Stored Procedures: Oracle's PL/SQL

How to Use Stored Procedures with Oracle's PL/SQL

Overview

- What are stored procedures?
 - Why do we need them?
- How stored procedures are used
- PL/SQL:
 - Language basics
 - Procedures & Functions
 - Database Operations and Cursors
 - Packages

11/19/02

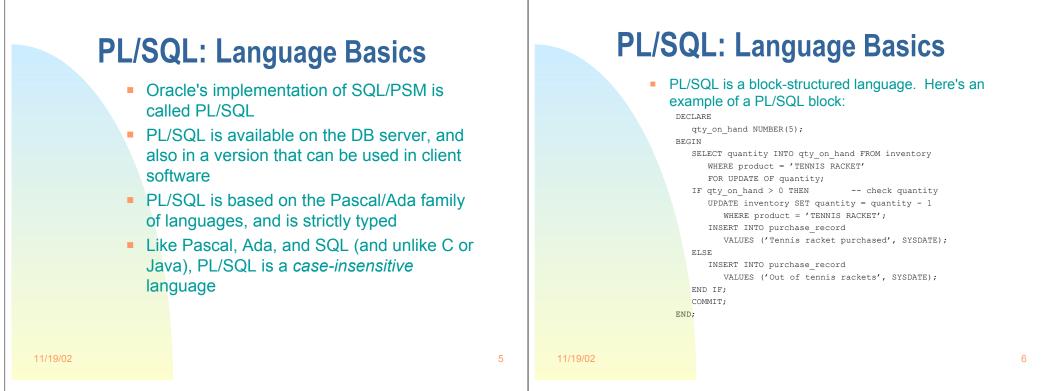
What are Stored Procedures?

- Simple: Procedures that are stored in the database, and executed there
 - The ISO SQL standard calls them SQL Persistent Stored Modules (SQL/PSM)
- Why are stored procedures good?
 - They improve abstraction
 - They improve performance
 - They improve maintainability
 - They improve security

How Stored Procedures are Used

- Write the procedure
- Test it (either locally, or in a test DB)
- Store it in the DB (in Oracle, use SQL*Plus)
- Grant controlled access to it
- Authorized DB clients can then call it:
 - Call is done on the client machine
 - Call is transferred to the DB, along with any parameters
 - Procedure is executed on the DB
 - Any data generated by the procedure is transferred back to the client

3



PL/SQL: Language Basics

- A PL/SQL block consists of:
 - A Declaration Section (optional -- starts with DECLARE)
 - An Execution Section (starts with BEGIN)
 - Within the Execution Section, an (optional) Exception Section (starts with EXCEPTION)
- A PL/SQL block ends with END;
- PL/SQL blocks may nest (i.e., you can have one or more blocks within a block)

PL/SQL: Language Basics

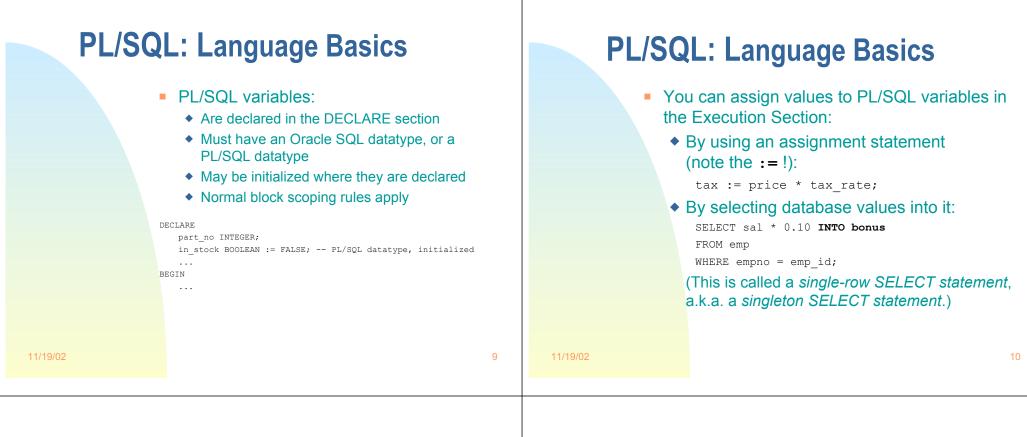
- PL/SQL has two kinds of comments:
 - Single-line comments: salary := salary + salary * 0.1;

-- Give 10% bonus

Multi-line comments:

/*

You should always place block-style comments before every procedure or function definition, describing its use, parameters and any return value. */



PL/SQL: Language Basics

- The PL/SQL IF statement has three forms:
 - ♦ IF-THEN IF <condition> THEN . . . END IF; ♦ IF-THEN-ELSE
 - IF <condition> THEN ELSE . . . END IF;
- IF-ELSEIF IF <condition-1> THEN . . . ELSEIF <condition-2> THEN . . . ELSEIF <condition-N> THEN . . . [ELSE ...]

