# Chapter 4: Cursors and Exception Handling

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | **127** | A **cursor** represents a work area or section of memory (also called the ‘context are) where an SQL statement is processed in the Oracle server; this area also contains the results of SQL statements. Cursors are a powerful mechanism for handling multiple rows of data retrieved with an SQL query. | | | | |
| 4 | **127** | **Implicit** **cursors** are declared automatically for all DML and SELECT statements issued in a PL/SQL block | | | | |
| 4 | **127** | **Explicit** **cursors** are declared and manipulated in PL/SQL code for handling rows returned by a SELECT statement. Using **explicit** **cursor** gives you complete control over every aspect of **cursor**. | | | | |
| 4 | **127** | |  |  |  | | --- | --- | --- | | **SQL Cursor Attributes** | | | | Attribute Name | Data type | Description | | SQL%ROWCOUNT | NUMBER | Number of rows affected by the SQL statement | | SQL%FOUND | BOOLEAN | TRUE if at least one row is affected by the SQL statement; otherwise, its FALSE | | SQL%NOTFOUND | BOOLEAN | TRUE if no rows are affected by the SQL statement, otherwise it’s FALSE | | | | | |
| 4 | **130** | After an **error is raised in a PL/SQL block**, processing actually jumps to the last section – **EXCEPTION**. | | | | |
| 4 | 132 | |  |  |  | | --- | --- | --- | | **Steps in using an Explicit Cursor** | | | | Step | Step Activity | Activity Description | | 1 | DECLARE | Creates a named cursor identified by a SELECT statement that doesn’t include an INTO clause. Values in the cursor are moved to PL/SQL variables with the FETCH setup. | | 2 | OPEN | Processes the query and creates the active set or rows available in the cursor. | | 3 | FETCH | Retrieves a row from the cursor into block variables. Each consecutive FETCH retrieves the next row in the cursor until all rows have been retrieved. | | 4 | CLOSE | Clears the active set of rows and frees the memory area used for the cursor. | | | | | |
|  |  | DECLARE  CURSOR cur\_basket IS  SELECT bi.idbasket, p.type, bi.price, bi.quantity  FROM bb\_basket bi JOIN bb\_product p  USING (idproduct) WHERE bi.idbasket = 6;  TYPE type\_basket IS RECORD  (basket bb\_basketitme.idbasket%TYPE, type bb\_product.type%TYPE,  price bb\_basketitem.quantity%TYPE);  rec\_basket type\_basket;  lv\_rate\_num NUMBER(2,2); lv\_tax\_num NUMBER(4,2) := 0;  BEGIN  OPEN cur\_basket;  LOOP  FETCH cur\_basket INTO rec\_basket;  EXIT WHEN cur\_basket%NOTFOUND  IF rec\_basket.type = ‘E’ THEN lv\_rate\_num := 0.05;  ELSIF rec\_basket.TYPE = ‘C’ then lv\_rate\_num := 0.03;  END IF;  lv\_tax\_num := lv\_tax\_num + ((rec\_basket.price \* rec\_basket.qty) \* lv\_rate\_num);  END LOOP;  CLOSE cur\_basket  END | | | Example of cursor for loop  DECLARE  CURSOR cur\_basket IS  SELECT bi.idbasket, p.type, bi.price, bi.quantity  FROM bb\_basket bi JOIN bb\_product p  USING (idproduct) WHERE bi.idbasket = 6;  lv\_rate\_num NUMBER(2,2);  lv\_tax\_num NUMBER(4,2) := 0;BEGIN  OPEN cur\_basket;  FOR rec\_basket IN cur\_basket LOOP  IF rec\_basket.type = ‘E’ THEN lv\_rate\_num := 0.05;  ELSIF rec\_basket.TYPE = ‘C’ then lv\_rate\_num := 0.03;  END IF;  lv\_tax\_num := lv\_tax\_num + ((rec\_basket.price \* rec\_basket.qty) \* lv\_rate\_num);  END LOOP;  CLOSE cur\_basket  END | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| 4 | 135 | A **CURSOR FOR l**oop is another form of coding an explicit cursor that helps you avoid cursor errors in your program. All the cursor handling in the executable section is done with a FOR loop statement. The OPEN, FETCH, EXIT WHEN EMPTY, and CLOSE actions are handled with:  FOR rec\_basket In cur\_basket LOOP  END LOOP; | | | | |
| 4 | **139** | * Some advanced cursor features, such as bulk operations, might not be available with a CURSOR FOR loop. * Optimization applies to queries but not DML statements * When many rows are being processed or the CURSOR FOR loop is being called by many users, bulk processing might still result in more efficient processing. | | | | |
| 4 | **139** | Parameters are values passed to the cursor when it’s opened and used in the cursor’s SELECT statement to determine what data the cursor contains. | | | | |
| 4 | **142** | A **cursor variable** is simply a pointer to a work area where a query can be processed. This method passes the pointer – rather than all the data – to the cursor. Starting with Oracle 9i, a REF CURSOR type named SYS\_REFCURSOR is defined. The processing is handled like an explicit cursor by using OPEN, FETCH, and close statements. However, the OPEN statement for a cursor variable provides the query to be processed.  **DECLARE**  cv\_prod **SYS**\_**REFCURSOR**;  rec\_item bb\_basketitem%ROWTYPE;  rec\_status bb\_basketstatus%ROWTYPE;  lv\_input1\_num NUMBER(2) := 3;  lv\_input2\_num NUMBER(2) := 4;  **BEGIN**  IF lv\_input1\_num = 3 THEN  **OPEN** cv\_prod FOR SELECT \* FROM bb\_basketitem WHERE idbasket = lv\_input1\_num;  LOOP  **FETCH** cv\_prod INTO rec\_item;  EXIT WHEN cv\_prod%NOTFOUND;  DBMS\_OUTPUT.PUT\_LINE(rec\_item.idproduct);  END LOOP;  END IF;  END; | | | | |
| 4 | **142** | The **BULK COLLECT** clause in an explicit cursor’s FETCH statement **improves performance** of multirow queries and DML actions. All data rows are retrieved with one FETCH because of the BULK COLLECT option. You can use **a LIMIT clause** to give you more control over the number of rows processes with the FETCH statement, thereby improving memory use on the database server. | | | | |
|  |  | DECLARE  CURSOR cur\_item IS SELECT \* FROM bb\_basketitem;  TYPE type\_item IS TABLE OF cur\_item%ROWTYPE INDEX BY PLS\_INTEGER;  tbl\_item type item  BEGIN  OPEN cur\_item;  FETCH cur\_item BULK COLLECT INTO tbl\_item;  FOR i IN 1..tbl\_item.COUNT LOOP  DBMS\_OUPUT.PUT\_LINE(tbl\_item(i).idbasketitem)  END LOOP  CLOSE cur\_item;  END; | | DECLARE  CURSOR cur\_item IS SELECT \* FROM bb\_basketitem;  TYPE type\_item IS TABLE OF cur\_item%ROWTYPE INDEX BY PLS\_INTEGER;  tbl\_item type item  BEGIN  OPEN cur\_item;  LOOP  FETCH cur\_item BULK COLLECT INTO tbl\_item LIMIT 1000;  FOR i IN 1..tbl\_item.COUNT LOOP  DBMS\_OUPUT.PUT\_LINE(tbl\_item(i).idbasketitem)  END LOOP  EXIT WHEN cur\_item%NOTFOUND;  END LOOP;  CLOSE cur\_item;  END; | |  |
| 4 | 144 | An **exception handler** is a mechanism for trapping an error that occurs in processing. The **EXCEPTION** section addresses two situations: An Oracle error has been raised or a user-defined error has been raised | | | | |
|  |  | |  |  | | --- | --- | | Partial List of Predefined Exceptions | | | Exception Name | Description | | NO\_DATA\_FOUND | A SELECT statement is a PL/SQL block retrieves no rows or a nonexistent row of an associative array that has been referenced. | | TOO\_MANY\_ROWS | A SELECT statement in a PL/SQL block retrieves more than one row. | | CASE\_NOT\_FOUND | No WHEN clause in the CASE statement is processed. | | ZERO\_DIVIDE | A division by zero is attempted | | DUP\_VALUE\_ON\_INDEX | A violation of unique or primary key column constraint is attempted. | | | | | |
| 4 | 149 | If an exception handler is needed for an Oracle error that is an **undefined exception** you need to declare an exception and associate an Oracle error number with it. A **PRAGMA** statement specifies using additional information that’s supplied when compiling and running the block. | | | | |
|  |  | DECLARE  **ex\_basket\_fk EXCEPTION;**  **PRAGMA EXCEPTION\_INIT(ex\_basket\_fk, -2292);** //exc name and Or #  BEGIN  DELETE FROM bb\_basket WHERE idbasket = 4;  **EXCEPTION**  **WHEN ex\_basket\_fk THEN**  DBMS\_OUTPUT.PUT\_LINE(‘Items still in the basket’);  END; | DECLARE  **ex\_prod\_update EXCEPTION**;  BEGIN  UPDATE bb\_prodcut  SET description = ‘Mill grinder with 5 grind settings!’  WHERE idproduct = 30;  IF SQL%NOTFOUND THEN  **RAISE ex)prod\_update**;  END IF  **EXCEPTION**  **When ex\_prod\_update THEN**  DBMS\_OUTPUT.PUT\_LINE(‘Invalid product ID entered’);  END; | | |  |
| 4 | 151 | A **user-defined exception** is one that a developer raises in a block to enforce a business rule. Oracle doesn’t raise error if UPDATE affects no rows. An exception must be raised in the executable section by using the RAISE command. A declared exception must be referred in the RAISE statement. | | | | |

# Ch 5: Procedures

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | **177** | A **‘named program unit’** or **subprogram** a PL/SQL block that’s been named so that it can be saved on the Oracle server and referenced when needed. The term **‘program unit’** is used for a block of code created to perform a specific task that might be needed in a number of applications. | | |
| 5 | **178** | A **procedure** is used to perform one or more tasks, can return no values or many values, and can be used only PL/SQL statement. A **function** contains a RETURN clause and is used to manipulate data and return a single value. | | |
| 5 | **178** | |  |  | | --- | --- | | **Named Program Unit Types** | | | Program Unit Type | Description | | Stored procedure | Performs a task and can receive and input values as well as return values to the calling program. It’s called explicitly from a program and is stored in the Oracle database. | | Application procedure | Same as a stored procedure except it’s saved in an Oracle application or library on the client side. | | Package | Groups related procedures and functions. It’s called from a program by name and is stored on the server side. | | Database trigger | Performs a task automatically when a DML action occurs on the table. It’s associated with and is stored in the Oracle database. | | Application trigger | Performs a task automatically when a particular event occurs. It’s stored in an Oracle application. | | | |
| 5 | **178** | To make program units flexible, you use **parameters**, which are mechanisms for sending values in and out of a program unit. Procedures can accept and return no, one or many parameters. A **parameter** must be assigned one of three **modes** which indicates which way the value supplied for the parameter flows: into the procedure, out of the procedure, or both. | | |
| 5 | **179** | |  |  | | --- | --- | | **Parameter Mode Types** | | | Mode | Description | | IN | Passes a value from the application environment to the procedure. This value is considered a constant because it can’t be changed in the procedure. This mode is the default if no mode is indicated | | OUT | Passes a value from the procedure to the application environment. If values are calculated or retrieved from the database in the procedure. OUT parameters are used to return those values to the calling environment. | | IN OUT | Allows passing a value in and out with the same parameter. The value sent out can be different from the value sent in. | | | |
| 5 | **179** | **CREATE OR REPLACE PROCEDURE**  procedure name  (param1\_name[mode] data\_type) is  Declaration section  BEGIN  Executable section  EXCEPTION  Exception handlers  END;  **CREATE OR REPLACE PROCEDURE** SHIP\_COST\_SP (p\_qty IN NUMBER,  p\_ship OUT NUMBER) AS  BEGIN  IF p\_qty>10 THEN p\_ship := 11.00;  ELSIF p\_qty>5 THEN p\_ship:= 8.00;  ELSE p\_ship := 5.00;  END IF;  END SHIP\_COST\_SP;/ | CREATE OR REPLACE PROCEDURE ORDER\_TOTAL\_SP  (p\_basketid IN bb\_basketitem.idbasket%TYPE,  p\_cnt OUT NUMBER,  p\_sub OUT NUMBER,  p\_ship OUT NUMBER,  p\_total OUT NUMBER)  AS  BEGIN  Dbms\_output('order total proc called');  SELECT SUM(quantity), SUM(quantity \* price)  INTO p\_cnt, p\_sub  FROM bb\_basketitem  WHERE idbasket = p\_basketid;  ship\_cost\_sp(p\_cnt, p\_ship);  p\_total := NVL(p\_sub,0)+NVL(p\_ship,0);  DBMS\_OUTPUT.PUT\_LINE('order total proc ended');  END ORDER\_TOTAL\_SP; | DECLARE  lv\_basket\_num bb\_basketitem.idbasketitem%TYPE := 16;  lv\_cnt\_num NUMBER(3);  lv\_sub\_num NUMBER(8,2);  lv\_ship\_num NUMBER(8,2);  lv\_total\_num NUMBER(8,2);  BEGIN  order\_total\_sp(lv\_basket\_num, lv\_cnt\_num, lv\_sub\_num,  lv\_ship\_num, lv\_total\_num);  dbms\_output(lv\_cnt\_num);  dbms\_output (lv\_sub\_num);  dbms\_output (lv\_ship\_num);  dbms\_output (lv\_total\_num);  END; |
| 5 | **180** | Values from the application that are passed to parameters are called **actual parameters**; parameters in a procedure declaration are **called formal parameters**. No size or precision information is included in parameter’s data types. The %TYPE and %ROWTYPE attributes can be used to provide a data type based on a table column. | | |
| 5 | **186** | The arguments used in the procedure call were passed with a **positional method**. **Named association** associates a value for each parameter by name in the calling statement. | | |
| 5 | **187** | With the IN OUT mode, the parameter can accept a value when the procedure is called, and this value can be changed in the procedure so that a different value could be returned to the calling environment | | |
| 5 | 196 | **Variable scope** specifies the area of a block that can identify a particular variable. | | |
| 5 | **198** | Nesting a block raises a new issue with variable scope, which specifies the area of a block that can identify a particular variable. | | |
| 5 | **198** | If an exception is raised in procedure B (which has been called from procedure A), control first moves to the EXCEPTION section of procedure B. If the exception is handled, control returns to procedure A. If the exception isn’t handled in procedure B, control moves to the EXCEPTION section of procedure A and looks for a matching handler. | | |

# Chapter 6 Functions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | **217** | A **function** is a program unit that performs a task, can receive input values, and returns values to the calling environment. The main difference between functions and procedures is that a function is part of an expression. Unlike procedures, it can’t serve as entire statement. This differenced is important because it means functions can be used in both SQL command and PL/SQL statements. | | |
| 6 | **218** | The syntax for a function is much like a procedure except **a RETURN statement** must be used to handle the function is to return. | | |
| 6 | **218** | **CREATE (OR REPLACE) FUNCTION function\_name**  [(parameter\_name (mode) data type)]  RETURN return\_value data type IS/AS  declaration section  BEGIN  executable section  RETURN variable\_name;  EXCEPTION  exception handlers  END; | CREATE FUNCTION ship\_calc\_sf(p\_qty IN NUMBER)  RETURN NUMBER **//declare data type**  IS  lv\_ship\_num NUMBER(5,2);  BEGIN  IF p\_qty>10 THEN lv\_ship\_num := 11.00;  ELSIF p\_qty>5 THEN lv\_ship\_num := 8.00;  ELSE lv\_ship\_num := 5.00;  END IF;  RETURN lv\_ship\_num; **//value to return**  END;/ | DECLARE  lv\_cost\_num NUMBER(5,2);  BEGIN  lv\_cost\_num := ship\_calc\_sf(12);  DBMS\_OUTPUT.PUT\_LINE(lv\_cost\_num);  END; |
| 6 | **219** | After the parameters listed in the header, a RETURN statement indicates the returned values data type. At least one RETURN statement must be included in the function body to specify which value to return. | | |
| 6 | 221 | Using a function in a SQL statement:  SELECT idbasket, quantity, shipping actual, ship\_calc\_sf(quantity) calc, ship\_calc\_sf(quantity) - shipping diff  FROM bb\_basket  WHERE orderplaced = 1; | | |
| 6 | **228** | A **RETURN statement** can also be used in a **procedure**, but it serves a different purpose than it does in a function. In a procedure, it’s used to control the flow of execution. It includes no arguments and is followed with a semicolon. When the RETRUN statement is executed, the procedure immediately return execution to the next statement after the procedural call. | | |
| 6 | **230** | IN parameter values **are passed by reference** means a pointer to the value in the actual parameter is created instead of copying the value from the actual parameter to the formal parameter. OUT and IN OUT parameters are passed by value, meaning the value is copied from the actual parameter to the formal parameter. This method is the default behavior. | | |
| 6 | **230** | You can override passing by value with the **NOCOPY** compiler hint. A **compiler hint** is a request a programmer includes in code asks Oracle to modify the fault processing in some manner. | | |

# Chapter 7 PL/SQL Packages

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | **249** | A **package** is a container that can hold multiple program units, such as procedures and function. It helps organize program units into related groups, establish private program units, share variables across program units, overload program units and improve performance | | |
| 7 | **250** | A **package** consists of two parts: A specification and a body. The **package specification** declares the package’s contents and is referred to as the ‘package header’. A specification is required and must be created before the body. **The package body** contains the full code for all objects in the package, such as procedures and functions. | | |
| 7 | **250** | A **package specification** can contain declarations for procedures, functions, variables, exceptions, cursors, and types. | | |
|  |  | create or replace  PACKAGE ordering\_pkg  IS  pv\_total\_num NUMBER (3, 2) := 0;  pv\_bonus\_num NUMBER (3, 2) := 0;  **FUNCTION** ship\_calc\_pf (p\_qty IN NUMBER)  RETURN NUMBER;  **PROCEDURE** order\_total\_pp  (p\_bsktid IN bb\_basketitem.idbasket%TYPE,  p\_cnt OUT NUMBER,  p\_sub OUT NUMBER,  p\_ship OUT NUMBER,  p\_total OUT NUMBER);  END ordering\_pkg; | **create or replace PACKAGE BODY ordering\_pkg IS**  **FUNCTION** ship\_calc\_pf (p\_qty IN NUMBER)  RETURN NUMBER  IS  lv\_ship\_num NUMBER(5,2);  BEGIN  IF p\_qty > 10 THEN lv\_ship\_num := 11.00;  ELSIF p\_qty > 5 THEN lv\_ship\_num := 8.00;  ELSE lv\_ship\_num := 5.00;  END IF;  RETURN lv\_ship\_num;  END ship\_calc\_pf;  END ordering\_pkg;  -----------------------------------------  **PROCEDURE** order\_total\_pp  (p\_bsktid IN bb\_basketitem.idbasket%TYPE,  p\_cnt OUT NUMBER,  p\_sub OUT NUMBER,  p\_ship OUT NUMBER,  p\_total OUT NUMBER**) IS**  BEGIN  SELECT SUM(quantity),SUM(quantity\*price)  INTO p\_cnt, p\_sub  FROM bb\_basketitem  WHERE idbasket = p\_bsktid;  p\_sub := p\_sub - (p\_sub \* pv\_bonus\_num);  p\_ship **:= ship\_calc\_pf(p\_cnt);**  p\_total := NVL(p\_sub,0) + NVL(p\_ship,0);  END order\_total\_pp; | DECLARE  lv\_bask\_num bb\_basketitem.idbasket%TYPE := 12;  lv\_cnt\_num NUMBER(3);  lv\_sub\_num NUMBER(8,2);  lv\_ship\_num NUMBER(8,2);  lv\_total\_num NUMBER(8,2);  BEGIN  **ordering\_pkg.order\_total\_pp(lv\_bask\_num, lv\_cnt\_num, lv\_sub\_num, lv\_ship\_num, lv\_total\_num);**  DBMS\_OUTPUT.PUT\_LINE(lpad('No. of items ordered: ',25) || lv\_cnt\_num);  DBMS\_OUTPUT.PUT\_LINE(lpad('Subtotal: ',25) || lv\_sub\_num);  DBMS\_OUTPUT.PUT\_LINE(lpad('Shipping cost: ',25) || lv\_ship\_num);  DBMS\_OUTPUT.PUT\_LINE(lpad('Total cost: ',25) || lv\_total\_num);  END;/ |
| 7 | **252** | The order of items declared in a specification isn’t important – unless one declaration item is referenced by another. The referenced item must be declared first. | | |
| 7 | **252** | A **package body** is the program unit containing the code for any procedures and functions declared in the specification. It must be created with the same name as the specification to tie the specification and body code together. In addition, all code in the package body’s procedure and function header sections must match declarations in the specification exactly. | | |
| 7 | **257** | **Package scope** is the range of visibility for a package element. Any elements declared in a package specification are considered public. | | |
| 7 | **264** | Package code is **cached**, as is package data, such as variables and cursors; After the first call, all other calls are processed much faster because the SQL query doesn’t have to be processed again.  **A forward declaration** declares a program un it in a package boy by placing the header code at the top of the package body’s code. | | |
| 7 | **272** | In packages, **overloading** is the capability to use the same name for multiple program units in the same package. For **overloading** to work, the formal parameters in the procedures or functions must differ in at least one of the following categories**: total number of formal parameters, data type family, or listed order.** | | |
| 7 | **272** | When compiling objects that call a package’s program units, only the package specification or program unit header is used for verification. For example, if you create a procedure that uses a packaged function in a SQL statement, when the procedure is compiled, only the function header information is reviewed to see whether it exists and has the correct parameters so the PL/SQL compiler doesn’t see the entire block of code for the function and can’t determine whether the restrictions on functions in SQL statements are being followed. | | |