Módulo 2
Bases de datos

UF3: LENGUAJE SQL: DCL Y EXTENSIÓN PROCEDIMENTAL 3

1. Tutorial de instalación SGBD Oracle 4
1.1. Instalación software JDK versión 8 4
1.2. Instalación servidor Oracle 7
1.3. Instalación cliente Oracle: SQL Developer 11

2. Creación bases de datos en Oracle 13

3. Gestión de usuarios 17
3.1. Creación, modificación y eliminación de usuarios 17
3.2. Espacios de tablas (tablespaces) 21
3.3. Perfiles y privilegios. Asignación de privilegios. 23
3.4. Definición de roles. Asignación y desasignación de roles a usuarios. 27
3.5. Normativa legal vigente sobre la protección de datos. 29

4. Programación en bases de datos 29
4.1. Entornos de desarrollo al entorno de las bases de datos. 30
4.2. Sintaxis del lenguaje de programación. 30
4.3. Procedimientos y funciones 39
4.4. Control de errores. 45
4.5. Cursores y transacciones. 47
4.6. Disparadores o *triggers* 59

UF4: BASES DE DATOS OBJETO-RELACIONALES 63

5. Uso de las bases de datos objeto-relacionales 63
5.1. Características 63
5.2. Tipos de datos objeto. 65
5.3. Definición de tipos de objeto. 66
5.4. Herencia. 70
5.5. Identificadores, referencias. 72
5.6. Tipos de datos colección. 73
5.7. Declaración e inicialización de objetos. 75
5.8. Sentencia SELECT. Inserción, modificación y borrado de objetos 77

BIBLIOGRAFÍA 79

UF3: Lenguaje SQL: DCL y extensión procedimental

En las Unidades Formativas anteriores empezamos con el tratamiento de las bases de datos: desde recoger las necesidades del cliente con el modelo entidad-relación, hasta su implementación en un SGBD en el que, mediante comandos SQL, diseñamos y codificamos una BDD.

Como programa informático hemos utilizado MYSQL, que es un software orientado al tratamiento de bases de datos en sitios web. Cuando ya hemos implementado la base de datos, podemos tratar la información almacenada en tablas mediante diferentes tipos de consultas.

En esta unidad formativa cambiaremos el sistema de gestión de base de datos a SQL Oracle. De esta manera podremos conocer los SGBD más utilizados en el ámbito laboral y poder trabajar con los programas más actuales. Además, utilizaremos un lenguaje de programación creado y utilizado expresamente por y para Oracle.

Para que podamos trabajar con este SGBD, en primer lugar, vamos a ver paso a paso el proceso de instalación de su software y su posterior utilización. Para ello crearemos tablas mediante diferentes comandos SQL.

1. Tutorial de instalación SGBD Oracle

1.1. Instalación software JDK

JDK (*Java Development Kit*) es una herramienta que todo programa implementado en Java necesita tener. Debido a que Oracle está codificado en este lenguaje, comenzaremos con su instalación.

1. Accederemos a la página web oficial de Oracle.

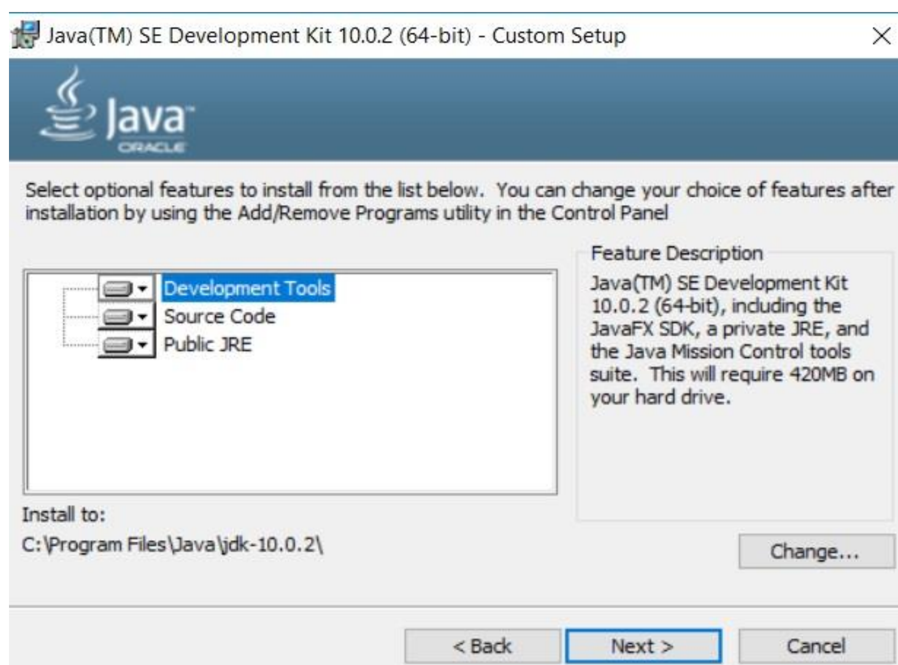
https://www.oracle.com/es/index.html

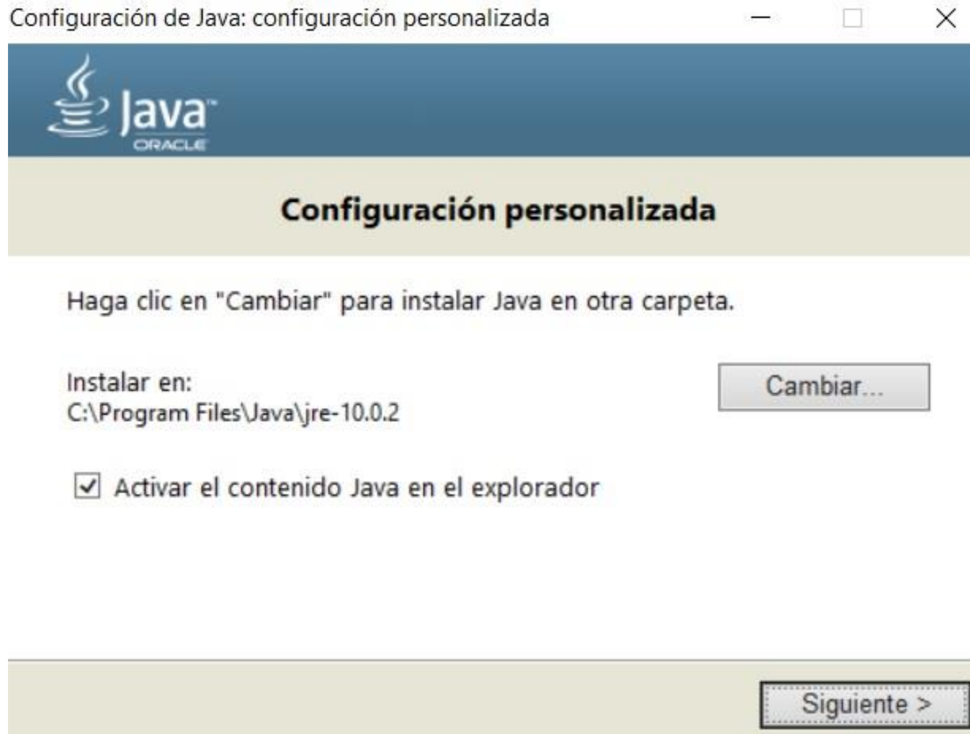
En el buscador introduciremos *JDK* y comenzaremos a descargarlo.

2. Aceptamos las condiciones, seleccionamos el sistema operativo adecuado para nuestro equipo y descargamos.

3. Una vez ejecutado el archivo resultante de la descarga, comenzaremos con el proceso de instalación.

4. Se trata de un proceso bastante sencillo: simplemente tendremos que escoger la ruta de la carpeta de instalación. Después continuaremos hasta que aparezca la opción para cerrar el proceso.




Configuración de Java: configuración personalizada
Java
ORACLE
Configuración personalizada
Haga clic en "Cambiar" para instalar Java en otra carpeta.
Instalar en:
C:\Program Files\Java\jre-10.0.2
Cambiar...
Activar el contenido Java en el explorador
Siguiente >

Java(TM) SE Development Kit 10.0.2 (64-bit) - Complete
Java
ORACLE
Java(TM) SE Development Kit 10.0.2 (64-bit) Successfully Installed
Click Next Steps to access tutorials, API documentation, developer guides, release notes and more to help you get started with the JDK.
Next Steps
Close

1.2. Instalación del servidor Oracle

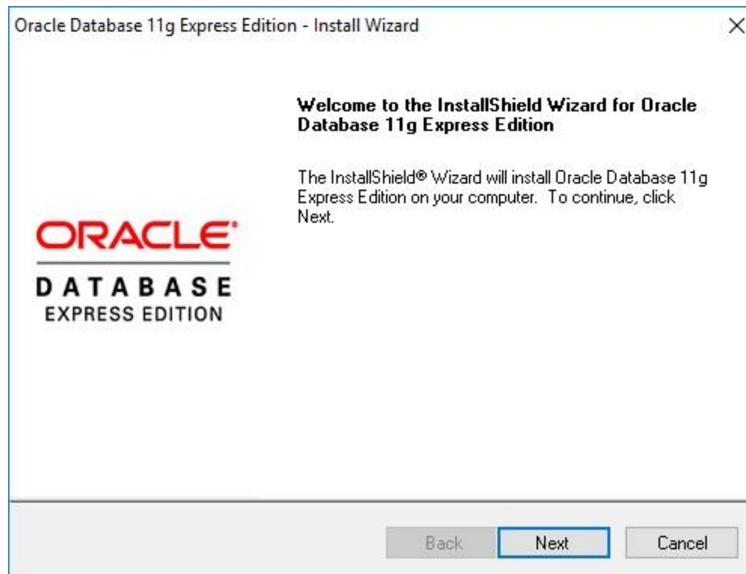
Una vez instalado JDK, pasaremos a la instalación del servidor de base de datos:

1. Empezamos **descargando el archivo de la página oficial de Oracle**, introduciendo en el buscador *Oracle database express*.

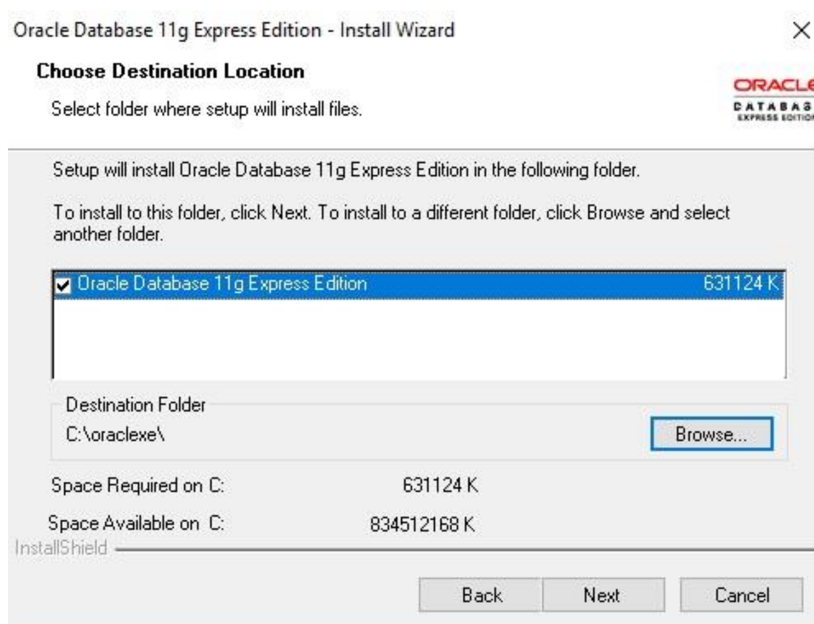
Una vez aceptadas las condiciones, iniciaremos sesión con una cuenta Oracle.

En caso de no tener esta cuenta con anterioridad a la instalación, la crearemos. Es gratuita.

2. El archivo pasará a descargarse una vez que completemos las credenciales de usuario e introduzcamos la contraseña.
3. Una vez descargado, haremos la **extracción del archivo** y lo **ejecutaremos** pulsando ***Setup***.

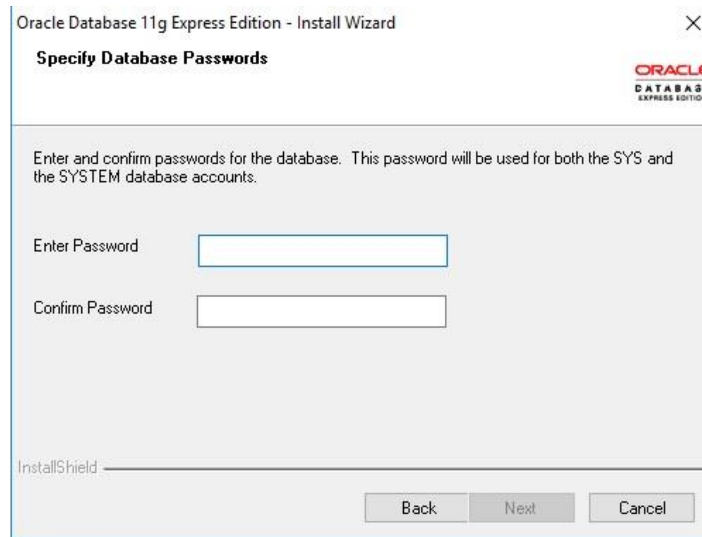


4. Para continuar el proceso, aceptaremos las condiciones y **elegiremos la ruta de instalación**.

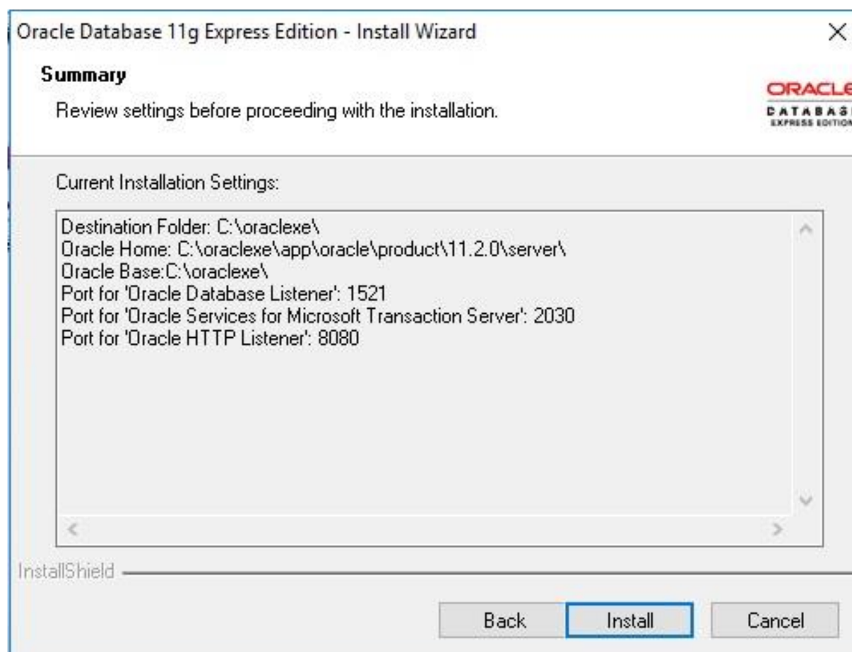


5. Este **SGBD** tiene por defecto dos usuarios (*SYS* y *SYSTEM*) que hacen funciones de administrador, es decir, tienen concedidos todos los permisos para realizar cualquier tratamiento en la base de datos.

   A continuación, elegiremos una contraseña. Nosotros hemos escogido *root*.



