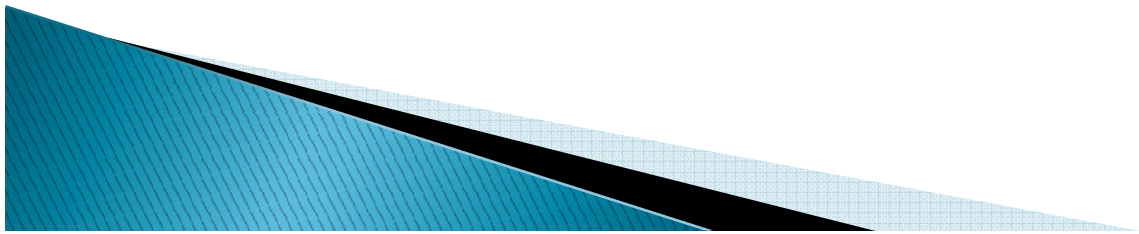
# Poznámky k úvodní části PL/SQL PACKAGE

- 2.Část PL/SQL

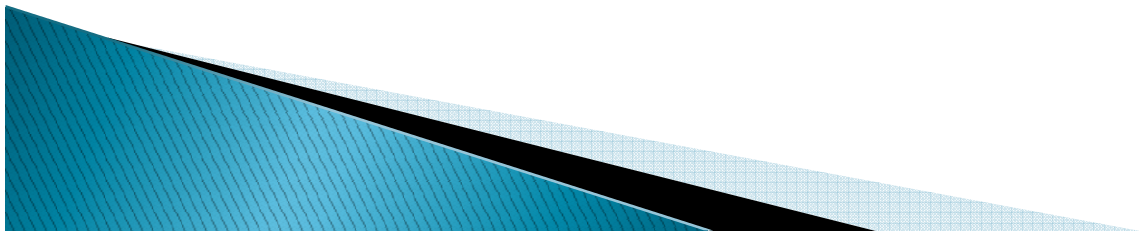
# Deklarace proměnných v PL/SQL

- jméno [CONSTANT] typ [:= výraz];
- typy
  - Standardní
  - BOOLEAN obsahující TRUE, FALSE, NULL
  - Kurzory
  - RECORD
  - TABLE



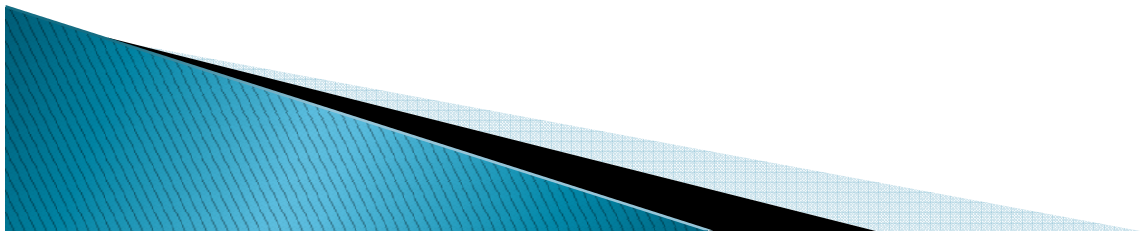
# Nativní datové typy Oracle

- NUMBER [(velikost[, přesnost])]
  - Číselný typ
    - velikost 1 .. 38 počet pamatovaných platných cifer
    - přesnost -84 .. 127 zaokrouhlení,
      pro záporné se zaokrouhluje na desítky, stovky, ...
  - NUMBER ~ reálné číslo
    NUMBER(10)  přirozené číslo na 10 cifer
    NUMBER(5,2)  -999.99 .. +999.99
    NUMBER(5,-2) -9999900 .. +9999900



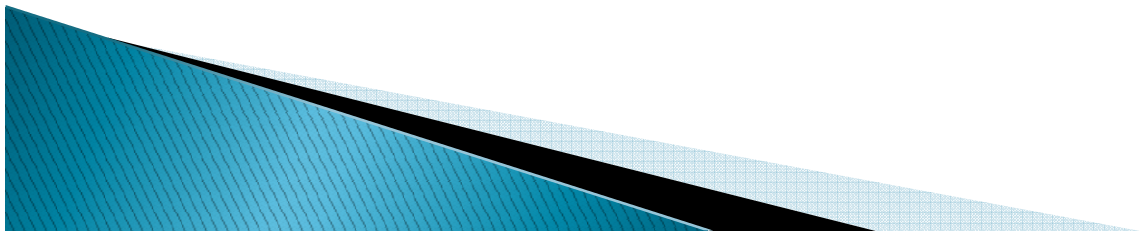
# Nativní datové typy Oracle

- VARCHAR2(velikost),
  VARCHAR(velikost)
  - Řetězec s proměnlivou délkou
    - velikost max. 4000 znaků (dop. max. 2000 znaků)
  - Oracle doporučuje používat VARCHAR2
  - Konstanty se uzavírají do apostrofů,
    Apostrof se v konstantách zdvojuje



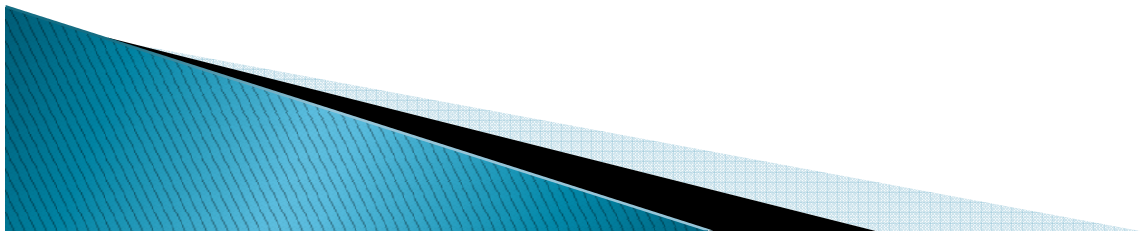
# Nativní datové typy Oracle

- CHAR[(velikost)]
  - Řetězec s pevnou délkou
    - velikost max. 2000 znaků, default 1 znak
  - Konstanty se uzavírají do apostrofů, Apostrof se v konstantách zdvojuje



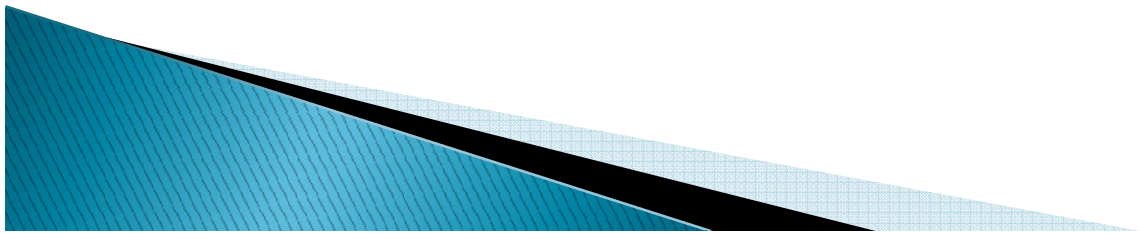
# Nativní datové typy Oracle

- DATE
  - Datum + čas s přesností na 1 sec.
  - Konstanty se uzavírají do apostrofů,
    Zápis by měl odpovídat nastavení jazyka na klientském počítači.
    Standardní americký formát je ’10-JAN-2005’



