# TEMA 7

# INDICE

# Uso de bases de datos objeto-relacionales.

**Caso práctico**

***Juan*** *ha realizado un curso de perfeccionamiento sobre programación con bases de datos, y ha conocido las ventajas que pueden ofrecer el uso de las bases de datos objeto-relacionales. Hasta ahora siempre había usado las bases de datos relacionales, pero el conocimiento de las bases de datos orientadas a objetos le ha ofrecido nuevas perspectivas para aprovechar de manera más óptima la reutilización de código, el trabajo colaborativo, y la interconexión de las bases de datos con otros lenguajes de programación orientados a objetos.*

# 1.- Características de las bases de datos objeto-relacionales.

**Caso práctico**

*__Ana__ ha sabido que __Juan__ ha realizado el curso de formación de bases de datos objeto-relacionales, y éste se ha ofrecido a compartir los conocimientos que ha adquirido. Lo primero va a ser es que conozca las características que tienen estos tipos de bases de datos, para que compruebe las diferencias que tienen respecto a las bases de datos relaciones que ha usado hasta ahora. Usarán la base de datos de Oracle que han utilizado anteriormente, pero ahora van a darle el enfoque de este nuevo tipo de bases de datos.*

Las **bases de datos objeto-relacionales** que vas a conocer en esta unidad son las referidas a aquellas que han evolucionado desde el modelo relacional tradicional a un modelo híbrido que utiliza además la tecnología orientada a objetos. Las **clases, objetos,** y **herencia** son directamente soportados en los esquemas de la base de datos y el lenguaje de consulta y manipulación de datos. Además da soporte a una extensión del modelo de datos con la creación personalizada de **tipos de datos y métodos**.

La base de datos de Oracle implementa el modelo orientado a objetos como una **extensión del modelo relacional**, siguiendo soportando la funcionalidad estándar de las bases de datos relacionales.

El modelo objeto-relacional ofrece las **ventajas** de las técnicas orientadas a objetos en cuanto a mejorar la reutilización y el uso intuitivo de los objetos, a la vez que se mantiene la alta capacidad de concurrencia y el rendimiento de las bases de datos relacionales.

Los tipos de objetos que vas a conocer, así como las características orientadas a objeto, te van a proporcionar un mecanismo para **organizar los datos y acceder a ellos a alto nivel**. Por debajo de la capa de objetos, los datos seguirán estando almacenados en columnas y tablas, pero vas a poder trabajar con ellos de manera más parecida a las entidades que puedes encontrar en la vida real, dando así más significado a los datos. En vez de pensar en términos de columnas y tablas cuando realices consultas a la base de datos, simplemente deberás seleccionar entidades que habrás creado, por ejemplo clientes o pedidos.

Podrás utilizar las características orientadas a objetos que vas a aprender en esta unidad a la vez que puedes seguir trabajando con los datos relacionalmente, o bien puedes usar de manera más exclusiva las técnicas orientadas a objetos.

En general, el modelo orientado a objetos es **similar al que puedes encontrar en lenguajes** como C++ y Java. La **reutilización** de objetos permite desarrollar aplicaciones de bases de datos más rápidamente y de manera más eficiente. Al ofrecer la base de datos de Oracle soporte nativo para los tipos de objetos, permite a los desarrolladores de aplicaciones con lenguajes orientados a objetos, acceder directamente a las **mismas estructuras de datos**creadas en la base de datos.

La programación orientada a objetos está especialmente enfocada a la construcción de componentes reutilizables y aplicaciones complejas. En **PL/SQL**, la programación orientada a objetos está basada en **tipos de objetos**. Estos tipos te permiten modelar objetos de la vida real, separar los detalles de los interfaces de usuarios de la implementación, y almacenar los datos orientados a objetos de forma permanente en una base de datos. Encontraras especialmente útil los tipos de objetos cuando realizas programas interconectados con Java u otros lenguajes de programación orientados a objetos.

Las tablas de bases de datos relacionales sólo contienen datos. En cambio, los objetos pueden incluir la posibilidad de realizar determinadas **acciones sobre los datos**. Por ejemplo, un objeto "Compra" puede incluir un método para calcular el importe de todos los elementos comprados. O un objeto

"Cliente" puede tener métodos que permitan obtener su historial de compras. De esta manera, una aplicación tan sólo debe realizar una llamada a dichos métodos para obtener esa información.

**Si deseas conocer más en detalle la definición de las bases de datos objeto-relacionales puedes visitar la web de wikipedia:**
http://es.wikipedia.org/wiki/Base_de_datos_objeto-relacional

## 2.- Tipos de datos objeto.

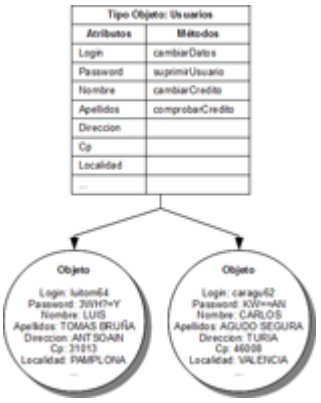
**Caso práctico**

*__Juan__ comienza a explicarle a __Ana__ uno de los conceptos básicos en las bases de datos orientadas a objetos. Se trata de los tipos de datos objeto. Le hace ver que cualquier objeto que manejamos a diario se puede considerar un posible tipo de objeto para una base de datos.*

Un **tipo de dato objeto** es un tipo de dato compuesto definido por el usuario. Representa una **estructura de datos** así como **funciones y procedimientos** para manipular datos. Como ya sabemos, las variables de un determinado tipo de dato escalar (`NUMBER`, `VARCHAR`, `BOOLEAN`, `DATE`, etc.) pueden almacenar un único valor. Las coleccionas permiten, como ya veremos, almacenar varios datos en una misma variable siendo todos del mismo tipo. Los tipos de datos objetos nos permitirán almacenar datos de distintos tipos, posibilitando además asociar código a dichos datos.

En el mundo real solemos pensar en un objeto, entidad, persona, animal, etc. como un conjunto de propiedades y acciones. Por ejemplo, el alumnado tiene una serie de propiedades como el nombre, fecha de nacimiento, DNI, curso, grupo, calificaciones, nombre del padre y de la madre, sexo, etc., y tiene asociadas una serie de acciones como matricular, evaluar, asistir a clase, presentar tareas, etc. Los tipos de datos objeto nos permiten representar esos valores y estos comportamientos de la vida real en una aplicación informática.

**Piensa en un objeto y enumera algunas de sus propiedades y acciones que se pueden realizar sobre él.**




Las variables que formen la estructura de datos de un tipo de dato objeto reciben el nombre de **atributos** (que se corresponde con sus propiedades). Las funciones y procedimientos del tipo de dato objeto se denominan **métodos**(que se corresponde con sus acciones).

Cuando se define un tipo de objeto, se crea una **plantilla abstracta** de un objeto de la vida real. La plantilla especifica los atributos y comportamientos que el objeto necesita en el entorno de la aplicación. Dependiendo de la aplicación a desarrollar se utilizarán sólo determinados atributos y comportamiento del objeto. Por ejemplo, en la gestión de la evaluación del alumnado es muy probable que no se necesite conocer su altura, peso, etc. o utilizar comportamientos como desplazarse, comer, etc., aunque formen todos ellos parte de las características del alumnado.

Aunque los atributos son públicos, es decir, visibles desde otros programas cliente, los programas deberían **manipular los datos únicamente a través de los métodos** (funciones y procedimientos) que se hayan declarado en el tipo objeto, en vez de asignar u obtener sus valores directamente. Esto es debido a que los métodos pueden hacer un chequeo de los datos de manera que se mantenga un estado apropiado en los mismos. Por ejemplo, si se desea asignar un curso a un miembro del alumnado, sólo debe permitirse que se asigne un curso existente. Si se permitiera modificar directamente el curso, se podría asignar un valor incorrecto (curso inexistente).

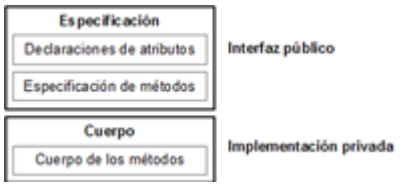
Durante la ejecución, la aplicación creará **instancias** de un tipo objeto, es decir, referencias a objetos reales con valores asignados en sus atributos. Por ejemplo, una instancia será un determinado miembro del alumnado con sus datos personales correspondientes.

# 3.- Definición de tipos de objeto.

**Caso práctico**

*Ya conoce **Ana** el concepto de las bases de datos orientadas a objetos y que los tipos de datos objeto son la base para ellas. Así que para asentar esos conocimientos se plantea modificar, con la ayuda de **Juan**, la base de datos de la plataforma de juegos on-line que hasta ahora era relacional para convertirla en una base de datos orientada a objetos.*

*Para ello, se plantea crear el tipo de objeto Usuario, pero necesita conocer cómo se declaran los tipos de datos objeto en Oracle.*




La estructura de la definición o declaración de un tipo de objeto está dividida en una especificación y un cuerpo. La **especificación** define el interfaz de programación, donde se declaran los atributos así como las operaciones (métodos) para manipular los datos. En el **cuerpo** se implementa el código fuente de los métodos.

Toda la información que un programa necesita para usar los métodos lo encuentra en la especificación. Se puede modificar el cuerpo sin cambiar la especificación, sin que ello afecte a los programas cliente.

En la especificación de un tipo de objeto, todos los **atributos debes declararlos antes que los métodos**. Si la especificación de un tipo de objeto sólo declara atributos, no es necesario que declares el cuerpo. Debes tener en cuenta también que no puedes declarar atributos en el cuerpo. Además, todas las declaraciones realizadas en la especificación del tipo de objeto son públicas, es decir, visibles fuera del tipo de objeto.

Por tanto, un tipo de objeto contiene (encapsula) datos y operaciones. Puedes declarar atributos y métodos en la especificación, pero no constantes (`CONSTANTS`), excepciones (`EXCEPTIONS`), cursores (`CURSORS`) o tipos (`TYPES`). Al menos debe tener un atributo declarado, y un máximo de 1000. En cambio los métodos son opcionales, por lo que se puede crear un tipo de objeto sin métodos.
Para definir un objeto en Oracle debes utilizar la sentencia `CREATE TYPE` que tiene el siguiente formato:

```
CREATE TYPE nombre_tipo AS OBJECT (
Declaración_atributos
Declaración_métodos
);
```

Siendo nombre_tipo el nombre deseado para el nuevo tipo de objeto. La forma de declarar los atributos y los métodos se verá en los siguientes apartados de esta unidad didáctica.

En caso de que el **nombre del tipo de objeto ya estuviera siendo usado** para otro tipo de objeto se obtendría un error. Si se desea reemplazar el tipo anteriormente creado, por el nuevo que se va a declarar, se puede añadir la cláusula `OR REPLACE` en la declaración del tipo de objeto:

```
CREATE OR REPLACE TYPE nombre_tipo AS OBJECT
```

Por ejemplo, para crear el tipo de objeto Usuario, **reemplazando la declaración** que tuviera anteriormente se podría hacer algo similar a lo siguiente:

```
CREATE OR REPLACE TYPE Usuario AS OBJECT (
Declaración_atributos
Declaración_métodos
);
```

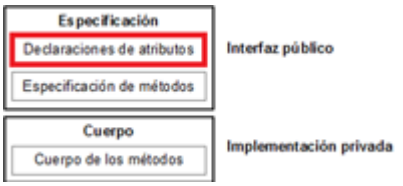
Si en algún momento deseas **eliminar el tipo de objeto** que has creado puedes utilizar la sentencia `DROP TYPE`:

```
DROP TYPE nombre_tipo;
```

Donde nombre_tipo debe ser el nombre del tipo de dato objeto que deseas eliminar. Por ejemplo, para el tipo de objetos anterior, deberías indicar:

```
DROP TYPE Usuario;
```

## 3.1.- Declaración de atributos.




La declaración de los atributos la puedes realizar de forma muy similar a declaración de las variables, es decir, utilizando un nombre y un tipo de dato. Dicho nombre debe ser único dentro del tipo de objeto, aunque puede ser reutilizado en otros tipos de objeto. El tipo de dato que puede almacenar un determinado atributo puede ser **cualquiera de los tipos de Oracle excepto** los siguientes:

- ✓ `LONG` y `LONG RAW`.
- ✓ `ROWID` y `UROWID`.
- ✓ Los tipos específicos PL/SQL `BINARY INTEGER` (y sus subtipos), `BOOLEAN`, `PLS INTEGER`, `RECORD`, `REF CURSOR`, `%TYPE`, y `%ROWTYPE`.
- ✓ Los tipos definidos dentro de un paquete PL/SQL.

Debes tener en cuenta que no puedes inicializar los atributos usando el operador de asignación, ni la cláusula `DEFAULT`, ni asignar la restricción `NOT NULL`.

El **tipo de dato de un atributo puede ser otro tipo de objeto**, por lo que la estructura de datos puede ser tan complicada como sea necesario.

```
CREATE OR REPLACE TYPE Usuario AS OBJECT (
   login VARCHAR2(10),
   nombre VARCHAR2(30),
   f_ingreso DATE,
   credito NUMBER
);
/
```

Después de haber sido creado el tipo de objeto, se pueden **modificar sus atributos** utilizando la sentencia `ALTER TYPE`. Si se desean añadir nuevos atributos se añadirá la cláusula `ADD ATTRIBUTE` seguida de la lista de nuevos atributos con sus correspondientes tipos de dato. Utilizando `MODIFY ATTRIBUTE` se podrán modificar los atributos existentes, y para eliminar atributos se dispone de manera similar de `DROP ATTRIBUTE`.

Aquí tienes varios ejemplos de modificación del tipo de objeto Usuario creado anteriormente:

```
ALTER TYPE Usuario DROP ATTRIBUTE f_ingreso;
```

```
ALTER TYPE Usuario ADD ATTRIBUTE (apellidos VARCHAR2(40), localidad VARCHAR2(50));
```

```
ALTER TYPE Usuario
   ADD ATTRIBUTE cp VARCHAR2(5),
   MODIFY ATTRIBUTE nombre VARCHAR2(35);
```

**En la web de Oracle puedes encontrar la sintaxis completa de la instrucción CREATE TYPE. Ahí podrás conocer que hay más posibilidades de uso:**
http://download.oracle.com/docs/cd/B13789_01/server.101/b10759/statements_8001.htm

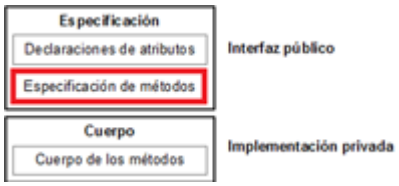
**En la web de Oracle puedes encontrar la sintaxis completa de la instrucción ALTER TYPE. Ahí podrás conocer que hay más posibilidades de uso:**
http://docs.oracle.com/cd/B14117_01/server.101/b10759/statements_4002.htm

## 3.2.- Definición de métodos.

Un **método es un subprograma** que declaras en la especificación de un tipo de objeto usando las palabras clave `MEMBER` o `STATIC`. Debes tener en cuenta que el nombre de un determinado método no puede puede ser el mismo nombre que el tipo de objeto ni el de ninguno de sus atributos. Como se verá más adelante, se pueden crear métodos con el mismo nombre que el tipo de objeto, pero dichos métodos tendrán una función especial.

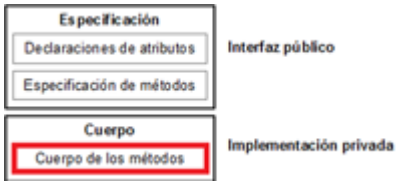
Al igual que los subprogramas empaquetados, los métodos tienen **dos partes: una especificación y un cuerpo**.

En la **especificación** o declaración se debe encontrar el nombre del método, una lista opcional de parámetros, y, en el caso de las funciones, un tipo de dato de retorno. Por ejemplo, observa la especificación del método incrementoCredito que se encuentra detrás de las declaraciones de los atributos:




```
CREATE OR REPLACE TYPE Usuario AS OBJECT (
   login VARCHAR2(10),
   nombre VARCHAR2(30),
   f_ingreso DATE,
   credito NUMBER,
   MEMBER PROCEDURE incrementoCredito(inc NUMBER)
);
/
```

En el **cuerpo** debes indicar el código que se debe ejecutar para realizar una determinada tarea cuando el método es invocado. En el siguiente ejemplo se desarrolla el cuerpo del método que se ha declarado antes:




```
CREATE OR REPLACE TYPE BODY Usuario AS
   MEMBER PROCEDURE incrementoCredito(inc NUMBER) IS
      BEGIN
         credito := credito + inc;
      END incrementoCredito;
END;
/
```

Por cada especificación de método que se indique en el bloque de especificación del tipo de objeto, debe existir su correspondiente cuerpo del método, o bien, el método debe declararse como `NOT INSTANTIABLE`, para indicar que el cuerpo del método se encontrará en un subtipo de ese tipo de objeto. Además, debes tener en cuenta que las cabeceras de los métodos deben coincidir exactamente en la especificación y en el cuerpo.

Al igual que los atributos, los parámetros formales se declaran con un nombre y un tipo de dato. Sin embargo, el tipo de dato de un parámetro no puede tener restricciones de tamaño. El tipo de datos puede ser cualquiera de los empleados por Oracle salvo los indicados anteriormente para los atributos. Las mismas restricciones se aplican para los tipos de retorno de las funciones.

El código fuente de los métodos no sólo puede escribirse en el lenguaje PL/SQL. También con otros lenguajes de programación como Java o C.

Puedes usar la sentencia `ALTER TYPE` para **añadir, modificar o eliminar métodos** de un tipo de objeto existente, de manera similar a la utilizada para modificar los atributos de un tipo de objeto.

## 3.3.- Parámetro SELF.

Un parámetro especial que puedes utilizar con los métodos `MEMBER` es el que recibe el nombre `SELF`. Este parámetro **hace referencia a una instancia (objeto) del mismo tipo de objeto**. Aunque no lo declares explícitamente, este parámetro siempre se declara automáticamente.

El **tipo de dato** correspondiente al parámetro `SELF` será el mismo que el del objeto original. En las funciones `MEMBER`, si no declaras el parámetro `SELF`, su modo por defecto se toma como `IN`. En cambio, en los procedimientos`MEMBER`, si no se declara, se toma como `IN OUT`. Ten en cuenta que no puedes especificar el modo `OUT` para este parámetro `SELF`, y que los métodos `STATIC` no pueden utilizar este parámetro especial.

Si se hace referencia al parámetro `SELF` **dentro del cuerpo de un método**, realmente se está haciendo referencia al objeto que ha invocado a dicho método. Por tanto, si utilizas `SELF.nombre atributo` o `SELF.nombre método`, estarás utilizando un atributo o un método del mismo objeto que ha llamado al método donde se encuentra utilizado el parámetro `SELF`.

```
MEMBER PROCEDURE setNombre(Nombre VARCHAR2) IS
   BEGIN
      /* El primer elemento (SELF.Nombre) hace referencia al atributo del tipo de objeto
mientras que el
      segundo (Nombre) hace referencia al parámetro del método */
      SELF.Nombre := Nombre;
   END setNombre;
```

**¿Cómo se denonima a los elementos que realizan determinadas acciones sobre los objetos?**

- ○ Atributos
- ◉ **Métodos**
- ○ Tipos de datos objeto
- ○ Parámetros

## 3.4.- Sobrecarga.

Al igual que ocurre con los subprogramas empaquetados, los métodos pueden ser sobrecargados, es decir, puedes **utilizar el mismo nombre para métodos diferentes** siempre que sus parámetros formales sean diferentes (en cantidad o tipo de dato).

Cuando se hace una llamada a un método, se comparan los parámetros actuales con los parámetros formales de los métodos que se han declarado, y se ejecutará aquél en el que haya una coincidencia entre ambos tipos de parámetros.

Ten en cuenta que no es válida la sobrecarga de dos métodos cuyos parámetros formales se diferencian únicamente en su modo, así como tampoco en funciones que se diferencien únicamente en el valor de retorno.

Estos dos ejemplos son correctos, ya que se diferencian en el número de parámetros:

```
MEMBER PROCEDURE setNombre(Nombre VARCHAR2)
MEMBER PROCEDURE setNombre(Nombre VARCHAR2, Apellidos VARCHAR2)
```

No es válido crear un nuevo método en el que se utilicen los mismos tipos de parámetros formales aunque se diferencien los nombres de los parámetros:

```
MEMBER PROCEDURE setNombre(Name VARCHAR2, Surname VARCHAR2)
```

**¿Son correctas las siguientes declaraciones de métodos?**

```
MEMBER FUNCTION getResultado(Valor VARCHAR2)
MEMBER FUNCTION getResultado(Valor INTEGER)
```

(•) **Verdadero** ( ) Falso

*Aunque ambos métodos tengan el mismo nombre, incluso aunque el nombre de los parámetros sean iguales, los tipos de los parámetros son diferentes, por lo que el método getResultado está sobrecargado correctamente*

## 3.5.- Métodos Constructores.

Cada tipo de objeto tiene un método constructor, que se trata de una **función con el mismo nombre que el tipo de objeto** y que se encarga de **inicializar los atributos y retornar una nueva instancia** de ese tipo de objeto.

Oracle crea un **método constructor por defecto** para cada tipo de objeto declarado, cuyos parámetros formales coinciden en orden, nombres y tipos de datos con los atributos del tipo de objeto.

También puedes declarar **tus propios métodos constructores**, reescribiendo ese método declarado por el sistema, o bien, definiendo un nuevo método con otros parámetros. Una de las ventajas de crear un nuevo método constructor personalizado es que se puede hacer una verificación de que los datos que se van a asignar a los atributos son correctos (por ejemplo, que cumplen una determinada restricción).

Si deseas reemplazar el método constructor por defecto, debes utilizar la sentencia `CONSTRUCTOR FUNCTION` seguida del nombre del tipo de objeto en el que se encuentra (recuerda que los métodos constructores tienen el mismo nombre que el tipo de objeto). A continuación debes indicar los parámetros que sean necesarios de la manera habitual. Por último, debes indicar que el valor de retorno de la función es el propio objeto utilizando la cláusula `RETURN SELF AS RESULT`.

Puedes crear **varios métodos constructores** siguiendo las restricciones indicadas para la sobrecarga de métodos.

En el siguiente ejemplo puedes ver la declaración y el cuerpo de un método constructor para el tipo de objeto Usuario. Como puedes comprobar, utiliza dos parámetros: login y crédito inicial. El cuerpo del método realiza un pequeño control para que en caso de que el crédito indicado sea negativo, se deje en cero:

```
CONSTRUCTOR FUNCTION Usuario(login VARCHAR2, credito NUMBER)
   RETURN SELF AS RESULT

CREATE OR REPLACE TYPE BODY Usuario AS
   CONSTRUCTOR FUNCTION Usuario(login VARCHAR2, credito NUMBER)
      RETURN SELF AS RESULT
   IS
      BEGIN
         IF (credito >= 0) THEN
            SELF.credito := credito;
         ELSE
            SELF.credito := 0;
         END IF;
         RETURN;
      END;
END;
```

# 4.- Utilización de objetos.

**Caso práctico**

*__Ada__ ha observado que __Juan__ y __Ana__ están muy volcados en el desarrollo de algo. Ellos le cuentan que están realizando pruebas para modificar sus bases de datos relacionales para darles funcionalidades orientadas a objetos.*

*De momento han realizado la declaración de los tipos de datos objeto, pero no pueden enseñarle a __Ada__ su funcionamiento práctico, puesto que todavía no han creado objetos de esos tipos que ya tienen creados.*

*Le piden un poco de paciencia, porque pronto podrán enseñarle los beneficios que puede reportar esta técnica de trabajo con bases de datos y para el desarrollo de aplicaciones.*

Una vez que dispones de los conocimientos necesarios para tener declarado un tipo de dato objeto, vamos a conocer la forma de utilizar los objetos que vayas a crear de ese tipo.

En los siguientes apartados vas a ver cómo puedes **declarar variables** que permitan almacenar objetos, darle unos **valores iniciales a sus atributos**, acceder al **contenido** de dichos atributos en cualquier momento, y **llamar a los métodos** que ofrece el tipo de objeto utilizado.

## 4.1.- Declaración de objetos.

Una vez que el tipo de objeto ha sido definido, éste puede ser utilizado para **declarar variables de objetos** de ese tipo en cualquier bloque PL/SQL, subprograma o paquete. Ese tipo de objeto lo puedes utilizar como tipo de dato para una variable, atributo, elemento de una tabla, parámetro formal, o resultado de una función, de igual manera que se utilizan los tipos de datos habituales como `VARCHAR` o `NUMBER`.