END IF;

PL/SQL: Language Basics



• Simple loop: LOOP

> . . . END LOOP;

 Numeric FOR loop: FOR <loop-index> IN [REVERSE] <low-num>...<hi-hum> LOOP

. . . END LOOP;

and two forms of exit from loop execution:

 Unconditional: EXIT [<label>];

- Cursor FOR loop: FOR <record-index> IN <cursor-name> LOOP
 - . . .
 - END LOOP;
- WHILE loop: WHILE <condition> LOOP . . .

END LOOP;

 Conditional: EXIT [<label>] WHEN <condition>;

11

PL/SQL: Language Basics **PL/SQL:** Language Basics There are four kinds of exceptions in PL/SQL: The optional Exception Section of a PL/SQL block contains Named system exceptions one or more Exception (WHEN) Handlers: Exceptions that have been declared by PL/SQL (in the DECLARE STANDARD PL/SQL package), and raised as a result of an error in PL/SQL or DB processing . . . BEGIN Named programmer-defined exceptions . . . Exceptions that are declared by the programmer, and raised EXCEPTION explicitly as a result of errors in application code. WHEN <exception-name> [OR <exception-name>]... Unnamed system exceptions THEN <executable-statements> Exceptions that are not declared by PL/SQL, but can be raised [WHEN <exception-name> [OR <exception-name>]... as a result of an error in PL/SQL or DB processing THEN Unnamed programmer-defined exceptions <executable-statements>]... Exceptions that are declared using an error number [WHEN OTHERS (between -20000 and -20999) and a message, and raised THEN on the server by the programmer using a <executable-statements>] RAISE APPLICATION ERROR call. END; 13 11/19/02 14

PL/SQL: Language Basics

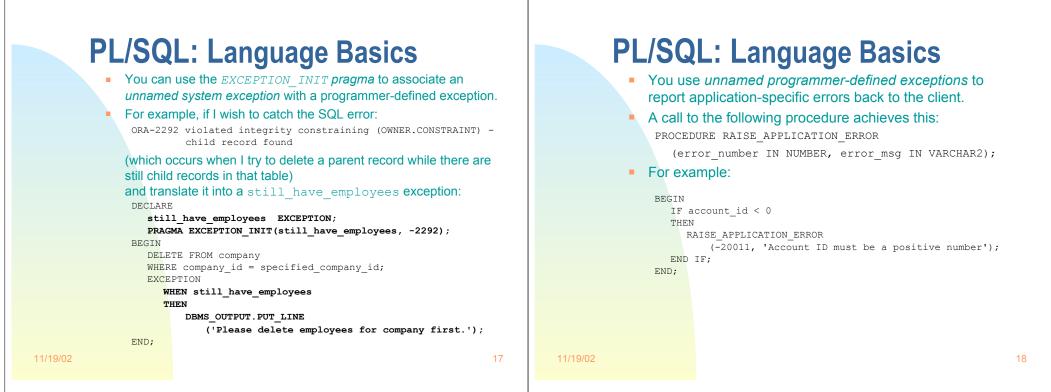
- Here are some named system exceptions:
 - CURSOR_ALREADY_OPEN
 - DUP_VAL_ON_INDEX
 - INVALID_CURSOR
 - INVALID_NUMBER
 - LOGIN_DENIED
 - NO_DATA_FOUND
 - NOT_LOGGED_ON
 - PROGRAM_ERROR
 - STORAGE_ERROR
 - TIMEOUT_ON_RESOURCE
 - TOO_MANY_ROWS
 - TRANSACTION_BACKED_OUT
 - VALUE_ERROR
 - ZERO_DIVIDE

PL/SQL: Language Basics

 Here is an example of the use of named programmerdefined exceptions:

DECLARE

inva	lid_account_no		EXCEPTION;
acco	unt balance negat	cive	EXCEPTION;
BEGIN			
<e.< th=""><th>xecutable stateme</th><th>ents></th><th></th></e.<>	xecutable stateme	ents>	
IF b	alance < 0		
THEN			
RA	AISE account_bala	nce_ne	gative;
END	IF;		
<e< th=""><th>xecutable stateme</th><th>ents></th><th></th></e<>	xecutable stateme	ents>	
EXCE	PTION		
WF	HEN invalid accou	nt no	
TH	HEN	_	
	<executable star<="" th=""><th>tements</th><th>></th></executable>	tements	>
WH	HEN account_balan	ce_nega	ative
TH	HEN		
	<executable star<="" th=""><th>tements</th><th>></th></executable>	tements	>
END;			



PL/SQL: Database Interactions

- You can execute SQL statements within a PL/SQL block.
- Normally, transactions are begun implicitly with the first SQL statement executed.
- You can specify the attributes of a transaction using a SET TRANSACTION statement:

SET TRANSACTION READ ONLY; SET TRANSACTION READ WRITE; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

PL/SQL: Database Interactions

 You can *commit* or *rollback* a transaction within a PL/SQL block:

COMMIT [WORK];

ROLLBACK [WORK];

PL/SQL: Database Interactions

- You can execute DML statements within a PL/SQL block:
 - INSERT, DELETE, and UPDATE statements can be executed in-line, normally.
 - SELECT ... INTO statements (*single-row select statements*) can be executed in-line, normally.
 - SELECT statements that [possibly] return multiple rows cannot be executed normally. Because such statements return sets of values, and PL/SQL is not a set-oriented language, they have to be handled specially, using *cursors*.
 - SQL statements in a PL/SQL block may refer to PL/SQL variables visible to that block

11/19/02

PL/SQL: Database Interactions

- A cursor is like a pointer into a table in the database.
- You *declare a cursor* in the declaration section of a PL/SQL block:

DECLARE

CURSOR employee_cursor IS SELECT * FROM employee;

- Then you use the cursor declaration in the execution section:
 - Use an OPEN statement to open the cursor
 - Use FETCH statements to fetch rows using the cursor
 - When done, use a CLOSE statement to close the cursor and release its resources. (Note: Locks, as usual, are not normally released until the transaction is committed or rolled back.)

11/19/02

21

PL/SQL: Database Interactions

Alternatively, you can use a *cursor* FOR loop in the execution section:

DECLARE

CURSOR employee_cursor IS SELECT * FROM employee; employee_record employee_cursor%ROWTYPE;

BEGIN

FOR employee_record IN employee_cursor LOOP

- -- Access column data in the employee record
- -- for the current row, and use it to execute
- -- other PL/SQL statements, including other SQL -- statements.

END LOOP;

END;

PL/SQL: Database Interactions

 To obtain information about the current status of your cursor, you use *cursor attributes*:

%FOUND	Returns TRUE if the record was fetched successfully, FALSE
	otherwise
%NOTFOUND	Returns TRUE if the record was not fetched successfully, FALSE
	otherwise
%ROWCOUNT	Returns the number of records that have been fetched from the
	cursor
%ISOPEN	Returns TRUE if the cursor is open, FALSE otherwise

PL/SQL: Database Interactions

```
    So far, we've just talked about PL/SQL blocks.

Here's an example of using cursor attributes:
    DECLARE
                                                                                          There are three kinds of "top-level" blocks:
      CURSOR employee cursor IS SELECT * FROM employee;

    An anonymous block

       employee record employee cursor%ROWTYPE;

    You can use an anonymous block directly in a client

    BEGIN
                                                                                                 program. It gets passed to the database for execution,
      IF NOT employee cursor%ISOPEN
                                                                                                 and its results passed back to the client. However, it
      THEN
                                                                                                 doesn't get stored in the database.
         OPEN employee_cursor;

    A procedure or function

      END IF;
      WHILE employee cursor%FOUND
                                                                                               In order to store executable code in the database, you
      LOOP
                                                                                                 have to use PL/SQL procedures and/or functions.
         DBMS OUTPUT.PUT LINE

    The basic difference between procedures and functions is

             ('Fetched record number ' ||
                                                                                                 that a function returns a single value, while a procedure
                TO CHAR (employee cursor%ROWCOUNT) );
                                                                                                 does not return any value.
         FETCH employee cursor INTO employee record;
       END LOOP;
      CLOSE employee cursor;
    END;
                                                                 25
```

PL/SQL: Procedures & Functions

```
    A PL/SQL procedure looks as follows:
        PROCEDURE <name> [ ( <parameter> [, <parameter> ... ] ) ]
        IS
            <declarations>
        BEGIN
            <executable statements>
            [ EXCEPTION
                 <exception handler> [ <exception handler> ] ... ]
        END [ <name> ] ;
        Note that the keyword DECLARE disappears in a
```

procedure, replaced by the keyword IS.