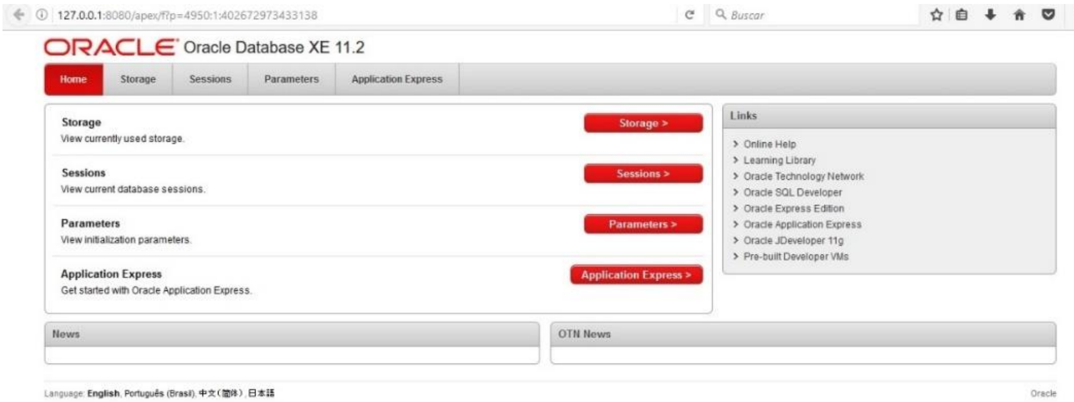
6. El proceso nos listará todo lo que va a instalarse a continuación.



7. Seleccionaremos ***Install*** y, a continuación, ***Finish*** para terminar el proceso. A continuación, reiniciaremos nuestro equipo.

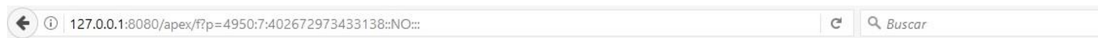
En principio, no necesitamos hacer ningún tipo de configuración en el servidor, pero sí comprobaremos su correcto funcionamiento. Para ello haremos doble clic en el acceso directo que encontraremos en nuestro escritorio: *Get starter with Oracle Database*.

De esta forma accedemos al servidor local configurado por defecto (**127.0.0.1**).



En esta ventana, veremos las distintas opciones disponibles para administrar una base de datos como, por ejemplo, la capacidad disponible en nuestro disco para almacenar diferentes bases de datos, las sesiones que actualmente tenemos disponibles, los parámetros o las aplicaciones instaladas en el servidor para el tratamiento de dichas bases de datos.

En este caso haremos una comprobación del correcto funcionamiento del proceso de instalación. Para ello nos dirigiremos hacia ***Application Express***, y accederemos mediante usuario y contraseña.



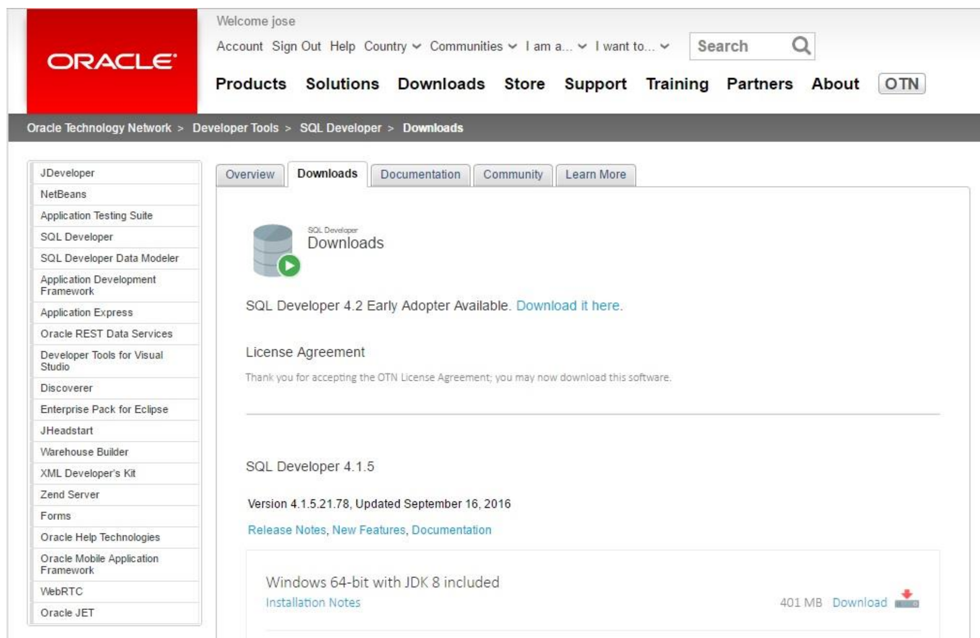
8. De esta forma, comprobaremos que el acceso a nuestro servidor Oracle es correcto.

1.3. Instalación del cliente Oracle: SQL Developer

Una vez realizada la instalación del servidor de la BDD procederemos a instalar un programa cliente para Oracle y, así, facilitar la administración de las diferentes bases de datos.

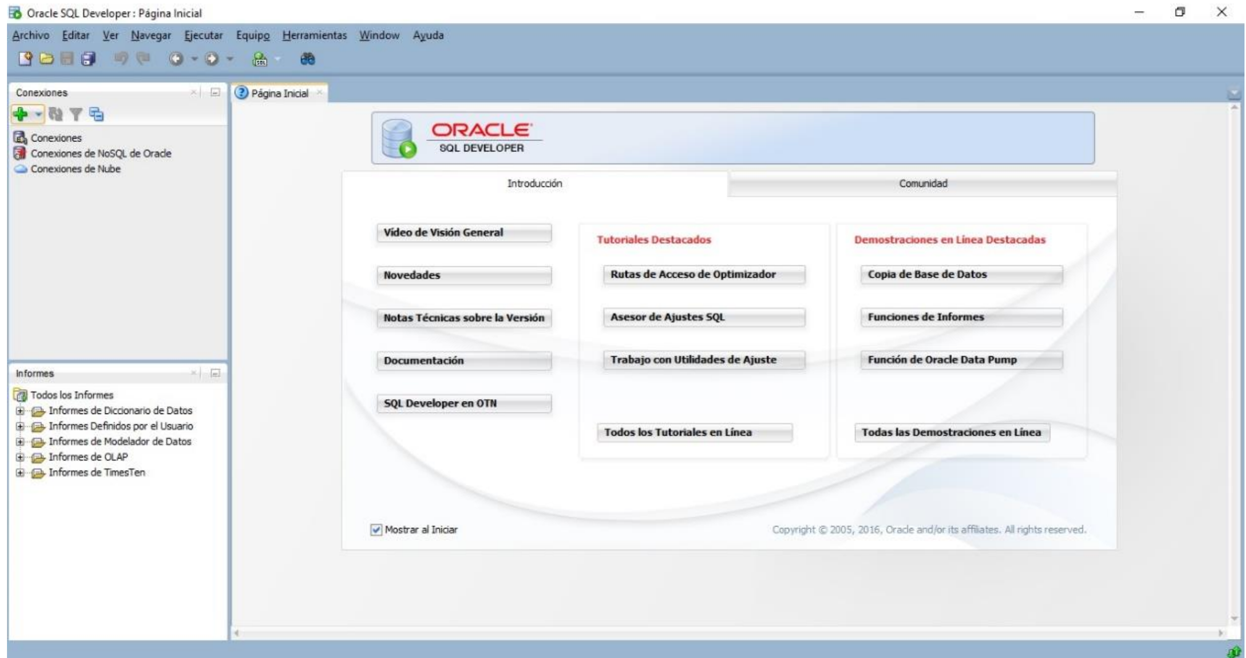
En esta ocasión hemos escogido **SQL Developer** (cliente por defecto de Oracle):

1. Introduciremos en el buscador ***SQL Developer download*** para su descarga.
2. Seleccionaremos la opción adecuada, y una vez que accedemos a la página oficial de Oracle, aceptaremos condiciones y elegiremos la versión que más se adapte a nuestro sistema operativo.



3. Igual que en el proceso anterior, iniciaremos sesión con nuestra cuenta Oracle y el software se descargará de forma automática.

4. El archivo se descargará en un formato comprimido. Una vez lo hayamos extraído haremos doble clic sobre el ejecutable. Este archivo no necesitará instalación.



5. Una vez comprobado que el cliente SQL Developer funciona correctamente, es recomendable crear un acceso directo de esta aplicación en el escritorio.
6. Una vez haya terminado el proceso de instalación es recomendable crear accesos directos en nuestro escritorio.

2. Creación de Bases de Datos en ORACLE

Antes de empezar con la creación de la base de datos vamos a enlazar el software cliente con el servidor.

Lo primero que haremos es comprobar que el servidor se está ejecutando.

De no ser así, basta con hacer doble clic sobre su icono en el escritorio.

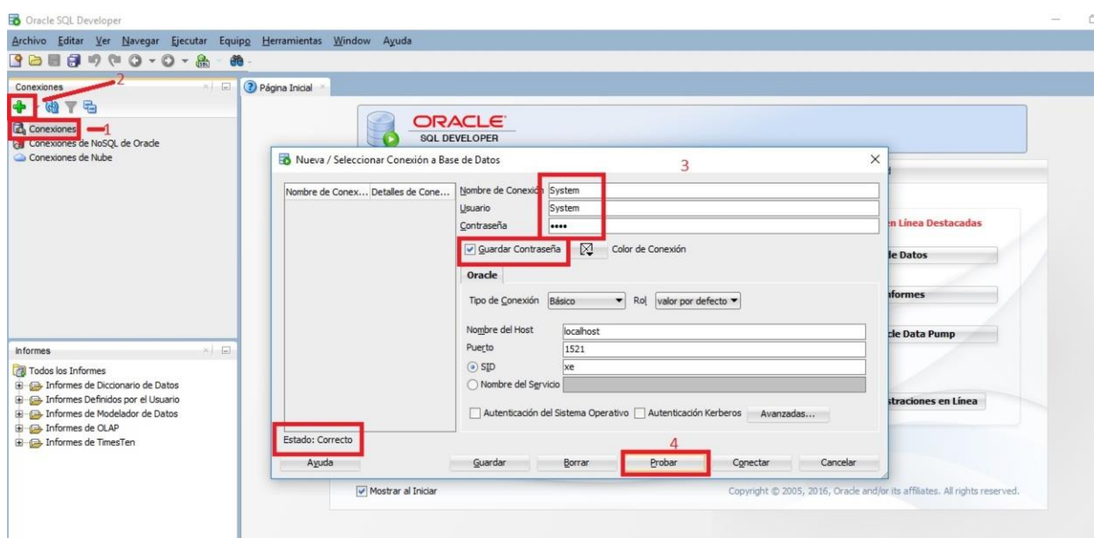
A continuación, ejecutaremos el cliente y, accediendo a conexiones, crearemos una nueva conexión (+).

Para simplificar la prueba, vamos a nombrar a la conexión exactamente igual que al usuario que la va a utilizar, por tanto, el nombre de la conexión va a ser SYSTEM.

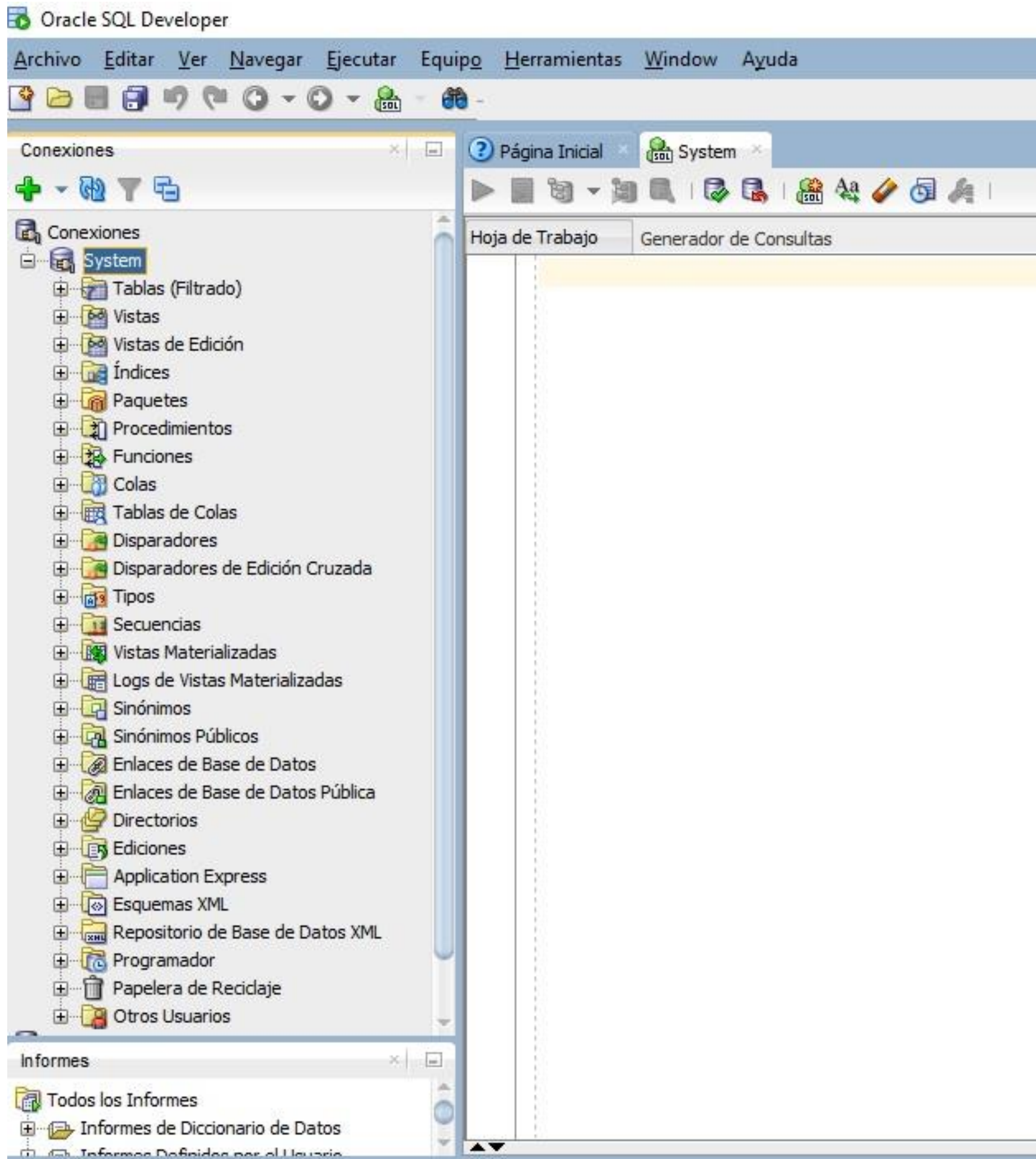
- **Usuario**: SYSTEM
- **Contraseña**: root

Si quisiéramos un usuario diferente tendríamos que crearlo primero en el servidor para que, después, pudiéramos utilizarlo en una conexión.

Podemos seleccionar la opción ***Guardar contraseña*** para que quede memorizada y no nos la vuelva a pedir. Seguidamente, probaremos la conexión comprobando que el estado es correcto. Una vez hecha esta comprobación, podremos guardar y conectar.

