Base de Datos II

Unidad 5: Bases de Datos Orientada a Objetos

Objetivos

- OQL
- Manifiesto OO
- Ventajas y desventajas de BDOO.
- Conceptos Base de Datos Objetos-Relacional.
- Definición de tipos, métodos.
- Ejemplo en Oracle.

ODMG - OQL

Una BD puede ser accedida de diferentes formas:

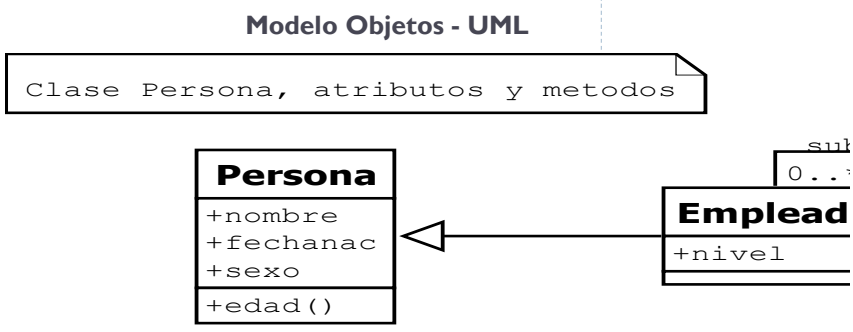
- **navegacionalmente**
  - **el programa navega de un objeto a otro utilizando las referencias entre objetos**
- **asociativamente**
  - **una consulta es especificada a través de una expresión**
  - **el OODBMS hace accesibles a los objetos que resuelven la consulta al usuario**

OQL (Object Query Language)

- Lenguaje de consultas al OODBMS
- Originario de la BDOO O2
- Basado en el modelo de objetos de ODMG
  - inspirado en SQL-92
  - con extensiones de orientación a objetos (objetos complejos, identidad de objeto, expresiones de camino, polimorfismo, llamada a métodos, "late-binding")
- Provee primitivas para manipular colecciones (conjuntos, listas, ...)
- Lenguaje de consultas exclusivamente
  - actualización por medio de operaciones definidas en el LPOO
- Basado en el sistema de tipos del LPOO
  - consultas de OQL pueden ser embebidas dentro del LPOO
  - consultas OQL pueden invocar métodos programadas en el LPOO

Entradas y resultados

- **Objetos consultados: colecciones**
  - **objetos con nombre (puntos de entrada en la BD)**
  - **extensiones de clase ("extend")**
- **Modelo ejemplo:**
  - **extend de Persona es Personas**
  - **extend de Empleado es Empleados**
  - **Presidente es el nombre de un objeto Persona**



Tipos de resultados

```
select distinct p.edad()
from Personas p
where p.nombre = "José"
```

- **Obtiene un conjunto con las diferentes edades de personas cuyo nombre es "José"**
  - **ya que edad es de tipo integer, el resultado es de tipo set<integer>**
  - **cláusula distinct especifica un resultado tipo set (en lugar de bag)**
  - **si se adiciona la claúsula order by p.edad resultado tipo lista**
- **Diferencia con SQL, donde una consulta resulta siempre en una tabla**

Resultados de consultas y caminos

- **El resultado puede ser de cualquier tipo del modelo ODMG**
- **No necesariamente se debe usar una consulta SELECT, se puede emplear los nombres persistentes:**
  - **departamentos; *// nombre del extent contiene la colección persistente de departamentos***
  - **Departmento_inf; *// objeto simple departmento asignado via método bind***
- **Una vez que se tiene un punto de entrada determinado, se puede emplear un camino para acceder a los atributos y objetos relacionados**
  - **Departmento_inf.director; *// retorna valor simple***
  - **Departamento_inf.director.categoria; *// retorna valor simple***
  - **Departamento_inf.profesorado; *// retorna un set de referencias a objetos Faculty***
  - **Departamento_inf.profesorado.categoría;**

Select *f.categoría from Departamento_inf.profesorado f*

Estructuras en el resultado

```
select distinct struct(id:p.edad(), sx:p.sexo)
from Personas p
where p.nombre = "José"
```

- **Obtiene un par con las edades y el sexo de cada persona**
- **El resultado es de tipo set<struct>**
- **Observar el uso de operaciones en consultas:**
  - **edad()**

Consultas anidadas en el resultado

```
select distinct struct(
nombre:emp.nombre,
subord:(select s
from emp.subordinados s
where s.salario > 5000))
from Empleados emp
```

- **Por cada empleado, obtiene un par compuesto por su nombre y el conjunto de subordinados**
- **Tipo del resultado:**
  - **set<struct(nombre:string,subord:bag<Empleado>)>**

Consultas anidadas en la clausula from

- **Obtener la edad y el sexo de los empleados de nivel 10 y de nombre José**

```
select struct(id:emp_grad.edad(),
              sx:emp_grad.sexo)
from (select emp from Empleados emp
      where emp.nivel = 10) as emp_grad
      where emp_grad.nombre = "José"
```

Consultas sin palabra clave SELECT

- **Nombres de objetos o de extends pueden ser utilizados directamente**
- **Ejemplos**
  - **obtener el presidente**

    presidente
  - **obtener los empleados subordinados del presidente**

    presidente.subordinados
  - **obtener el conjunto de todas las personas**

    Personas

Objetos y literales en resultados

Consultas pueden retornar:

- **objetos (o colecciones de objetos)**
- **literales (o colecciones de literales)**

OID de objeto puede ser:

- **OID del objeto en la BD**

  select emp from Empleados
- **OID generado por el procesador de consultas**

  select Persona(nombre:..., fnac:...) ...

Expresiones de camino

- **Utilizando una expresión de camino es posible "navegar" entre objetos a través de relaciones**
- **OQL admite dos notaciones**
  - **persona.conyuge**
  - **persona->conyuge**

Ejemplo de navegación

BD genealógica

persona.conyuge.direccion.ciudad.nombre

- **nombre: atributo**
- **ciudad: navegación de relaciones**
- **direccion: obtenido por nombre de atributo**
- **conyuge: otra persona, obtenida por navegación de relaciones**
- **persona: una persona**

Expresiones de camino y relaciones para "n"

- **Expresiones de camino se utilizan para navegar en relaciones a "1"**
  - **persona.hijos.nombre**
- **Consulta incorrecta:**
  - **persona.hijos es una lista de personas**
  - **(set<Persona>)**
- **Forma correcta:**

  select hijo.nombre

  from persona.hijos as hijo
  - **tipo de resultado: bag<string>**

Varias colecciones en la clausula FROM

```
select hijo.direccion
from Personas pers, pers.hijos hijo
```