PL/SQL: Procedures & Functions

PL/SQL: Procedures & Functions

<exception handler> [<exception handler>] ...]
END [<name>];

 The return datatype of a function may be any datatype (and sometimes complex structures) supported by PL/SQL.

PL/SQL: Procedures & Functions

- A PL/SQL procedure or function may accept zero or more parameters.
- If the procedure or function has zero parameters, both the procedure/function definition and a call to it dispense with the parentheses. (This is the Pascal/Ada style.)

```
PROCEDURE do work
                      -- procedure definition
IS
BEGIN
  do more work;
                      -- call to another procedure
END doWork ;
FUNCTION does nothing RETURN BOOLEAN IS
BEGIN
  RETURN does even less; -- call function, returns value
```

29

END;

PL/SQL: Procedures & Functions

A parameter for a PL/SQL procedure or function has the following form: <parameter-name> [<parameter-mode>] <parameter-type> where <parameter-mode> is: IN | OUT | IN OUT The parameter mode may be one of: -- (the default) specifies the parameter is read-only IN -- specifies the parameter is write-only ◆ OUT • IN OUT -- specifies the parameter is read-write For example: PROCEDURE predict activity (last date IN DATE, -- input only task desc IN OUT VARCHAR2, -- input and output next date out OUT DATE) -- output only 11/19/02 IS ... 30

PL/SQL: Procedures & Functions

```
A parameter for a PL/SQL procedure or function (or any other
PL/SQL variable declaration) can specify a datatype:
```

- ◆ A SQL datatype: INTEGER, FLOAT, VARCHAR, etc.
- A PL/SQL datatype: BOOLEAN, a record type, etc.
- An anchored datatype:

```
<variable-name> <type-attribute>%TYPE
```

```
where <type-attribute> can be any of the following:
```

- A previously declared PL/SQL variable name
- A table column in the format "table.column"

For example:

total sales NUMBER(20,2);

```
monthly sales total sales%TYPE;
```

```
comp id company.company id%TYPE;
```

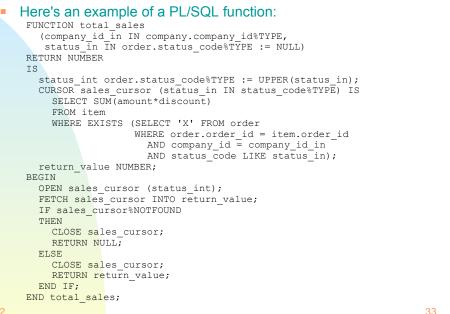
```
You can also anchor to a NOT NULL datatype
```

(PL/SQL variables can be declared to be NOT NULL, as well as columns in tables.)

PL/SQL: Procedures & Functions

```
Here's an example of a PL/SQL procedure:
 PROCEDURE apply discount
   (company id in IN company.company id%TYPE,
    discount in IN NUMBER)
 TS
   min discount CONSTANT NUMBER := .05;
   max discount CONSTANT NUMBER := .25;
   invalid discount EXCEPTION;
 BEGIN
   IF discount in BETWEEN min discount AND max discount
   THEN
     UPDATE item
       SET item amount = item amount* (1-discount in)
      WHERE EXISTS (SELECT 'x' FROM order
                   WHERE order.order id = item.order id
                     AND order.company id = company id in);
     IF SQL%ROWCOUNT = 0
     THEN
       RAISE NO DATA FOUND;
     END IF;
   ELSE
     RAISE invalid discount;
   END IF;
   EXCEPTION
     WHEN invalid discount
     THEN DBMS OUTPUT.PUT LINE ('The specified discount is invalid');
     WHEN NO DATA FOUND
     THEN DBMS OUTPUT.PUT LINE ('No orders for company: ' ||
                                TO_CHAR(company_id_in));
 END apply_discount;
```