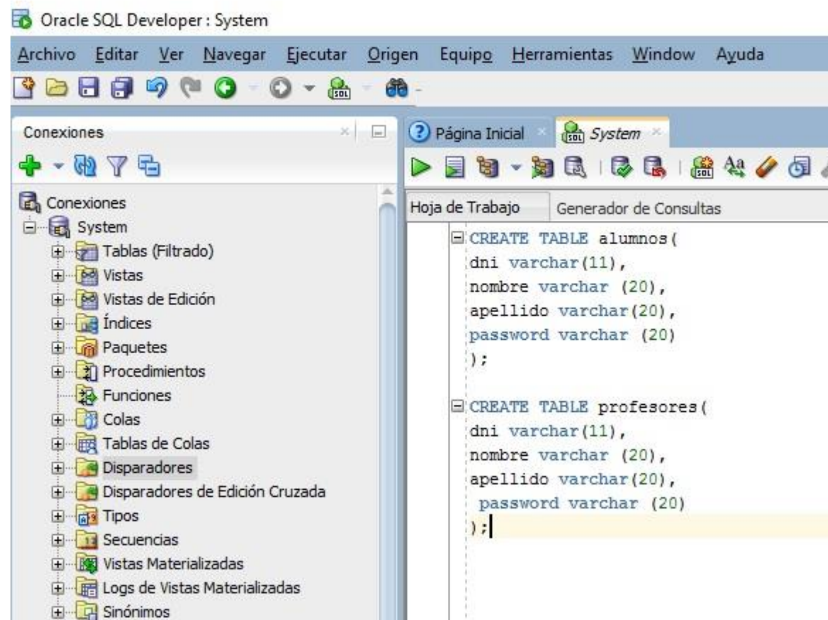


Si hemos seguido los pasos correctamente, tendremos una conexión a la base de datos perfectamente funcional y con todos los privilegios.

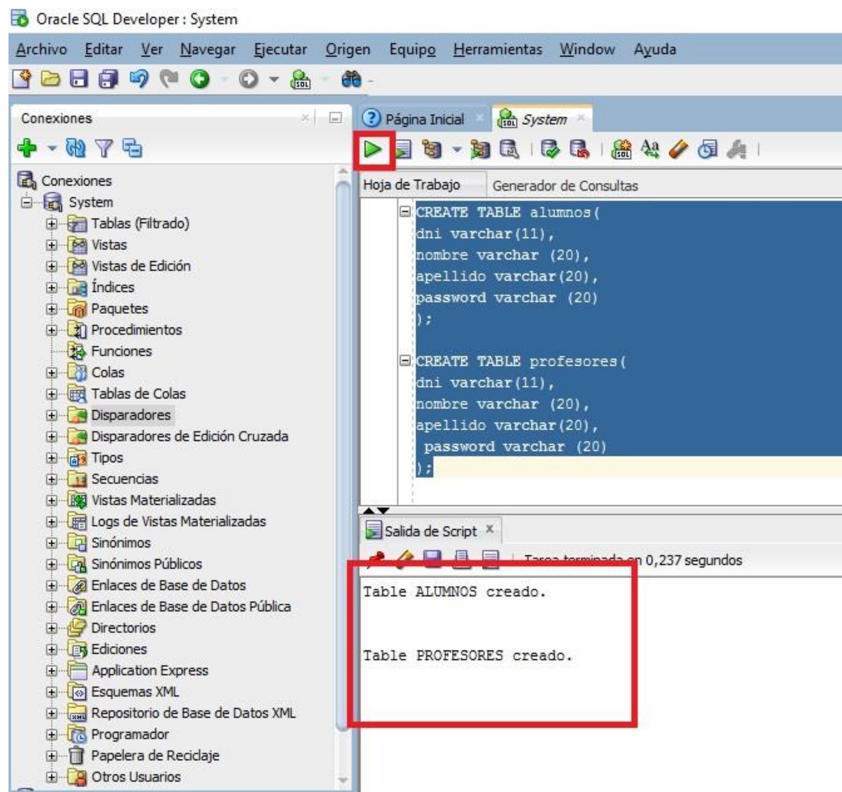


A continuación, veremos los diferentes pasos para crear las tablas dentro de una conexión que actuará como base de datos:

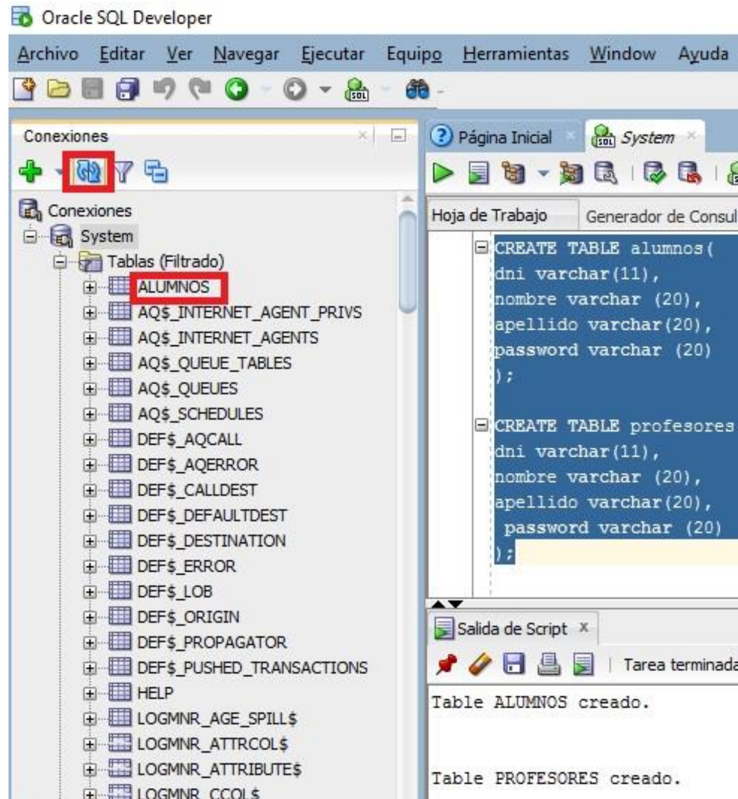
1. **Crearemos las tablas** mediante comandos SQL.

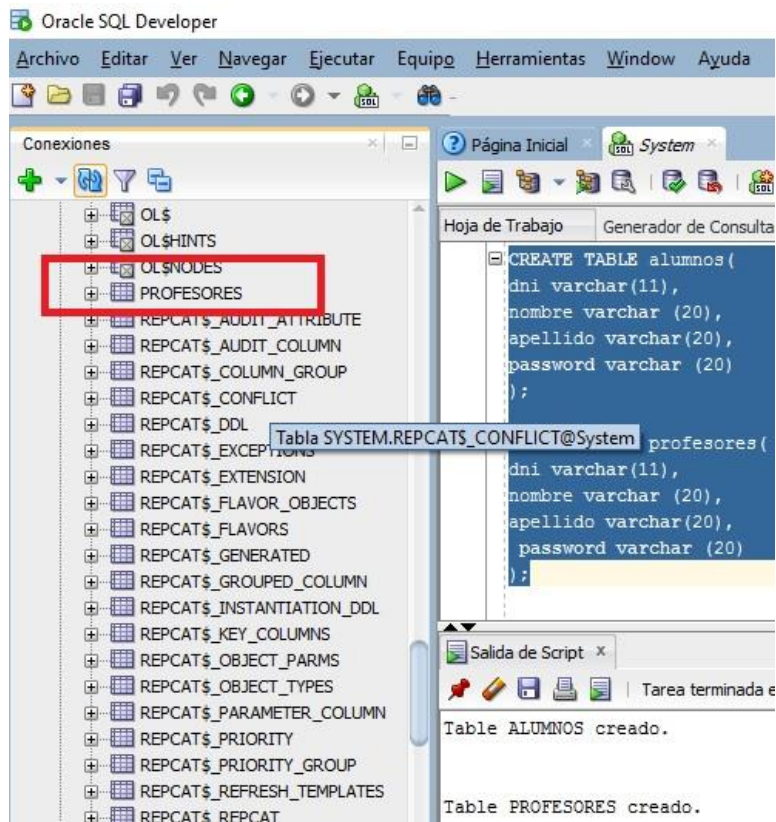


2. A continuación, **seleccionaremos los comandos** y pulsamos sobre el acceso directo ***sentencia de ejecución***.



3. Para comprobar que las órdenes han sido efectuadas con éxito, refrescaremos la conexión y, en el apartado de tablas, deberán aparecer las que hayamos creado.





3. Gestión de usuarios

Las bases de datos tienen una lista válida de usuarios a los que se le permite la conexión al sistema. Por ese motivo se debe crear una cuenta para cada usuario en la que se especifique: nombre de usuario, método de autentificación, *tablespace* permanente o temporal y perfil de usuario.

Vamos a diferenciar **las dos formas por las que se puede autentificar un usuario**:

- **Autentificación por base de datos**: la administración de la cuenta es de tipo usuario/contraseña, de forma que se va a guardar encriptada y su autentificación se realizada por Oracle.
- **Autentificación externa**: la cuenta la mantiene Oracle aunque la administración de la contraseña y la autentificación de usuario es realizada externamente por el sistema operativo. El usuario tiene la posibilidad de conectarse a la base de datos sin necesidad de indicar el nombre del usuario o la clave (es el mismo nombre de usuario del S.O.).

3.1. Creación, modificación y eliminación de usuarios

- **Cuentas administrativas**

Como hemos podido comprobar a la hora de la instalación de ORACLE, existen, por defecto, dos cuentas con permisos administrativos concedidos para realizar tareas de optimización y monitorización de las bases de datos.

- **SYS**: funciona como superadministrador de la base de datos (rol de DBA) y no interesa modificar su esquema porque es donde se crea el diccionario de datos.
- **SYSTEM**: contiene el mismo rol que la anterior y, por defecto, tiene una serie de tablas y vistas administrativas ya creadas.
- **SYSMAN**: realiza tareas administrativas utilizando Enterprise Manager.
- **DBSMNP**: controla la aplicación Enterprise Manager.

- **Privilegios administrativos**

Existen dos privilegios de sistema asociados a la gestión de administrador de una base de datos. En Oracle podemos distinguir entre:

- **SYSDBA** permite:
  - Iniciar o frenar diferentes instancias de una base de datos.
  - Crear, modificar o borrar bases de datos.
  - Recuperar bases de datos y conceder privilegios de sistema.
- **SYSOPER**: realiza las mismas funciones que el anterior excepto crear, borrar y recuperar bases de datos.

La vista ***v$PWFILE_USERS*** nos permite examinar a los usuarios administrativos.

- **Características de los usuarios de Oracle**

Los usuarios, como hemos indicado anteriormente, deben poseer un nombre. Veamos las restricciones que existen a la hora de su creación:

  - **Nombre usuario**: debe ser único e irrepetible. Su longitud máxima no debe sobrepasar los 30 caracteres. Además, solamente puede contener caracteres alfanuméricos y los signos ‘$’ y ‘_’ como caracteres especiales.
  - **Configuración física**: espacio que posee el usuario para almacenar su información y límite de almacenamiento. En Oracle se denomina *tablespace*.
  - **Perfil asociado**: son los diferentes recursos de los que dispone el usuario del sistema.
  - **Privilegios y roles**: concesión de funciones que pueden realizar los usuarios.

- **Estado de una cuenta de usuario**
  - **Abierta**: el usuario puede trabajar sin problema en las acciones habilitadas.
  - **Bloqueada**: el usuario no puede realizar ninguna acción mientras que se encuentre en este estado.
  - **Expirada**: la cuenta de usuario ha agotado el tiempo máximo del que disponía.
  - **Expirada y bloqueada**.
  - Expirada en **periodo de gracia**: está en los últimos momentos de uso antes de pasar a estado de expirada.

- **Creación de usuario**

Antes de comenzar con la creación de usuarios, debemos asegurarnos de que nos conectamos a un usuario que posee permisos concedidos para la creación de usuarios.

Existen dos formas diferentes para conectarnos a un usuario:

1. Mediante la creación de una conexión.
   Como hemos visto en el apartado anterior con el usuario SYSTEM.

2. Mediante comandos en Oracle (Run SQL Command Line).

Run SQL Command Line

```
SQL*Plus: Release 11.2.0.2.0 Production on Jue Abr 13 18:11:32 2017

Copyright (c) 1982, 2014, Oracle.  All rights reserved.

SQL> connect System
Enter password:
Connected.
SQL>
```

La sentencia para crear una cuenta de usuario que permita la autentificación de este en el SGBD con un nivel determinado de privilegios es:

CÓDIGO:

```
-- Sintaxis
CREATE USER nombre_usuario IDENTIFIED BY 'passwords' [opciones];
```

CÓDIGO:

```
-- Ejemplo
CREATE USER ilerna IDENTIFIED BY 'root';
```

Para saber a qué usuario estamos conectados utilizaremos el siguiente comando:

CÓDIGO:

```
-- Ejemplo
SHOW USER
```

Si estamos trabajando con las hojas de trabajo del cliente de Oracle *SQL Developer*, hay que tener en cuenta que, al crear un usuario y conectarnos a él, una vez terminemos el script, este se desconectará automáticamente. En el caso de estar utilizando la consola de comandos, este usuario se quedará conectado hasta que cerremos la conexión o la consola.

Sentencia de creación y conexión:

```
CREATE USER ilerna IDENTIFIED BY 'root';
GRANT CREATE SESSION TO ilerna;
CONN ilerna / root
```

Cuando se crea un usuario también podemos elegir las siguientes opciones:

CÓDIGO:

```
-- Sintaxis
CREATE UER nombre {IDENTIFIED BY 'contraseña'|
EXTERNALLY |  GLOBALLY AS nombreGlobal}
[DEFAULT TABLESPACE tableSpacePorDefecto]
[TEMPORARY TABLESPACE tableSpacetTemporal]
[QUOTA {cantidad [K|M] | UNLIMITED} ON tablespace
[QUOTA {cantidad [K|M] | UNLIMITED} ON tablespace [...]] ]
[PASSWORD EXPIRE]
[ACCOUNT {UNLOCK|LOCK}];
[PROFILE {perfil | DEFAULT}]
```

CÓDIGO:

```
-- Ejemplo
CREATE USER ilerna IDENTIFIED BY 'root'
DEFAULT TABLESPACE 'Alumnos'
QUOTA 15M ON 'Alumnos'   /*Se dan 15MBytes de espacio en el tablespace*/
```

- **Modificación de usuario**

CÓDIGO:

```
-- Sintaxis
ALTER USER nombre_usuario [opciones];
-- Ejemplo
ALTER USER ilerna ACCOUNT UNLOCK;
```

Para quitarle el límite de cuota:

CÓDIGO:

```
-- Ejemplo
ALTER USER ilerna QUOTA UNLIMITED ON Alumnos;
```

- **Borrado de usuario**

CÓDIGO:

```
-- Sintaxis
DROP USER nombre_usuario [CASCADE];
```

La opción CASCADE elimina primero los objetos que están asociados al usuario y después el usuario.

CÓDIGO:

```
-- Ejemplo
DROP USER ilerna;
```

Cada vez que borremos un usuario, borraremos también el esquema asociado a sus objetos.

Una vez que ya hemos visto la creación, modificación y borrado de usuarios, pasaremos a ver cómo realizar consultas sobre usuarios ya creados.

Mediante **DBA_USERS** se muestra la lista y configuración de los usuarios del sistema. Es conveniente utilizar **DESCRIBE DBA_USERS** para visualizar su estructura.

CÓDIGO:

```
-- Ejemplo
DESC DBA_USERS;
```

3.2. Espacios de tablas (*tablespaces*)

Los espacios de tablas son almacenes para estructuras de una base de datos (tablas, vistas, procedimientos, etc.). Una base de datos consta de uno o más espacios de tablas.

