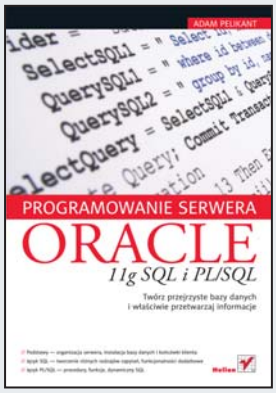
Programowanie serwera Oracle 11g SQL i PL/SQL



Autor: Adam Pelikant
ISBN: 978-83-246-2429-4
Format: 158×235, stron: 336

Twórz przejrzyste bazy danych i właściwie przetwarzaj informacje

- Podstawy – organizacja serwera, instalacja bazy danych i końcówki klienta
- Język SQL – tworzenie różnych rodzajów zapytań, funkcjonalności dodatkowe
- Język PL/SQL – procedury, funkcje, dynamiczny SQL

Bazy danych Oracle od lat stanowią najlepszą alternatywę dla wszystkich tych, którzy potrzebują funkcjonalnych i pojemnych struktur przechowywania danych, wyposażonych dodatkowo w możliwość wszechstronnego przeszukiwania i zestawiania potrzebnych informacji. Jednak podstawowa wiedza na temat środowiska Oracle nie wystarczy, aby zaprojektować naprawdę przejrzystą, prostą w obsłudze bazę. Do tego potrzebna jest solidna wiedza, którą znajdziesz właśnie w tym podręczniku.

„Programowanie serwera Oracle 11g SQL i PL/SQL" to kontynuacja książki Adama Pelikanta „Bazy danych. Pierwsze starcie", a poruszane w niej zagadnienia są bardziej zaawansowane, choć przy odrobinie samozaparcia także nowicjusz w tej dziedzinie będzie w stanie przyswoić sobie zawartą tu praktyczną wiedzę. Oprócz organizacji serwera, instalacji bazy danych i składni języka SQL szczegółowo omówione są tutaj różne rodzaje zapytań w tym języku (prostych i złożonych), a także funkcje rozszerzenia proceduralnego PL/SQL. W książce opisano także zastosowanie Javy do tworzenia oprogramowania po stronie serwera oraz funkcje analityczne, stanowiące wstęp do przetwarzania OLAP. Całość uzupełniono praktycznymi przykładami, obrazującymi działanie poszczególnych konstrukcji i procedur.

- Organizacja serwera
- Instalacja bazy i końcówki klienta
- Zapytania wybierające, modyfikujące dane i tworzące tabele
- Dodatkowe funkcjonalności SQL
- Procedury składowane i wyzwalane
- Funkcje w PL/SQL
- Pakiety, kursory, transakcje
- Dynamiczny SQL
- Zastosowanie Javy do tworzenia oprogramowania po stronie serwera
- Elementy administracji – zarządzanie uprawnieniami z poziomu SQL
- Obiektowość w Oracle

Wydajna, bezpieczna i prosta w obsłudze – zaprojektuj doskonałą bazę danych!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 7.
Procedury składowane

Zamiast tworzyć bloki anonimowe, wygodniej jest organizować kod w nazwane procedury. W środowisku baz danych noszą one miano procedur składowanych. W przypadku języków wyższego rzędu taka organizacja kodu podyktowana jest chęcią utrzymania jego przejrzystości — jeśli jego fragment ma być wykonywany wielokrotnie, warto go umieścić w procedurze i odwoływać się do niego tylko przez jej wywoływanie. Także różne funkcjonalności, zadania kodu są argumentem za jego podziałem. Podstawowa składnia polecenia tworzącego procedurę ma postać:

```
CREATE PROCEDURE nazwa
IS
BEGIN
-- ciało procedury
END;
```

Zamiast słowa kluczowego `IS` można stosować tożsame słowo kluczowe `AS`. Ciałem procedury może być dowolny zestaw instrukcji PL/SQL oraz dowolne zapytania SQL, za wyjątkiem zapytania wybierającego `SELECT`.

W przypadku środowisk baz danych wydaje się jednak, że argumenty związane z organizacją kodu nie są tu najważniejsze. Dla procedur składowanych najpóźniej przy pierwszym ich wykonaniu generowany jest plan wykonania zapytań wchodzących w ich skład, ich kod jest wstępnie optymalizowany, a następnie są one prekompilowane. W ten sposób na serwerze przechowywana jest procedura w dwóch postaciach: przepisu na jej utworzenie `CREATE PROCEDURE` oraz skompilowanego kodu. Przy każdym następnym wykonaniu odwołujemy się już do postaci skompilowanej. Sprawia to, że procedury są wykonywane szybciej i wydajniej niż ten sam kod w formie skryptu. Sprawa nie jest tak oczywista w przypadku często wykonywanych zapytań. Plany wykonania każdego zapytania przechowywane są w pamięci współdzielonej (*SGA — System Global Area*) i jeżeli wykonamy je ponownie, wykorzystana zostanie jego przetworzona postać. Należy jednak pamiętać, że zgodność dwóch zapytań sprawdzana jest z dokładnością do znaku, nie jest natomiast analizowana semantyka. Poza tym informacje o planach zapytań są nadpisywane na stare definicje w momencie wyczerpania zasobów pamięci współdzielonej. Bez względu na te uwagi możemy jednak przyjąć, że wydajność przetwarzania procedury składowanej jest większa niż w przypadku równoważnego jej skryptu (bloku anonimowego). Jeżeli chcemy usunąć definicję procedury, możemy użyć następującego polecenia:

```
DROP PROCEDURE nazwa;
```

