A PL/SQL programozás alapjai

Oracle 9i környezetben

Hajnal Tibor
főiskolai adjunktus

2005

Tartalomjegyzék

1. Alapfogalmak

Karakterkészlet
- Angol ABC kis és nagybetűi (ékezetet ne használjunk)
- számjegyek 0-9
- speciális karakterek

PL/SQL-ben a kis és nagy betűk nem különböznek.

Aritmetikai operátorok

+	összeadás
-	kivonás
*	szorzás
/	osztás
**	hatványozás

Összehasonlító operátorok

<>, !=, ^=	nem egyenlő
<	kisebb
>	nagyobb
=	egyenlő

Egyéb szimbólumok

( )	lista elhatároló jel
;	utasítás vége jel
.	szeparátor (felh1.tabla)
’	karakterlánc határoló jel ’szöveg’
:=	értékadás x:=x+1;
||	konkatenáció nev:=vezeteknev||’ ’||keresztnev;
--	megjegyzés (egy sor)
/* */	megjegyzés

Azonosítók
- Az azonosítónak betűvel kell kezdődnie.
  ezjo, ezis123, 34eznem
- A kezdőbetűt követheti egy vagy több betű, számjegy, vagy a $, #, _ szimbólumok.
  x1, x_y, dfg#$
- Az azonosító nem lehet hosszabb 30 karakternél.
- Az azonosítóban nem szerepelhet szóköz.
  ez nem azonosito

Idézőjeles azonosítók

A PL/SQL-ben idézőjelek közé is zárhatjuk az azonosítókat, ekkor bármilyen karaktert használhatunk.
pl: ”Ez egy azonosító”, ”+/- és ez is”
Idézőjeles esetben a nagy és kisbetűk különböznek.

Foglalt szavak

for, loop, if ...
Ezek nem használhatók azonosítóként.

2. Fontosabb adattípusok

2.1. Numerikus típusok

- **number**
  Numerikus adat tárolására használható.

  Deklaráció:
  num_valtozo number(számjegyek, tizedesek)
  számjegyek: a számjegyek maximális száma, értéke 1 és 38 között lehet (elhagyva automatikusan 38 lesz a hossz)
  tizedesek: az összes számjegy közül mennyi legyen a tizedes

  pl:
  num_valtozo number(12,2);
  Ez legfeljebb 10 egész és 2 tizedes tárolására használható.

  Ha a tizedeseket megadó szám hiányzik, akkor egész tárolására használható a változó.
  egesz_valtozo number(3);

  Ha viszont negatív számot adunk meg, akkor a tizedesvesszőtől balra történik kerekítés annyi számjegyig, amennyit megadunk.
  kerekitett_valtozo number (4,-2); pl: 1234 tárolása esetén a változóba 1200 kerül.

  A deklarációnál kezdőérték is adható.
  kezdo_valtozo number(3):=100;

  Altípusai:
  - dec, decimal, numeric: fixpontos számok, max 38 decimális számjegyig
  - double precision, float: lebegőpontos számok, max 126 bináris számjegyig (kb. 38 decimális)
  - real: lebegőpontos szám, max 63 bináris számjegyig (kb. 18 decimális)
  - integer, int, smallint: egész számok, max 38 decimális számjegyig

- **binary_integer**
  Egész értéket kezel -2147483647..2147483647 tartományban. Fixpontosan tárolja az értékeket, így gyorsabb a műveletvégzés, mint a number adattípussal.

  Altípusai:
  natural 0..2147483647
  naturaln 0..2147483647 és NOT NULL azaz nem vehet fel nullértéket. Emiatt kezdőértékkel kell deklarálni.
  positive 1..2147483647
  positiven 1..2147483647 és NOT NULL
  singtype -1,0,1

- **pls_integer**
  Tárolási tartománya a binary_integer típusával azonos, de a tárolás kettes komplemensben történik, így még gyorsabb műveletvégzést biztosít. Túlcsordulás esetén azonban kivétel váltódik ki, míg binary_integer-nél ilyenkor automatikusan number típussá konvertálódik, amely nagyobb tartományú ábrázolást biztosít.

2.2. Karakteres típusok

- **char**
  Fix hosszúságú karakterláncok tárolására alkalmas.

  Deklaráció:
  valtozonev char(hossz karakterekben)

  pl:
  c_valt char(10);

  kezdőérték adása:
  c_valt char(10):=’kezdőérték’;

  Ha a megadott hossznál rövidebb sztringet kap értékül, akkor az szóközökkel kiegészülve kerül tárolásra.

- **varchar2**
  Változó hosszúságú alfanumerikus típus. A char típustól annyiban különbözik, hogy a megadott maximális hossznál rövidebb sztringek esetén nincs kiegészítés a teljes hosszra, vagyis csak a megadott karaktersorozat kerül tárolásra.

  Deklaráció:
  valtozonev varchar2(max_hossz karakterekben);

  pl:
  vc_valt varchar2(10);

  kezdőérték adása:
  vc_valt varchar2(10):=’kezdőérték’;

2.3. Dátum típus

- **date**
  Dátum tárolására alkalmas.
  datum_valt date;

2.4. Logikai típus

- **boolean**
  True vagy false értéket vehet fel.

  Deklaráció:
  log_valt boolean;

2.5. Bináris típus

- **raw**
  Bináris adatok kezelésére alkalmas. Például képek tárolása oldható meg vele.

  Deklaráció:
  valtozonev raw(h);
  Ahol h a hosszat határozza meg bájtokban, értéke 1..32767 tartományba eshet.

2.6. LOB típusok

Multimédiás adatok (szöveg, kép, hang, videó) kezelésére alkalmas, max 4GB méretig.

- **bfile**
  Nagyméretű bináris állományokat tárol, de nem az adatbázisban, hanem az operációs rendszer szintjén. Lényegében az adatbázisban egy mutató mutat a megfelelő fájlra.

- **blob**
  Nagyméretű bináris objektumokat tárol az adatbázisban. Maximális méret 4GB.

- **clob**
  Nagyméretű karaktersorozatokat tárol az adatbázisban. Maximális méret 4GB.

3. PL/SQL blokkszerkezet

Blokktípusok:

- **Névtelen blokk**
  Névvel nem rendelkező PL/SQL kód. Az adatbázisban nem kerül tárolásra. A kliens felületen begépelve egyszer lefut, de külső szöveges állományban is tárolhatjuk, amit a START fájlnév paranccsal lehet futtatni. Ilyen névtelen blokk a triggerekben használt blokk is.

  Szerkezete:

  ```
  [DECLARE]
      változók deklarálása
  BEGIN
      utasítások
  [EXCEPTION]
      kivételkezelés
  END;
  /
  ```

- **Eljárás (procedure)**
  Az adatbázisban tárolt program. Névvel rendelkező PL/SQL blokk. Műveletek végezhetők vele, értékeket lehet átadni neki, illetve adhat is vissza értéket. Meghívható manuálisan és más PL/SQL blokkból is.
  (Szerkezetét később tárgyaljuk.)

- **Függvény (function)**
  Az eljáráshoz hasonló. Mindig van egy visszatérési értéke, amelyet a hívás helyére ad vissza. Ezért a függvényeket kifejezés részeként hívjuk meg.
  pl: atlag:=atlag_fv(x,y);
  (Szerkezetét később tárgyaljuk.)