Por defecto, Oracle crea los siguientes *tablespaces*:

1. SYSTEM (diccionario de datos)
2. SYSAUX (componentes opcionales de la BBDD)
3. TEMP
4. UNDOTBS1 (*undo tablespace*, para *rollbacks* de transacciones)
5. USERS

- **Operaciones con espacios de tablas**

Vemos la sintaxis de creación de un *tablespace*, la cual contiene todas las opciones que le podemos indicar:

CÓDIGO:

```
-- Sintaxis
CREATE [TEMPORARY / UNDO] TABLESPACE <tablespace_nombre>
DATAFILE / TEMPFILE '<datafile01_name y el Path donde creamos el fichero>' SIZE
<INTEGER N>[,
'<datafile02_name y el Path donde creamos el fichero>' SIZE <INTEGER N>[,
'<datafile0N_name y el Path donde creamos el fichero>' SIZE <INTEGER N>[, ...]]]
BLOCKSIZE <DB_BLOCK_SIZE parameter /2k/4k/8k/16k/32k>
AUTOEXTEND { [OFF/ON (NEXT <INTEGER K/M> MAXSIZE<INTEGER K/M>) / UNLIMITED] }
LOGGING/NOLOGGING (Logging default)
ONLILE/OFFLINE (Online default)
EXTENT MANAGEMENT {[DICTIONARY]
[LOCAL default (AUTOALLOCATE / UNIFORM<INTEGER K/M>)]}
PERMANENT / TEMPORARY (Permanent default)
MINIMUM EXTENT
DEFAULT STORAGE { [INITIAL <INTEGER K/M>]
[NEXT <INTEGER K/M>]
[PCTINCREASE <INTEGER K/M>]
[MINEXTENTS <INTEGER>]
[MAXEXTENTS <INTEGER> / UNLIMITED]
[FREELISTS <INTEGER>]
[FREELISTS GROUPS <INTEGER>]
[OPTIMAL <INTEGER> / NULL]
[BUFFER_POOL <DEFAULT/KEEP/RECYCLE>]}
CHUNK <INTEGER K/M>
NOCACHE;
```

Para crear un *tablespace* asociada a un usuario lo primero que haremos es crear el espacio de tablas relacionado con un fichero físico de nuestro equipo:

CÓDIGO:

```
-- Ejemplo
CREATE TABLESPACE ejercicios DATAFILE
```

Y, a continuación, creamos el usuario relacionado con la tabla de espacios anterior:

CÓDIGO:

```
-- Ejemplo
CREATE USER ilerna IDENTIFIED BY 'root'
DEFAULT TABLESPACE ejercicios;
```

3.3. Perfiles y Privilegios. Asignación de privilegios

- **Perfiles**

Los **perfiles** definen un conjunto de **límites de recursos** de la base de datos y del sistema operativo para los distintos usuarios. Existe un perfil por defecto llamado *DEFAULT* que tiene todos los recursos asignados con valor *UNLIMITED*. Este perfil se asigna por defecto a no ser que se especifique uno distinto.

Cada usuario tiene únicamente un perfil en un instante de tiempo determinado.

Necesita el privilegio del sistema *CREATE PROFILE* cuya sintaxis es:

CÓDIGO:

```
-- Sintaxis
CREATE PROFILE <nombrePerfil> LIMIT
<parámetro 1><valor>|UNLIMITED|DEFAULT
<parámetro N><valor>|UNLIMITED|DEFAULT;
```

Se asigna el valor ***UNLIMITED*** si es un parámetro de recurso que se puede usar una cantidad de veces ilimitada, esto suele darse en el caso de parámetros que tienen asignada una contraseña que no ha fijado límites.

Mientras que el valor ***DEFAULT*** se va a utilizar cuando queremos omitir algún parámetro asignado a un usuario por defecto.

- **Privilegios**

Un usuario de una base de datos posee una serie de privilegios que le permiten manipular los objetos de esta, es decir, representa los derechos que se le han asignado a los usuarios de una determinada base de datos como:

- Ejecutar un tipo de sentencia SQL.
- Acceder a un objeto de otro usuario.
- Ejecutar distintos procedimientos.

Los privilegios son asignados a usuarios de manera explícita o, preferiblemente, a roles, intentando no excederse en la concesión de permisos.

Podemos diferenciar entre **dos tipos de privilegios**:

- **Privilegios de sistema**: ofrecen la posibilidad de realizar determinadas acciones de administración en la base de datos en cualquier esquema. Afecta a todos los usuarios. Por ejemplo, ***CREATE USER*** o ***CREATE TABLE***.
- **Privilegios de objetos**: ofrecen la posibilidad a un usuario de acceder, manipular o ejecutar objetos concretos (tablas, vistas, secuencias, procedimientos, funciones o paquetes). Por ejemplo, borrar o consultar filas de una tabla concreta.

El comando que asigna estos privilegios es ***GRANT***.

De la misma forma que se le pueden asignar privilegios, usando el comando ***REVOKE*** se le pueden denegar.

Para asignar privilegios, la **sintaxis** del comando *GRANT* es:

CÓDIGO:

```
-- Sintaxis
GRANT tipo_privilegio [ ( columna ) ]  [, tipo_privilegio [ ( columna ) ] ...
ON  { nombre_tabla |  *  |  *.*  | base_datos.nombre_tabla }
TO usuario [ IDENTIFIED BY [ PASSWORD ] 'password' ] ] ...
[ WITH option [, opción ] ...
```

CÓDIGO:

```
GRANT OPTION
| MAX_QUERIES_PER_HOUR count
| MAX_UPDATES_PER_HOUR count
| MAX_CONNECTIONS_PER_HOUR count
| MAX_USER_CONNECTIONS count
```

En este caso:

- ***tipo_privilegio***: es el tipo de permiso que se le asigna al usuario (*select, insert, update*, etc.).
- Los **objetos sobre los que puede operar el usuario y los privilegios** que se le pueden aplicar se muestran en las siguientes **expresiones**:
  - ***nombre_tabla*** → sobre la tabla *nombre_tabla*.
  - ******* → sobre todas las tablas de la base de datos que se está utilizando.
  - ***.**** → sobre todas las tablas de todas las bases de datos.
  - ***Base_datos.**** → sobre todas las tablas de la base de datos *base_datos*.
  - ***Base_datos.nombre_tabla*** → sobre la tabla *nombre_tabla* que pertenece a la base de datos *base_datos*.
- ***TO***: indica el usuario al que se quiere otorgar el permiso. Si este no existe, se crea con el *password* indicado mediante la cláusula ***IDENTIFIED BY***.
- **WITH** permite indicar ciertas opciones:

Expresión	Función
GRANT OPTION	Concede a otros usuarios los permisos que tiene el primer usuario.
MAX_QUERIES_PER_HOUR count	Restringe el número de consultas por hora que puede realizar un usuario.
MAX_UPDATES_PER_HOUR count	Restringe el número de modificaciones por hora que puede realizar un usuario.
MAX_CONNECTIONS_PER_HOUR count	Restringe las conexiones por hora que realice un usuario.
MAX_USER_CONNECTIOS count	Limita el número de conexiones simultáneas que puede tener un usuario.

- **Tipos de privilegios:**

Privilegio	Significado
ALL	Da todos los permisos simples excepto *GRANT OPTION*.
ALTER	Permite el uso de *ALTER TABLE*.
ALTER ROUTINE	Modifica o borra rutinas almacenadas.
CREATE	Permite el uso de *CREATE TABLE*.
CREATE ROUTINE	Crea rutinas almacenadas.
CREATE TEMPORARY TABLES	Permite el uso de *CREATE TEMPORARY TABLE*.
CREATE USER	Permite el uso de *CREATE USER*, *DROP USER*, *RENAME USER* y *REVOKE*.
CREATE VIEW	Permite el uso de *CREATE VIEW*.
DELETE	Permite el uso de *DELETE*.
DROP	Permite el uso de *DROP TABLE*.
EXECUTE	Permite al usuario ejecutar retinas almacenadas
FILE	Permite el uso de *SELECT*, *INTO OUTFILE* Y *LOAD DATA INFILE*.
INDEX	Permite el uso de *CREATE INDEX* y *DROP INDEX*.
INSERT	Permite el uso de *INSERT*.
LOCK TABLES	Permite el uso de *LOCK TABLES* en tablas par alas que tenga el permiso *SELECT*.
PROCESS	Permite el uso de *SHOW FULL PROCESSLIST*.
RELOAD	Permite el uso de *FLUSH*.
REPLICATION CLIENT	Permite al usuario preguntar dónde están los servidores maestros o esclavos.
REPLICATION SLAVE	Necesario para los esclavos de replicación.
SELECT	Permite el uso de *SELECT*.
SHOW DATABASES	Muestra todas las bases de datos.
SHOW VIEW	Permite el uso de *SHOW CREATE VIEW*.
SHUTDOWN	Permite el uso *demysqladmin shutdown*.
UPDATE	Permite el uso de *UPDATE*.
USAGE	Sinónimo de *no privileges* permite únicamente la conexión al gestor.
GRANTOPTION	Posibilita dar permisos.

Veamos algunos **ejemplos de concesión de permisos**:

- Si deseamos que el usuario ***ilerna@localhost*** solo pueda seleccionar las columnas *nombreCliente, apellido y dirección* procederemos a ejecutar la siguiente sentencia:

CÓDIGO:

```
-- Ejemplo
GRANT SELECT (Nombre, Apellido, Dirección) ON alumnos TO ilerna@localhost;
'/home/oracle/oradata/orcl/ejercicios.dbf ' SIZE 1000M autoextend on;
```

- Si deseamos otorgar permisos de *select* a todas las tablas de todas las bases de datos, permitiendo al usuario ceder esos permisos a otros usuarios.

CÓDIGO:

```
-- Ejemplo
GRANT SELECT ON *.* TO ilerna@localhost WITH GRANT OPTION;
'/home/oracle/oradata/orcl/ejercicios.dbf ' SIZE 1000M autoextend on;
```

Con esta instrucción damos el derecho indicado sobre la tabla que señalemos al usuario citado. La última cláusula, que es opcional, permite al usuario que se le conceda el permiso y la posibilidad de ceder el permiso a otra persona.

El DBA tendrá todos los permisos con la cláusula ***ALL***, tanto para concederlos como para revocarlos.

CÓDIGO:

```
-- Ejemplo
REVOKE ALL PRIVILEGES ON *.* FROM ilerna@localhost
```

3.4. Definición de roles. Asignación y desasignación de roles a usuarios

Un **rol** es un **grupo de privilegios** a los que se les asigna un nombre. El empleo de roles facilita la administración, de esta forma no hace falta especificar uno a uno los privilegios que se conceden a cada nuevo usuario. Basta con asignarle un rol para que herede todos los privilegios de este.

Entre sus principales características, podemos destacar:

- Se puede otorgar a cualquier usuario o rol, pero no a sí mismo ni de forma circular.
- Puede tener contraseña.
- Posee un nombre único en la base de datos.
- No forma parte de ningún esquema.

Trabajar con roles ofrece una serie de **beneficios** que, a continuación, vamos a señalar:

- Simplifican el manejo de privilegios. Se pueden asignar diferentes permisos a un rol y este también puede ser asignado a distintos usuarios.
- Maneja los privilegios de forma dinámica, es decir, si se modifican los privilegios asociados al rol se actualizan dichos privilegios en todos los usuarios que lo posean de manera inmediata.
- Disponibilidad selectiva de privilegios. Los roles asignados a un usuario pueden ser activados o desactivados temporalmente y se pueden proteger con clave.
- El uso de roles disminuye el número de *GRANT* almacenados en el diccionario de datos, por lo que mejora la productividad.

Oracle proporciona roles predefinidos para ayudar a la administración de las bases de datos, como:

- ***CONNECT***: incluye únicamente el privilegio *CREATE SESSION* (que permite conectar a la base de datos).

  Si el usuario se crea con *OEM* este rol se asigna automáticamente.

- ***RESOURCE***: incluye *CREATE CLUSTER*, *CREATE INDEXTYPE*, *CREATE OPERATOR*, *CREATE PROCEDURE*, *CREATE SEQUENCE*, *CREATE TABLE*, *CREATE TIGGER* y *CREATE TYPE*.

  Además, otorga el privilegio *UNLIMITED TABLESPACE*.

- ***DBA***: incluye todos los privilegios del sistema mediante la opción *WITH ADMIN OPTION*.
  - No incluye arrancar o parar la base de datos.
  - *SYS* y *SYSTEM* lo poseen.

Las definiciones de los roles están almacenadas en el diccionario de datos.

Normalmente se crea un rol y a posteriori se le asignan privilegios. Así, el grupo de usuarios pertenecientes a este rol adoptan sus privilegios.

Veamos un ejemplo en el que primero nos creamos un rol y, a continuación, le asignamos permisos para que estos los adquieran un grupo de usuarios.

CÓDIGO:

```
-- Ejemplo
CREATE ROLE rol_ilerna;
GRANT CREATE ANY TABLE,
DROP TABLE,
INSERT ON Alumnos TO rol_ilerna;
GRANT rol_ilerna TO ilerna;
```

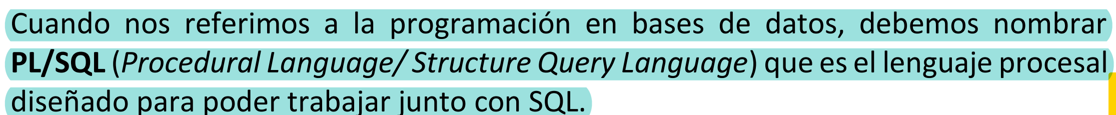
3.5. Normativa legal vigente sobre la protección de datos

La **Ley Orgánica 15/1999 del 13 de diciembre de Protección de Datos de Carácter Personal, (LOPD)** tiene como objetivo: garantizar y proteger, en todo lo relativo a los datos personales, las libertades públicas y los derechos fundamentales de las personas físicas, y especialmente de su honor, intimidad y privacidad personal y familiar.

Regula el tratamiento que se realiza de los datos de carácter personal, excluyendo los datos recogidos para uso doméstico, las materias clasificadas del estado y aquellos ficheros que recogen datos sobre terrorismo clasificados y otras formas de delincuencia organizada.

La organización encargada del cumplimiento de dicha orden es la **Agencia Española de Protección de Datos**, de ámbito estatal.

En el siguiente enlace podemos acceder al BOE para descargarnos dicha ley y poder estudiarla más detenidamente: Ley orgánica de protección de datos



Cuando nos referimos a la programación en bases de datos, debemos nombrar **PL/SQL** (*Procedural Language/ Structure Query Language*) que es el lenguaje procesal diseñado para poder trabajar junto con SQL.

Está incluido en **Oracle Database Server** y, a continuación, vamos a detallar sus características principales:

- Integrado con SQL.
- Control de errores y excepciones.
- Uso de variables.
- Estructuras de control de flujo.
- Soporta programación orientada a objetos.
- Programación modular: procedimientos y funciones.

4.Programación de bases de datos

4.1. Entornos de desarrollo al entorno de las Bases de Datos

Un **intérprete de comandos** es una aplicación cliente cuya única misión es enviar comandos al SGBD y mostrar los resultados devueltos por el SGBD en pantalla.

Oracle. SQL *Plus

Podemos invocarlo desde el sistema operativo ejecutando el siguiente comando:

CÓDIGO:

```
-- Sintaxis

Sqlplus [{usuario[/password][@database] | / |/nolog }] [AS {SYSDBA | SYSOPER}]
```

4.2. Sintaxis del lenguaje de programación

Tipos de datos, identificadores, variables, consultas:

- **Bloques** de código anónimo

Los bloques son la **forma más básica de programar en PL/SQL**. Son fragmentos de código que no se almacenan en la estructura de la base de datos y para ejecutarlo solo es necesario introducirlos en la consola como si se tratase de SQL. Un bloque lógico está estructurado en tres partes:

1. **Parte de declaraciones** (opcional): declararemos todas las variables, constantes, cursores y excepciones definidas por el usuario.
2. **Parte ejecutable** (requerida): podremos hacer uso de las sentencias de control que ofrecen los lenguajes de programación:
   a. Secuencias, órdenes del lenguaje, asignaciones, llamadas a funciones o procedimientos, etc. Se colocará una detrás de otra y las separaremos con ;.
   b. La iteración o bucle repetirá una sentencia o secuencia de sentencias mientras se cumpla, o hasta que deje de cumplirse una condición.