PL/SQL: Procedures & Functions



PL/SQL: Packages

- It is a good idea to organize your PL/SQL procedures and functions (and other objects) into one or more packages.
 There are two parts to a package:

 The package specification -- the declaration of the package interface:
 PACKAGE <package-name>
 [declarations of variables and types]
 [specifications of modules]
 END <package body -- the implementation
 PACKAGE BODY <package-name>
 [declarations of variables and types]
 [declarations of variables and types]

END <package-name>;

11/19/02

PL/SQL: Packages

- A major benefit of packages is that they provide modularization of your procedures, functions, cursors, variables, etc.
- The specification defines the *public* parts of the package -- those that are visible to the outside world.
- The package body defines the *private* parts of the package -- those that are not visible to the outside world. This allows the implementation to be private, and perhaps changed over time.

PL/SQL: Packages

- When calling PL/SQL procedures and functions that reside inside a package:
 - From inside the same package:

call_me(arg1, arg2);

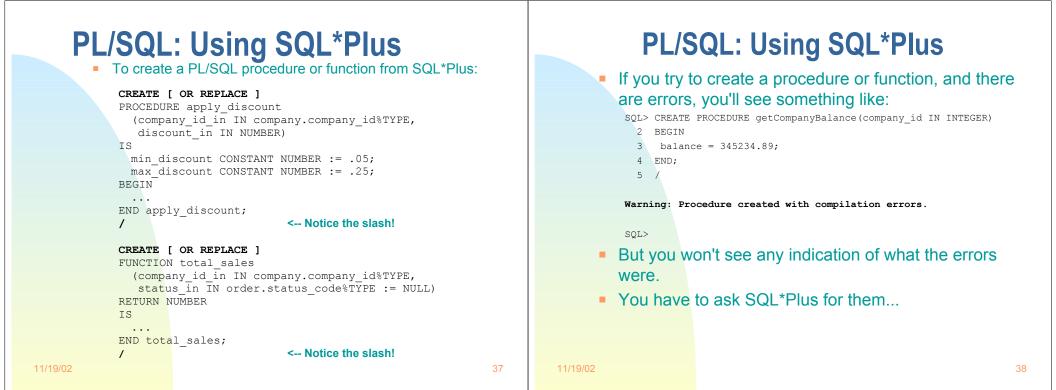
• From outside the package, but from within the same schema as the stored package:

my_package.call_me(arg1, arg2);

From outside the package, and outside the stored package schema:

my_schema.my_package.call_me(arg1, arg2);

 When calling PL/SQL procedures and functions that do not reside inside a package, *omit the package name*.



PL/SQL: Using SQL*Plus

The SHOW ERRORS command will give you more information:

SOL> show errors

Errors for PROCEDURE GETCOMPANYBALANCE:

LINE/COL ERROR

2/1 PLS-00103: Encountered the symbol "BEGIN" when expecting one of the following:

; is with authid deterministic parallel_enable as

- We left out an IS keyword.
- However, this is only one error of several, so let's fix it, and try again...

PL/SQL: Using SQL*Plus

Try again:

SQL> CREATE PROCEDURE getCompanyBalance(company_id IN INTEGER) IS
2 BEGIN

- 3 balance = 345234.89;
- 4 END;

5 /

CREATE PROCEDURE getCompanyBalance(company_id IN INTEGER) IS

```
ERROR at line 1:
```

ORA-00955: name is already used by an existing object

 Whoops! Even though there were compilation errors, it still stored the procedure in the database schema.

11/19/02

PL/SQL: Using SQL*Plus **PL/SQL: Using SQL*Plus** Try yet again: • Try SHOW ERRORS again: SQL> CREATE OR REPLACE SOL> show errors 2 PROCEDURE getCompanyBalance(company id IN INTEGER) IS Errors for PROCEDURE GETCOMPANYBALANCE: 3 BEGIN balance = 345234.89; LINE/COL ERROR 5 END; 6 / 3/10 PLS-00103: Encountered the symbol "=" when expecting one of the following: Warning: Procedure created with compilation errors. := . (@%; What's wrong now? Ask SQL*Plus again... The symbol ":= was inserted before "=" to continue. OK, let's fix the := problem, and try again... 41 11/19/02 42

PL/SQL: Using SQL*Plus

Try again:

SQL> CREATE OR REPLACE

- 2 PROCEDURE getCompanyBalance(company_id IN INTEGER) IS
- 3 BEGIN
- 4 balance := 345234.89;
- 5 END;
- 6 /

Warning: Procedure created with compilation errors.

What's wrong now? Let's try that SHOW ERRORS again...

PL/SQL: Using SQL*Plus

Try SHOW ERRORS again:

SQL> show errors Errors for PROCEDURE GETCOMPANYBALANCE:

LINE/COL ERROR

- 3/2 PLS-00201: identifier 'BALANCE' must be declared
- 3/2 PL/SQL: Statement ignored

• OK, let's fix that, and hope it's the last one...

PL/SQL: Using SQL*Plus

Try again:

SQL> CREATE OR REPLACE

- 2 PROCEDURE getCompanyBalance(company_id IN INTEGER) IS
- 3 balance REAL;
- 4 BEGIN
- 5 balance := 345234.89;
- 6 END;
- 7 /

Procedure created.



11/19/02

PL/SQL: Using SQL*Plus

	Of course, we forgot that, if we want to create a procedure
	to return a value, it can't be a procedure, but must be a
	function. So let's change it:
	SQL> CREATE OR REPLACE
	<pre>2 FUNCTION getCompanyBalance(company_id IN INTEGER)</pre>
	3 RETURN REAL
	4 IS
	5 balance REAL;
	6 BEGIN
	7 balance := 345234.89;
	8 RETURN balance;
	9 END;
	10 /
	CREATE OR REPLACE
	*
	ERROR at line 1:
	ORA-00955: name is already used by an existing object
	Rats!!!!
45	40

PL/SQL: Using SQL*Plus

We must first drop the procedure, before we can create a function of the same name:

SQL> DROP PROCEDURE getCompanyBalance;

Procedure dropped.

SQL> CREATE OR REPLACE

- 2 FUNCTION getCompanyBalance(company_id IN INTEGER)
- 3 RETURN REAL
- 4 IS
- 5 balance REAL;
- 6 BEGIN
- 7 balance := 345234.89;
- 8 RETURN balance;
- 9 END;
- 10 /

Function created.

Eureka!

PL/SQL: Using SQL*Plus

- What if we want to interact with the procedure or function from within SQL*Plus?
- There is a special PL/SQL package, DBMS_OUTPUT. It contains the following procedures:
 - PUT (a VARCHAR2)
 - PUT (a NUMBER)
 - PUT (a DATE)
 - PUT_LINE (a VARCHAR2)
 - PUT_LINE (a NUMBER)
 - PUT_LINE(a DATE)
 - NEW_LINE
 - and a number of others
- They allow the PL/SQL procedure or function to communicate with the client.

PL/SQL: Using SQL*Plus

Let's try it:

SQL> CREATE OR REPLACE

- 2 PROCEDURE helloWorld
- 3 IS
- 4 BEGIN
- 5 DBMS_OUTPUT.PUT_LINE('Hello, PL/SQL world!');
- 6 END;
- 7 /

Procedure created.

SQL> execute helloWorld;

PL/SQL procedure successfully completed.

So why didn't we get any output?

11/19/02

49

PL/SQL: Using SQL*Plus

It turns out you need to tell SQL*Plus to take notice of the server output:

SQL> set serveroutput on SQL> execute helloWorld; Hello, PL/SQL world!

PL/SQL procedure successfully completed.

Finally!

11/19/02

PL/SQL: Using SQL*Plus

- What if you have created your PL/SQL object in the schema, but you don't have its source handy?
- No problem; you can query the database to get the source:

SQL> SELECT TEXT

- 2 FROM USER_SOURCE
- 3 WHERE name = 'HELLOWORLD'
- 4 AND type = 'PROCEDURE'
- 5 ORDER BY LINE;

TEXT

PROCEDURE helloWorld

IS

BEGIN
DBMS_OUTPUT.PUT_LINE('Hello, PL/SQL world!');
END;

PL/SQL: Privileges Required

- In order to create a procedural object, you must have the CREATE PROCEDURE system privilege (which is part of the RESOURCE role)
- If the procedural object will be *placed in another user's* schema, then you must have CREATE ANY PROCEDURE privilege
- To allow another user to execute your procedural object, that user must be granted EXECUTE privilege on the object:

GRANT EXECUTE ON my_procedure TO bryan;

 Bryan will then be able to execute that procedure, even if he does not have privileges on any of the tables which the procedure uses.

51