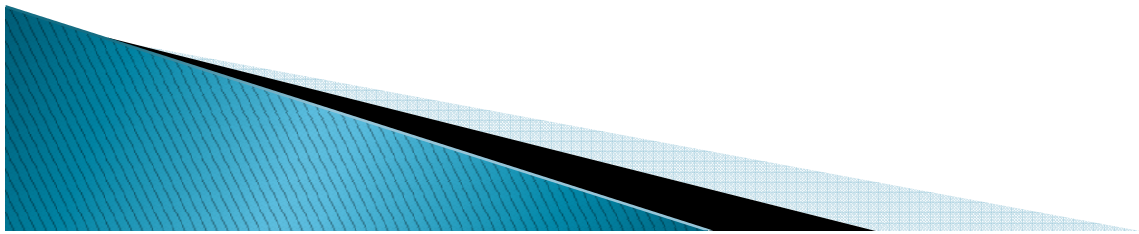
# Nativní datové typy Oracle

- RAW(velikost)
  - Obdoba CHAR pro binární data
  - Konstanty se zapisují do apostrofů v hexadecimální podobě
  - Pro konverze se používají funkce RAWTOHEX a HEXTORAW
    např. HEXTORAW(’7D7E’)



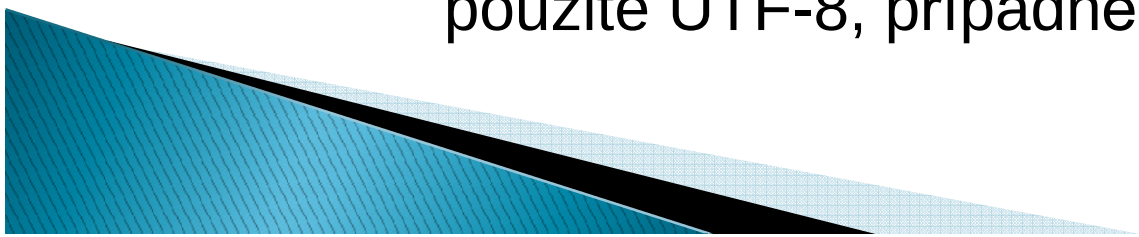
# Nativní datové typy Oracle

- LONG, LONG RAW
  - Až 2GB dlouhé řetězce, nebo binární data
  - Mají velmi mnoho omezení
    - Jen jediný sloupec tohoto typu v tabulce
    - Téměř žádné povolené operace s nimi
  - Oracle je již nedoporučuje používat
  - Nahrazeny LOB typy (Large OBject)



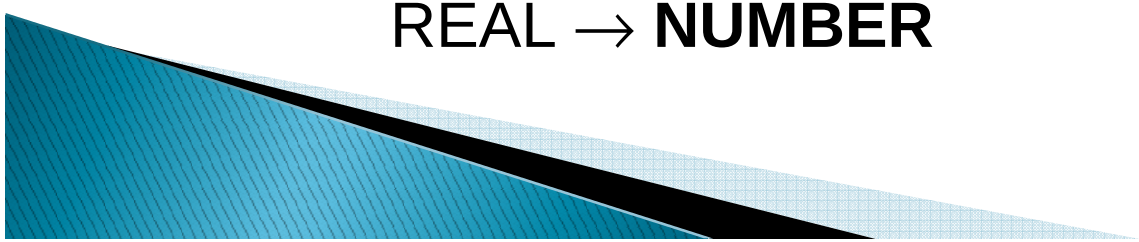
# Nativní datové typy Oracle

- NCHAR, NVARCHAR2
  - Ke každému znakovému typu existuje typ začínající znakem N (National)
  - Liší se kódováním, použitým na straně serveru
  - Standardní typy používají hlavní znakovou sadu databáze
  - Národní typy používají národní znakovou sadu, která může (ale nemusí) být menší, než hlavní.
  - Význam má, pokud je jako hlavní znaková sada použité UTF-8, případně UTF-16



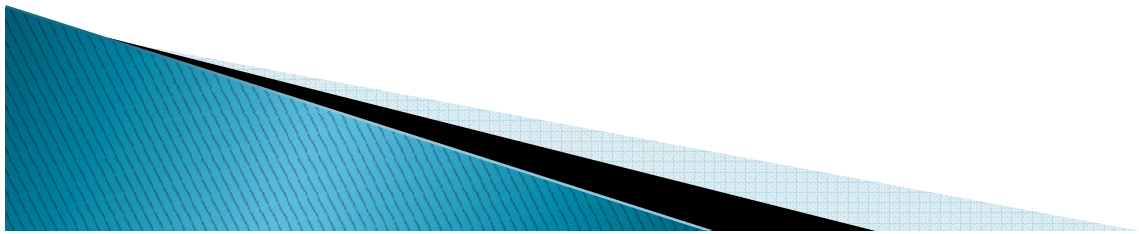
# ANSI SQL datové typy

- CHARACTER(n) → **CHAR**(n)
  CHARACTER VARYING(n)
  CHAR VARYING(n) → **VARCHAR**(n)
  NATIONAL CHARACTER(n) → **NCHAR**(n)
  NATIONAL CHARACTER VARYING(n)
  NATIONAL CHAR VARYING(n)
  NCHAR VARYING(n) → N**VARCHAR**(n)
  NUMERIC(p,s) → NUMBER(p,s)
  INTEGER, INT, SMALLINT → NUMBER(38)
  FLOAT(b)
  DOUBLE PRECISION
  REAL → **NUMBER**



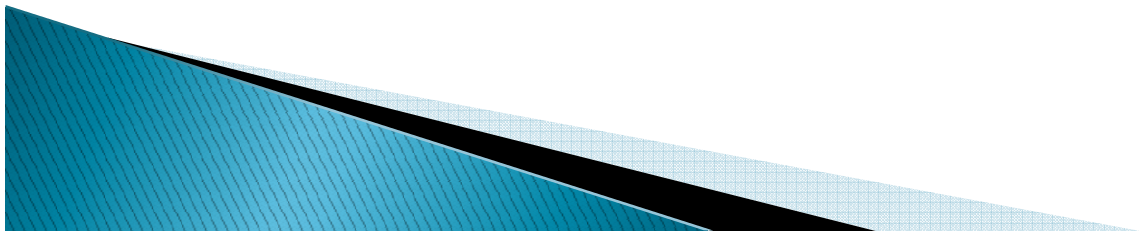
# Upozornění – opakovaně!

- Hlídat na úrovni databáze všechny manipulace s daty, které ohlídat jdou
  - Cokoli jde zadat uživatelem špatně, bude zadáno špatně
    - Integritní omezení, triggery
  - Čištění nekonzistentních dat je namáhavé a opravy jsou často nemožné
  - Lépe ohlídat vše centrálně, než v každé aplikaci zvlášť



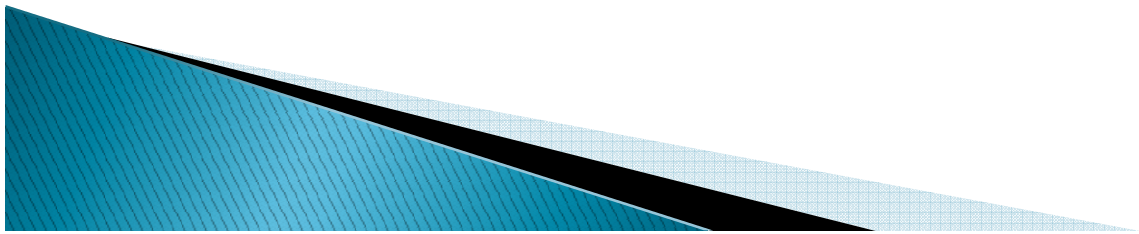
# Deklarace typu v PL/SQL

- Explicitně jménem
  - X NUMBER(7,2);
- Kopírováním typu jiné proměnné
  - Y X**%Type**;
- Kopírováním typu sloupce
  - E EMP.Ename**%Type**;
- Kopírováním typu řádku
  - R EMP**%RowType**;
    - Record, obsahující pole odpovídající sloupcům
    - Přístup pomocí obvyklé tečkové notace



# Příkazy PL/SQL

- Prázdný příkaz
  NULL;
- Přiřazovací příkaz
  proměnná := výraz;
- SQL příkaz
  UPDATE Emp SET Sal = Sal*1.05;
  DELETE FROM Emp
  WHERE EmpNo=1;

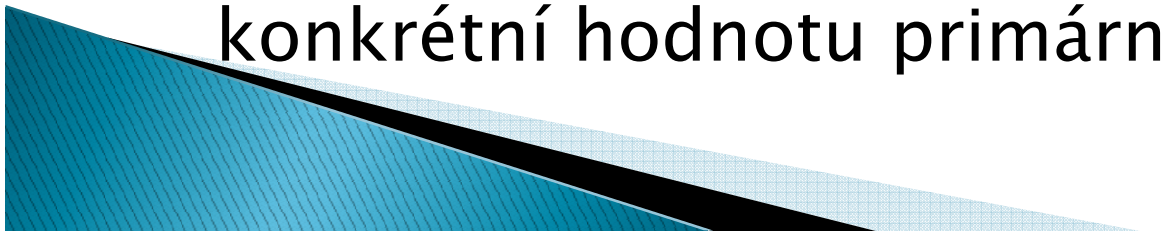


# Práce s daty v tabulkách pomocí PL/SQL

- PL/SQL – integrovaný v rámci databázové platformy

```
SELECT {*} [seznam_polozek]
INTO
  [seznam_polozek_nebo_promenna_typu_zaznam]
FROM nazev_tabulky
WHERE podmínka_vyberu
```

- Podmínkou úspěšnosti je vždy vrácení jednoho řádku – záleží na WHERE – ve většině případů se podmínka v klauzuli vztahuje na konkrétní hodnotu primárního klíče



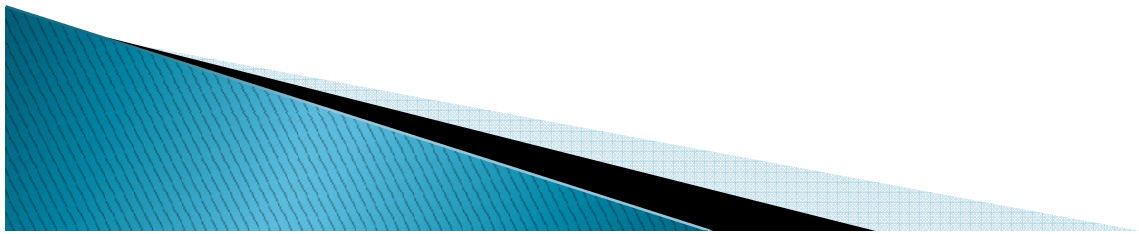
# Práce s daty v tabulkách pomocí PL/SQL – příklad

```
SET SERVEROUT ON
DECLARE
  v_firma zakaznici.firma%TYPE;
  v_dluh zakaznici.dluh%TYPE;
BEGIN
SELECT firma, dluh INTO v_firma, v_dluh FROM
   zakaznici WHERE id_zak = 5;
  DBMS_OUTPUT.PUT_LINE('v_firma = '|| v_firma);
  DBMS_OUTPUT.PUT_LINE('v_dluh = '|| v_dluh);
END;
```

# Píšeme čitelně aneb komentáře

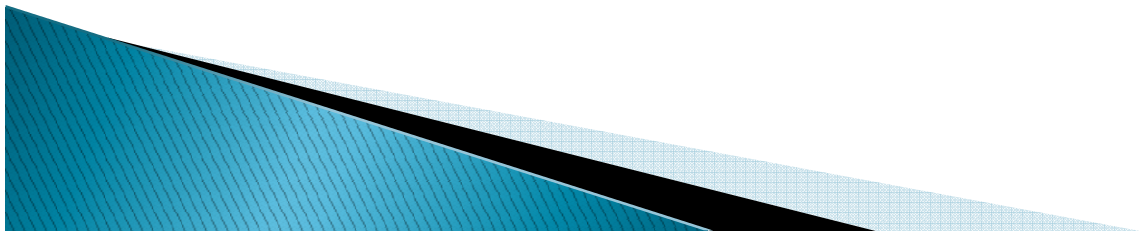
- Každý programovací jazyk podporuje komentáře, které vám umožní psát přehledně a strukturovaně
(odkazuji na literární programování)

- Tyto komentáře podporuje mimo PL/SQL také standardní Oracle SQL
- Jednořádkový komentář `-- komentář`
- Víceřádkový komentář `/* komentář */`



# Operátory

| Operátor | Operace |
|---|---|
| ** | umocňování |
| +,- | unární plus, mínus |
| *,/ | násobení, dělení |
| +,-,\|\| | sčítání, odečítání, slučování |
| =,>,<, <=,>=,<>,!=,~=,^= IS NULL, LIKE, BETWEEN, IN | porovnávání |
| NOT | logická negace |
| AND | logický součin |
| OR | logický součet |



# Proměnné

- Musí se před prvním použitím deklarovat
- Podporované datové typy:
  - libovolný SQL typ mimo LOB identifikátoru (`BLOB`, `CLOB`, `BFILE`)
  - speciální PL/SQL typ (`BINARY_INTEGER`, `PLS_INTEGER`, `BOOLEAN`) nebo jejich podtyp
  - složené datové typy (`RECORD`, `VARRAY`, `TABLE`)
- `PLS_INTEGER` umožňuje přímou práci s ALU, výhody: velká rychlost, range check

# Deklarace proměnné

- Příklad deklarace proměnné:

```
DECLARE
   cislo NUMBER(4);
   i PLS_INTEGER;
   ks POSITIVE NOT NULL := 3;
   uspech BOOLEAN DEFAULT TRUE;
   text VARCHAR2(3000);
BEGIN
   -- programový blok
END;
```

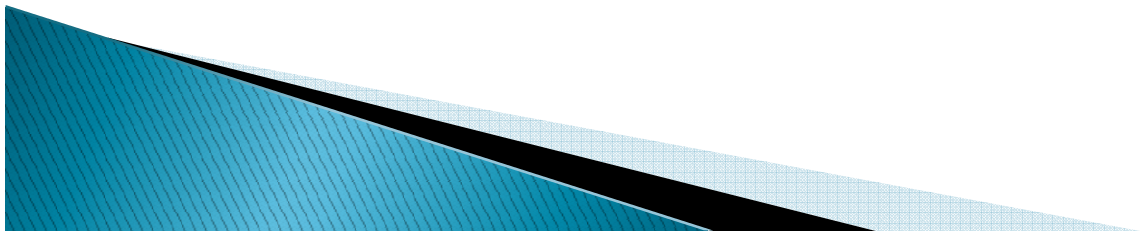
# Přiřazení hodnoty proměnné

- Existují tři možnosti přirazení hodnoty deklarované proměnné:
  - příkazem pro přiřazení hodnoty ( := )
  - výběrem hodnoty z dotazu, jehož výsledkem je jeden řádek (konkrétní hodnota):

```
SELECT AVG(sal) INTO my_sal
FROM emp;
```

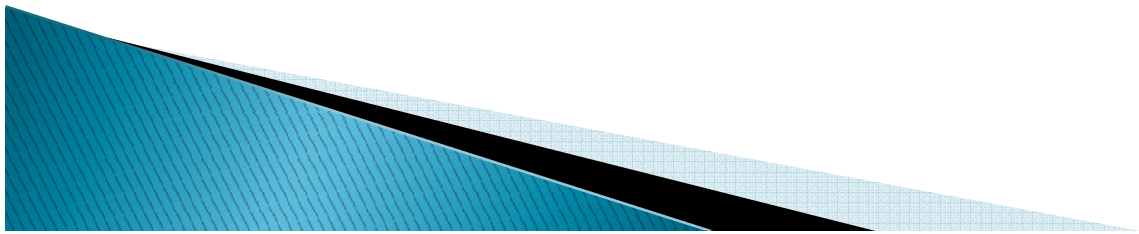
  - výstupního hodnota procedury (tzv. OUT nebo IN OUT parametr procedury):

```
adjust_salary(7788, my_sal);
```



# Užití proměnné

- Proměnné je možné dále užívat jak v PL/SQL, tak v SQL kódu (příkazy SQL mohou být volně užívány v PL/SQL bloku)
- Typické užití proměnné:
  - výpočet výrazu
  - parametry v SQL (přímým zadáním do SQL)
  - předání parametru to procedury či funkce
  - podmínky a cykly – výraz, iterátor
- Proměnné jsou deklarovány (platné) jen pro následující blok (viditelnost, tzv. scope)



# Konstanty

- Představují read-only proměnné
- Musí být inicializovány v okamžiku deklarace ( `DEFAULT`, `:=` ) stejně jako `NOT NULL` proměnné
- Klíčové slovo `CONSTANT`
- Př.

```
DECLARE pi CONSTANT REAL DEFAULT
3.14159265358979323846264383279502;
```

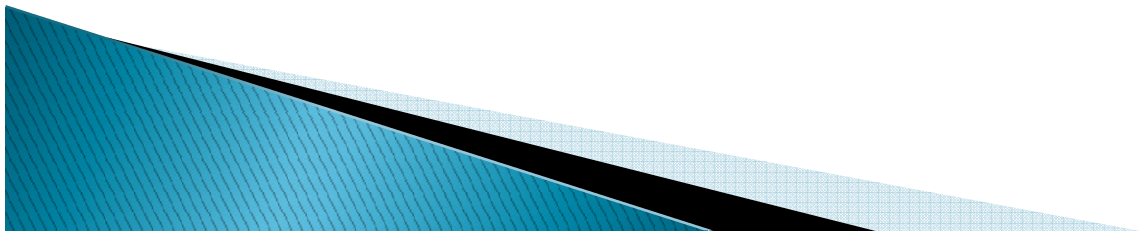
- Tzv. prázdné řetězce jsou vlastně `NULL` (zero-length strings are `NULL`s)
- Databázové položky mají před proměnnými přednost (PL/SQL vychází z SQL)

# Způsob vyhodnocování

- Oracle PL/SQL používá pro vyhodnocení výrazů na místě podmínek

  zkrácenou Boolskou logiku

  tzn. vyhodnocuje zleva doprava a při prvním pozitivní úspěchu OR podmínky
  či prvním negativním úspěchu
  AND podmínky končí
  s vyhodnocováním příslušné části výrazu



# Cyklus s podmínkou na konci

- Kombinace `IF` a `EXIT` umožňuje přerušit cyklus kdykoliv v jeho průběhu
- Speciální případ je rozhodování s `EXIT` jako poslední příkaz cyklu – tzv. cyklus s podmínkou na konci (typický `repeat`)
- Místo zdlouhavého zápisu `IF – EXIT` lze užívat zápisu `EXIT – WHEN`

```
LOOP
   EXIT WHEN podmínka;
END LOOP;
```

# Pojmenovávání cyklů

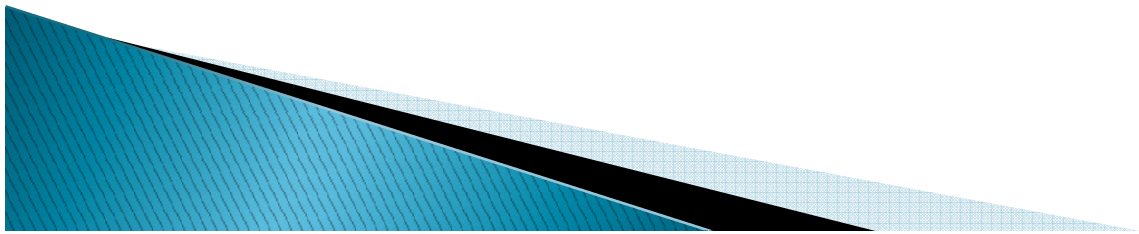
- Jednotlivé cykly (LOOP) či vícehodnotové rozhodování (CASE) lze pojmenovat pomocí návěští, ukončit pak lze lib. cyklus

```
<<outer>>
LOOP
    …
    LOOP
        …
        EXIT outer WHEN podmínka;
    END LOOP;
END LOOP outer;
```

# Sekvenční cyklus

- Tzv. cyklus s pevným počtem opakování
- Iterátor (čítač) je implicitně deklarován jako typ `INTEGER` a je viditelný (scope) jen uvnitř cyklu
- Klasický `for` cyklus s hranicemi `<l, h>`

```
FOR counter IN [ REVERSE ] l..h LOOP
   -- posloupnost příkazů
END LOOP;
```



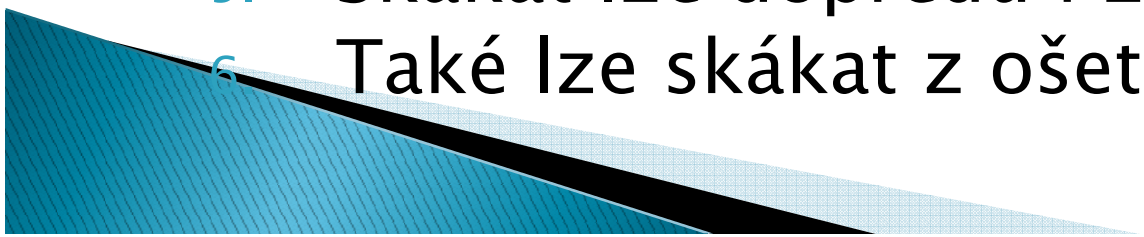
# Posloupnost příkazů

- Není možné ukončit EXIT (slouží jen pro cykly), je nutné užít příkaz RETURN
- Pro skok na návěští lze užít příkazu GOTO
- Návěští definujeme pomocí <<*label*>>

```
BEGIN
   …
   GOTO insert_row;
   …
   <<insert_row>>
   INSERT INTO emp VALUES …
END;
```

# Omezení skoků

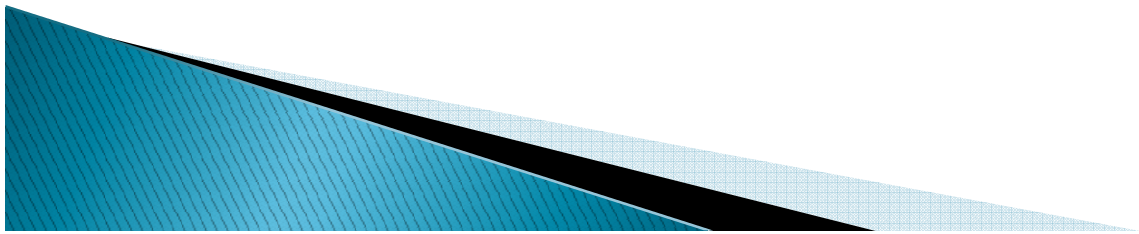
1. Nelze provést skok do větvení–rozhodování, těla všech druhů iterací či podbloku
2. Také nelze skákat mezi různými bloky stejné úrovně (např. THEN a ELSE blok)
3. Rovněž nelze skákat z podprogramu do volajícího programu
4. Naopak lze provést skok z uvedeného do vyšších programových bloků
5. Skákat lze dopředu i zpět
6. Také lze skákat z ošetření výjimek zpět



# Zvláštní příkaz NULL

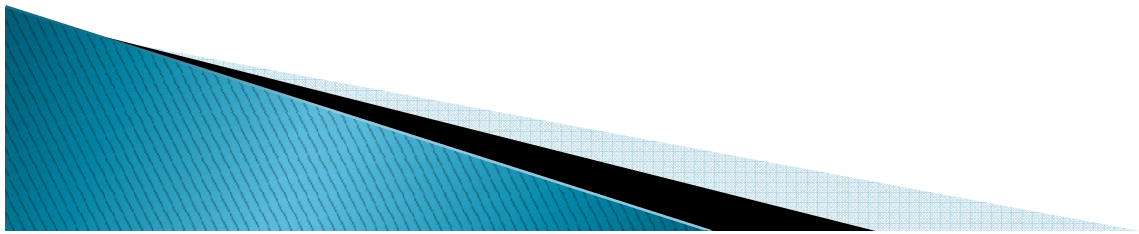
- Příkaz NULL představuje tělo podprogramu nevykonávajícího žádnou činnost
- Je užitečný tam, kde syntaxe nutí zadat příkaz či příkazy, ale my potřebujeme ponechat tuto část prázdnou
- Také je vhodný pro navrhování programu metodou shora-dolů

```
NULL;
```



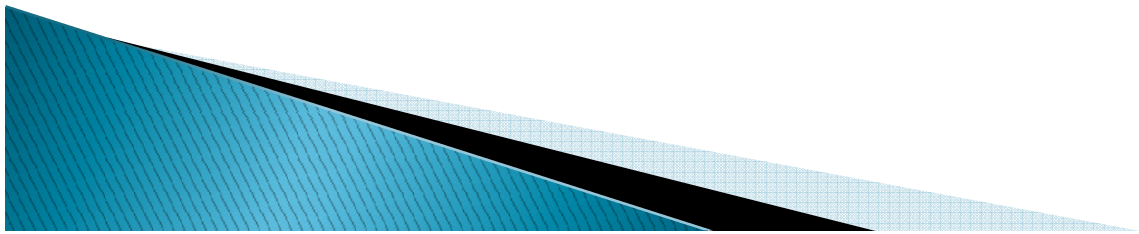
# Parametrické kurzory

- Některé kurzory mohou být zavislé na parametru (tzv. parametrické kurzory)
- Deklarují se s uvedením parametrů a jejich typů – tyto parametry lze potom užít v SQL příkazu
- Při otevírání kurzoru (nebo v části IN cyklu FOR) potom specifikujeme konkrétní hodnoty pro parametry
- Parametry lze užít i v řetězcích typu LIKE



# Příklad parametrického kurzoru

```
DECLARE
   CURSOR c1(min IN NUMBER) IS
        SELECT emp_id, name
        FROM emp
        WHERE age > min
        ORDER BY age DESC;
BEGIN
   OPEN c1(18);
```



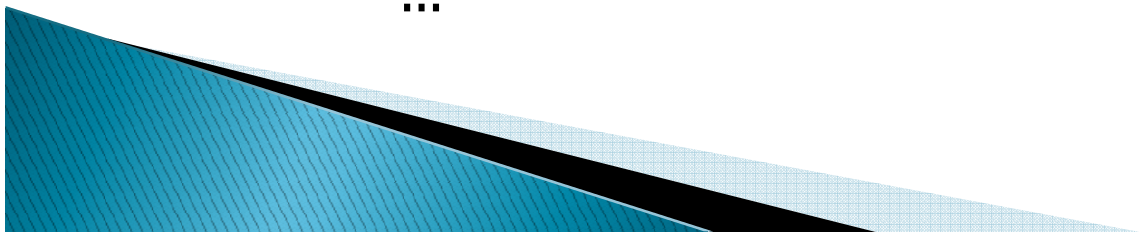
# Omezení kurzorů

1. Kurzory nemohou zajistit variantní tvorbu SQL dotazu (např. přidání či vypuštění podmínky selekce či stanovení řazení)
2. Jsou určeny pro statické SQL dotazy
3. Existuje PL/SQL podpora také pro dynamické SQL dotazy, které jsou tvořeny „za běhu“
4. Některé dynamické SQL dotazy (tzv. nativní) pracují podobně jako kurzory
5. Musí být ale jednotného typu projekce

# Užití nativního dynSQL

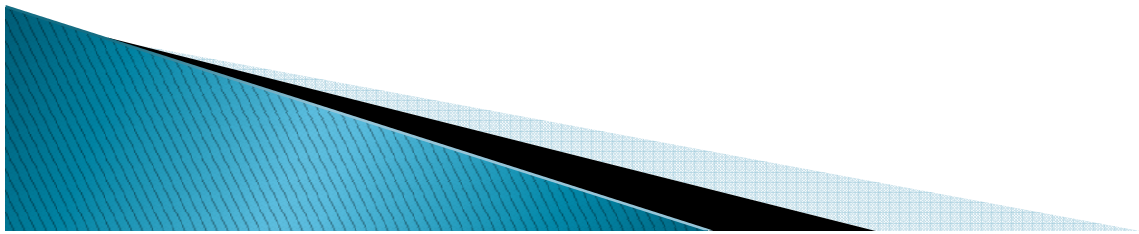
- Definice kurzoru jako tzv. odkazu
- Zadefinování dotazu klauzulí FOR

```
DECLARE
   TYPE typ_c1 IS REF CURSOR;
   c1 typ_c1;
   record emp%ROWTYPE;
BEGIN
   OPEN c1 FOR
        ‘SELECT id FROM emp’;
   …
```



# Chybová hlášení Oracle

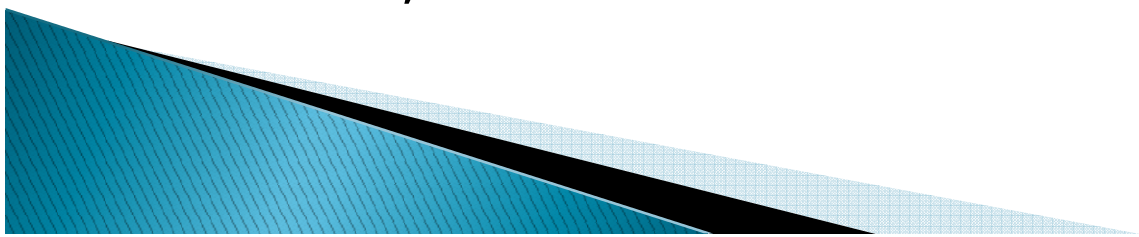
- Oracle generuje chybová hlášení v rozsahu 0 až −19999
- Uživatel si může dále dodefinovat chybová hlášení v rozsahu −20001 až −20999
- Ke každé chybě lze stanovit text hlášení s délkou cca 2000 znaků
- Chyba 0 znamená úspěch
- Detekci chyb lze řešit v části `EXCEPTION`



# Řízení výjimek

- Specifikováním akronymu výjimky (obecné pojmenování) v části EXCEPTION zabráníme implicitní akci
  (konec PL/SQL kódu a oznámení chyby)
  a můžeme provést vlastní akci

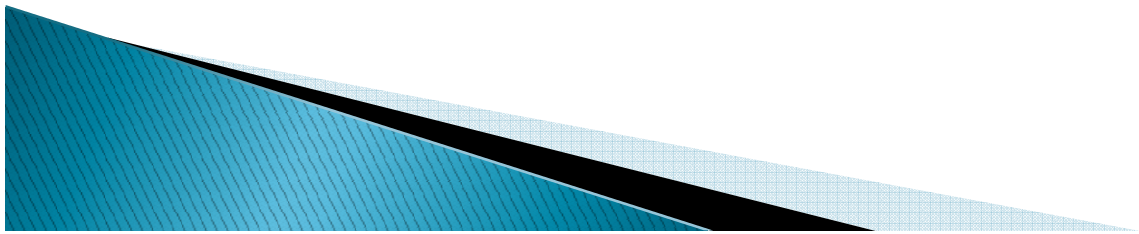
```
EXCEPTION ZERO_DIVIDE THEN
   -- ošetření příslušné výjimky
END;
```



# Vícenásobná detekce výjimek

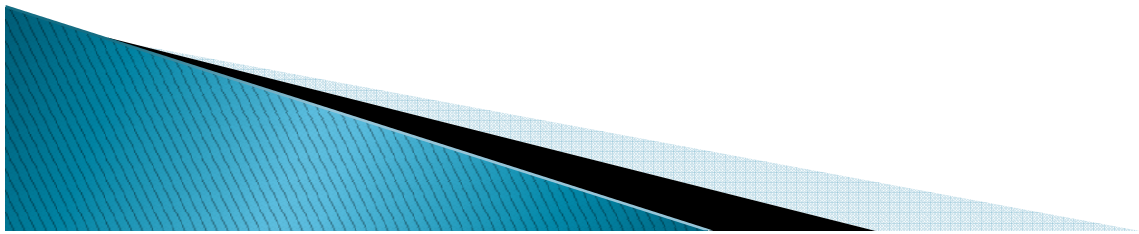
- Je nutné v části EXCEPTION použít výběr pomocí klauzule WHEN

```
EXCEPTION
   WHEN ZERO_DIVIDE THEN
        -- ošetření
   WHEN OTHERS THEN
        -- ošetření
END;
```



# Založení vlastní chyby

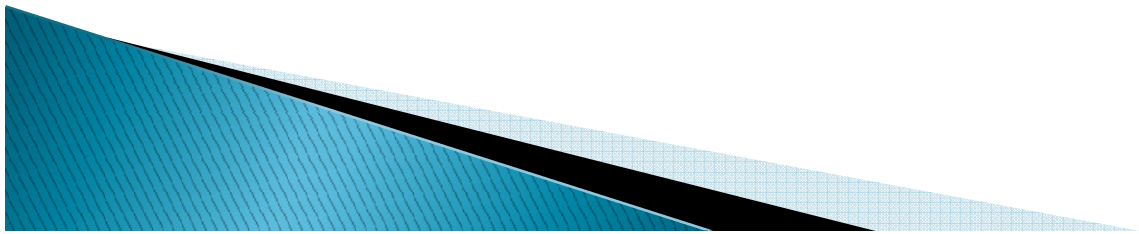
- Existuje procedura `RAISE_APPLICATION_ERROR` s parametry kód chyby a text chybového hlášení
- Často se vyvolává v části pro řešení výjimek (po slově `EXCEPTION` v příslušném `WHEN`) spolu s příkazy pro potvrzení nebo rušení transakcí (informace do budoucnosti)



# Vyvolání výjimky

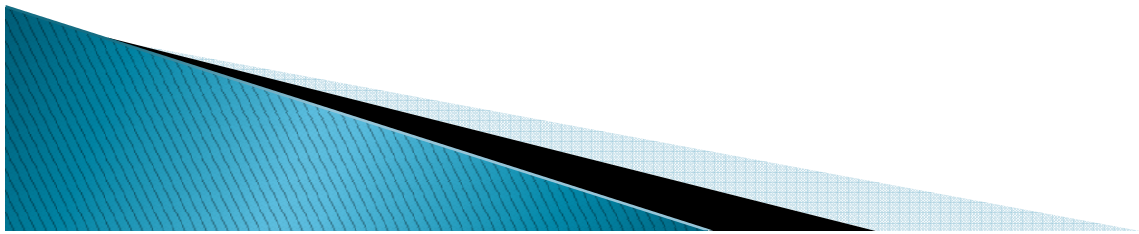
- Zadefinováním nové výjimky v části DECLARE jako typ EXCEPTION vytvoříme další akronym pro vlastní výjimky (ošetřitelné v části EXCEPTION)
- Výjimky (standardní i nově definované) lze vyvolat klauzulí

```
RAISE výjimka;
```



# Programové balíky

- Programový balík je sdružením řady funkcí a procedur s vlastním jmenným prostorem a vlastním persistentním prostorem pro proměnné v rámci jedné session
- To umožňuje uchovávat hodnoty v rámci session pro řadu procedur a funkcí
- Ne náhodná analogie s objekty

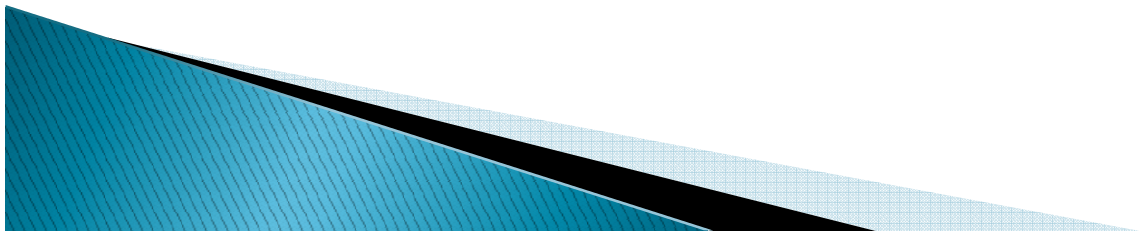


# Tvorba balíků

- Dvě části – veřejné deklarace a tělo programového balíku
- CREATE PACKAGE *balík* AS *kód*;
- CREATE PACKAGE BODY *balík* AS *kód*;
- Uvnitř deklarací pouze plné hlavičky funkcí a procedur určených pro volání mimo balík
- V těle ostatní deklarace proměnných, definice příslušných veřejných objektů a také další privátní podprogramy
- Zpřístupnění pomocí tečkové notace

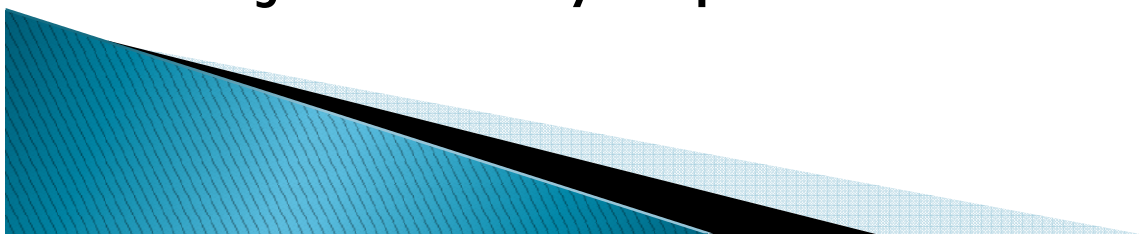
# Externí procedury funkce

- Je možné definovat PL/SQL procedury a funkce, které jsou ve skutečnosti pouze voláními externích funkcí v jiných programovacích jazycích
- Místo PL/SQL kódu se za AS uvede
  - `LANGUAGE JAVA NAME 'třída';`
  - `LANGUAGE C NAME 'jméno'`
    `LIBRARY 'sdílená knihovna';`



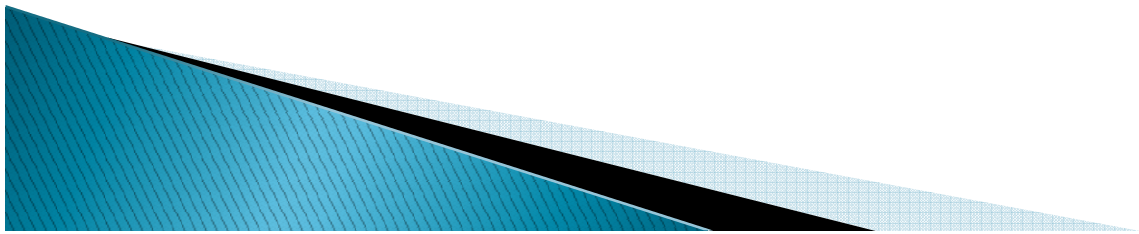
# Pokročilé datové struktury PL/SQL

- **Kolekce** – seřazené homogenní skupiny prvků (odpovídá polím či seznamům), všechny **prvky kolekce** jsou jednoznačně identifikovány zejména **pořadím**, doména je přiřazena celé kolekci
- **Záznamy** – způsob uložení heterogenních dat stejné logické příslušnosti, jsou členěny na jednotlivá **pole**, které jsou identifikovány pojmenováním – doména je přiřazena jednotlivým polím



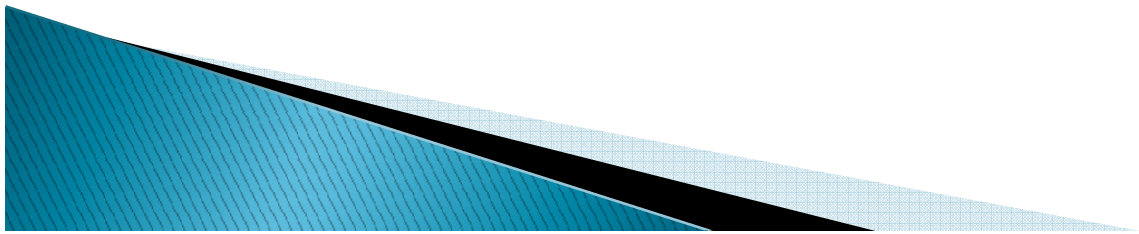
# Kolekce

- Tři základní druhy kolekcí:
  - **index-by table** (asociativní pole) – identifikace prvků pomocí řetězce nebo indexu (čísla)
  - **nested table** (hnízděná tabulka) – odpovídá virtuální databázové tabulce, práce s ní je sekvenční



# Kolekce

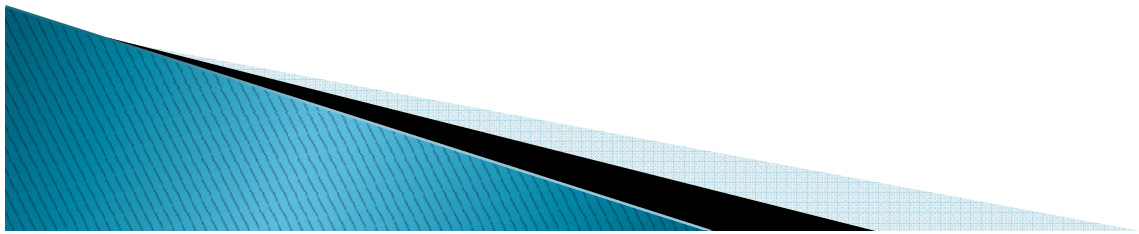
- **variable-size array** (dynamické pole) – tzv. varrays odpovídají klasickým dynamickým polím, uchovávají definovaný počet hodnot, pomalejší přístup SQL nástroji než k nested tables



# Asociativní pole

- Struktura známá z programovacích jazyků
- Různé typy indexů – integer nebo řetězce
- DECLARE

```
   TYPE tsal IS TABLE OF NUMBER
        INDEX BY VARCHAR2(64);
   sal tsal;
BEGIN
   sal('Jan Novák') := 12000;
END;
```

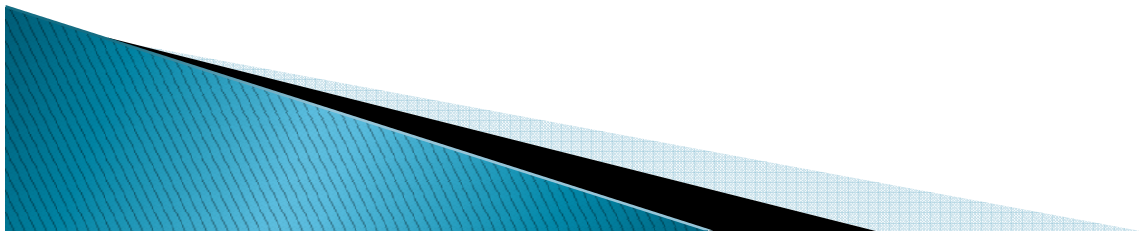


# Nested tables

- Představují pole – množina (set, bag) v některých programovacích jazycích
- Lze ukládat tyto tabulky v databázových tabulkách (oboustranná kompatibilita)
- Deklarace:

```
DECLARE TYPE ntable IS TABLE
    OF element_type;
```

- Typ prvku může být lib. PL/SQL typ mimo odkazu (REF) a kurzoru (CURSOR)

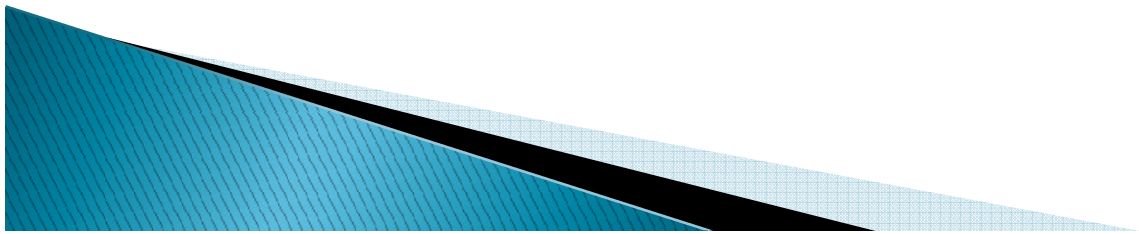


# Dynamické pole

- Představuje klasické pole jiných jazyků – obsahuje nejvýše definovaný počet prvků homogenního typu
- Typ prvku odpovídá *element_type* v nested tables
- Deklarace:

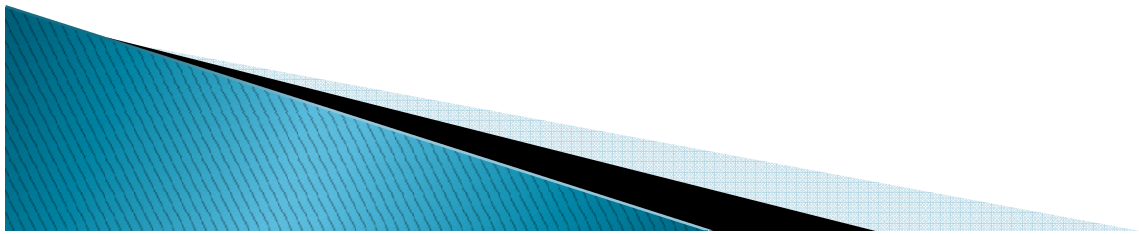
```
DECLARE TYPE calendar IS
   VARRAY(366) OF DATE;
```

- Přístup: *collection*(*item_index*);



# Přístup k prvkům kolekcí

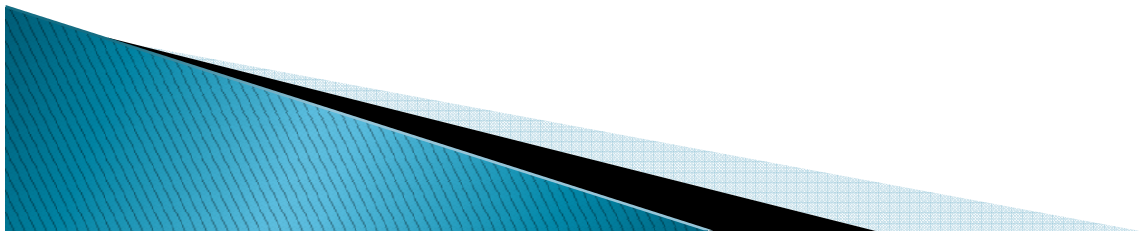
- Jsou definovány metody pro přístup k prvkům kolekcí – `FIRST`, `LAST`, `PRIOR`, `NEXT`, `COUNT`, `EXISTS`, `EXTEND`, `TRIM`, `DELETE`
- Tyto metody se uvádějí jako prvky kolekci (např. *`emails`*`.COUNT`)
- Chování podobné vyšším programovacím jazykům (např. Perlu)



# Bulk-bind mechanismus

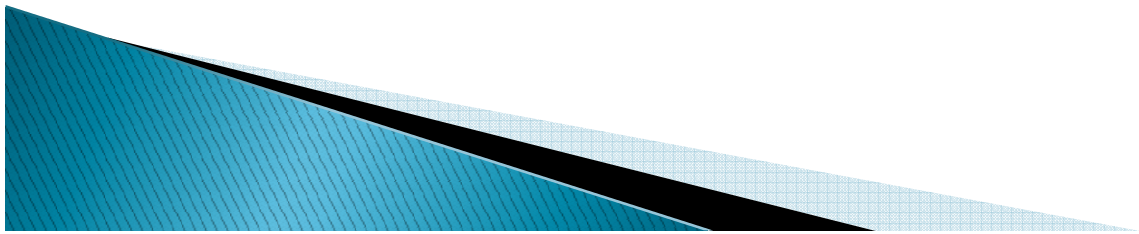
- Řídící konstrukce FORALL představuje cyklus přes všechny prvky, který provádí tzv. bulk-bind mechanismus umožňující seskupit jednotlivé SQL příkazy do jednoho celku
- Výsledný SQL příkaz se provede jen jednou

```
FORALL i IN depts.FIRST..depts.LAST
   DELETE FROM emp
        WHERE deptno = depts(i);
```



# Míchání SQL a kolekcí

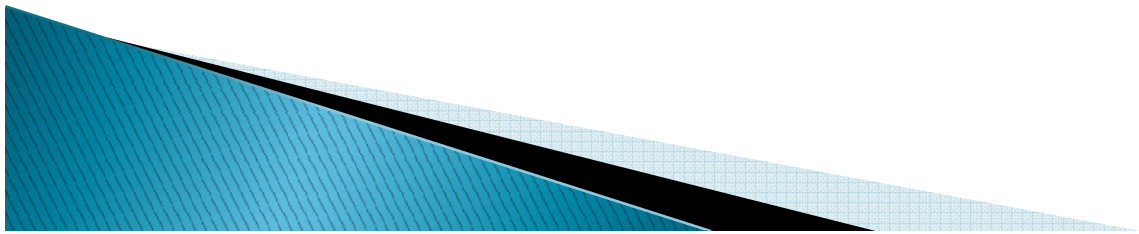
- Deklarací typu (CREATE  TYPE) lze vytvořit typ obsahující nested table nebo dynamické pole
- Tento typ může být doménou atributu nějaké relace (tabulky)
- Nelze ovšem užít asociativní pole jako typ v SQL relaci (tabulce)
- Elementárním typem může být i záznam



# Užití kolekcí uvnitř SQL

- Použije se tzv. konstruktor kolekce:
  *TypeName*(*element*$_1$, ..., *element*$_n$)
- Při použití v SQL se na místo prvku kolekce dosadí přímo kolekce:

```
INSERT INTO people
    (name, emails) VALUES
    ('Jan Novák',
    TEmail('novak@seznam.cz',
        'jn@email.cz'));
```



# Zisk dat z kolekce do PL/SQL

- Využitím BULK COLLECT INTO klauzule lze získat hodnoty z vícehodnot. dotazu do jedné kolekce

```
DECLARE
   TYPE tnum IS TABLE OF emp.no%TYPE;
   TYPE tname IS TABLE OF emp.nm%TYPE;
   enums tnum;
   names tname;
BEGIN
   SELECT no, nm BULK COLLECT
        INTO enums, names FROM emp;
END;
```

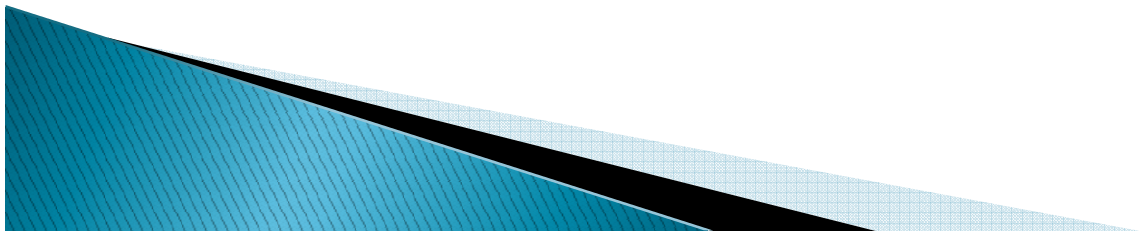
- Existuje příkaz FETCH…LIMIT pro omezení počtu ř.

# Záznamy

- Záznamy jsou heterogenní datové struktury
- Odpovídají záznamům v program. jazycích
- Deklarace:

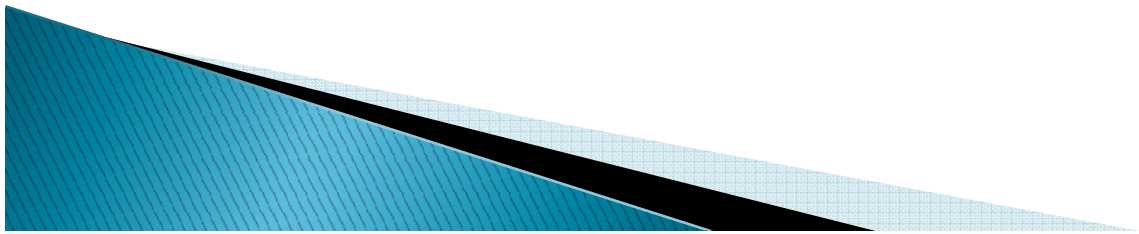
```
DECLARE TYPE adress IS RECORD (
   street VARCHAR2(50);
   city VARCHAR2(30);
   postal NUMBER(5) );
```

- Přístup: `myaddress.city`



# Tabulkové funkce

- Chceme získat výstup funkce jako datový zdroj v jiném SQL
- Použijeme funkci vracející nějakou kolekci (deklarujeme pomocí CREATE TYPE vhodnou nested table potřebného typu – struktury vracené tabulky)
- Pomocí TABLE() přetypujeme výstup této funkce v části FROM



# Příklad – definice

```
CREATE TYPE rada IS TABLE OF NUMBER;
CREATE FUNCTION seq (num NUMBER)
  RETURN rada AS
     sekvence rada;
     i NUMBER;
  BEGIN
     sekvence := rada();
     sekvence.EXTEND(num);
     FOR i IN 1 .. num LOOP
          sekvence(i) := i;
     END LOOP;
     RETURN sekvence;
  END;
```

# Příklad – použití

- Je možné nyní získat n-prvkovou řadu prostým voláním:

```
SELECT column_value
    FROM TABLE(seq(n));
```

- Uvedený mechanismu skrývá rozsáhlé využití – možnost generovat tabulky funkcemi on-line