3. **Parte de tratamiento de excepciones** (opcional): bloque opcional de código donde podremos tratar los posibles errores que genere la parte ejecutable del código.

CÓDIGO:

```
-- Sintaxis
[ DECLARE
-- declaraciones de objetos (variables, cursores, excepciones definidas
-- por el usuario)]
BEGIN /*comienza la parte ejecutable */
-- comandos y sentencias SQL y PL/SQL
[EXCEPTION /*comienza la parte de tratamiento de excepciones */
   -- acciones para realizar cuando se producen errores      ]
END;
[ / ]  -- fin de sentencia en consola de comandos
```

El símbolo /, en el caso de estar ejecutando el código por la consola de SQL, nos cierra el bloque de código y este se ejecuta; en caso de no ponerlo, la consola se queda esperando más líneas de código.

Este símbolo también lo utilizaremos en el caso de lanzar dos bloques distintos de código en el SQL Developer, como por ejemplo al lanzar dos *creates* juntos, tendríamos que separar los bloques ejecutables con el símbolo /.

Un **bloque PL/SQL** puede ser un **bloque anónimo**, un **procedimiento** o una **función**.

El objetivo de diseño PL/SQL es conseguir modularidad, es decir, simplificar la complejidad del problema dividiéndolo en problemas más sencillos y fáciles de implementar. Para ello se utilizan procedimientos y funciones:

- **Variables**

Uso

Las variables se declaran y se pueden inicializar en la sección declarativa de un bloque PL/SQL.

Permite pasar valores a un subprograma PL/SQL mediante parámetros. Existen tres formas:

- ***IN***: el valor viene del proceso de llamada, es un dato de entrada y su valor no se cambia. Es el valor por defecto.
- ***OUT***: en una salida de programa sin error, el valor del argumento devuelve al proceso de llamada.
- ***IN OUT***: es una variable de entrada/salida.

○ **Declaración**

La sintaxis para la declaración de variables es:

CÓDIGO:

```
-- Sintaxis
nombre_variable [CONSTANT] tipo_dato [ NOT NULL ] [ := | DEFAULT | expresión ] ;
```

CONSTANT declara la variable como constante. Su valor no puede cambiar. Debe ser inicializada.

○ **Tipos**

Los tipos de datos de las variables PL/SQL son:

- **Compuesto**: los tipos compuestos son las tablas, registros, tablas anidadas y *arrays*.
- ***LOB***: los tipos de datos *LOB* permiten almacenar bloques de datos no estructurados como gráficos, imágenes, vídeos y texto de hasta cuatro gigabytes de tamaño.

- **Escalar:**

Tipo	
BINARY-INTEGER	Almacena enteros con signo. Subtipos: *NATURAL*, *POSITIVO*
NUMBER[(precisión, escala)]	Almacena números con punto fijo o flotante
CHAR[(longitud, máxima)]	Almacena cadenas de caracteres de longitud fija
VARCHAR2(longitud, máxima)	Almacena cadenas de caracteres de longitud variable
LONG	Almacena cadenas de caracteres de longitud variable, tamaño máximo 2 Gb
RAW	Almacena objetos binarios
LONG RAW	Almacena objetos binarios de hasta 2 Gb
BOOLEAN	Almacena *TRUE*, *FALSE* o *NULL*
DATE	Almacena valores de fecha
ROWID	Dirección física de una fila de la BDD

Existen otras variables que no son PL/SQL y podemos clasificar en:

- Variables ***BIND***.
- Variables ***HOST***.
-

- **Operadores**

<table>
<tr><th>Operador</th><th>Acción</th></tr>
<tr><td>**</td><td>Potencia</td></tr>
<tr><td>+ - (unarios)</td><td>Signo positivo o negativo</td></tr>
<tr><td>*
/</td><td>Multiplicación
División</td></tr>
<tr><td>+
-
||</td><td>Suma
Resta
Concatenación</td></tr>
<tr><td>=, <, >, <=
>=, <>, ¡=
IS NULL, LIKE
BETWEEN, In</td><td>Comparaciones: igual, menor, mayor, menor o igual mayor o igual, distinto, distinto
Es null, como
Entre, en</td></tr>
<tr><td>NOT
AND
OR</td><td>Negación lógica de un tipo boolean
Operador AND lógico entre tipos de dato boolean
Operador OR lógico entre tipos de dato boolean</td></tr>
</table>

- **Estructuras de control de flujo**

En la programación estructurada se debe hacer uso de distintos tipos de **estructuras de control** que vamos a describir a continuación.

- **Selección**

Este tipo de estructuras ejecutan un conjunto de instrucciones dependiendo de si se cumple o no una determinada condición.

- Sentencia **IF**

Evalúa una expresión y en función de si el valor de esta es verdadero o falso se ejecutan unas instrucciones u otras.

CÓDIGO:

```
-- Sintaxis
IF condición THEN
        instrucciones;
[ELSIF condición THEN
        instrucciones; ]
[ELSE
        instrucciones; ]
END IF;
```

- Sentencia **CASE**

Evalúa un conjunto de condiciones hasta encontrar alguna que se cumpla y ejecuta las instrucciones que esta posee asociada.

CÓDIGO:

```
-- Sintaxis
CASE [expresión]
WHEN { condición 1 | valor1 } THEN
   bloque_instrucciones_1
WHEN { condición 2 | valor2 } THEN
   bloque_instrucciones_2
ELSE
   bloque_instrucciones_por_defecto
END CASE;
```

- **Iteración**

Consiste en la repetición de un conjunto de instrucciones un número determinado de veces. A continuación, se describen distintos tipos de estructuras de iteración.

- Estructura repetitiva básica:

CÓDIGO:

```
-- Sintaxis
LOOP
   sentencias;
END LOOP;
```

En estos ejemplos de bucles vamos a utilizar la sentencia ***dbms_output.put_line*** para ver los valores de la variable *cont* por pantalla, de tal forma que nos ayuda a entender mejor las condiciones de salida de los bucles. Para que esta sentencia funcione correctamente tenemos que lanzar la sentencia **SET SERVEROUTPUT ON,** que activa el visionado de los mensajes por consola.

Estas sentencias vienen explicadas en el punto 4.5 de cursores y transacciones en el apartado de *Herramientas de depuración y control de código*.

En esta unidad formativa también encontraremos algunas funciones de Oracle, las cuales harán referencia a la fecha, al identificador de la fila, etc.

Ej.: Sysdate: fecha diaria.

Rowid: identificador de la fila.

User: usuario del sistema de base de datos.

NLS_Session_Parameters: parámetros de sesión.

DUAL: tabla del sistema donde encontramos la fecha y otros parámetros.

Este bucle estará repitiéndose infinitamente ya que no existe ninguna condición que permita salir de él. Es conveniente evaluar si dentro del bucle existe alguna condición que provoque la salida del mismo. Podemos realizarlo de dos formas:

1. Usando ***WHEN***: saliéndonos del bucle cuando se cumpla cierta esa condición, mediante la cláusula **EXIT**

CÓDIGO:

```
-- Ejemplo
SET SERVEROUTPUT ON;
DECLARE
CONT INT := 0;
BEGIN
LOOP
   CONT := CONT + 1;
   EXIT WHEN CONT >= 10;
END LOOP;
END;
```

2. Usando ***IF***: forzando a salir de la iteración mediante *EXIT*.

CÓDIGO:

```
-- Ejemplo
SET SERVEROUTPUT ON;
DECLARE
CONT INT := 0;
BEGIN
LOOP
   CONT := CONT + 1;
   IF CONT >= 10 THEN
      EXIT;
   END IF;
END LOOP;
```

- Esquema ***WHILE***

CÓDIGO:

```
-- Sintaxis
WHILE condición LOOP
   instrucciones;
END LOOP;
```

- Esquema ***FOR***: Este bucle sólo lo podemos usar cuando conocemos de antemano el número de repeticiones o iteraciones que queremos realizar.

CÓDIGO:

```
-- Sintaxis
FOR índice IN [ REVERSE ] valor_inicial .. valor_final
LOOP
   instrucciones;
END LOOP;
```

En este caso:

- *Índice* no hace falta declararlo ya que lo hace implícitamente al usar la estructura ***FOR***, pero no se puede usar fuera de él.
- ***REVERSE*** recorre el bucle en sentido inverso, es decir, va decrementando el valor de índice.

Vamos a ver algunos ejemplos muy sencillos de cómo utilizar las sentencias de iteración que acabamos de ver:

CÓDIGO:

```
-- Ejemplo de iteración WHILE
SET SERVEROUTPUT ON;
DECLARE
   cont INTEGER := 0;
   fin BOOLEAN := TRUE;
BEGIN
   WHILE fin LOOP
      cont := cont + 1;
      dbms_output.put_line(cont);
      IF cont >= 5 THEN
         fin := FALSE;
         cont := 0;
      END IF;
   END LOOP;
END;
```

CÓDIGO:

```
-- Ejemplo de iteración FOR
SET SERVEROUTPUT ON;
DECLARE
   cont INTEGER := 0;
   calc INTEGER := 0;
BEGIN
   FOR cont IN 0 .. 5
   LOOP
      calc := cont * 5;
      dbms_output.put_line(calc);
   END LOOP;
END;
```

4.3. Procedimientos y funciones

PL/SQL posibilita la utilización de procedimientos o funciones que permitan la **reutilización de código**.

La **definición** de un procedimiento o función tienen dos partes:

- **Especificación**: se declara el nombre del subprograma. Esta declaración está formada por el nombre y los tipos de los argumentos. Si hablamos de funciones, y solo en este caso, también se indicará el valor devuelto.
- **Cuerpo**: formado por sentencias PL/SQL que realizarán el objetivo para el cual se creó el procedimiento o la función.

- **Procedimientos**

Un procedimiento está formado por un conjunto de sentencias que se van a ejecutar cada vez que el procedimiento se invoque.

CÓDIGO:

```
-- Sintaxis
CREATE [ OR REPLACE ] PROCEDURE [ esquema ].nombre_procedure
(nombre_parametro { IN | OUT | IN OUT } tipo_de_dato, …)
{ IS | AS }
        declaraciones_de_variables;
        declaraciones_de_constantes;
        declaraciones_de_cursores;
BEGIN
        cuerpo_del_subprograma;
[EXCEPTION
        bloque_de_excepciones; ]
END;
```

En este caso:

- ***CREATE [OR REPLACE] PROCEDURE*** crea el procedimiento. Si se usa ***REPLACE*** se asegura de que, si existiera ya ese procedimiento, lo reemplace o sustituya.
- ***nombre_procedure*** indica el nombre del procedimiento.

- **Parámetro formal:** nombre del parámetro definido en la función, donde funcionan como variables locales con la particularidad de que se inician con el valor que tengan los parámetros actuales en el momento de llamarlo.
- **Parámetro actual:** son los valores que tienen los parámetros cuando se llaman a la función, es el valor que se asignan en la invocación a los parámetros formales.
- Parámetros para utilizar en la ejecución del procedimiento:
  - ***IN***: valor por defecto. El parámetro es de entrada. Cualquier modificación realizada en el procedimiento no afectará a su valor fuera del procedimiento, el parámetro actual no puede cambiar.
  - ***OUT***: el parámetro es de salida, es decir, en el procedimiento se le va a asignar un valor y este va a afectar al parámetro actual, pero no sirve para pasar un valor al procedimiento a través del parámetro formal.
  - ***IN OUT***: el parámetro se usa tanto para entrada como para salida.
- ***IS | AS***: no existe diferencia entre las dos, podemos poner la que deseemos.
- Declaraciones de variables, constantes y cursores.
- Entre *BEGIN* y *END* escribiremos el cuerpo del procedimiento.

- **Funciones**

Una función está formada por un conjunto de sentencias que se van a ejecutar cada vez que esta se invoque.

Esta definición coincide con la de procedimiento, ¿en qué se diferencian? La función va a devolver siempre un resultado que será un tipo de datos concreto que ha sido definido en su declaración.

Para devolver dicho valor se usará la palabra reservada ***RETURN***. Por ese motivo, deberá aparecer al menos una vez a lo largo del código.

CÓDIGO:

```
-- Sintaxis
CREATE [ OR REPLACE ] FUNCTION [ esquema ]. nombre_función
        ( nombre_parámetro tipo_de_dato, … )
RETURN tipo_de_dato
{ IS | AS }
        declaraciones_de_variables;
        declaraciones_de_constantes;
        declaraciones_de_cursores;
BEGIN
        cuerpo_del_subprograma;
[ EXCEPTION
      bloque_de_excepciones; ]
END;
```

En este caso:

- ***Tipo_de_dato***: indicaremos el tipo de dato que se va a devolver.

Realizar la llamada a los procedimientos o funciones que hemos creado podemos hacerlo de varias formas.

CÓDIGO:

```
-- Sintaxis
EXECUTE nombre_procedimiento;
CALL nombre_procedimiento();
--Podemos ejecutar el procedimiento dentro de un bloque
BEGIN
   nombre_procedimiento [ (Parametros) ];
END;
```

A continuación, vamos a ver unos ejemplos donde se ilustra el trabajo con procedimientos y funciones.

CÓDIGO:

```
-- Ejemplo
SET SERVEROUTPUT ON;
CREATE OR REPLACE PROCEDURE mensaje
IS
BEGIN
   dbms_output.put_line('¿Cómo estás? Este es un mensaje de prueba');
END;
-- Para ejecutar el procedimiento
EXECUTE mensaje;
```

CÓDIGO:

```
-- Ejemplo
SET SERVEROUTPUT ON;
CREATE OR REPLACE PROCEDURE duplicarEntero (num IN OUT INT)
AS
BEGIN
   num := 2 * num;
END;
-- Para ejecutar el procedimiento
DECLARE
    i int := 5;
BEGIN
   duplicarEntero(i);
   dbms_output.put_line(i);
END;
```

CÓDIGO:

```
-- Ejemplo
CREATE OR REPLACE FUNCTION Paridad(entero INT)
RETURN VARCHAR2 --Devuelve si el numero proporcionado es par o no
IS
esPar VARCHAR2(15);
BEGIN
   IF MOD(entero,2) = 0 THEN
      esPar := 'Es par';
   ELSE
      esPar := 'No es par';
   END IF;
   RETURN(esPar);
END;
-- Para ejecutar el procedimiento
SET SERVEROUTPUT ON;
BEGIN
   dbms_output.put_line(Paridad(1));
END;
```

CÓDIGO:

```
-- Ejemplo
CREATE OR REPLACE FUNCTION num2mes(num INT)
RETURN VARCHAR2 – Devuelve el mes
IS
 mes VARCHAR2(20);
BEGIN
   CASE num
      WHEN 1 THEN mes := 'Enero';
      WHEN 2 THEN mes := 'Febrero';
      WHEN 3 THEN mes := 'Marzo';
      WHEN 4 THEN mes := 'Abril';
      WHEN 5 THEN mes := 'Mayo';
      WHEN 6 THEN mes := 'Junio';
      WHEN 7 THEN mes := 'Julio';
      WHEN 8 THEN mes := 'Agosto';
      WHEN 9 THEN mes := 'Septiembre';
      WHEN 10 THEN mes := 'Octubre';
      WHEN 11 THEN mes := 'Noviembre';
      WHEN 12 THEN mes := 'Diciembre';
      ELSE mes := 'Error';
   END CASE;
   RETURN(mes);
END;
```

4.4. Control de errores

Una excepción es el resultado que se obtiene tras ejecutar un bloque PL/SQL que posee un error.