Wielokrotnie możemy chcieć modyfikować kod procedury. Ponowne wykonanie polecenia CREATE PROCEDURE spowoduje wykrycie obiektu o tej samej nazwie i wyświetlenie komunikatu o błędzie. W związku z tym musimy najpierw usunąć definicję procedury, a następnie utworzyć ją ponownie. Zamiast tego możemy posłużyć się składnią CREATE OR REPLACE PROCEDURE, która, w zależności od tego, czy procedura o danej nazwie już istnieje, czy też nie, stworzy ją od podstaw lub nadpisze na istniejącej nową definicję. Z punktu widzenia programisty nie da się rozróżnić, która z tych akcji została wykonana (taki sam komunikat — *Procedura została pomyślnie utworzona*), dlatego, korzystając z tej składni, należy się upewnić, czy nadpisujemy kod właściwej procedury! Przykładem może być utworzenie procedury, której zadaniem będzie zamiana wszystkich nazwisk w tabeli *Osoby* na pisane dużymi literami.

```
CREATE OR REPLACE PROCEDURE up
IS
BEGIN
UPDATE Osoby SET Nazwisko=UPPER(Nazwisko);
END;
```

Jak widać, ciało procedury zawiera zapytanie aktualizujące UPDATE zgodne ze składnią SQL. Jej wywołanie może odbyć się na przykład z wnętrza bloku anonimowego o postaci:

```
BEGIN
up;
END;
```

Załóżmy, że w ramach tego samego skryptu chcielibyśmy sprawdzić poprawność wykonania procedury zapytaniem wybierającym SELECT. Zgodnie z wcześniejszą uwagą, zamieszczenie go w ciele bloku anonimowego jest niedozwolone. Również umieszczenie zapytania wybierającego bezpośrednio po słowie kluczowym END; zakończy się komunikatem o błędzie. Stąd konieczność podzielenia skryptu na dwie części, które z punktu widzenia serwera stanowić będą osobne fragmenty. Znakiem odpowiedzialnym za taki podział jest slash (/), przy czym części będą traktowane jako fragmenty kodu PL/SQL lub zapytania SQL, w zależności od zawartości. Skrypt może zostać podzielony w ten sposób na dowolną liczbę niezależnych fragmentów.

```
BEGIN
up;
END;
/
SELECT Nazwisko FROM Osoby;
```

Innym przykładem realizacji procedury składowanej jest zastosowanie jej do przepisywania nazwisk i wzrostu osób do tabeli *wys_tab*, przy czym rekordy posortowane będą malejąco według wzrostu. Należy zauważyć, że tabela *wys_tab* o odpowiedniej strukturze musi już istnieć w schemacie użytkownika.

```
CREATE OR REPLACE PROCEDURE wysocy
IS
BEGIN
INSERT INTO wys_tab(Nazwisko,Wzrost)
SELECT Nazwisko, Wzrost FROM Osoby
ORDER BY Wzrost DESC;
END wysocy;
```

W przedstawionym przykładzie definicja tworzonej procedury jest kończona poleceniem `END nazwa;`, gdzie `nazwa` jest jej nazwą. Pominięcie jej w tym poleceniu nie pociąga za sobą żadnych zmian. Kończenie definicji procedury w prezentowany sposób jest wskazane ze względów organizacyjnych, porządkowych, zwłaszcza wtedy, kiedy skrypt zawiera większą liczbę złożonych procedur, co utrudnia znalezienie końców definicji przy jego poprawianiu.

Dotychczas prezentowane przykłady były procedurami niepobierającymi żadnych danych z wywołującego je skryptu — były bezparametrowe. Brak parametrów wywołania jest szczególnie widoczny w drugim przypadku, gdzie do tabeli *wys_tab* trafiają wszyscy pracownicy — wszyscy są traktowani jako wysocy. Zmodyfikujmy tę procedurę tak, aby do tabeli docelowej przepisywane były tylko osoby wyższe od pewnego progu danego w postaci parametru.

```
CREATE OR REPLACE PROCEDURE wysocy
(mm number)
IS
BEGIN
INSERT INTO wys_tab(Nazwisko,wzrost)
SELECT Nazwisko, wzrost FROM Osoby
WHERE wzrost > mm
ORDER BY wzrost DESC;
END wysocy;
```

Jak pokazano, parametry procedury definiowane są w nawiasie po jej nazwie, a na minimalną definicję składa się nazwa i typ parametru, przy czym typ podawany jest bez definiowania rozmiaru, czyli `number`, a nie `number(10)`. Podanie rozmiaru w tym miejscu powoduje wyświetlenie komunikatu o błędzie. Zdefiniowanie parametru procedury sprawia, że jest on traktowany tak jak zadeklarowana zmienna i nie może zostać zadeklarowany ponownie w jej ciele. Może on zostać użyty w dowolnym miejscu definicji procedury; w przedstawionym przykładzie został wykorzystany do sformułowania warunku w klauzuli `WHERE` zapytania wstawiającego wiersze.

