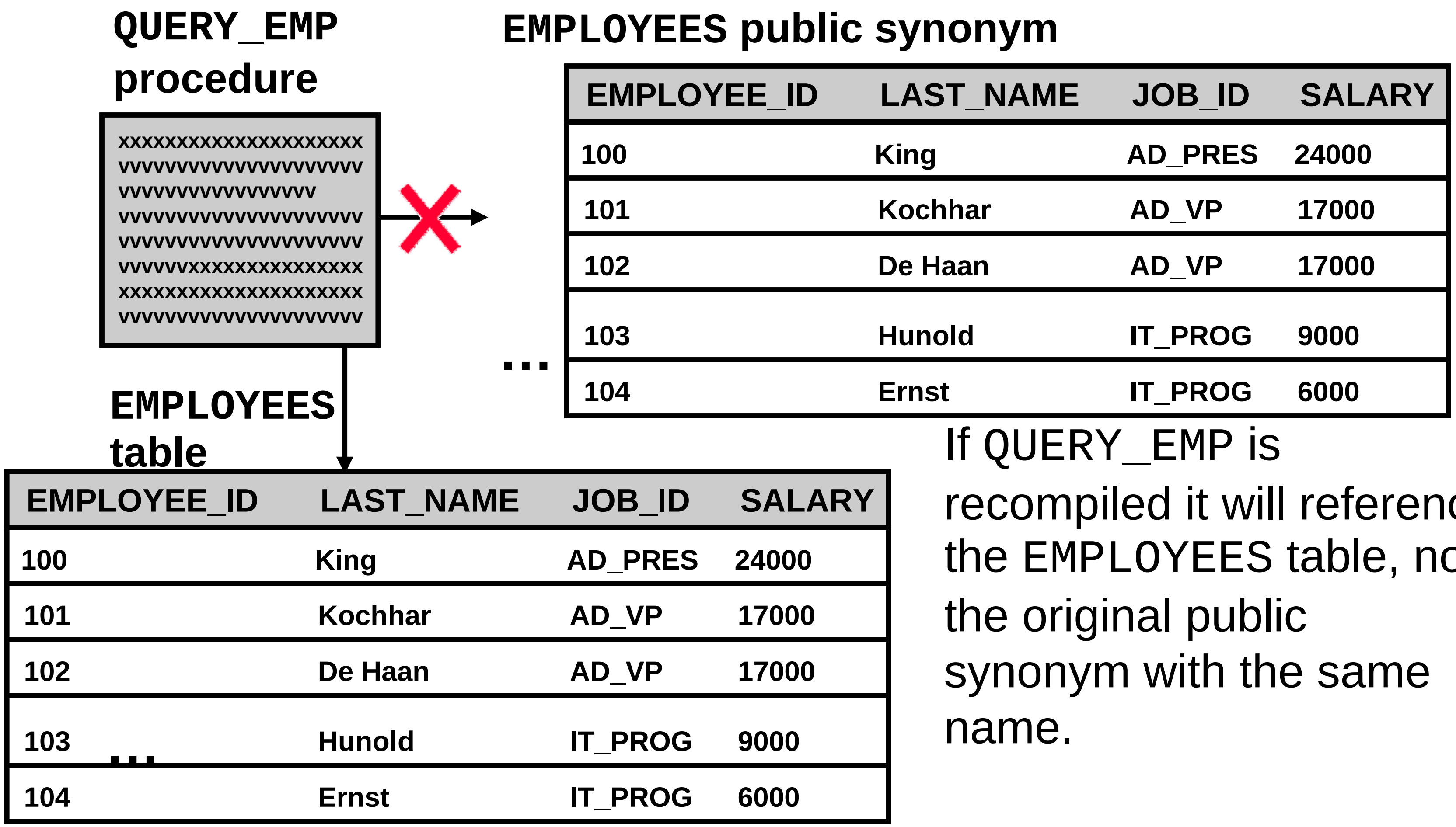
**EMPLOYEES table**

| EMPLOYEE_ID | LAST_NAME | JOB_ID | SALARY |
|---|---|---|---|
| 100 | King | AD_PRES | 24000 |
| 101 | Kochhar | AD_VP | 17000 |
| 102 | De Haan | AD_VP | 17000 |
| 103 | Hunold | IT_PROG | 9000 |
| 104 | Ernst | IT_PROG | 6000 |

ORACLE Academy

## A Third Scenario of Local Naming Dependencies

If dependent objects references a public synonym, and a new object is created in your schema with the same name as the synonym, the dependent objects are invalidated.



**EMPLOYEES public synonym**

| EMPLOYEE_ID | LAST_NAME | JOB_ID | SALARY |
|---|---|---|---|
| 100 | King | AD_PRES | 24000 |
| 101 | Kochhar | AD_VP | 17000 |
| 102 | De Haan | AD_VP | 17000 |
| 103 | Hunold | IT_PROG | 9000 |
| 104 | Ernst | IT_PROG | 6000 |

**EMPLOYEES table**

| EMPLOYEE_ID | LAST_NAME | JOB_ID | SALARY |
|---|---|---|---|
| 100 | King | AD_PRES | 24000 |
| 101 | Kochhar | AD_VP | 17000 |
| 102 | De Haan | AD_VP | 17000 |
| 103 | Hunold | IT_PROG | 9000 |
| 104 | Ernst | IT_PROG | 6000 |

If QUERY_EMP is recompiled it will reference the EMPLOYEES table, not the original public synonym with the same name.

## Recompiling a PL/SQL Program Unit

Recompilation is done automatically, but can be manually done:

- Is handled automatically through implicit run-time recompilation
- Is handled through explicit recompilation with the ALTER statement:

```
ALTER PROCEDURE [SCHEMA.]procedure_name COMPILE;
```

```
ALTER FUNCTION  [SCHEMA.]function_name  COMPILE;
```

```
ALTER PACKAGE [SCHEMA.]package_name
   COMPILE [PACKAGE | SPECIFICATION | BODY];
```

```
ALTER TRIGGER trigger_name COMPILE;
```

ORACLE Academy

## Unsuccessful Recompilation

Automatic or manual recompiling of dependent procedures and functions is unsuccessful when:

- The referenced object is dropped or renamed
- The data type of the referenced column is changed
- The referenced column is dropped
- A referenced view is replaced by a view with different columns
- The parameter list of a referenced procedure is modified

## Successful Recompilation

Automatic or manual recompiling of dependent procedures and functions is successful if:

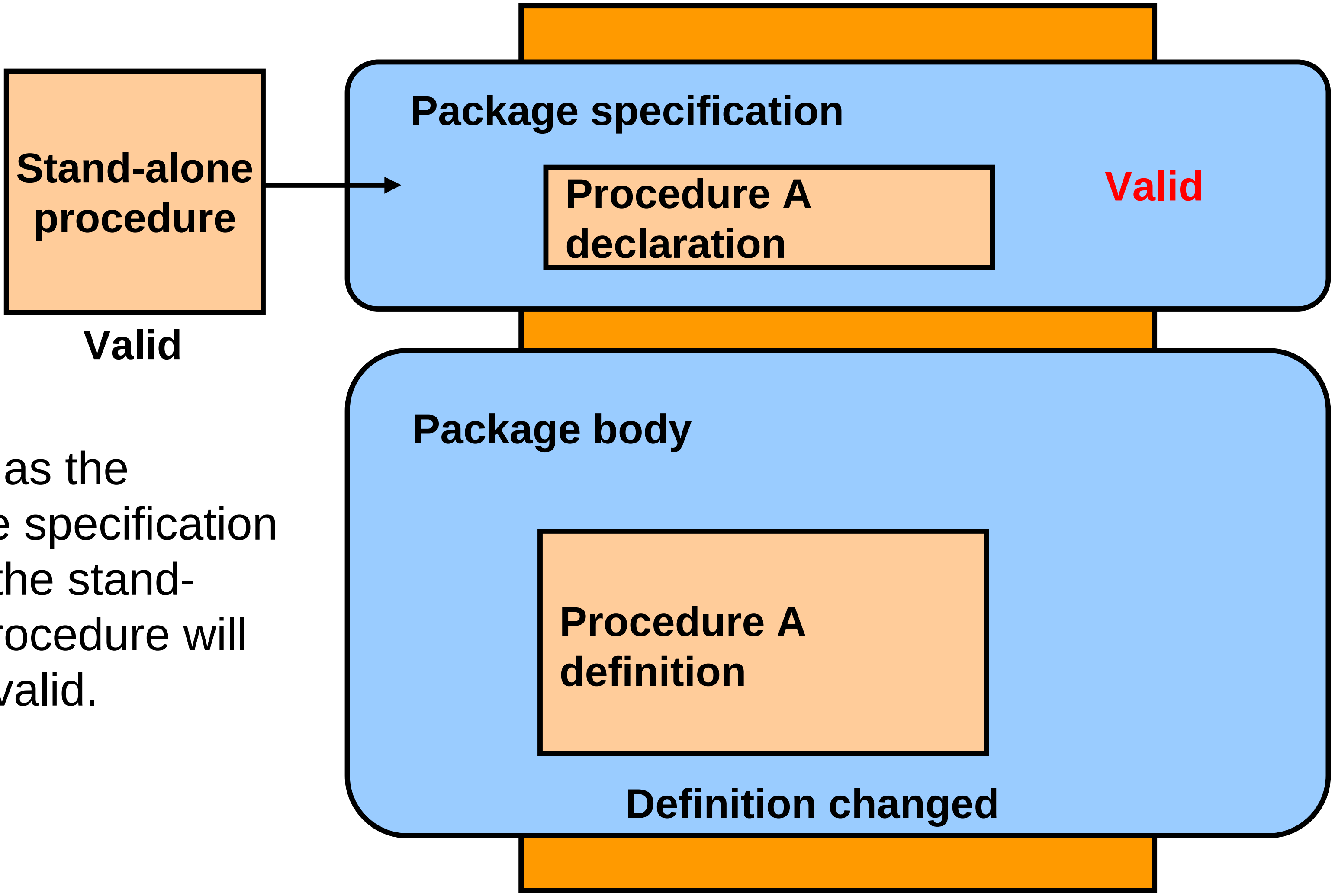
- The referenced table has new columns
- The data type of referenced columns has not changed
- A private table is dropped, but a public table that has the same name and structure exists
- The PL/SQL body of a referenced procedure has been modified and recompiled successfully

## Recompilation of Procedures

Minimize dependency failures by:

- Declaring records with the `%ROWTYPE` attribute
- Declaring variables with the `%TYPE` attribute
- Querying with the `SELECT *` notation
- Including a column list with `INSERT` statements

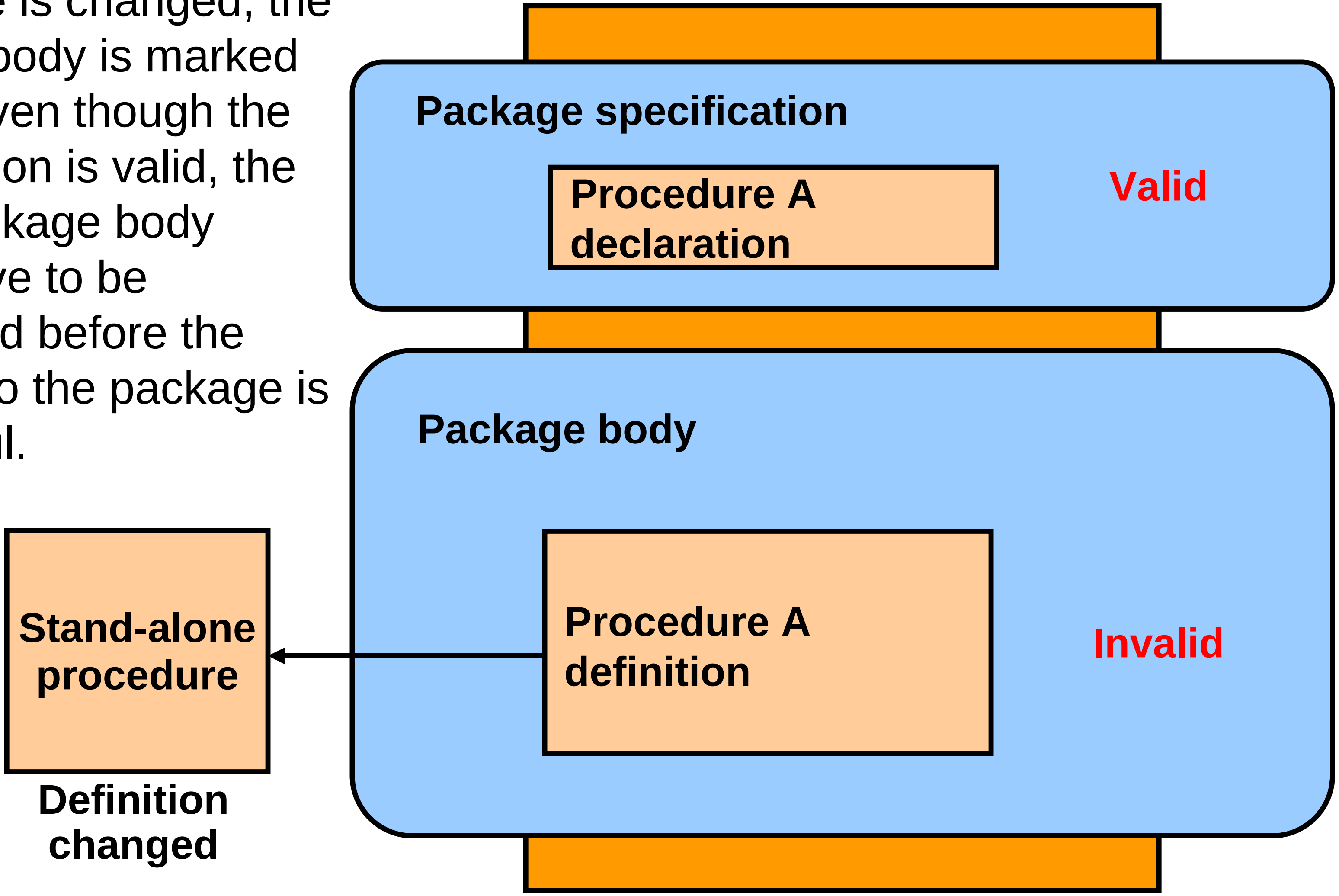
## Packages and Dependencies



As long as the package specification is valid the stand-alone procedure will remain valid.

## Packages and Dependencies

If the stand-alone procedure is changed, the package body is marked invalid. Even though the specification is valid, the entire package body would have to be recompiled before the next call to the package is successful.


Package specification
Procedure A declaration
Valid
Package body
Procedure A definition
Invalid
Stand-alone procedure
Definition changed