La **llamada al tratamiento de excepciones** puede producirse de **dos formas**:

1. Ocurre un error y la excepción se lanza de forma **automática**.
2. La lanzamos nosotros de forma explícita usando la sentencia ***RAISE***.

CÓDIGO:

```
-- Sintaxis
EXCEPTION
    WHEN exception1 [ or exception2 … ] THEN
            sentencias_para_ejecutar;
    WHEN exception3 [ or exception4 … ] THEN
            sentencias_para_ejecutar;
    WHEN OTHERS THEN
            sentencias_para_ejecutar;
END;
```

- **TIPOS:**

Podemos clasificarlas en:

- **Excepciones Oracle predefinidas:** Oracle las lanza de forma automática. Son excepciones estándares predefinidas a las que se ha dado nombre:

Nombre	Descripción
NO_DATA_FOUND	La sentencia *SELECT* no devuelve ningún valor.
TOO_MANY_ROWS	La sentencia *SELECT* devuelve más de una fila.
INVALID_CURSOR	Se está haciendo referencia a un cursor no válido.
ZERO_DIVIDE	Se está intentando realizar una división de un número entre cero.
CASE_NOT_FOUND	Ninguna de las condiciones de la sentencia *WHEN* en la estructura *CASE* se corresponde con el valor evaluado y no existe cláusula *ELSE*.
CURSOR_ALREADY_OPEN	El cursor que intenta abrirse ya está abierto.
INVALID_NUMBER VALUE_ERROR	La conversión de una cadena a valor numérico no es posible porque la cadena no representa un valor numérico válido.
VALUE_ERROR	Error ocurrido en alguna operación aritmética, de conversión o trucado.
LOGIN_DENIED	Un programa está intentado acceder a la base de datos con un usuario o *password* incorrecto.
NOT_LOGGED_ON	Un programa está intentado ejecutar una acción en la base de datos sin haber formalizado previamente la conexión.
TIMEOUT_ON_RESOURCE	Se ha acabado el tiempo que el SGBD pude esperar por algún recurso.
OTHERS	Es la opción por defecto. Interceptará todos los errores no tenidos en cuenta en las condiciones WHEN anteriores.

Por ejemplo:

CÓDIGO:

```
-- Ejemplo
DECLARE
      identificador NUMBER;
BEGIN
   SELECT nombre INTO identificador
   FROM Alumno
   WHERE cod = 100;
EXCEPTION
   WHEN NO_DATA_FOUND OR TOO_MANY_ROWS THEN
            DBMS_OUTPUT.PUT_LINE ('El error es: ' || SQLCODE );
            DBMS_OUTPUT.PUT_LINE ( SQLERRM );
END;
```

- **Excepciones Oracle NO-predefinidas**

Si se declaran en la sección declarativa pueden ser tratadas por Oracle de forma automática. Se debe definir el código Oracle asociado a esa excepción numérica, para ello se utiliza la sentencia *PRAGMA EXCEPTION_INIT*.

Si estamos utilizando programas como SQL Developer, hay que tener en cuenta que, cuando tenemos un error en el código, este nos muestra en la consola el código del error, ya que hay tal cantidad de errores que memorizarlos es imposible.

CÓDIGO:

```
-- Ejemplo
DECLARE
 fecha_incorrecta EXCEPTION;
 PRAGMA EXCEPTION_INIT (fecha_incorrecta, -01847);
 fecha DATE;
BEGIN
 fecha := TO_DATE('32-12-2017');
 dbms_output.put_line(fecha);
 EXCEPTION
  WHEN fecha_incorrecta THEN
   dbms_output.put_line('La fecha introducida es incorrecta');
END;
```

- **Errores definidos por el usuario**

No tienen relación con Oracle. Son condiciones que se determina que son errores. Se declaran en la sección declarativa, pero es necesario tratarlas de forma explícita.

Se lanza con la sentencia **RIASE:**

CÓDIGO:

```
-- Ejemplo
DECLARE
   negativo EXCEPTION;
   valor NUMBER;
BEGIN
    valor:= -1;
    IF valor < 0 THEN
        RAISE negativo;
    END IF;
EXCEPTION
    WHEN negativo THEN
      dbms_output.put_line('Números negativos NO permitidos');
END;
```

4.5. Cursores y transacciones

Los **cursores** son las diferentes estructuras que nos permiten recorrer la salida de una consulta realizada. **PL/ SQL** utiliza cursores para gestionar las diferentes instrucciones ***SELECT***. Recordemos que un cursor es un conjunto de registros devuelto por una instrucción SQL. Una definición más técnica para los cursores podría ser los fragmentos de memoria que tenemos reservados para procesar los resultados de una consulta ***SELECT***.

Debemos señalar los **dos tipos de cursores** que podemos tener:

- **Implícitos:** utilizados para operaciones *SELECT INTO*. Vamos a hacer uso de ellos cuando la consulta devuelve un único registro.
- **Explícitos:** estos son declarados y controlados por el programador cuando una consulta devuelve un conjunto de registros. Aunque, algunas veces, se utilizan en consultas que devuelven un único registro por razones de eficiencia. Son más rápidos.

Definimos los cursores de la misma forma que si fuera una variable de PL/ SQL. Aunque si bien es cierto, los cursores implícitos, no necesitan declaración.

Los cursores explícitos son necesarios cuando vamos a procesar instrucciones *SELECT* que devuelven más de una fila y/o atributos. En este caso, irán combinados con una estructura de bloque.

Los cursores, además, pueden utilizar parámetros que se declaran junto con el cursor.

- **Cursores implícitos**
  - **Declaración**

Los cursores implícitos se utilizan para realizar consultas ***SELECT*** que devuelven un único registro. Para utilizarlos, debemos tener en cuenta:

- Con cada cursor implícito debe existir la palabra clave ***INTO***.
- Las variables que reciben los datos devueltos por el cursor tienen que contener el mismo tipo de dato que las columnas de la tabla.
- Los cursores implícitos solo pueden devolver una única fila. En caso de que se devuelva más de una fila (o ninguna fila) se producirá una excepción. Por medio de PL/SQL gestionaremos los errores.

El siguiente ejemplo muestra un cursor implícito:

CÓDIGO:

```
-- Ejemplo
SET SERVEROUTPUT ON;
DECLARE
vnombre VARCHAR2(50);
BEGIN
SELECT nombre INTO vnombre  FROM Alumno  WHERE codigo=14;
DBMS_OUTPUT.PUT_LINE('El nombre del alumno es : ' || vnombre);
END;
```

La salida del programa generaría la siguiente línea:

```
El nombre del alumno es: Juan.
```

- **Excepciones asociadas a los cursores implícitos**

Los cursores implícitos solo pueden devolver una fila, por lo que pueden producirse determinadas excepciones. Las más comunes que se pueden encontrar son ***no_data_found*** y ***too_many_rows***.

La siguiente tabla explica brevemente estas excepciones:

Excepción	Explicación
NO_DATA_FOUND	Se produce cuando una sentencia *SELECT* intenta recuperar datos, pero ninguna fila satisface sus condiciones. Es decir, cuando no hay datos.
TOO_MANY_ROWS	Dado que cada cursor implícito solo es capaz de recuperar una fila, esta excepción detecta la existencia de más de una fila.

- **Cursores explícitos en PL/ SQL**

Los cursores son **áreas de memoria que almacenan datos extraídos de la Base de Datos mediante una consulta *SELECT*** o por manipulación de datos con sentencias de actualización o inserción de datos.

La utilización de cursores es necesaria cuando:

- Se precisa tratamiento fila a fila.
- En sentencias *SELECT* que devuelven más de una fila.

  - **Operaciones**

El cursor debe ser declarado (en la zona declare) antes de usarlo. Debe tener un nombre y estar asociado a una consulta determinada.

CÓDIGO:

```
-- Sintaxis
DECLARE
CURSOR nombre_cursor IS sentencia_SELECT;
```

Entre las diferentes operaciones que se pueden realizar:

1. ***Open***: para trabajar con un cursor debemos abrirlo e inicializarlo para que devuelva las filas.

   Al hacer *OPEN* del cursor ya se ejecuta la consulta y pone todas las filas devueltas en el conjunto activo.

CÓDIGO:

```
-- Sintaxis
OPEN nombre_cursor;
```

2. ***Fetch***: el cursor va a devolver la siguiente fila en el conjunto activo. Los datos devueltos se almacenan en variables de control o en un registro.

   Vamos a realizar esto haciendo uso de la orden *FETCH*.

CÓDIGO:

```
-- Sintaxis
FETCH nombre_cursor CURSOR INTO { [ var1, varN ] | nombre_registro };
```

Cada vez que se realice un *FETCH* el cursor avanza a la siguiente fila recuperada, por lo que es necesario comprobar los atributos del cursor para ver si el cursor tiene filas o ya ha llegado al final.

Se debe recorrer el cursor hasta encontrar la información que interese o hasta que no haya más filas. También es necesario usar un bucle para repetir estas acciones hasta conseguir el resultado buscado.

Un bucle de cursor ***FOR*** de forma implícita:

- Declara una variable *REGISTRO* de tipo *ROWTYPE*.
- Abre el cursor.
- De forma repetitiva realiza el *FETCH* de las filas sobre la variable registro.
- Cierra el cursor cuando todas las filas han sido procesadas.

3. ***Close***: para finalizar debemos cerrar el cursor. Conseguimos así desactivarlo y liberar la memoria que ocupaban los datos recuperados por este al abrirlo.

CÓDIGO:

```
-- Sintaxis
CLOSE nombre_cursor;
```

Existe un número limitado de cursores que pueden estar abiertos a la vez en la base de datos. Además, una vez cerrado no es posible recuperar datos hasta que no se abra de nuevo.

Para declarar un cursor debemos emplear la siguiente sintaxis:

CÓDIGO:

```
-- Sintaxis
DECLARE  CURSOR nombre_cursor IS <consulta_SELECT> ;
BEGIN
OPEN nombre_cursor;  -- Abrir el cursor
FETCH nombre_cursor INTO VARIABLES_PLSQL  -- Recorrer el cursor
CLOSE nombre_cursor ;  -- Cerrar el cursor
END;
```

También podemos declarar los posibles parámetros que necesite el cursor:

CÓDIGO:

```
-- Sintaxis
DECLARE  CURSOR nombre_cursor (parametro1 tipo1, parametro2
tipo2 parametro3 tipo3, .... ) IS <consulta> ;
BEGIN
OPEN nombre_cursor;  -- Abrir el cursor
FETCH nombre_cursor INTO VARIABLES_PLSQL  -- Recorrer el cursor
CLOSE nombre_cursor ;  -- Cerrar el cursor
END;
```

Los cursores implícitos no se pueden manipular por el usuario, pero Oracle sí permite el uso de sus atributos:

nombre_cursor %atributo_deseado;

Los atributos pueden ser:

Atributo	Significado
%ISOPEN	Devuelve un valor *booleano*, *TRUE* si está abierto el cursor o *FALSE* si está cerrado.
%NOTFOUND	Devuelve un valor *booleano*, *TRUE* si tras la recuperación más reciente no se recuperó ninguna fila.
%FOUND	Devuelve un valor *booleano*, *TRUE* si tras la recuperación más reciente se recuperó una fila.
%ROWCOUNT	Retorna el número de filas devueltas hasta el momento.

Cuando trabajamos con cursores debemos considerar una serie de aspectos:

- Cuando un cursor está cerrado no se puede leer.
- Cuando leemos un cursor debemos comprobar el resultado de la lectura utilizando los atributos de los cursores.
- Cuando se cierra el cursor es ilegal tratar de usarlo.
- Es ilegal tratar de cerrar un cursor que ya está cerrado o no ha sido abierto.

- **Atributos de cursores**

Pueden tomar los valores ***TRUE***, ***FALSE*** o ***NULL*** dependiendo de la situación:

Atributo	Antes de abrir	Al abrir	Durante la recuperación	Al finalizar la recuperación	Después de cerrar
%NOTFOUND	ORA-1001	NULL	FALSE	TRUE	ORA-1001
%FOUND	ORA-1001	NULL	TRUE	FALSE	ORA-1001
%ISOPEN	FALSE	TRUE	TRUE	TRUE	FALSE
%ROWCOUNT	ORA-1001	0	*	**	ORA-1001

** Número de registros que ha recuperado hasta el momento.*

*** Número total de registros.*

- **Manejo del cursor**

Por medio de ciclo *LOOP* podemos iterar a través del cursor. Debemos tener cuidado al agregar una condición para salir del bucle.

Vamos a ver varias formas de iterar a través de un cursor. La primera es utilizando un bucle *LOOP* con una sentencia *EXIT* condicionada:

CÓDIGO:

```
-- Sintaxis
OPEN nombre_cursor;
   LOOP
      FETCH nombre_cursor INTO lista_variables;
      EXIT WHEN nombre_cursor%NOTFOUND;
     /* Procesamiento de los registros recuperados */
   END LOOP;
 CLOSE nombre_cursor;
```

A continuación, vamos a ver un ejemplo donde se ilustra el trabajo con un cursor explícito.

Debemos tener en cuenta que, al leer los datos del cursor, debemos hacerlo sobre variables que sean del mismo tipo de datos de la tabla o tablas de las que trata el cursor, y tendremos que crear un bucle para recorrer los distintos registros que nos puede devolver la consulta.

CÓDIGO:

```
-- Ejemplo
SET SERVEROUTPUT ON;
DECLARE
  CURSOR calumno
  IS
  SELECT nombre, apellidos
  FROM Alumnos;
  vnombre VARCHAR2(50);
  vapellidos VARCHAR2(50);
BEGIN
  OPEN calumno;
  LOOP
    FETCH calumno INTO vnombre, vapellidos;
    EXIT WHEN calumno%NOTFOUND;
    dbms_output.put_line('Nombre: ' || vnombre || ' apellido: ' || vapellidos);
  END LOOP;
  CLOSE calumno;
END;
```

Podemos simplificar el ejemplo, utilizando el atributo del tipo ***%ROWTYPE*** sobre el cursor y cambiamos el bucle de iteración de registros para ver distintas formas de realizar un mismo ejemplo.

CÓDIGO:

```
-- Ejemplo
SET SERVEROUTPUT ON;
DECLARE
  CURSOR calumno
  IS
  SELECT nombre, apellidos
  FROM Alumnos;
  registro calumno%ROWTYPE;
BEGIN
  OPEN calumno;
  FETCH calumno INTO registro;
  WHILE calumno%FOUND LOOP
    dbms_output.put_line('Nombre: ' || registro.nombre
                                        || ' apellido: ' || registro.apellidos);
    FETCH calumno INTO registro;
  END LOOP;
  CLOSE calumno;
END;
```

- **Cursores de actualización**

  - **Declaración**

Se declaran de la misma forma que los explícitos, añadiendo ***FOR UPDATE*** al final de la sentencia ***SELECT***.

CÓDIGO:

```
-- Sintaxis
CURSOR nombre_cursor IS
    instrucción_SELECT
    FOR UPDATE [OF column_list] [NOWAIT]
```

Para actualizar los datos del cursor ejecutamos la sentencia ***UPDATE*** y especificamos la cláusula ***WHERE CURRENT OF <cursor_name>.***

CÓDIGO:

```
-- Sintaxis
UPDATE <nombre_tabla> SET
<campo_1> = <valor_1>
[,<campo_2> = <valor_2>]
    WHERE CURRENT OF <cursor_name>
```

Al trabajar con cursores de actualización se debe tener en cuenta que generan bloqueos en la base de datos.

A continuación, vamos a ver un ejemplo de cómo utilizar los cursores de actualización.