4. A PL/SQL programozás kezdeti lépései

4.1. Bevezető feladat

Kezdjük talán a szokásos „hello világ” típusú programmal.
Ehhez először adjuk ki a következő parancsot:

```
set serveroutput on;
```

Ezzel egy környezeti változót állítunk át, annak érdekében, hogy a szerver által küldött szöveg megjelenjen a képernyőn. Egyébként a környezeti változók állapota a **show all** paranccsal megtekinthető.
Az SQL plus felületre gépeljük be a következő névtelen PL/SQL blokkot:

```
begin
dbms_output.put_line('Hello világ!');
end;
/
```

A blokkot SQL plus környezetben a / karakterrel le kell zárni!

Ha hibátlan volt a kód, akkor a következő üzenetet kapjuk:

```
A PL/SQL eljárás sikeresen befejeződött.
```

A „Hello világ!” felirat pedig megjelent a képernyőn.

A begépelt kódot, ha újra akarjuk futtatni, akkor azt újra kellene gépelni. Ez elég kényelmetlen megoldás lenne, ezért van lehetőség arra, hogy a kódot egy szövegfájlban tároljuk, és onnan futtassuk.
Hozzunk létre egy szövegfájlt, mondjuk **szkript.sql** névvel. Bárhol létrehozható a lemezen, de készüljünk fel arra, hogy futtatáskor a teljes elérési utat meg kell adni. Ha ezt el akarjuk kerülni, akkor az Oracle munkakönyvtárába készítsük el, amely alapértelmezésben a **BIN** könyvtár az Oracle főkönyvtárán belül. Ha oda hozzuk létre a fájlt, akkor csak a nevét kell megadni futtatáskor.
Írjuk be a fájlba az előző kódot, majd adjuk ki a futtatásra a következő parancsot:

```
start szkript
```

(Ha a kiterjesztés SQL, akkor futtatáskor ennek megadása elhagyható)
Az eredmény ugyanaz mint az előző esetben, amikor begépeltük a kódot.

A szkriptállományoknak fontos szerepe van. Több SQL utasítás is futtatható egymás után egy ilyen állomány segítségével. SQL parancsokkal tölthetjük fel a fájlt, majd futtatva azt egymás után végrehajtásra kerülnek a parancsok.

4.2. Adatok bekérése futás közben

SQLplus környezetben az **&** karakter segítségével lehetséges futás közben adatot bekérni a felhasználótól.
Ha az **&** karakter szerepel egy karaktersorozat elején, akkor futás közben megjelenik egy üzenet a képernyőn, ami arra kér bennünket, hogy adjuk meg a karaktersorozat értékét. A megadott érték bekerül a programkódba a karaktersorozat helyére.
Fontos megemlíteni, hogy a folyamat a kód futásának legelején lezajlik, a vezérlés a program első sorára csak ezután kerül. Emiatt nem lehet az adatbekérés előtt más utasítást végrehajtani.
A módszer használható utasítások valamely paraméterének megadásakor, vagy változók értékének megadásakor is.
Ügyelni kell azonban a paraméter, vagy változó adattípusára:

karakteres típusnál idézőjelek közé kell írni a karaktersorozatot : ’&szöveg’
numerikus típusnál nem kell idézőjel: &szöveg

A karaktersorozatra vonatkozik még két megkötés:

- nem tartalmazhat szóközt
- max 30 karakter hosszú lehet

Példa:
Módosítsuk a „Hello világ” típusú programunkat úgy, hogy futás közben kérje be a kiírandó szöveget. A kiíratő utasítás paraméterét futás közben adjuk meg.

```
begin
dbms_output.put_line('&Kiírandó_szöveg');
end;
/
```

Nézzünk egy példát konstans és változó deklarálására. Készítsük el a következő kis programot, amely kiszámítja egy kör területét, melynek sugarát futás közben kérjük be.

```
declare
 pi constant number(8,7):=3.1415926;
 terulet number(14,2);
begin
 terulet:=&sugár**2*pi;
 dbms_output.put_line(terulet);
end;
/
```

Rövidebb megoldás:

```
begin
dbms_output.put_line(’A kör területe=’||&sugár**2*3.1415926);
end;
/
```

4.3. Feladatok

1. Írjunk egy programot, amely bekéri egy személy vezeték- illetve keresztnevét, majd összefűzi őket, úgy, hogy egy szóköz kerüljön közéjük és kiírja a képernyőre.
2. Készítsünk sebességszámító programot. Kérjük be a megtett utat, és az ehhez szükséges időt. Írjuk ki a sebességet km/h-ban és m/s-ban is. Jelenjen meg a két bekért érték is.

5. Vezérlési szerkezetek

5.1. Feltételes elágaztatás (IF)

IF - THEN

```
IF feltétel THEN
 utasítások;
END IF;
```

Az IF-THEN utasítások egymásba ágyazhatók:

```
IF feltétel1 THEN
 IF feltétel2 THEN
  utasítások;
 END IF;
END IF;
```

IF - THEN - ELSE

```
IF feltétel THEN
 utasítások;
ELSE
 utasítások;
END IF;
```

Ezek az utasítások is egymásba ágyazhatók:

```
IF feltétel1 THEN
 utasítások;
ELSE
  IF feltétel2 THEN
   utasítások;
  ELSE
   utasítások;
  END IF;
END IF;
```

IF - THEN - ELSIF

```
IF feltétel1 THEN
 utasítások;
ELSIF feltétel2 THEN
 utasítások;
ELSE
 utasítások;
END IF;
```

5.2. Többirányú feltételes elágaztatás (CASE)

Az első igaz feltétel utáni ágak feltételei már nem kerülnek kiértékelésre.

5.2.1. CASE utasítás szelektor használatával

```
declare
 szelektor char(1);
begin
 szelektor:='&szín_kód';
 case szelektor
        when 'p' then dbms_output.put_line('piros');
        when 'z' then dbms_output.put_line('zöld');
        when 's' then dbms_output.put_line('sárga');
        when 'k' then dbms_output.put_line('kék');
        else dbms_output.put_line('A szín ismeretlen');
 end case;
end;
/
```

5.2.2. CASE utasítás szelektor nélkül

```
declare
 szam int;
begin
 szam:=4;
 case
        when szam mod 2=0 then dbms_output.put_line('páros');
        when szam<5 then dbms_output.put_line('kisebb ötnél');
        when szam>5 then dbms_output.put_line('nagyobb ötnél');
        else dbms_output.put_line('A szám 5-tel egyenlő');
 end case;
end;
/
```

6. Ciklusszervezés

6.1. LOOP - EXIT - END LOOP

```
LOOP
 utasítások;
  IF kilépési feltétel THEN
   EXIT;
  END IF;
END LOOP;
```

Példa: Írjuk ki a képernyőre a 20-nál kisebb páros természetes számokat.

```
declare
 szamlalo number(2):=2;
begin
 loop
  dbms_output.put_line(szamlalo);
  szamlalo:=szamlalo+2;
  if szamlalo>18 then
   exit;
  end if;
 end loop;
end;
/
```

6.2. LOOP - EXIT WHEN - END LOOP

Az előző konstrukcióhoz hasonló, a kilépési feltétel vizsgálatánál megspóroljuk az **if** utasítást.

```
LOOP
 utasítások;
 EXIT WHEN kilépési feltétel;
END LOOP;
```

Az előző példa ezzel a megoldással:

```
declare
 szamlalo number(2):=2;
begin
 loop
  dbms_output.put_line(szamlalo);
  szamlalo:=szamlalo+2;
  exit when szamlalo>18;
 end loop;
end;
/
```

6.3. WHILE - LOOP - END LOOP

Elöl tesztelő ciklus. Minden iteráció előtt megvizsgálja a futási feltételt.

```
WHILE futási feltétel LOOP
```

```
 utasítások;
END LOOP;
```

Oldjuk meg a példánkat így is:

```
declare
 szamlalo number(2):=2;
begin
 while szamlalo<20 loop
  dbms_output.put_line(szamlalo);
  szamlalo:=szamlalo+2;
 end loop;
end;
/
```

6.4. FOR - IN - LOOP - END LOOP

Ezzel a konstrukcióval előre meghatározott számú esetben hajthatjuk végre a ciklus magját. A ciklusváltozót előre nem kell deklarálni.

```
FOR ciklusváltozó IN [REVERSE] alsóhatár..felsőhatár LOOP
 utasítások;
END LOOP;
```

A példánk:

```
begin
 for szamlalo in 1..9 loop
  dbms_output.put_line(szamlalo*2);
 end loop;
end;
/
```

6.5. Feladatok

1. Futás közben kérjünk be a felhasználótól egy számot. Vizsgáljuk meg, hogy a szám páros e vagy páratlan. Írassuk ki a vizsgálat eredményét.
2. Készítsünk egy számológépet, amely a négy alapművelet elvégzésére képes. A program kérje be a felhasználótól a műveleti jelet és a két operandus értékét. Végezze el a kért műveletet és az eredményt írja ki. Nem létező műveleti jel megadását is jelezze. (A feladatot a CASE szerkezet segítségével oldjuk meg.)
3. Írassuk ki 1-től 10-ig a természetes számokat a képernyőre növekvő, majd csökkenő sorrendben. (LOOP - END LOOP ciklust használjunk.)
4. Jelenítsük meg a szorzótáblákat 1-től 10-ig, az alábbi formában:
   1*1=1
   1*2=2
   .
   .
   .
   10*9=90
   10*10=100
   (FOR ciklust használjunk.)

7. Kivételek

A kód végrehajtása közben különböző hibák, azaz kivételek fordulhatnak elő. Erről a szerver hibaüzenettel értesít minket. Lehetőségünk van viszont arra, hogy magunk kezeljük le a különböző kivételeket. Ez azt jelenti, hogy a kivétel kiváltódásakor nem a szerver által küldött hibaüzenet jelenik meg, hanem az általunk definiált feladat hajtódik végre.
A kódblokk végén van lehetőség a kivételkezelő rész készítésére, amelyet az **exception** kulcsszó vezet be. Amikor a kód végrehajtása közben hiba következik be, akkor a vezérlés automatikusan a kivételkezelő részre kerül.

Néhány kivétel:

- **no_data_found** Akkor következik be, ha egy **select** utasítás 0 sort eredményez.
- **too_many_rows** Akkor következik be, ha egy **select**, aminek egy sort kellett volna visszaadnia, több sort ad eredményül.
- **dup_val_on_index** Akkor következik be, ha **insert** vagy **update** utasítás egyedi indexben többször előforduló értéket eredményezne.
- **value_error** Akkor lép fel ha egy változó értéke sérül. pl: ha egy **varchar2(2)**-es változónak **'ABC'** értéket akarunk adni.
- **zero_divide** Nullával való osztásnál lép fel.

A kivételkezelő rész szerkezete:

```
BEGIN
 ...
EXCEPTION
 WHEN kivétel1 THEN
  utasítások;
  ...
 WHEN kivétel2 [OR kivétel3 ...] THEN
  utasítások;
  ...
 WHEN OTHERS THEN
  utasítások;
END;
/
```

A **when others** rész akkor fut le, ha általunk nem kidolgozott kivétel következik be. Ha elhagyjuk, akkor ilyenkor az alapértelmezett hibaüzenetet kapjuk. Mindig a kivételkezelő végére kell írni! He ezt írnánk előre, akkor minden kivétel esetén ő futna le.

Példa: Írjunk egy egyszerű osztó programot, amely lekezeli a nullával való osztás esetét.

```
declare
 x int;
 y int;
begin
 x:=&osztandó;
 y:=&osztó;
 dbms_output.put_line(x/y);
exception
 when zero_divide then dbms_output.put_line('0-val való osztás történt!');
```

```
 when others then dbms_output.put_line('Egyéb hiba történt!');
end;
/
```

Saját kivételek deklarálása

Saját magunk is deklarálhatunk kivételt, ilyenkor a programkódban kell azt kiváltanunk a **raise** utasítás segítségével.

Szerkezete:

```
DECLARE
 kivételnév EXCEPTION;
BEGIN
 .
 .
 RAISE kivételnév;
 .
 .
EXCEPTION
 WHEN kivételnév THEN...
END;
/
```

Az előző példa megoldása saját kivétel segítségével:

```
declare
 x int;
 y int;
 hiba exception;
begin
 x:=&osztandó;
 y:=&osztó;
 if y=0 then
  raise hiba;
 end if;
 dbms_output.put_line(x/y);
exception
 when hiba then dbms_output.put_line('0-val való osztás történt!');
 when others then dbms_output.put_line('Egyéb hiba történt!');
end;
```

8. Adatmanipulációs SQL utasítások használata (INSERT, UPDATE, DELETE)

PL/SQL kódban ugyanúgy használhatók, mint egyszerű SQL utasítások. Hatásukban, és formájukban nincs változás.

Példa:
Hozzunk létre egy táblát, amely autókkal kapcsolatos adatokat tárol:
create table auto(rendszam char(7), tipus varchar2(30), evjarat number(4));

Vegyünk fel két rekordot:
insert into auto values('UPD-123','Opel',1991);
insert into auto values('DLT-123','Trabant',1981);

A következő programban példát láthatunk az adatmanipulációs utasítások használatára:
```
begin
insert into auto values('AAA-123','Lada',1984);
insert into auto (rendszam,tipus) values('BBB-123','Skoda');
update auto set evjarat=1993 where rendszam='UPD-123';
delete from auto where rendszam='DLT-123';
end;
/
```

A tábla tartalma a várt eredményt mutatja, a PL/SQL kódban kiadott SQL utasítások ugyanúgy elvégezték feladatukat, mintha egyszerűen a kliensfelületen keresztül adtuk volna ki őket.

9. Kurzorok, vagyis a lekérdező SQL utasítás használata (SELECT)

A SELECT utasítás nem hajtható végre olyan formában, mint egy egyszerű SQL utasítás. A SELECT-nek ugyanis eredménye van, ami egyszerű esetben a képernyőn megjelenik. PL/SQL kódban viszont gondoskodnunk kell valamilyen tárolóeszközről, amibe a SELECT eredménye bekerül. Ezt a tárolóeszközt hívjuk kurzornak.

A kurzorok használata során olyan változókra van szükség, amely adattípusa egyező kell, hogy legyen a kezelt adattábla mezőinek, vagy akár egész sorának típusával.
Lehet manuálisan deklarálni őket oly módon, hogy megvizsgáljuk a tábla mezőjének típusát, majd ezzel azonos típusra deklaráljuk a változót.
Van azonban kényelmesebb megoldás, ugyanis megadhatjuk a deklarációnál azt is, hogy egy már létező objektum (pl: tábla mezője) típusával legyen azonos a változó típusa.

Változó deklarálása egy tábla mezőjével azonos típusra:

változónév táblanév.mezőnév%TYPE;

Rekord típusú összetett változó deklarálása, amely egy tábla rekordszerkezetével lesz azonos:

változónév táblanév%ROWTYPE;

A változó összetett, egyes részeire a szeparátor segítségével tudunk hivatkozni:

változónév.mezőnév

Rekord típusú változó deklarációja manuálisan:

Először egy összetett adattípust deklarálunk:

```
TYPE típusnév IS RECORD
 (alváltozó_1 típus,
  alváltozó_2 típus,
  .
  .

  alváltozó_n típus);
```

Majd erre a típusra deklaráljuk a rekord típusú változót:

változónév típusnév;

A kurzoroknak két típusa létezik:

- Implicit kurzor
- Explicit kurzor

9.1. Implicit kurzorok

Automatikusan létrejön, amikor egy változónak egy lekérdezés eredményét adjuk értékül. Amikor egy lekérdezés pontosan egy sort ad vissza, akkor tökéletesen megfelel a feladatnak. Implicit kurzorok használata esetén a SELECT utasítás annyiban különbözik egy közönséges SQL SELECT-től, hogy kiegészül egy INTO utasításrésszel. Az INTO után egy változót kell megadni, amelybe a select eredménye kerül.

Szintaxis:
SELECT mezők FROM tábla INTO változó WHERE ...

Az implicit kurzorok egy visszaadott sort várnak, ha ez nem így történik, akkor kivétel következik be. Ennek oka, hogy az INTO után megadott változóba csak egy adat fér be. Több adat illetve nullérték nem adható értékül a változónak. Implicit kurzorok használata közben leggyakrabban a **no_data_found** és a **too_many_rows** kivételek következnek be.

Példa: Írjunk egy programot, amely segíti az **auto** táblába az újabb járművek felvételét. A program ellenőrizze felvitel előtt, hogy a rendszám létezik e már a táblában. Ha igen, akkor a beszúrás ne történjen meg és erről tájékoztató üzenet jelenjen meg a képernyőn.

```
declare
 v_rendszam auto.rendszam%type;
 v_tipus auto.tipus%type;
 v_evjarat auto.evjarat%type;
 ell_rendszam auto.rendszam%type;
begin
 v_rendszam:='&rendszám';
 v_tipus:='&típus';
 v_evjarat:=&évjárat;
 select rendszam into ell_rendszam from auto where rendszam=v_rendszam;
 dbms_output.put_line('Ilyen rendszámú autó már van felvéve!');
```

```
exception
 when no_data_found then
 insert into auto values(v_rendszam, v_tipus, v_evjarat);
 commit;
end;
/
```

9.2. Explicit kurzorok

Mi magunk deklaráljuk, a változók deklarálásához hasonlóan. Explicit kurzort kell használni, ha a lekérdezés nulla, vagy egynél több sort eredményez. Több sort eredményező lekérdezések feldolgozásához a kurzort ciklussal kombináljuk.

Deklaráció:
CURSOR kurzornév IS select utasítás;
A kurzor a deklarációnál megadott egyszerű SQL SELECT eredményét kapja meg.

A kurzort használat előtt meg kell nyitni:
OPEN kurzornév;

A kurzort a vele való munka befejeztével le kell zárni:
CLOSE kurzornév;

A megnyitott kurzorból a FETCH utasítással lehet sort lehívni:
FETCH kurzornév INTO változónév;
A sor a megadott változóba kerül. Az első FETCH az első sort hívja le, a második a másodikat, és így tovább, amíg a sorok el nem fogynak.

A kurzorok rendelkeznek attribútumokkal, melyek segítségével, a kurzorral kapcsolatos különböző információkhoz juthatunk.
A kurzorattribútumok:

- **kurzornév%FOUND** : Logikai értéket tartalmaz. Igaz, ha a FETCH utasításnak sikerült sort lehívnia, hamis, ha nem. A FETCH akkor nem tud sort lehívni, ha a kurzor 0 sort tartalmaz, vagy ha már az utolsó sor is lehívásra került.
- **kurzornév%NOTFOUND** : Logikai értéket tartalmaz, a FOUND ellentéte.
- **kurzornév%ROWCOUNT** : Numerikus értéket tartalmaz. A FETCH utasítással eddig lekért sorok számát adja vissza.

Az előző példát készítsük el explicit kurzor segítségével:

```
declare
 v_rendszam auto.rendszam%type;
 v_tipus auto.tipus%type;
 v_evjarat auto.evjarat%type;
 puffer auto.rendszam%type;
 cursor ell_kurzor is
  select rendszam from auto where rendszam=v_rendszam;
begin
 v_rendszam:='&rendszám';
 v_tipus:='&típus';
 v_evjarat:=&évjárat;
```

```
 open ell_kurzor;
  fetch ell_kurzor into puffer;
  if ell_kurzor%found then
   dbms_output.put_line('Ilyen rendszámú autó már van felvéve!');
  end if;
  if ell_kurzor%notfound then
   insert into auto values(upper(v_rendszam), v_tipus, v_evjarat);
   commit;
  end if;
 close ell_kurzor;
end;
/
```

Példa: írassuk ki a képernyőre a 3 legöregebb autót.

A feladat előtt a táblát töltsük fel tesztadatokkal:

```
delete from auto;
insert into auto values('AAA-001','audi',1998);
insert into auto values('AAA-002','volvo',1998);
insert into auto values('AAA-222','bmw',1998);
insert into auto values('BBB-001','barkasz',1982);
insert into auto values('BBB-002','lada',1985);
insert into auto values('AAA-123','trabant',1989);
insert into auto values('SSS-001','skoda',1975);
insert into auto values('VVV-001','vw',1969);
commit;
```

```
declare
 v_rekord auto%rowtype;
 cursor oreg_kurzor is
  select rendszam,tipus,evjarat from auto order by evjarat;
begin
 open oreg_kurzor;
 loop
  fetch oreg_kurzor into v_rekord;
  dbms_output.put_line(v_rekord.rendszam||' '||v_rekord.tipus||' '||v_rekord.evjarat);
  exit when oreg_kurzor%rowcount>=3;
 end loop;
 close oreg_kurzor;
end;
/
```

9.3. A kurzor for ciklus

Akkor használjuk, amikor a kurzorban lévő lekérdezés összes során végig akarunk lépni valamilyen okból (műveletvégzés, stb). Ilyenkor nem kell a kurzort megnyitni, illetve lezárni, mindez automatikusan történik. A kurzor használatának megkezdésekor automatikusan létrejön egy rekord típusú változó a kurzor deklarációjánál megadott oszlopoknak megfelelő

mezőkkel (gyakorlatilag ez a ciklusváltozó). A ciklusmag a kurzor összes sorára végrehajtódik, majd ha már elfogytak az adatok, akkor automatikusan lezárul.

Szinatxis:
FOR rekordváltozó IN kurzornév LOOP
ciklusmag;
END LOOP;

Példa: Az 1985-nél korábban gyártott autókat selejtezésre szánjuk, ezért töröljük a táblából ezen járművek adatait. (Természetesen a feladat egyszerű SQL delete utasítással megoldható, de itt most nem ez a lényeg...)

```
declare
 cursor selejt_kurzor is select * from auto;
begin
 for selejt_rekord in selejt_kurzor loop
  if selejt_rekord.evjarat<1985 then
   delete from auto where rendszam=selejt_rekord.rendszam;
  end if;
 end loop;
 commit;
end;
/
```

Ha a select utasítást beágyazzuk a for ciklusba, akkor még kurzordeklarációra sincs szükség:

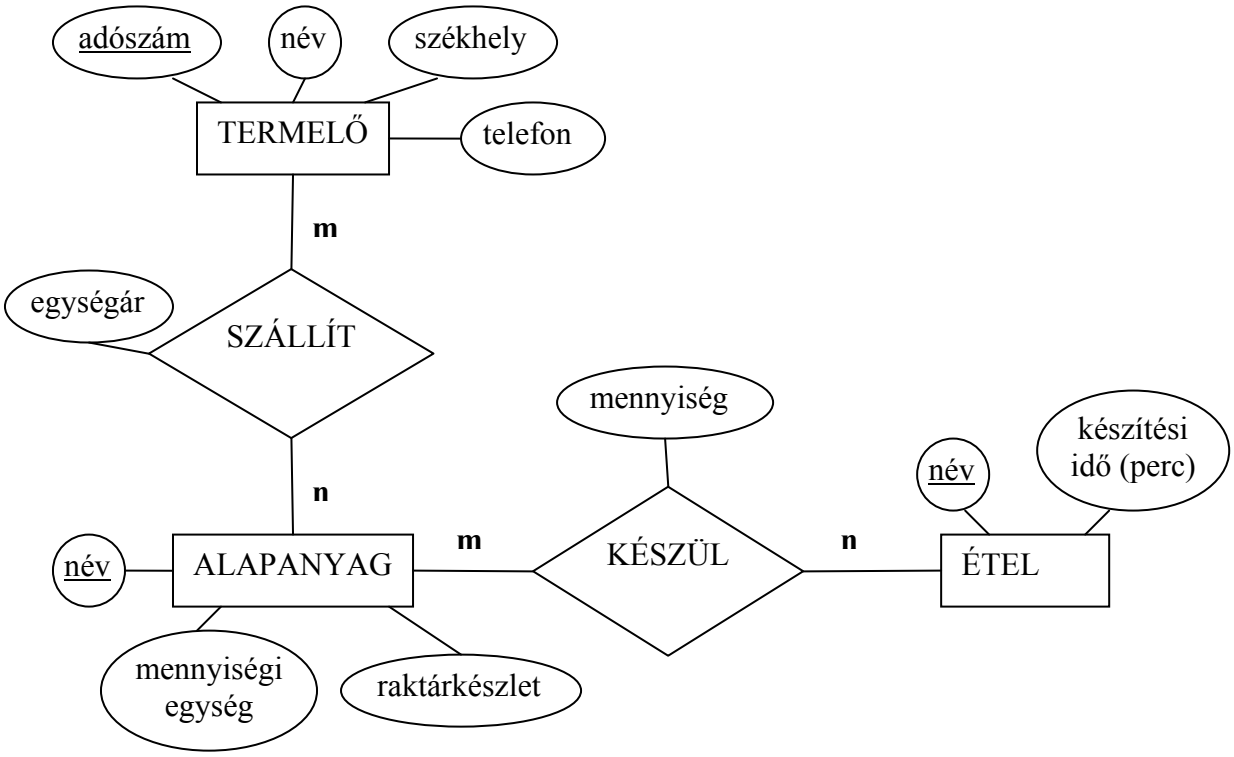
```
begin
 for selejt_rekord in (select * from auto) loop
  if selejt_rekord.evjarat<1985 then
   delete from auto where rendszam=selejt_rekord.rendszam;
  end if;
 end loop;
 commit;
end;
/
```

9.4. Feladatok

1. Írassuk ki a legöregebb autó adatait. A megoldáshoz implicit kurzort használjunk. Kezeljük le azt az esetet, ha az adattábla üres, és azt is, ha a legrégebbi évjárat több autónál is szerepel.
2. Rendezzük le az **auto** tábla sorait rendszám szerint. A rendezettség fizikai legyen, vagyis a rekordok sorrendje megfeleljen a rendszám szerinti sorrendnek.
3. Írassuk azokat az autókat, amelyeket ugyanabban az évben gyártottak.

10. Gyakorló feladatok

Egy étterem nyilvántartó rendszerének egyed-kapcsolat modellje a következő:



Az egyed-kapcsolat modell leképezéséből az alábbi relációs modellt kaptuk:

TERMELO (adoszam, nev, szekhely, telefon)

SZALLIT (adoszam, anyagnev, egysegar)

ALAPANYAG (nev, mennyegys, raktarkeszlet)

KESZUL (etelnev, anyagnev, mennyiseg)

ETEL (nev, ido)

A mezők típusa:

TERMELO tábla
adoszam: char(10)
nev: varchar2(30)
szekhely: varchar2(50)
telefon: char(9)

SZALLIT tábla
adoszam: char(10)
anyagnev: varchar2(20)
egysegar: number(5)

ALAPANYAG tábla
nev: varchar2(20)
mennyegys: varchar2(2)
raktarkeszlet: number(5,2)

KESZUL tábla
etelnev: varchar2(30)
anyagnev: varchar2(20)
mennyiseg: number(3,2)

ETEL tábla
nev: varchar2(30)
ido: number(3)

Feladatok:

1. **feladat**
   Készítsünk egy olyan programot, amely kiírja azon alapanyagok minden adatát (név, mennyiségi egység, raktárkészlet), amelyek raktáron lévő mennyisége egy felhasználótól bekért érték alatt van.

   Megoldás:

```
declare
bekert number(3);
begin
bekert:=&minimális_készlet;
for rekord in (select * from alapanyag) loop
 if rekord.raktarkeszlet<bekert then
  dbms_output.put_line(rekord.nev||' '||rekord.mennyegys||' '||rekord.raktarkeszlet);
 end if;
end loop;
end;
/
```

   Rövidebb megoldás:

```
begin
for rekord in (select * from alapanyag where raktarkeszlet<&minimális_készlet) loop
 dbms_output.put_line(rekord.nev||' '||rekord.mennyegys||' '||rekord.raktarkeszlet);
end loop;
end;
/
```

2. feladat

Írjuk ki a képernyőre azt a 10 alapanyagot, melyekből a legkevesebb van raktáron. Jelenjen meg az alapanyagok minden adata, a hozzájuk kapcsolódó termelők minden adata és az egységáruk is.

Megoldás:

```
declare
 szaml number(2):=0;
 puffer alapanyag.nev%type:=' ';
 cursor tizek_kurzor is
  select alapanyag.nev,mennyegys,raktarkeszlet,egysegar,
   termelo.adoszam,termelo.nev as tnev,szekhely,telefon
   from alapanyag,szallit,termelo
   where alapanyag.nev=szallit.anyagnev
   and termelo.adoszam=szallit.adoszam order by raktarkeszlet,alapanyag.nev;
 rekord tizek_kurzor%rowtype;
begin
 open tizek_kurzor;
 loop
 fetch tizek_kurzor into rekord;
  if rekord.nev<>puffer then
   puffer:=rekord.nev;
   szaml:=szaml+1;
  end if;
 exit when szaml>10;
dbms_output.put_line(rekord.nev||' '||rekord.mennyegys||' '||rekord.raktarkeszlet
||' '||rekord.egysegar||' '||rekord.adoszam||' '||rekord.tnev
||' '||rekord.szekhely||' '||rekord.telefon);
 end loop;
 close tizek_kurzor;
end;
/
```

Másik megoldás:

```
declare
 cursor anyagkurzor is select * from alapanyag order by raktarkeszlet;
 anyagrekord alapanyag%rowtype;
 termelorekord termelo%rowtype;
begin
 open anyagkurzor;
 while anyagkurzor%rowcount< 10 loop
  fetch anyagkurzor into anyagrekord;
   dbms_output.put_line('----------');
   dbms_output.put_line(anyagrekord.nev||' '||anyagrekord.mennyegys||'
'||anyagrekord.raktarkeszlet);
   for szallitrekord in (select * from szallit where anyagnev=anyagrekord.nev) loop
    select * into termelorekord from termelo where adoszam=szallitrekord.adoszam;
    dbms_output.put_line(termelorekord.adoszam||' '||termelorekord.nev||' '||
      termelorekord.szekhely||' '||termelorekord.telefon||' '||szallitrekord.egysegar);
   end loop;
 end loop;
 close anyagkurzor;
end;
/
```

3. **feladat**
   Írjuk ki azon ételek nevét, amelyekhez valamely alapanyagból nincs 100 adagra való raktáron. Jelenjen meg az is, hogy mennyi az aktuális készlet és mennyi hiányzik 100 adaghoz.

4. **feladat**
   Írjunk egy olyan programot, amely kitörli a bekért adószámú termelőt, a szállít táblában lévő rekordjaival együtt. Ha van olyan alapanyag, melyet csak ő szállított, akkor írja ki a képernyőre, hogy melyik ez az anyag, ugyanis új termelőt kell keresni hozzá.

5. **feladat**
   Készítsünk sietős vendégek számára ételválasztást megkönnyítő programot. A program kérje be a maximális várakozási időt. Az ennél rövidebb készítési idejű ételek közül írja ki azokat, amelyekhez van alapanyag raktáron. Írja ki azt is, hogy hány adagra van alapanyag.

6. **feladat**
   Egy bekért ételnév alapján írjuk ki annak minimális és maximális alapanyagköltségét.

   Megoldás:

```
declare
egysegarvalt szallit.egysegar%type;
minar number(5);
maxar number(5);
begin
 minar:=0;
 maxar:=0;
 for etel_rekord in (select * from keszul where etelnev='&étel')
 loop
  select min(egysegar) into egysegarvalt from szallit where
anyagnev=etel_rekord.anyagnev;
  minar:=minar+egysegarvalt*etel_rekord.mennyiseg;
  select max(egysegar) into egysegarvalt from szallit where
anyagnev=etel_rekord.anyagnev;
  maxar:=maxar+egysegarvalt*etel_rekord.mennyiseg;
 end loop;
dbms_output.put_line('Minimális anyagköltség: '||minar||' Ft');
dbms_output.put_line('Maximális anyagköltség: '||maxar||' Ft');
end;
/
```

7. **feladat**
   Írjunk egy olyan programot, amely bekéri a készíteni kívánt étel nevét és az adag számát. Nézze meg, hogy van e hozzá elég alapanyag, ha van, vonja le a raktárkészletből. Ha nincs, írja ki, hogy miből mennyit kell rendelni és, hogy melyik termelőtől. Ha több termelő is rendelkezik a kívánt alapanyaggal, akkor a legolcsóbbat válassza ki.

11. Triggerek

A trigger egy olyan program, amely bizonyos események bekövetkezése esetén automatikusan lefut.

Az esemény, amely elindíthatja a triggert:

- Táblán, vagy nézeten végrehajtott INSERT, UPDATE vagy DELETE utasítás.
- Felhasználói be- és kijelentkezés.
- Adatbázis elindítása, leállítása.
- Szerverhibák.
- Egyes DDL utasítások.

Táblákhoz, vagy nézetekhez készített triggerek (Insert, update vagy delete indítja) szorosan kapcsolódnak az objektumhoz, amihez kapcsolva lettek. Ha töröljük a táblát, vagy nézetet, amihez trigger van kapcsolva, akkor a trigger is törlődik.

A többi trigger, séma vagy adatbázis szinten hozható létre, az adatbázis adminisztrátor jogkörébe tartozik az elkészítésük.

Triggerek típusai:

- Utasításszintű trigger: 1-szer hajtódik végre. Nem függ a kiváltó utasítás által kezelt sorok számától.
  Például, ha egy update utasítás több sort is módosít, a trigger akkor is csak egyszer fut le. Akkor is lefut, ha egy sort sem érint az update.

- Sorszintű trigger: annyiszor hajtódik végre ahány sort érint. A FOR EACH ROW sorról ismerhető fel.
  Például, ha egy update több sort is érint, akkor minden egyes módosított sor újra aktiválja a triggert.

A triggerek időzítése:

- Before: előbb a trigger fut le, a kiváltó esemény (pl. insert) csak utána.
  Csak táblához kapcsolható.
- After: A kiváltó esemény végrehajtása után fut a trigger.
  Csak táblához kapcsolható.
- Instead of: A kiváltó művelet helyett a trigger fut le.
  Csak sorszintű lehet. Csak nézethez kapcsolható.

11.1. Trigger létrehozása:

```
CREATE [OR REPLACE] TRIGGER triggernév
BEFORE/AFTER/INSTEAD OF DELETE/INSERT/UPDATE [OF oszlop[,oszlop2]…]
        [OR DELETE/INSERT/UPDATE [OF oszlop[,oszlop2]…]]…
ON tábla/nézet
[FOR EACH ROW]
[WHEN feltétel]
[DECLARE lokális változók]
```

```
BEGIN
  törzs (PL/SQL blokk)
END;
/
```

A trigger törzse: Szabályos, névtelen PL/SQL blokk.

Sorszintű triggerekben a rekordok módosítás előtti és utáni értékeire az OLD és NEW szavakkal lehet hivatkozni.

Update: OLD (módosítás előtti érték), NEW(módosítás utáni érték)
Delete: Csak az OLD használható.
Insert: Csak a NEW használható.

Használhatók a WHEN feltételben: NEW.mezőnév,
vagy a törzsben műveletnél: :NEW.mezőnév

Trigger engedélyezése, tiltása
ALTER TRIGGER triggernév ENABLE/DISABLE;

Trigger törlése
DROP TRIGGER triggernév;

11.2. Példa utasításszintű és sorszintű triggerre

Az éttermi adatbázis ALAPANYAG táblájában, ha a raktárkészlet módosul, akkor arról egy szöveges figyelmeztető szöveg jelenjen meg.

Először utasításszintű triggert készítünk:

```
CREATE OR REPLACE TRIGGER anyag_upd
 AFTER UPDATE OF raktarkeszlet ON alapanyag
BEGIN
 DBMS_OUTPUT.PUT_LINE('A raktárkészlet módosult!');
END;
/
```

Próbáljuk ki az alábbi UPDATE paranccsal a trigger működését.
```
UPDATE alapanyag SET raktarkeszlet=raktarkeszlet*1.1;
```

Módosítsuk a triggert sorszintűvé:

```
CREATE OR REPLACE TRIGGER anyag_upd
 AFTER UPDATE OF raktarkeszlet ON alapanyag
 FOR EACH ROW
BEGIN
 DBMS_OUTPUT.PUT_LINE('A raktárkészlet módosult!');
END;
/
```

Újra próbáljuk ki az előző UPDATE parancsot.

Több esemény figyelése:
Példa: Minden beszúrásról és a raktarkeszlet mező módosításáról kapjunk figyelmeztető üzenetet.

11.3. Több esemény figyelése

Minden beszúrásról és a raktarkeszlet mező módosításáról kapjunk figyelmeztető üzenetet.

```
CREATE OR REPLACE TRIGGER anyag_ins_upd
BEFORE INSERT OR UPDATE OF raktarkeszlet ON alapanyag
BEGIN
 IF INSERTING THEN
  DBMS_OUTPUT.PUT_LINE('Új sor felvétele történt!');
 ELSE
  DBMS_OUTPUT.PUT_LINE('A raktárkészlet mező módosult!');
 END IF;
END;
/
```

11.4. Példa old és new használatára

Készítsünk egy ALAPANYAG_BACKUP nevű táblát, amelybe felvesszük az ALAPANYAG tábla azon sorainak a régi értékét, amelyeknél a raktárkészlet több mint 100-zal megnőtt.

```
create table alapanyag_backup (nev varchar2(20) primary key, mennyegys varchar2(2),
raktarkeszlet number(5,2));

CREATE OR REPLACE TRIGGER szaz
BEFORE UPDATE ON alapanyag
FOR EACH ROW
WHEN (new.raktarkeszlet>old.raktarkeszlet+100)
BEGIN
 delete from alapanyag_backup where nev=:old.nev;
 insert into alapanyag_backup values(:old.nev, :old.mennyegys, :old.raktarkeszlet);
END;
/
```

11.5. Példa INSTEAD OF triggerre

Hozzunk létre egy nézetet az alapanyag tábla teljes tartalmáról, majd kapcsoljunk, hozzá egy triggert, amely letiltja a rekordok törlését a nézeten keresztül.

```
CREATE VIEW anyag_nezet AS SELECT * FROM alapanyag;

CREATE OR REPLACE TRIGGER nincs_torles
INSTEAD OF DELETE ON anyag_nezet
```

```
FOR EACH ROW
BEGIN
 DBMS_OUTPUT.PUT_LINE('Nem lehet törölni!');
END;
/
```

A trigger csak sorszintű lehet (INSTEAD OF miatt), ha elhagyjuk a FOR EACH ROW sort, akkor is sorszintű lesz.

11.6. Feladatok

1. Hozzunk létre egy olyan triggert, amelyet a TERMELO táblával kapcsolatos változtatások (INSERT, UPDATE, DELETE) indítanak el. A trigger küldjön egy üzenetet arról a képernyőre, hogy milyen esemény váltotta ki a működését. Ha a változás több sort is érint, az üzenet akkor is csak egyszer jelenjen meg.

2. Készítsünk egy olyan triggert, amit az ETEL táblából való törlés aktivál. Ha törölni akarunk egy rekordot az ETEL táblából, akkor előbb a KESZUL táblából kell a rá mutató rekordokat törölni. A trigger feladata legyen ezek törlése, mielőtt az ETEL kívánt rekordja törlődne.

3. Az alapanyag táblához kapcsoljunk egy olyan triggert, amely a név mező módosítására indul el. A név mezőt addig nem lehet módosítani, amíg mutat rá rekord a KESZUL és a SZALLIT táblából. Ezért a módosítás előtt módosítsuk a trigger segítségével az említett két táblában az anyagnev mezőt a kívánt értékre.

12. Tárolt eljárások és függvények készítése

Az eljárások és függvények névvel ellátott PL/SQL blokkok. Az adatbázisban kerülnek tárolásra, bármikor meghívhatók.

12.1. Eljárás létrehozása

```
CREATE OR REPLACE PROCEDURE eljárásnév
                              [(paraméter IN/OUT/IN OUT adattípus,
                              paraméter 2 IN/OUT/IN OUT adattípus, ...)]
AS
 deklarációk
BEGIN
 utasítások
[EXCEPTION
 kivételkezelés]
END;
/
```

Értékeket a hívó környezetből és vissza paramétereken keresztül vihetünk át.

- Az IN típusú paraméterrel a hívó környezettől vesz át értéket az eljárás.
- Az OUT típusúval a hívó környezetnek ad vissza értéket az eljárás.
- Az IN OUT típusút oda-vissza értékadásra lehet használni.

12.1.1. Bevezető példa eljárás készítésére

Készítsünk egy eljárást, amely híváskor átvesz egy karaktersorozatot, majd ezt kiírja a képernyőre.

```
create or replace procedure kiir(szoveg in varchar2)
 as
begin
 dbms_output.put_line(szoveg);
end;
/
```

Ne feledjük az eljárás futatása előtt kiadni a **set serveroutput on;** parancsot!

Eljárás futtatása:

```
execute kiir('Hello világ!')
```

Az átadott szöveg megjelent a képernyőn.

12.1.2. Példa IN típusú paraméter használatára

Hozzunk létre egy táblát a következők szerint:

```
create table korterulet(sugar number(5), terulet number(14,2));
```

Majd készítsünk egy eljárást, amely a paraméterként átvett sugárérték alapján kiszámítja a kör területét és beírja a hozzá tartozó sugárral együtt a táblába.

```
create or replace procedure szamit(sugar in number)
as
 pi constant number(8,7):=3.1415926;
 terulet number(14,2);
begin
 terulet:=pi*sugar**2;
 insert into korterulet values(sugar, terulet);
end;
/
```

Ezután hívjuk meg az eljárást néhányszor átadva neki a sugár értékét, majd nézzük meg a tábla tartalmát.

Példa: Készítsünk egy másik eljárást, amely az átadott sugárértéket megkeresi a táblában és törli a mellette szereplő területértéket. Ha nem talál ilyen rekordot, akkor ezt jelezze egy képernyőre küldött üzenettel.

```
create or replace procedure torol(sug in number)
as
sugarvalt number(5);
begin
 select sugar into sugarvalt from korterulet where sugar=sug;
 update korterulet set terulet=0 where sugar=sugarvalt;
exception
 when no_data_found then
 dbms_output.put_line('Nincs ilyen sugárérték a táblában!');
end;
/
```

12.1.3. Példa OUT típusú paraméter használatára

Alakítsuk át a **szamit** eljárást a következők szerint:

```
create or replace procedure szamit(sugar in number, terulet out number)
as
 pi constant number(8,7):=3.1415926;
begin
 terulet:=pi*sugar**2;
end;
/
```

Az eljárásnak ezzel adtunk egy OUT paramétert, amely segítségével adatot fog visszaadni a hívó környezetnek.

Hozzuk létre az **alap** eljárást, amely alapértékekkel feltölti a táblát:

```
create or replace procedure alap
as
ter number(14,2);
begin
 delete from korterulet;
```

```
 for i in 1..10 loop
  szamit(i,ter);
  insert into korterulet values(i,ter);
 end loop;
 commit;
end;
/
```

Ez az eljárás amint látható nem tartalmaz paramétereket. Egy másik eljárás (a **szamit**) meghívására is láthatunk példát.

12.2. Függvény létrehozása

```
CREATE OR REPLACE FUNCTION függvénynév
                                    [(argumentum IN/OUT/IN OUT adattípus,
                                    argumentum2 IN/OUT/IN OUT adattípus, ...)]
    RETURN adattípus
    AS
     deklarációk
    BEGIN
     utasítások
     RETURN érték;
    [EXCEPTION
     kivételkezelés]
    END;
    /
```

Függvényeknél is megadható OUT illetve IN OUT típusú argumentum is, de ezeket itt lehetőleg kerüljük.

Példa függvény készítésére és meghívására

Készítsünk függvényt, amely kiszámítja egy kör területét a híváskor átadott sugár alapján.

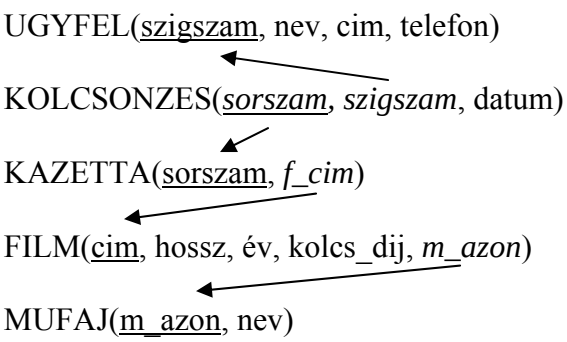
```
    create or replace function szamol (sugar in number)
    return number
    as
     pi constant number(8,7):=3.1415926;
     ter number(14,2);
    begin
     ter:=pi*sugar**2;
     return(ter);
    end;
    /
```

A függvényeket kifejezés részeként hívjuk meg, hiszen gyakorlatilag egy értéket képviselnek:

```
    insert into korterulet values (20,szamol(20));
```

13. Gyakorló feladatok

A feladatok egy videotéka nyilvántartási rendszerével kapcsolatosak. Az adatbázis relációs modellje a következő:



A mezők típusa:

UGYFEL tábla
szigszam: char(10)
nev: varchar2(30)
cim: varchar2(50)
telefon: varchar2(10)

KOLCSONZES tábla
sorszam: number
szigszam: char(10)
datum: date

KAZETTA tábla
sorszam: number
f_cim: varchar2(30)

FILM tábla
cim: varchar2(30)
hossz: number(3)
év: number(4)
kolcs_dij: number(4)
m_azon: char(1)

MUFAJ tábla
m_azon: char(1)
nev: varchar2(20)

Hozzuk létre a táblákat és töltsük fel tesztadatokkal a **videoteka.sql** nevű szkript futtatásával. Egy szekvenciára is szükség lesz, mert ez alapján sorszámozzuk a kazettákat (A szkript ezt is létrehozza).

1. feladat

Írjunk egy eljárást, amely segítségével új ügyfelek vihetők fel az **ugyfel** táblába. Az eljárás ellenőrizze felvitel előtt, hogy a megadott adatok hossza belefér-e abba a hosszba, amit a mező típusánál megadtunk. Hosszabb adat esetén egy saját magunk által definiált kivételt idézzünk elő. A kivétel kezelésénél írassuk ki a képernyőre a probléma okát.

Megoldás:

```
create or replace procedure p_ugyfel
 (sz in char,
 n in varchar2,
 c in varchar2,
 t in varchar2)
as
 hiba exception;
begin
 if length(sz)>10 or
    length(n)>30 or
    length(c)>50 or
    length(t)>10
  then raise hiba;
 end if;
 insert into ugyfel values(sz,n,c,t);
 commit;
exception
 when hiba then
  dbms_output.put_line('Túl hosszú adatot adott meg!');
end;
/
```

2. feladat

Írjunk egy eljárást, amely az új filmek felvitelét végzi el. Most az egyszerűség kedvéért tekintsünk el mindenféle értékvizsgálattól. Az eljárás kapja meg paraméterként a **film** tábla mezőinek megfelelő értékeket, illetve egy számot, amely a kívánt példányszámot jelenti. A **film** táblában hozza létre az új rekordot. A **kazetta** táblában pedig hozzon létre annyi rekordot, ahány példányt akarunk az új filmről tárolni. A **kazetta** tábla elsődleges kulcsát a **szekv** nevű szekvencia segítségével hozzuk létre.

Megoldás:

```
create or replace procedure p_film
(cime in varchar2,
 hossza in number,
 eve in number,
 kolcs_dija in number,
 mufajazon in char,
 peldanyszam in number)
as
begin
 insert into film values(cime, hossza, eve, kolcs_dija, mufajazon);
 for i in 1..peldanyszam loop
  insert into kazetta values(szekv.nextval,cime);
 end loop;
commit;
end;
/
```

3. feladat

Készítsünk eljárást, amely a kazetták kölcsönzésével kapcsolatos feladatokat végzi el. Paraméterül kapja meg a film címét és a kölcsönző ügyfél igazolványszámát. Ha nem találja az **ugyfel** táblában a megadott igazolványszámot, vagy a filmcímet a **film** táblában, akkor azt hibaüzenettel jelezze. Ellenőrizze, hogy a kért film bent van-e. Ha minden kazettát kikölcsönöztek, amely ezt a filmet tartalmazza, akkor erről küldjön üzenetet. Ha pedig egyébként minden feltétele adott a kölcsönzésnek, akkor készüljön el a megfelelő bejegyzés a **kolcsonzes** táblába a mai dátummal.

Megoldás:

```
create or replace procedure p_kolcsonzes
(filmcime in varchar2, szszam in char)
as
 filmhiany exception;
 talalt boolean;
 kaziszam number;
 szsz char(10);
 kaz_puffer number;
 cursor kaz_kurzor is select sorszam from kazetta where f_cim=filmcime;
 cursor kolcs_kurzor is select sorszam from kolcsonzes;
begin
 select szigszam into szsz from ugyfel where szigszam=szszam;
 open kaz_kurzor;
 fetch kaz_kurzor into kaz_puffer;
 if kaz_kurzor%notfound then raise filmhiany;
 end if;
  loop
   talalt:=false;
    for kolcs_rekord in kolcs_kurzor loop
     if kolcs_rekord.sorszam=kaz_puffer then talalt:=true;
     end if;
    end loop;
   kaziszam:=kaz_puffer;
   fetch kaz_kurzor into kaz_puffer;
   exit when kaz_kurzor%notfound or talalt=false;
  end loop;
 close kaz_kurzor;
 if talalt=true then
  dbms_output.put_line('Sajnos a kért film ki van kölcsönözve.');
 else
  insert into kolcsonzes values(kaziszam,szszam,sysdate);
  commit;
 end if;
exception
 when no_data_found then
  dbms_output.put_line('Nincs ilyen kódú ügyfél az adatbázisban!');
 when filmhiany then
  dbms_output.put_line('Nincs ilyen című filmünk az adatbázisban!');
end;
/
```

4. feladat
Készítsünk egy eljárást, amely régi, lejárt filmek törlésének feladatát látja el. Bemenő paraméterként vegye át a törlendő film címét. A film törlése előtt törölni kell a kazettákat, amelyek ezt a filmet tartalmazzák. Viszont ha valamelyik kazetta ki van kölcsönözve, akkor előbb a kölcsönzés táblából kell törölni, utána törölhetjük csak a kazetta táblából.

5. feladat
Készítsünk egy függvényt, amely a filmek kölcsönzési díját számítja ki. Kaland, akció, scifi, thriller és horror esetén szorozzuk meg 15-el a film hosszát, majd ebből adjunk annyi % árengedményt, ahány éves a film. Az aktuális dátum év részét numerikus formában az *extract (year from current_date)*
paranccsal kapjuk meg.
A többi műfajnál, a kölcsönzési díj a film hosszának 10-szerese legyen, árengedményt ne adjunk.
A függvény működését egy új film felvételével ellenőrizhetjük.