Jeżeli chcemy użyć w definicji procedury więcej niż jednego parametru, ich lista powinna być rozdzielona przecinkami. W przykładzie przedstawiono procedurę, która wstawia do tabeli *Dodatki* wiersze zawierające sumę wartości brutto i stałej danej drugim parametrem procedury *Dodatek* dla osoby, której identyfikator zawiera pierwszy z parametrów *Num*.

```
CREATE OR REPLACE PROCEDURE Dodaj
(Num number, Dodatek number)
IS
BEGIN
INSERT INTO Dodatki
SELECT Brutto+Dodatek FROM Zarobki
WHERE IdOsoby = Num;
END;
```

Jak przedstawiono to w dotychczasowych przykładach, zawartość procedury mogą stanowić wszystkie zapytania modyfikujące dane, ale również zapytania tworzące lub modyfikujące strukturę bazy. Czasami jednak, zamiast wykonywać jakieś operacje na danych, chcemy dokonać operacji zwracającej wynik w postaci zmiennej, np. policzyć czy

podsumować jakąś wielkość zapisaną w bazie. W takim przypadku w ciele procedury wykorzystujemy bardzo często składnię `SELECT ... INTO zmienna`. Powoduje ona, że wyznaczona zapytaniem wartość skalarna jest przypisywana do zmiennej występującej po słowie kluczowym `INTO`. Zapytanie takie nie skutkuje wyprowadzeniem danych na standardowe wyjście. Nie jest ono elementem SQL, a co za tym idzie, może być używane tylko we wnętrzu procedur i funkcji albo w blokach anonimowych PL/SQL. Jeżeli dokonujemy takiego przypisania do zmiennej, która jest parametrem procedury, to parametr taki musi mieć sufiks `OUT` wskazujący, że jest on przekazywany z procedury do miejsca jej wywołania. Jeśli nie zdefiniujemy parametru jako „wychodzącego", to próba przypisania mu wartości spowoduje pojawienie się komunikatu o błędzie. Wszystkie parametry, które nie mają jawnie określonego kierunku przekazywania danych, są domyślnie typu `IN`, czyli przekazują dane tylko do procedury. Dopuszczalny jest jeszcze trzeci przypadek, w którym zarówno zmienna przekazywana jest do procedury, jak i obliczona wewnątrz procedury wartość przekazywana jest na zewnątrz. W takiej sytuacji zmienna jest opisywana parą `IN OUT`. Prezentowana procedura zlicza osoby wyższe, niż to określa wartość progowa dana pierwszym parametrem.

```
CREATE OR REPLACE PROCEDURE wysocy
(mm number, ile OUT number)
IS
BEGIN
SELECT COUNT(wzrost) INTO ile FROM Osoby
WHERE wzrost > mm;
END wysocy;
```

Wywołanie tak zdefiniowanej procedury z bloku anonimowego wymaga zadeklarowania zmiennej, do której zostanie przypisany parametr typu `OUT`. Musi ona mieć typ zgodny z parametrem formalnym, ale wymagane jest też podanie jego rozmiaru, o ile typ nie oferuje rozmiaru domyślnego. W prezentowanym przykładzie zadeklarowano zmienną `ile` jako `number` (bo domyślnie `number ≡ number(10)`). W przypadku zmiennych znakowych rozmiar pola musi być dany jawnie (`varchar(11)`). Wywołania procedury dokonujemy przez podanie jej nazwy oraz zdefiniowanie wartości parametrów — czyli podanie listy parametrów aktualnych. Do parametrów typu `IN` możemy przypisywać zarówno zmienne, jak i stałe, natomiast do parametrów typu `OUT` musimy przypisać zmienne. W prezentowanym przykładzie nazwa zmiennej oraz parametru w definicji procedury jest taka sama, co jest powszechnie stosowaną notacją, choć ze względów składniowych zgodność nazw nie jest wymagana.

```
SET SERVEROUTPUT ON;
DECLARE
ile number;
BEGIN
wysocy(1.8, ile);
DBMS_OUTPUT.PUT_LINE(ile);
END;
```

Oczywiście procedura może zawierać więcej niż jeden parametr typu `OUT`. Załóżmy, że oprócz liczby osób o wzroście większym od wartości progowej chcemy wyznaczyć ich średni wzrost. Możemy zastosować dwa zapytania agregujące, które określą interesujące nas wartości, jednak bardziej wydajne jest użycie podwójnego przypisania w zapytaniu `SELECT ... INTO`. Jeśli zwraca ono jeden wiersz, to po słowie kluczowym `INTO` musimy umieścić tyle zmiennych, ile wyrażeń jest wyznaczanych w jego pierwszej części.

```
CREATE OR REPLACE PROCEDURE wysocy
(mm number, ile OUT number, sr OUT real)
IS
BEGIN
SELECT COUNT(wzrost), AVG(wzrost) INTO ile, sr FROM Osoby
WHERE wzrost > mm;
END wysocy;
```

Dla każdego parametru może zostać zdefiniowana wartość domyślna. Wykonujemy to przez podanie po typie zmiennej słowa kluczowego `DEFAULT`, po którym ustanawiana jest wartość. W prezentowanym przykładzie przyjęto, że domyślną wartością progu, od którego zliczamy osoby wysokie, jest `1.7`.

```
CREATE OR REPLACE PROCEDURE wysocy
(mm NUMBER DEFAULT 1.7, ile OUT NUMBER)
IS
BEGIN
SELECT COUNT(wzrost) INTO ile FROM Osoby
WHERE wzrost > mm;
END wysocy;
```

Dla procedury ze zdefiniowaną wartością domyślną poprawne jest wywołanie, w którym podajemy wartość posiadającego ją parametru. Wówczas wartość podana przy wywołaniu „nadpisuje" się na domyślną.

```
SET SERVEROUTPUT ON;
DECLARE
ile number;
BEGIN
wysocy(1.8,ile);
DBMS_OUTPUT.PUT_LINE(ile);
END;
```

Jeżeli jednak chcemy przy tak zdefiniowanych parametrach odwołać się do wartości domyślnej, to musimy zastosować wywołanie nazewnicze o postaci `parametr=>zmienna`, gdzie `parametr` jest nazwą parametru formalnego procedury, a `zmienna` jest wartością aktualną tego parametru w miejscu, z którego ją wywołujemy. Notację tę możemy odczytywać jako „**parametr staje się zmienną**". W przypadku takiego wywołania zmienna, która nie została w nim wymieniona, będzie miała przypisaną wartość domyślną. Gdyby w definicji procedury pominięta w wywołaniu zmienna nie miała zdefiniowanej wartości domyślnej, to takie wywołanie spowodowałoby wyświetlenie komunikatu o błędzie.

```
SET SERVEROUTPUT ON;
DECLARE
ile number;
BEGIN
wysocy(ile => ile);
DBMS_OUTPUT.PUT_LINE(ile);
END;
```