CÓDIGO:

```
-- Ejemplo
SET SERVEROUTPUT ON;
DECLARE
   CURSOR calumno IS
   SELECT nombre
   FROM alumnos
   FOR UPDATE;
   vnombre VARCHAR2(50);
BEGIN
   dbms_output.put_line('Valores antes de actualizarse');
   OPEN calumno;
   FETCH calumno INTO vnombre;
   WHILE calumno%found LOOP
      dbms_output.put_line('Nombre: '|| vnombre;);
      UPDATE alumnos
      SET nombre = nombre || '.'
      WHERE CURRENT OF calumno;
      FETCH calumno INTO vnombre;
   END LOOP;
   CLOSE calumno;
   COMMIT;

   dbms_output.put_line('');
   dbms_output.put_line('Valores después de actualizarse');
   OPEN calumno;
   FETCH calumno INTO vTitulo,iAno,vProtagonista;
   WHILE calumno %found LOOP
      dbms_output.put_line('Nombre: '|| vnombre;);
      FETCH calumno INTO vnombre;
   END LOOP;
   CLOSE calumno;
END;
```

- **Herramientas de depuración y control de código**

PL/SQL no está pensado para interactuar directamente con un usuario final, por ese motivo carece de instrucciones especializadas en la entrada y salida, teclado y pantalla.

Existen algunos métodos que nos permiten mostrar el resultado de lo que se está haciendo por pantalla, así como poder asignar valores a las variables desde teclado para ir probando cómo se comportan los programas con distintos valores en las variables, sin necesidad de ir modificando el código en cada caso.

1. La salida

Debemos activar la variable *SERVEROUTPUT* de Oracle. Solo tendremos que hacerlo una vez:

CÓDIGO:

```
-- Sintaxis
SET SERVEROUTPUT ON;
```

Tras su activación, se pueden mostrar textos, variables y constantes en pantalla usando la siguiente instrucción:

CÓDIGO:

```
-- Sintaxis
DBMS_OUTPUT.PUT_LINE ( cadena_de_salida );
```

CÓDIGO:

```
-- Ejemplos
-- Texto
DBMS_OUTPUT.PUT_LINE ('hola' );
-- Variables
DBMS_OUTPUT.PUT_LINE ( nombre_variable );
-- Constantes
DBMS_OUTPUT.PUT_LINE ( sysdate );
-- Combinaciones
DBMS_OUTPUT.PUT_LINE ('cadena de texto ' || contador || sysdate );
```

2. La entrada

La forma de leer valores desde teclado y conseguir asignarlos a una variable, consiste en utilizar el símbolo & seguido del mensaje que se visualizará al pedir la entrada del valor.

Este mensaje debe ser una cadena de caracteres sin espacios en blanco. Para separar las palabras del mensaje se recomienda hacerlo con guiones bajos.

Este tipo de recogida de valores se realizará al iniciar la ejecución del procedimiento, independientemente del lugar donde se realice la petición dentro del código.

- Entrada de valores a variables por teclado:

CÓDIGO:

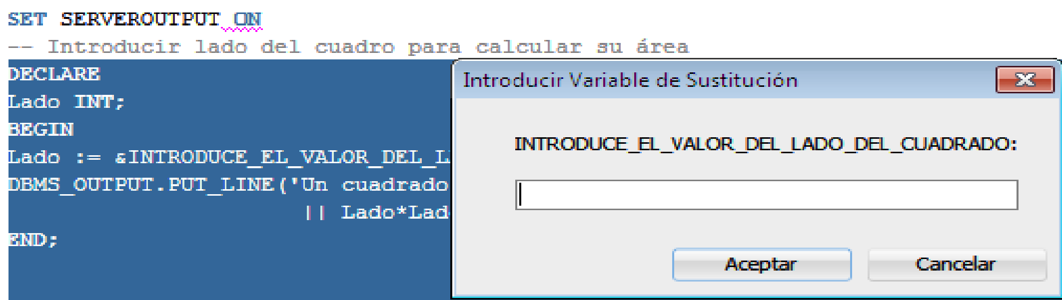
```
-- Sintaxis
nombreVariable := &Texto_para_mostrar;
```

Por ejemplo:

CÓDIGO:

```
-- Ejemplo
SET SERVEROUTPUT ON;
DECLARE
    Lado INT;
BEGIN
  -- Introducir lado del cuadrado para calcular su área
    Lado := &INTRODUCE_EL_VALOR_DEL_LADO_DEL_CUADRADO;
    DBMS_OUTPUT.PUT_LINE ('Un cuadrado de lado' || lado ||
    ' tiene un área de ' || Lado*Lado );
END;
```

Al ejecutar este bloque nos solicitarán el lado del cuadrado:



4.6. Disparadores o *triggers*

Un *trigger* es un **módulo PL/SQL compilado y almacenado en la Base de Datos que tiene asociada una tabla y que se ejecuta al llevar a cabo una instrucción SQL**. Van a ser muy útiles cuando queremos:

1. **Forzar reglas de integridad** que son difíciles de definir a partir de *constraints*.
2. Realizar **cambios en la base de datos** de forma transparente al usuario.
3. **Sincronizar el mantenimiento de tablas duplicadas** que están localizadas en nodos distintos de una base de datos distribuida.
4. **Generar automáticamente valores de columnas derivadas** en base a un valor proporcionado por una sentencia.

Uno de los usos más frecuentes de los *triggers* es **definir reglas y restricciones de integridad** de los datos, que sean complejas. Es importante que los datos de una base de datos estén sujetos a reglas de integridad predefinidas. Oracle permite definir y forzar que se verifiquen las reglas de integridad declarando restricciones de integridad y definiendo *triggers*.

Una **restricción de integridad** es un método declarativo de definir una restricción para una columna de una tabla. Oracle soporta:

1. ***Not Null***: no admite valores nulos.
2. ***Unique***: solo admite valores únicos.
3. ***Primary Key***: clave primaria.
4. ***Check***: reglas complejas.
5. ***Foreign Key***: restricciones de integridad referencial.

Aunque podemos definir y forzar cualquier tipo de regla de integridad se recomienda que solo se use en los siguientes casos:

- Definir acciones de integridad referencia que no se puedan forzar con *Foreign Key*.
- Forzar reglas de integridad referencia si las tablas padre e hija están en nodos distintos de una BDD distribuida.
- Forzar reglas complejas que no se puedan definir con restricciones de integridad.

CÓDIGO:

```
-- Sintaxis
CREATE [ OR REPLACE ] TRIGGER nombre_trigger
[ BEFORE | AFTER ]  [ DELETE | INSERT | UPDATE [ OF columnas ] ]
[ OR ]  [ DELETE | INSERT | UPDATE [ OF columnas ] ] ...]
ON tabla
[ FOR EACH ROW [ WHEN condición TRIGGER ] ]
[ DECLARE ]
-- declaración de variables locales
BEGIN
-- Instrucciones de ejecución
 [ EXCEPTION ]
-- instrucciones de excepción
END;
```

En este caso:

- ***BEFORE | AFTER*** indica en qué momento queremos que se ejecute el *trigger*, antes o después de la acción SQL:
    - ***BEFORE***: la usaremos cuando queremos revisar los cambios antes que se hagan efectivos sobre la tabla, para poder modificarlos en caso de que incumplan alguna restricción.
    - ***AFTER***: la usaremos en aquellos casos en los que se quiera consultar o cambiar la tabla en cuestión, ya que los *triggers* solo pueden llevar a cabo estas operaciones después de que los cambios iniciales son aplicados a la tabla y ésta se encuentra en un estado estable.

- ***DELETE | INSERT | UPDATE***: acciones SQL que disparan el *trigger*.
- ***FOR EACH ROW***: indicamos que el *trigger* se disparará por cada fila de la tabla sobre la cual se lleva a cabo la operación SQL.
- ***WHEN***: podemos incluir una condición de disparo del *trigger*.
- **Cuerpo del programa**, entre ***BEGIN*** y ***END***, se puede codificar cualquier orden de consulta o manipulación de datos, y llamadas a funciones o procedimientos como en cualquier código PL/SQL, respetando la integridad de la BDD. No se puede usar el control de transacciones ***COMMIT*** y ***ROLLBACK***. Los procedimientos y funciones a las que invoque el *trigger* deben cumplir también las restricciones anteriores.

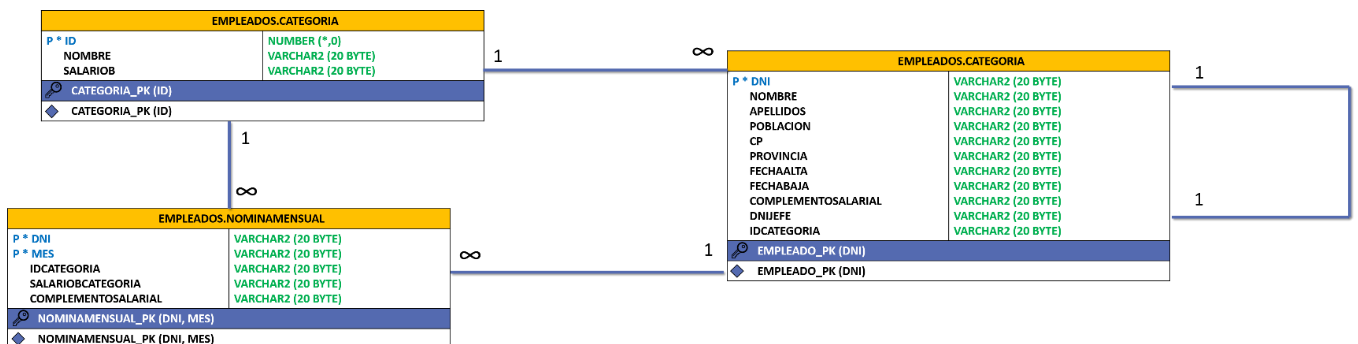
Un *trigger* puede dispararse por causa de más de una operación SQL, podemos utilizar los siguientes predicados condicionales:

- ***INSERTING***: devuelve ***TRUE*** cuando el *trigger* ha sido disparado por una orden *INSERT*.
- ***DELETING***: devuelve ***TRUE*** cuando el *trigger* ha sido disparado por una orden *DELETE*.
- ***UPDATING***: devuelve ***TRUE*** cuando el *trigger* ha sido disparado por una orden ***UPDATE***.
- ***UPDATING*** **(columna)**: devuelve ***TRUE*** cuando el *trigger* ha sido disparado por una orden ***UPDATE*** y la columna ha sido modificada.

Para hacer referencia en el código al valor nuevo o al valor viejo de las columnas que estamos modificando hacemos uso de las palabras reservadas ***OLD*** y ***NEW*** siguiendo las siguientes reglas:

- En las **modificaciones de una fila**, la referencia al valor antes de ser modificado (:OLD) y al valor después de la modificación (:NEW).
- En la **introducción de valores nuevos** se hace referencia solo al nuevo valor (:NEW).
- En los **borrados de valores** se hace referencia solo al antiguo valor (:OLD).
- En la **condición *WHEN*** de los *triggers* se deben utilizar quitando los dos puntos, es decir, *NEW* y *OLD*.

Por ejemplo, para las siguientes tablas:



- *Trigger* para validar las fechas de alta y de baja:

CÓDIGO:

```
-- Ejemplo
CREATE OR REPLACE TRIGGER empleados_fechaBaja_fechaAlta
BEFORE INSERT OR UPDATE OF FechaAlta, FechaBaja ON EMPLEADO
FOR EACH ROW
BEGIN
IF (:new.FechaBaja IS NOT NULL) AND (:new.FechaBaja <= :new.FechaAlta)
THEN
RAISE_APPLICATION_ERROR (-20600, 'La fecha de la baja debe ser nula o mayor
que la fecha de alta');
END IF;
END;
-- Sentencia con la que haríamos saltar el error del trigger creado
INSERT INTO EMPLEADO (FechaAlta, FechaBaja)
VALUES ('10-05-2017', '08-05-2017');
```

- *Trigger* para validar los empleados supervisados por el jefe:

CÓDIGO:

```
-- Ejemplo
CREATE OR REPLACE TRIGGER empleado_supervisores
BEFORE INSERT OR UPDATE OF DNIjefe ON EMPLEADO
FOR EACH ROW
DECLARE
        -- variable para contar los empleados subordinados
          v_supervisados INTEGER;
BEGIN
      SELECT count(*) INTO v_supervisados
      FROM empleado WHERE DNIjefe = :new.v_DNIjefe;
      -- cuenta los subordinados antes de actualizar o insertar
     IF (v_supervisados >= 10 ) THEN
          -- cuneta antes de la inserción o actualización del código jefe
          RAISE_APPLICATION_ERROR (-20601, 'El jefe no puede supervisar a más de 10
      empleados');
    END IF;
END;
```

UF4: Bases de Datos Objeto-Relacionales

1. Uso de las bases de datos objeto-relacionales

Las BBDD que hemos estado viendo hasta ahora ofrecen la posibilidad de crear entidades y atributos.

Las bases de datos pueden modelar o ir modificando todas estas características, mientras que las **bases de datos objeto-relacionales** son como una extensión del modelo relacional que hemos visto. No utilizan algunas reglas que antes sí llevábamos a cabo, pero añaden otra nueva solución a la hora de modelar la información.

La mayor diferencia que existe entre los dos modelos es la existencia de tipos de objetos. Los tipos se crean mediante sentencias **ODL (*Object Definition Lenguage*)** y van a ser la base del desarrollo de las bases de datos objeto- relacionales.

Cuando tengamos que definir el modelo objeto-relacional añadiremos a las características anteriores (entidades, relaciones, atributos y reglas) los tipos.

Tipos de objetos

Un tipo de objeto representa una entidad del mundo real que consta de las siguientes partes:

- Un nombre que permite identificar el tipo objeto
- Unos atributos que caracterizan el objeto
- Unos métodos que definen las operaciones sobre los datos de ese tipo escritos en PL/SQL.

1.1. Características

La principal característica que debemos destacar de las bases de datos objetos-relacionales es que combinan el modelo relacional, evolucionan con la incorporación de conceptos del modelo orientado a objetos.

Por ejemplo, este tipo de bases de datos permiten la creación de nuevos tipos de datos manteniendo todo lo que ya posee del modelo entidad-relación, entidades, atributos, relaciones y reglas.

En el modelo objeto- relacional:

1. Cada registro de una tabla se considera un **objeto**.
2. Y la definición de la tabla, su **clase**.

Este modelo tiene capacidad para gestionar tipos de datos complejos, lo cual, contradice algunas de las restricciones establecidas por el modelo relacional.

- Permite campos multivaluados, con lo cual, contradice frontalmente la 1FN.

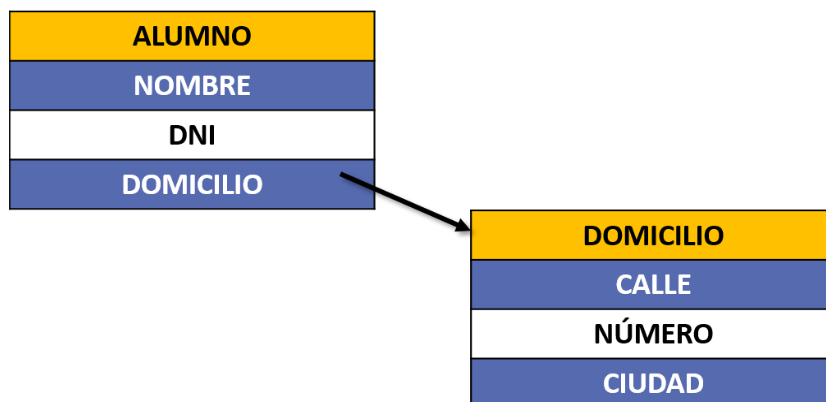
ALUMNOS		
DNI	**NOMBRE**	**ASIGNATURAS**
123456P	Paco	{M01, M03A}
321456O	Marta	{M05, M02B}

- Las tablas dejan de ser elementos bidimensionales para pasar a convertirse a estructuras de datos bastante más complejas.

Por ejemplo: una columna puede ser tipo *Asignatura* conteniendo toda la información de ésta y no únicamente su clave ajena.

ALUMNOS		
DNI	**NOMBRE**	**ASIGNATURAS**
123456P	Paco	{**aNombre**: Bases de datos **aCodigo**:M02A}

- De la misma forma, es posible también que los valores que adopte un campo sean registros de otra tabla, es decir, colecciones de objetos. Es lo que se conoce como **modelo relacional anidado**.



Por otra parte, las bases de datos relacionales se caracterizan porque se basan en una programación orientada a objetos (POO) que se desarrolla basándose en tres elementos fundamentales: **encapsulamiento**, **herencia** y **polimorfismo**.

- **Encapsulamiento**

Conocemos por encapsulamiento al **mecanismo que vamos a seguir para agrupar los atributos y métodos** dentro de un nuevo concepto que denominamos **clase.**

Mediante el uso de clases podemos abstraer un tipo de objeto como un conjunto de datos y acciones que están agrupadas. Si utilizamos el encapsulamiento debemos tener definidos todos los métodos que vayan a formar parte de las clases que intervengan. Estos métodos, deben acompañar al objeto.

- **Herencia**

La herencia es el **mecanismo por el cual una clase derivada va a heredar los atributos de otra.** Actúa de forma jerárquica, es decir, puede seguir el principio de derivación (de arriba a abajo) o de generalización (de abajo a arriba).

Todo aquello que esté definido en la clase principal, no hace falta que vuelva a definirse en los tipos derivados a menos que queramos hacer cualquier modificación.

- **Polimorfismo**

Se utiliza el polimorfismo cuando **una clase derivada debe verse como la clase principal.** Esto significa que se va a disponer de un objeto de una clase derivada, pero nos vamos a referir a él como si fuera un objeto de una clase principal.
Además, nos permite ejecutar diversas funciones que han sido sustituidas, como, por ejemplo: *overrided* (sobrecargadas) o *overwritten* (sobrescritas). PL/ SQL utiliza la sobrecarga.

1.2. Tipos de datos Objeto

Los objetos han conseguido entrar en el mundo de las bases de datos en forma de dominios, actuando como el tipo de datos de una columna determinada.

A continuación, vamos a ver **dos implicaciones muy importantes que se producen por el hecho de utilizar una clase como un dominio**:

- Es posible almacenar múltiples valores en una columna de una misma fila ya que un objeto suele contener múltiples valores. Sin embargo, si se utiliza una clase como dominio de una columna, en cada fila esa columna solo puede contener un objeto de la clase (se sigue manteniendo la restricción del modelo relacional de contener valores atómicos en la intersección de cada fila con cada columna).
- Es posible almacenar procedimientos en las relaciones porque un objeto está enlazado con el código de los procesos que sabe realizar (los métodos de su clase).

Otro modo de incorporar objetos en las bases de datos relacionales es construyendo tablas de objetos, donde cada fila es un objeto.

Ya que un sistema objeto–relacional es un sistema relacional que permite almacenar objetos en sus tablas, la base de datos sigue sujeta a las restricciones que se aplican a todas las bases de datos relacionales y conserva la capacidad de utilizar operaciones de concatenación (*JOIN*) .

Los usuarios pueden definir sus propios tipos de datos a partir de los tipos básicos provistos por el sistema, o por otros tipos de datos predefinidos anteriormente por el usuario. Estos tipos de datos pueden pertenecer a dos categorías distintas: objetos y colecciones.

1.3. Definición de tipos de Objeto.

En Oracle los tipos de objetos son aquellos tipos de datos que han sido definidos por el usuario. Los objetos se presentan de forma abstracta, realmente se siguen almacenando en columnas y tablas.

Para poder crear tipos de objetos se debe hacer uso de la sentencia ***CREATE TYPE***. Está compuesta por los siguientes elementos:

1. Para identificar el tipo de objetos se utiliza un **nombre**.
2. Unos **atributos** que pueden ser de un tipo de datos básico o de un tipo definido por el usuario, que representan la estructura y los valores de los datos de ese tipo.
3. Unos **métodos** que son procedimientos o funciones. Se declaran con la cláusula **MEMBER**

CÓDIGO:

```
-- Ejemplo
CREATE OR REPLACE TYPE persona AS OBJECT (
   nombre VARCHAR2(30);
   teléfono VARCHAR2(20)
);
```

Se puede observar que cada objeto persona tendrá dos atributos: el nombre y el teléfono.

Vamos a ver, a continuación, la definición de un tipo objeto:

CÓDIGO:

```
-- Sintaxis
CREATE OR REPLACE TYPE nombreObjeto AS OBJECT (
  atributo1 TIPO,
  atributo2 TIPO,
  atributoN TIPO,
  MEMBER FUNCTION nombreFuncion1 RETURN TIPO,
  MEMBER FUNCTION nombreFuncion2(nomVar TIPO) RETURN TIPO,
  MEMBER PROCEDURE nombreProcedimiento
  PRAGMA RESTRICT REFERENCES (nombreFuncion1, restricciones),
  PRAGMA RESTRICT REFERENCES (nombreFuncion2, restricciones),
  PRAGMA RESTRICT REFERENCES (nombreProcedimiento, restricciones)
);
```

Un método miembro de un tipo de objeto 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 va a acceder a los valores de la una variable dentro de un módulo
- No puede modificar una variable de un módulo excepto si se invoca desde una cláusula SELECT, VALUES o SET
- No puede invocar a otro módulo o subprograma que rompa alguna de las reglas anteriores.

La directiva **PRAGMA_REFERENCES** se utiliza para forzar las reglas anteriores.

CÓDIGO:

```
-- Sintaxis

PRAGMA RESTRICT REFERENCES ({DEFAULT | método}, {RNDS, WNDS, RNPS, WNPS},
[RNDS, WNDS, RNPS, WNPS]});
```

Donde *restricciones* puede ser cualquiera de las siguientes o incluso una combinación de ellas:

- ***WINDS***: evita que el método pueda modificar las tablas de la base de datos.
- ***RNDS***: evita que el método pueda leer las tablas de la base de datos.
- ***WNPS***: evita que el método modifique variables del paquete PL/SQL.
- ***RNPS***: evita que el método pueda leer las variables del paquete PL/SQL.

Veamos ahora la definición del cuerpo de un objeto:

CÓDIGO:

```
-- Sintaxis
CREATE OR REPLACE TYPE BODY nombreObjeto AS
  MEMBER FUNCTION nombreFuncion
  RETURN TIPO
  IS
  BEGIN
     --Código de la función
  END nombreFuncion;
  MEMBER FUNCTION nombreFuncion2(nomVar TIPO)
  RETURN TIPO
  IS
  BEGIN
     --Código de la función 2
  END nombreFuncion2;
  MEMBER PROCEDURE nombreProcedimiento
  IS
  BEGIN
    --Código del proceso
  END nombreProcedimiento;
END;
```

Por ejemplo:

CÓDIGO:

```
-- Ejemplo Definición de clase
CREATE OR REPLACE TYPE Persona AS OBJECT(
    idpersona NUMBER,
    dni VARCHAR2(9),
    nombre VARCHAR2(15),
    apellidos VARCHAR2(30),
    fecha_nac DATE,
    MEMBER FUNCTION muestraEdad RETURN NUMBER,
    PRAGMA RESTRICT_REFERENCES (muestraEdad, WNDS)
);
```

CÓDIGO:

```
-- Ejemplo Cuerpo de clase
CREATE OR REPLACE TYPE BODY persona AS
  MEMBER FUNCTION muestraEdad RETURN NUMBER
  IS
  BEGIN
    RETURN to_char(sysdate, 'YYYY')-to_char(fecha_nac, 'YYYY');
  END muestraEdad;
END;
```

Una vez que ya tenemos definida la clase y el cuerpo, podemos utilizar este objeto como si de un tipo cualquiera se tratase.

CÓDIGO:

```
-- Ejemplo
SET SERVEROUTPUT ON;
DECLARE
  Trabajador1 Persona;
BEGIN
  Trabajador1:= NEW Persona(1, '123456', 'Alberto', 'Olivia',
                            '22/12/1989');
  dbms_output.put_line('El empleado '|| Trabajador1.nombre
                       ||' ha sido creado');
  dbms_output.put_line('Edad del empleado: '
                       ||Trabajador1.muestraEdad());
END;
```

- **Colecciones**

Para poder contar con atributos multievaluados (1:N) es necesario que los organicemos en una estructura de datos, una estructura es conocida como *array* o colección. Para crear un atributo de tipo *array* debemos usar la siguiente sintaxis.

CÓDIGO:

```
-- Ejemplo
-- Lista de un tipo primitivo varchar2
CREATE TYPE colec_nombres AS VARRAY(10) OF VARCHAR2 (20);
-- Lista de un tipo de usuario: Persona
CREATE TYPE colec_personas AS VARRAY(10) OF Persona;
```

Una vez que ya hemos creado un tipo de colección podemos utilizarla dentro de la definición de una clase:

CÓDIGO:

```
-- Ejemplo
CREATE OR REPLACE TYPE Departamento AS OBJECT(
        NombreDept VARCHAR2(100),
        Empleados colec_personas
);
```

- **Tablas derivadas de objetos**

De la misma forma que creamos objetos haciendo uso de los tipos definidos, también podemos hacerlo con las tablas:

CÓDIGO:

```
-- Ejemplo
CREATE TABLE Empleados (
        NombreDept VARCHAR2(50),
        Datos_Empleado Persona
);
```

Para hacer un *insert* debemos indicar el tipo de objeto que estamos insertando:

CÓDIGO:

```
-- Ejemplo
INSERT INTO Empleados
VALUES ('Informática', Persona(1, '1111', 'Paco' , 'González', '23/12/1988'));
```

1.4. Herencia

Una de las principales ventajas de la programación orientada a objetos (POO) es la herencia. Gracias a la herencia se pueden crear **superclases abstractas** para que, en adelante, se puedan crear **subclases más específicas** que hereden atributos y métodos de las superclases.

En las bases de datos objeto-relacional se puede hacer algo parecido en lo que se refiere a los tipos de objetos. Pueden crearse **subtipos** de objetos a partir de otros supertipos de objetos que ya han sido creados con anterioridad.

El supertipo define los atributos o métodos que van a compartir con los subtipos, todos los objetos que hereden de él. A parte de estos, los subtipos también **pueden definir sus propios atributos y métodos**. También pueden redefinir los métodos del supertipo, para adoptarlo a sus necesidades. Este proceso se conoce como polimorfismo.

Veamos, por ejemplo, desde el objeto general supertipo *TIPO_PERSONA* podemos definir el subtipo *TIPO_EMPLEADO* que va a heredar las características de su supertipo *TIPO_PERSONA*.

El tipo objeto especializado *TIPO_EMPLEADO* puede tener atributos adicionales o incluso puede redefinir métodos del objeto padre *TIPO_PERSONA* ampliando la funcionalidad de sus tipos objeto.

- **Tipo persona**

CÓDIGO:

```
-- Ejemplo

CREATE TYPE tipo_person AS OBJECT (
   nombre VARCHAR2(25),
   fecha_nac DATE,
   MEMBER FUNCTION edad RETURN NUMBER,
   MEMBER FUNCTION printme RETURN VARCHAR2
) NOT FINAL;
```

La cláusula NOT FINAL hace referencia a que este objeto no es el último; es decir, cuando creemos subtipos de objetos que cuelguen de este, tendremos que declarar el objeto con esta cláusula. En el caso de no poner la cláusula al crear este objeto, cuando queramos colgar otros objetos de este nos devolverá error, aunque nos compilara el objeto.

- **Subtipo empleado**: para crear un subtipo utilizamos la cláusula *UNDER*.

CÓDIGO:

```
-- Ejemplo

CREATE TYPE tipo_empleado UNDER tipo_person (
     sueldo NUMBER,
     MEMBER FUNCTION paga RETURN NUMBER,

    OVERRIDING MEMBER FUNCTION printme RETURN VARCHAR2);
```

Heredamos los atributos y los métodos añadiendo características propias de *tipo_person* y sobreescribimos el método *printme*

1.5. Identificadores, referencias

El tipo *REF* o referencia es un contenedor de un identificador de objeto (*object identifier – OID*). Es un puntero a un objeto.

Su utilidad se basa, principalmente, en apuntar a un objeto que ya existe y así evitar tener que duplicar la información que el objeto almacena.

Por ejemplo, si se desea crear una tabla de libros en la que cada libro tiene su autor, y esos autores ya están almacenados en una tabla de objetos, se podría hacer uso de la palabra reservada **REF** para indicar que el autor ya existe y, por tanto, solo se almacena una referencia a ese autor en la tabla de libros.

Con la palabra reservada **DEREF**, obtenemos el valor de un objeto al que apunta dicha referencia.

CÓDIGO:

```
-- Ejemplo
CREATE TYPE autor AS OBJECT (
        id INTEGER,
        nombre VARCHAR(100),
        direccion VARCHAR(255)
);
CREATE TYPE libro AS OBJECT (
        id INTEGER,
        nombre VARCHAR(200),
        escritor REF autor
);
```

1.6. Tipos de datos colección

Los tipos para colecciones se definen para poder implementar relaciones 1:N. Un dato de tipo colección está formado por un número indefinido de elementos, todos del mismo tipo. Así es posible almacenar un conjunto de tuplas en un único atributo, en forma de *array* o de tabla anidada.

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 paréntesis.

- **Tipo *Varray***

Un *array* es un conjunto de elementos ordenados del mismo tipo. Todos los elementos que lo componen poseen asociado un índice que indica la posición que este ocupa dentro del *array*.

CÓDIGO:

```
-- Ejemplo
CREATE TYPE colec_telefono AS VARRAY (10) OF VARCHAR2 (30);
```

Con esta instrucción, definimos un nuevo tipo de datos llamado *colec_telefono* que va a tener como máximo grupos de 10 valores y, además, van a tener 30 caracteres como máximo.

Creamos la tabla para representar la siguiente estructura:

CÓDIGO:

```
-- Ejemplo
CREATE TABLE alumno (
   Id NUMBER,
   Nombre VARCHAR2(20),
   Telefonos colec_telefono
);
```

ID	Nombre	Teléfonos
1	Carmen	(955701212, 614785222)
2	Antonio	(957842563, 645458523)

A la hora de insertar datos lo hacemos como en cualquier tabla:

CÓDIGO:
```
-- Ejemplo
INSERT INTO alumno
VALUES (1, 'Carmen' colec_telefono ('955701212', '614785222'));
```

Podemos comprobar que es necesario especificar el tipo *colec_telefono* para que el SGBD realice las comprobaciones necesarias.

- **Tablas anidadas**

Una tabla es un conjunto de elementos del mismo tipo. Se diferencian del tipo anterior porque en este tipo no existe un orden predefinido.

Las tablas anidadas poseen una serie de restricciones:

- Solo pueden tener una columna.
- El tipo de dato que almacena puede ser básico o definido por el usuario.

Si redefinimos nuestro ejemplo anterior, redefinimos el tipo alumnos de la siguiente forma:

CÓDIGO:
```
-- Ejemplo
CREATE TYPE tabla_telefono AS TABLE OF VARCHAR2 (30);
```

Creamos la tabla basándola en el tipo *tabla_alumno*:

CÓDIGO:
```
-- