



VNIVERSITATIS VALÈNCIA

Departament d'Informàtica

**BASES DE DATOS
OBJETO-RELACIONALES**

Wladimiro Díaz Villanueva

Curso 2001-2002

Índice General

1	Introducción	3
2	Tipos de objetos	3
2.1	Estructura de un tipo de objeto	4
2.2	Componentes de un tipo de objeto	5
2.2.1	Atributos	5
2.2.2	Métodos	6
2.2.3	El parámetro SELF	6
2.2.4	Sobrecarga	7
2.2.5	Métodos MAP y ORDER	8
2.2.6	Constructores	9
2.3	Pragma RESTRIC_REFERENCES	9
3	Declaración e inicialización de objetos	10
3.1	Declaración de objetos	11
3.2	Inicialización de objetos	11
3.3	Objetos sin inicializar en PL/SQL	11
3.4	Acceso a los atributos	12
4	Invocación de constructores y métodos	13
4.1	Paso de parámetros a un constructor	13
4.2	Invocación de métodos	13
5	Compartición de objetos	14
5.1	Utilización de referencias	15
5.2	Limitaciones en la definición de tipos	16
6	Manipulación de objetos	16
6.1	Selección de objetos	17
6.2	El operador VALUE	17
6.3	El operador REF	18
6.3.1	Referencias colgadas (<i>dangling refs</i>)	19
6.4	El operador Deref	19
6.5	Inserción de objetos	20
6.6	Actualización de objetos	21
6.7	Borrado de objetos	22

Bases de Datos Objeto-Relacionales

1 Introducción

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 si la siguiente generación de la tecnología de bases de datos de uso común sería una base de datos orientada a objetos o una base de datos basada en SQL con extensiones orientadas a objetos. Los partidarios de esta 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 familiares a los usuarios, por lo que estos pueden pasar sus aplicaciones actuales y bases de datos relaciones al nuevo modelo sin tener que reescribirlas y posteriormente ir adaptando sus 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 puede ser un problema importante cuando surgen necesidades que no se han previsto en el diseño original. También resulta problemático realizar la interfaz entre las herramientas SQL estándar y las bases de datos orientadas a objetos puras. En realidad, una de las razones principales por las cuales las empresas adoptaron las bases de datos relacionales tan rápidamente fue su capacidad para crear consultas *ad hoc*.

Para ilustrar la tecnología objeto-relacional utilizaremos como ejemplo el modelo que implementa la base de datos Oracle8. Una de las ventajas que presenta este modelo es que la integración en Oracle8 de las representaciones relacional y orientada a objetos es semánticamente limpia y considerablemente más potente que la adoptada por otros fabricantes. Esto facilita el diseño de bases de datos eficientes y compactas.

2 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; es decir, 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.

2.1 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 1 se representa gráficamente la estructura de un tipo de objeto.

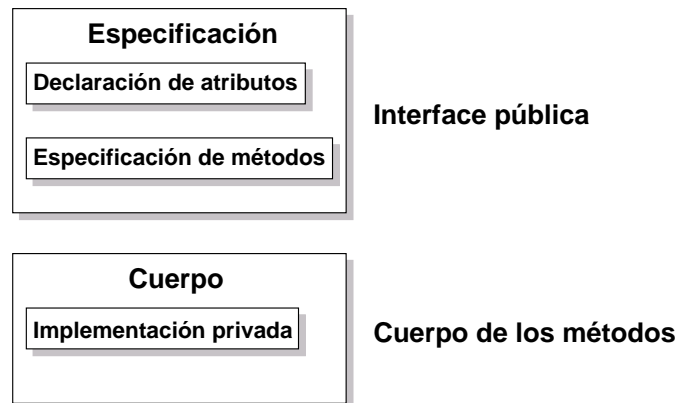


Figura 1: Estructura de un tipo de objeto.

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

En una especificación de tipo de objeto los atributos deben declararse antes que cualquiera de los métodos. Por tanto, si una especificación de tipo sólo declara atributos, el cuerpo es innecesario. Tampoco es posible declarar atributos en el cuerpo del tipo de objeto.

Todas las declaraciones en la especificación del tipo son públicas, es decir, visibles fuera del tipo de objeto. Sin embargo, el cuerpo puede contener declaraciones privadas, que definen métodos necesarios para el funcionamiento interno del tipo de objeto. El ámbito de las declaraciones privadas es local al cuerpo del objeto.

Para entender mejor esta estructura utilizaremos el siguiente ejemplo, en el que se define un tipo de objeto para manipular números complejos. En este ejemplo, 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.rpart, 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;
/

```

2.2 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. Al menos un atributo es requerido y los métodos son opcionales.

2.2.1 Atributos

Como las variables, un atributo se declara mediante un nombre y un tipo. El nombre debe ser único dentro del tipo de objeto (aunque puede reutilizarse en otros objetos) y 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.

Tampoco se puede inicializar un atributo en la declaración empleando el operador de asignación o la cláusula DEFAULT, del mismo modo que no se puede imponer la restricción NOT NULL. Sin embargo, los objetos se pueden almacenar 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. Por ejemplo, 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.

2.2.2 Métodos

En general, 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 en una especificación de tipo debe existir el correspondiente cuerpo del método. El PL/SQL compara la especificación del método y el cuerpo *token a token*, por lo que las cabeceras deben coincidir palabra a palabra.

En un tipo de objeto, los métodos pueden hacer referencia a los atributos y a los otros métodos sin cualificador, como se muestra en el siguiente ejemplo:

```
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;
/
```

2.2.3 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 implícita 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 transform (SELF IN OUT Complex) ...
```

El modo de acceso por omisión de `SELF`, es decir, 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. Como muestra el siguiente ejemplo, 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 el maximo comun divisor 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;
/
```

2.2.4 Sobrecarga

Los métodos del mismo tipo (funciones y procedimientos) se pueden sobrecargar, esto es, 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.

2.2.5 Métodos MAP y ORDER

Los valores de un tipo escalar, como CHAR o REAL, tienen un orden predefinido que permite compararlos. Sin embargo, 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 Rational 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 necesariamente 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.

Supongamos que `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 que indica que `SELF` es respectivamente 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 numero negativo vale.  
        ELSEIF id > c.id THEN  
            RETURN 1;    -- Cualquier numero positivo vale.  
        ELSE  
            RETURN 0;  
        END IF;  
    END match;  
END;
```



```

        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 (que son más fáciles de comparar). Un método ORDER simplemente 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. Sin embargo, si no se declara ninguno, sólo se pueden comparar la igualdad o desigualdad de dos objetos y sólo en SQL. Dos objetos son iguales sólo 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). Un método ORDER es menos eficiente: debe invocarse repetidamente ya que compara sólo dos objetos cada vez.

2.2.6 Constructores

Cada tipo de objeto tiene un constructor que es una función definida por el sistema con el mismo nombre que el objeto. El constructor 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, esto es, 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, por lo que el usuario debe invocarlo explícitamente.

2.3 Pragma RESTRICT_REFERENCES

Para ejecutar una sentencia SQL que invoca a una función miembro, Oracle debe conocer el *nivel de pureza* de la función, es decir, 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) o requerir que un módulo mantenga un cierto estado entre diferentes sesiones de usuario. Por tanto, 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 una de las reglas anteriores. Tampoco puede hacer referencias a una vista que incumpla estas reglas.

Es posible utilizar la directiva de compilación `PRAGMA_REFERENCES` para forzar estas reglas. 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. La sintaxis es la siguiente:

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

Por 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*) y 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 incluido el constructor definido por el sistema. Por 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.

3 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 subprograma 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.

3.1 Declaración de objetos

Los tipos de objetos se declaran del mismo modo que cualquier tipo interno. Por 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. En el siguiente ejemplo, 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 ...
```

y 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 ...
```

3.2 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*. Esto es, el 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`. Del mismo modo, 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 resumen, si asignamos el no-valor `NULL` a un objeto, éste se convierte en atómicamente nulo, como se ilustra en el siguiente 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
```

3.3 Objetos sin inicializar en PL/SQL

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

- Los atributos de un objeto no inicializado se evalúan en cualquier 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); -- Inicializa r
  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.

3.4 Acceso a los atributos

Para acceder o cambiar los valores de un atributo se emplea la notación punto (`'.'`). El siguiente ejemplo ilustra esa notación:

```
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),
    ...
);
/

```

Observe que `zip_code` es un atributo de tipo `Address` y que `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
```

4 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, como se muestra en el siguiente ejemplo:

```

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;
/

```

4.1 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 en el siguiente ejemplo:

```

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

```

4.2 Invocación de métodos

Como el los subprogramas de un paquete, los métodos se invocan usando la notación punto. En el ejemplo siguiente se invoca al 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, como se indica a continuación:

```

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 vacía de parámetros: `'()`'. En sentencias de procedimiento, la lista vacía de parámetros es opcional, excepto cuando se encadenan llamadas, en cuyo caso 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, ya que los procedimientos no se invocan como parte de una expresión. Por ejemplo, la siguiente sentencia es ilegal:

```
r.normaliza().reciproco; -- ilegal
```

Del mismo modo, 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.

5 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
  phone_number VARCHAR2(15),
  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;
```

5.1 Utilización de referencias

Revisando el objeto de tipo Person de la sección 5, observamos que podemos diseñar una comunidad que puede compartir la misma casa (Home). Para ello podemos utilizar el modificador de tipo REF que declara una referencia (almacena un puntero al objeto):

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

Es importante destacar cómo las referencias entre personas y Homes y entre personas entre sí definen relaciones que se dan en el mundo real.

Es posible declarar referencias como variables, parámetros, campos o atributos. Además, se pueden utilizar referencias como parámetros IN y OUT en funciones y procedimientos. Sin embargo, no es posible navegar a través de referencias. En el ejemplo siguiente se muestra un intento ilegal de navegar a través de una referencia a un objeto:

```
DECLARE  
    p_ref      REF Person;  
    phone_no   VARCHAR2(15);  
BEGIN  
    ...  
    phone_no = p_ref.phone_number; -- Illegal!
```

Para llevar a cabo esta operación es necesario utilizar el operador `DEREF`, a través del cual se puede acceder al objeto (ver apartado 6.4).

5.2 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 ejemplo que sigue, la primera sentencia `CREATE TYPE` es ilegal ya que 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,  
    ...  
);  
/
```

En el ejemplo anterior, cambiar el orden de las sentencias `CREATE TYPE` no soluciona el problema, ya que ambos tipos son mutuamente dependientes, esto es, dependen el uno del otro a través de una referencia. Para resolver este problema se utiliza una sentencia `CREATE TYPE` especial denominada *definición previa de tipo*, que permite la creación de tipos de objetos mutuamente dependientes.

Para resolver el problema anterior, basta con incluir la siguiente sentencia:

```
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.

6 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.

En el siguiente *script* de `SQL*Plus`, 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. Sin embargo, el tipo de objeto existe independientemente de cualquier tabla y puede utilizarse para crear objetos mediante otros métodos.

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 referencia al objeto.

6.1 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,
    phone_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 -- Otra tabla de objetos de tipo Person
        SELECT * FROM persons p
        WHERE p.last_name LIKE '%smith';
```

6.2 El operador VALUE

Como es de esperar, 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). Por ejemplo, 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'
```

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. Es posible utilizar estas variables para acceder y modificar el objeto que contienen:

```

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

```

6.3 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. En el siguiente ejemplo, primero se recuperan una o más referencias a objetos de tipo `Person` y después se insertan en la tabla `person_refs`:

```

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

```

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'; -- óslo una fila
  ...
END;

```

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;

```

6.3.1 Referencias colgadas (*dangling refs*)

Si el objeto al cual apunta una referencia es borrado, la referencia queda “colgada”¹ (apuntando a un objeto inexistente). Para comprobar si se produce esta condición se puede utilizar el predicado SQL `IS DANGLING`. Por 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;
```

6.4 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`.

En el ejemplo que sigue se derreferencia una referencia a un objeto `Person` de la tabla `DUAL`². 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 valida
     a un objeto almacenado en una tabla de objetos */
  SELECT Deref(p_ref) INTO p1 FROM DUAL;
  name := p1.last_name;
```

Es posible utilizar el operador `DEREF` en sentencias SQL sucesivas para derreferenciar 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 pr_ref contiene una referencia valida */
  SELECT Deref(pr_ref) INTO pr FROM DUAL;
  SELECT Deref(pr.p_ref) INTO p FROM DUAL;
  name := p.last_name;
  ...
```

¹Traducción directa del término inglés *dangling*

²`DUAL` es una tabla estándar de Oracle que contiene exactamente una fila. Esta tabla es útil en aquellas sentencias que requieren una tabla “fantasma” para garantizar que existe un resultado y en la que los datos son irrelevantes.

```
END;  
/
```

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
```

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';
```

Por último, en el siguiente 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';
```

6.5 Inserción de objetos

Para añadir objetos a una tabla de objetos se utiliza el comando `UPDATE`. Por ejemplo, para insertar un objeto `Person` en la tabla de objetos `persons` utilizamos la siguiente línea:

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

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

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

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;
```

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.

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.

6.6 Actualización de objetos

Para modificar los atributos de un objeto en una tabla de objetos se utiliza la sentencia `UPDATE`, como se ilustra a continuación:

```

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;

```

6.7 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 como se muestra en el ejemplo:

```

BEGIN
  DELETE FROM persons p
    WHERE p.home_address = '108 Palm Dr';
  ...
END;

```