3 Working with Collections

Objectives

After completing this lesson, you should be able to do the following:

- **Describe an object type**
- **Create an object type specification**
- **Implement the constructor method on objects**
- **Create collections**
  - **Nested table collections, varray collections**
  - **Associative arrays, string indexed collections**
- **Use collections methods**
- **Manipulate collections**
- **Distinguish between the different types of collections and when to use them**

Objectives

In this lesson, you are introduced to PL/SQL programming using collections, including user-defined object types and constructor methods.

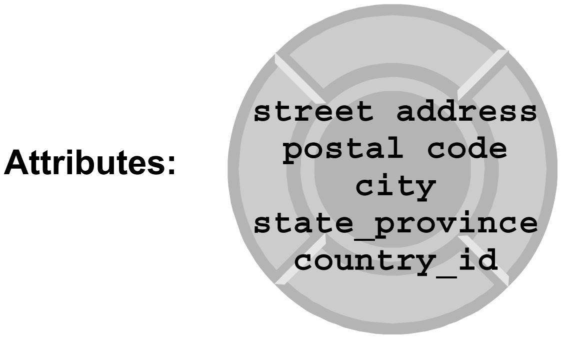
Oracle object types are user-defined data types that make it possible to model complex real-world entities such as customers and purchase orders as unitary entities—objects—in the database.

A collection is an ordered group of elements, all of the same type (for example, phone numbers for each customer). Each element has a unique subscript that determines its position in the collection.

Collections work like the set, queue, stack, and hash table data structures found in most third-generation programming languages. Collections can store instances of an object type and can also be attributes of an object type. Collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms. You can define collection types in a PL/SQL package, then use the same types across many applications.

Understanding the Components of an Object Type

- An object type is a user-defined composite data type.
- An object type encapsulates a data structure.



- Object types can be transient or persistent.

Object Types

Object types are abstractions of the real-world entities used in application programs. They are analogous to Java and C++ classes. You can think of an object type as a template, and an object as a structure that matches the template. Object types can represent many different data structures. Object types are schema objects, subject to the same kinds of administrative control as other schema objects.

An object, such as a car, an order, or a person, has specific attributes and behaviors. You use an object type to maintain this perspective. An object type is a user-defined composite data type that encapsulates a data structure.

- The variables that make up the data structure are called attributes.
- The data structure formed by a set of attributes is public (visible to client programs).

Persistent Versus Transient Objects

For persistent objects, the associated object instances are stored in the database. Persistent object types are defined in the database with the `CREATE` SQL statement. Transient objects are defined programmatically with PL/SQL and differ from persistent objects the way they are declared, initialized, used, and deleted. When the program unit finishes execution, the transient object no longer exists, but the type exists in the database.

Transient objects are defined as an instance of a persistent object type; therefore, transient object attributes cannot be PL/SQL data types.

Creating an Object Type

- **Syntax**

```
CREATE [OR REPLACE] TYPE type_name
  AS OBJECT
    ( attribute1 datatype,
      attribute2 datatype,
      ...
    );
```

- **Example of a persistent object type**

```
CREATE TYPE cust_address_typ
  AS OBJECT
    ( street_address     VARCHAR2(40)
    , postal_code        VARCHAR2(10)
    , city               VARCHAR2(30)
    , state_province     VARCHAR2(10)
    , country_id         CHAR(2)
    );
/
```

street address
postal code
city
state_province
country_id

Creating an Object Type

Creating an object type is similar to creating a package specification. In the object type definition, you list the attributes and data types for the object that you are creating (similar to defining variables in a package specification).

Object Methods

Object types can include procedures and functions to manipulate the object attributes. The procedures and functions that characterize the behavior are called methods. These methods are named when you define the object type. Another component of defining an object is creating the object type specification. It is similar to a package specification. In the object type specification, you define the code for the methods. Methods are covered in detail in the *Oracle Database Application Developer's Guide - Object-Relational Features* manual.

Using an Object Type

You can use an object type as an abstract data type for a column in a table:

```
CREATE TABLE customers
    ( customer_id        NUMBER(6)     ...
    , cust_first_name    VARCHAR2(20)  ...
    , cust_last_name     VARCHAR2(20)  ...
    , cust_address       cust_address_typ
    ...
```

```
DESCRIBE customers

Name                          Null?    Type
----------------------------- -------- ----------
CUSTOMER_ID                   NOT NULL NUMBER(6)
CUST_FIRST_NAME               NOT NULL VARCHAR2(20)
CUST_LAST_NAME                NOT NULL VARCHAR2(20)
CUST_ADDRESS                           CUST_ADDRESS_TYP
...
```

Using an Object Type

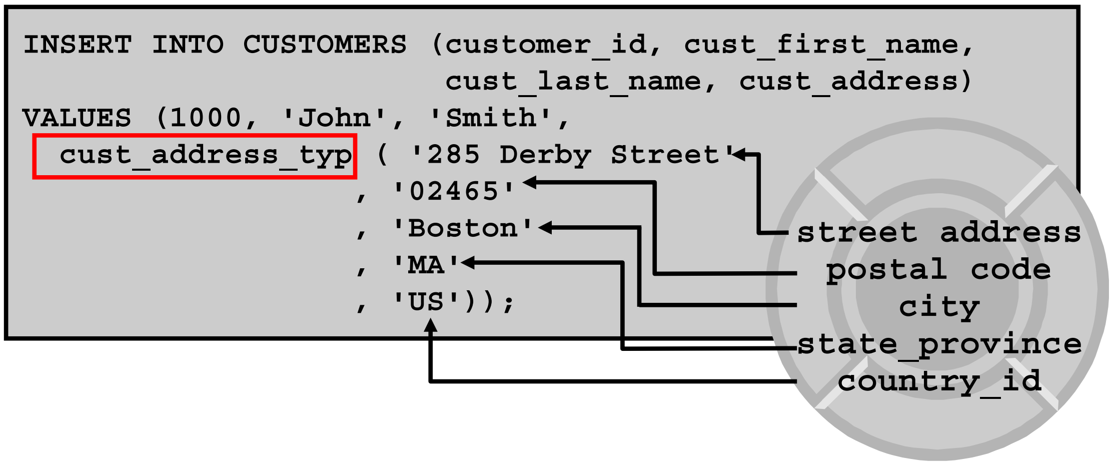
You can use object types as a column's data type. For example, in the CUSTOMERS table, the column CUST_ADDRESS has the data type of CUST_ADDRESS_TYP. This makes the CUST_ADDRESS column a multicelled field where each component is an attribute of the CUST_ADDRESS_TYP.

You can also use object types as abstract data types for variables in a PL/SQL subroutine. In this example, a local variable is defined to hold a value of CUST_ADDRESS_TYP type. This variable is a transient object—it exists for the duration of the execution of the program.

```
DECLARE
  v_address cust_address_typ;
BEGIN
  SELECT cust_address
    INTO v_address
    FROM customers
    WHERE customer_id = 101;
  DBMS_OUTPUT.PUT_LINE (v_address.street_address);
END;
/

514 W Superior St
PL/SQL procedure successfully completed.
```

Using Constructor Methods



```
INSERT INTO CUSTOMERS (customer_id, cust_first_name,
                        cust_last_name, cust_address)
VALUES (1000, 'John', 'Smith',
  cust_address_typ ( '285 Derby Street'
                   , '02465'
                   , 'Boston'
                   , 'MA'
                   , 'US'));
```

Using Constructor Methods

Every object type has a constructor method. A constructor is an implicitly defined function that is used to initialize an object. For arguments, it takes the values of the attributes for an object. PL/SQL never calls a constructor implicitly, so you must call it explicitly. You can make constructor calls wherever function calls are allowed. The constructor method has:

- The same name as the object type
- Formal parameters that exactly match the attributes of the object type (the number, order, and data types are the same)
- A return value of the given object type

In the example shown, the `cust_address_typ` constructor is used to initialize a row in the `cust_address` column of the `CUSTOMERS` table. The `cust_address_typ` constructor matches the `typ_ cust_address` object and has five arguments that match the attributes for the `typ_ cust_address` object.

Retrieving Data from Object Type Columns

```
SELECT customer_id, cust_first_name, cust_last_name, cust_address
FROM   customers
WHERE  customer_id = 1000;

CUSTOMER_ID CUST_FIRST_NAME      CUST_LAST_NAME
----------- -------------------- --------------------
CUST_ADDRESS(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
--------------------------------------------------------------------------
       1000 John                 Smith
CUST_ADDRESS_TYP ('285 Derby Street', '02465', 'Boston', 'MA', 'US')
```

1

```
SELECT c.customer_id, c.cust_address.street_address,
       c.cust_address.city, c.cust_address.state_province,
       c.cust_address.postal_code, c.cust_address.country_id
FROM   customers c
WHERE  customer_id = 1000;

CUSTOMER_ID CUST_ADDRESS.STREET_ADDRESS
----------- ----------------------------------------
CUST_ADDRESS.CITY              CUST_ADDRE CUST_ADDRE CU
------------------------------ ---------- ---------- --
       1000 285 Derby Street
Boston                         MA         02465      US
```

2

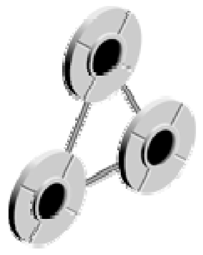
Retrieving Data from Object Type Columns

You retrieve information from object type columns by using a `SELECT` statement. You can view the results as a set of the constructor type (1), or in a flattened form (2).

The flattened form is useful when you access Oracle collection columns from relational tools and APIs, such as ODBC.

Understanding Collections

- A collection is a group of elements, all of the same type.
- Collections work like arrays.
- Collections can store instances of an object type and, conversely, can be attributes of an object type.
- Types of collections in PL/SQL:
  - Nested tables
  - Varrays
  - Associative arrays
  - String indexed collections
  - `INDEX BY pls_integer`



Collections

A collection is a group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection. Collections work like the arrays found in most third-generation programming languages. They can store instances of an object type and, conversely, can be attributes of an object type. Collections can also be passed as parameters. You can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

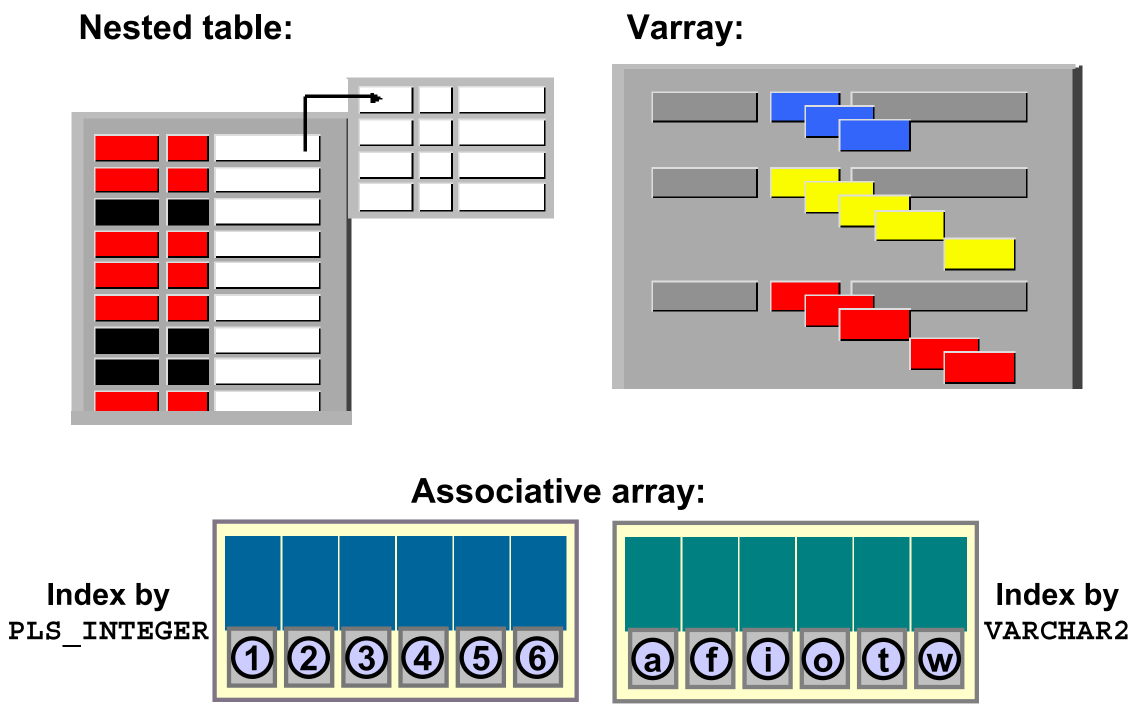
Object types are used not only to create object relational tables, but also to define collections.

You can use any of the three categories of collections:

- Nested tables can have any number of elements.
- A varray is an ordered collection of elements.
- Associative arrays (known as “index-by tables” in previous Oracle releases) are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be an integer or a string.

Note: Associative arrays indexed by integer are covered in the prerequisite courses: *Oracle Database 10g: Program with PL/SQL* and *Oracle Database 10g: Develop PL/SQL Program Units* and will not be emphasized in this course.

Describing the Collection Types



Collections (continued)

PL/SQL offers three collection types:

Nested Tables

A nested table holds a set of values. In other words, it is a table within a table. Nested tables are unbounded, meaning the size of the table can increase dynamically. Nested tables are available in both PL/SQL as well as the database. Within PL/SQL, nested tables are like one-dimensional arrays whose size can increase dynamically. Within the database, nested tables are column types that hold sets of values. The Oracle database stores the rows of a nested table in no particular order. When you retrieve a nested table from the database into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. This gives you array-like access to individual rows. Nested tables are initially dense but they can become sparse through deletions and therefore have nonconsecutive subscripts.

Varrays

Variable-size arrays, or varrays, are also collections of homogeneous elements that hold a fixed number of elements (although you can change the number of elements at run time). They use sequential numbers as subscripts. You can define equivalent SQL types, allowing varrays to be stored in database tables. They can be stored and retrieved through SQL, but with less flexibility than nested tables. You can reference the individual elements for array operations, or manipulate the collection as a whole.

Collections (continued)

Varrays (continued)

Varrays are always bounded and never sparse. You can specify the maximum size of the varray in its type definition. Its index has a fixed lower bound of 1 and an extensible upper bound. A varray can contain a varying number of elements, from zero (when empty) to the maximum specified in its type definition.

To reference an element, you can use the standard subscripting syntax.

Associative Arrays

Associative arrays are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be either integer (`PLS_INTEGER`) or character (`VARCHAR2`) based.

When you assign a value using a key for the first time, it adds that key to the associative array. Subsequent assignments using the same key update the same entry. It is important to choose a key that is unique. For example, key values may come from the primary key of a database table, from a numeric hash function, or from concatenating strings to form a unique string value.

Because associative arrays are intended for storing temporary data rather than storing persistent data, you cannot use them with SQL statements such as `INSERT` and `SELECT INTO`. You can make them persistent for the life of a database session by declaring the type in a package and assigning the values in a package body.

Choosing a PL/SQL Collection Type

If you already have code or business logic that uses some other language, you can usually translate that language's array and set types directly to PL/SQL collection types.

- Arrays in other languages become varrays in PL/SQL.
- Sets and bags in other languages become nested tables in PL/SQL.
- Hash tables and other kinds of unordered lookup tables in other languages become associative arrays in PL/SQL.

If you are writing original code or designing the business logic from the start, consider the strengths of each collection type and decide which is appropriate.

Listing Characteristics for Collections

	PL/SQL Nested Tables	DB Nested Tables	PL/SQL Varrays	DB Varrays	PL/SQL Associative Arrays
Maximum size	No	No	Yes	Yes	Dynamic
Sparsity	Can be	No	Dense	Dense	Yes
Storage	N/A	Stored out of line	N/A	Stored in-line (if < 4,000 bytes)	N/A
Ordering	Does not retain ordering and subscripts	Does not retain ordering and subscripts	Retains ordering and subscripts	Retains ordering and subscripts	Retains ordering and subscripts

Choosing Between Nested Tables and Associative Arrays

- Use associative arrays when:
  - You need to collect information of unknown volume
  - You need flexible subscripts (negative, non-sequential, or string based)
  - You need to pass the collection to and from the database server (use associative arrays with the bulk constructs)
- Use nested tables when:
  - You need persistence
  - You need to pass the collection as a parameter

Choosing Between Nested Tables and Varrays

- Use varrays when:
  - The number of elements is known in advance
  - The elements are usually all accessed in sequence
- Use nested tables when:
  - The index values are not consecutive
  - There is no predefined upper bound for index values
  - You need to delete or update some elements, but not all the elements at once
  - You would usually create a separate lookup table, with multiple entries for each row of the main table, and access it through join queries

Using Collections Effectively

- Varrays involve fewer disk accesses and are more efficient.
- Use nested tables for storing large amounts of data.
- Use varrays to preserve the order of elements in the collection column.
- If you do not have a requirement to delete elements in the middle of a collection, favor varrays.
- Varrays do not allow piecewise updates.

Guidelines for Using Collections Effectively

- Because varray data is stored in-line (in the same tablespace), retrieving and storing varrays involves fewer disk accesses. Varrays are thus more efficient than nested tables.
- To store large amounts of persistent data in a column collection, use nested tables. This way the Oracle server can use a separate table to hold the collection data which can grow over time. For example, when a collection for a particular row could contain 1 to 1,000,000 elements, a nested table is simpler to use.
- If your data set is not very large and it is important to preserve the order of elements in a collection column, use varrays. For example, if you know that in each row the collection will not contain more than ten elements, you can use a varray with a limit of ten.
- If you do not want to deal with deletions in the middle of the data set, use varrays.
- If you expect to retrieve the entire collection simultaneously, use varrays.
- Varrays do not allow piecewise updates.

Note: If your application requires negative subscripts, you can use only associative arrays.

Creating Collection Types

Nested table in the database:

```
CREATE [OR REPLACE] TYPE type_name AS TABLE OF
Element_datatype [NOT NULL];
```

Nested table in PL/SQL:

```
TYPE type_name IS TABLE OF element_datatype
[NOT NULL];
```

Varray in the database:

```
CREATE [OR REPLACE] TYPE type_name AS VARRAY
(max_elements) OF element_datatype [NOT NULL];
```

Varray in PL/SQL:

```
TYPE type_name IS VARRAY  (max_elements) OF
element_datatype [NOT NULL];
```

Associative array in PL/SQL (string indexed):

```
TYPE type_name IS TABLE OF  (element_type)
INDEX BY VARCHAR2(size)
```

Creating Collection Types

To create a collection, you first define a collection type, and then declare collections of that type. The slide above shows the syntax for defining nested table and varray collection types in both the database (persistent) and in PL/SQL (transient), and defining a string indexed collection in PL/SQL.

Creating Collections in the Database

You can create a nested table or a varray data type in the database, which makes the data type available to use in places such as columns in database tables, variables in PL/SQL programs, and attributes of object types.

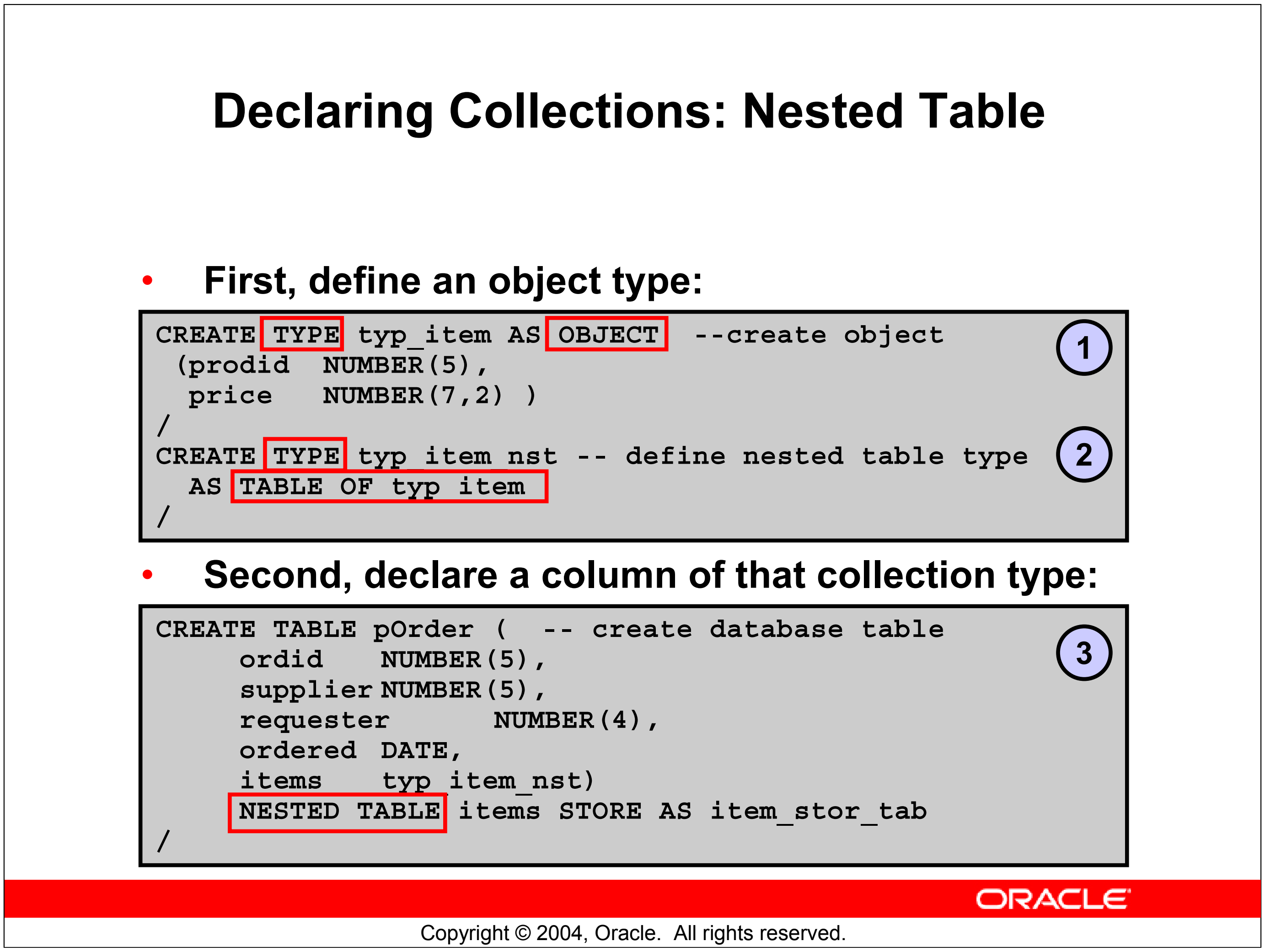
Before you can define a database table containing a nested table or varray, you must first create the data type for the collection in the database.

Use the syntax shown in the slide to create collection types in the database.

Creating Collections in PL/SQL

You can also create a nested table or a varray in PL/SQL. Use the syntax shown in the slide to create collection types in PL/SQL. You can create associative arrays in PL/SQL only.

Note: Collections can be nested. In Oracle9*i* and later, collections of collections are possible.



Declaring Collections: Nested Table

- First, define an object type:

```
CREATE TYPE typ_item AS OBJECT  --create object          1
 (prodid  NUMBER(5),
  price   NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type     2
  AS TABLE OF typ_item
/
```

- Second, declare a column of that collection type:

```
CREATE TABLE pOrder (  -- create database table          3
     ordid    NUMBER(5),
     supplier NUMBER(5),
     requester      NUMBER(4),
     ordered  DATE,
     items    typ_item_nst)
     NESTED TABLE items STORE AS item_stor_tab
/
```

ORACLE

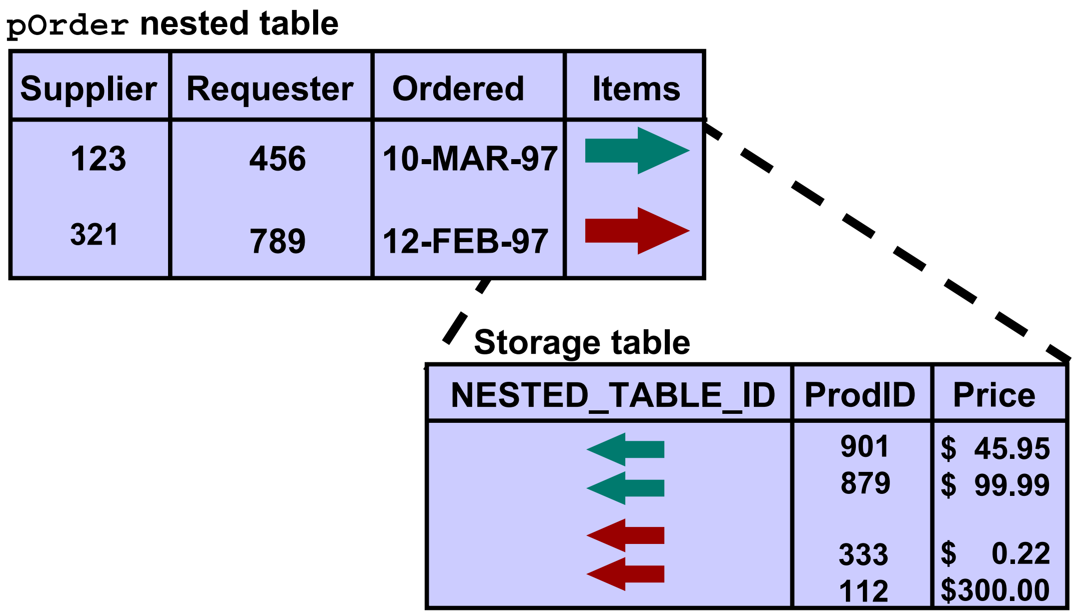
Declaring Collections: Nested Table

To create a table based on a nested table, perform the following steps:

1. Create the `typ_item` type, which holds the information for a single line item.
2. Create the `typ_item_nst` type, which is created as a table of the `typ_item` type.
   Note: You must create the `typ_item_nst` nested table type based on the previously declared type because it is illegal to declare multiple data types in this nested table declaration.
3. Create the `pOrder` table and use the nested table type in a column declaration, which will include an arbitrary number of items based on the `typ_item_nst` type. Thus, each row of `pOrder` may contain a table of items.
   The `NESTED TABLE STORE AS` clause is required to indicate the name of the storage table in which the rows of all the values of the nested table reside. The storage table is created in the same schema and the same tablespace as the parent table.
   Note: The dictionary view `USER_COLL_TYPES` holds information about collections.

Understanding Nested Table Storage

Nested tables are stored out-of-line in storage tables.



`pOrder` **nested table**

Supplier	Requester	Ordered	Items
123	456	10-MAR-97	→
321	789	12-FEB-97	→

Storage table

NESTED_TABLE_ID	ProdID	Price
←	901	$ 45.95
←	879	$ 99.99
←	333	$ 0.22
←	112	$300.00

Nested Table Storage

The rows for all nested tables of a particular column are stored within the same segment. This segment is called the *storage table*.

A storage table is a system-generated segment in the database that holds instances of nested tables within a column. You specify the name for the storage table by using the `NESTED TABLE STORE AS` clause in the `CREATE TABLE` statement. The storage table inherits storage options from the outermost table.

To distinguish between nested table rows belonging to different parent table rows, a system-generated nested table identifier that is unique for each outer row enclosing a nested table is created.

Operations on storage tables are performed implicitly by the system. You should not access or manipulate the storage table, except implicitly through its containing objects.

Privileges of the column of the parent table are transferred to the nested table.

Declaring Collections: Varray

- **First, define a collection type:**

```
CREATE TYPE typ_Project AS OBJECT(  --create object        1
   project_no NUMBER(2),
   title      VARCHAR2(35),
   cost       NUMBER(7,2))
/
CREATE TYPE typ_ProjectList AS VARRAY (50) OF typ_Project
       -- define VARRAY type                               2
/
```

- **Second, declare a collection of that type:**

```
CREATE TABLE department (  -- create database table       3
   dept_id  NUMBER(2),
   name     VARCHAR2(15),
   budget   NUMBER(11,2),
   projects typ_ProjectList)  -- declare varray as column
/
```

ORACLE

Example

The example above shows how to create a table based on a varray.

1. Create the `typ_project` type, which holds information for a project.
2. Create the `typ_ projectlist` type, which is created as a varray of the project type. The varray contains a maximum of 50 elements.
3. Create the `department` table and use the varray type in a column declaration. Each element of the varray will store a project object.

This example demonstrates how to create a varray of phone numbers, then use it in a `CUSTOMERS` table (The `OE` sample schema uses this definition.):

```
CREATE TYPE phone_list_typ
AS VARRAY(5) OF VARCHAR2(25);
/
CREATE TABLE customers
(customer_id NUMBER(6)
,cust_first_name VARCHAR2(50)
,cust_last_name VARCHAR2(50)
,cust_address cust_address_typ(100)
,phone_numbers phone_list_typ
...
);
```

Working with Collections in PL/SQL

- You can declare collections as formal parameters of procedures and functions.
- You can specify a collection type in the `RETURN` clause of a function specification.
- Collections follow the usual scoping and instantiation rules.

```
CREATE OR REPLACE PACKAGE manage_dept_proj AS
  TYPE typ_proj_details IS TABLE OF typ_Project;
  ...
  PROCEDURE allocate_proj
    (propose_proj IN typ_proj_details);
  FUNCTION top_project (n NUMBER)
     RETURN typ_proj_details;
  ...
```

Working with Collections

There are several points about collections that you must know when working with them:

- You can declare collections as the formal parameters of functions and procedures. That way, you can pass collections to stored subprograms and from one subprogram to another.
- A function's `RETURN` clause can be a collection type.
- Collections follow the usual scoping and instantiation rules. In a block or subprogram, collections are instantiated when you enter the block or subprogram and cease to exist when you exit. In a package, collections are instantiated when you first reference the package and cease to exist when you end the database session.

In the example in the slide, a nested table is used as the formal parameter of a packaged procedure, the data type of an `IN` parameter for the procedure `ALLOCATE_PROJ`, and the return data type of the `TOP_PROJECT` function.

Initializing Collections

Three ways to initialize:

- Use a constructor.
- Fetch from the database.
- Assign another collection variable directly.

```
DECLARE           --this example uses a constructor
  v_accounting_project typ_ProjectList;
BEGIN
  v_accounting_project :=
    typ_ProjectList
      (typ_Project (1, 'Dsgn New Expense Rpt', 3250),
       typ_Project (2, 'Outsource Payroll', 12350),
       typ_Project (3, 'Audit Accounts Payable',1425));
   INSERT INTO department
     VALUES(10, 'Accounting', 123, v_accounting_project);
...
END;
/
```

Initializing Collections

Until you initialize it, a collection is atomically null (that is, the collection itself is null, not its elements). To initialize a collection, you can use one of the following means:

- Use a constructor, which is a system-defined function with the same name as the collection type. A constructor allows the creation of an object from an object type. Invoking a constructor is a way to instantiate (create) an object. This function "constructs" collections from the elements passed to it. In the example shown above, you pass three elements to the `typ_ProjectList()` constructor, which returns a varray containing those elements.
- Read an entire collection from the database using a fetch.
- Assign another collection variable directly. You can copy the entire contents of one collection to another as long as both are built from the same data type.

Initializing Collections

1

```
DECLARE      -- this example uses a fetch from the database
  v_accounting_project typ_ProjectList;
BEGIN
  SELECT  projects
    INTO  v_accounting_project
    FROM  department
    WHERE dept_id = 10;
...
END;
/
```

2

```
DECLARE -- this example assigns another collection
        -- variable directly
  v_accounting_project typ_ProjectList;
  v_backup_project     typ_ProjectList;
BEGIN
  SELECT  projects
    INTO  v_accounting_project
    FROM  department
    WHERE dept_id = 10;
  v_backup_project := v_accounting_project;
END;
/
```

Initializing Collections (continued)

In the first example shown above, an entire collection from the database is fetched into the local PL/SQL collection variable.

In the second example shown above, the entire contents of one collection variable are assigned to another collection variable.

Referencing Collection Elements

Use the collection name and a subscript to reference a collection element:

- **Syntax:**

```
collection_name(subscript)
```

- **Example:**

```
v_accounting_project(1)
```

- **To reference a field in a collection:**

```
v_accounting_project(1).cost
```

Referencing Collection Elements

Every element reference includes a collection name and a subscript enclosed in parentheses. The subscript determines which element is processed. To reference an element, you can specify its subscript by using the following syntax:

```
        collection_name(subscript)
```

In the preceding syntax, *subscript* is an expression that yields a positive integer. For nested tables, the integer must lie in the range 1 to 2147483647. For varrays, the integer must lie in the range 1 to maximum_size.

Using Collection Methods

- `EXISTS`
- `COUNT`
- `LIMIT`
- `FIRST and LAST`
- `PRIOR and NEXT`
- `EXTEND`
- `TRIM`
- `DELETE`

```
collection_name.method_name [(parameters)]
```

Using Collection Methods

You can use collection methods from procedural statements but not from SQL statements.

Function or Procedure	Description
`EXISTS`	Returns `TRUE` if the nth element in a collection exists, otherwise, `EXISTS(N)` returns `FALSE`
`COUNT`	Returns the number of elements that a collection contains
`LIMIT`	For nested tables that have no maximum size, `LIMIT` returns `NULL`; for varrays, `LIMIT` returns the maximum number of elements that a varray can contain
`FIRST` and `LAST`	Returns the first and last (smallest and largest) index numbers in a collection, respectively
`PRIOR` and `NEXT`	`PRIOR(n)` returns the index number that precedes index `n` in a collection; `NEXT(n)` returns the index number that follows index `n`.
`EXTEND`	Appends one null element. `EXTEND(n)` appends `n` elements; `EXTEND(n, i)` appends `n` copies of the `i`th element
`TRIM`	Removes one element from the end; `TRIM(n)` removes n elements from the end of a collection
`DELETE`	Removes all elements from a nested or associative array table. `DELETE(n)` removes the `n`th element ; `DELETE(m, n)` removes a range. **Note:** Does not work on varrays.

Using Collection Methods

Traverse collections with methods:

```
DECLARE
  i INTEGER;
  v_accounting_project typ_ProjectList;
BEGIN
  v_accounting_project := typ_ProjectList(
    typ_Project (1,'Dsgn New Expense Rpt', 3250),
    typ_Project (2, 'Outsource Payroll', 12350),
    typ_Project (3, 'Audit Accounts Payable',1425));
  i := v_accounting_project.FIRST ;
  WHILE i IS NOT NULL LOOP
    IF v_accounting_project(i).cost > 10000 then
      DBMS_OUTPUT.PUT_LINE('Project too expensive: '
                           || v_accounting_project(i).title);
    END IF;
    i := v_accounting_project.NEXT (i);
  END LOOP;
END;
/
```

Traversing Collections

In the example shown, the `FIRST` method finds the smallest index number, the `NEXT` method traverses the collection starting at the first index. The output from this block of code shown above is:

```
Project too expensive: Outsource Payroll
```

You can use the `PRIOR` and `NEXT` methods to traverse collections indexed by any series of subscripts. In the example shown, the `NEXT` method is used to traverse a varray.

`PRIOR(n)` returns the index number that precedes index `n` in a collection. `NEXT(n)` returns the index number that succeeds index `n`. If `n` has no predecessor, `PRIOR(n)` returns `NULL`. Likewise, if `n` has no successor, `NEXT(n)` returns `NULL`. `PRIOR` is the inverse of `NEXT`.

`PRIOR` and `NEXT` do not wrap from one end of a collection to the other.

When traversing elements, `PRIOR` and `NEXT` ignore deleted elements.

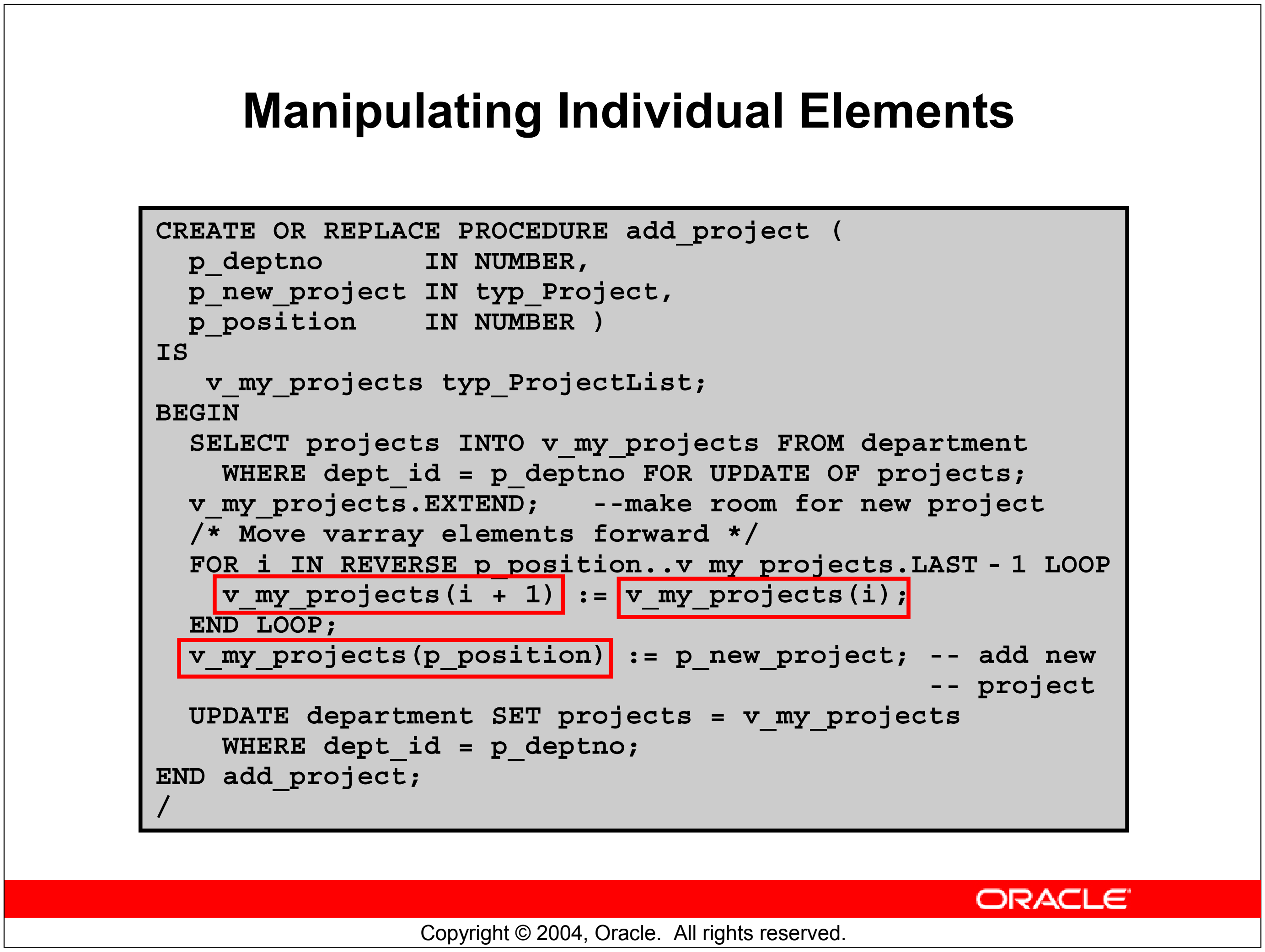
Using Collection Methods

```
DECLARE
  v_my_projects   typ_ProjectList;
  v_array_count   INTEGER;
  v_last_element  INTEGER;
BEGIN
  SELECT projects INTO v_my_projects FROM department
    WHERE dept_id = 10;
  v_array_count := v_my_projects.COUNT ;
  dbms_output.put_line('The # of elements is: ' ||
                       v_array_count);
  v_my_projects.EXTEND ;    --make room for new project
  v_last_element := v_my_projects.LAST ;
  dbms_output.put_line('The last element is: ' ||
                       v_last_element);
  IF v_my_projects.EXISTS(5) THEN
    dbms_output.put_line('Element 5 exists!');
  ELSE
    dbms_output.put_line('Element 5 does not exist.');
  END IF;
END;
/
```

Example

The block of code shown uses the `COUNT`, `EXTEND`, `LAST`, and `EXISTS` methods on the `my_projects` varray. The `COUNT` method reports that the `projects` collection holds three projects for department 10. The `EXTEND` method creates a fourth empty project. Using the `LAST` method reports that four projects exist. When testing for the existence of a fifth project, the program reports that it does not exist. The output from this block of code is as follows:

```
The # of elements is: 3
The last element is: 4
Element 5 does not exist.
PL/SQL procedure successfully completed.
```



Manipulating Individual Elements

```
CREATE OR REPLACE PROCEDURE add_project (
  p_deptno       IN NUMBER,
  p_new_project IN typ_Project,
  p_position     IN NUMBER )
IS
   v_my_projects typ_ProjectList;
BEGIN
  SELECT projects INTO v_my_projects FROM department
    WHERE dept_id = p_deptno FOR UPDATE OF projects;
  v_my_projects.EXTEND;    --make room for new project
  /* Move varray elements forward */
  FOR i IN REVERSE p_position..v_my_projects.LAST - 1 LOOP
    v_my_projects(i + 1) := v_my_projects(i);
  END LOOP;
  v_my_projects(p_position) := p_new_project; -- add new
                                                -- project
  UPDATE department SET projects = v_my_projects
    WHERE dept_id = p_deptno;
END add_project;
/
```

Manipulating Individual Elements

You must use PL/SQL procedural statements to reference the individual elements of a varray in an `INSERT`, `UPDATE`, or `DELETE` statement. In the example shown in the slide, the stored procedure inserts a new project into a department's project at a given position.

To execute the procedure, pass the department number to which you want to add a project, the project information, and the position where the project information is to be inserted.

```
EXECUTE add_project(10, -
            typ_Project(4, 'Information Technology', 789), 4)
SELECT * FROM department;
   DEPT_ID NAME                BUDGET
---------- --------------- ----------
PROJECTS(PROJECT_NO, TITLE, COST)
-----------------------------------------------------------
        10 Accounting             123
PROJECTLIST(PROJECT(1, 'Dsgn New Expense Rpt', 3250),
PROJECT(2, 'Outsource Payroll', 12350),
PROJECT(3, 'Audit Accounts Payable', 1425),
PROJECT(4, 'Information Technology', 789))
```

Avoiding Collection Exceptions

Common exceptions with collections:

- `COLLECTION_IS_NULL`
- `NO_DATA_FOUND`
- `SUBSCRIPT_BEYOND_COUNT`
- `SUBSCRIPT_OUTSIDE_LIMIT`
- `VALUE_ERROR`

Avoiding Collection Exceptions

In most cases, if you reference a nonexistent collection element, PL/SQL raises a predefined exception.

Exception	**Raised when:**
`COLLECTION_IS_NULL`	You try to operate on an atomically null collection
`NO_DATA_FOUND`	A subscript designates an element that was deleted
`SUBSCRIPT_BEYOND_COUNT`	A subscript exceeds the number of elements in a collection
`SUBSCRIPT_OUTSIDE_LIMIT`	A subscript is outside the legal range
`VALUE_ERROR`	A subscript is null or not convertible to an integer

Avoiding Collection Exceptions

Common exceptions with collections:

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  nums NumList;          -- atomically null
BEGIN
  /* Assume execution continues despite the raised
exceptions. */
  nums(1) := 1;          -- raises COLLECTION_IS_NULL
  nums := NumList(1,2);  -- initialize table
  nums(NULL) := 3        -- raises VALUE_ERROR
  nums(0) := 3;          -- raises SUBSCRIPT_OUTSIDE_LIMIT
  nums(3) := 3;          -- raises SUBSCRIPT_BEYOND_COUNT
  nums.DELETE(1);        -- delete element 1
  IF nums(1) = 1 THEN    -- raises NO_DATA_FOUND
...
```

Example

In the first case, the nested table is atomically null. In the second case, the subscript is null. In the third case, the subscript is outside the legal range. In the fourth case, the subscript exceeds the number of elements in the table. In the fifth case, the subscript designates a deleted element.

Working with Collections in SQL

Querying collections:

```
SELECT * FROM department;

   DEPT_ID NAME                 BUDGET
---------- -------------- ----------
PROJECTS(PROJECT_NO, TITLE, COST)
---------------------------------------------------------
        10 Accounting              123
PROJECTLIST(PROJECT(1, 'Dsgn New Expense Rpt', 3250), ...
oll', 12350), PROJECT(3, 'Audit Accounts Payable', ...
```

- **Querying a collection column in the `SELECT` list nests the elements of the collection in the result row with which the collection is associated.**
- **To unnest results from collection queries, use the `TABLE` expression in the `FROM` clause.**

Querying Collections

You can use two general ways to query a table that contains a column or attribute of a collection type. One way returns the collections nested in the result rows that contain them. By including the collection column in the `SELECT` list, the output shows as a row associated with the other row output in the `SELECT` list.

Another method to display the output is to unnest the collection such that each collection element appears on a row by itself. You can use the `TABLE` expression in the `FROM` clause to unnest a collection.

Working with Collections in SQL

TABLE expression:

```
SELECT d1.dept_id, d1.budget, d2.*
FROM   department d1, TABLE(d1.projects) d2;

  DEPT_ID  BUDGET PROJECT_NO TITLE                           COST
------- ------- ---------- ------------------------- ----------
     10     123          1 Dsgn New Expense Rpt            3250
     10     123          2 Outsource Payroll              12350
     10     123          3 Audit Accounts Payable          1425
     10     123          4 Information Technology           789
```

- **Enables you to query a collection in the FROM clause like a table**
- **Can be used to query any collection value expression, including transient values such as variables and parameters**

Querying Collections with the TABLE Expression

To view collections in a conventional format, you must unnest, or flatten, the collection attribute of a row into one or more relational rows. You can do this by using a TABLE expression with the collection. A TABLE expression enables you to query a collection in the FROM clause like a table. In effect, you join the nested table with the row that contains the nested table without writing a JOIN statement.

The collection column in the TABLE expression uses a table alias to identify the containing table.

You can use a subquery with the TABLE expression:

```
SELECT *
  FROM TABLE(SELECT d.projects
               FROM department d
               WHERE d.dept_id = 10);
```

You can use a TABLE expression in the FROM clause of a SELECT statement embedded in a CURSOR expression:

```
SELECT d.dept_id, CURSOR(SELECT * FROM TABLE(d.projects))
FROM   department d;
```

Working with Collections in SQL

- The Oracle database supports the following DML operations on nested table columns:
  - Inserts and updates that provide a new value for the entire collection
  - Piecewise updates
  - Inserting new elements into the collection
  - Deleting elements from the collection
  - Updating elements of the collection
- The Oracle database does not support piecewise updates on varray columns.
  - Varray columns can be inserted into or updated as atomic units.

DML Operations on Nested Table Columns

You can perform DML operations on nested table columns by providing inserts and updates that supply a new value for the entire collection. Previously, the `pOrder` table was defined with the `ITEMS` column as a nested table.

```
DESCRIBE pOrder
Name                         Null?    Type
-------------------------- -------- -------------
ORDID                                 NUMBER(5)
SUPPLIER                              NUMBER(5)
REQUESTER                             NUMBER(4)
ORDERED                               DATE
ITEMS                                 ITEM_NST_TYP
```

This example inserts rows by providing a new value for the entire collection:

```
INSERT INTO pOrder
  VALUES (500, 50, 5000, sysdate,
     typ_item_nst(typ_item (55, 555)));
1 row created.

INSERT INTO pOrder
  VALUES (800, 80, 8000, sysdate,
    typ_item_nst (typ_item (88, 888)));
1 row created.
```

Working with Collections in SQL

Piecewise DML on nested table columns:

- `INSERT`

```
INSERT INTO TABLE
  (SELECT p.items FROM pOrder p WHERE p.ordid = 500)
VALUES (44, 444);
```

- `UPDATE`

```
UPDATE TABLE
  (SELECT p.items FROM  pOrder p
   WHERE  p.ordid = 800) i
SET     VALUE(i) = typ_item(99, 999)
WHERE   i.prodid = 88;
```

- `DELETE`

```
DELETE FROM TABLE
 (SELECT p.items FROM pOrder p WHERE p.ordid = 500) i
WHERE i.prodid = 55;
```

ORACLE

DML on Nested Table Columns

For piecewise updates of nested table columns, the DML statement identifies the nested table value to be operated on by using the `TABLE` expression.

Using Set Operations on Collections

Oracle Database 10*g* supports the following multiset comparison operations on nested tables:

Set Operation	Description
Equal, Not equal Comparisons	The (=) and (<>) conditions return a Boolean value indicating whether the input nested tables are identical or not.
`IN` Comparisons	The `IN` condition checks whether a nested table is in a list of nested tables.
Subset of Multiset Comparison	The `SUBMULTISET [OF]` condition checks whether a nested table is a subset of another nested table.
Member of a Nested Table Comparison	The `MEMBER [OF]` or `NOT MEMBER [OF]` condition tests whether an element is a member of a nested table.
Empty Comparison	The `IS [NOT] EMPTY` condition checks whether a given nested table is empty or not empty, regardless of whether any of the elements are NULL.
Set Comparison	The `IS [NOT] A SET` condition checks whether a given nested table is composed of unique elements.

Comparisons of Collections

Starting with the Oracle Database 10*g* release, you can use comparison operators and ANSI SQL multiset operations on nested tables.

The conditions listed in this section allow comparisons of nested tables. There is no mechanism for comparing varrays.

In addition to the conditions listed in the slide, you can also use the multiset operations with nested tables:

- The `CARDINALITY` function returns the number of elements in a varray or nested table.
- The `COLLECT` function is an aggregate function which would create a multiset from a set of elements.
- The `MULTISET EXCEPT` operator inputs two nested tables and returns a nested table whose elements are in the first nested table but not in the second nested table.
- The `MULTISET INTERSECT` operator returns a nested table whose values are common in the two input nested tables.
- The `MULTISET UNION` operator returns a nested table whose values are those of the two input nested tables.

This list is not complete. For detailed information, refer to “Support for Collection Data types” in *Oracle Database Application Developer’s Guide - Object-Relational Features.*

Using Set Operations on Collections

```
CREATE OR REPLACE TYPE billdate IS TABLE OF DATE;
/
ALTER TABLE pOrder ADD
 (notice billdate , payments billdate)
    NESTED TABLE notice   STORE AS notice_store_tab
    NESTED TABLE payments STORE AS payments_store_tab;
/
UPDATE pOrder
  SET notice   = billdate('15-JAN-02', '15-FEB-02', '15-MAR-02'),
      payments = billdate('15-FEB-02', '15-MAR-02', '15-JAN-02')
  WHERE ordid = 500;
```

```
SELECT  ordid
FROM    pOrder
WHERE   notice = payments;

 ORDID
------
   500
```

Equality and Non-Equality Predicates (= and <>)

The equal (=) and not equal (<>) operators can be used to compare nested tables. A Boolean result is returned from the comparison. Nested tables are considered equal if they have the same named type, the same cardinality, and their elements are equal. To make nested table comparisons, the element type needs to be comparable.

In this example, the `pOrder` table is altered. Two columns are added. Both columns are nested tables that hold the `DATE` data type. Dates are entered into the two columns for a specific order number. These two collections are compared with the equality predicate.

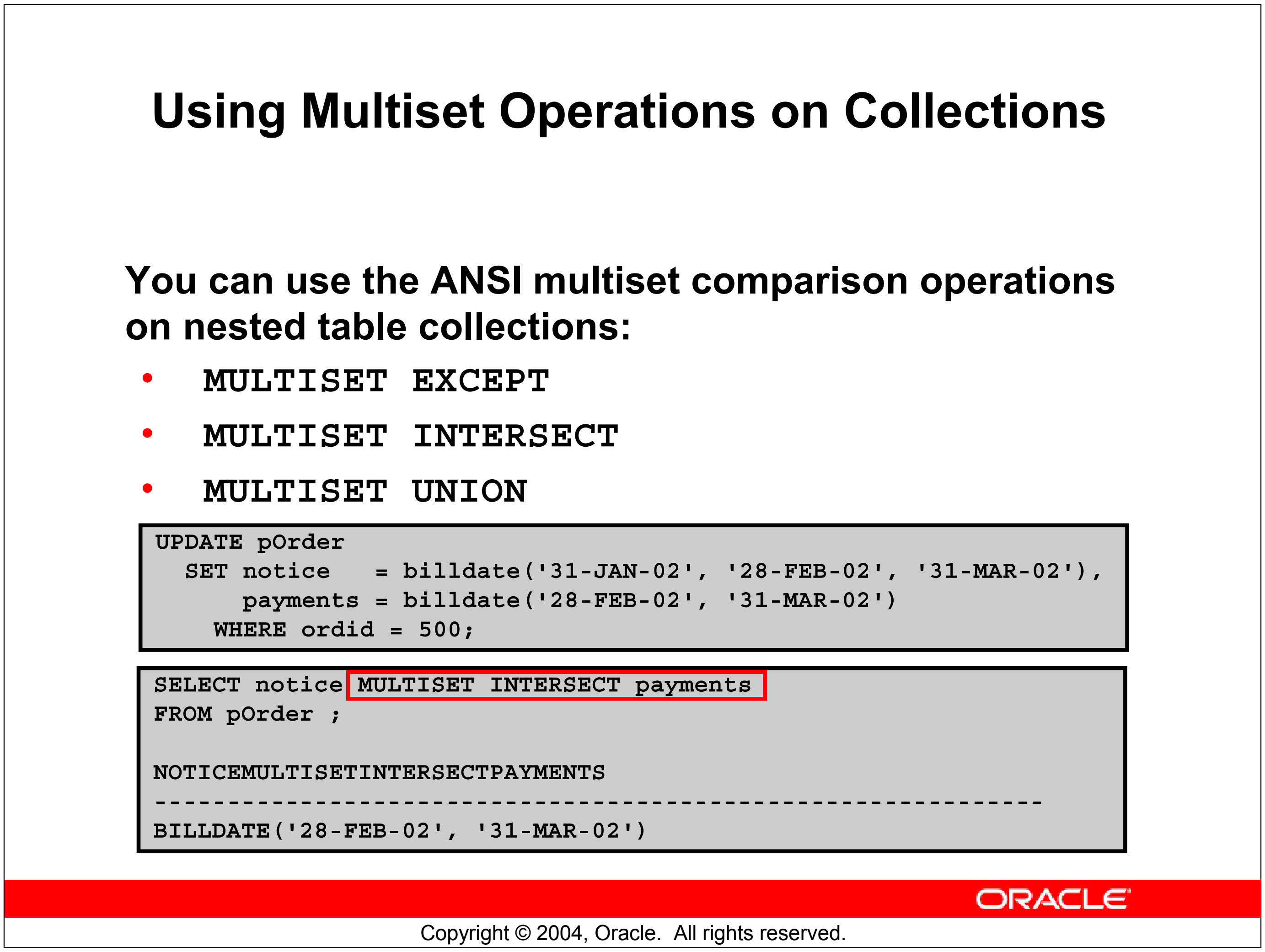
Using Set Operations on Collections

```
CREATE OR REPLACE TYPE typ_billdate IS TABLE OF DATE;
/
DECLARE
  b1       BOOLEAN;
  notice   typ_billdate;
  payments typ_billdate;
BEGIN
  notice   := typ_billdate('15-JAN-02','15-FEB-02','15-MAR-02');
  payments := typ_billdate('15-FEB-02','15-MAR-02','15-JAN-02');
  b1       := notice = payments;
  IF b1 THEN
    dbms_output.put_line('They are equal.');
  ELSE
    dbms_output.put_line('They are NOT equal.');
  END IF;
END;
/

Type created.
They are equal.
PL/SQL procedure successfully completed.
```

Set Operations in PL/SQL

In this example, two nested tables are defined in a PL/SQL block. Both are nested tables of the DATE data type. Date values are entered into the nested tables and the set of values in one nested table is compared using the equality operator with the set of values in the other nested table.



Using Multiset Operations on Collections

You can use the ANSI multiset comparison operations on nested table collections:

- `MULTISET EXCEPT`
- `MULTISET INTERSECT`
- `MULTISET UNION`

```
UPDATE pOrder
  SET notice   = billdate('31-JAN-02', '28-FEB-02', '31-MAR-02'),
      payments = billdate('28-FEB-02', '31-MAR-02')
    WHERE ordid = 500;
```

```
SELECT notice MULTISET INTERSECT payments
FROM pOrder ;

NOTICEMULTISETINTERSECTPAYMENTS
-------------------------------------------------------------
BILLDATE('28-FEB-02', '31-MAR-02')
```

Multiset Operations

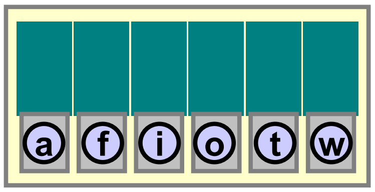
In this example, the `MULTISET INTERSECT` operator finds the values that are common in the two input tables, `NOTICE` and `PAYMENTS`, and returns a nested table with the common results.

By using the `CAST` operator, you can convert collection-typed values of one type into another collection type. You can cast a named collection (such as a varray or a nested table) into a type-compatible named collection. For more information about the `CAST` operator with the `MULTISET` operand, see the topic "CAST" in *Oracle Database SQL Reference 10g Release 1.*

Using String Indexed Associative Arrays

Associative arrays:

- Indexed by strings can improve performance
- Are pure memory structures that are much faster than schema-level tables
- Provide significant additional flexibility



When to Use String Indexed Arrays

Starting with Oracle9*i* Database *Release 2*, you can use `INDEX BY VARCHAR2` tables (also known as string indexed arrays). These tables are optimized for efficiency by implicitly using the B*-tree organization of the values.

The `INDEX BY VARCHAR2` table is optimized for efficiency of lookup on a non-numeric key, where the notion of sparseness is not really applicable. In contrast, `INDEX BY PLS_INTEGER` tables are optimized for compactness of storage on the assumption that the data is dense.

Using String Indexed Associative Arrays

```
CREATE OR REPLACE PROCEDURE report_credit
  (p_last_name    customers.cust_last_name%TYPE,
   p_credit_limit customers.credit_limit%TYPE)
IS
  TYPE  typ_name IS TABLE OF customers%ROWTYPE
    INDEX BY customers.cust_email%TYPE;
  v_by_cust_email   typ_name;
  i VARCHAR2(30);

  PROCEDURE load_arrays IS
  BEGIN
    FOR rec IN  (SELECT * FROM customers WHERE cust_email IS NOT NULL)
      LOOP
        -- Load up the array in single pass to database table.
         v_by_cust_email (rec.cust_email) := rec;
      END LOOP;
  END;

BEGIN
  ...
```

Using String Indexed Arrays

If you need to do heavy processing of customer information in your program that requires going back and forth over the set of selected customers, you can use string indexed arrays to store, process, and retrieve the required information.

This can also be done in SQL, but probably in a less efficient implementation. If you need to do multiple passes over a significant set of static data, you can instead move it from the database into a set of collections. Accessing collection-based data is much faster than going through the SQL engine.

After you have transferred the data from the database to the collections, you can use string- and integer-based indexing on those collections to, in essence, mimic the primary key and unique indexes on the table.

In the REPORT_CREDIT procedure shown, you or a customer may need to determine whether a customer has adequate credit. The string indexed collection is loaded with the customer information in the LOAD_ARRAYS procedure. In the main body of the program, the collection is traversed to find the credit information. The e-mail name is reported in case there is more than one customer with the same last name.

Using String Indexed Associative Arrays

```
...
BEGIN
  load_arrays;
  i:= v_by_cust_email.FIRST;
  dbms_output.put_line ('For credit amount of: ' || p_credit_limit);
  WHILE i IS NOT NULL LOOP
    IF v_by_cust_email(i).cust_last_name = p_last_name
    AND v_by_cust_email(i).credit_limit > p_credit_limit
      THEN dbms_output.put_line ( 'Customer '||
        v_by_cust_email(i).cust_last_name || ': ' ||
        v_by_cust_email(i).cust_email ||  has credit limit of: ' ||
        v_by_cust_email(i).credit_limit);
    END IF;
    i := v_by_cust_email.NEXT(i);
  END LOOP;
END report_credit;
/
```

```
EXECUTE report_credit('Walken', 1200)
For credit amount of: 1200
Customer Walken: Emmet.Walken@LIMPKIN.COM has credit limit of: 3600
Customer Walken: Prem.Walken@BRANT.COM has credit limit of: 3700

PL/SQL procedure successfully completed.
```

Using String Indexed Arrays (continued)

In this example, the string indexed collection is traversed using the `NEXT` method.

A more efficient use of the string indexed collection is to index the collection with the customer e-mail. Then you can immediately access the information based on the customer e-mail key. You would need to pass the e-mail name instead of the customer last name.

Using String Indexed Arrays (continued)

Here is the modified code:

```
CREATE OR REPLACE PROCEDURE report_credit
  (p_email    customers.cust_last_name%TYPE,
   p_credit_limit customers.credit_limit%TYPE)
IS
  TYPE  typ_name IS TABLE OF customers%ROWTYPE
    INDEX BY customers.cust_email%TYPE;
  v_by_cust_email   typ_name;
  i VARCHAR2(30);

  PROCEDURE load_arrays IS
  BEGIN
    FOR rec IN  (SELECT * FROM customers
                 WHERE cust_email IS NOT NULL) LOOP
        v_by_cust_email (rec.cust_email) := rec;
    END LOOP;
  END;

BEGIN
  load_arrays;
  dbms_output.put_line
    ('For credit amount of: ' || p_credit_limit);
  IF v_by_cust_email(p_email).credit_limit > p_credit_limit
        THEN dbms_output.put_line ( 'Customer '||
          v_by_cust_email(p_email).cust_last_name ||
          ': ' || v_by_cust_email(p_email).cust_email ||
          ' has credit limit of: ' ||
          v_by_cust_email(p_email).credit_limit);
  END IF;
END report_credit;
/

EXECUTE report_credit('Prem.Walken@BRANT.COM', 100)
For credit amount of: 100
Customer Walken: Prem.Walken@BRANT.COM has credit limit of:
3700

PL/SQL procedure successfully completed.
```

Summary

In this lesson, you should have learned how to:

- Identify types of collections
  - Nested tables
  - Varrays
  - Associative arrays
- Define nested tables and varrays in the database
- Define nested tables, varrays, and associative arrays in PL/SQL
  - Access collection elements
  - Use collection methods in PL/SQL
  - Identify raised exceptions with collections
  - Decide which collection type is appropriate for each scenario

Summary

Collections are a grouping of elements, all of the same type. The types of collections are nested tables, varrays, and associative arrays. You can define nested tables and in the database. Nested tables, varrays, and associative arrays can be used in a PL/SQL program.

When using collections in PL/SQL programs, you can access collection elements, use predefined collection methods, and use exceptions that are commonly encountered with collections.

There are guidelines for using collections effectively and to determine which collection type is appropriate under specific circumstances.

Practice Overview

This practice covers the following topic:

- **Analyzing collections**
- **Using collections**

Practice Overview

In this practice, you will analyze collections for common errors, then you will create a collection and write a PL/SQL package to manipulate the collection.

For detailed instructions on performing this practice, see Appendix A, "Practice Solutions."

Practice 3

Collection Analysis

1. Examine the following definitions. Run the `lab_03_01.sql` script to create these objects.

```
CREATE TYPE typ_item AS OBJECT  --create object
 (prodid  NUMBER(5),
  price   NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type
  AS TABLE OF typ_item
/
CREATE TABLE pOrder (  -- create database table
     ordid NUMBER(5),
     supplier   NUMBER(5),
     requester  NUMBER(4),
     ordered    DATE,
     items typ_item_nst)
     NESTED TABLE items STORE AS item_stor_tab
/
```

2. The code shown below generates an error. Run the `lab_03_02.sql` script to generate and view the error.

```
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE, NULL);
  -- insert the items for the order created
  INSERT INTO THE (SELECT items
                   FROM   pOrder
                   WHERE  ordid = 1000)
    VALUES(typ_item(99, 129.00));
END;
/
```

Why is the error occurring?

How can you fix the error?

Collection Analysis (continued)

3. Examine the following code. This code produces an error. Which line causes the error, and how do you fix it?
(**Note:** You can run the `lab_03_03.sql` script to view the error output).

```
DECLARE
  TYPE credit_card_typ
  IS VARRAY(100) OF VARCHAR2(30);

  v_mc   credit_card_typ := credit_card_typ();
  v_visa credit_card_typ := credit_card_typ();
  v_am   credit_card_typ;
  v_disc credit_card_typ := credit_card_typ();
  v_dc   credit_card_typ := credit_card_typ();

BEGIN
  v_mc.EXTEND;
  v_visa.EXTEND;
  v_am.EXTEND;
  v_disc.EXTEND;
  v_dc.EXTEND;
END;
/
```

Practice 3 (continued)

Using Collections

In the following practice exercises, you will implement a nested table column in the `CUSTOMERS` table and write PL/SQL code to manipulate the nested table.

4. Create a nested table to hold credit card information.

   Create an object type called `typ_cr_card`. It should have the following specification:

   ```
   card_type  VARCHAR2(25)
   card_num   NUMBER
   ```

   Create a nested table type called `typ_cr_card_nst` that is a table of `typ_cr_card`. Add a column to the `CUSTOMERS` table called `credit_cards`. Make this column a nested table of type `typ_cr_card_nst`. You can use the following syntax:

   ```
   ALTER TABLE customers ADD
   (credit_cards typ_cr_card_nst
      NESTED TABLE credit_cards STORE AD c_c_store_tab);
   ```

5. Create a PL/SQL package that manipulates the `credit_cards` column in the `CUSTOMERS` table.

   Open the `lab_03_05.sql` file. It contains the package specification and part of the package body. Complete the code so that the package:
   - Inserts credit card information (the credit card name and number for a specific customer.)
   - Displays credit card information in an unnested format.

   ```
   CREATE OR REPLACE PACKAGE credit_card_pkg
   IS
     PROCEDURE update_card_info
       (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
      VARCHAR2);
     PROCEDURE display_card_info
       (p_cust_id NUMBER);
   END credit_card_pkg;  -- package spec
   /
   ```

Practice 3 (continued)

Using Collections (continued)

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
 -- cards exist, add more

 -- fill in code here

    ELSE -- no cards for this customer, construct one

 -- fill in code here

    END IF;
  END update_card_info;


  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;

 -- fill in code here to display the nested table
 -- contents

  END display_card_info;
END credit_card_pkg;  -- package body
/
```

Practice 3 (continued)

Using Collections (continued)

6. Test your package with the following statements and output:

```
EXECUTE credit_card_pkg.display_card_info(120)
Customer has no credit cards.
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
    (120, 'Visa', 11111111)
PL/SQL procedure successfully completed.

SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
-------------------------------------------------
  TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111))

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
    (120, 'MC', 2323232323)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
    (120, 'DC', 4444444)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
PL/SQL procedure successfully completed.
```

Practice 3 (continued)

Using Collections (continued)

7. Write a SELECT statement against the credit_cards column to unnest the data. Use the TABLE expression.

   For example, if the SELECT statement returns:

```
SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
--------------------------------------------------------
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111),
  TYP_CR_CARD('MC', 2323232323), TYP_CR_CARD('DC', 4444444))
```

   then rewrite it using the TABLE expression so the results look like:

```
-- Use the table expression so that the result is:
CUSTOMER_ID CUST_LAST_NAME  CARD_TYPE        CARD_NUM
----------- --------------- ------------- -----------
        120 Higgins         Visa             11111111
        120 Higgins         MC             2323232323
        120 Higgins         DC                4444444
```