- **los elementos de la cláusula FROM están interrelacionados**
- **Personas pers: indica que la consulta recorre el extent completo de Persona**
- **pers.hijos hijo: indica que, para cada persona pers, el conjunto de hijos será recorrido (pers.hijos)**

Clausula WHERE relación entre colecciones

```
select hijo.direccion
from Personas pers, pers.hijos hijo
where pers.direccion.calle="San Martin"
and count(pers.hijos) > 1
and pers.direccion.ciudad =
hijo.direccion.ciudad
```

- **Obtiene el conjunto de direcciones de los hijos de personas con más de un hijo, cuyos padres viven en la calle "San Martin" y cuyos hijos viven en la misma ciudad**

WHERE - join

- **Al igual que SQL, es posible realizar join entre colecciones**

```
select p
from Personas p, Ciudades c
where p.nombre = c.nombre
```

- **Obtiene las personas cuyo nombre es idéntico a un nombre de ciudad**

Invocación de operaciones

- **Operaciones (métodos) pueden ser invocadas en la misma forma que atributos.**
- **Pueden tener argumentos.**
- **El usuario no necesita conocer si la invocación es a un atributo o a una operación (salvo que existan parámetros).**
- **Ejemplo: clase Persona posee el método edad()**

```
select max (select hijo.edad
from p.hijos hijo)
from Personas p
where p.nombre = "José"
```

Ejemplo: operaciones

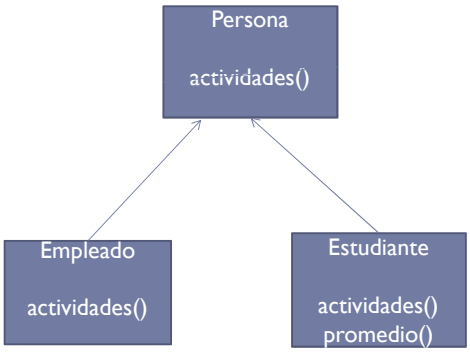
- **primogenito(): método de Persona que retorna una instancia de persona**
- **vive_en(): método de Persona que retorna TRUE si la persona vive en la ciudad pasada como argumento (string)**

```
select p.primogenito.direccion.calle
from Personas p
where p.vive_en("Santa Fe")
```

Polimorfismo

- **En tiempo de ejecución se decide qué método invocar**

```
select p.actividades
from Personas p
```



Definición estática de clase casting de tipos

- **Puede ser necesario indicar que un objeto pertenece a una subclase específica**

```
select ((Estudiante)p).promedio
from Personas p
where "estudiar" in p.actividades
```

- **(Estudiante) indica que el objeto siguiente es de tipo Estudiante, y no una Persona genérica**

Operadores de colección – Funciones agregación

- **Los resultados por lo general implican colecciones, las operaciones de agregación (min, max, sum, count, avg) operan sobre la colección**
- **count : retorna un tipo entero, el resto retorna el mismo tipo de la colección count (s in tiene_asig_secun('Ingeniería de Sistemas'));**

```
// retorna la cantidad de estudiantes que tienen estudios intermedios en ISI
avg (Select a.prom
from alumno a
where a.especialidad_en.nombred = 'Ingeniería de Sistemas'
and a.clase = 'tercer año');
// retorna el promedio de todos los estudiantes del tercer año de ISI
```

- **Las funciones de agregación se pueden usar en cualquier parte de la consulta:**

```
Select d.nombre
from departmentos d
where count(d.tiene_especialidad_en) >100;
// muestra todos los departamentos con mas de 100 alumnos
```

Operador de grupo – Group by

- **Es similar a la cláusula group by en SQL conjunto de objetos con igual valor**
- **Select struct(nombre_dept, num_alum_con_especialidad: count(partititon))**
  from alumno a
  group by nombre_dept:a.especialidad_en.nombred;
  Resultado:
  set<struct(nombre_dept:string,integer)>>
- **La sintaxis en general sería:**
  group by f1:e1, f2:e2, …,fk:ek
  donde f1:e1, f2:e2, …,fk:ek es la lista de atributos de partición donde fi define el nombre del atributo de partición y ei define una expresión relacionada con la partición

Operador de grupo – Group by

- **El resultado de aplicar el grupo es un conjunto de estructuras:**
  set<struct(f1:t1, f2:t2,..., fk:tk, partition:bag<B>)>
  donde ti es el tipo retornado por ei, partition es el nombre (clave) asignado al resultado y B es una estructura cuyos campos son las variables de iteración declaradas en el from (a en la consulta anterior)
- **También se puede aplicar una cláusula having para imponer una condición al grupo:**

select nombre_dept, avg_prom: avg(select p.a.prom from partition p)
from Alumnos a
group by nombre_dept:a.especialidad_en.nombred
having count (partition) > 100;

Operadores in, for all, exists

- **(e in c) retorna verdadero si el elemento e es miembro de la colección c**
- **(for all v in c:b) retorna verdadero si todos los elementos de la colección c satisfacen b**
- **(exists v in c:b) retorna verdadero si hay al menos un elemento en c que satisface b**
- **En el ejemplo de mas abajo, la subconsulta devuelve el conjunto de cursos que completó cada estudiante, el conector in retorna si 'Bases de Datos II' está en la colección de cada estudiante**

```
select a.nombrep.nombre, a.nombrep.apellido
from alumnos a
where 'Bases de Datos II' in
(select c.nombrec
from c in a.secciones_realizadas.seccion.de_curso)
```

Operadores in, for all, exists

- **Juan in tiene_la_especialidad('Ingeniería en Sistemas') //**
  Si Juan tiene un minor (título intermedio) en Ingeniería de Sistemas
- **Estan todos los estudiantes graduados en Ingeniería de Sistemas dirigidos por un profesor de Ingeniería de Sistemas?**

```
for all g in
(select a
from alumnos_licenciados a
where a.especialidad_en.nombred = 'Ingenieria de Sistemas')
:g.asesor in departmento_inf.tiene_profesorado;
```

- **Esta consulta ilustra también como los atributos, relaciones y herencia de operaciones se aplican en las consultas: si bien a es un iterador que recorre todos los estudiantes graduados se puede escribir a.tiene_la_especialidad porque la relación tiene_la_especialidad se hereda vía EXTENDS de Alumnos**

Manifiesto Malcom Atkinson

En 1989 se hizo el manifiesto de BDOO. Tiene 13 características obligatorios y 4 opcionales.