Por ejemplo, para declarar una variable denominada u1, que va a permitir almacenar un objeto del tipo Usuario, debes hacer la siguiente declaración:

```
u1 Usuario;
```

En la declaración de cualquier **procedimiento o función**, incluidos los métodos del mismo tipo de objeto o de otro, se puede utilizar el tipo de dato objeto definido para indicar que debe **pasarse como parámetro un objeto** de dicho tipo en la llamada. Por ejemplo pensemos en un procedimiento al que se le debe pasar como parámentro un objeto del tipo Usuario:

```
PROCEDURE setUsuario(u IN Usuario)
```

La llamada a este método se realizaría utilizando como parámetro un objeto, como el que podemos tener en la variable declarada anteriormente:

```
setUsuario(u1);
```

De manera semejante una función puede **retornar objetos**:

```
FUNCTION getUsuario(codigo INTEGER) RETURN Usuario
```

Los objetos se crean durante la ejecución del código como instancias del tipo de objeto, y cada uno de ellos puede contener valores diferentes en sus atributos.

El **ámbito de los objetos** sigue las mismas reglas habituales en PL/SQL, es decir, en un bloque o subprograma los objetos son creados (instanciados) cuando se entra en dicho bloque o subprograma y se destruyen automáticamente cuando se sale de ellos. En un paquete, los objetos son instanciados en el momento de hacer referencia al paquete y dejan de existir cuando se finaliza la sesión en la base de datos.

**Suponiendo que tienes declarado el tipo de objeto Factura. ¿Cuál de las siguientes declaraciones de variable para guardar un objeto de ese tipo es correcta?**

- Factura factura1;
- factura1 := Factura;
- **factura1 Factura;**

## 4.2.- Inicialización de objetos.

Para **crear o instanciar un objeto** de un determinado tipo de objeto, debes hacer una **llamada a su método constructor**. Esto lo puedes realizar empleando la **instrucción** `NEW` seguido del nombre del tipo de objeto como una llamada a una función en la que se indican como parámetros los valores que se desean asignar a los atributos inicialmente. En una asignación también puedes optar por hacer eso mismo omitiendo la palabra `NEW`.

El **orden de los parámetros** debe coincidir con el orden en el que están declarados los atributos, así como los tipos de datos. El formato sería como el siguiente:

```
variable_objeto := NEW Nombre_Tipo_Objeto (valor_atributo1, valor_atributo2, ...);
```

Por ejemplo, en el caso del tipo de objeto Usuario:

```
u1 := NEW Usuario('luitom64', 'LUIS ', 'TOMAS BRUÑA', '24/10/07', 100);
```

En ese momento se **inicializa el objeto**. Hasta que no se inicializa el objeto llamando a su constructor, el objeto tiene el valor `NULL`.

Es habitual inicializar los objetos en su declaración.

```
u1 Usuario := NEW Usuario('luitom64', 'LUIS ', 'TOMAS BRUÑA', '24/10/07', 100);
```

La llamada al método constructor se puede realizar en cualquier lugar en el que se puede hacer una llamada a una función de la forma habitual. Por ejemplo, la llamada al método constructor puede ser utilizada **como parte de una expresión**.

Los **valores de los parámetros** que se pasan al constructor cuando se hace la llamada, son asignados a los atributos del objeto que está siendo creado. Si la llamada es al método constructor que incorpora Oracle por defecto, debes indicar un parámetro para cada atributo, en el mismo orden en que están declarados los atributos. Ten en cuenta que los atributos, en contra de lo que ocurre con variables y constantes, no pueden tener valores por defecto asignados en su declaración. Por tanto, los valores que se desee que tengan inicialmente los atributos de un objeto instanciado deben indicarse como parámetros en la llamada al método constructor.

Existe la posibilidad de **utilizar los nombres de los parámetros formales** en la llamada al método constructor, en lugar de utilizar el modelo posicional de los parámetros. De esta manera no es obligatorio respetar el orden en el que se encuentran los parámetros reales respecto a los parámetros formales, que como se ha comentado antes coincide con el orden de los atributos.

```
DECLARE
   u1 Usuario;
BEGIN
   u1 := NEW Usuario('user1', -10);
   /* Se mostrará el crédito como cero, al intentar asignar un crédito negativo */
   dbms_output.put_line(u1.credito);
END;
/
```

**¿Cuál de las siguientes inicializaciones de objetos es correcta para el tipo de objeto Factura, suponiendo que dispone de los atributos número (INTEGER), nombre (VARCHAR2) e importe (NUMBER)?**

- **factura1 := NEW Factura(3, 'Juan Álvarez', 30.50);**
- factura1 = Factura(3, 'Juan Álvarez', 30.50);
- factura1 := NEW Factura('Juan Álvarez', 3, 30.50);
- factura1 := NEW (3, 'Juan Álvarez', 30.50);

## 4.3.- Acceso a los atributos de objetos.

Para hacer referencia a un atributo de un objeto debes utilizar el **nombre de dicho atributo**, **utilizando el punto** para acceder al valor que contiene o bien para modificarlo. Antes debe ir **precedido del objeto** cuyo atributo deseas conocer o modificar.

```
nombre_objeto.nombre_atributo
```

Por ejemplo, la consulta del valor de un atributo puede utilizarse como parte de una asignación o como parámetro en la llamada a una función:

```
unNombre := usuario1.nombre;
dbms_output.put_line(usuario1.nombre);
```

La **modificación del valor** contenido en el atributo puede ser similar a la siguiente:

```
usuario1.nombre:= 'Nuevo Nombre';
```

Los nombres de los atributos pueden ser encadenados, lo que permite el acceso a atributos de tipos de objetos anidados. Por ejemplo, si el objeto sitio1 tiene un atributo del tipo de objeto Usuario, se accedería al atributo del nombre del usuario con:

```
sitio1.usuario1.nombre
```

Si se utiliza en una expresión el acceso a un atributo de un objeto que **no ha sido inicializado**, se evalúa como `NULL`. Por otro lado, si se intenta asignar valores a un objeto no inicializado, éste lanza una excepción `ACCESS INTO NULL`.

Para comprobar si un objeto es `NULL` se puede utilizar el operador de comparación `IS NULL` con el que se obtiene el valor `TRUE` si es así.

De manera similar, al intentar hacer una llamada a un método de un objeto que no ha sido inicializado, se lanza una excepción `NULL_SELF_DISPATCH`. Si se pasa como parámetro de tipo IN, los atributos del objeto `NULL` se evalúan como `NULL`, y si el parámetro es de tipo `OUT` o `IN OUT` lanza una excepción al intentar modificar el valor de sus atributos.

**¿Cuál de las siguientes expresiones es correcta para asignar el valor 50 al atributo importe del objeto factura1?**

- **factura1.importe := 50;**
- importe.factura1 := 50;
- 50 := factura1.importe;

## 4.4.- Llamada a los métodos de los objetos.

Al igual que las llamadas a subprogramas, puedes invocar a los métodos de un tipo de objetos **utilizando un punto entre el nombre del objeto y el del método**. Los parámetros reales que se pasen al método se indicarán separados por comas, entre paréntesis, después del nombre del método.

```
usuario1.setNombreCompleto('Juan', 'García Fernández');
```

Si el método **no tiene parámetros**, se indicará la lista de parámetros reales vacía (sólo con los paréntesis), aunque se pueden omitir los paréntesis.

```
credito := usuario1.getCredito();
```

Las llamadas a métodos pueden **encadenarse**, en cuyo caso, el orden de ejecución de los métodos es de derecha a izquierda. Se debe tener en cuenta que el método de la izquierda **debe retornar un objeto** del tipo correspondiente al método de la derecha.

Por ejemplo, si dispones de un objeto sitio1 que tiene declarado un método getUsuario el cual retorna un objeto del tipo Usuario, puedes realizar con ese valor retornado una llamada a un método del tipo de objeto Usuario:

```
sitio1.getUsuario.setNombreCompleto('Juan', 'García Fernández');
```

Los **métodos** `MEMBER` son invocados utilizando una instancia del tipo de objeto:

```
nombre_objeto.metodo()
```

En cambio, los **métodos** `STATIC` se invocan usando el tipo de objeto, en lugar de una de sus instancias:

```
nombre_tipo_objeto.metodo()
```

**¿Cuál de las siguientes llamadas al método getImporte es correcto para el objeto factura1?**

- valor := getImporte.factura1();
- **valor := factura1.getImporte();**
- valor := getImporte().factura1;

## 4.5.- Herencia.

El lenguaje PL/SQL admite la herencia simple de tipos de objetos, mediante la cual, puedes **definir subtipos** de los tipos de objeto. Estos subtipos, o tipos heredados, **contienen todos los atributos y métodos del tipo padre**, pero además pueden contener **atributos y métodos adicionales**, o incluso sobreescribir métodos del tipo padre.

Para indicar que un tipo de objeto es **heredado** de otro hay que usar la **palabra reservada** `UNDER`, y además hay que tener en cuenta que el tipo de objeto del que **hereda** debe tener la **propiedad** `NOT FINAL`. Por defecto, los tipos de objeto se declaran como `FINAL`, es decir, que no se puede crear un tipo de objeto que herede de él.

Si no se indica lo contrario, siempre se pueden crear objetos (instancias) de los tipos de objeto declarados. Indicando la opción `NOT INSTANTIABLE` puedes declarar tipos de objeto de los que **no se pueden crear objetos**. Estos tipos tendrán la función de ser padres de otros tipos de objeto.

En el siguiente ejemplo puedes ver cómo se crea el tipo de objeto Persona, que se utilizará heredado en el tipo de objeto UsuarioPersona. De esta manera, este último tendrá los atributos de Persona

más los atributos declarados en UsuarioPersona. En la creación del objeto puedes observar que se deben asignar los valores para todos los atributos, incluyendo los heredados.

```
CREATE TYPE Persona AS OBJECT (
   nombre VARCHAR2(20),
   apellidos VARCHAR2(30)
) NOT FINAL;
/

CREATE TYPE UsuarioPersona UNDER Persona (
   login VARCHAR(30),
   f_ingreso DATE,
   credito NUMBER
);
/

DECLARE
   u1 UsuarioPersona;
BEGIN
   u1 := NEW UsuarioPersona('nombre1', 'apellidos1', 'user1', '01/01/2001', 100);
   dbms output.put line(u1.nombre);
END;
/
```

**Un tipo de objeto que se ha declarado como hijo de otro, ¿hereda los atributos y métodos del tipo de objeto padre?**

(•) Verdadero ( ) Falso

# 5.- Métodos MAP y ORDER.

**Caso práctico**

***Juan*** *se ha dado cuenta de que va a tener un problema cuando necesite realizar comparaciones y órdenes entre objetos del mismo tipo.*

*Cuando tenga una serie de objetos de tipo Usuario, y necesite realizar una operación de ordenación entre ellos, ¿cómo debe hacerlo? Podría indicar que se realizaran las consultas de forma ordenada en función de alguno de los atributos del tipo de objeto. Ha estudiado el funcionamiento de los métodos MAP y ORDER, y ha comprobado que son los que le van a ofrecer un mecanismo sencillo para ello.*

Las instancias de un tipo de objeto no tienen un orden predefinido. Si deseas establecer un orden en ellos, con el fin de **hacer una ordenación o una comparación, debes crear un método** MAP.

Por ejemplo, si haces una comparación entre dos objetos Usuario, y deseas saber si uno es mayor que otro, ¿en base a qué criterio se hace esa comparación? ¿Por el orden alfabético de apellidos y nombre? ¿Por la fecha de alta? ¿Por el crédito? Hay que establecer con un método `MAP` cuál va a ser el valor que se va a utilizar en las comparaciones.

Para crear un método MAP debes declarar un método que **retorne el valor que se va a utilizar para hacer las comparaciones**. El método que declares para ello debe empezar su declaración con la palabra MAP:

```
CREATE OR REPLACE TYPE Usuario AS OBJECT (
   login VARCHAR2(30),
   nombre VARCHAR2(30),
   apellidos VARCHAR2(40),
   f_ingreso DATE,
   credito NUMBER,
   MAP MEMBER FUNCTION ordenarUsuario RETURN VARCHAR2
);
/
```

En el cuerpo del método se debe retornar el valor que se utilizará para realizar las comparaciones entre las instancias del tipo de objeto. Por ejemplo, si se quiere establecer que las comparaciones entre objetos del tipo Usuario se realice considerando el orden alfabético habitual de apellidos y nombre:

```
CREATE OR REPLACE TYPE BODY Usuario AS
   MAP MEMBER FUNCTION ordenarUsuario RETURN VARCHAR2 IS
   BEGIN
      RETURN (apellidos || ' ' || nombre);
   END ordenarUsuario;
END;
/
```

El lenguaje PL/SQL utiliza los métodos `MAP` para evaluar expresiones lógicas que resultan valores booleanos como objeto1 > objeto2, y para realizar las comparaciones implícitas en las cláusulas `DISTINCT`, `GROUP BY` y `ORDER BY`.

Cada tipo de objeto **sólo puede tener un método** `MAP` **declarado**, y sólo puede retornar alguno de los siguientes tipos: `DATE`, `NUMBER`, `VARCHAR2`, `CHARACTER` o `REAL`.

## 5.1.- Métodos ORDER.

De forma **similar al método** MAP, puedes declarar en cualquier tipo de objeto un método `ORDER` que te permitirá **establecer un orden** entre los objetos instanciados de dicho tipo.

Cada tipo de objeto **sólo puede contener un método** `ORDER`, el cual debe **retornar un valor numérico que permita establecer el orden entre los objetos**. Si deseas que un objeto sea menor que otro puedes retornar, por ejemplo, el valor -1. Si vas a determinar que sean iguales, devuelve 0, y si va a

ser mayor, retorna 1. De esta manera, considerando el valor retornado por el método `ORDER`, se puede establecer si un objeto es menor, igual o mayor que otro en el momento de hacer una ordenación entre una serie de objetos del mismo tipo.

Para declarar qué método va a realizar esta operación, debes indicar la palabra `ORDER` delante de su declaración. Debes tener en cuenta que va a **retornar un valor numérico (`INTEGER`)**, y que necesita que se indique un **parámetro que será del mismo tipo de objeto**. El objeto que se indique en ese parámetro será el que se compare con el objeto que utilice este método.

En el siguiente ejemplo, el sistema que se va a establecer para ordenar a los usuarios se realiza en función de los dígitos que se encuentran a partir de la posición 7 del atributo login.

```
CREATE OR REPLACE TYPE BODY Usuario AS
   ORDER MEMBER FUNCTION ordenUsuario(u Usuario) RETURN INTEGER IS
   BEGIN
      /* La función substr obtiene una subcadena desde la posición indicada hasta el final*/
      IF substr(SELF.login, 7) < substr(u.login, 7) THEN
         RETURN -1;
      ELSIF substr(SELF.login, 7) > substr(u.login, 7) THEN
         RETURN 1;
      ELSE
         RETURN 0;
      END IF;
   END;
END;
```

Con ese ejemplo, el usuario con login 'luitom64' se considera mayor que el usuario 'caragu62', ya que '64' es mayor que '62'.

El método debe retornar un número negativo, cero, o un número positivo que significará que el objeto que utiliza el método (`SELF`) es menor, igual, o mayor que el objeto pasado por parámetro.
Debes tener en cuenta que puedes declarar **un método `MAP` o un método `ORDER`, pero no los dos**.
Cuando se vaya a ordenar o mezclar un alto número de objetos, es preferible usar un método `MAP`, ya que en esos casos un método `ORDER` es menos eficiente.

**¿Los métodos MAP sólo sirven para evaluar expresiones lógicas que resultan valores booleanos?**

○ Verdadero ◉ **Falso**

*También se pueden utilizar en consultas con las opciones DISTINCT, GROUP BY y ORDER BY*

# 6.- Tipos de datos colección.

**Caso práctico**

*__Ana__ ha estudiado los vectores y matrices que se usan en varios lenguajes de programación. Le pregunta a __Juan__ si con las bases de datos objeto-relacionales se puede utilizar algo parecido para almacenar los objetos instanciados. Él le dice que para eso existen las colecciones.*

*Para que puedas almacenar en memoria un conjunto de datos de un determinado tipo, la base de datos de Oracle te ofrece los tipos de datos __colección__.*

Una **colección** es un conjunto de elementos del mismo tipo. Puede comparase con los **vectores y matrices** que se utilizan en muchos lenguajes de programación. En este caso, las colecciones sólo pueden tener **una dimensión** y los elementos se indexan mediante un valor de tipo numérico o cadena de caracteres.

La base de datos de Oracle proporciona los tipos `VARRAY` y `NESTED TABLE` (tabla anidada) como tipos de datos colección.

- ✓ Un `VARRAY` es una colección de elementos a la que se le establece una dimensión máxima que debe indicarse al declararla. Al tener una longitud fija, la eliminación de elementos no ahorra espacio en la memoria del ordenador.
- ✓ Una `NESTED TABLE` (**tabla anidada**) puede almacenar cualquier número de elementos. Tienen, por tanto, un tamaño dinámico, y no tienen que existir forzosamente valores para todas las posiciones de la colección.
- ✓ Una variación de las tablas anidadas son los **arrays asociativos**, que utilizan valores arbitrarios para sus índices. En este caso, los índices no tienen que ser necesariamente consecutivos.

Cuando necesites almacenar un número fijo de elementos, o hacer un recorrido entre los elementos de forma ordenada, o si necesitas obtener y manipular toda la colección como un valor, deberías utilizar el tipo `VARRAY`.

En cambio, si necesitas ejecutar consultas sobre una colección de manera eficiente, o manipular un número arbitrario de elementos, o bien realizar operaciones de inserción, actualización o borrado de forma masiva, deberías usar una`NESTED TABLE`.

Las colecciones pueden ser declaradas como una instrucción SQL o en el bloque de declaraciones de un programa PL/SQL. El tipo de dato de los elementos que puede contener una colección declarada en PL/SQL es cualquier tipo de dato PL/SQL, excepto REF CURSOR. Los elementos de las colecciones declaradas en SQL, además **no pueden ser de los tipos**: `BINARY_INTEGER`, `PLS_INTEGER`, `BOOLEAN`, `LONG`, `LONG RAW`, `NATURAL`, `NATURALN`, `POSITIVE`,`POSITIVEN`, `REF CURSOR`, `SIGNTYPE`, `STRING`.

Cualquier tipo de objetos declarado previamente puede ser utilizado como tipo de elemento para una colección.

Una tabla de la base de datos puede contener **columnas que sean colecciones**. Sobre una tabla que contiene colecciones se podrán realizar operaciones de consulta y manipulación de datos de la misma manera que se realiza con tablas con los tipos de datos habituales.

En este documento puedes encontrar más información sobre los tipos de datos colección entre otros conceptos de las bases de datos objetos-relacionales:
Anexo I - Soporte objeto-relacional en Oracle 8.

**¿Qué tipo de colección tiene un tamaño fijo?**

(•) VARRAY ( ) NESTED TABLE ( ) Array Asociativo

## 6.1.- Declaración y uso de colecciones.

La **declaración** de estos tipos de colecciones sigue el formato siguiente:

```
TYPE nombre_tipo IS VARRAY (tamaño max) OF tipo elemento;
TYPE nombre_tipo IS TABLE OF tipo_elemento;
TYPE nombre_tipo IS TABLE OF tipo_elemento INDEX BY tipo_índice;
```

Donde nombre_tipo representa el nombre de la colección, tamaño_max es el número máximo de elementos que podrá contener la colección, y tipo_elemento es el tipo de dato de los elementos que forman la colección. El tipo de elemento utilizado puede ser también cualquier tipo de objetos declarado previamente.

En caso de que la declaración se haga en SQL, fuera de un subprograma PL/SQL, se debe declarar con el formato `CREATE [OR REPLACE] TYPE`:

```
CREATE TYPE nombre_tipo IS ...
```

El tipo_índice representa el tipo de dato que se va a utilizar para el índice. Puede ser `PLS_INTEGER`, `BINARY INTEGER` o `VARCHAR2`. En este último tipo se debe indicar entre paréntesis el tamaño que se va a utilizar para el índice, por ejemplo, `VARCHAR2(5)`.

Hasta que no sea inicializada una colección, ésta es `NULL`. Para **inicializar** una colección debes utilizar el constructor, que es una función con el mismo nombre que la colección. A esta función se le pasa como parámetros los valores iniciales de la colección. Por ejemplo:

```
DECLARE
   TYPE Colores IS TABLE OF VARCHAR(10);
   misColores Colores;
BEGIN
   misColores := Colores('Rojo', 'Naranja', 'Amarillo', 'Verde', 'Azul');
END;
```

La inicialización se puede realizar en el bloque de código del programa, o bien, directamente en el bloque de declaraciones como puedes ver en este ejemplo:

```
DECLARE
   TYPE Colores IS TABLE OF VARCHAR(10);
   misColores Colores := Colores('Rojo', 'Naranja', 'Amarillo', 'Verde', 'Azul');
```

Para **obtener uno de los elementos** de la colección o modificar su contenido debes indicar el nombre de la colección seguido, entre paréntesis, del índice que ocupa el elemento deseado. Tanto en los `VARRAY` como en `NESTED TABLE`, el primer elemento tiene el índice 1.

Por ejemplo, para mostrar en pantalla el segundo elemento ('Naranja') de la colección Colores:

```
dbms_output.put_line(misColores(2));
```

En el siguiente ejemplo se modifica el contenido de la posición 3:

```
misColores(3) := 'Gris';
```

En el siguiente ejemplo puedes comprobar cómo pueden utilizarse las colecciones para almacenar sus datos en una tabla de la base de datos, así como la utilización con sentencias de consulta y manipulación de los datos de la colección que se encuentra en la tabla.

```
CREATE TYPE ListaColores AS TABLE OF VARCHAR2(20);
/
CREATE TABLE flores (nombre VARCHAR2(20), coloresFlor ListaColores)
   NESTED TABLE coloresFlor STORE AS colores_tab;

DECLARE
   colores ListaColores;
BEGIN
   INSERT INTO flores VALUES('Rosa', ListaColores('Rojo','Amarillo','Blanco'));
   colores := ListaColores('Rojo','Amarillo','Blanco','Rosa Claro');
   UPDATE flores SET coloresFlor = colores WHERE nombre = 'Rosa';
```

```
   SELECT coloresFlor INTO colores FROM flores WHERE nombre = 'Rosa';
END;/
```

# 7.- Tablas de objetos.

**Caso práctico**

*Aunque **Ana** ya ha aprendido a almacenar objetos en memoria gracias a las colecciones de objetos, necesita almacenarlos de manera persistente en una base de datos. **Juan** va a enseñarle que puede realizarlo de manera muy parecida a la gestión de tablas que ya conoce, en la que se usan los tipos de datos habituales de las bases de datos.*

Después de haber visto que un grupo de objetos se puede almacenar en memoria mediante colecciones, vas a ver en este apartado que también se pueden **almacenar los objetos en tablas** de igual manera que los tipos de datos habituales de las bases de datos.

Los tipos de datos objetos se pueden usar para formar una tabla exclusivamente formado por elementos de ese tipo, o bien, para usarse como un tipo de columna más entre otras columnas de otros tipos de datos.

En caso de que desees crear una **tabla formada exclusivamente por un determinado tipo de dato objeto**, (tabla de objetos) debes utilizar la sentencia `CREATE TABLE` junto con el tipo de objeto de la siguiente manera:

```
CREATE TABLE NombreTabla OF TipoObjeto;
```

Siendo `NombreTabla` el nombre que deseas dar a la tabla que va a almacenar los objetos del tipo TipoObjeto. Por ejemplo, para crear la tabla UsuariosObj, que almacene objetos del tipo Usuario:

```
CREATE TABLE UsuariosObj OF Usuario;
```

Debes tener en cuenta que si una tabla hace uso de un tipo de objeto, **no podrás eliminar ni modificar la estructura de dicho tipo de objeto**. Por tanto, desde el momento en que el tipo de objeto sea utilizado en una tabla, no podrás volver a definirlo.

Para poder crear este ejemplo previamente debes tener declarado el tipo de objeto Usuario, que se ha utilizado en apartados anteriores.

Al crear una tabla de esta manera, estamos consiguiendo que podamos almacenar objetos del tipo Usuario en una tabla de la base de datos, quedando así sus datos **persistentes** mientras no sean eliminados de la tabla. Anteriormente hemos instanciado objetos que se han guardado en variables, por lo que al terminar la ejecución, los objetos, y la información que contienen, desaparecen. Si esos objetos se almacenan en tablas no desaparecen hasta que se eliminen de la tabla en la que se encuentren.

Cuando se instancia un objeto con el fin de almacenarlo en una tabla, dicho objeto no tiene identidad fuera de la tabla de la base de datos. Sin embargo, el tipo de objeto existe independientemente de cualquier tabla, y puede ser usado para crear objetos en cualquier modo.

Las tablas que sólo contienen filas con objetos, reciben el nombre de **tablas de objetos**.

En la siguiente imagen se muestra el contenido de la tabla que incluye dos filas de objetos de tipo Usuario. Observa que los atributos del tipo de objeto se muestran como si fueran las columnas de la tabla:


En este documento puedes encontrar información general sobre las bases de datos objeto-relacionales, con algunos ejemplos de creación de tablas de objetos:
Bases de datos orientadas a objetos. (0.19 MB)

## 7.1.- Tablas con columnas tipo objeto.

Puedes usar cualquier tipo de objeto, que hayas declarado previamente, para utilizarlo como un **tipo de dato de una columna más en una tabla** de la base de datos. Así, una vez creada la tabla, puedes utilizar cualquiera de las sentencias SQL para insertar un objeto, seleccionar sus atributos y actualizar sus datos.

Para crear una tabla en el que alguno de sus columnas sea un tipo de objeto, simplemente debes hacerlo como si fuera una columna como las que has utilizado hasta ahora, pero en el tipo de dato debes especificar el tipo de objeto.

Por ejemplo, podemos crear una tabla que contenga, entre otras columnas, una columna de objetos del tipo Usuario que hemos utilizado anteriormente.

```
CREATE TABLE Gente (
   dni VARCHAR2(10),
   unUsuario Usuario,
   partidasJugadas SMALLINT
);
```

Como puedes comprobar en la siguiente imagen, los datos del campo unUsuario se muestran como integrantes de cada objeto Usuario, a diferencia de la tabla de objetos que has visto en el apartado anterior. Ahora todos los atributos del tipo de objeto Usuario no se muestran como si fueran varias columnas de la tabla, sino que forman parte de una única columna.

**En las tablas con columnas de tipo objeto, ¿los atributos se muestran como columnas de la tabla?**

(•) Falso ( ) Verdadero

*En las tablas con columnas de tipo objeto, se muestran los atributos como parte del objeto*

## 7.2.- Uso de la sentencia Select.

De manera similar a las consultas que has realizado sobre tablas sin tipos de objetos, **puedes utilizar la sentencia** `SELECT` para obtener datos de las filas almacenadas en tablas de objetos o tablas con columnas de tipos de objetos.

El uso más sencillo sería para mostrar todas las filas contenidas en la tabla:

```
SELECT * FROM NombreTabla;
```

Como puedes apreciar en la imagen, la tabla que forme parte de la consulta puede ser una tabla de objetos (como la tabla UsuariosObj), o una tabla que contiene columnas de tipos de objetos (como la tabla Gente).

En las sentencias `SELECT` que utilices con objetos, puedes incluir cualquiera de las **cláusulas y funciones de agrupamiento** que has aprendido para la sentencia `SELECT` que has usado anteriormente con las tablas que contienen columnas de tipos básicos. Por ejemplo, puedes utilizar: `SUM`, `MAX`, `WHERE`, `ORDER`, `JOIN`, etc.

Es habitual utilizar **alias** para hacer referencia al nombre de la tabla. Observa, por ejemplo, la siguiente consulta, en la que se desea obtener el nombre y los apellidos de los usuarios que tienen algo de crédito:

```
SELECT u.nombre, u.apellidos FROM UsuariosObj u WHERE u.credito > 0
```

Si se trata de una **tabla con columnas de tipo objeto**, el acceso a los atributos del objeto se debe realizar indicando previamente el nombre asignado a la columna que contiene los objetos:

```
SELECT g.unUsuario.nombre, g.unUsuario.apellidos FROM Gente g;
```

En este documento puedes encontrar información general sobre las bases de datos objeto-relacionales, con algunos ejemplos de uso de la sentencia `SELECT` con tablas de objetos:
Anexo II - Tipos de Objeto en PL/SQL.

## 7.3.- Inserción de objetos.

Evidentemente, no te servirá de nada una tabla que pueda contener objetos sino conocemos la manera de insertar objetos en la tabla.

La manera que tienes para ello **es la misma** que has utilizado para introducir datos de cualquier tipo habitual en las tablas de la base de datos: usando la sentencia `INSERT` de SQL.

En las tablas habituales, cuando querías añadir una fila a una tabla que tenía un campo `VARCHAR` le suministrabas a la sentencia `INSERT` un dato de ese tipo. Pues si la tabla es de un determinado tipo de objetos, o si posee un campo de un determinado tipo de objeto, tendrás que suministrar a la sentencia `INSERT` un objeto instanciado de su tipo de objeto correspondiente.

Por tanto, si queremos insertar un Usuario en la tabla Gente que hemos creado en el apartado anterior, previamente debemos crear el objeto o los objetos que se deseen insertar. A continuación podremos utilizarlos dentro de la sentencia INSERT como si fueran valores simplemente.

```
DECLARE
   u1 Usuario;
   u2 Usuario;
BEGIN
   u1 := NEW Usuario('luitom64', 'LUIS', 'TOMAS BRUNA', '24/10/2007', 50);
   u2 := NEW Usuario('caragu72', 'CARLOS', 'AGUDO SEGURA', '06/07/2007', 100);
   INSERT INTO UsuariosObj VALUES (u1);
   INSERT INTO UsuariosObj VALUES (u2);
END;
```

De una manera más directa, puedes crear el objeto dentro de la sentencia `INSERT` directamente, sin necesidad de guardar el objeto previamente en una variable:

```
INSERT INTO UsuariosObj VALUES (Usuario('luitom64', 'LUIS', 'TOMAS BRUNA', '24/10/2007', 50));
```

Podrás comprobar los resultados haciendo una consulta `SELECT` sobre la tabla de la manera habitual:

```
SELECT * FROM UsuariosObj;
```

De manera similar puedes realizar la **inserción de filas en tablas con columnas de tipo objeto**. Por ejemplo, para la tabla Gente que posee entre sus columnas, una de tipo objeto Usuario, podríamos usar el formato de instanciar el objeto directamente en la sentencia `INSERT`, o bien, indicar una variable que almacena un objeto que se ha instanciado anteriormente:

```
INSERT INTO Gente VALUES ('22900970P', Usuario('luitom64', 'LUIS', 'TOMAS BRUNA',
'24/10/2007', 50), 54);

INSERT INTO Gente VALUES ('62603088D', u2, 21);
```

## 7.4.- Modificación de objetos.

Si deseas modificar un objeto almacenado en una tabla tan sólo tienes que utilizar las **mismas sentencias SQL** que disponías para modificar registros de una tabla. ¿Recuerdas la sentencia `UPDATE`? Ahora puedes volver a utilizarla para modificar también los objetos de la tabla, de igual manera que cualquier otro tipo de dato.

Hay una pequeña diferencia en la forma de especificar los nombre de los campos afectados, en función del tipo de tabla: según sea una tabla de objetos, o bien una tabla con alguna columna de tipo objeto.

Si se trata de una tabla de objetos, se hará referencia a los atributos de los objetos justo detrás del nombre asignado a la tabla. Sería algo similar al formato siguiente:

```
UPDATE NombreTabla
   SET NombreTabla.atributoModificado = nuevoValor
   WHERE NombreTabla.atributoBusqueda = valorBusqueda;
```

Continuando con el ejemplo empleado anteriormente, vamos a suponer que deseas modificar los datos de un determinado usuario. Por ejemplo, modifiquemos el crédito del usuario identificado por el login 'luitom64', asignándole el valor 0.

```
UPDATE UsuariosObj
   SET UsuariosObj.credito = 0
   WHERE UsuariosObj.login = 'luitom64';
```

Es muy habitual abreviar el nombre de la tabla con un **alias**:

```
UPDATE UsuariosObj u
   SET u.credito = 0
   WHERE u.login = 'luitom64';
```

Pero no sólo puedes cambiar el valor de un determinado atributo del objeto. Puedes **cambiar un objeto por otro** como puedes ver en el siguiente ejemplo, en el que se sustituye el usuario con login 'caragu72' por otro usuario nuevo.

```
UPDATE UsuariosObj u SET u = Usuario('juaesc82', 'JUAN', 'ESCUDERO LARRASA', '10/04/2011', 0)
WHERE u.login = 'caragu72';
```

Si se trata de una tabla con columnas de tipo objeto, se debe hacer referencia al nombre de la columna que contiene los objetos:

```
UPDATE NombreTabla
   SET NombreTabla.colObjeto.atributoModificado = nuevoValor
   WHERE NombreTabla.colObjeto.atributoBusqueda = valorBusqueda;
```

A continuación puedes ver un ejemplo de actualización de datos de la tabla que se había creado con una columna del tipo de objeto Usuario. Recuerda que a la columna en la que se almacenaban los objetos de tipo Usuario se le había asignado el nombre unUsuario:

```
UPDATE Gente g
   SET g.unUsuario.credito = 0
   WHERE g.unUsuario.login = 'luitom64';
```

O bien, puedes cambiar todo un objeto por otro, manteniendo el resto de los datos de la fila sin modificar, como en el siguiente ejemplo, donde los datos de DNI y partidasJugadas no se cambia, sólo se cambia un usuario por otro.

```
UPDATE Gente g
   SET g.unUsuario = Usuario('juaesc82', 'JUAN', 'ESCUDERO LARRASA', '10/04/2011', 0)
   WHERE g.unUsuario.login = 'caragu72';
```

## 7.5.- Borrado de objetos.

Por supuesto, no nos puede faltar una sentencia que nos permita eliminar determinados objetos almacenados en tablas. Al igual que has podido comprobar en las operaciones anteriores, tienes a tu disposición la misma sentencia que has podido utilizar en las operaciones habituales sobre tablas. En este caso de borrado de objetos deberás utilizar la sentencia `DELETE`.

El modo de uso de `DELETE` sobre objetos almacenados en tablas es muy similar al utilizado hasta ahora:

```
DELETE FROM NombreTablaObjetos;
```

Recuerda que si no se indica ninguna condición, se **eliminarán todos** los objetos de la tabla, por lo que suele ser habitual utilizar la sentencia `DELETE` con una condición detrás de la cláusula `WHERE`. Los objetos o filas de la tabla que**cumplan con la condición** indicada serán los que se eliminen.

```
DELETE FROM NombreTablaObjetos WHERE condición;
```

Observa el siguiente ejemplo en el que se borrarán de la tabla UsuariosObj, que es una tabla de objetos, los usuarios cuyo crédito sea 0. Observa que se utiliza un alias para el nombre de la tabla:

```
DELETE FROM UsuariosObj u WHERE u.credito = 0;
```

De manera similar se puede realizar el borrado de filas en tablas en las que alguna de sus columnas son objetos. Puedes comprobarlo con el siguiente ejemplo, donde se utiliza la tabla Gente, en la que una de sus columnas (unUsuario) es del tipo de objeto Usuario que hemos utilizado en otros apartados anteriores.

```
DELETE FROM Gente g WHERE g.unUsuario.credito = 0;
```

Esta sentencia, al igual que las anteriores, se puede **combinar con otras consultas** `SELECT`, de manera que en vez de realizar el borrado sobre una determinada tabla, se haga sobre el resultado de una consulta, o bien que la condición que determina las filas que deben ser eliminadas sea también el resultado de una consulta. Es decir, todo lo aprendido sobre las operaciones de manipulación de datos sobre las tablas habituales, se puede aplicar sobre tablas de tipos de objetos, o tablas con columnas de tipos de objetos.

## 7.6.- Consultas con la función VALUE.

Cuando tengas la necesidad de **hacer referencia a un objeto en lugar de alguno de sus atributos**, puedes utilizar la función `VALUE` junto con el nombre de la tabla de objetos o su alias, **dentro de una sentencia** `SELECT`. Puedes ver a continuación un ejemplo de uso de dicha función para hacer inserciones en otra tabla (Favoritos) del mismo tipo de objetos:

```
INSERT INTO Favoritos SELECT VALUE(u) FROM UsuariosObj u WHERE u.credito >= 100;
```

Esa misma función `VALUE` puedes utilizarla para hacer comparaciones de igualdad entre objetos, por ejemplo, si deseamos obtener datos de los usuarios que se encuentren en las tablas Favoritos y UsuariosObj.

```
SELECT u.login FROM UsuariosObj u JOIN Favoritos f ON VALUE(u)=VALUE(f);
```

Observa la diferencia en el uso cuando se hace la comparación con una columna de tipo de objetos. En ese caso la referencia que se hace a la columna (g.unUsuario) permite obtener directamente un objeto, sin necesidad de utilizar la función `VALUE`.

```
SELECT g.dni FROM Gente g JOIN Favoritos f ON g.unUsuario=VALUE(f);
```

Usando la **cláusula** `INTO` **podrás guardar en variables el objeto obtenido** en las consultas usando la función VALUE. Una vez que tengas asignado el objeto a la variable podrás hacer uso de ella de cualquiera de las formas que has visto anteriormente en la manipulación de objetos. Por ejemplo, puedes acceder a sus atributos, formar parte de asignaciones, etc.

En el siguiente ejemplo se realiza una consulta de la tabla UsuariosObj para obtener un determinado objeto de tipo Usuario. El objeto resultante de la consulta se guarda en la variable u1. Esa variable se utiliza para mostrar en pantalla el nombre del usuario, y para ser asignada a una segunda variable, que contendrá los mismos datos que la primera.

```
DECLARE
   u1 Usuario;
   u2 Usuario;
BEGIN
   SELECT VALUE(u) INTO u1 FROM UsuariosObj u WHERE u.login = 'luitom64';
```

```
   dbms_output.put_line(u1.nombre);
   u2 := u1;
   dbms output.put line(u2.nombre);
END;
```

**En este documento puedes encontrar información general sobre las bases de datos objeto-relacionales, incluyendo información y ejemplos de la función VALUE:**
Anexo III - Bases de datos objeto-relacionales.

## 7.7.- Referencias a objetos.

El paso de objetos a un método resulta ineficiente cuando se trata de objeto de gran tamaño, por lo que es más conveniente **pasar un puntero a dicho objeto**, lo que permite que el método que lo recibe pueda hacer referencia a dicho objeto sin que sea necesario que se pase por completo. Ese puntero es lo que se conoce en Oracle como una **referencia (`REF`)**.

Al compartir un objeto mediante su referencia, **los datos no son duplicados**, por lo que cuando se hace cualquier cambio en los atributos del objeto, se producen en un único lugar.

Cada objeto almacenado en una tabla tiene un **identificador de objeto** que identifica de forma única al objeto guardado en una determinada fila y sirve como una referencia a dicho objeto.

Las referencias se crean utilizando el modificador `REF` delante del tipo de objeto, y se puede usar con variables, parámetros, campos, atributos, e incluso como variables de entrada o salida para sentencias de manipulación de datos en SQL.

```
CREATE OR REPLACE TYPE Partida AS OBJECT (
   codigo INTEGER,
   nombre VARCHAR2(20),
   usuarioCreador REF Usuario
);
/

DECLARE
   u ref REF Usuario;
   p1 Partida;
BEGIN
   SELECT REF(u) INTO u_ref FROM UsuariosObj u WHERE u.login = 'luitom64';
   p1 := NEW Partida(1, 'partida1', u_ref);
END;
/
```

Hay que tener en cuenta que sólo se pueden usar **referencias a tipos de objetos que han sido declarados previamente**. Siguiendo el ejemplo anterior, no se podría declarar el tipo Partida antes que el tipo Usuario, ya que dentro del tipo Partida se utiliza una referencia al tipo Usuario. Por tanto, primero debe estar declarado el tipo Usuario y luego el tipo Partida.

El problema surge cuando tengamos dos tipos que utilizan referencias mutuas. Es decir, un atributo del primer tipo hace referencia a un objeto del segundo tipo, y viceversa. Esto se puede solucionar haciendo una **declaración de tipo anticipada**. Se realiza indicando únicamente el nombre del tipo de objeto que se detallará más adelante:

```
CREATE OR REPLACE TYPE tipo2;
/
CREATE OR REPLACE TYPE tipo1 AS OBJECT (
   tipo2 ref REF tipo2
   /*Declaración del resto de atributos del tipo1*/
);
/
CREATE OR REPLACE TYPE tipo2 AS OBJECT (
   tipo1 ref REF tipo1
   /*Declaración del resto de atributos del tipo2*/
);
```

```
/
```

## 7.8.- Navegación a través de referencias.

Debes tener en cuenta que **no se puede acceder directamente a los atributos de un objeto referenciado que se encuentre almacenado en una tabla**. Para ello, puedes utilizar la función `DEREF`.

Esta función **toma una referencia a un objeto y retorna el valor** de ese objeto.

Vamos a verlo en un ejemplo suponiendo que disponemos de las siguientes variables declaradas

```
u_ref REF Usuario;
u1 Usuario;
```

Si `u ref` hace referencia a un objeto de tipo Usuario que se encuentra en la tabla `UsuariosObj`, para obtener información sobre alguno de los atributos de dicho objeto referenciado, hay que utilizar la función `DEREF`.

Esta función se utiliza **como parte de una consulta** `SELECT`, por lo que hay que utilizar una tabla tras la cláusula `FROM`. Esto puede resultar algo confuso, ya que las referencias a objetos apuntan directamente a un objeto concreto que se encuentra almacenado en una determinada tabla. Por tanto, no debería ser necesario indicar de nuevo en qué tabla se encuentra. Realmente es así. Podemos hacer referencia a cualquier tabla en la consulta, y la función `DEREF` nos devolverá el objeto referenciado que se encuentra en su tabla correspondiente.

La base de datos de Oracle ofrece la **tabla** DUAL para este tipo de operaciones. Esta tabla es creada de forma automática por la base de datos, es accesible por todos los usuarios, y **tiene un solo campo y un solo registro**. Por tanto, es como una tabla comodín.

```
SELECT DEREF(u_ref) INTO u1 FROM Dual;
dbms_output.put_line(u1.nombre);
```

Por tanto, para obtener el objeto referenciado por una variable `REF`, debes **utilizar una consulta sobre cualquier tabla**, independientemente de la tabla en la que se encuentre el objeto referenciado. Sólo existe la condición de que siempre se obtenga una solo fila como resultado. Lo más **cómodo es utilizar esa tabla** DUAL. Aunque se use esa tabla comodín, **el resultado será un objeto** almacenado en la tabla UsuariosObj.

**En este documento puedes encontrar información general sobre todo lo tratado en esta unidad, incluyendo ejemplos de tablas de objetos con uso de referencias:**
Anexo IV - Bases de Datos Objeto-Relacionales en Oracle 8.

# Anexo I - Soporte objeto-relacional en Oracle 8

**Original en: http://alarcos.inf-cr.uclm.es/doc/bbddavanzadas/08-09/Documentacion-PracticaII.pdf**

## 1. Registros PL/SQL

Un registro es un grupo de datos relacionados, almacenados en campos, cada uno de los cuales tiene su propio nombre y tipo y que se tratan como una sola unidad lógica.

Los campos de un registro pueden ser inicializados y pueden ser definidos como `NOT NULL`. Aquellos campos que no sean inicializados explícitamente, se inicializarán a `NULL`.
Los registros pueden estar anidados.

### 1.1. Declaración de tipos y de variables registro

Declaración del tipo registro:
Sintaxis:

```
TYPE nombre_tipo_reg IS RECORD
(declaración_campo [,declaración_campo]...);
```

Donde `declaración campo` tiene la forma:

```
nombre_campo
{tipo campo | variable%TYPE | table.columns%TYPE | table%ROWTYPE }
 [ [NOT NULL] {:=  | DEFAULT} expresión]
```

Declaración de una variable del tipo registro:

```
id_variable nombre_tipo_reg;
```

### 1.2. Acceso a un campo de una variable de tipo registro

```
 id_variable.nombre_campo
```

### 1.3. Asignación de valores

- ✓ Puede asignarse un valor directamente a un campo:
  `Id.campo := 3;`
- ✓ Puede asignarse valor a todos los campos de un registro utilizando una sentencia `SELECT`. En este caso hay que tener cuidado en especificar las columnas en el orden conveniente según la declaración de los campos del registro.
- ✓ Puede asignarse un registro a otro siempre y cuando sean del mismo tipo

### 1.4. Declaración de registros con el atributo %ROWTYPE

Se puede declarar un registro basándose en una colección de columnas de una tabla o una vista de la base de datos mediante el atributo `%ROWTYPE`.

Por ejemplo, si tengo una tabla empleado declarada como:

```
CREATE TABLE empleado(
id number,
nombre char(10),
apellido char(20),
dirección char(30));
```

Puedo declarar una variable de tipo registro como:

```
 DECLARE
 reg_emp empleado%ROWTYPE;
```

Lo cual significa que el registro `reg emp` tendrá la siguiente estructura:

```
id number,
nombre char(10),
apellido char(20),
dirección char(30)
```

De esta forma se podrán asignar valores a los campos de un registro a través de un select sobre la tabla a partir de la cual se creo el registro.

## 2. Tablas PL/SQL

Una tabla PL/SQL :

- ✓ Es similar a un array
- ✓ Tiene dos componentes: Una clave primaria de tipo `BINARY_INTEGER` que permite indexar en la tabla PL/SQL y una columna de escalares o registros que contiene los elementos de la tabla PL/SQL
- ✓ Puede incrementar su tamaño dinámicamente.

### 2.1. Creación de una tabla

Declaración de un tipo tabla PL/SQL.
Sintaxis:

```
TYPE nombre_tipo_tabla IS TABLE OF
{tipo columna | variable%TYPE | tabla.columna%TYPE}[NOT NULL]
INDEX BY BINARY_INTEGER ;
```

Declaración de una variable del tipo

```
nombre_var nombre_tipo_tabla;
```

No es posible inicializar las tablas en la inicialización.

### 2.2. Referenciar un elemento de la tabla

Sintaxis:

```
Pl/sql_nombre_tabla(valor_primary_key);
```

El rango de binary integer es –2147483647.. 2147483647, por lo tanto el índice puede ser negativo, lo cual indica que el índice del primer valor no tiene que ser necesariamente el uno.

## 3. Tablas PL/SQL de registros

Es posible declarar elementos de una tabla PL/SQL como de tipo registro.
Para referenciar un elemento se hará de la siguiente forma:

```
nombre_tabla(índice).nombre_campo
```

## 4. Funciones para el manejo de tablas PL/SQL

Función `EXISTS(i)`. Utilizada para saber si en un cierto índice hay almacenado un valor. Devolverá `TRUE` si en el índice i hay un valor.
Función `COUNT`. Devuelve el número de elementos de la tabla PL/SQL.
Función `FIRST`. Devuelve el menor índice de la tabla. `NULL` si está vacía.
Función `LAST`. Devuelve el mayor índice de la tabla. NULL si está vacía.
Función `PRIOR (n)`. Devuelve el número del índice anterior a n en la tabla.
Función `NEXT (n)`. Devuelve el número del índice posterior a n en la tabla.
Función `TRIM`. Borra un elemento del final de la tabla PL/SQL. `TRIM(n)` borra **n** elementos del final de la tabla PL/SQL.
Función `DELETE`. Borra todos los elementos de la tabla PL/SQL. `DELETE(n)` borra el correspondiente al índice **n**. `DELETE(m,n)` borra los elementos entre **m** y **n**.

# 5. Tablas anidadas (Nested tables)

Un tipo colección es aquel que maneja varias variables como una unidad. La versión 2 de PL/SQL sólo tenía un tipo de dato colección, las tablas PL/SQL vistas anteriormente.

La versión 8 de Oracle PL/SQL añade dos tipos colección nuevos las tablas anidadas y los varrays. Cada uno de estos tipos de colecciones puede interpretarse como un tipo de objeto con atributos y métodos.

Las tablas anidadas son muy parecidas a las tablas PL/SQL (también llamadas indexadas). Las tablas anidadas añaden funcionalidad a las indexadas al añadir métodos de colección adicionales y la capacidad de almacenar tablas anidadas en una tabla de la base de datos. Estas tablas también pueden manejarse directamente utilizando órdenes SQL.

Aparte de esto, la funcionalidad básica de una tabla anidada es la misma que la de una tabla PL/SQL

## 5.1. Sintaxis

```
TYPE nombre_tabla IS TABLE OF tipo_tabla [NOT NULL];
```

La única diferencia con la declaración de una tabla indexada es que ahora no aparece la cláusula `INDEX BY BINARY_INTEGER`.

## 5.2. Inicialización de una tabla

Para inicializar una tabla anidada hay que utilizar el constructor (igual que se hace con los objetos). Este constructor tendrá el mismo nombre que la tabla anidada. A continuación podrán ponerse los valores con los que queramos inicializar o nada (en cuyo caso se inicializa a una tabla sin elementos, que es diferente que una tabla a `NULL`).

Ejemplo.

```
set serveroutput on;
DECLARE
TYPE tabla numero IS TABLE OF NUMBER;
var tabla 1  tabla numero := tabla numero(0);
var tabla 2 tabla numero := tabla numero(1, 2, 3, 4);
var_tabla_3 tabla_numero := tabla_numero();
var_tabla_4 tabla_numero;
BEGIN
   var tabla 1(1):= 123;
   DBMS_OUTPUT.PUT_LINE('Valor 1 de var_1: ' || var_tabla_1(1));
   DBMS_OUTPUT.PUT_LINE('Valor 1 de var_2: ' || var_tabla_2(1));
   IF var_tabla_3 IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('var_tabla_3 SI es NULL');
   ELSE
      DBMS OUTPUT.PUT LINE('var tabla 3 NO es NULL');
   END IF;
   IF var_tabla_4 IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('var_tabla_4 SI es NULL');
   ELSE
      DBMS_OUTPUT.PUT_LINE('var_tabla_4 NO es NULL');
   END IF;
END;
/
```

La salida será:

```
Valor 1 de var_1: 123
Valor 1 de var_2: 1
var_tabla_3 NO es NULL
var_tabla_4 SI es NULL
```

### 5.3. Claves en la inicialización

Cuando se inicializa una tabla utilizando el constructor, los elementos de la tabla se numeran secuencialmente desde 1 hasta el número de elementos especificado en la llamada al constructor. Es posible eliminar un elemento mediante el uso de `DELETE` (que se verá más adelante en esta misma práctica). En caso de borrado en una tabla anidada de la base de datos, las claves se renumeran para seguir siendo secuenciales.

Ejemplo

```
set serveroutput on;
DECLARE
TYPE tabla_numero IS TABLE OF NUMBER;
var_tabla_2 tabla_numero := tabla_numero(10, 20, 30, 40);
BEGIN
   DBMS OUTPUT.PUT LINE('Nro elem de var tabla 2: ' ||
   var tabla 2.count);
   DBMS OUTPUT.PUT LINE('Primer elem: ' || var tabla 2.first);
   DBMS OUTPUT.PUT LINE('Último elem: ' || var tabla 2.last);
END;
/
```

La salida será:

```
NRO ELEM DE VAR_TABLA_2: 4
PRIMER ELEM: 1
ÚLTIMO ELEM: 4
```

### 5.4. Adición de elementos a una tabla existente

Aunque las tablas no tienen un tamaño fijo, no se puede asignar un valor a un elemento que todavía no existe. Para poder añadir elementos hay que utilizar el método `EXTEND` (que se verá más adelante en esta misma práctica).

Ejemplo.

```
set serveroutput on;
DECLARE
TYPE tabla numero IS TABLE OF NUMBER;
var tabla 2 tabla numero := tabla numero(10, 20, 30, 40);
BEGIN
   var_tabla_2 (1) := 50;
   DBMS_OUTPUT.PUT_LINE('Primer elemento de var_tabla_2: ' ||
   var_tabla_2(1) );
END;
/
```

La salida es:

```
PRIMER ELEMENTO DE VAR_TABLA_2: 50
PROCEDIMIENTO PL/SQL TERMINADO CON ÉXITO.
```

Pero si pongo:

```
set serveroutput on;
DECLARE
TYPE tabla numero IS TABLE OF NUMBER;
var_tabla_2 tabla_numero := tabla_numero(10, 20, 30, 40);
BEGIN
   var_tabla_2 (1) := 50;
   DBMS_OUTPUT.PUT_LINE('Primer elemento de var_tabla_2: ' ||
   var_tabla_2(1) );
   Var tabla 2 (5) := 60;
   DBMS_OUTPUT.PUT_LINE('Quinto elemento de var_tabla_2: ' ||
   var_tabla_2(5));
END;
/
```

Salida:

```
PRIMER ELEMENTO DE VAR_TABLA_2: 50
DECLARE
```

```
*
ERROR EN LÍNEA 1:
ORA-06533: SUBÍNDICE MAYOR QUE EL RECUENTO
ORA-06512: EN  LÍNEA 7
```

## 5.5. Tablas anidadas en la base de datos

Una tabla anidada se puede almacenar como una columna de una tabla. Para ello hay que definir la tabla anidada con la orden `CREATE TYPE` para crear el tipo de la tabla anidada en lugar de `TYPE`.

Ejemplo

```
CREATE TYPE inf_libro AS OBJECT(
   titulo varchar2(40),
   nombre_autor varchar2(40),
   isbn number
);
CREATE TYPE  isbn libros AS TABLE OF inf libro;
CREATE TABLE prestamo (
   fecha_entrega date,
   nro_socio number(10),
   libros_prestados isbn_libros)
   NESTED TABLE  libros_prestados STORE AS prestados_tabla;
```

## 5.6. Manipulación de tablas completas

Una tabla anidada almacenada en una tabla de la base de datos se puede manipular en su integridad o pueden manipularse sus filas individuales. Cualquiera de los dos casos, se pueden utilizar órdenes SQL. El índice de la tabla anidada sólo puede utilizarse cuando la tabla está en PL/SQL.

### *Inserción*

Para insertar una fila en una tabla anidada se utiliza la orden `INSERT`. Hay que tener en cuenta que la tabla se crea e inicializa en PL/SQL y después se inserta en la base de datos.

Ejemplo inserción.

```
DECLARE
libros isbn libros := isbn libros(inf libro('La ruta no natural', 'Macario Polo',
1234567));
BEGIN
   INSERT INTO prestamo VALUES (sysdate, 12,
   isbn libros( inf libro('Métricas para bases de datos', 'Coral Calero', 234567),
      inf libro('La amigdalitis de Tarzán', 'Alfredo Bryce', 3456)));

   INSERT INTO prestamo VALUES (sysdate, 24, libros);
END;
/
```

### *Modificación*

De forma similar, se utiliza `UPDATE` para modificar la tabla almacenada

Ejemplo modificación

```
DECLARE
libros isbn_libros := isbn_libros(inf_libro('La ruta no natural', 'Macario Polo',
1234567), inf_libro('La amigdalitisTarzá', 'Alfredo Bryce', 3456));
BEGIN
   UPDATE prestamo
   SET libros_prestados = libros
   WHERE nro_socio = 24;
END;
/
```

### *Eliminación*

`DELETE` puede eliminar una fila que contenga una tabla anidada.

Ejemplo eliminación

```
BEGIN
   DELETE FROM prestamo WHERE nro_socio = 24;
END;
```

### *Selección*

Cuando se recupera una tabla anidada en una variable PL/SQL, se asignan claves comenzando por 1 hasta llegar al número de elementos que contiene la tabla. Este valor puede determinarse mediante el método `COUNT`, el cual se describe más adelante. Se trata de las mismas claves establecidas por el constructor.

Ejemplo selección

```
Set serveroutput on;
DECLARE
libros isbn_libros;
i NUMBER;

BEGIN
   SELECT libros_prestados INTO libros FROM prestamo WHERE nro_socio=12;
FOR i IN 1 .. libros.count LOOP
   DBMS_OUTPUT.PUT_LINE('Título: ' || libros(i).titulo|| ' del elemento: '|| i);
END LOOP;
END;
/
```

Salida:

```
Título: Métricas para bases de datos del elemento: 1
Título: La amigdalitis de Tarzán del elemento: 2
Procedimiento PL/SQL terminado con éxito.
```

## 6. VARRAYS

Un varray se manipula de forma muy similar a las tablas indexadas o anidadas pero se implementa de forma diferente. Los elementos en el varray se almacenan comenzando en el índice 1 hasta la longitud máxima declarada en el tipo varray.

### 6.1. Declaración de un varray

El tipo_elementos puede especificarse utilizando `%TYPE`. Sin embargo, no puede ser `BOOLEAN`, `NCHAR`, `NCLOB`, `NVARCHAR(2)`, `REF CURSOR`, `TABLE` u otro `VARRAY`.

```
 TYPE nombre_tipo IS VARRAY (tamaño_maximo) OF tipo_elementos;
```

### 6.2. Inicialización de un varray

De forma similar a las tablas, los `VARRAY` se inicializan utilizando un constructor.

Ejemplo

```
DECLARE
   TYPE tipo_numeros IS VARRAY(20) OF NUMBER(3);
 nros uno tipo numeros;
 nros dos tipo numeros := tipo numeros(1,2);
 nros tres tipo numeros := tipo numeros(NULL);
 BEGIN
   IF nros_uno IS NULL THEN
      DBMS_OUTPUT.PUT_LINE(' nros_uno es NULL');
   END IF;
   IF nros tres IS NULL THEN
      DBMS_OUTPUT.PUT_LINE(' nros_tres es NULL');
   END IF;
   IF nros_tres(1) IS NULL THEN
      DBMS_OUTPUT.PUT_LINE(' nros_tres(1) es NULL');
   END IF;
END;
/
```

Salida:

```
NROS_UNO ES NULL
NROS_TRES(1) ES NULL
PROCEDIMIENTO PL/SQL TERMINADO CON ÉXITO.
```

### 6.3. Manipulación de los elementos de un varray

Como las tablas anidadas, el tamaño inicial de un `VARRAY` se establece mediante el número de elementos utilizados en el constructor usado para declararlo. Si se hacen asignaciones a elementos que queden fuera del rango se producirá un error.

Ejemplo.

```
DECLARE
   TYPE t cadena IS VARRAY(5) OF VARCHAR2(10);
   v_lista t_cadena:= ('Paco', 'Pepe', 'Luis');
BEGIN
   v lista(2) := 'Lola';
   v lista(4) := 'Está mal';
END;
/
```

El tamaño de un `VARRAY` podrá aumentarse utilizando la función `EXTEND` que se verá más adelante, pero nunca con mayor dimensión que la definida en la declaración del tipo. Por ejemplo, la variable `v_lista` que sólo tiene 3 valores definidos por lo que se podría ampliar pero nunca más allá de cinco.

### 6.4. Varrays en la base de datos

Los VARRAYS pueden almacenarse en las columnas de la base de datos. Sin embargo, un varray sólo puede manipularse en su integridad, no pudiendo modificarse sus elementos individuales de un varray.

Ejemplo

```
CREATE OR REPLACE TYPE lista_libros AS VARRAY(10) OF inf_libro;
CREATE TABLE ejemplo(
   id number,
   libros inf_libro);
```

### 6.5. Manipulación de varrays almacenados

Para modificar un varray almacenado, primero hay que seleccionarlo en una variable PL/SQL. Luego se modifica la variable y se vuelve a almacenar en la tabla.

## 7. VARRAYS y Tablas Anidadas

Los varrays y las tablas anidadas son colecciones y, como tales, tienen algunos aspectos similares:

- ✓ Ambos tipos permiten el acceso a elementos individuales utilizando la notación con subíndices
- ✓ Ambos tipos pueden almacenarse en la base de datos

Pero también existen diferencias:

- ✓ Los varrays tienen un tamaño máximo y las tablas anidadas no
- ✓ Los varrays se almacenan junto con la tabla que los contiene mientras que las tablas anidadas se almacenan en una tabla separada, que puede tener diferentes características de almacenamiento
- ✓ Cuando están almacenados en la base de datos, los varrays mantienen el orden y los valores de los subíndices para los elementos, mientras que las tablas anidadas no.
- ✓ Los elementos individuales se pueden borrar de una tabla anidada usando el método TRIM por lo que el tamaño de la tabla disminuye. Un varray siempre tiene un tamaño constante.

## 8. Métodos de colecciones

Los métodos de colecciones sólo se pueden llamar desde órdenes procedimentales y no desde órdenes SQL.

Todos se verán con un ejemplo basado en la siguiente declaración:

```
CREATE OR REPLACE TYPE NumTab AS TABLES OF NUMBER;
CREATE OR REPLACE TYPE NumVar AS VARRAY(25) OF NUMBER;
```

### *EXISTS*

Se usa para averiguar si en realidad existe el elemento referenciado.

Sintaxis:

```
EXISTS(n)
```

Donde n es una expresión entera. Si el elemento existe (incluso aunque sea NULL) devuelve TRUE.

```
LOOP
IF tabla.EXISTS(v_count) THEN
   EXISTE
   v count:=v count+1;
ELSE
   EXIT;
END IF;
END LOOP;
```

### *COUNT*

Devuelve un número entero correspondiente al número de elementos que tiene actualmente una colección.

```
 VarTab NumTab:=(1,2,3);
 VarVec NumVar := (-1, -2, -3, -4);
 BEGIN
   DBMS...(VarTab.COUNT, VarVec.COUNT);
 END;
```

### *LIMIT*

Devuelve el número máximo actual de elementos de una colección. Siempre devuelve `NULL` cuando se aplica a una tabla anidada (ya que no tienen tamaño máximo). Para los `VARRAY` devolverá el máximo valor definido en la declaración.

### *FIRST y LAST*

`FIRST` devuelve el índice del primer elemento de la colección y `LAST` el último. En un `VARRAY`, `FIRST` siempre devuelve 1 y `LAST` siempre devuelve el valor de `COUNT`.

### *NEXT y PRIOR*

Se utilizan para incrementar o decrementar la clave de una colección.

`NEXT(n)` Clave del elemento inmediatamente posterior a **n**.

`PRIOR(n)` Clave del elemento inmediatamente anterior a **n**.

Si no existe (el posterior o anterior) devuelve NULL.

`EXTEND`

Se usa para añadir elementos al final de una tabla anidada. Tiene tres formatos:

`EXTEND` Añade un elemento `NULL` con índice `LAST +1` al final de la tabla

`EXTEND(n)` Añade **n** elementos con valor `NULL` al final de la tabla.

`EXTEND(n,i)` Añade **n** copias del elemento i al final de la tabla.

Si la tabla se ha creado con restricciones NOT NULL sólo se podrá utilizar el último formato.

### *DELETE*

Elimina uno o más elementos de una tabla anidada. Tiene tres formatos:
`DELETE` Elimina la tabla completa
`DELETE(n)` Elimina el elemento con el índice **n**
`DELETE(m,n)` Elimina todos los elementos entre **m** y **n**.

Si un elemento que se va a borrar no existe, `DELETE` no da error y lo salta

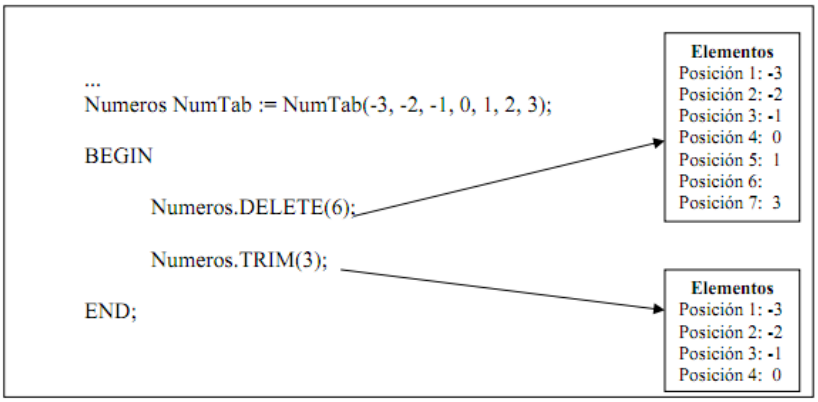
### *TRIM*

Elimina elementos del final de una tabla anidada. `TRIM` no tiene efecto cuando se usa sobre un `varray`.

Tiene dos formatos:
`TRIM` Elimina el elemento del final de la colección
`TRIM (n)` Elimina **n** elementos.

Si n es mayor que `COUNT` se genera una excepción. `TRIM` opera sobre el tamaño interno de la colección, incluyendo cualquier elemento eliminado con un `DELETE`.

Ejemplo.




## 9. Paquetes y resolución del problema de las tablas mutantes

Finalmente, vamos a ver como solucionar el problema de las tablas mutantes utilizando disparadores y tablas PL/SQL.

Una tabla es mutante sólo para los disparadores a nivel de fila. No se puede usar, sin más, un disparador a nivel de orden, porque, por lo general, necesitamos acceder a valores que han sido modificados (de ahí el problema de las tablas mutantes).

La solución consiste en crear dos disparadores, uno a nivel de fila y otro a nivel de orden:

- ✓ En el disparador con nivel de fila almacenamos (en una estructura de datos apropiada) los datos que queremos consultar (los que provocan el error de tabla mutante)
- ✓ En el disparador con nivel de orden realizamos la consulta (pero sobre los datos almacenados en lugar de sobre la tabla)

La mejor forma de almacenar los valores es en una tabla PL/SQL y aunar todas las operaciones descritas dentro de un paquete.

Por ejemplo, dada la siguiente tabla:

```
CREATE TABLE estudiantes(
id   INTEGER PRIMARY KEY,
```

```
especialidad VARCHAR2(20));
```

Con los siguientes valores:

```
insert into estudiantes values(1001, 'Historia');
insert into estudiantes values(1002, 'Historia');
insert into estudiantes values(1003, 'Informática');
insert into estudiantes values(1004, 'Matemáticas');
insert into estudiantes values(1005, 'Informática');
insert into estudiantes values(1006, 'Historia');
insert into estudiantes values(1007, 'Informática');
insert into estudiantes values(1008, 'Matemáticas');
insert into estudiantes values(1009, 'Historia');
insert into estudiantes values(10010, 'Informática');
insert into estudiantes values(10011, 'Historia');
```

y creamos el siguiente disparador:

```
CREATE OR REPLACE TRIGGER limite_especialidad
BEFORE INSERT OR UPDATE OF especialidad ON estudiantes
FOR EACH ROW
DECLARE
maxEstudiantes CONSTANT NUMBER:=5;
EstudiantesActuales NUMBER;
BEGIN
SELECT COUNT(*) INTO EstudiantesActuales
FROM estudiantes
WHERE especialidad = :new.especialidad;
IF EstudiantesActuales+1>maxEstudiantes THEN
   RAISE APPLICATION ERROR(-20000, 'Demasiados estudiantes
en la especialidad: '|| :new.especialidad);
END IF;
END limite_especialidad;
```

Si ejecutamos

```
UPDATE estudiantes
SET especialidad = 'Historia'
WHERE id=1003;
```

Nos da un error de tabla mutante.

Para arreglarlo definimos el siguiente paquete:

```
CREATE OR REPLACE PACKAGE DatosEstudiantes AS
TYPE TipoEspecialidad IS TABLE OF estudiantes.especialidad%TYPE
INDEX BY BINARY_INTEGER;
TYPE TipoIdentificador IS TABLE OF estudiantes.id%TYPE INDEX BY
BINARY_INTEGER;
EspecEst TipoEspecialidad;
IdEst TipoIdentificador;
NumEntradas BINARY INTEGER:=0;
END DatosEstudiantes;
```

Creamos el disparador a nivel de fila para almacenar los nuevos datos:

```
CREATE OR REPLACE TRIGGER FilaLimiteEspecialidad
BEFORE INSERT OR UPDATE OF especialidad ON estudiantes
FOR EACH ROW
BEGIN
DatosEstudiantes.NumEntradas := DatosEstudiantes.NumEntradas + 1;
DatosEstudiantes.EspecEst(DatosEstudiantes.NumEntradas) :=
:new.especialidad;
DatosEstudiantes.IdEst(DatosEstudiantes.NumEntradas) := :new.id;
END FilaLimiteEspecialidad;
```

Creamos el disparador a nivel de orden para dar la funcionalidad que queríamos:

```
CREATE OR REPLACE TRIGGER OrdenLimiteEspecialidad
AFTER INSERT OR UPDATE OF especialidad ON estudiantes
DECLARE
maxEstudiantes CONSTANT NUMBER:=5;
EstudiantesActuales NUMBER;
EstudianteId estudiantes.id%TYPE;
```

```
LaEspecialidad estudiantes.especialidad%TYPE;
BEGIN
FOR indice IN 1..DatosEstudiantes.NumEntradas LOOP
EstudianteId := DatosEstudiantes.IdEst(indice);
LaEspecialidad := DatosEstudiantes.EspecEst(indice);
SELECT COUNT(*) INTO EstudiantesActuales
   FROM estudiantes
   WHERE especialidad = LaEspecialidad;
IF EstudiantesActuales+1>maxEstudiantes THEN
   RAISE APPLICATION ERROR(-20000, 'Demasiados estudiantes
en la especialidad: '|| LaEspecialidad);
END IF;
END LOOP;
DatosEstudiantes.NumEntradas := 0;
END OrdenLimiteEspecialidad;
```

Si de nuevo ejecutamos

```
UPDATE estudiantes
SET especialidad = 'Historia'
WHERE id=1003;
```

Ya no nos da el error de tabla mutante sino que nos devuelve:

DEMASIADOS ESTUDIANTES EN LA ESPECIALIDAD: HISTORIA

Pero si ejecutamos:

```
UPDATE estudiantes
SET especialidad = 'Filología'
WHERE id=1003;
```

**Sintaxis del bucle FOR:**

```
FOR i IN valor_ini.. valor_fin LOOP
   Cuerpo For
END LOOP;
```

Para escribir valores por pantalla:

```
dbms_output.put_line ('text' ||var);
```

Pero para que funcione, al comenzar la sesión hay que ejecutar

```
set serveroutput on;
```

# Anexo II - Tipos de Objeto en PL/SQL

**Original en: http://www.kybele.etsii.urjc.es/docencia/BD/2011-2012/Material/%5BBD-2010-2011%5DPLSQL.ObjectTypes.pdf**

## Introducción

Una BDOR soporta las dos tecnologías: relacional y objeto-relacional

Aportaciones OR

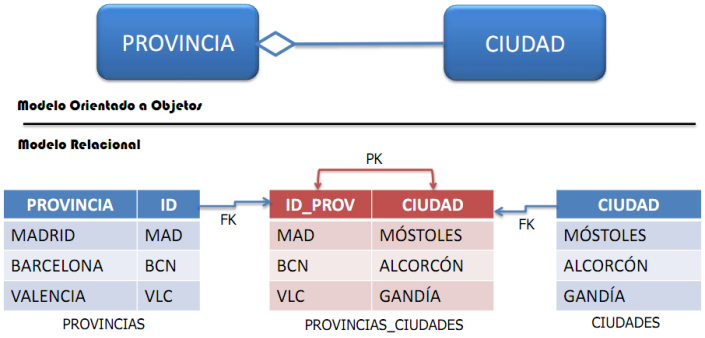
- ✓ UDTs
  - ➔ Atributos / Métodos …
- ✓ Tipos Complejos en una columna
- ✓ Referencias

Oracle incorpora estas mejoras desde la versión 8i

Las estructuras de almacenamiento siguen siendo tablas, pero se pueden usar mecanismos de orientación al objeto para la gestión de datos

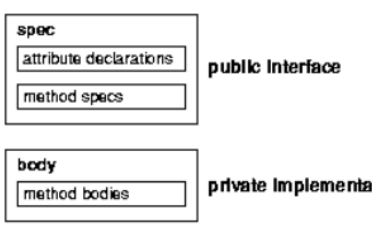
Cada objeto tiene un tipo, se almacena en una fila de una tabla y tiene un identificador que permite referenciarlo desde otros objetos (otras filas de otras tablas)

Los objetos permiten modelar mejor las relaciones "parte-todo".




## Estructura de un tipo de objeto

Los parámetros no deben restringirse en tamaño. Se invocan igual que se recupera un campo (Especificando los parámetros si fuera el caso). Definir un tipo no implica reservar espacio para objetos del tipo




```
CREATE [OR REPLACE] TYPE Complejo AS OBJECT (
parte_r   REAL, -- attributo
parte_e   REAL,
MEMBER FUNCTION suma (x Complejo) RETURN Complejo, -- metodo
MEMBER FUNCTION resta (x Complejo) RETURN Complejo,
```

```
CREATE [OR REPLACE] TYPE BODY Complejo AS
   MEMBER FUNCTION suma (x Complejo) RETURN Complejo IS
   BEGIN
      RETURN Complejo(parte_r + x.parte_r, parte_e + x.parte_e);
   END plus;

   MEMBER FUNCTION resta (x Complejo) RETURN Complejo IS
   BEGIN
      RETURN Complejo(parte_r - x.parte_r, parte_e - x.parte_e);
   END less;
END;
```

## Tablas de objetos

Las tablas tipadas son tablas que almacenan objetos del tipo sobre el que han sido definidas. Cada fila almacena un objeto

También podemos verlas como una tabla con una única columna del tipo objeto y una tabla con tantas columnas como el tipo objeto

Almacenan un objeto en cada fila y permiten acceder a los campos del objeto como si fueran columnas de la tabla. Las restricciones se definen sobre la tabla y se aplicarán SÓLO sobre aquellos objetos almacenados en dicha tabla

```
CREATE OR REPLACE TYPE Tipo_Coche AS OBJECT (
    Marca          VARCHAR2(25),
    Modelo         VARCHAR2(25),
    Matricula      VARCHAR2(9))
/

CREATE TABLE COCHES OF Tipo_Coche;
```

```
CREATE OR REPLACE TYPE Tipo_Coche AS OBJECT (
    Marca          VARCHAR2(25),
    Modelo         VARCHAR2(25),
    Matricula      VARCHAR2(9))
/

CREATE TABLE COCHES OF Tipo_Coche;

ALTER TABLE COCHES ADD
                CONSTRAINT PK_COCHE PRIMARY KEY (Marca, Modelo);
```

## Métodos

Implementan el comportamiento de los objetos del tipo y pueden ser `MEMBER`, `STATIC` o `CONSTRUCTOR`

```
CREATE TYPE Tipo_Cubo AS OBJECT (
        largo           INTEGER,
        ancho           INTEGER,
        alto            INTEGER,
        MEMBER FUNCTION superficie RETURN INTEGER,
        MEMBER FUNCTION volumen RETURN INTEGER,
        MEMBER PROCEDURE mostrar ());
/
CREATE TYPE BODY Tipo_Cubo AS
        MEMBER FUNCTION volume RETURN INTEGER IS
        BEGIN
                RETURN largo * ancho * alto;
                -- RETURN SELF.largo * SELF.ancho * SELF.alto;
        END;
        MEMBER FUNCTION superficie RETURN INTEGER IS
        BEGIN
                RETURN 2 * (largo * ancho + largo * alto + ancho * alto);
        END;
        MEMBER PROCEDURE mostrar ()
        IS
        BEGIN
                DBMS_OUTPUT.PUT_LINE('Largo: ' || largo || ' - ' || 'Ancho: ' || ancho || ' - ' || 'Alto: ' || alto);
                DBMS_OUTPUT.PUT_LINE('Volumen: ' || volumen || ' - ' || 'Superficie: ' || superficie);
        END;
END;
/
```

La variable SELF permite referirse al objeto sobre el que se invocó la función/procedimiento

Método MEMBER

```
CREATE TYPE Tipo_Cubo AS OBJECT (…);
/

CREATE TYPE BODY Tipo_Cubo AS …
END;
/

CREATE TABLE Cubos of Tipo_Cubo;
INSERT INTO Cubos VALUES(Tipo_Cubo (10, 10, 10));
INSERT INTO Cubos VALUES(Tipo_Cubo (3, 4, 5));
SELECT * FROM Cubos;
SELECT c.volumen(), c.superficie () FROM cubos c
WHERE c.largo = 10;

DECLARE
        mi_cubo      Tipo_Cubo;
BEGIN

  SELECT VALUE(c) INTO mi_cubo FROM Cubos c
  WHERE c.largo = 10;
  mi_cubo.mostrar ();
END;
/
DROP TABLE Cubos;
DROP TYPE tipo_Cubo FORCE;
```

Métodos STATIC:
Operaciones globales, que no son de los objetos, sino del tipo

```
CREATE TYPE Tipo_Cubo AS OBJECT(
 ….,
STATIC PROCEDURE nuevoCubo (
  v_largo      INTEGER,
  v_ancho      INTEGER,
  v_alto       INTEGER));
/
CREATE TYPE BODY Tipo_Cubo AS
STATIC PROCEDURE nuevoCubo (
                v_largo         INTEGER,
                v_ancho         INTEGER,
                v_alto          INTEGER)
IS
  sqlstmt VARCHAR2(100);
BEGIN
  sqlstmt := 'INSERT INTO Cubos '||schname||'.'||tabname|| '
             VALUES (Tipo_Cubo(largo, ancho, alto))';
  EXECUTE IMMEDIATE sqlstmt;
END;
/

BEGIN
   Tipo_Cubo.nuevoCubo(1, 1, 1);
END;
/
```

## Constructores

Cada vez que se crea un tipo de objeto, Oracle crea automáticamente un método constructor identificado por el mismo nombre del tipo. Es una función que devuelve una nueva instancia del tipo definido por el usuario y establece los valores de sus atributos.

Recibe como parámetros los atributos del tipo. Debe ser siempre explícitamente invocado cada vez que se desee crear un objeto del tipo

### Métodos constructor

```
CREATE OR REPLACE TYPE Tipo_Coche AS OBJECT (
    Marca          VARCHAR2(25),
    Modelo         VARCHAR2(25),
    Matricula      VARCHAR2(9))
/

CREATE OR REPLACE TYPE Tipo_Persona AS OBJECT (
    Nombre         VARCHAR2(25),
    Coche          Tipo_Coche)
/

CREATE TABLE PERSONAS OF Tipo_Persona;

INSERT INTO PERSONAS VALUES ('Ramón Ramirez',
                             Tipo_Coche('CITROEN', '2-CV', 'M-9999999'));
```

Llamada al constructor del tipo para crear el objeto embebido

### Declarar e inicializar objetos

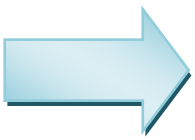
Podemos usar un tipo de objeto igual que cualquier otro tipo de dato. Una buena práctica es inicializar los objetos al declararlos

```
DECLARE
  r Tipo_Persona; -- r toma valor NULL
BEGIN
  IF r IS NULL THEN ... – esta comparación devuelve TRUE
  IF r > (2/3) … -- pero esta comparación TAMBIÉN devuelve NULL
  r := Tipo_Persona('Ramón Ramírez', NULL);
```

### Acceso a los atributos de un objeto

Para acceder a las propiedades de un objeto se utiliza la notación punto (**.**)

```
CREATE TYPE Tipo_Coche AS OBJECT (
    Marca          VARCHAR2(25),
    Modelo         VARCHAR2(25),
    Matricula      VARCHAR2(9));
/

CREATE TYPE Tipo_Persona AS OBJECT (
    Nombre         VARCHAR2(25),
    Coche          Tipo_Coche);
/
```



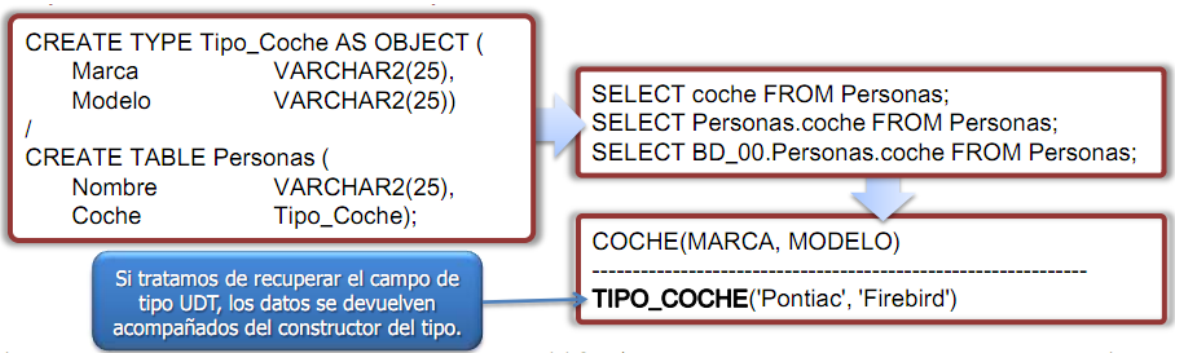
```
DECLARE
  v_persona Tipo_Persona;
BEGIN
  v_persona.coche.marca := 'CITROEN';
  -- error ACCESS_INTO_NULL
  v_persona := Tipo_Persona(NULL,…);
  v_persona.coche.marca := 'CITROEN';
  …
END;
```

### Variables de correlación

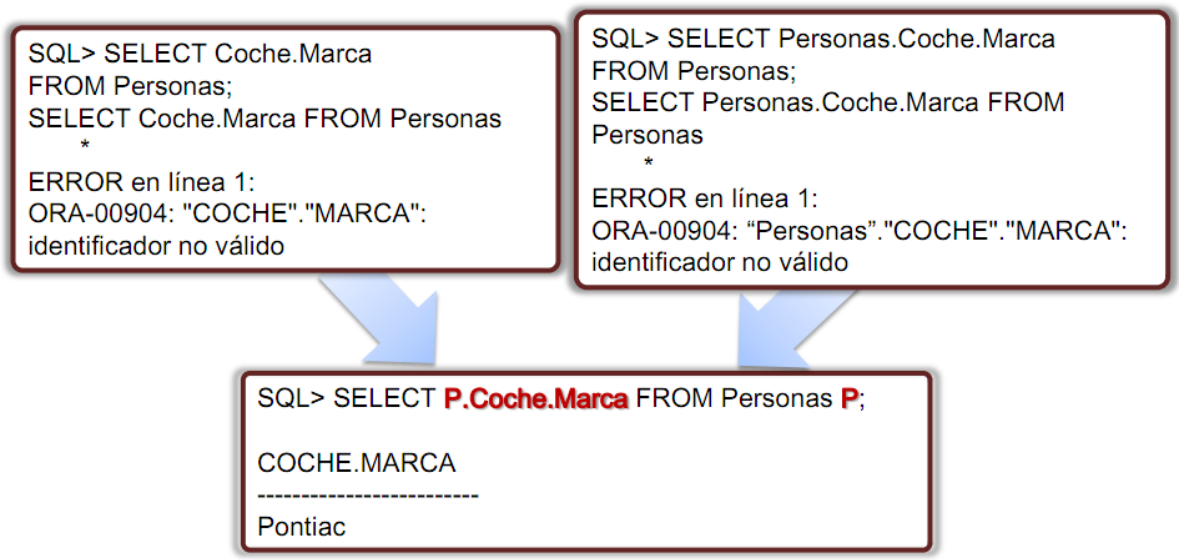
Son un tipo especial de variable PL/SQL. En general, las correlation variables se identifican con los alias de tabla

```
SELECT E.Nombre FROM Empleado E;
SELECT Empleado.Nombre FROM Empleado;
```

Son opcionales y podemos omitirlas, pero cuando trabajamos con UDTs es OBLIGATORIO utilizarlas para acceder a los campos del UDT




Igualmente, para acceder a los campos del UDT por separado hay que utilizar las correlation variables




Las correlation variables permiten resolver ambigüedades

```
SQL> SELECT Cliente.Coche.Marca;
```

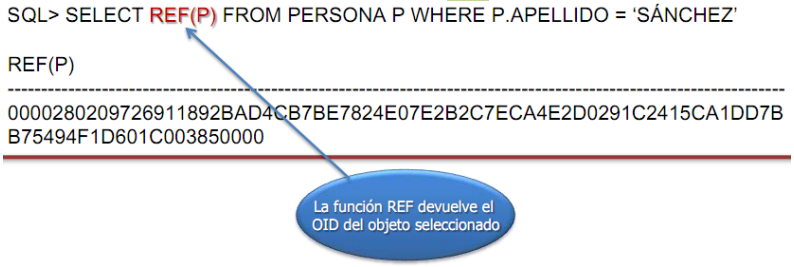
`Cliente` es un nombre de usuario, `Coche` es una tabla y `Marca` una columna
`Cliente` es un nombre de tabla, `Coche` de columna y `Marca` un campo de esa columna

El uso de la variable resuelve el problema

```
SQL> SELECT C.Coche.Marca;
```

### OID - Object IDentifier

Cada fila de una tabla tipada (`OR`) tendrá un identificador del objeto fila → OID
Para guardar esos identificadores Oracle utiliza un tipo `REF`



### *Tipos referencia*

Cada fila (objeto fila) podrá ser referenciada como un objeto a través de su OID

```
CREATE OR REPLACE TYPE Tipo_Persona AS OBJECT (
    Nombre    VARCHAR2(25),
    Coche     Tipo_Coche)
/

CREATE OR REPLACE TYPE Tipo_Empresa AS OBJECT (
    Nombre    VARCHAR2(25),
    NIF       VARCHAR2(25),
    Director  REF Tipo_Persona)
/
CREATE TABLE PERSONAS OF Tipo_Persona;
CREATE TABLE EMPRESAS OF Tipo_Empresa;

SQL> DESC EMPRESAS
 Nombre                                    ¿Nulo?   Tipo
 ----------------------------------------- -------- ------------------------------------
 NOMBRE                                             VARCHAR2(25)
 NIF                                                VARCHAR2(25)
 DIRECTOR                                           TIPO_PERSONA()


SQL> SELECT * FROM EMPRESAS;
Nombre                    NIF          DIRECTOR
------------------------- ------------ ----------------
'Juan Nadie'              '000001'     0000280209726911892BAD4CB7BE7824E07E ....
```

Para obtener una referencia a un objeto utilizamos el operador `REF`

```
DECLARE
  persona_ref REF Persona;
BEGIN
  SELECT REF(p) INTO persona_ref FROM Personas p
  WHERE p.nombre = 'Pepe Pérez';
END;
/
```

Una columna de tipo `REF` guarda un puntero a una fila de la otra tabla. Contiene el OID de dicha fila

```
INSERT INTO EMPRESAS VALUES ('ACME', '666',
        (SELECT REF(P) FROM PERSONAS P WHERE P.NOMBRE LIKE '%Ramirez'));

SELECT * FROM EMPRESAS;
NOMBRE   NIF  DIRECTOR
---------- ----  --------------------------------------------------------------------------------
ACME       666  0000220208EB2D8E93BE39430EB8D325E255E81F5E2F8521EDFD3F4
```

Las referencias no se pueden navegar en PL/SQL

```
DECLARE
  p_ref REF Persona;
  telefono VARCHAR2(15);
BEGIN
  telefono := p_ref.telefono; -- no permitido
  ..............
```

En su lugar hay que usar la función `DEREF` (o el paquete `UTL_REF`), que permite obtener el objeto referenciado

```
DECLARE
  p1 Persona;
  p_ref REF Persona;
  nombre VARCHAR2(15);
BEGIN ...
  -- Si p_ref tiene una referencia válida a una fila de una tabla
  SELECT DEREF(p_ref) INTO p1 FROM dual;
  nombre :=   p1.nombre;
  .....
SQL> SELECT DEREF(E.Director) FROM Empresas E;

DEREF(E.DIRECTOR)(NOMBRE, COCHE(MARCA, MODELO, MATRICULA))
--------------------------------------------------------------------------------
TIPO_PERSONA('Ramón Ramirez', TIPO_COCHE('CITROEN', '2-CV', 'M-9999999'))
```

En cambio si se pueden navegar en SQL

```
CREATE OR REPLACE TYPE Tipo_Persona AS OBJECT (
    Nombre        VARCHAR2(25),
    Coche         Tipo_Coche)
/
CREATE OR REPLACE TYPE Tipo_Empresa AS OBJECT (
    Nombre        VARCHAR2(25),
    NIF           VARCHAR2(25),
    Director      REF Tipo_Persona)
/
CREATE TABLE PERSONAS OF Tipo_Persona;
CREATE TABLE EMPRESAS OF Tipo_Empresa (
    SCOPE FOR (Director) IS Personas);

SELECT E.Director.Nombre FROM Empresas E
WHERE E.Nombre = 'ACME';
```

DEREF implícito

Podemos restringir el conjunto de objetos a los que apuntará la `REF` a los contenidos en una única tabla. De esta forma el almacenamiento ocupará menos espacio y el acceso será más eficiente

```
CREATE OR REPLACE TYPE Tipo_Persona AS OBJECT (
    Nombre          VARCHAR2(25),
    Coche           Tipo_Coche)
/
CREATE OR REPLACE TYPE Tipo_Empresa AS OBJECT (
    Nombre          VARCHAR2(25),
    NIF             VARCHAR2(25),
    Director        REF Tipo_Persona)
/
CREATE TABLE PERSONAS OF Tipo_Persona;
CREATE TABLE EMPRESAS OF Tipo_Empresa (
    SCOPE FOR (Director) IS Personas);
```

## Operador VALUE

Para obtener el objeto almacenado en una fila (y no sólo el valor de los campos de dicho objeto) se necesita la función `VALUE`

```
SELECT * FROM Personas;

NOMBRE
--------------------
COCHE(MARCA, MODELO, MATRICULA)
--------------------------------------------------------------------------------
Ramón Ramirez
TIPO_COCHE('CITROEN', '2-CV', 'M-9999999')
```

La consulta sólo devuelve el valor de los campos del objeto de la clase Tipo_Persona

```
SQL> SELECT VALUE(P) FROM PERSONAS P;

VALUE(P)(NOMBRE, COCHE(MARCA, MODELO, MATRICULA))
--------------------------------------------------------------------------------
TIPO_PERSONA('Ramón Ramirez', TIPO_COCHE('CITROEN', '2-CV', 'M-9999999'))
```

La consulta devuelve el objeto de la clase Tipo_Persona

# Gestión de objetos

Al recuperar el objeto completo, de pueden realizar operaciones con él: modificarlo, insertarlo …

```
SQL> SET SERVEROUTPUT ON
DECLARE
  v_persona     Tipo_Persona;
BEGIN
  SELECT VALUE(P) INTO v_persona FROM PERSONAS P WHERE NOMBRE LIKE '%Ram%';
  DBMS_OUTPUT.PUT_LINE(v_persona.nombre);
  DBMS_OUTPUT.PUT_LINE(v_persona.coche.marca);
  DBMS_OUTPUT.PUT_LINE(v_persona.coche.modelo);
END;
/
Ramón Ramirez
CITROEN
2-CV

DECLARE
     v_persona     Tipo_Persona;
BEGIN
    v_persona := Tipo_Persona('Mamen Tido', Tipo_Coche('SEAT', '600', 'B-8888888'));
    INSERT INTO Personas VALUES (v_persona);
END;
/
```

Como hemos creado el objeto antes, no necesitamos invocar el constructor del tipo

## Forward Type Definitions

A la hora de crear un tipo sólo podemos referirnos a otros tipos que ya estén creados. Esto representa un problema en caso de referencias circulares

```
CREATE TYPE Empleado AS OBJECT (
  nombre VARCHAR2(20),
  dept REF Departmento,
  ... );

CREATE TYPE Departamento AS OBJECT (
  number INTEGER,
  jefe REF Empleado,
  ... );
```

Para resolverlo se utilizan las Forward Type Definitions:

- ✓ Declarar A, crear B y crear A
- ✓ Recompilar A

```
CREATE TYPE Empleado; -- tipo incompleto

CREATE TYPE Departamento AS OBJECT (
  number INTEGER,
  jefe REF Empleado,
  ... );

CREATE TYPE Empleado AS OBJECT (
  nombre VARCHAR2(20),
  dept REF Departmento,
  ... );
```

# Tipos Colección

Oracle soporta dos tipos `VARRAYS`:

- ✓ colección ordenada de elementos de tamaño fijo
- ✓ `NESTED TABLES`: colección no ordenada y de tamaño variable

Como el tipo objeto, incorpora constructores por defecto que hay que utilizar para crear objetos colección

Sus parámetros serán los elementos de la colección

## Creación

```
CREATE OR REPLACE TYPE Tipo_Persona AS OBJECT (
    Nombre          VARCHAR2(25),
    DNI             VARCHAR2(10))
/

CREATE OR REPLACE TYPE Tipo_Personas AS
        VARRAY(15) OF Tipo_Persona
/

CREATE OR REPLACE TYPE Tipo_Empresa AS OBJECT (
    Nombre          VARCHAR2(25),
    NIF             VARCHAR2(25),
    Empleados       Tipo_Personas)
/
CREATE TABLE EMPRESAS OF Tipo_Empresa ();
```

```
CREATE OR REPLACE TYPE Tipo_Personas AS
    TABLE OF Tipo_Persona
/
```

```
CREATE TABLE Empresas OF Tipo_Empresa
  NESTED TABLE Empleados STORE AS Lista_Emp;
/
```

Constructor del tipo colección

```
INSERT INTO EMPRESA VALUES ('ACME', '00000000X',
        Tipo_Personas(Tipo_Persona('Pepe Pèrez', '999999999M')));
```

Constructor del tipo objecto

## Consulta

La consulta estándar recupera los datos anidados

```
SELECT E.Empleados FROM  Empresa(E) WHERE E.Nombre = 'ACME'

Empleados(Nombre, DNI)
--------------------------------------------------------------------------------
Tipo_Personas(Tipo_Persona('Pepe Pèrez', '999999999M')
```

Devuelve los datos anidados

Mientras que la expresión `TABLE` permite descomponerlos

```
SELECT Emp.* FROM Empresa E, TABLE(E.Empleados) Emp
WHERE E.Nombre = 'ACME'

Nombre DNI
---------- --------------------------
'Pepe Pèrez', '999999999M'
```

Esta sería la forma *intutiitva* de realizar la consulta, pero al ejecutarla obtendríamos un error

```
SELECT E.Emp.* FROM Empresa E
WHERE E.Nombre = 'ACME'
```

## Operaciones DML

Operaciones que actúan sobre toda la colección o sobre elementos aislados

Las segundas utilizan el operador `TABLE`

No se soportan actualizaciones individuales en `VARRAYs`

```
-- INSERCIÓN
INSERT INTO TABLE(SELECT E.Empleados FROM  Empresa(E)
                  WHERE E.Nombre = 'ACME')
VALUES ('Mamen Tido', '12345678A')

--ACTUALIZACIÓN
UPDATE TABLE (SELECT E.Empleados FROM  Empresa(E)
                  WHERE E.Nombre = 'ACME')  E
SET VALUE(E) = ('Mamen Tido Mucho', '12345678A')
WHERE E.NOMBRE = 'Mamen Tido';

-- BORRADO
DELETE FROM TABLE(SELECT E.Empleados FROM  Empresa(E)
                  WHERE E.Nombre = 'ACME')  E
WHERE E.NOMBRE = 'Mamen Tido';
```

## Operadores

Obtener información sobre la colección

- ✓ `COUNT` devuelve el número de filas.
- ✓ `EXISTS` devuelve `TRUE` si la fila existe.
- ✓ `FIRST/LAST` devuelve el índice de la primera y última fila.
- ✓ `NEXT/PRIOR` devuelve la fila anterior o posterior a la actual.
- ✓ `LIMIT` informa del número máximo de elementos que puede contener .

Modificar los elementos de la colección

- ✓ `DELETE` borra uno o más elementos usando su índice.
- ✓ `EXTEND` añade nuevas filas.
- ✓ `TRIM` elimina filas.

## PL/SQL

```
CREATE TYPE Tipo_Nombres_Dep IS VARRAY(7) OF VARCHAR2(30);
/
CREATE TABLE Departamentos (
                region                  VARCHAR2(25),
                nombres_dep             Tipo_Nombres_Dep);

BEGIN
  INSERT INTO Departamentos VALUES ('Europe', Tipo_Nombres_Dep('Shipping','Sales','Finance'));
  INSERT INTO Departamentos VALUES('Americas', Tipo_Nombres_Dep('Sales','Finance','Shipping'));
  INSERT INTO Departamentos VALUES('Asia', Tipo_Nombres_Dep('Finance','Payroll','Shipping','Sales'));
COMMIT;
END;
/

DECLARE
  v_nombres   Tipo_Nombres_Dep := Tipo_Nombres_Dep('Benefits', 'Advertising', '
              Contracting', 'Executive', 'Marketing');
  v_nombres2 Tipo_Nombres_Dep;
BEGIN
  UPDATE Departamentos SET nombres_dep = v_nombres WHERE region = 'Europe';
  COMMIT;
  SELECT nombres_dep INTO v_nombres2 FROM Departamentos WHERE region = 'Europe';
  FOR i IN v_nombres2.FIRST .. V_nombres2.LAST
  LOOP
     DBMS_OUTPUT.PUT_LINE('Departamentos = ' || v_nombres2(i));
  END LOOP;
END;
/
```

Recorre la colección de nombres de departamento

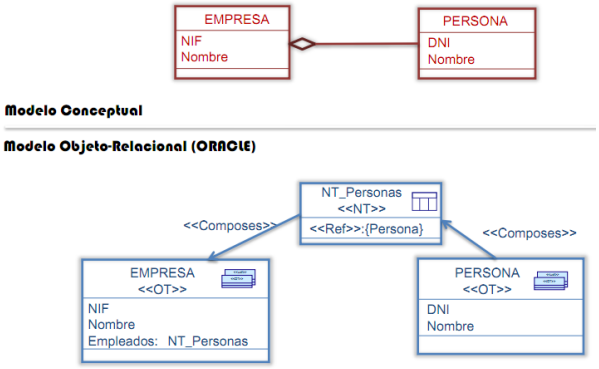
## Cursores y colecciones

```
CREATE TYPE Tipo_Nombres_Dep IS VARRAY(7) OF VARCHAR2(30);
/
CREATE TABLE Departamentos (
                region                  VARCHAR2(25),
                nombres_dep             Tipo_Nombres_Dep);

BEGIN
  INSERT INTO Departamentos VALUES ('Europe', Tipo_Nombres_Dep('Shipping','Sales','Finance'));
COMMIT;
END;
/

DECLARE
  CURSOR c_depts IS SELECT * FROM Departamentos;
  v_region      VARCHAR2(25);
  v_nombres   Tipo_Nombres_Dep;
BEGIN
  OPEN c_depts;
  LOOP
    FETCH c_depts INTO v_region, v_nombres;
    EXIT WHEN c_depts%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('REGION: ' || v_region);
    FOR i IN v_nombres2.FIRST .. V_nombres2.LAST
    LOOP
      DBMS_OUTPUT.PUT_LINE(' - Departamento = ' || '(' || i || ') → ' || v_nombres2(i));
    END LOOP;
  END LOOP;
END;
/
```

Carga una fila (región + lista de nombres de departamento) en las dos variables utilizadas

## Colecciones de tipo REF

La consulta a colecciones de tipos `REF` crea algunos problemas




## Tipos colección

```
CREATE TYPE Tipo_Persona AS OBJECT (
  DNI           VARCHAR2(30),
  Nombre        VARCHAR2(30))
/

CREATE TYPE Tipo_Personas AS TABLE OF REF Tipo_Persona
/

CREATE TYPE Tipo_Persona AS OBJECT (
  NIF           VARCHAR2(30),
  Nombre        VARCHAR2(30),
  Empleados     Tipo_Personas)
/

CREATE TABLE Personas OF Tipo_Persona;

CREATE TABLE Empresas OF Tipo_Empresa
  NESTED TABLE Empleados STORE AS Lista_Emp;
```

```
INSERT INTO Personas VALUES ('0000001', 'Mamen Tido');
INSERT INTO Personas VALUES ('0000002', 'Francisco Rupto');

DECLARE
  v_p1          REF Tipo_Persona;
  v_p2          REF Tipo_Persona;
BEGIN
  SELECT REF(P) INTO v_p1 FROM Personas P
    WHERE P.DNI = '0000001';
  SELECT REF(P) INTO v_p2 FROM Personas P
    WHERE P.DNI = '0000002';
  INSERT INTO Empresas E VALUES ('12345678A', 'ACME',
    Tipo_Personas(vp1, vp2));
END;
/
```

Una alternativa pasa por crear un tipo intermedio que proporcione el nombre columna ...

```
SELECT E.Nombre, EMP.Nombre FROM Empresas E,
TABLE(E.Empleados) EMP;
```

Esta consulta eleva un error, el problema es que el resultado es un OID sin más. Necesitamos un nombre de columna ...

```
SELECT E.Nombre, EMP.COLUMN_VALUE.Nombre
FROM Empresas E, TABLE(E.Empleados) EMP;
```

```
CREATE TYPE Tipo_REF_Persona AS OBJECT (
  Persona       REF Tipo_Persona)
/

CREATE TYPE Tipo_Personas AS TABLE OF Tipo_REF_Persona
/

SELECT E.Nombre, EMP.Persona.Nombre FROM Empresas E,
TABLE(E.Empleados) EMP;
```

# Tipos de Objeto en PL/SQL

SQL permite

## Capture

Cuando una declaración de tipo de un ámbito diferente impide que el compilador resuelva correctamente un nombre o referencia

Para evitar estos errores, Oracle define una serie de reglas, entre otras:

- ✓ Especificar una alias para cualquier tabla involucrada en una sentencia DML
- ✓ Preceder cualquier nombre de columna con el alias dado a la tabla correspondiente
- ✓ Evitar el uso de alias que coincidan con el nombre del esquema

## Resolución de nombres: uso de alias

```
CREATE TYPE Tipo1 AS OBJECT (
        a NUMBER);
CREATE TABLE Tab1 (
        tab2 Tipo1);
CREATE TABLE Tab2 (
        x NUMBER);
SELECT * FROM Tab1 T1  -- alias con el mismo nombre que el esquema
  WHERE EXISTS (SELECT * FROM T1.Tab2 WHERE x = T1.tab2.a);
-- T1.tab2.a se resuelve apuntando al atributo a de la columna tab2 de la tabla Tab1
```

```
CREATE TYPE Tipo1 AS OBJECT (
        a NUMBER);
CREATE TABLE Tab1 (
        tab2 Tipo1);
CREATE TABLE Tab2 (
        x NUMBER,
        a NUMBER);
SELECT * FROM Tabla1 T1  -- alias con el mismo nombre que el esquema
  WHERE EXISTS (SELECT * FROM T1.Tabla2 WHERE x = T1.tab2.a);
-- Ahora T1.tab2.a se resuelve apuntando al atributo a de la la tabla Tab2
```

```
SELECT * FROM Tabla1 tb1
WHERE EXISTS (SELECT * FROM T1.Tabla2 tb2
                    WHERE tb2.x = tb1.tab2.a);
-- Evitamos ambigüedades siguiendo las reglas de nombrado
```

# Anexo III - Bases de datos objeto-relacionales

**Original en: http://informatica.uv.es/iiguia/DBD/Teoria/capitulo_4.pdf**

**El término base de datos objeto-relacional se usa para describir una base de datos que ha evolucionado desde el modelo relacional hasta una base de datos híbrida, que contiene ambas tecnologías:**

**relacional y de objetos.**

Durante muchos años ha habido debates sobre cómo sería la siguiente generación de la tecnología de bases de datos de uso común:

- ✓ Las bases de datos orientadas a objetos.
- ✓ Una base de datos basada en SQL con extensiones orientadas a objetos.

Los partidarios de la segunda opción esgrimen varias razones para demostrar que el modelo objeto relacional dominará:

- ✓ Las bases de datos objeto-relacionales tales como Oracle8i son compatibles en sentido ascendente con las bases de datos relacionales actuales y que además son familiares a los usuarios.
  Los usuarios pueden pasar sus aplicaciones actuales sobre bases de datos relaciones al nuevo modelo sin tener que reescribirlas.
  Posteriormente se pueden ir adaptando las aplicaciones y bases de datos para que utilicen las funciones orientadas a objetos.
- ✓ Las primeras bases de datos orientadas a objetos puras no admitían las capacidades estándar de consulta ad hoc de las bases de datos SQL. Esto también hace que resulte problemático realizar la interfaz entre las herramientas SQL estándar y las bases de datos orientadas a objetos puras.

**Una de las principales razones por las que las bases de datos relacionales tuvieron un éxito tan rápido fue por su capacidad para crear consultas ad hoc.**

## Tecnología objeto-relacional

Para ilustrar la tecnología objeto-relacional utilizaremos como ejemplo el modelo que implementa la base de datos Oracle8.

### Tipos de objetos

El modelo relacional está diseñado para representar los datos como una serie de tablas con columnas y atributos.

Oracle8 es una base de datos objeto-relacional: incorpora tecnologías orientadas a objetos. En este sentido, permite construir tipos de objetos complejos, entendidos como:

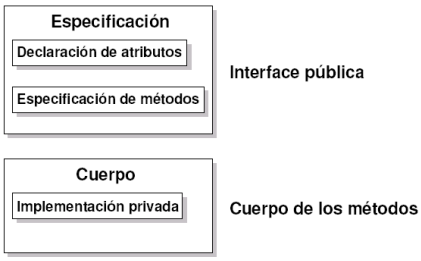
- ✓ Capacidad para definir objetos dentro de objetos.
- ✓ Cierta capacidad para encapsular o asociar métodos con dichos objetos.

### Estructura de un tipo de objeto

Un tipo de objeto consta de dos partes: especificación y cuerpo:

- ✓ La **especificación** constituye la interface a las aplicaciones; aquí se declaran las estructuras de datos (conjunto de atributos) y las operaciones (métodos) necesarios para manipular los datos.
- ✓ El **cuerpo** define los métodos, es decir, implementa la especificación.

En la figura se representa la estructura de un tipo de objeto:



Toda la información que un cliente necesita para utilizar los métodos se encuentra en la especificación.
Es conveniente pensar en la especificiación como en la interface operacional y en el cuerpo como en una caja negra. Esto permite depurar, mejorar o reemplazar el cuerpo sin necesidad de modificar la especificación y sin afectar, por tanto, a las aplicaciones cliente.

### Características:

En una especificación de tipo de objeto los atributos deben declararse antes que cualquiera de los métodos.

Si una especificación de tipo sólo declara atributos, el cuerpo es innecesario.

Todas las declaraciones en la especificación del tipo son públicas. Sin embargo, el cuerpo puede contener declaraciones privadas, que definen métodos internos del tipo de objeto.

El ámbito de las declaraciones privadas es local al cuerpo del objeto.

### Ejemplo:

Un tipo de objeto para manipular números complejos. Un número complejo se representa mediante dos números reales (parte real y parte imaginaria respectivamente) y una serie de operaciones asociadas:

```
CREATE TYPE Complex AS OBJECT (
   rpart REAL,
   ipart REAL,
   MEMBER FUNCTION plus(x Complex) RETURN Complex ,
   MEMBER FUNCTION less(x Complex) RETURN Complex ,
   MEMBER FUNCTION times(x Complex) RETURN Complex ,
   MEMBER FUNCTION divby(x Complex) RETURN Complex
) ;
/
CREATE TYPE BODY Complex AS
   MEMBER FUNCTION plus (x Complex) RETURN Complex IS
   BEGIN
      RETURN Complex(rpart + x.rpat, ipart + x.ipart) ;
   END plus ;
   MEMBER FUNCTION less(x Complex) RETURN Complex IS
   BEGIN
      RETURN Complex(rpart - x.rpart, ipart - x.ipart) ;
   END less;
   MEMBER FUNCTION times(x Complex ) RETURN Complex IS
   BEGIN
      RETURN Complex(rpart * x.rpart - ipart * x.ipart,
      rpart * x.ipart + ipart * x.rpart ) ;
   END times;
   MEMBER FUNCTION divby(x Complex) RETURN Complex IS
      z REAL := x.rpart **2 + x.ipart **2;
   BEGIN
      RETURN Complex ((rpart * x.rpart + ipart * x.ipart) / z ,
         (ipart * x.rpart - rpart * x.ipart) / z) ;
   END divby ;
END;
```

## Componentes de un tipo de objeto

Un tipo de objeto encapsula datos y operaciones, por lo que en la especificación sólo se pueden declarar atributos y métodos, pero no constantes, excepciones, cursores o tipos.
Se requiere al menos un atributo y los métodos son opcionales.

### Atributos

1. Como las variables, un atributo se declara mediante un nombre y un tipo.
2. El nombre debe ser único dento del tipo de objeto (aunque puede reutilizarse en otros objetos)
3. El tipo puede ser cualquier tipo de Oracle excepto:
   - ✓ `LONG` y `LONG RAW`.

- ✓ `NCHAR`, `NCLOB` y `NVARCHAR2`.
- ✓ `MLSLABEL` y `ROWID`.
- ✓ Los tipos específicos de PL/SQL: `BINARY INTEGER` (y cualquiera de sus subtipos), `BOOLEAN`, `PLS_INTEGER`, `RECORD`, `REF CURSOR`, `%TYPE` y `%ROWTYPE`.
- ✓ Los tipos definidos en los paquetes PL/SQL.

4. Tampoco se puede inicializar un atributo en la declaración empleando el operador de asignación o la cláusula `DEFAULT`.
5. Del mismo modo, no se puede imponer la restricción `NOT NULL`.
6. Sin embargo, los objetos se pueden almancenar en tablas de la base de datos en las que sí es posible imponer restricciones.

Las estructuras de datos pueden llegar a ser muy complejas:
- ✓ El tipo de un atributo puede ser otro tipo de objeto (denominado entonces tipo de objeto anidado).
- ✓ Esto permite construir tipos de objeto complejos a partir de objetos simples.
- ✓ Algunos objetos, tales como colas, listas y árboles son dinámicos (pueden crecer a medida que se utilizan).
- ✓ También es posible definir modelos de datos sofisticados utilizando tipos de objeto recursivos, que contienen referencias directas o indirectas a ellos mismos.

## Métodos

Un método es un subprograma declarado en una especificación de tipo mediante la palabra clave `MEMBER`.

El método no puede tener el mismo nombre que el tipo de objeto ni el de ninguno de sus atributos.

Muchos métodos constan de dos partes: especificación y cuerpo.
- ✓ La especificación consiste en el nombre del método, una lista opcional de parámetros y en el caso de funciones un tipo de retorno.
- ✓ El cuerpo es el código que se ejecuta para llevar a cabo una operación específica.

Para cada especificación de método debe existir el cuerpo del método.

El PL/SQL compara la especificación del método y el cuerpo token a token, por lo que las cabeceras deben coincidir.

Los métodos pueden hacer referencia a los atributos y a los otros métodos sin cualificador:

```
CREATE TYPE Stack AS OBJECT (
   top INTEGER,
   MEMBER FUNCTION full RETURN BOOLEAN,
   MEMBER FUNCTION push(n IN INTEGER) ,
   . . .
) ;
/
CREATE TYPE BODY Stack AS
   . . .
   MEMBER FUNCTION push (n IN INTEGER) IS
   BEGIN
      IF NOT full THEN
         top := top + 1 ;
. . .
      END push;
   END;
/
```

## El parámetro SELF

Todos los métodos de un tipo de objeto aceptan como primer parámetro una instancia predefinida del mismo tipo denominada `SELF`.

Independientemente de que se declare implicita o explícitamente, `SELF` es siempre el primer parámetro pasado a un método.

Por ejemplo, el método transform declara `SELF` como un parámetro `IN OUT`:

```
CREATE TYPE Complex AS OBJECT (
   MEMBER FUNCTION t rans form (SELF IN OUT Complex ) . . .
```

El modo de acceso de `SELF` cuando no se declara explícitamente es:

- ✓ En funciones miembro el acceso de `SELF` es `IN`.
- ✓ En procedimientos, si `SELF` no se declara, su modo por omisión es `IN OUT`.

**En el cuerpo de un método, `SELF` denota al objeto a partir del cual se invocó el método.**

Los métodos pueden hacer referencia a los atributos de `SELF` sin necesidad de utilizar un cualificador:

```
CREATE FUNCTION gcd ( x INTEGER, y INTEGER) RETURN INTEGER AS
-- Encuentra e l máximo común d i v i s o r de x e y
   ans INTEGER;
BEGIN
   IF x < y THEN ans := gcd(x , y ) ;
   ELSE ans := gcd (y , x MOD y ) ;
   ENDIF;
   RETURN ans ;
END;
/
CREATE TYPE Rational AS
   num INTEGER,
   den INTEGER,
   MEMBER PROCEDURE normalize ,
   . . .
) ;
/
CREATE TYPE BODY Rational AS
   MEMBER PROCEDURE normalize IS
   g INTEGER;
   BEGIN
      -- Estas dos sentencias son equivalentes
      g := gcd(SELF.num, SELF.den ) ;
      g := gcd(num, den ) ;
      num := num / g ;
      den := den / g ;
   END normalize;
   . . .
END;
/
```

## Sobrecarga

Los métodos del mismo tipo (funciones y procedimientos) se pueden sobrecargar: es posible utilizar el mismo nombre para métodos distintos si sus parámetros formales difieren en número, orden o tipo de datos.

Cuando se invoca uno de los métodos, el PL/SQL encuentra el cuerpo adecuado comparando la lista de parámetros actuales con cada una de las listas de parámetros formales.

La operación de sobrecarga no es posible en las siguientes circunstancias:

- ✓ Si los parámetros formales difieren sólo en el modo.
- ✓ Si las funciones sólo difieren en el tipo de retorno.

## Métodos MAP y ORDER

Los valores de un tipo escalar, como `CHAR` o `REAL`, tienen un orden predefinido que permite compararlos.

Las instancias de un objeto carecen de un orden predefinido. Para ordenarlas, el PL/SQL invoca a un método de `MAP` definido por el usuario.

En el siguiente ejemplo, la palabra clave `MAP` indica que el método `convert` ordena los objetos `Rational` proyectándolos como números reales:

```
CREATE TYPE Rational AS OBJECT (
   num INTEGER,
   den INTEGER,
   MAP MEMBER FUNCTION convert RETURN REAL,
   . . .
) ;
/
CREATE TYPE BODY Rat ional AS
   MAP MEMBER FUNCTION convert RETURN REAL IS
   -- Convierte un numero racional en un real
   BEGIN
      RETURN num / den ;
   END convert;
   . . .
END;
/
```

El PL/SQL usa esta función para evaluar expresiones booleanas como x >y y para las comparaciones implícitas que requieren las cláusulas `DISTINCT`, `GROUP BY` y `ORDER BY`.

Un tipo de objeto puede contener sólo una función de `MAP`, que debe carecer de parámetros y debe devolver uno de los siguientes tipos escalares: `DATE`, `NUMBER`, `VARCHAR2` y cualquiera de los tipos ANSI SQL (como `CHARACTER` o `REAL`).

Alternativamente, es posible definir un método de ordenación (`ORDER`). Un método `ORDER` utiliza dos parámetros: el parámetro predefinido `SELF` y otro objeto del mismo tipo. En el siguiente ejemplo, la palabra clave `ORDER` indica que el método match compara dos objetos.

**Ejemplo:**

`c1` y `c2` son objetos del tipo `Customer`. Una comparación del tipo `c1>c2` invoca al método `match` automáticamente. El método devuelve un número negativo, cero o positivo (`SELF` es menor, igual o mayor que el otro parámetro):

```
CREATE TYPE Customer AS OBJECT (
   id NUMBER,
   name VARCHAR2(20) ,
   addr VARCHAR2(30) ,
   ORDER MEMBER FUNCTION match(c Customer ) RETURN INTEGER
) ;
/
CREATE TYPE BODY Customer AS
   ORDER MEMBER FUNCTION match(c Customer ) RETURN INTEGER IS
   BEGIN
      IF id < c.id THEN
         RETURN -1; -- Cualquier número negativo vale.
      ELSEIF id > c.id THEN
         RETURN 1; -- Cualquier número positivo vale.
      ELSE
         RETURN 0;
      END IF;
   END;
END;
/
```

Un tipo de objeto puede contener un único método ORDER, que es una función que devuelve un resultado numérico.

Es importante tener en cuenta los siguientes puntos:
- ✓ Un método `MAP` proyecta el valor de los objetos en valores escalares. Un método `ORDER` compara el valor de un objeto con otro.
- ✓ Se puede declarar un método `MAP` o un método `ORDER`, pero no ambos.
- ✓ Si se declara uno de los dos métodos, es posible comparar objetos en SQL o en un procedimiento. Si no se declara ninguno, sólo es posible comparar la igualdad o desigualdad de dos objetos y sólo en SQL. Dos objeto sólo son iguales si los valores de sus atributos son iguales.
- ✓ Cuando es necesario ordenar un número grande de objetos es mejor utilizar un método `MAP` (ya que una llamada por objeto proporciona una proyección escalar que es más fácil de ordenar). `ORDER` es menos eficiente: debe invocarse repetidamente ya que compara sólo dos objetos cada vez.

### Constructores

Cada tipo de objeto tiene un constructor: función definida por el sistema con el mismo nombre que el objeto. Se utiliza para inicializar y devolver una instancia de ese tipo de objeto.

Oracle genera un constructor por omisión para cada tipo de objeto. Los parámetros del constructor coinciden con los atributos del tipo de objeto: los parámetros y los atributos se declaran en el mismo orden y tienen el mismo nombre y tipo.

**PL/SQL nunca invoca al constructor implícitamente: el usuario debe invocarlo explícitamente.**

### Pragma RESTRIC_REFERENCES

Para ejecutar una sentencia SQL que invoca a una función miembro, Oracle debe conocer el nivel de **pureza** de la función: la medida en que la función está libre de **efectos colaterales**.

Los **efectos colaterales** pueden impedir:
- ✓ La paralelización de una consulta.
- ✓ Dar lugar a resultados dependientes del orden (y por tanto indeterminados).
- ✓ Requerir que un módulo mantenga un cierto estado entre diferentes sesiones de usuario.

Una función miembro debe cumplir las siguientes características:
- ✓ No puede insertar, actualizar o borrar las tablas de la base de datos.
- ✓ No se puede ejecutar en paralelo o remotamente si lee o escribe los valores de una variable en un módulo.
- ✓ No puede escribir una variable de un módulo excepto si se invoca desde una cláusula `SELECT`, `VALUES` o `SET`.
- ✓ No puede invocar a otro método o subprograma que rompa alguna de las reglas anteriores. Tampoco puede hacer referencias a una vista que imcumpla estas reglas.
- ✓ La directiva de compilación `PRAGMA_REFERENCES` se utiliza para forzar las reglas anteriores.
- ✓ La sentencia `PRAGMA` indica al compilador PL/SQL que debe denegar a la función miembro el acceso a las tablas de la base de datos, variables de un paquete o ambos.
- ✓ La sentencia pragma se codifica después del método sobre el que actúa en la especificación del tipo de objeto.

Sintáxis:

```
PRAGMA RESTRICT_REFERENCES ({DEFAULT | nombre_método } ,
{RNDS | WNDS | RNPS | WNPS} [,{RNDS | WNDS | RNPS | WNPS} ]...) ;
```

Ejemplo:

La siguiente sentencia pragma impide al método de `MAP convert`:
- ✓ Leer el estado de la base de datos (Read No Database State).
- ✓ Modificar el estado de la base de datos (Write No Database State)
- ✓ Leer el estado de un paquete o módulo (Read No Package State).

✓ Modificar el estado de un paquete (Write No Package State):

```
CREATE TYPE Rational AS OBJECT (
   num INTEGER,
   den INTEGER,
   MAP MEMBER FUNCTION convert RETURN REAL,
      . . .
   PRAGMA RESTRICT REFERENCES ( convert,RNDS,WNDS,RPNS,WNPS)
) ;
/
```

**Un método sólo se puede invocar en consultas paralelas si se indican las cuatro limitaciones anteriores.**

Si se utiliza la palabra clave `DEFAULT` en lugar del nombre del método, la sentencia pragma se aplica a todas las funciones miembro incluído el constructor definido por el sistema.
Ejemplo:

La siguiente sentencia `pragma` limita a todas las funciones miembro la modificación del estado de la base de datos o de los paquetes:

```
PRAGMA RESTRICT_REFERENCES (DEFAULT, WNDS, WNPS)
```

Es posible declarar un pragma para cada función miembro, que predomina sobre cualquier pragma por omisión definido para el objeto.

## Declaración e inicialización de objetos

Una vez que se ha definido un tipo de objeto y se ha instalado en el esquema de la base de datos, es posible usarlo en cualquier bloque PL/SQL.

Las instancias de los objetos se crean en tiempo de ejecución.
Estos objetos siguen las reglas normales de ámbito y de instanciación.

- ✓ En un bloque o subprograma, los objetos locales son instanciados cuando se entra en el bloque o subproprograma y dejan de existir cuando se sale.
- ✓ En un paquete, los objetos se instancian cuando se referencia por primera vez al paquete y dejan de existir cuando finaliza la sesión.

### Declaración de objetos

Los tipos de objetos se declaran del mismo modo que cualquier tipo interno.

Ejemplo:

En el bloque que sigue se declara un objeto r de tipo Racional y se invoca al constructor para asignar su valor. La llamada asigna los valores 6 y 8 a los atributos `num` y `den` respectivamente:

```
DECLARE
   r Racional;
   BEGIN
      r := Racional(6,8);
      DBMS_OUTPUT.PUT_LINE(r.num); -- muestra 6
```

También es posible declarar objetos como parámetros formales de funciones y procedimientos, de modo que es posible pasar objetos a los subprogramas almacenados y de un subprograma a otro.

Ejemplos:

1. Aquí se emplea un objeto de tipo `Account` para especificar el tipo de dato de un parámetro formal:

```
DECLARE
```

```
   . . .
   PROCEDURE open_acct(new_acct IN OUT Account) IS ...
```

2. En el siguiente ejemplo se declara una función que devuelve un objeto de tipo `Account`:

```
DECLARE
   . . .
   FUNCTION get_acct(acct_id IN INTEGER)
      RETURN Account IS ...
```

## Inicialización de objetos

Hasta que se inicializa un objeto, invocando al constructor para ese tipo de objeto, el objeto se dice que es Atómicamente nulo:
El propio objeto es nulo, no sólo sus atributos.
Un objeto nulo siempre es diferente a cualquier otro objeto. De hecho, la comparación de un objeto nulo con otro objeto siempre resulta NULL.
Si se asigna un objeto con otro objeto atómicamente nulo, el primero se convierte a su vez en un objeto atómicamente nulo (y para poder utilizarlo debe ser reinicializado).
En general, si asignamos el no-valor NULL a un objeto, éste se convierte en atómicamente nulo
Ejemplo:

```
DECLARE
   r Racional ;
BEGIN
   r Racional := Racional (1,2); -- r = 1/2
   r := NULL; -- r atómicamente nulo
   IF r IS NULL THEN ... -- la condición resulta TRUE
```

Una buena práctica de programación consiste en inicializar los objetos en su declaración, como se muestra en el siguiente ejemplo:

```
DECLARE
r Racional := Racional (2,3); -- r = 2/3
```

## Objetos sin inicializar en PL/SQL

PL/SQL se comporta del siguiente modo cuando accede a objetosvsin inicializar:

- ✓ Los atributos de un objeto no inicializado se evalúan envcualquier expresión como `NULL`.
- ✓ Intentar asignar valores a los atributos de un objeto sin inicializar provoca la excepción predefinida `ACCESS_INTO_NULL`.
- ✓ La operación de comparación `IS NULL` siempre produce `TRUE` cuando se aplica a un objeto no inicializado o a cualquiera de sus atributos.

Existe por tanto, una sutil diferencia entre objetos nulos y objetos con atributos nulos. El siguiente ejemplo intenta ilustrar esa diferencia:

```
DECLARE
   r Relacional ; -- r es atómicamente nulo
   BEGIN
      IF r IS NULL THEN ... -- TRUE
      IF r.num IS NULL THEN ... -- TRUE
      r := Racional (NULL,NULL) ; -- Inicializar
      r.num = 4 ;    -- Exito: r ya no es atómicamente nulo aunque
                     -- sus atributos son nulos
      r := NULL;     -- r es de nuevo atómicamente nulo
      r.num := 4 ;   -- Provoca la excepción ACCESS INTO NULL
   EXCEPTION
      WHEN ACCESS INTO NULL THEN
         ...
   END;
/
```

La invocación de los métodos de un objeto no inicializado está permitida, pero en este caso:

- ✓ `SELF` toma el valor `NULL`.
- ✓ Cuando los atributos de un objeto no inicializado se pasan como parámetros `IN`, se evalúan como `NULL`.

- ✓ Cuando los atributos de un objeto no inicializado se pasan como parámetros `OUT` o `IN OUT`, se produce una excepción si se intenta asignarles un valor.

### Acceso a los atributos

Para acceder o cambiar los valores de un atributo se emplea la notación punto ('.'):

```
DECLARE
   r Racional := Racional (NULL, NULL) ;
   numerador INTEGER;
   denominador INTEGER;
BEGIN
   ...
   denominador := r.den ;
   r.num = numerador ;
```

Los nombres de los atributos pueden encadenarse, lo que permite acceder a los atributos de un tipo de objeto anidado. Por ejemplo, supongamos que definimos los tipos de objeto `Address` y `Student` como sigue:

```
CREATE TYPE Address AS OBJECT (
   street      VARCHAR2(30) ,
   city        VARCHAR2(20) ,
   state       CHAR(2) ,
   zip_code    VARCHAR2(5)
) ;
/
CREATE TYPE Student AS OBJECT (
   name           VARCHAR2(20) ,
   home_address   Address ,
   phone_number   VARCHAR2(10) ,
   status         VARCHAR2(10) ,
   advisor name   VARCHAR2(20) ,
   . . .
) ;
/
```

`zip code` es un atributo de tipo Address
`Address` es el tipo de dato del atributo `home address` del tipo de objeto `Student`.
Si s es un objeto `Student`, para acceder al valor de su `zip code` se emplea la siguiente notación:
`s.home_address.zip_code`

## Invocación de constructores y métodos

La invocación de un constructor está permitida en cualquier punto en donde se puede invocar una función.
Como las funciones, un constructor se invoca como parte de una expresión.

```
DECLARE
   r1 Racional := Racional (2,3) ;
   FUNCTION average(x Racional, y Racional)
      RETURN Racional IS
   BEGIN
      ...
   END;
BEGIN
   r1 := average(Racional(3,4), Racional(7,11)) ;
   IF (Racional(5,8) > r1) THEN
      ...
   END IF ;
END;
/
```

### Paso de parámetros a un constructor

Cuando se pasan parámetros a un constructor la invocación asigna valores iniciales a los atributos del objeto que se está instanciando.

Es necesario suministrar parámetros para cada uno de los atributos ya que, a diferencia de las constantes y variables, los atributos carecen de la cláusula `DEFAULT`.

También es posible invocar al constructor utilizando la notación con nombre en lugar de la notación posicional, como se muestra:

```
BEGIN
   r := Racional (den => 6, num => 5);
   -- asigna num = 5 y den = 6
```

## Invocación de métodos

Los métodos se invocan usando la notación punto.

Ejemplo: invocación del método normaliza que divide los atributos `num` y `den` por el mayor común divisor:

```
DECLARE
   r Racional ;
BEGIN
   r := Racional(6,8);
   r.normaliza;
   DBMS_OUTPUT.PUT_LINE(r.num) ; -- muestra 3
```

Es posible encadenar las llamadas a los métodos:

```
DECLARE
   r Racional := Racional(6,8);
BEGIN
   r.reciproco().normaliza;
   DBMS_OUTPUT.PUT_LINE(r.num); -- muestra 4
```

La ejecución se realiza de izquierda a derecha: primero se invoca la función reciproco y después la función normaliza.

En las sentencias SQL la invocación de un métodos sin parámetros requiere la lista vacia de parámetros: '()'.

En sentencias de procedimiento la lista vacia de parámetros es opcional, excepto cuando se encadenan llamadas: es obligatoria para todas las llamadas excepto la última.

No es posible encadenar invocaciones a métodos adicionales a la derecha de la invocación de un procedimiento (los procedimientos no son parte de una expresión). La siguiente sentencia es ilegal:

```
r.normaliza().reciproco; -- i l e g a l
```

Cuando se encadenan dos llamadas a función, el resultado de la primera función debe ser un objeto que puede ser pasado a la segunda función.

## Compartición de objetos

La mayoría de los objetos del mundo real son considerablemente más grandes y complejos que el tipo Relacional. Por ejemplo, consideremos los siguientes tipos de objeto:

```
CREATE TYPE Address AS OBJECT (
   street_address    VARCHAR2( 35 ) ,
   city              VARCHAR2( 15 ) ,
   state             CHAR( 2 ) ,
   zip code          INTEGER
) ;
/
CREATE TYPE Person AS OBJECT (
   first_name        VARCHAR2( 15 ) ,
   last_name         VARCHAR2( 15 ) ,
   birthday          DATE,
   home_address      Address , -- Objeto anidado
   pone_number       VARCHAR2( 1 5 ) ,
   ss_number         INTEGER
```

```
   ...
) ;
/
```

Los objetos de tipo `Address` tienen más del doble de atributos que los del tipo Relacional y los objetos de tipo `Person` todavía tienen más atributos, incluyendo uno de tipo `Address`.

Cuando se utilizan objetos grandes, resulta ineficiente pasar copias de él entre subprogramas. En estas circunstancias tiene más sentido compartir el objeto.

Esto se puede hacer si el objeto cuenta con un identificador de objeto.

Para compartir objetos se utilizan referencias (`refs` de forma abreviada). Una `ref` es un puntero al objeto.

La compartición de objetos proporciona dos ventajas importantes:
- ✓ La información no se duplican innecesariamente.
- ✓ Cuando se actualiza un objeto compartido, el cambio se produce sólo en un lugar y cualquier referencia al objeto puede recuperar los valores actualizados inmediatamente.

En el ejemplo siguiente, obtenemos las ventajas de la compartición definiendo el tipo de objeto Home y creando una tabla que almacena las instancias de ese tipo:

```
CREATE TYPE Home AS OBJECT (
   address     VARCHAR2(35),
   owner       VARCHAR2(25),
   age         INTEGER,
   style       VARCHAR(15),
   floor_plan  BLOB,
   price       REAL(9,2),
   . . .
) ;
/
. . .
CREATE TABLE homes OF Home;
```

## Utilización de referencias

Revisando la definición anterior del objeto de tipo `Person`, observamos que podemos diseñar una comunidad que puede compartir la misma casa (`Home`).

```
CREATE TYPE Person AS OBJECT (
   first_name     VARCHAR2(15),
   last_name      VARCHAR2(15),
   birthday       DATE,
   home_address   REF Home , -- Compartido con la familia
   phone number   VARCHAR2( 15 ) ,
   ss_number      INTEGER
   mother         REF Person , -- Miembros de la familia
   father         REF Person ,
   ...
) ;
/
```

Note cómo las referencias entre `Person` y `Homes` y entre `Person` entre sí definen relaciones que se dan en el mundo real.

Es posible declarar referencias como variables, parámetros, campos o atributos.

Se pueden utilizar referencias como parámetros `IN` y `OUT` en funciones y procedimientos.

Pero no es posible navegar a través de referencias.

Ejemplo: un intento ilegal de navegar a través de una referencia a un objeto:

```
DECLARE
   p_ref    REF Person ;
   pone_no VARCHAR2( 1 5 ) ;
BEGIN
   ...
   pone_no = p_ref.phone_number ; -- Ilegal!
```

Para llevar a cabo esta operación es necesario utilizar el operador `DEREF`, a través del cual se puede acceder al objeto.

## Limitaciones en la definición de tipos

En la creación de un tipo sólo se puede hacer referencia a objetos que ya existan en el esquema de objetos.

En el siguiente ejemplo la primera sentencia `CREATE TYPE` es ilegal porque hace referencia al objeto de tipo Departament que todavía no existe:

```
CREATE TYPE Employee AS OBJECT (
   name VARCHAR2(20),
   dept REF Departament, -- Ilegal!
   ...
) ;
/
CREATE TYPE Departament AS OBJECT (
   number   INTEGER,
   manager REF Employee,
   ...
) ;
/
```

Cambiar el orden de las sentencias `CREATE TYPE` no soluciona el problema, ya que ambos tipos son mutuamente dependientes.

Para resolver el problema se utiliza una sentencia `CREATE TYPE` especial denominada definición previa de tipo, que permite la creación de tipos de objetos mutuamente dependientes.

```
CREATE TYPE Departament;   -- Definición previa de tipo
                           -- En este punto, Departament es un tipo de objeto incompleto
```

El tipo creado mediante una definición previa de tipo se denomina tipo de objeto incompleto ya que carece de atributos y métodos hasta que se defina en su totalidad.

Un tipo incompleto impuro cuenta con atributos, pero compila con errores semántico (no sintácticos) al hacer referencia a un tipo indefinido. Por ejemplo, la siguiente sentencia `CREATE TYPE` compila con errores debido a que el tipo de objeto Address todavía no está definido:

```
CREATE TYPE Customer AS OBJECT (
   id       NUMBER,
   name     VARCHAR2(20) ,
   addr     Address , -- todavía indefinido
   phone    VARCHAR2(15)
) ;
/
```

Esto permite retrasar la definición del tipo de objeto `Address`.

Más aún, las referencias al tipo incompleto Customer están disponibles para otras aplicaciones.

## Manipulación de objetos

Es posible utilizar un tipo de objeto en una sentencia `CREATE TABLE` para especificar el tipo de una columna.

Una vez que la tabla se ha creado, se pueden utilizar las sentencias SQL para insertar un objeto, seleccionar sus atributos, invocar los métodos definidos y actualizar su estado.

Ejemplos:

1. La sentencia `INSERT` invoca al constructor del tipo Racional para insertar su valor. La sentencia `SELECT` recupera el valor del atributo num y la sentencia `UPDATE` invoca al método reciproco, que devuelve un valor Relacional después de invertir los valores de num y den. Observe que se requiere un alias de la tabla cuando se hace referencia a un atributo o método.

```
CREATE TABLE numbers(rn Racional, ...) ;
INSERT INTO numbers(rn) VALUES(Racional(3,62));
SELECT n.rn.num INTO my num FROM numbers n WHERE ...
UPDATE numbers n SET n.rn = n.rn.reciproco WHERE ...
```

   Cuando se crea un objeto de este modo, carece de identidad fuera de la tabla de la base de datos.

2. En el siguiente ejemplo se crea una tabla que almacena en sus filas objetos del tipo Relacional. Este tipo de tablas, cuyas filas contienen un tipo de objetos, se denominan tablas de objetos. Cada columna en una fila se corresponde con un atributo del tipo de objeto:

```
CREATE TABLE racional_nums OF Racional ;
```

Cada fila en una tabla de objetos cuenta con un identificador de objeto, que identifica de forma unívoca al objeto almacenado en dicha fila y sirve como una refencia al objeto.

### Selección de objetos

Supongamos que ejecutamos el siguiente script de SQL*Plus, que crea un tipo de objeto denominado `Person` y una tabla de objetos `persons` junto con algunos valores:

```
CREATE TYPE Person AS OBJECT (
   first name     VARCHAR2(15),
   last name      VARCHAR2(15),
   birthday       DATE,
   home_address   Address,
   pone_number    VARCHAR2(15) ,
) ;
/
CREATE TABLE persons OF Person ;
/
```

La siguiente subconsulta produce como resultado un conjunto de filas que contienen sólo atributos de los objetos `Person`:

```
BEGIN
   INSERT INTO employees
      -- emplyees es otra tabla de objetos de tipo Person
      SELECT * FROM persons p
         WHERE p.last_name LIKE '%smith';
```

### El operador VALUE

El comando `VALUE` devuelve el valor de un objeto.

`VALUE` requiere como argumento una variable de correlación (en este contexto, variable de correlación es una fila o alias de tabla asociado a una fila en una tabla de objetos).

Ejemplos:

1. Para obtener un conjunto de objetos Person se puede utilizar el comando `VALUES` del siguiente modo:

```
BEGIN
  INSERT INTO employees
     SELECT VALUE(p) FROM persons p
        WHERE p.last_name LIKE '%smith'
```

2. En el siguiente ejemplo, se utiliza el operador `VALUE` para obtener un objeto Person específico:

```
DECLARE
  p1 PERSON;
  p2 PERSON;
BEGIN
  SELECT VALUE(p) INTO p1 FROM persons p
     WHERE p.last name = 'Kroll';
  p2 := p1 ;
  ...
END;
```

   Después de ejecutar la consulta SQL, la variable p1 contiene un objeto Person local, que es una copia del objeto almacenado en la tabla persons. Del mismo modo, p2 contiene otra copia local del objeto.
3. Es posible utilizar las variables anteriores para acceder y modificar el objeto que contienen:

BEGIN

```
p1.last_name := p1.last_name || 'Jr';
```

## El operador REF

El operador `REF` se utiliza para obtener una referencia a un objeto.

Como el operador VALUE, toma como argumento una variable de correlación.

Ejemplos:

1. En el siguiente ejemplo, primero se recuperan una o más referencias a objetos de tipo `Person` y despu¶es se insertan en la tabla `person refs`:

```
BEGIN
  INSERT INTO person_refs
     SELECT REF(p) FROM persons p
        WHERE p.last_name LIKE '%smith';
```

2. En el siguiente ejemplo se obtienen simultáneamente una referencia y un atributo:

```
DECLARE
  p ref          REF Person ;
  taxpayer_id    VARCHAR2( 9 ) ;
BEGIN
  SELECT REF(p ), p.ss number INTO p ref, taxpayer id
     FROM persons p
     WHERE p.last name = 'Parker' ; -- sólo una fila
  ...
END;
```

3. En este último ejemplo, se actualizan los atributos de un objeto `Person`:

```
DECLARE
   p_ref          REF Person;
   my_last_name   VARCHAR2(15) ;
   . . .
BEGIN
   ...
   SELECT REF(p) INTO p_ref FROM persons p
      WHERE p.last_name = my_last_name;
   UPDATE persons p
      SET p = Person ('Jill', 'Anders', '11-NOV-67' , ...)
      WHERE REF(p) = p_ref ;
END;
```

## Referencias colgadas (dangling refs)

Si el objeto al cual apunta una referencia es borrado, la referencia queda "*colgada*" (*dangling*, apuntando a un objeto inexistente).

Para comprobar si se produce esta condición se puede utilizar el predicado `SQL IS DANGLING`.

Ejemplo:

Supongamos que la columna `manager` en la tabla relacional `department` contiene referencias a objetos `Employee` almacenados en
una tabla de objetos. Para convertir todas las referencias colgadas en nulos, podemos utilizar la siguiente sentencia `UPDATE`:

```
BEGIN
   UPDATE department SET manager = NULL
      WHERE manager IS DANGLING;
```

### El operador DEREF

No es posible navegar a través de referencias en procedimientos SQL. Para esto es necesario utilizar el operador `DEREF` (abreviatura del término inglés dereference: derreferenciar un puntero es obtener el valor al cual apunta).

`DEREF` toma como argumento una referencia a un objeto y devuelve el valor de dicho objeto. Si la referencia está colgada, `DEREF` devuelve el valor `NULL`.

Ejemplos:

1. En el ejemplo que sigue se derreferencia una referencia a un objeto `Person`. En estas circunstancias, no es necesario especificar una tabla de objetos ni un criterio de búsqueda ya que cada objeto almacenado en una tabla de objetos cuenta con un identificador de objeto único e inmutable que es parte de cada referencia a un objeto.

```
DECLARE
  p1 Person ;
  p_ref REF Person;
  name VARCHAR2(15);
BEGIN
  ...
  /* Supongamos que p_ref contiene una referencia válida
     a un objeto almacenado en una tabla de objetos */
SELECT DEREF(p_ref) INTO p1 FROM DUAL;
name := p1.last_name ;
```

2. Es posible utilizar el operador `DEREF` en sentencias SQL sucesivas para derreferencias referencias, como se muestra en el siguiente ejemplo:

```
CREATE TYPE PersonRef AS OBJECT (p_ref REF Person);
/
DECLARE
   name     VARCHAR2(15);
   pr_ref   REF PersonRef;
   pr       PersonRef;
   p        Person;
BEGIN
   ...
   /* Supongamos que prref contiene
      una referencia válida */
   SELECT DEREF(pr_ref) INTO pr FROM DUAL;
   SELECT DEREF(pr.p ref) INTO p FROM DUAL;
   name := p.last name ;
   . . .
END;
/
```

3. En procedimientos SQL la utilización del operador DEREF es ilegal. En sentencias SQL se puede utilizar la notación punto para navegar a través de referencias. Por ejemplo, la siguiente sentencia es legal:

```
table_alias.object_column.ref_attribute
table alias.object column.ref attribute.attribute
table_alias.ref_column.attribute
```

4. Supongamos ahora que ejecutamos el siguiente script SQL*Plus que crea los tipos de objeto `Address` y `Person` y la tabla de objetos `persons`:

```
CREATE TYPE Address AS OBJECT (
     street_address    VARCHAR2(35),
     city              VARCHAR2(15),
     state             CHAR(2),
     zip_code          INTEGER
) ;
```

```
/
CREATE TYPE Person AS OBJECT (
    first name     VARCHAR2(15),
    last name      VARCHAR2(15),
    birthday       DATE,
    home_address   Address ,
    phone_number   VARCHAR2(15),
) ;
/
CREATE TABLE persons OF Person;
/
```

El atributo `home address` es una referencia a una columna en la tabla de objetos `persons`, que a su vez contiene referencias a objetos `Address` almacenados en otra tabla indeterminada.

Tras introducir algunos elementos en la tabla, es posible obtener una dirección particular derreferenciando su referencia, como se muestra en el siguiente ejemplo:

```
DECLARE
    addr1 Address,
    addr2 Address,
    ...
BEGIN
    SELECT DEREF(home_address) INTO addr1 FROM persons p
       WHERE p.last_name = 'Derringer';
```

5. Por último, en este ejemplo se navega a través de la columna de referencias `home address` hasta el atributo `street`. En este caso se requiere un alias a la tabla:

```
DECLARE
    my_street VARCHAR2(25),
    ...
BEGIN
    SELECT p.home address.street INTO my street
       FROM persons p
          WHERE p.last_name = 'Lucas';
```

## Inserción de objetos

Para añadir objetos a una tabla de objetos se utiliza el comando `UPDATE`.
Ejemplos:

1. Para insertar un objeto `Person` en la tabla de objetos `persons` utilizamos la siguiente línea:

```
BEGIN
  INSERT INTO persons
     VALUES ('Jenifer', 'Lapidus', ...);
```

2. Alternativamente, es posible utilizar el constructor para el objeto de tipo `Person`:

```
BEGIN
  INSERT INTO persons
     VALUES (Person('Albert', 'Brooker', ...));
```

3. En el siguiente ejemplo, se utiliza la cláusula `RETURNING` para almacenar una referencia a `Person` en una variable local. Es importante destacar como esta cláusula simula una sentencia `SELECT`. La cláusula `RETURNING` se puede utilizar también en sentencias `UPDATE` y `DELETE`.

```
DECLARE
  p1_ref REF Person,
  p2 ref REF Person,
  ...
BEGIN
  INSERT INTO persons p
     VALUES(Person('Paul', 'Chang', ...))
     RETURNING REF(p) INTO p1_ref ;
  INSERT INTO persons p
     VALUES (Person('Ana', 'Thorne' ,...))
     RETURNING REF(p) INTO p2_ref ;
```

4. Para insertar objetos en una tabla de objetos se puede utilizar una consulta que devuelva un objeto del mismo tipo, como se muestra en el siguiente ejemplo:

```
BEGIN
  INSERT INTO persons2
     SELECT VALUE(p) FROM persons p
        WHERE p.last_name LIKE '%Jones';
```

Las filas copiadas a la tabla de objetos `persons2` cuentan con identificadores de objeto nuevos, ya que los identificadores de objeto son únicos.

5. El siguiente script crea la tabla relacional `department` que cuenta con una columna de tipo `Person`; después inserta una fila en la tabla. Es importante destacar cómo el constructor `Person()` proporciona un valor para la columna manager:

```
CREATE TABLE department (
   dept_name   VARCHAR2(20),
   manager     Person,
   location    VARCHAR2(20));
/
INSERT INTO department
   VALUES('Payroll',
      Person('Alan', 'Tsai', ...),
      'Los Angeles');
/
```

El nuevo objeto `Persona` almacenado en la columna `manager` no es referenciable, ya que al estar almacenada en una columna (y no en una fila) carece de identificador de objeto.

### Actualización de objetos

Para modificar los atributos de un objeto en una tabla de objetos se utiliza la sentencia `UPDATE`
Ejemplo:

```
BEGIN
   UPDATE persons p SET p.home_address = '341 Oakdene Ave'
      WHERE p.last name = 'Brody';
      ...
   UPDATE persons p SET p = Person('Beth', 'Steinberg', ...)
      WHERE p.last_name = 'Steinway';
      . . .
END;
```

### Borrado de objetos

Para eliminar objetos (filas) de una tabla de objetos se utiliza la sentencia `DELETE`.
Para eliminar objetos selectivamente se utiliza la cláusula `WHERE`.
Ejemplo:

```
BEGIN
   DELETE FROM persons p
      WHERE p.home address = '108 Palm Dr';
      . . .
END;
```

# Anexo IV - Bases de Datos Objeto-Relacionales en Oracle 8

**Original en: http://www3.uji.es/~mmarques/e16/teoria/lib_cap9.pdf**

## 1. Introducción

Debido a los requerimientos de las nuevas aplicaciones, en su octava versión, el sistema gestión de bases de datos relacionales Oracle 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. Por esta razón, se dice que se trata de un modelo de datos objetorelacional.

Oracle 8 proporciona mecanismos para que el usuario pueda definir sus propios tipos de datos, cuya estructura puede ser compleja, y que se pueden aplicar para asignar un tipo a una columna de una tabla. También 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 8 también proporciona mecanismos para asociar métodos a tipos, y constructores para diseñar tipos de datos multivaluados (colecciones) y tablas anidadas. La mayor deficiencia de este sistema es la imposibilidad de definir jerarquías de especialización y herencia, lo cual es una importante desventaja con respecto a las bases de datos orientadas a objetos.

## 2. Tipos de Datos Definidos por el Usuario

Los usuarios de Oracle 8 pueden definir sus propios tipos de datos, pudiendo ser de dos categorías: tipos de objetos (`object types`) y tipos para colecciones (`collection types`). Para construir los tipos de usuario se utilizan los tipos básicos provistos por el sistema y otros tipos de usuario previamente definidos. Un tipo define una estructura y un comportamiento común para un conjunto de datos de las aplicaciones.

### 2.1 Tipos de objetos

Un tipo de objetos representa a una entidad del mundo real. Un tipo de objetos se compone de los siguientes elementos:

- ✓ Un nombre que sirve para identificar el tipo de los objetos.
- ✓ Unos 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.
- ✓ Unos métodos que son procedimientos o funciones escritos en el lenguaje PL/SQL (almacenados en la base de datos), o escritos en C (almacenados externamente).

Los tipos de objetos pueden interpretarse como plantillas a las que se adaptan los objetos de ese tipo. A continuación se da un ejemplo de cómo definir el tipo de datos `Direccion_T` en el lenguaje de definición de datos de Oracle 8, y como utilizar este tipo de datos para definir el tipo de datos de los objetos de la clase de `Clientes_T`.

**DEFINICIÓN ORIENTADA A OBJETOS**

```
define type Direccion_T:
   tuple [calle:string,
   ciudad:string,
   prov:string,
   codpos:string]

define class Clientes_T
   type tuple [clinum: integer,
      clinomb:string,
      direccion:Direccion_T,
      telefono: string,
      fecha-nac:date]
   operations edad():integer
```

**DEFINICIÓN EN ORACLE**

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

CREATE TYPE clientes_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)
) ;
```

## 2.2 Métodos

La especificación de un método se hace junto con 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 `Clientes T`, entonces `x.edad()` calcula la edad del cliente almacenado en `x`. La definición del cuerpo de un método en PL/SQL se hace de la siguiente manera:

```
CREATE OR REPLACE TYPE BODY clientes_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;
```

### *2.2.1 Constructores de tipo*

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')
clientes_t( 2347,
   'José Pérez Ruíz',
   direccion_t('Calle Eo', 'Onda', 'Castellón', '34568'),
   '696-779789',
   12/12/1981
   )
```

### 2.2.2 Métodos de comparación

Para comparar los objetos de cierto tipo es necesario indicar a Oracle cuál es el criterio de comparación. Para hacer esto hay que escoger entre un método `MAP` o `ORDER`, debiéndose definir al menos uno de estos métodos por cada tipo de objetos que necesiten ser comparados. La diferencia entre ellos es la siguiente:

- ✓ Un método de `MAP` sirve para indicar cuál de los atributos del tipo se va a utilizar para ordenar los objetos del tipo, y por lo tanto se puede utilizar para comparar los objetos de ese tipo por medio de los operadores de comparación típicos (**<**, **>**). Por ejemplo la siguiente declaración permite decir que los objetos del tipo `clientes_t` se van a comparar por su atributo clinum.

```
CREATE TYPE clientes 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 clientes t AS
   MAP MEMBER FUNCTION ret_value RETURN NUMBER IS
      BEGIN
         RETURN clinum
      END;
   END;
```

- ✓ 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 debe devolver un valor negativo si el primero es mayor que el segundo, un valor positivo si ocurre lo contrario y un cero si ambos son iguales. El siguiente ejemplo define un orden para el tipo `clientes t` diferente al anterior. Solo una de estas definiciones puede ser válida a un tiempo.

```
CREATE TYPE clientes_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 clientes t AS
   ORDER MEMBER FUNCTION cli_ordenados (x IN clientes_t)
      RETURN INTEGER IS
      BEGIN
         RETURN clinum - x.clinum; /*la resta de los dos números clinum*/
      END;
   END;
```

Para un tipo de objetos que no tenga definido ninguno de estos métodos, Oracle es incapaz de deducir cuándo un objeto es mayor o menor que otro. Sin embargo sí que puede determinar cuándo dos objetos del mismo tipo son iguales. Para hacer esto, 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.

## 2.3 Tablas de objetos

Después de definir los tipos, éstos pueden utilizarse para definir otros 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 clientes_t
   (clinum PRIMARY KEY);
CREATE TABLE clientes_antiguos_tab (
   año NUMBER,
   cliente clientes t
   ) ;
```

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 objetos. 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.

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,
'José 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 = 'José 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.

## 2.4 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 objetos, se podrá actualizar a una referencia a un objeto del tipo adecuado independientemente 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.

```
CREATE TABLE clientes_tab OF clientes_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
);
```

Cuando se borran objetos de la base de datos, puede ocurrir que otros objetos que referencien a los borrados queden en un estado inconsistente. Estas referencias se denominan *dangling references*, y Oracle proporciona un predicado que permite comprobar cuando sucede esto. El predicado se denomina `IS DANGLING`.

## 2.5 Tipos para colecciones

Para poder implementar relaciones `1:N`, en Oracle 8 es posible definir tipos para colecciones. Un dato de tipo colección está formado por un número indefinido de elementos, todos del mismo tipo. De esta manera en un atributo es posible almacenar un conjunto de tuplas en forma de array (`VARRAY`), o en forma de tabla anidada.

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

Los argumentos de entrada de estas funciones son el conjunto de elementos que forman la colección separados por comas y entre paréntesis, y el resultado es un valor del tipo colección. 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.

### *2.5.1 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');
```

Un tipo `VARRAY` se puede utilizar para:

- ✓ Definir el tipo de datos de una columna de una tabla relacional.
- ✓ Definir el tipo de datos de un atributo de un tipo de objetos.
- ✓ 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`.
En el siguiente ejemplo, se quiere definir un tipo de datos para almacenar una lista ordenada de teléfonos (tipo `list`, en el tipo `set` no existe orden). Este tipo se utiliza después para asignárselo a un atributo del tipo de objetos `clientes_t`.

| DEFINICIÓN ORIENTADA A OBJETOS | DEFINICIÓN EN ORACLE |
|---|---|
| `define type Lista_Tel_T: list(string);`<br><br>`define class Clientes_T:`<br>`  tuple [clinum: integer,`<br>`  clinomb:string,`<br>`  direccion:Direccion T,`<br>`  lista_tel: Lista_Tel_T];` | `CREATE TYPE lista_tel_t AS`<br>`  VARRAY(10) OF VARCHAR2(20) ;`<br>`CREATE TYPE clientes_t AS OBJECT (`<br>`  clinum NUMBER,`<br>`  clinomb VARCHAR2(200),`<br>`  direccion direccion t,`<br>`  lista_tel lista_tel_t );` |

La principal limitación que presenta los tipos `VARRAY` es que en las consultas es imposible poner condiciones sobre los elementos almacenados dentro. Desde una consulta SQL, los valores de un `VARRAY` sólamente pueden ser accedidos y recuperados en un bloque. Es decir no se puede acceder a los elementos de un `VARRAY` individualmente. Sin embargo desde un programa PL/SQL si que es posible definir un bucle que itere sobre los elementos de un `VARRAY` (ver sección 4.2.5).

### 2.5.2 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 objetos 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 objetos.
En el siguiente ejemplo, se declara una tabla que después es anidada en el tipo `ordenes_t`. Los pasos de todo el diseño son los siguientes.

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

```
define type Linea_T:
  tuple [linum:integer,
  item:string,
  cantidad:integer,
  descuento:real];
```

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

**2-** Se define el tipo de 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 datos de una columna de una tabla relacional.
- ✓ Definir el tipo de datos de un atributo de un tipo de objetos.
- ✓ Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

**3-** Se define el tipo de objetos `ordenes t` que en el atributo pedido almacena una tabla anidada del tipo `lineas pedido t`.

```
define class Ordenes T:
   tuple [ordnum:integer,
   cliente:Clientes T,
   fechpedido:date,
   fechentrega:date,
   pedido:set(Linea_T),
   direcentrega:Direccion_T];
```

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

**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 hacerlo 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 lineas 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 órdenes se almacenan externamente a la tabla de órdenes, 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`).

Al contrario que los VARRAY, los elementos de las tablas anidadas si pueden ser accedidos individualmente, y es posible poner condiciones de recuperación sobre ellos.

Como veremos en la próxima sección, una forma conveniente de acceder individualmente a los elementos de una tabla anidada es por medio de un cursor anidado. Además, las tablas anidadas pueden ser indexadas.

# 3. Inserción y Acceso a los Datos

## 3.1 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 una consulta. Los alias se utilizan para acceder al contenido de las tablas, pero hay que tener cuidado de 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);
```

La diferencia entre las dos 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;` **BIEN**
2. `SELECT c1.nombre FROM ptab2;` **MAL**
3. `SELECT p.c1.nombre FROM ptab2 p;` **BIEN**
4. `SELECT p.c1.nombre FROM ptab3 p;` **BIEN**
5. `SELECT p.nombre FROM ptab3 p;` **MAL**

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 5 muestran cómo acceder a los atributos de los objetps referenciados desde un atributo de la tabla ptab3.

En conclusión, para facilitar la formulación de consultas y evitar errores se recomienda utilizar alias para acceder a todas las tablas que contengan objetos con o sin identidad, y para acceder a las columnas de las tablas en general.

## 3.2 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 instrucción inserta una orden en la tabla definida en la sección 2.4.

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

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

Observemos el siguiente ejemplo.

```
CREATE TYPE persona_t AS OBJECT (
   nombre VARCHAR2(30),
   jefe REF persona_t ) ;
```

Si x es una variable que representa a un objeto de tipo persona_t, entonces las dos expresiones siguientes son equivalentes:

```
1. x.jefe.nombre
2. y.nombre, y=DEREF(x.jefe)
```

Para obtener una referencia a un objeto de una tabla de objetos se puede aplicar el operador REF de la manera que se muestra en el siguiente ejemplo:

```
CREATE TABLE persona_tab OF persona_t;
DECLARE ref_persona REF persona_t;
SELECT REF(pe) INTO ref_persona
   FROM persona_tab pe WHERE pe.nombre= 'José Pérez Ruíz';
```

Simétricamente, para recuperar un objeto desde una referencia es necesario usar DEREF, como se muestra en este ejemplo que visualiza los datos del jefe de la persona indicada:

```
SELECT DEREF(pe.jefe)
FROM persona_tab pe WHERE pe.nombre= 'José Pérez Ruíz';
```

## 3.3 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 tienen argumentos hay que especificar los paréntesis aunque estén

vacíos. Por ejemplo, si tb es una tabla con la columna c de tipo de objetos t, y t tiene un método m sin argumentos de entrada, la siguiente consulta es correcta:

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

### 3.4 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 el campo que contiene las tuplas anidadas vacío.
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 cuando dentro de una tabla anidada se guardan referencias a otros objetos. En el siguiente ejemplo se ilustra la manera de realizar estas operaciones sobre la tabla de ordenes (`ordenes_tab`) definida en la sección 2.5.2.

```
INSERT INTO ordenes_tab --inserta una orden
   SELECT 3001, REF(C),
          SYSDATE,'30-MAY-1999',
          lineas_pedido_t(),
          NULL
      FROM clientes_tab C
      WHERE C.clinum= 3 ;
INSERT INTO THE ( --selecciona el atributo pedido de la orden
   SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 3001
   )
   SELECT 30, REF(S), 18, 30 --inserta una linea de pedido anidada
      FROM items_tab S
      WHERE S.itemnum = 3011;
```

Para poner condiciones sobre las tuplas de una tabla anidada, se pueden utilizar cursores dentro de un **SELECT** o desde un programa PL/SQL de la manera explicada en la sección 4.2.5. Veamos aquí un ejemplo de acceso con cursores. Utilizando el ejemplo de la sección 2.5.2, vamos a recuperar el número de las ordenes, sus fechas de pedido y las líneas de pedido que se refieran al item 'CH4P3'.

```
SELECT ord.ordnum, ord.fechpedido,
   CURSOR (SELECT * FROM TABLE(ord.pedido) lp WHERE lp.item= 'CH4P3')
FROM ordenes_tab ord;
```

La cláusula **THE** también sirve para seleccionar las tuplas de una tabla anidada. La sintaxis es como sigue:

```
SELECT ... FROM THE (subconsulta) WHERE ...
```

Por ejemplo, para seleccionar las primeras dos líneas de pedido de la orden 8778 se hace:

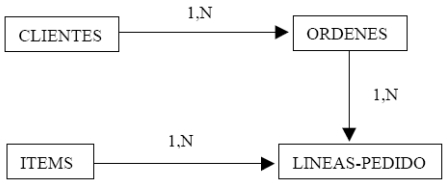
```
SELECT lp FROM THE
(SELECT ord.pedido FROM ordenes_tab ord WHERE ord.ordnum= 8778) lp
WHERE lp.linum<3;
```

## 4. Una Base de Datos Ejemplo

Partiendo de una base de datos para gestionar los pedidos de los clientes, veamos como se puede proporcionar una solución relacional y otra objeto_relacional en Oracle 8.

### 4.1 Modelo lógico para una base de datos relacional

```
CLIENTES(clinum, clinomb, calle, ciudad, prov, codpos, tel1, tel2, tel3)
ORDENES(ordnum, clinum, fechpedido, fechentrega, callent, ciuent, provent, codpent)
ITEMS(numitem, precio, tasas)
LINEAS(linum, ordnum, numitem, cantidad, descuento)
```




### *4.1.1 Implementación relacional con Oracle 8*

Se crean tablas normalizadas y con claves ajenas para representar las relaciones.

```
CREATE TABLE clientes (
   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 ordenes (
   ordnum NUMBER,
   clinum NUMBER REFERENCES clientes,
   fechpedido DATE,
   fechaentrega DATE,
   callent VARCHAR2(200),
   ciuent VARCHAR2(200),
   provent CHAR(2),
   codpent VARCHAR2(20),
   PRIMARY KEY (ordnum)
) ;
CREATE TABLE items (
   numitem NUMBER PRIMARY KEY,
   precio NUMBER,
   tasas NUMBER
) ;
CREATE TABLE lineas (
   linum NUMBER,
   ordnum NUMBER REFERENCES ordenes,
   numitem NUMBER REFERENCES items,
   cantidad NUMBER,
   descuento NUMBER,
   PRIMARY KEY (ordnum, linum)
) ;
```

## 4.2 Modelo lógico para una base de datos orientada a objetos

Primero vamos utilizar el lenguaje de definición de bases de datos orientadas a objetos visto en el tema 5 para definir el esquema de la base de datos que después crearemos en Oracle 8.

```
define type Lista_Tel_T: type list(string);
define type Direccion_T: type tuple [ calle:string,
   ciudad:string,
   prov:string,
   codpos:string];
define class Clientes_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];
define type Linea_T: type tuple [linum:integer,
   item:Item_T,
   cantidad:integer,
   descuento:real];
define type Lineas_Pedido_T: type set(Linea_T);
define class Ordenes_T: type tuple [ ordnum:integer,
   cliente:Clientes T,
   fechpedido:date,
   fechentrega:date,
   pedido:Lineas Pedido T
   direcentrega:Direccion_T];
```

### 4.2.1 Implementación objeto-relacional con Oracle 8

Aquí se indica como definir todos los tipos anteriores en Oracle 8.

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

### 4.2.2 Creación de tablas de objetos

Ahora vamos a crear las tablas donde almacenar los objetos de la aplicación.

```
CREATE TABLE clientes_tab OF clientes_t
   (clinum PRIMARY KEY);
CREATE TABLE items_tab OF item_t
   (itemnum PRIMARY KEY) ;
CREATE TABLE ordenes_tab OF ordenes_t (
   PRIMARY KEY (ordnum),
   SCOPE FOR (cliente) IS clientes_tab
)
NESTED TABLE pedido STORE AS pedidos_tab ;
ALTER TABLE pedidos_tab
   ADD (SCOPE FOR (item) IS items_tab) ;
```

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

### 4.2.3 Inserción de objetos en las tablas

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

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

```
INSERT INTO clientes_tab
```

```
   VALUES (
      1, 'Lola Caro',
      direccion t('12 Calle Lisboa', 'Nules', 'CS', '12678'),
      lista tel t('415-555-1212')
   ) ;
INSERT INTO clientes_tab
   VALUES (
      2, 'Jorge Luz',
      direccion_t('323 Calle Sol', 'Valencia', 'V', '08820'),
      lista tel t('609-555-1212','201-555-1212')
   ) ;
INSERT INTO clientes_tab
   VALUES (
      3, 'Jose Perez',
      direccion t('12 Calle Colon', 'Castellon', 'ES', '12001'),
      lista tel t('964-555-1212', '609-543-1212',
      '201-775-1212','964-445-1212')
   ) ;
INSERT INTO clientes_tab
   VALUES (
      4, 'Ana Gil',
      direccion t('5 Calle Sueca', 'Burriana', 'ES', '12345'),
      lista_tel_t()
   ) ;
REM inserción en la tabla ORDENES_TAB*********************************
```

Nótese como en estas definiciones se utiliza el operador `REF` para obtener una referencia a un objeto de `clientes tab` y almacenarlo en la columna de otro objeto de `ordenes tab`.

La palabra clave `THE` se 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.

```
REM Ordenes del cliente 1*********************************
   INSERT INTO ordenes_tab
      SELECT 1001, REF(C),
         SYSDATE,'10-MAY-1999',
         lineas_pedido_t(),
         NULL
      FROM clientes_tab C
      WHERE C.clinum= 1 ;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 1001
   )
      SELECT 01, REF(S), 12, 0
      FROM items_tab S
      WHERE S.itemnum = 1534;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 1001
   )
      SELECT 02, REF(S), 10, 10
      FROM items_tab S
      WHERE S.itemnum = 1535;
REM Ordenes del cliente 2*********************************
   INSERT INTO ordenes_tab
      SELECT 2001, REF(C),
         SYSDATE,'20-MAY-1999',
         lineas_pedido_t(),
         direccion_t('55 Madison Ave','Madison','WI','53715')
      FROM clientes_tab C
      WHERE C.clinum= 2;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 2001
   )
      SELECT 10, REF(S), 1, 0
      FROM items_tab S
      WHERE S.itemnum = 1004;
```

```
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum= 2001
   )
   VALUES( linea_t(11, NULL, 2, 1) ) ;
REM Ordenes del cliente 3**********************************
   INSERT INTO ordenes_tab
      SELECT 3001, REF(C),
         SYSDATE,'30-MAY-1999',
         lineas_pedido_t(),
         NULL
      FROM clientes_tab C
      WHERE C.clinum= 3 ;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 3001
   )
      SELECT 30, REF(S), 18, 30
      FROM items_tab S
      WHERE S.itemnum = 3011;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 3001
   )
      SELECT 32, REF(S), 10, 100
      FROM items_tab S
      WHERE S.itemnum = 1535;
**************************************
   INSERT INTO ordenes_tab
      SELECT 3002, REF(C),
         SYSDATE,'15-JUN-1999',
         lineas_pedido_t(),
         NULL
      FROM clientes_tab C
      WHERE C.clinum= 3 ;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 3002
   )
      SELECT 34, REF(S), 200, 10
      FROM items tab S
      WHERE S.itemnum = 4534;
REM Ordenes del cliente 4**********************************
   INSERT INTO ordenes_tab
      SELECT 4001, REF(C),
         SYSDATE,'12-MAY-1999',
         lineas_pedido_t(),
         direccion_t('34 Nave Oeste','Nules','CS','12876')
      FROM clientes_tab C
      WHERE C.clinum= 4;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 4001
   )
      SELECT 41, REF(S), 10, 10
      FROM items_tab S
      WHERE S.itemnum = 2004;
   INSERT INTO THE (
      SELECT P.pedido
      FROM ordenes_tab P
      WHERE P.ordnum = 4001
   )
      SELECT 42, REF(S), 32, 22
      FROM items_tab S
      WHERE S.itemnum = 5535;
```

### *4.2.4 Borrado de los objetos, las tablas y los tipos de usuario*

```
DELETE FROM ordenes_tab;
DROP TABLE ordenes_tab;
DELETE FROM clientes_tab;
DROP TABLE clientes_tab;
```

```
DELETE FROM items_tab;
DROP TABLE items_tab;
DROP TYPE ordenes_t;
DROP TYPE lineas_pedido_t;
DROP TYPE linea_t;
DROP TYPE item_t;
DROP TYPE clientes_t;
DROP TYPE lista_tel_t;
DROP TYPE direccion_t;
```

### 4.2.5 Definición de métodos para los tipos

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

```
CREATE TYPE ordenes_t AS OBJECT (
   ordnum NUMBER,
   cliente REF clientes_t,
   fechpedido DATE,
   fechentrega DATE,
   pedido lineas_pedido_t,
   direcentrega direccion_t,
   MEMBER FUNCTION
      coste_total RETURN NUMBER,
      PRAGMA RESTRICT_REFERENCES(coste_total, WNDS, WNPS) );
   CREATE TYPE BODY ordenes_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;
```

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.

### 4.2.6 Consultas a la base de datos anterior

**1- Consultar la definición de la tabla de clientes.**

```
describe clientes_tab;
```

```
NAME                           NULL? TYPE
------------------------------ -------- ----
CLINUM                         NOT NULL NUMBER
CLINOMB                                 VARCHAR2(200)
DIRECCION                               DIRECCION_T
LISTA_TEL                               LISTA_TEL_T
```

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

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

```
1 ROW CREATED.
```

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

```
select clinomb from clientes_tab where clinum=2;
```

```
CLINOMB
--------------------------------------------------------------------
Jorge Luz
UNIVERSITAT JAUME I
158
```

```
update clientes_tab
   set clinomb='Pepe Puig' where clinum=5;
```

```
1 row updated.
```

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

```
select direccion from clientes_tab where clinum=2;
```

```
DIRECCION(CALLE, CIUDAD, PROV, CODPOS)
--------------------------------------------------------------------
DIRECCION_T('Calle Sol', 'Valencia', 'VA', '08820')
```

```
update clientes_tab
   set direccion=direccion_t('Calle Luna','Castello','CS',68734')
   where clinum=2;
```

```
1 row updated.
```

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

```
select * from clientes_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')
```

También se puede consultar así:

```
select value(C) from clientes_tab C where C.clinum=1;
```

```
VALUE(C)(CLINUM, CLINOMB, DIRECCION(CALLE, CIUDAD, PROV, CODPOS),
LISTA_TEL)
--------------------------------------------------------------------
CLIENTES_T(1, 'Lola Caro', DIRECCION_T('Calle Luna', 'Castellon', 'CS',
'64827'), LISTA_TEL_T('415-555-1212'))
```

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

```
1 row updated.
```

**6- Visualizar el nombre del cliente que ha realizado la orden número 1001.**

```
select o.cliente.clinomb from ordenes_tab o where o.ordnum=1001;
```

```
CLIENTE.CLINOMB
--------------------------------------------------------------------
Lola Caro
DISEÑO DE SISTEMAS DE BASES DE DATOS
159
```

**7- Visualizar todos los detalles del cliente que ha realizado la orden número 1001.**

```
select DEREF(o.cliente) from ordenes_tab o where o.ordnum=1001;
```

```
DEREF(O.CLIENTE)(CLINUM, CLINOMB, DIRECCION(CALLE, CIUDAD, PROV, CODPOS),
LISTA_TEL
```

```
--------------------------------------------------------------------
CLIENTES_T(1, 'Lola Caro', DIRECCION_T('Calle Luna', 'Castellon', 'CS',
'64827'), LISTA_TEL_T('415-555-1212', '6348635872'))
```

De la siguiente manera se obtiene la referencia al objeto, la cuál es ininteligible.

```
select o.cliente from ordenes_tab o where o.ordnum=1001;
CLIENTE
--------------------------------------------------------------------
00002EA5F6693E4A73F8E003960286473F83EA5F6693E3E73F8E0039680286473F8
```

**8- Visualizar el número de todos los items que se han pedido en la orden número 3001.**

```
select cursor(select p.item.itemnum from table(o.pedido) p)
   from ordenes_tab o where o.ordnum=3001;
CURSOR(SELECTP.ITEM.
--------------------
CURSOR STATEMENT : 1
CURSOR STATEMENT : 1
ITEM.ITEMNUM
------------
3011
1535
```

**9- Seleccionar el número de orden y el coste total de las ordenes hechas por el cliente número 3.**

```
select o.ordnum, o.coste_total() from ordenes_tab o
   where o.cliente.clinum=3;
ORDNUM O.COSTE_TOTAL()
---------- ----------------
3001 817566.44
3002 1006800
```