W definicji procedury zmieniamy kolejność parametrów tak, że drugi z nich ma przypisaną wartość domyślną. Reszta procedury pozostaje bez zmian.

```
CREATE OR REPLACE PROCEDURE wysocy
(ile OUT NUMBER, mm NUMBER DEFAULT 1.7)
IS
BEGIN
SELECT COUNT(wzrost) INTO ile FROM Osoby
WHERE wzrost > mm;
END wysocy;
```

W takim przypadku, ponieważ przy odwołaniu do wartości domyślnej pomijamy ostatni parametr, dopuszczalne jest zastosowanie wywołania pozycyjnego. Jest ono dozwolone wtedy, gdy pominięciu podlega *n* ostatnich parametrów posiadających wartości domyślne.

```
SET SERVEROUTPUT ON;
DECLARE
ile number;
BEGIN
wysocy(ile);
DBMS_OUTPUT.PUT_LINE(ile);
END;
```

Oczywiście bardziej ogólne jest wywołanie nazewnicze, które, bez względu na to, na których pozycjach znajdują się wartości domyślne, jest zawsze poprawne.

```
SET SERVEROUTPUT ON;
DECLARE
ile number;
BEGIN
wysocy(ile => ile);
DBMS_OUTPUT.PUT_LINE(ile);
END;
```

Wywołanie nazewnicze jest dopuszczalne również wtedy, kiedy podajemy pełny zestaw parametrów. W takim przypadku kolejność ich wymieniania nie odgrywa żadnej roli.

```
SET SERVEROUTPUT ON;
DECLARE
ile number;
BEGIN
wysocy(mm=>1.8, ile=>ile);
DBMS_OUTPUT.PUT_LINE(ile);
END;
```

Do wersji 11. możliwe było stosowanie albo tylko wywołania pozycyjnego, albo nazewniczego — jednoczesne użycie obu tych typów nie było dozwolone. Od wersji 11. istnieje już taka możliwość. Przeanalizujmy to na przykładzie procedury o trzech parametrach numerycznych. Pierwsze dwa są typu `IN`, a trzeci typu `OUT`. W ciele procedury wartość parametru wyjściowego c jest wyznaczana w postaci ilorazu dwóch pierwszych parametrów.

```
CREATE OR REPLACE PROCEDURE dziel
(a real, b real, c OUT real)
AS
BEGIN
c:=a/b;
END;
```

W przykładowym skrypcie pokazane zostały poprawne wywołania tej procedury: w pełni pozycyjne, z dwoma pierwszymi parametrami danymi pozycyjnie oraz z danym pozycyjnie pierwszym parametrem.

```
DECLARE
res real;
BEGIN
dziel(10, 9, res);
DBMS_OUTPUT.PUT_LINE('Wynik ' || res);
dziel(10, 9, c => res);
DBMS_OUTPUT.PUT_LINE('Wynik ' || res);
dziel(10, b => 9, c => res);
DBMS_OUTPUT.PUT_LINE('Wynik ' || res);
dziel(10, c => res, b => 9);
DBMS_OUTPUT.PUT_LINE('Wynik ' || res);
END;
```

Nie każde wywołanie mieszane jest jednak dopuszczalne. Dozwolone są tylko takie przypadki, w których pierwsze na liście parametry podstawiane są pozycyjnie, a pozostałe nazewniczo. Kolejny przykład pokazuje niepoprawne użycie wywołania mieszanego, gdzie błąd polega na tym, że środkowy parametr jest dany nazewniczo, czyli, inaczej mówiąc, że po parametrze danym nazewniczo istnieje choć jeden dany pozycyjnie.

```
DECLARE
res real;
BEGIN
dziel(10, b => 9, res);
DBMS_OUTPUT.PUT_LINE('Wynik ' || res);
END;
```

W przypadku próby wykonania takiego bloku anonimowego wygenerowany zostanie komunikat o błędzie w następującej postaci:

```
Error report:
ORA-06550: linia 12, kolumna 15:
PLS-00312: skojarzenie parametrów przez nazwę może nie implikować skojarzenia przez pozycję
ORA-06550: linia 12, kolumna 1:
PL/SQL: Statement ignored
06550. 00000 - "line %s, column %s:\n%s"
*Cause: Usually a PL/SQL compilation error.
*Action:
```

Ciekawą możliwością zastosowania procedury jest wykonywanie zapytań składanych z fragmentów, np. danych statycznymi napisami czy też zawartych w zmiennej. Zbudujmy procedurę, która w zależności od wartości parametru zmieni sposób zapisu pola *Nazwisko* w tabeli *Osoby*. Wykonanie takiego zapytania, składającego się z fragmentów napisów, wymaga użycia polecenia `EXECUTE IMMEDIATE`.

```
CREATE OR REPLACE PROCEDURE exe_tekst
(typ varchar2)
IS
BEGIN
EXECUTE IMMEDIATE
'UPDATE osoby SET Nazwisko=' || typ || '(Nazwisko)';
END exe_tekst;
```