Características obligatorias de orientación a objetos:

- Deben soportarse objetos complejos
- Deben soportarse mecanismos de identidad de los objetos
- Debe soportarse la encapsulación
- Deben soportarse los tipos o clases
- Los tipos o clases deben ser capaces de heredar de sus ancestros
- Debe soportarse el enlace dinámico
- El DML debe ser computacionalmente complejo
- El conjunto de todos los tipos de datos debe ser ampliable

Manifiesto Malcom Atkinson

Características obligatorias de SGBD:

- Debe proporcionarse persistencia a los datos
- El SGBD debe ser capaz de gestionar bases de datos de muy gran tamaño
- El SGBD debe soportar a usuarios concurrentes
- El SGBD debe ser capaz de recuperarse de fallos hardware y software
- El SGBD debe proporcionar una forma simple de consultar los datos.

Características opcionales:

- Herencia múltiple
- Comprobación de tipos e inferencia de tipos
- Sistema de base de datos distribuido
- Soporte de versiones

Ventajas de BDOO

- **Mayor capacidad de modelado. El modelado de datos orientado a objetos permite modelar el 'mundo real' de una manera mucho más fiel. Esto se debe a:**
  - un objeto permite encapsular tanto un estado como un comportamiento
  - un objeto puede almacenar todas las relaciones que tenga con otros objetos
  - los objetos pueden agruparse para formar objetos complejos (herencia).
- **Ampliabilidad. Esto se debe a:**
  - Se pueden construir nuevos tipos de datos a partir de los ya existentes.
  - Agrupación de propiedades comunes de diversas clases e incluirlas en una superclase, lo que reduce la redundancia.
  - Reusabilidad de clases, lo que repercute en una mayor facilidad de mantenimiento y un menor tiempo de desarrollo.

Ventajas de BDOO

- Lenguaje de consulta más expresivo. El acceso navegacional desde un objeto al siguiente es la forma más común de acceso a datos en un SGBDOO. Mientras que SQL utiliza el acceso asociativo. El acceso navegacional es más adecuado para gestionar operaciones como los despieces, consultas recursivas, etc.
- Adecuación a las aplicaciones avanzadas de base de datos. Hay muchas áreas en las que los SGBD tradicionales no han tenido excesivo éxito como el CAD, CASE, OIS, sistemas multimedia, etc. en los que las capacidades de modelado de los SGBDOO han hecho que esos sistemas sí resulten efectivos para este tipo de aplicaciones.
- Mayores prestaciones. Los SGBDOO proporcionan mejoras significativas de rendimiento con respecto a los SGBD relacionales. Aunque hay autores que han argumentado que los bancos de prueba usados están dirigidos a aplicaciones de ingeniería donde los SGBDOO son más adecuados.

Inconvenientes de BDOO

- **Carencia de un modelo de datos universal. No hay ningún modelo de datos que esté universalmente aceptado para los SGBDOO y la mayoría de los modelos carecen una base teórica.**
- **Carencia de experiencia. Todavía no se dispone del nivel de experiencia del que se dispone para los sistemas tradicionales.**

Inconvenientes de BDOO

- Competencia. Con respecto a los SGBDR y los SGBDOR. Estos productos tienen una experiencia de uso considerable. SQL es un estándar aprobado y ODBC es un estándar de facto. Además, el modelo relacional tiene una sólida base teórica y los productos relacionales disponen de muchas herramientas de soporte que sirven tanto para desarrolladores como para usuarios finales.
- La optimización de consultas compromete la encapsulacion. La optimización de consultas requiere una compresión de la implementación de los objetos, para poder acceder a la base de datos de manera eficiente. Sin embargo, esto compromete el concepto de encapsulación.
- El modelo de objetos aún no tiene una teoría matemática coherente que le sirva de base.

BDOR – Tecnología BDOR de Oracle

El término Base de Datos Objeto Relacional (BDOR) se usa para describir una base de datos que ha evolucionado desde el modelo relacional hacia otra más amplia que incorpora conceptos del paradigma orientado a objetos. Por tanto, un Sistema de Gestión Objeto-Relacional (SGBDOR) contiene ambas tecnologías: relacional y de objetos.

Una idea básica de las BDOR es que el usuario pueda crear sus propios tipos de datos, para ser utilizados en aquella tecnología que permita la implementación de tipos de datos predefinidos.

Además, las BDOR permiten crear métodos para esos tipos de datos. Con ello, este tipo de SGBD hace posible la creación de funciones miembro usando tipos de datos definidos por el usuario, lo que proporciona flexibilidad y seguridad.

Estos sistemas gestionan tipos de datos complejos con un esfuerzo mínimo y albergan parte de la aplicación en el servidor de base de datos. Permiten almacenar datos complejos de una aplicación dentro de la BDOR sin necesidad de forzar los tipos de datos tradicionales.

BDOR – Tecnología BDOR de Oracle

Debido a los requerimientos de las nuevas aplicaciones, el sistema de gestión de bases de datos relacionales de Oracle, desde versión 8.i, ha sido significativamente extendido con conceptos del modelo de bases de datos orientadas a objetos. De esta manera, aunque las estructuras de datos que se utilizan para almacenar la información siguen siendo tablas, los usuarios pueden utilizar muchos de los mecanismos de orientación a objetos para definir y acceder a los datos.

Oracle proporciona mecanismos para que el usuario pueda definir sus propios tipos de datos, cuya estructura puede ser compleja, y se permite la asignación de un tipo complejo (dominio complejo) a una columna de una tabla. Además, se reconoce el concepto de objetos, de tal manera que un objeto tiene un tipo, se almacena en cierta fila de cierta tabla y tiene un identificador único (OID). Estos identificadores se pueden utilizar para referenciar a otros objetos y así representar relaciones de asociación y de agregación.

Oracle también proporciona mecanismos para asociar métodos a tipos, y constructores para diseñar tipos de datos multivaluados (colecciones) y tablas anidadas.

Tipos de Datos definidos por el usuario

Un tipo de dato define una estructura y un comportamiento común para un conjunto de datos de las aplicaciones. Los usuarios de Oracle pueden definir sus propios tipos de datos mediante dos categorías: tipos de objetos (**object type)** y tipos para colecciones **(collection type).** Para construir los tipos de usuario se utilizan los tipos básicos provistos por el sistema y otros tipos de usuario previamente definidos.

Tipos de Objetos

Un tipo de objeto representa una entidad del mundo real y se compone de los siguientes elementos:

- Su nombre que sirve para identificar el tipo de los objetos.
- Sus atributos que modelan la estructura y los valores de los datos de ese tipo. Cada atributo puede ser de un tipo de datos básico o de un tipo de usuario.
- Sus métodos (procedimientos o funciones) escritos en lenguaje PL/SQL (almacenados en la BDOR), o escritos en C (almacenados externamente).

Los tipos de objetos actúan como plantillas para los objetos de cada tipo. A continuación vemos un ejemplo de cómo definir el tipo de dato **Direccion_T** en el lenguaje de definición de datos de Oracle, y cómo utilizar este tipo de dato para definir el tipo de dato de los objetos de la clase de **Cliente_T.**

Ejemplo de definición

Definición OO	Definición en Oracle
define type Direccion_T: **tuple [**calle:string, ciudad:string, prov:string, codpos:string] **define class Cliente_T** **type tuple [**clinum: integer, clinomb:string, direccion:Direccion_T, telefono: string, fecha-nac:date] **operations** edad():integer	**CREATE TYPE direccion_t AS OBJECT (** calle VARCHAR2(200), ciudad VARCHAR2(200), prov CHAR(2), codpos VARCHAR2(20)); **CREATE TYPE cliente_t AS OBJECT (** clinum NUMBER, clinomb VARCHAR2(200), direccion direccion_t, telefono VARCHAR2(20), fecha_nac DATE, MEMBER FUNCTION edad RETURN NUMBER, PRAGMA RESTRICT_REFERENCES(edad,WNDS));

Métodos

La especificación de un método se hace junto a la creación de su tipo, y debe llevar siempre asociada una directiva de compilación (PRAGMA RESTRICT_REFERENCES), para evitar que los métodos manipulen la base de datos o las variables del paquete PL/SQL. Tienen el siguiente significado:

- WNDS: no se permite al método modificar las tablas de la base de datos
- WNPS: no se permite al método modificar las variables del paquete PL/SQL
- RNDS: no se permite al método leer las tablas de la base de datos
- RNPS: no se permite al método leer las variables del paquete PL/SQL

Los métodos se pueden ejecutar sobre los objetos de su mismo tipo. Si **x** es una variable PL/SQL que almacena objetos del tipo **Cliente_T,** entonces **x.edad**() calcula la edad del cliente almacenado en **x.**

Métodos

La definición del cuerpo de un método en PL/SQL se hace de la siguiente manera:

```
CREATE OR REPLACE TYPE BODY cliente_t AS
   MEMBER FUNCTION edad RETURN NUMBER IS
      a NUMBER;
      d DATE;
   BEGIN
      d:= today();
      a:= d.año – fecha_nac.año;
      IF (d.mes < fecha_nac.mes) OR
         ((d.mes = fecha_nac.mes) AND (d.dia < fecha_nac.dia))
      THEN a:= a-1;
      END IF;
      RETURN a;
   END;
END;
```

Constructores

En Oracle, todos los tipos de objetos tienen asociado por defecto un método que construye nuevos objetos de ese tipo de acuerdo a la especificación del tipo. El nombre del método coincide con el nombre del tipo, y sus parámetros son los atributos del tipo. Por ejemplo las siguientes expresiones construyen dos objetos con todos sus valores.

```
direccion_t('Avenida Sagunto', 'Puzol', 'Valencia', 'E-23523')
cliente_t( 2347,
   'Juan Pérez Ruíz',
   direccion_t('Calle Eo', 'Onda', 'Castellón',
                '34568'),
        '696-779789',
        12/12/1981)
```

Métodos de comparación

Para comparar los objetos de cierto tipo es necesario indicar a Oracle cuál es el criterio de comparación. Para ello, hay que escoger entre un método **MAP u ORDER,** debiéndose definir al menos uno de estos métodos por cada tipo de objeto que necesite ser comparado. La diferencia entre ambos es la siguiente:

- Un método **MAP** sirve para indicar cuál de los atributos del tipo se utilizará para ordenar los objetos del tipo, y por tanto se puede utilizar para comparar los objetos de ese tipo por medio de los operadores de comparación aritméticos (**<,** >).

Métodos de Comparación

MAP:

Por ejemplo, la siguiente declaración permite decir que los objetos del tipo **cliente_t** se van a comparar por su atributo clinum.

```
CREATE TYPE cliente_t AS OBJECT (
        clinum NUMBER,
        clinomb VARCHAR2(200),
        direccion direccion_t,
        telefono VARCHAR2(20),
        fecha_nac DATE,
MAP MEMBER FUNCTION ret_value RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(ret_value,WNDS,WNPS, RNPS, RNDS),
                /*instrucciones a PL/SQL*/
MEMBER FUNCTION edad RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(edad,WNDS));
CREATE OR REPLACE TYPE BODY cliente_t AS
    MAP MEMBER FUNCTION ret_value RETURN NUMBER IS
        BEGIN
            RETURN clinum
        END;
END;
```

Métodos de Comparación

Un método **ORDER** utiliza los atributos del objeto sobre el que se ejecuta para realizar un cálculo y compararlo con otro objeto del mismo tipo que toma como argumento de entrada.

Este método devolverá un valor negativo si el parámetro de entrada es mayor que el atributo, un valor positivo si ocurre lo contrario y un cero si ambos son iguales.

Métodos de Comparación

Order: El siguiente ejemplo define un orden para el tipo **cliente_t** diferente al anterior. Sólo una de estas definiciones puede ser válida en un tiempo dado.

```
CREATE TYPE cliente_t AS OBJECT (
        clinum NUMBER,
        clinomb VARCHAR2(200),
        direccion direccion_t,
        telefono VARCHAR2(20),
        fecha_nac DATE,
ORDER MEMBER FUNCTION
    cli_ordenados (x IN clientes_t) RETURN INTEGER,
PRAGMA RESTRICT_REFERENCES(
    cli_ordenados, WNDS, WNPS, RNPS, RNDS),
MEMBER FUNCTION edad RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(edad,WNDS));
CREATE OR REPLACE TYPE BODY cliente_t AS
        ORDER MEMBER FUNCTION cli_ordenados (x IN cliente_t)
            RETURN INTEGER IS
        BEGIN
            RETURN clinum - x.clinum; /*la resta de los dos números clinum*/
        END;
END;
```

Métodos de Comparación

Si un tipo de objeto no tiene definido ninguno de estos métodos, Oracle es incapaz de deducir cuándo un objeto es mayor o menor que otro. Sin embargo, sí puede determinar cuándo dos objetos del mismo tipo son iguales. Para ello, el sistema compara el valor de los atributos de los objetos uno a uno:

- Si todos los atributos son no nulos e iguales, Oracle indica que ambos objetos son iguales.
- Si alguno de los atributos no nulos es distinto en los dos objetos, entonces Oracle dice que son diferentes.
- En otro caso, Oracle dice que no puede comparar ambos objetos.

