

## 5.1 Procediments emmagatzemats en PL/pgSQL

- PL/pgSQL
- Procediments emmagatzemats en PL/pgSQL
- Paràmetres
- Variables
- Sentències condicionals
- Sentències iteratives
- Cursors Explícits
- Gestió d'Errors



## PL/pgSQL

PL/pgSQL és un dels diferents llenguatges que ofereix PostgreSQL per implementar procediments emmagatzemats.

El PL/pgSQL que s'explica en aquestes transparències és un subconjunt del PL/pgSQL corresponent a la versió de PostgreSQL instal·lada al laboratori de la FIB.



## Base de dades exemple

```
create table clients(  
  dni varchar(9) primary key,  
  nom varchar(15) not null,  
  cognom1 varchar(15) not null,  
  cognom2 varchar(15) not null,  
  carrer varchar(20) not null,  
  num_carrer varchar(4) not null,  
  cp char(5) not null,  
  ciutat varchar(15) not null,  
  qtt_com integer  
);
```

```
create table comandes (  
  num_com integer primary key,  
  dni varchar(9) not null references clients,  
  data_arribada date not null,  
  import_total integer );
```

```
create table items(  
  num_item integer primary key,  
  preu_unitat integer not null);
```

```
create table items_comanda(  
  num_item integer references items,  
  num_com integer references comandes,  
  quantitat integer not null,  
  primary key(num_item,num_com));
```

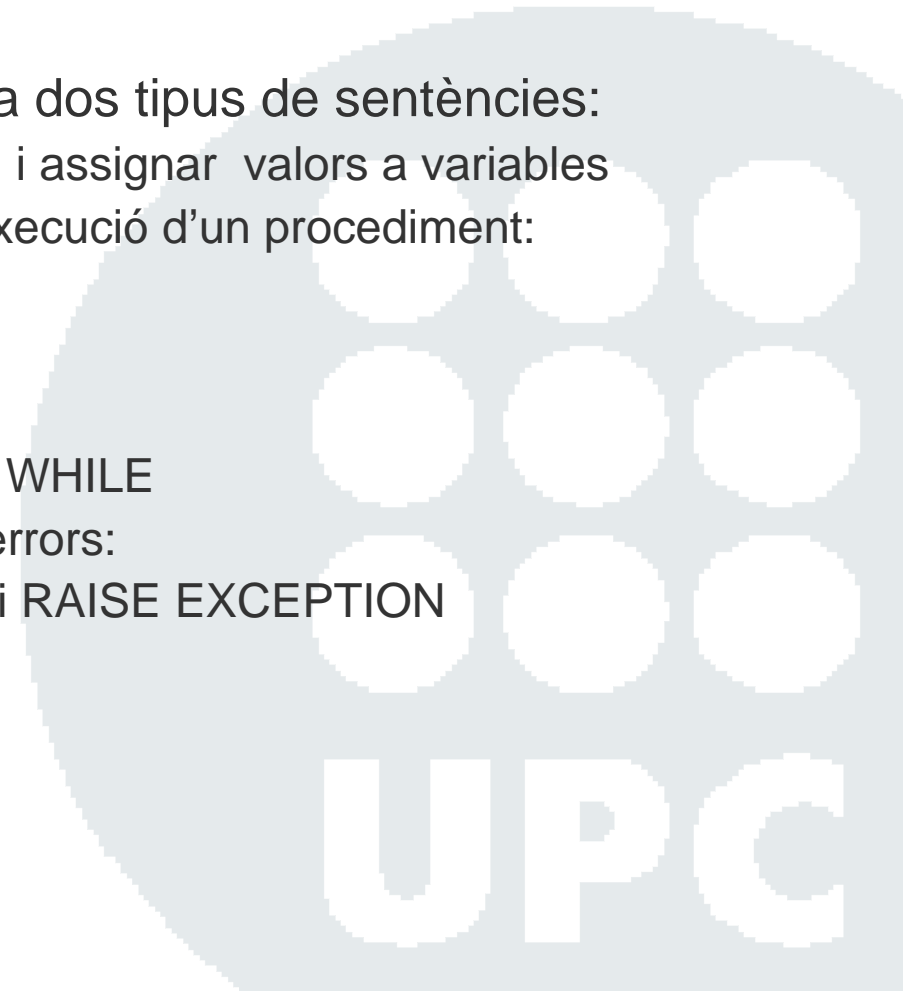
Quantitat de  
comandes del  
client

Import  
de la comanda

Quantitat de l'item  
num\_item que s'ha  
comprat en la  
comanda num\_com.

## Procediments emmagatzemats en PL/pgSQL

- Les sentències de PL/pgSQL les utilitzarem dins del cos d'un procediment, és a dir, entre les sentències CREATE FUNCTION i END de la funció.
- Bàsicament, PL/pgSQL proporciona dos tipus de sentències:
  - Sentències per definir (DECLARE) i assignar valors a variables
  - Sentències per controlar el flux d'execució d'un procediment:
    - Sentències condicionals:
      - Sentència IF
    - Sentències iteratives:
      - Sentències LOOP, FOR i WHILE
    - Sentències per fer la gestió d'errors:
      - Sentències EXCEPTION i RAISE EXCEPTION



## Paràmetres: Retorn d'una única tupla

```
CREATE FUNCTION nom_proc (param_entrada tipus) RETURNS tipus_retorn AS $$  
.....  
    RETURN variable_retorn;  
END;  
$$LANGUAGE plpgsql;
```

Nom de la variable que té el valor que retorna la funció

Paràmetre d'entrada:  
nom i tipus

Tipus de retorn de la funció

En cas de no retornar res es posa "void"

UPC

## Paràmetres: Retorn d'una única tupla - Exemple

Aquest procediment obté la ciutat on viu el client amb el DNI que es passa com a paràmetre d'entrada.

```
CREATE FUNCTION trobar_ciutat(dni_client varchar(9))
RETURNS varchar(15) AS $$
DECLARE
    ciutat_client varchar(15);
BEGIN
    SELECT ciutat INTO ciutat_client
    FROM clients
    WHERE dni=dni_client;

    RETURN ciutat_client;
END;
$$LANGUAGE plpgsql;
```

DNI del client

Tipus que  
tindrà la  
variable de retorn

Variable de  
retorn de la  
ciutat on viu  
el client

## Paràmetres: Retorn d'un conjunt de tuples

```
CREATE FUNCTION nom_proc(param_entrada tipus) RETURNS SETOF tipus AS $$  
    .....  
    RETURN NEXT ciutat_client;  
    .....  
END;  
$$LANGUAGE plpgsql;
```

- Cal usar la clàusula **RETURN NEXT**. Aquesta clàusula no acaba el procediment, sinó que va retornant a cada execució els valors de la variable. El procediment acaba quan s'executa un RETURN sense NEXT, o quan s'arriba al final.

- Per retornar un conjunt de tuples cal utilitzar SETOF quan especifiquem el tipus que retorna la funció.

## Paràmetres: Retorn d'un conjunt de tuples - Exemple

Aquest procediment retorna una tupla per cada enter que hi ha entre 0 i el valor del paràmetre d'entrada MAX.

```
CREATE FUNCTION exemple_retorn_n_tuples(max integer)
RETURNS SETOF integer AS $$
DECLARE
    i integer := 0;
BEGIN
    LOOP
        i:=i+1;
        RETURN NEXT i;
        EXIT WHEN i = max;
    END LOOP;
    RETURN;
END;
$$LANGUAGE plpgsql;
```

→ SETOF del tipus  
que tindrà la  
variable de retorn

→ RETURN NEXT  
que s'invoca  
tantes vegades  
com tuples es  
vol retornar



## Variables: Declaració

- El valor d'una variable s'emmagatzema en memòria volàtil i per tant, no són considerades objectes de la BD
- Totes les variables definides dins d'un procediment són variables locals.
- L'àmbit de visibilitat d'una variable local queda restringit al procediment a on s'hagi definit
- Sintaxis:

```
Nom_variable [CONSTANT] type [NOT NULL] [{DEFAULT | :=}expression];
```

```
DECLARE  
  nom_client char(15);  
  carrer varchar(20) not null;  
  edat integer default 18;  
  num constant integer default 0;  
  
  dni_client clients.dni%TYPE;
```

Podem utilitzar els mateixos tipus de dades que els utilitzats a les columnes d'una taula.

És possible especificar que el tipus de dades d'una variable és idèntic al tipus de dades d'una determinada columna d'una taula mitjançant la clàusula TYPE.

- Si no s'inicialitzen les variables, per defecte prenen valor NULL

## Variables: Utilització

- Bàsicament, és possible utilitzar variables dins d'un procediment emmagatzemat en les situacions següents:
  - En sentències SQL

```
CREATE FUNCTION....  
DECLARE  
    dni_client clients.dni%TYPE;  
    ciutat_client varchar(15);  
BEGIN  
    ....  
    SELECT ciutat INTO ciutat_client  
    FROM clients  
    WHERE dni=dni_client;  
    ...  
END;
```

- En sentències de PL/PGSQL per
  - Assignar-hi valors
  - Calcular valors
  - Controlar el flux d'execució d'un procediment

## Variables: Creació de nous tipus

En alguns casos, com per exemple quan un procediment ha de retornar tuples amb un conjunt d'atributs ens cal definir un nou tipus.

```
CREATE TYPE tipusAdressa AS (  
    carrer varchar(20),  
    num_carrer varchar(4),  
    ciutat varchar (15));
```

Creació prèvia  
al procediment

```
CREATE FUNCTION exNoustipus()  
RETURNS tipusAdressa AS $$  
DECLARE
```

```
    adreassa tipusAdressa;  
    carrer varchar(20);
```

Declaració d'una  
variable del tipus

```
BEGIN
```

```
    ....  
    adreassa.ciutat:='Badalona';  
    ....  
    carrer := adreassa.carrer;  
    ....
```

Utilització i accés  
dels diferents valors  
de la variable

```
RETURN adreassa;  
END;  
$$ LANGUAGE plpgsql;
```

## Variables: Creació de nous tipus - Exemple

Obtenir l'adreça (concretament el carrer, num\_carrer i ciutat) d'un client

```
CREATE TYPE TAdressa AS (  
    carrer varchar(20),  
    num_carrer varchar(4),  
    ciutat varchar(15)  
);  
  
CREATE FUNCTION trobar_adressa_client (dni_client clients.dni%type)  
RETURNS TAdressa AS $$  
DECLARE  
    dadesCli TAdressa;  
BEGIN  
    SELECT carrer,num_carrer,ciutat INTO dadesCli  
    FROM clients  
    WHERE dni=dni_client;  
  
    RETURN dadesCli;  
END;  
$$LANGUAGE plpgsql;  
  
select * FROM trobar_adressa_client('45678900');
```

## Variables: Assignacions de valors

Només té sentit per sentències o procediments que només retornen una fila:

- Sentència d'assignació de PL/PGSQL

```
numComclient := (SELECT num_com  
                    FROM clients  
                    WHERE dni=dni_client);
```

- Sentència SELECT ... INTO de l'SQL

```
SELECT ciutat INTO ciutat_client  
FROM clients  
WHERE dni=dni_client;
```

- Assignar a una variable el que retorna un procediment:

```
imp_comada := import_una_com(numero_com);
```

o bé

```
select * from import_una_com(numero_com) into imp_comanda;
```

## Sentències condicionals

- La sentència IF serveix per a establir condicions en el flux d'execució d'un procediment:

```
IF condició THEN bloc de sentències  
ELSE bloc de sentències  
END IF;
```

- Podem establir diferents nivells d'aniuament mitjançant la clàusula

```
IF ... THEN ... ELSEIF ... THEN ... ELSE...END IF;
```

- Condicions:

- Per especificar les condicions podem utilitzar:
  - Operadors lògics: AND, OR, NOT
  - Operadors de comparació: =, <, <=, >, >=, etc
  - Predicats propis d'SQL: BETWEEN, IN, IS NULL, EXISTS
  - Variable PL/pgSQL: FOUND
  - Consultes SQL

## Sentències condicionals - Exemple

Obté el descompte del client amb el DNI que es passa per paràmetre.  
Aquest descompte depèn del nombre de comandes del client.

```
CREATE FUNCTION calcul_desc_client(dni_client clients.dni%type)
RETURNS integer AS $$
DECLARE
    descompte INTEGER;
    qttComClient INTEGER;
BEGIN
    IF (EXISTS (SELECT * FROM clients WHERE dni=dni_client)) THEN
        qttComclient:=(SELECT qtt_com FROM clients WHERE dni=dni_client);
        IF (qttComclient=0) THEN descompte:=0;
        ELSIF (qttComClient<5) THEN descompte:=1;
        ELSIF (qttComClient<10) THEN descompte:=3;
        ELSIF (qttComClient<15) THEN descompte:=5;
        ELSE descompte:=10;
        END IF;
    END IF;
    RETURN descompte;
END;
$$LANGUAGE plpgsql;
```

Execució de la funció:

<pre>select * from calcul_desc_client('35678111') as descompte;</pre>	
<div> <div>Panel de Salida</div> <div> <div>Salida de datos</div> <div>Comentar</div> <div>Mensajes</div> <div>Historial</div> </div> </div>	
	descompte integer
1	3

## Sentències condicionals – Variable FOUND

La variable FOUND és de tipus booleà.

FOUND té en principi el valor False.

El seu valor pot canviar quan s'executen les sentències següents:

- Una sentència SELECT ... INTO posa FOUND a True si el select obté una fila, i a False si no s'obté cap fila.
- Una sentència UPDATE, INSERT o DELETE posa FOUND a True si com a mínim una fila es veu afectada per la sentència, i a False si no queda afectada cap fila.
- Una sentència FOR. Dintre de cada iteració del FOR, el valor de la variable pot canviar segons les sentències que s'hi executen. Però en sortir del FOR és posa FOUND a True si s'ha iterat una o més vegades, sinó es posa a False.

FOUND és una variable local en un procediment. Qualsevol canvi en aquesta variable afecta només al procediment on aquest canvi es produeix.

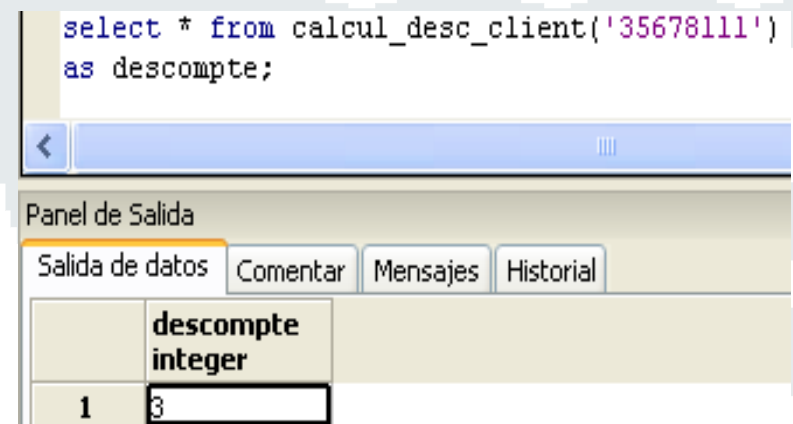


## Sentències condicionals – Variable FOUND - Exemple

Obté el descompte del client amb el DNI que es passa per paràmetre.  
Aquest descompte depèn del nombre de comandes del client.

```
CREATE FUNCTION calcul_desc_client(dni_client clients.dni%type)
RETURNS integer AS $$
DECLARE
    descompte INTEGER;
    qttComClient INTEGER;
BEGIN
    SELECT qtt_com into qttComclient FROM clients WHERE dni=dni_client;
    IF FOUND THEN
        IF (qttComclient=0) THEN descompte:=0;
        ELSIF (qttComClient<5) THEN descompte:=1;
        ELSIF (qttComClient<10) THEN descompte:=3;
        ELSIF (qttComClient<15) THEN descompte:=5;
        ELSE descompte:=10;
    END IF;
    END IF;
    RETURN descompte;
END;
$$LANGUAGE plpgsql;
```

Execució de la funció:



Panel de Salida	
Salida de datos	
	descompte integer
1	3

## Sentències iteratives FOR, WHILE i LOOP

### ■ Sentència **FOR**

- S'utilitza habitualment per iterar sobre el conjunt de tuples retornades per una consulta SQL.

```
FOR target IN query
LOOP statements END LOOP;
```

- Es pot utilitzar també quan sabem a priori el nombre d'iteracions a executar.

```
FOR name IN [ REVERSE ] expression .. expression
LOOP statements END LOOP;
```

### ■ Sentències **WHILE** i **LOOP**.

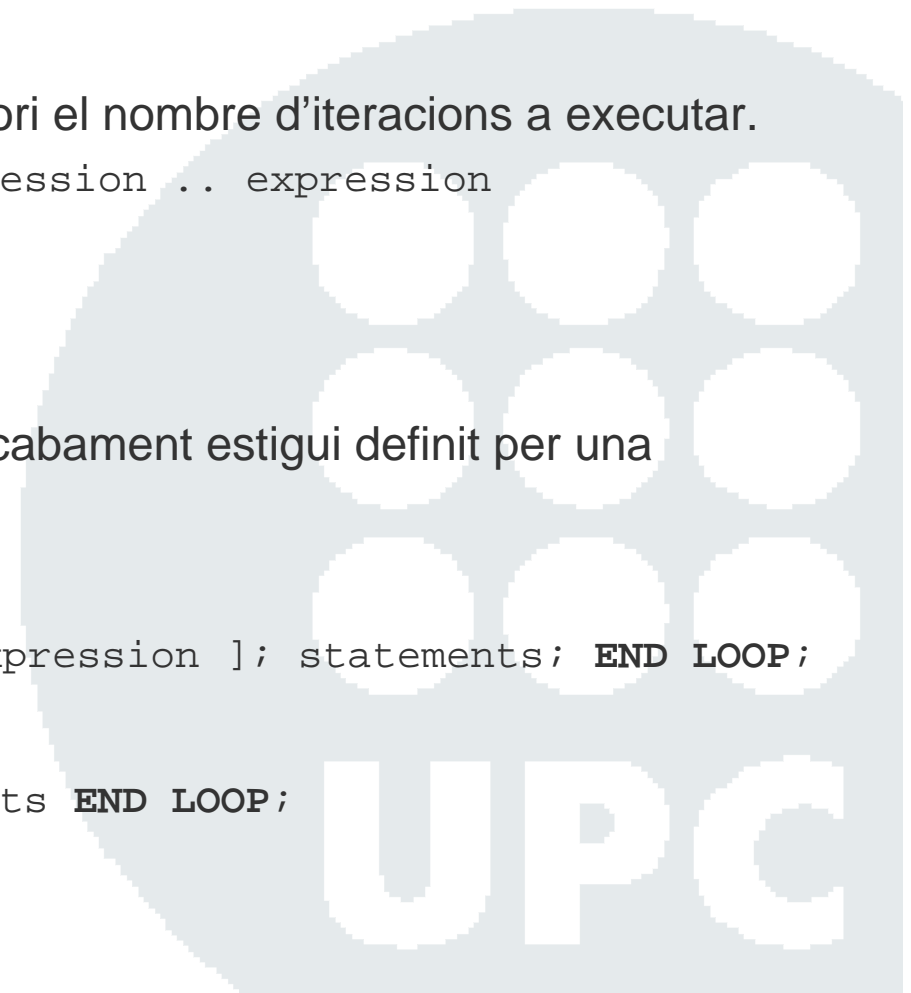
- S'utilitzen per definir bucles on el seu acabament estigui definit per una expressió condicional.

- Sentència **LOOP** :

```
LOOP statements EXIT [ WHEN expression ]; statements; END LOOP;
```

- Sentència **WHILE**:

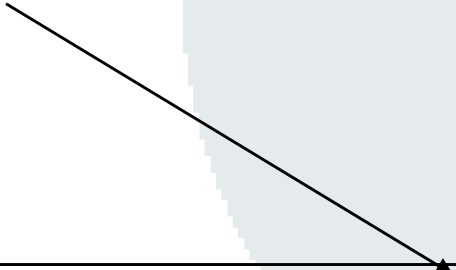
```
WHILE expression LOOP statements END LOOP;
```



## Sentències iteratives: Exemple utilització FOR (1)

Obté les dades de tots els clients d'una determinada ciutat. Indica amb una "P" els clients que són preferents (import total de comandes superior a 60000).

```
CREATE TYPE Tdades_client_tip AS (  
    dni_client VARCHAR(9),  
    nom_client VARCHAR(15),  
    cognom1 VARCHAR(15),  
    pref CHAR(1));  
  
CREATE FUNCTION clients_ciutat_tip(ciutat_client clients.ciutat%type)  
RETURNS SETOF Tdades_client_tip AS $$  
DECLARE  dades_clients Tdades_client_tip;  
BEGIN  
FOR dades_clients IN SELECT dni,nom,cognom1  
                        FROM clients  
                        WHERE ciutat=ciutat_client LOOP  
    dades_clients.pref := es_preferent(dades_clients.dni_client);  
    return next dades_clients;  
END LOOP;  
RETURN;  
END;  
$$LANGUAGE plpgsql;
```



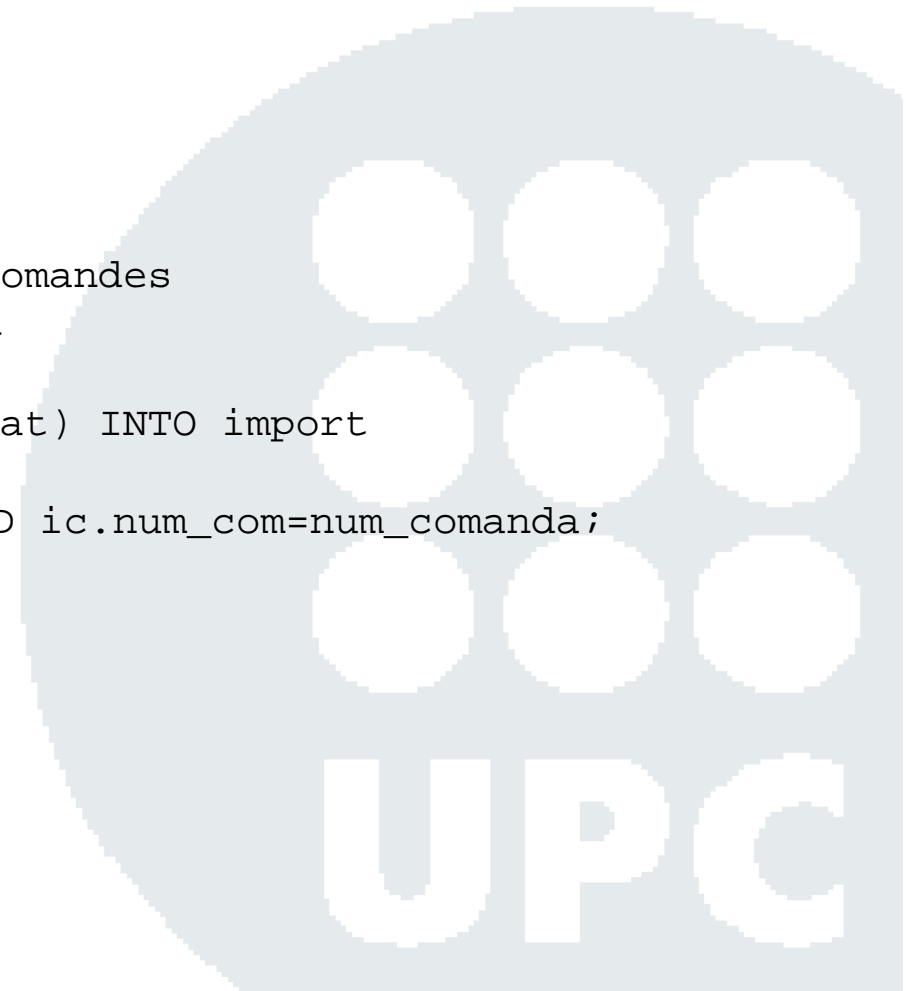
Procediment que comprova  
si un client és preferent

## Sentències iteratives: Exemple utilització FOR (2)

Modifica l'import de la comanda a la taula comandes per cadascuna de les comandes d'un determinat client.

```
CREATE FUNCTION import_totes_com(dni_client clients.dni%type)
RETURNS void AS $$
DECLARE
    num_comanda comandes.num_com%type;
    import comandes.import_total%type;
BEGIN
    FOR num_comanda IN SELECT num_com FROM comandes
                        WHERE dni=dni_client
        LOOP
            SELECT SUM(ic.quantitat*i.preu_unitat) INTO import
                FROM items_comanda ic, items i
                WHERE i.num_item=ic.num_item AND ic.num_com=num_comanda;
            UPDATE comandes
                SET import_total=import
                WHERE num_com=num_comanda;
        END LOOP;

    END;
$$LANGUAGE plpgsql;
```



## Sentències iteratives: Exemple utilització FOR (2) - Execució

<pre>select * from clients;</pre>					<pre>select * from comandes;</pre>				
Panel de Salida					Panel de Salida				
Salida de datos Comentar Mensajes Historial					Salida de datos Comentar Mensajes Historial				
	dni character var	nom character var	cognom1 character var	cognom2 character var		num_com integer	dni character var	data_arribad. date	import_total integer
1	45678900	Maria	Puig	March	1	1	45678900	2008-01-30	
2	54670900	Pere	Gomez	Perez	2	3	45678900	2008-01-30	
3	35678111	Josep	Sorroca	Camps	3	2	54670900	2008-01-30	
4	35678112	Josep	Sorroca	Camps					

Si s'executa el procediment anterior per al client '54670999' el resultat és:

<pre>select import_totes_com('54670900'); select * from comandes;</pre>				
Panel de Salida				
Salida de datos Comentar Mensajes Historial				
	num_com integer	dni character var	data_arribad. date	import_total integer
1	1	45678900	2008-01-30	
2	3	45678900	2008-01-30	
3	2	54670900	2008-01-30	240

## Sentències iteratives: Exemple utilització WHILE

```
CREATE FUNCTION incrementar_preu() RETURNS void as $$
BEGIN
  WHILE EXISTS(SELECT * FROM items WHERE preu_unitat < 25) LOOP
    UPDATE items
    SET preu_unitat = preu_unitat + 5
    WHERE preu_unitat < 25;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Contingut de la taula Items abans  
i després d'executar el procediment.

	num_item integer	preu_unitat integer
1	1	15
2	2	20
3	3	15
4	4	20
5	5	15
6	6	50
7	7	40

	num_item integer	preu_unitat integer
1	1	25
2	2	25
3	3	25
4	4	25
5	5	25
6	6	50
7	7	40

Que hauria passat si en comptes  
d'utilitzar un WHILE  
haguéssim utilitzat un FOR ?

## Cursors explícits (Alternativa que NO utilitzarem a laboratori)

- Una altra manera d'accedir a un conjunt de tuples és via cursors explícits. Hi ha altres SGBDs que només tenen aquesta opció. L'estàndard admet les dues.
  - Un cursor és una estructura de dades que permet accedir a cada una de les files que retorna una consulta SQL.
  - Addicionalment PostgreSQL permet treballar amb cursors explícits per a accedir a cada una de les files retornades per una consulta SQL (**nosaltres no ho farem servir a laboratori**).
  - Per treballar amb cursors explícits cal:
    1. Declarar el cursor i associar-lo a una consulta (DECLARE ... CURSOR FOR)
    2. Obrir el cursor (OPEN)
    3. Obtenir la primera fila de la consulta associada al cursor (FETCH)
    4. Executar el bloc de sentències del bucle
    5. Obtenir la següent fila de la consulta associada al cursor (FETCH)
    6. Tornar a executar els passos 4) i 5) mentre hi hagin files que formin part del resultat de la consulta.
    7. Tancar el cursor (CLOSE)

## Cursors explícits (Alternativa que NO utilitzarem a laboratori)

```
CREATE FUNCTION clients_ciutat(ciutat_client clients.ciutat%type) RETURNS SETOF varchar(9) AS $$  
DECLARE  
    cursor_clients CURSOR FOR SELECT dni  
                                FROM clients  
                                WHERE ciutat=ciutat_client;  
    dni_client clients.dni%type;  
BEGIN  
OPEN cursor_clients;  
LOOP  
    FETCH cursor_clients INTO dni_client;  
    EXIT WHEN NOT FOUND;  
    return next dni_client;  
END LOOP;  
close cursor_clients;  
END;  
$$LANGUAGE plpgsql;  
  
select * FROM clients_ciutat('Barcelona');
```

Panel de Salida

Salida de datos

Comentar

Mensajes

Historial

	clients_ciutat character var
1	45678900
2	54670900



## Gestió d'errors

Quan es produeix un error dintre d'un procediment podem:

- **No capturar-lo:** El procediment falla i es retorna l'error concret al nivell superior (JDBC, editor d'SQL, a un altre procediment..)
- **Capturar-lo:**
  - **Opció 1.** El procediment falla i es retorna una **excepció** determinada pel programador al nivell superior.
  - **Opció 2.** El procediment té èxit i es retorna un **codi d'error** determinat pel programador al nivell superior.
  - **Opció 3.** El procediment té èxit i s'insereix l'error en un **taula d'errors**.
  - **Opció 4.** El procediment té èxit i l'error es tractat dins del procediment.

## Gestió d'errors

### Tipus d'errors que es poden produir:

- Errors predefinitos pel propi SGBD, p.e. el codi d'error número 23505 a PostgreSQL vol dir “Unique violation”. Es produeixen quan alguna instrucció que s'executa provoca alguna excepció pròpia del SGBD.
- Errors d'usuari. Específics del procediment (P0001 a PostgreSQL). Codi d'error quan dins d'un procediment s'executa una instrucció `RAISE EXCEPTION`.

### Instruccions per gestionar els errors:

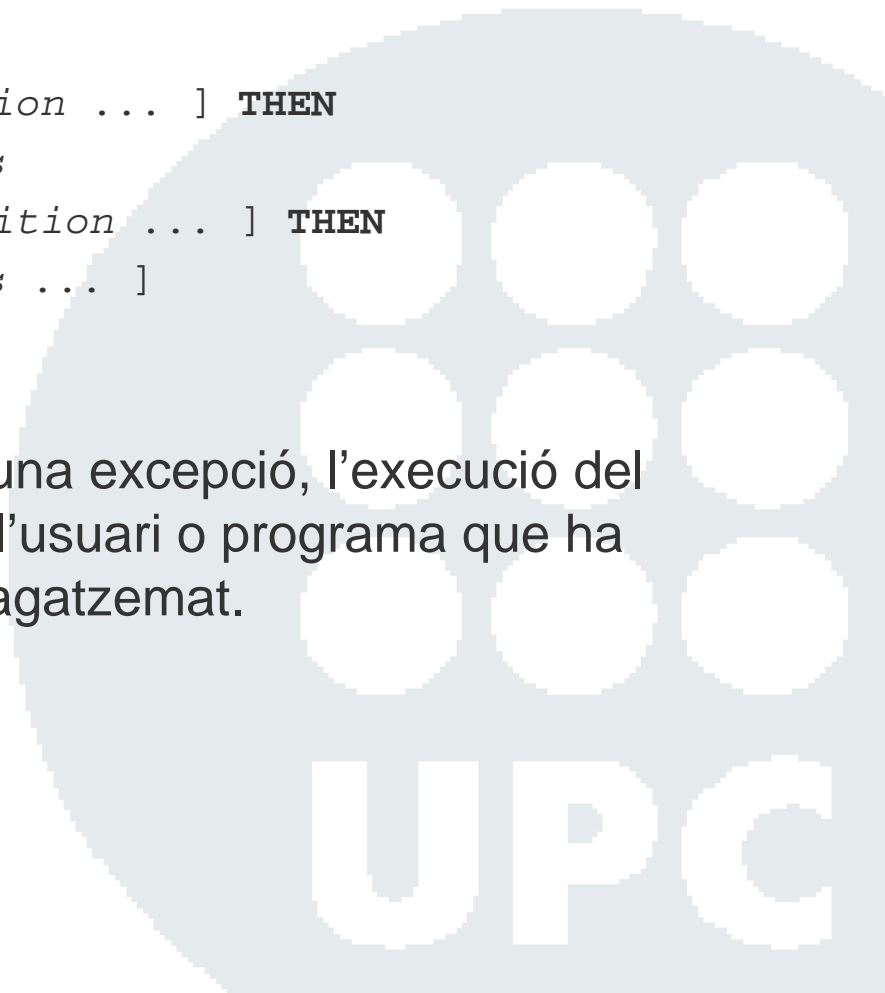
- **EXCEPTION:** Accions a dur a terme en cas de que es produeixin excepcions.
- **RAISE EXCEPTION:** Serveix per a que el programador pugui generar els seus propis errors dins d'un procediment

## Gestió d'errors

- Es pot utilitzar el bloc BEGIN... amb la clàusula EXCEPTION per tractar els errors que es produeixen dins d'un procediment.

```
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
        handler_statements ... ]
END;
```

- Si dins del bloc EXCEPTION es produeix una excepció, l'execució del procediment finalitza i l'error es reportat a l'usuari o programa que ha demanat l'execució del procediment emmagatzemat.



## Gestió d'errors: Opció 1 - Captura i retorn d'excepcions

```
CREATE FUNCTION nova_linia_op1(item integer, com integer, qtt integer)
RETURNS void AS $$
BEGIN
    IF (qtt < 12)
        THEN RAISE EXCEPTION 'Quantitat % inferior a 12', qtt;
        ELSEIF (qtt > 600)
            THEN RAISE EXCEPTION 'Quantitat % superior a 600', qtt;
    END IF;
    INSERT INTO items_comanda
        VALUES (item, com, qtt);
    RETURN;

    EXCEPTION
        WHEN raise_exception THEN
            RAISE EXCEPTION '%', SQLERRM;
        WHEN foreign_key_violation THEN
            RAISE EXCEPTION 'La comanda o el item no existeixen';
        WHEN OTHERS THEN
            RAISE EXCEPTION 'Error intern';
END;
$$LANGUAGE plpgsql;
```

## Gestió d'errors: Opció 2 - Captura i retorn d'un codi de retorn

```
CREATE TYPE TError AS (  
    codi varchar(5),  
    motiu varchar(50));  
  
CREATE FUNCTION nova_linia_op2(item integer, com integer, qtt integer)  
RETURNS TError AS $$  
DECLARE error TError;  
BEGIN  
    IF (qtt < 12) THEN RAISE EXCEPTION 'Quantitat % inferior a 12', qtt;  
    ELSEIF (qtt > 600) THEN RAISE EXCEPTION 'Quantitat % superior a 600', qtt;  
    END IF;  
    INSERT INTO items_comanda VALUES (item, com, qtt);  
    error.codi:='0';  
    RETURN error;  
EXCEPTION  
    WHEN raise_exception THEN  
        error.codi:=SQLSTATE; error.motiu :=SQLERRM;  
        RETURN error;  
    WHEN foreign_key_violation THEN  
        error.codi:=SQLSTATE; error.motiu:='La comanda o el item no existeixen';  
        RETURN error;  
    WHEN OTHERS THEN  
        error.codi:=SQLSTATE; error.motiu:='Error intern';  
        RETURN error;  
END;  
$$LANGUAGE plpgsql;
```

## Gestió d'errors: Opció 3 - Captura i inserció a taula d'errors

```
CREATE TABLE t_errors (  
    codi varchar(5),  
    motiu varchar(50));  
  
CREATE FUNCTION nova_linia_op3(item integer, com integer, qtt integer)  
RETURNS void AS $$  
BEGIN  
    IF (qtt < 12) THEN RAISE EXCEPTION 'Quantitat % inferior a 12', qtt;  
    ELSEIF (qtt > 600) THEN RAISE EXCEPTION 'Quantitat % superior a 600', qtt;  
    END IF;  
    INSERT INTO items_comanda VALUES (item, com, qtt);  
    RETURN;  
    EXCEPTION  
        WHEN raise_exception THEN  
            INSERT INTO t_errors VALUES (SQLSTATE,SQLERRM);  
            RETURN;  
        WHEN foreign_key_violation THEN  
            INSERT INTO t_errors VALUES (SQLSTATE,'La comanda o el item no existeixen');  
            RETURN;  
        WHEN OTHERS THEN  
            INSERT INTO t_errors VALUES (SQLSTATE,'Error intern');  
            RETURN;  
END;  
$$LANGUAGE plpgsql;
```

## Gestió d'errors: Opció 4 – Captura i resolució del motiu de l'error en el mateix procediment

```
CREATE FUNCTION insercions(d1 integer, d2 integer) returns void AS $$  
  
BEGIN  
    INSERT INTO prova VALUES (d1,d2);  
EXCEPTION  
    WHEN undefined_table THEN  
        CREATE TABLE prova(  
            a integer primary key,  
            b integer);  
  
END;  
$$LANGUAGE plpgsql;
```

- El procediment intenta recuperar-se de l'error que s'ha produït.
- En aquest exemple, si la taula prova no existeix, es produeix un error. El procediment intenta solucionar-lo creant la taula.
- ALERTA: Una solució d'aquest tipus no sempre és possible !!

## Annex:

### Invocació d'un procediment des d'un programa JDBC – Exemple

#### Execució de sentències de crida a procediments emmagatzemats

```

Create table A(x integer primary key,y varchar(10));
Create function C (y varchar(10)) Returns int AS $$
Declare
    aux int;
Begin
    Insert into A Values (1,y);
    Select Max(x) into aux From A;
    Return aux;
END;
$$LANGUAGE plpgsql;
// Fragment de codi Java
Statement s=null;
CallableStatement cs=null;

try
{
    // creem un Statement
    s = c.createStatement ();

    // creem el CallableStatement
    cs = c.prepareCall (" {? = call C (?) }");

    cs.setString (1,"David");
    ResultSet rs = cs.executeQuery ();

    rs.next ();

    int codi = rs.getInt (1);
    System.out.println ("David te el codi "+codi);

}
catch (SQLException se)
{
    System.out.println ("Error a l'executar les
    sentències.");
}

```