Bardziej eleganckim rozwiązaniem jest zastosowanie zmiennej pomocniczej zap, co oczywiście nie wpływa na sposób wykonania. Przykład ten pokazuje natomiast, że miejscem deklaracji zmiennych lokalnych procedury (takich, które są widoczne tylko w jej ciele) jest obszar między słowami kluczowymi IS oraz BEGIN. Przy deklarowaniu zmiennych w tym miejscu nie wolno stosować słowa kluczowego DECLARE.

```
CREATE OR REPLACE PROCEDURE exe_tekst
(typ varchar2)
IS
zap varchar2(111);
BEGIN
zap:= 'UPDATE osoby SET Nazwisko=' || typ || '(Nazwisko)';
EXECUTE IMMEDIATE zap
END exe_tekst;
```

Przy okazji prezentowania tego przykładu chcę przedstawić nieco archaiczne wywołanie procedury z zastosowaniem polecenia CALL. Poprawnymi parametrami są tu nazwy funkcji operujących na zmiennych łańcuchowych: UPPER — przepisanie łańcucha do postaci pisanej tylko dużymi literami, LOWER — przepisanie łańcucha do postaci pisanej tylko małymi literami, INITCAP — przepisanie łańcucha do postaci pisanej od dużej litery. Poprawne jest również użycie pustego ciągu znaków '' lub ciągu składającego się z samych spacji ' '. Zaletą tego typu wywołania jest możliwość zastosowania bezpośrednio po nim zapytania wybierającego, które ma za zadanie sprawdzić poprawność wykonania procedury.

```
CALL exe_tekst('UPPER');
SELECT * FROM osoby;
```

W większości sytuacji będziemy jednak stosować klasyczne wywołanie wewnątrz bloku anonimowego. W takim przypadku zastosowanie w nim zapytania wybierającego zgodnie z wymogami składni nie jest dozwolone, jeśli jednak chcemy w ramach tego samego skryptu wykonać blok anonimowy i zapytanie wybierające, musimy rozdzielić je znakiem /. Faktycznie powoduje to, że pomimo tego, iż oba elementy zapisane są w tym samym miejscu, stanowią one dwa przetwarzane po sobie skrypty — jeden PL/SQL, drugi SQL. Każdy dłuższy skrypt może zostać podzielony znakami / na mniejsze fragmenty, które będą przetwarzane szeregowo, oczywiście pod warunkiem, że każdy z nich jest składniowo poprawny.

```
BEGIN
exe_tekst('INITCAP');
END;
/
SELECT * FROM osoby;
```

Kolejny przykład przedstawia zastosowanie w definicji procedury wywołania wbudowanej procedury RAISE_APPLICATION_ERROR, która powoduje wygenerowanie (ustanowienie) błędu użytkownika o numerze danym pierwszym parametrem oraz komunikacie stanowiącym drugi z parametrów. Należy zauważyć, że numeracja błędów w Oracle przebiega przez wartości ujemne, a dla błędów użytkownika zarezerwowano przedział *<–29999, –20001>*. Wywołanie powyższej procedury powoduje przerwanie działania programu i wyświetlenie komunikatu o błędzie.

```
CREATE OR REPLACE PROCEDURE Blad
IS
BEGIN
RAISE_APPLICATION_ERROR (-20205, 'Błąd programu');
END blad;
```

Bardziej złożonym przykładem zastosowania RAISE_APPLICATION_ERROR jest wykorzystanie jej podczas wykonywania procedury o ograniczonym zestawie danych wejściowych, co ma miejsce np. w opracowanej poprzednio procedurze exe_tekst. Jeżeli parametr wejściowy nie jest jednym z dopuszczalnych elementów wymienionych na liście, generowany jest błąd z odpowiednim komentarzem. W przeciwnym przypadku wykonywane jest za pomocą polecenia EXECUTE IMMEDIATE zapytanie. Należy zwrócić uwagę na fakt, że do porównania z elementami listy użyto zmiennej przetworzonej do postaci pisanej dużymi literami UPPER(typ), gdyż przy porównywaniu łańcuchów Oracle rozróżnia ich wielkość. Pozwala to na podanie w wywołaniu procedury nazwy funkcji modyfikującej łańcuch pisanej w dowolny sposób (literami małymi, dużymi oraz różnej wielkości).

```
CREATE OR REPLACE PROCEDURE exe_tekst
(typ varchar2)
IS
BEGIN
IF UPPER(typ) NOT IN('UPPER', 'LOWER', 'INITCAP') THEN
RAISE_APPLICATION_ERROR (-20205, 'Zła funkcja');
ELSE
EXECUTE IMMEDIATE
'UPDATE osoby SET Nazwisko=' || typ || '(Nazwisko)';
END IF;
END exe_tekst;
```

Możemy przekształcić procedurę, tak aby w jej ciele użyć wywołania poprzednio utworzonej procedury generującej błąd przetwarzania o nazwie Blad.

```
CREATE OR REPLACE PROCEDURE exe_tekst
(typ varchar2)
IS
zap varchar2(111);
BEGIN
IF UPPER(typ) NOT IN('UPPER', 'LOWER', 'INITCAP') THEN
Blad;
ELSE
zap:= 'UPDATE osoby SET Nazwisko=' || typ || '(Nazwisko)';
EXECUTE IMMEDIATE zap;
END IF;
END exe_tekst;
```

Błędy mogą się jednak pojawiać podczas przetwarzania nie tylko na skutek celowej działalności programisty, ale mogą być też spowodowane nie zawsze dającymi się przewidzieć zdarzeniami, błędnymi wywołaniami, nieodpowiednimi parametrami etc. Możemy mówić wtedy o sytuacji wyjątkowej — o powstaniu wyjątku. Takie zdarzenia mogą zostać w PL/SQL obsłużone, oprogramowane.

Rozważmy przykład procedury, której zadaniem jest określenie, czy pracownik o danym numerze identyfikacyjnym *IdOsoby*, wskazanym parametrem `num`, istnieje w tabeli *Osoby*. Jeśli tak, drugi z parametrów (`status`) ma przyjąć wartość `1`; w przypadku przeciwnym `0`. W celu realizacji tego zadania zastosowano zapytanie wybierające zwracające do zmiennej pomocniczej `kto` identyfikator osoby. Jeżeli pracownik o danym identyfikatorze istnieje, wartość zwrócona przez zapytanie i wartość parametru będą takie same, jeśli jednak takiego pracownika nie ma, zapytanie wybierające nie zwróci żadnego wiersza. Spowoduje to, że próba podstawienia pod zmienną `kto` zakończy się błędem przetwarzania: *Nie znaleziono żadnych wierszy*. Stan ten możemy wykorzystać, wprowadzając sekcję `EXCEPTION` i oprogramowując wyjątek `NO_DATA_FOUND`, którego obsługa wykonywana jest według schematu `WHEN nazwa_wyjątku THEN instrukcje`. W naszym przypadku obsługa wyjątku zawiera podstawienie odpowiedniej wartości pod zmienną `status` oraz wypisanie komunikatu.

```
CREATE OR REPLACE PROCEDURE czy_jest
(num IN NUMBER, status OUT NUMBER)
IS
kto NUMBER;
BEGIN
SELECT IdOsoby INTO kto
ROM Osoby WHERE IdOsoby = num;
IF (kto = num) THEN
status := 1;
DBMS_OUTPUT.PUT_LINE ('Pracownik istnieje');
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
status := 0;
DBMS_OUTPUT.PUT_LINE('Pracownik nie istnieje');
WHEN OTHERS THEN
NULL;
END;
```

W środowisku Oracle zdefiniowano wiele wyjątków, których nazwy symboliczne i przyczyny wystąpienia zawiera tabela 7.1. Jeżeli gdziekolwiek w ciele procedury występuje sytuacja wyjątkowa, przetwarzanie przenoszone jest do sekcji obsługi wyjątków. Najpierw sprawdzane jest to, czy wyjątek, który przerwał przetwarzanie, jest zgodny z nazwą symboliczną występującą w sekcji `EXCEPTION`. Następnie wykonywane są instrukcje znajdujące się po słowie kluczowym `THEN` jego obsługi, po czym przerywane jest przetwarzanie procedury i następuje powrót do miejsca, z którego została ona wywołana. Jeśli w trakcie przetwarzania procedury pojawią się jednocześnie dokładnie dwa wyjątki, co jest możliwe, chociaż mało prawdopodobne, to obsłużony zostanie tylko ten, który występuje jako pierwszy na liście w sekcji obsługi wyjątków. Na szczególną uwagę zasługuje wyjątek o nazwie `OTHERS`, który obsługuje wszystkie inne, dotąd nieobsłużone. Gdyby znalazł się on na pierwszym miejscu listy obsługi wyjątków, to bez względu na to, jakie zdarzenie powodowałoby wystąpienie sytuacji wyjątkowej, wykonywana byłaby zawsze ta sama sekcja znajdująca się po `OTHERS`. Od wersji 9. Oracle umieszczanie obsługi wyjątku `OTHERS` przed obsługą jakiegokolwiek innego jest zabronione składniowo. Innymi słowy, jego obsługa musi być ostatnim elementem sekcji obsługi wyjątków. W prezentowanym przykładzie zastosowano minimalną obsługę wyjątku (sekcja nie może być pusta) — `NULL`, nie rób nic. Bez względu na to, czy wyjątki są w procedurze obsługiwane, czy też nie, zasady jej wywołania pozostają bez zmian.

Tabela 7.1. *Wykaz najczęściej występujących wyjątków serwera*

Nazwa wyjątku	Numer błędu	Opis
NO_DATA_FOUND	*ORA-01403*	Jednowierszowe zapytanie wybierające SELECT nie zwróciło danych.
TOO_MANY_ROWS	*ORA-01422*	Zapytanie wybierające SELECT zwróciło więcej niż jeden wiersz.
INVALID_CURSOR	*ORA-01001*	Niedopuszczalna operacja na kursorze.
ZERO_DIVIDE	*ORA-01476*	Próba dzielenia przez zero.
DUP_VAL_ON_INDEX	*ORA-00001*	Próba wstawienia powtarzającej się wartości w pole, dla którego ustanowiono indeks unikatowy.
INVALID_NUMBER	*ORA-01722*	Konwersja łańcucha na liczbę zakończyła się niepowodzeniem.
CURSOR_ALREADY_OPEN	*ORA-06511*	Próba otwarcia kursora, który już został otwarty.
LOGIN_DENIED	*ORA-01017*	Próba zalogowania się z nieodpowiednim hasłem lub loginem.
NOT_LOGGED_ON	*ORA-01012*	Próba wykonania polecenia operującego na bazie danych bez uprzedniego zalogowania się.
PROGRAM_ERROR	*ORA-06501*	PL/SQL napotkał wewnętrzny problem podczas przetwarzania.
STORAGE_ERROR	*ORA-06500*	PL/SQL dysponuje zbyt małymi zasobami pamięci lub pamięć została uszkodzona.
TIMEOUT_ON_RESOURCE	*ORA-00051*	Został przekroczony czas oczekiwania na odpowiedź bazy danych.
ACCESS_INTO_NULL	*ORA-06530*	Próba przypisania do zmiennej wartości niezainicjowanej (NULL).
CASE_NOT_FOUND	*ORA-06592*	Żadna z wartości określonych warunkami WHEN w poleceniu CASE nie jest prawdziwa, a nie występuje sekcja ELSE.

Wywołanie pozycyjne procedury może mieć postać:

```
SET SERVEROUTPUT ON;
DECLARE
kto NUMBER;
status NUMBER;
BEGIN
kto:=1;
czy_jest (kto, status);
DBMS_OUTPUT.PUT_LINE(status);
END;
```

Wywołanie nazewnicze można zrealizować według schematu:

```
SET SERVEROUTPUT ON;
DECLARE
kto NUMBER;
status NUMBER;
BEGIN
kto := 1;
czy_jest (status => status, num => kto);
DBMS_OUTPUT.PUT_LINE(status);
END;
```

Przedstawiony poprzednio przykład jest niewątpliwie akademicki, ponieważ wykorzystuje sekcję obsługi wyjątków do wyznaczania parametru OUT. W praktyce, jeśli w tym miejscu jest wyznaczany jakiś parametr wyjściowy, to jest on odpowiedzialny za kodowanie sposobu zakończenia przetwarzania, np. 0 — przetwarzanie zakończone sukcesem, <>0 — przetwarzanie zakończone niepowodzeniem; konkretna wartość koduje przyczynę. Naszą procedurę moglibyśmy więc doprowadzić do postaci, w której wykorzystywalibyśmy funkcję agregującą COUNT.

```
CREATE OR REPLACE PROCEDURE czy_jest
(num IN NUMBER, status OUT NUMBER, ok OUT NUMBER)
IS
BEGIN
ok:=0;
SELECT COUNT(IdOsoby) INTO status
FROM Osoby WHERE IdOsoby = num;
IF (status = 1) THEN
DBMS_OUTPUT.PUT_LINE ('Pracownik istnieje');
ELSE
DBMS_OUTPUT.PUT_LINE('Pracownik nie istnieje');
END IF;
EXCEPTION
WHEN OTHERS THEN
ok := 99;
DBMS_OUTPUT.PUT_LINE('Błąd przetwarzania');
END;
```

Zamiana zapytania wybierającego na takie, które zawiera funkcję zliczającą rekordy, spowoduje, że jeśli pracownik o danym numerze istnieje, to policzony zostanie 1 rekord, a jeśli nie istnieje — 0 rekordów. Jak widać, zmiana taka sprawia, że przypisanie może od razu dotyczyć zmiennej wychodzącej oraz że wyjątek NO_DATA_FOUND nie pojawia się. Zawsze jednak możliwe jest wystąpienie innych błędów przetwarzania, stąd obsługa wyjątku OTHERS, w której ustawiono zmienną ok na wartość różną od zera, w tym przypadku 99, co koduje stan pojawienia się błędu. Zwyczajowo pierwszą linijką ciała procedury jest ustawienie zmiennej kodującej sposób wykonania na 0 — przetwarzanie zakończone poprawnie.

Istnieje formalna możliwość obsłużenia dwóch lub więcej wyjątków w tym samym miejscu sekcji ich obsługi. W tym celu po słowie kluczowym WHEN łączymy nazwy symboliczne wyjątków operatorem logicznym OR.

```
CREATE OR REPLACE PROCEDURE czy_jest
(num IN NUMBER, status OUT NUMBER, ok OUT NUMBER)
IS
BEGIN
ok := 0;
SELECT COUNT(IdOsoby) INTO status
FROM Osoby WHERE IdOsoby = num;
IF (status = 1) THEN
DBMS_OUTPUT.PUT_LINE ('Pracownik istnieje');
ELSE
DBMS_OUTPUT.PUT_LINE('Pracownik nie istnieje');
END IF;
EXCEPTION
WHEN INVALID_NUMBER OR NO_DATA_FOUND THEN
```

```
ok := 11;
DBMS_OUTPUT.PUT_LINE('Błąd wartości');
WHEN OTHERS THEN
ok := 99;
DBMS_OUTPUT.PUT_LINE('Błąd przetwarzania');
END;
```

Nie zawsze jest tak, że wyjątek musi wiązać się z formalnym błędem przetwarzania. Czasami wygodnym jest, aby pewne sytuacje nieprowadzące do błędów formalnych były traktowane jako wyjątkowe. Mówimy wtedy o wyjątkach użytkownika, które muszą zostać zdefiniowane, wykryte i które powinny zostać obsłużone. W prezentowanym przykładzie za sytuację wyjątkową będziemy chcieli uznać fakt, że nie ma osób o wzroście wyższym od progu danego parametrem. Jak widać, taka sytuacja nie spowoduje powstania błędu przetwarzania — trzeba więc ją wykryć.

```
CREATE OR REPLACE PROCEDURE licz
(mini NUMBER, ile out INT)
IS
brakuje EXCEPTION;
BEGIN
SELECT COUNT(IdOsoby) INTO ile FROM Osoby
WHERE Wzrost > mini;
IF (ile = 0) THEN
RAISE brakuje;
END IF;
EXCEPTION
WHEN brakuje THEN
RAISE;
WHEN OTHERS THEN
NULL;
END;
```

Deklaracji wyjątku użytkownika, tak samo jak każdej innej zmiennej, dokonujemy w sekcji deklaracji i nadajemy mu typ `EXCEPTION` (jak widać, to słowo kluczowe pełni podwójną rolę: jest określeniem typu oraz sygnalizuje początek sekcji obsługi wyjątków). W przykładzie zdefiniowano wyjątek o nazwie brakuje. Po zliczeniu osób o wzroście wyższym od wartości progowej instrukcją warunkową sprawdzono, czy ich liczba jest równa `0`, po czym dla takiego przypadku poleceniem `RAISE nazwa_wyjątku` ustawiono (wygenerowano) błąd użytkownika. Obsługi wyjątku użytkownika dokonujemy na takich samych zasadach jak wyjątków wbudowanych (systemowych). W przykładzie zastosowano drugą, po minimalnej obsłudze `NULL`, najczęściej spotykaną metodę — użycie polecenia `RAISE`, które odpowiada za wygenerowanie wyjątku. Spowoduje to propagację wyjątku do miejsca, z którego procedura została wywołana. Można powiedzieć, że nie da się z sekcji obsługi wyjątków przenieść się do tej samej sekcji, więc wyjątek zgłoszony w sekcji obsługi wyjątków musi zostać obsłużony „piętro wyżej" — w procedurze wywołującej.

Bardzo często wykorzystujemy przy obsłudze wyjątków dwie wbudowane funkcje PL/SQL: `SQLCODE` — zwracającą numer wyjątku (błędu), oraz `SQLERRM` — wyświetlającą związany z tym błędem (wyjątkiem) komunikat. Sposób ich zastosowania ilustruje następny przykład.

```
CREATE OR REPLACE PROCEDURE licz
(mini NUMBER, ile out INT)
IS
brakuje EXCEPTION;
BEGIN
SELECT COUNT(IdOsoby) INTO ile FROM Osoby
WHERE Wzrost > mini;
IF (ile = 0) THEN
RAISE brakuje;
END IF;
EXCEPTION
WHEN brakuje THEN
DBMS_OUTPUT.PUT_LINE('Nie ma takich');
DBMS_OUTPUT.PUT_LINE('kod - ' || SQLCODE);
DBMS_OUTPUT.PUT_LINE('opis - ' || SQLERRM);
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
```

Niestety, w tym miejscu czeka na programistę niemiła niespodzianka. Wszystkie wyjątki zdefiniowane przez użytkownika mają taki sam numer (`-1`) oraz komunikat (*Wyjątek użytkownika*). Mogą przez to stać się nierozróżnialne, jeśli w procedurze będzie ich więcej niż jeden. W takim przypadku możemy zainicjować wyjątek użytkownika wyjątkiem systemowym, stosując dyrektywę `PRAGMA EXCEPTION_INIT`. Posiada ona dwa parametry: pierwszym jest nazwa inicjowanego błędu użytkownika, a drugim reprezentujący go numer błędu systemowego. Od tej chwili błąd użytkownika będzie przejmował po błędzie systemowym jego atrybuty: numer i komunikat.

```
CREATE OR REPLACE PROCEDURE licz
(mini NUMBER, ile out INT)
IS
brakuje EXCEPTION;
PRAGMA EXCEPTION_INIT(brakuje,-13467);
BEGIN
SELECT COUNT(IdOsoby) INTO ile FROM Osoby
WHERE Wzrost > mini;
IF (ile = 0) THEN
RAISE brakuje;
END IF;
EXCEPTION
WHEN brakuje THEN
DBMS_OUTPUT.PUT_LINE('Nie ma takich');
DBMS_OUTPUT.PUT_LINE('kod - ' || SQLCODE);
DBMS_OUTPUT.PUT_LINE('opis - ' || SQLERRM);
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
```

Takie rozwiązanie ma tę wadę, że musimy dopasowywać komunikat systemowy do własnych potrzeb, co nie zawsze jest łatwe. Pozostaje więc zrezygnować z definiowania błędu użytkownika na rzecz stosowania bezpośrednio polecenia `RAISE_APPLICATION_ERROR` lub opisanej poprzednio procedury `Blad`, która to polecenie zawiera — wtedy obsługa nastąpi w sekcji `OTHERS`. Możliwe jest również korzystanie przy obsłudze błędu użytkownika z polecenia `RAISE_APPLICATION_ERROR` w sekcji obsługi wyjątków.

```
CREATE OR REPLACE PROCEDURE licz
(mini NUMBER, ile out INT)
IS
brakuje EXCEPTION;
BEGIN
SELECT COUNT(IdOsoby) INTO ile FROM Osoby
WHERE Wzrost > mini;
IF (ile = 0) THEN
RAISE brakuje;
END IF;
EXCEPTION
WHEN brakuje THEN
RAISE_APPLICATION_ERROR (-20001, 'Nie ma takich');
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
```

Wymuszenie wystąpienia błędu *ad hoc* w sekcji obsługi wyjątków spowoduje jednak jego propagację do miejsca wywołania i wtedy powinniśmy umieścić, np. w bloku anonimowym, sekcję obsługi wyjątków obsługującą taki przypadek — choćby na poziomie zdarzenia `OTHERS`.

```
SET SERVEROUTPUT ON;
DECLARE ile NUMBER;
BEGIN
licz(1.8, ile);
DBMS_OUTPUT.PUT_LINE (ile);
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
```

Z czasem, kiedy będziemy tworzyli coraz bardziej skomplikowane elementy proceduralne PL/SQL (procedury, funkcje), coraz większe będzie ryzyko, że podczas ich generowania popełnimy błędy formalne, składniowe. Standardowo, jeśli w procedurze znajdą się takie błędy, otrzymamy komunikat: *Procedura utworzona z błędami kompilacji*. Jeśli chcemy otrzymać bardziej złożoną informację o popełnionych podczas tworzenia ostatniego elementu proceduralnego błędach, możemy wykonać polecenie:

```
SHOW ERRORS;
```

Pomimo że wygenerowana w ten sposób informacja jest zdecydowanie bardziej szczegółowa, należy z dużym dystansem podchodzić do wskazywanych linii kodu, w których wykryto nieprawidłowości. Bardzo często wskazanie to wynika z wcześniej popełnionych błędów. Prostym wnioskiem jest ten, że procedurę powinniśmy poprawiać, począwszy od błędów najwcześniej wykrytych, co w większości przypadków daje poprawne rezultaty.