Tablas de Objetos

- Una vez definidos los tipos, éstos pueden utilizarse para definir nuevos tipos, tablas que almacenen objetos de esos tipos, o para definir el tipo de los atributos de una tabla. Una tabla de objetos es una clase especial de tabla que almacena un objeto en cada fila y que facilita el acceso a los atributos de esos objetos como si fueran columnas de la tabla. Por ejemplo, se puede definir una tabla para almacenar los clientes de este año y otra para almacenar los de años anteriores de la siguiente manera:

```
CREATE TABLE clientes_año_tab OF cliente_t
  (clinum PRIMARY KEY);
CREATE TABLE clientes_antiguos_tab (
  año NUMBER,
  cliente cliente_t);
```

Tablas de Objetos

La diferencia entre la primera y la segunda tabla es que la primera almacena objetos con su propia identidad (**OID**) y la segunda no es una tabla de objetos, sino una tabla con una columna con un tipo de datos de objeto. Es decir, la segunda tabla tiene una columna con un tipo de datos complejo pero sin identidad de objeto. Además de esto, Oracle permite considerar una tabla de objetos desde dos puntos de vista:

- Como una tabla con una sola columna cuyo tipo es el de un tipo de objetos.
- Como una tabla que tiene tantas columnas como atributos los objetos que almacena.

Tablas de Objetos - Ejemplo

Por ejemplo, se puede ejecutar una de las dos instrucciones siguientes. En la primera instrucción, la tabla **clientes_año_tab** se considera como una tabla con varias columnas cuyos valores son los especificados. En el segundo caso, se la considera como con una tabla de objetos que en cada fila almacena un objeto. En esta instrucción la cláusula **VALUE** permite visualizar el valor de un objeto.

```
INSERT INTO clientes_año_tab VALUES(
        2347,
        'Juan Pérez Ruíz',
        direccion_t('Calle Castalia','Onda','Castellón','34568'),
        '696-779789',
        12/12/1981);
SELECT VALUE(c) FROM clientes_año_tab c
WHERE c.clinomb = 'Juan Pérez Ruíz';
```

Las reglas de integridad, de clave primaria, y el resto de propiedades que se definan sobre una tabla, sólo afectan a los objetos de esa tabla, es decir no se refieren a todos los objetos del tipo asignado a la tabla.

Referencias entre objetos

Los identificadores únicos asignados por Oracle a los objetos que se almacenan en una tabla, permiten que éstos puedan ser referenciados desde los atributos de otros objetos o desde las columnas de tablas. El tipo de datos proporcionado por Oracle para soportar esta facilidad se denomina **REF.** Un atributo de tipo REF almacena una referencia a un objeto del tipo definido, e implementa una relación de asociación entre los dos tipos de objetos. Estas referencias se pueden utilizar para acceder a los objetos referenciados y para modificarlos; sin embargo, no es posible operar sobre ellas directamente. Para asignar o actualizar una referencia se debe utilizar siempre **REF** o **NULL.**

Cuando se define una columna de un tipo a **REF**, es posible restringir su dominio a los objetos que se almacenen en cierta tabla. Si la referencia no se asocia a una tabla sino que sólo se restringe a un tipo de objeto, se podrá actualizar a una referencia a un objeto del tipo adecuado con independencia de la tabla donde se almacene. En este caso su almacenamiento requerirá más espacio y su acceso será menos eficiente. El siguiente ejemplo define un atributo de tipo REF y restringe su dominio a los objetos de cierta tabla.

Referencias entre objetos

El siguiente ejemplo define un atributo de tipo REF y restringe su dominio a los objetos de cierta tabla.

```
CREATE TABLE clientes_tab OF cliente_t;
CREATE TYPE ordenes_t AS OBJECT (
  ordnum NUMBER,
  cliente REF clientes_t,
  fechpedido DATE,
  direcentrega direccion_t);
CREATE TABLE ordenes_tab OF ordenes_t (
  PRIMARY KEY (ordnum),
  SCOPE FOR (cliente) IS clientes_tab);
```

Referencias entre objetos

- Cuando se borran objetos de la BD, puede ocurrir que otros objetos que referencien a los borrados queden en estado inconsistente. Estas referencias se denominan *dangling references, y Oracle* proporciona el predicado llamado **IS DANGLING** que permite comprobar cuándo sucede esto.

Tipos de datos colección

Para poder implementar relaciones **1:N**, Oracle permite definir tipos colección. Un dato de tipo colección está formado por un número indefinido de elementos, todos del mismo tipo. De esta manera, es posible almacenar en un atributo un conjunto de tuplas en forma de array (**VARRAY**), o en forma de tabla anidada.

Al igual que los tipo objeto, los tipo colección también tienen por defecto unas funciones constructoras de colecciones cuyo nombre coincide con el del tipo.

En Oracle es posible diferenciar entre un valor nulo y una colección vacía. Para construir una colección sin elementos se puede utilizar la función constructora del tipo seguida por dos paréntesis sin elementos dentro.

El tipo VARRAY

Un array es un conjunto ordenado de elementos del mismo tipo. Cada elemento tiene asociado un índice que indica su posición dentro del array. Oracle permite que los **VARRAY** sean de longitud variable, aunque es necesario especificar un tamaño máximo cuando se declara el tipo **VARRAY**. Las siguientes declaraciones crean un tipo para una lista ordenada de precios, y un valor para dicho tipo.

```
CREATE TYPE precios AS VARRAY(10) OF NUMBER(12);
    precios('35','342','3970');
```

Se puede utilizar el tipo **VARRAY** para:

- Definir el tipo de dato de una columna de una tabla relacional.
- Definir el tipo de dato de un atributo de un tipo de objeto.
- Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

Cuando se declara un tipo **VARRAY** no se produce ninguna reserva de espacio. Si el espacio que requiere lo permite, se almacena junto con el resto de columnas de su tabla, pero si es demasiado largo (más de 4000 bytes) se almacena aparte de la tabla como un **BLOB.**

El tipo VARRAY

- En el siguiente ejemplo, se define un tipo de datos para almacenar una lista ordenada de teléfonos: el tipo **list** (ya que en el tipo set no existe orden). Este tipo se utiliza después para asignárselo a un atributo del tipo de objeto **cliente_t.**

Definición OO	Definición Oracle
define type Lista_Tel_T: **list(string);** **define class Cliente_T:** **tuple** [clinum: integer, clinomb:string, direccion:Direccion_T, lista_tel: Lista_Tel_T];	**CREATE TYPE lista_tel_t AS VARRAY(10) OF VARCHAR2(20) ;** **CREATE TYPE cliente_t AS OBJECT (** clinum NUMBER, clinomb VARCHAR2(200), direccion direccion_t, lista_tel lista_tel_t);

El tipo VARRAY

La principal limitación del tipo **VARRAY** es que en las consultas es imposible poner condiciones sobre los elementos almacenados dentro. Desde una consulta SQL, los valores de un **VARRAY** solamente pueden ser accedidos y recuperados como un bloque. Es decir, no se puede acceder individualmente a los elementos de un **VARRAY**. Sin embargo, desde un programa PL/SQL si que es posible definir un bucle que itere sobre los elementos de un **VARRAY.**

Tablas anidadas

Una tabla anidada es un conjunto de elementos del mismo tipo sin ningún orden predefinido. Estas tablas solamente pueden tener una columna que puede ser de un tipo de datos básico de Oracle, o de un tipo de objeto definido por el usuario. En este último caso, la tabla anidada también puede ser considerada como una tabla con tantas columnas como atributos tenga el tipo de objeto.

Tablas anidadas - Ejemplo

El siguiente ejemplo declara una tabla que después será anidada en el tipo **ordenes_t.** Los pasos de todo el diseño son los siguientes:

1. Se define el tipo de objeto **linea_t** para las filas de la tabla anidada.

```
CREATE TYPE linea_t AS OBJECT (
  linum NUMBER,
  item VARCHAR2(30),
  cantidad NUMBER,
  descuento NUMBER(6,2));
```

Tablas anidadas - Ejemplo

2. Se define el tipo colección tabla **lineas_pedido_t** para después anidarla.

```
CREATE TYPE lineas_pedido_t AS TABLE OF linea_t ;
```

Esta definición permite utilizar el tipo colección **lineas_pedido_t** para**:**

- Definir el tipo de dato de una columna de una tabla relacional.
- Definir el tipo de dato de un atributo de un tipo de objetos.
- Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

Tablas anidadas - Ejemplo

3. Se define el tipo objeto **ordenes_t** y su atributo pedido almacena una tabla anidada del tipo **lineas_pedido_t.**

```
CREATE TYPE ordenes_t AS OBJECT (
    ordnum NUMBER,
    cliente REF cliente_t,
    fechpedido DATE,
    fechentrega DATE,
    pedido lineas_pedido_t,
    direcentrega direccion_t) ;
```


Tablas anidadas - Ejemplo

4. Se define la tabla de objetos **ordenes_tab** y se especifica la tabla anidada del tipo **lineas_pedido_t.**

```
CREATE TABLE ordenes_tab OF ordenes_t
    (ordnum PRIMARY KEY,
SCOPE FOR (cliente) IS clientes_tab)
NESTED TABLE pedido STORE AS pedidos_tab) ;
```

Este último paso es necesario realizarlo porque la declaración de una tabla anidada no reserva ningún espacio para su almacenamiento. Lo que se hace es indicar en qué tabla (**pedidos_tab)** se deben almacenar todas las líneas de pedido que se representen en el atributo pedido de cualquier objeto de la tabla **ordenes_tab.** Es decir, todas las líneas de pedido de todas las ordenes se almacenan externamente a la tabla de **ordenes,** en otra tabla especial. Para relacionar las tuplas de una tabla anidada con la tupla a la que pertenecen, se utiliza una columna oculta que aparece en la tabla anidada por defecto. Todas las tuplas de una tabla anidada que pertenecen a la misma tupla tienen el mismo valor en esta columna (**NESTED_TABLE_ID).**

Tablas anidadas - Ejemplo

A diferencia de los VARRAY, los elementos de las tablas anidadas (NESTED_TABLE) sí pueden ser accedidos individualmente, y es posible poner condiciones de recuperación sobre ellos. Además, las tablas anidadas pueden estar indexadas.

Inserción y Acceso a los datos

- Alias:

  En una base de datos con tipos y objetos, lo más recomendable es utilizar siempre alias para los nombres de las tablas. El alias de una tabla debe ser único en el contexto de la consulta. Los alias sirven para acceder al contenido de la tabla, pero hay que saber utilizarlos adecuadamente en las tablas que almacenan objetos.

  El siguiente ejemplo ilustra cómo se deben utilizar.

  ```
  CREATE TYPE persona AS OBJECT (nombre VARCHAR(20));
  CREATE TABLE ptab1 OF persona;
  CREATE TABLE ptab2 (c1 persona);
  CREATE TABLE ptab3 (c1 REF persona);
  ```

Inserción y Acceso a los datos

Alias:

La diferencia entre las dos primeras tablas está en que la primera almacena objetos del tipo **persona,** mientras que la segunda tabla tiene una columna donde se almacenan valores del tipo **persona.**

Considerando ahora las siguientes consultas, se ve cómo se puede acceder a estas tablas.

1. **SELECT nombre FROM ptab1;** **Correcto**
2. **SELECT c1.nombre FROM ptab2;** **Incorrecto**
3. **SELECT p.c1.nombre FROM ptab2 p;** **Correcto**
4. **SELECT p.c1.nombre FROM ptab3 p;** **Correcto**
5. **SELECT p.nombre FROM ptab3 p;** **Incorrecto**

En la primera consulta **nombre** es considerado como una de las columnas de la tabla ptab1, ya que los atributos de los objetos se consideran columnas de la tabla de objetos. Sin embargo, en la segunda consulta se requiere la utilización de un alias para indicar que **nombre** es el nombre de un atributo del objeto de tipo **persona** que se almacena en la columna c1. Para resolver este problema no es posible utilizar los nombres de las tablas directamente: **ptab2.c1.nombre** es incorrecto. Las consultas 4 y 5muestran cómo acceder a los atributos de los objetos referenciados desde un atributo de la tabla **ptab3.**

Inserción y Acceso a los datos

- Inserción de Referencias:

La inserción de objetos con referencias implica la utilización del operador **REF** para poder insertar la referencia en el atributo adecuado. La siguiente sentencia inserta una orden de pedido.

```
INSERT INTO ordenes_tab
SELECT 3001, REF(C),'30-MAY-1999', NULL
  --se seleccionan los valores de los 4 atributos de la tabla
FROM cliente_tab C WHERE C.clinum= 3;
```

El acceso a un objeto desde una referencia **REF** requiere primero referenciar al objeto. Para realizar esta operación, Oracle proporciona el operador **DEREF.** No obstante, utilizando la notación de punto también se consigue referenciar a un objeto de forma implícita.

Inserción y Acceso a los datos

- Llamadas a métodos:

Para invocar un método hay que utilizar su nombre y unos paréntesis que encierren sus argumentos de entrada. Si el método no tiene argumentos, se especifican los paréntesis aunque estén vacíos. Por ejemplo, si **tb** es una tabla con la columna **c** de tipo de objeto **t, y t** tiene un método **m** sin argumentos de entrada, la siguiente consulta es correcta:

SELECT p.c.m( ) FROM tb p;

Inserción y Acceso a los datos

- Inserción en tablas anidadas:

Además del constructor del tipo de colección disponible por defecto, la inserción de elementos dentro de una tabla anidada puede hacerse siguiendo estas dos etapas:

1. Crear el objeto con la tabla anidada y dejar vacío el campo que contiene las tuplas anidadas.
2. Comenzar a insertar tuplas en la columna correspondiente de la tupla seleccionada por una subconsulta.

Para ello, se tiene que utilizar la palabra clave **THE** con la siguiente sintaxis:

INSERT INTO THE (subconsulta) (tuplas a insertar)

Esta técnica es especialmente útil si dentro de una tabla anidada se guardan referencias a otros objetos.

Ejemplo de Trabajo - Oracle

- Partiremos de una base de datos para gestionar los pedidos de los clientes, y veremos cómo Oracle permite proporcionar una solución relacional y otra objeto-relacional.
- Modelo lógico BDR:

  CLIENTE(clinum, clinomb, calle, ciudad, prov, codpos, tel1, tel2, tel3)

  PEDIDO (ordnum, clinum, fechped, fechentrega, callent, ciuent, provent, codpent)

  ITEM(numitem, precio, tasas)

  LINEA(linum, ordnum, numitem, cantidad, descuento)

Ejemplo de Trabajo - Oracle

Creamos las tablas normalizadas y con claves externas para representar las relaciones.

```
CREATE TABLE cliente (
    clinum NUMBER,
    clinomb VARCHAR2(200),
    calle VARCHAR2(200),
    ciudad VARCHAR2(200),
    prov CHAR(2),
    codpos VARCHAR2(20),
    tel1 VARCHAR2(20),
    tel2 VARCHAR2(20),
    tel3 VARCHAR2(20),
PRIMARY KEY (clinum));

CREATE TABLE pedido (
    ordnum NUMBER,
    clinum NUMBER REFERENCES cliente,
    fechpedido DATE,
    fechaentrega DATE,
    callent VARCHAR2(200),
    ciuent VARCHAR2(200),
    provent CHAR(2),
    codpent VARCHAR2(20),
PRIMARY KEY (ordnum));
```

Ejemplo de Trabajo - Oracle

```
CREATE TABLE item (
   numitem NUMBER PRIMARY KEY,
   precio NUMBER,
   tasas NUMBER);

CREATE TABLE linea (
   linum NUMBER,
   ordnum NUMBER REFERENCES pedido,
   numitem NUMBER REFERENCES item,
   cantidad NUMBER,
   descuento NUMBER,
PRIMARY KEY (ordnum, linum));
```

Ejemplo de Trabajo - Oracle

- Modelo lógico de una BDOO:

```
define type Lista_Tel_T: type list(string);
define type Direccion_T: type tuple [ calle:string,
   ciudad:string,
   prov:string,
   codpos:string];
define class Cliente_T: type tuple [ clinum: integer,
   clinomb:string,
   direccion:Direccion_T,
   lista_tel: Lista_Tel_T];
define class Item_T: type tuple [ itemnum:integer,
   precio:real,
   tasas:real];
```

Ejemplo de Trabajo - Oracle

- Modelo lógico de una BDOO:

define type Linea_T: type tuple [linum:integer,
item:Item_T,
cantidad:integer,
descuento:real];
define type Linea_Pedido_T: type set(Linea_T);
define class Pedido_T: type tuple [ordnum:integer,
cliente:Cliente_T,
fechpedido:date,
fechentrega:date,
pedido:Lineas_Pedido_T
direcentrega:Direccion_T];

Ejemplo de Trabajo - Oracle

- Implementación Objeto-Relacional con Oracle: Definición de tipos

CREATE TYPE lista_tel_t **AS VARRAY(10) OF VARCHAR2(20) ;**

CREATE TYPE direccion_t **AS OBJECT (**
calle VARCHAR2(200),
ciudad VARCHAR2(200),
prov CHAR(2),
codpos VARCHAR2(20)) ;

Ejemplo de Trabajo - Oracle

- Implementación Objeto-Relacional con Oracle: Definición de tipos

```
CREATE TYPE cliente_t AS OBJECT (
    clinum NUMBER,
    clinomb VARCHAR2(200),
    direccion direccion_t,
    lista_tel lista_tel_t) ;
CREATE TYPE item_t AS OBJECT (
    itemnum NUMBER,
    precio NUMBER,
    tasas NUMBER) ;
CREATE TYPE linea_t AS OBJECT (
    linum NUMBER,
    item REF item_t,
    cantidad NUMBER,
    descuento NUMBER) ;
CREATE TYPE lineas_pedido_t AS TABLE OF linea_t ;
CREATE TYPE pedido_t AS OBJECT (
    ordnum NUMBER,
    cliente REF cliente_t,
    fechpedido DATE,
    fechentrega DATE,
    pedido lineas_pedido_t,
    direcentrega direccion_t) ;
```

Ejemplo de Trabajo - Oracle

- Implementación Objeto-Relacional con Oracle: Creación de Tablas Objetos

```
CREATE TABLE cliente_tab OF cliente_t
    (clinum PRIMARY KEY);
CREATE TABLE item_tab OF item_t
    (itemnum PRIMARY KEY) ;
CREATE TABLE pedido_tab OF pedido_t (
    PRIMARY KEY (ordnum),
    SCOPE FOR (cliente) IS cliente_tab)
    NESTED TABLE pedido STORE AS pedido_tab ;
ALTER TABLE pedido_tab
    ADD (SCOPE FOR (item) IS item_tab) ;
```

Esta última declaración sirve para restringir el dominio de los objetos referenciados desde **item** a aquellos que se almacenan en la tabla **item_tab.**

Ejemplo de Trabajo - Oracle

- Inserción de Objetos:

```
REM inserción en la tabla ITEM_TAB**********
INSERT INTO item_tab VALUES(1004, 6750.00, 2);
INSERT INTO item_tab VALUES(1011, 4500.23, 2);
INSERT INTO item_tab VALUES(1534, 2234.00, 2);
INSERT INTO item_tab VALUES(1535, 3456.23, 2);
INSERT INTO item_tab VALUES(2004, 33750.00, 3);
INSERT INTO item_tab VALUES(3011, 43500.23, 4);
INSERT INTO item_tab VALUES(4534, 5034.00, 6);
INSERT INTO item_tab VALUES(5535, 34456.23, 5);
REM inserción
```

Ejemplo de Trabajo - Oracle

- Inserción de Objetos:

Nótese cómo en estas definiciones se utilizan los constructores del tipo de objeto **direccion_t** y el tipo de colección **lista_tel_t.**

```
INSERT INTO cliente_tab
  VALUES (
        1,'Lola Caro',
        direccion_t('12 Calle Lisboa','Nules','CS','12678'),
        lista_tel_t('415-555-1212')) ;

INSERT INTO cliente_tab
  VALUES (
        2,'Jorge Luz',
        direccion_t('323 Calle Sol','Valencia','V','08820'),
        lista_tel_t('609-555-1212','201-555-1212')) ;
```

Ejemplo de Trabajo - Oracle

- Inserción de Objetos:

Nótese cómo en estas definiciones se utiliza el operador **REF** para obtener una referencia a un objeto de cliente_tab y almacenarlo en la columna de otro objeto de pedido_tab. La palabra clave **THE s**e utiliza para designar la columna de las tuplas que cumplen la condición del **WHERE**, donde se deben realizar la inserción. Las tuplas que se insertan son las designadas por el segundo **SELECT**, y el objeto de la orden debe existir antes de comenzar a insertar líneas de pedido.

```
INSERT INTO pedido_tab
    SELECT 1001, REF(C),
    SYSDATE,'10-MAY-1999',
    linea_pedido_t(),
    NULL
    FROM cliente_tab C
    WHERE C.clinum= 1 ;
INSERT INTO THE (
    SELECT P.pedido
    FROM pedido_tab P
    WHERE P.ordnum = 1001
    )
    SELECT 01, REF(S), 12, 0
    FROM item_tab S
    WHERE S.itemnum = 1534;
```

Ejemplo de Trabajo - Oracle

- Definición de Métodos para los tipos:

El siguiente método calcula la suma de los valores de las líneas de pedido correspondientes a la orden de pedido sobre la que se ejecuta.

```
CREATE TYPE pedido_t AS OBJECT (
    ordnum NUMBER,
    cliente REF cliente_t,
    fechpedido DATE,
    fechentrega DATE,
    pedido linea_pedido_t,
    direcentrega direccion_t,
MEMBER FUNCTION
coste_total RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(coste_total, WNDS, WNPS) );
```

Ejemplo de Trabajo - Oracle

- Definición de Métodos para los tipos:

```
CREATE TYPE BODY pedido_t AS
   MEMBER FUNCTION coste_total RETURN NUMBER IS
      i INTEGER;
      item item_t;
      linea linea_t;
      total NUMBER:=0;
BEGIN
   FOR i IN 1..SELF.pedido.COUNT LOOP
      linea:=SELF.pedido(i);
      SELECT DEREF(linea.item) INTO item FROM DUAL;
      total:=total + linea.cantidad * item.precio;
   END LOOP;
   RETURN total;
END;
END;
```

Ejemplo de Trabajo - Oracle

- La palabra clave **SELF** permite referirse al objeto sobre el que se ejecuta el método.
- La palabra clave **COUNT** sirve para contar el número de elementos de una tabla o de un array. Junto con la instrucción **LOOP** permite iterar sobre los elementos de una colección, en nuestro caso las líneas de pedido de una orden.
- El **SELECT** es necesario porque Oracle no permite utilizar **DEREF** directamente en el código PL/SQL.

Ejemplo de Trabajo - Oracle

- Consultas a BDOR:

1. Consultar la definición de la tabla de clientes.

   SQL> describe cliente_tab;

2. **Insertar en la tabla de clientes a un nuevo cliente con todos sus datos.**

```
SQL> insert into cliente_tab
    values(5,'John smith',
    direccion_t('67 rue de percebe','Gijon','AS',
    '73477'),lista_tel_t('7477921749','83797597827'));
```

Ejemplo de Trabajo - Oracle

- Consultas a BDOR:

3. Consultar y modificar el nombre del cliente número 2.

```
SQL> select clinomb from cliente_tab
  where clinum=2;

SQL> update cliente_tab
        set clinomb='Pepe Puig' where clinum=5;
```

Ejemplo de Trabajo - Oracle

- Consultas a BDOR:

4. Consultar y modificar la dirección del cliente número 2.

```
SQL> select direccion from cliente_tab where
  clinum=2;
DIRECCION(CALLE, CIUDAD, PROV, CODPOS)
--------------------------------------------------------------------
DIRECCION_T('Calle Sol', 'Valencia', 'VA', '08820')
SQL> update cliente_tab
      set direccion=direccion_t('Calle Luna','Castello',
      'CS',68734')
      where clinum=2;
```

Ejemplo de Trabajo - Oracle

- Consultas a BDOR:

5. Consultar todos los datos del cliente número 1 y añadir un nuevo teléfono a su lista de teléfonos.

```
SQL> select * from cliente_tab where clinum=1;
CLINUM
----------
CLINOMB
--------------------------------------------------------------
DIRECCION(CALLE, CIUDAD, PROV, CODPOS)
--------------------------------------------------------------
LISTA_TEL
--------------------------------------------------------------
1
Lola Caro
DIRECCION_T('Calle Luna', 'Castellon', 'CS', '64827')
LISTA_TEL_T('415-555-1212')

SQL> update cliente_tab
      set lista_tel=lista_tel_t('415-555-1212',
      '6348635872')
      where clinum=1;
```

Mapeo Objeto Relacional (ORM)

- Los ORM son herramientas de software que permiten trabajar con los datos persistidos en nuestras bases de datos relacionales como si ellos fueran parte de una base de datos orientada a objetos (virtual).
- En aplicaciones estándar, realizadas sobre bases de datos relacionales, la funcion del ORM es **transformar un registro en objeto y viceversa**, en pos de realizar operaciones de consulta y persistencia directamente sobre los objetos.

Mapeo Objeto Relacional

- Existen variados componentes que se pueden utilizar a tal fin en tecnología .NET (muchos de los cuales son portaciones de componentes existentes en JAVA).
- El mas utilizado y el que posee un mayor soporte en la actualidad es NHIbernate (http://www.hibernate.org/343.html), aunque tambien contamos con otros muy buenos como ORM.net oWilson ORM Mapper.
- Existen además, herramientas que "encapsulan" un ORM e incorporan funcionalidad adicional como ActiveRecord.net, que utiliza NHibernate para persistir, pero además facilita su uso notablemente evitándonos la realización de archivos de mapeo (necesarios para relacionar nuestras tablas con nuestras clases) e incorpora validaciones diversas sobre nuestras clases.

¿Cuándo usar ORM?

Consideremos usar un ORM si:

- Tenemos la posibilidad de diseñar la base de datos, ya que de esa manera, podremos seguir prácticas que facilitarán la interacción ORM-DB.
- Queremos abstraernos del motor de base de datos (SQL Server, Oracle) y poder cambiarlo sin demasiado inconveniente).

Y evitemos utilizarlo cuando:

- Recibamos bases de datos que posean características tales como tablas con varios campos como clave principal o sin claves primarias y/o foráneas, ya que eso podría dificultar el mapeo y las consultas.
- Necesitemos ejecutar procesos BATCH con millones de registros

Objetivos

- OQL
- Manifiesto OO
- Ventajas y desventajas de BDOO.
- Conceptos Base de Datos Objetos-Relacional.
- Definición de tipos, métodos.
- Ejemplo en Oracle.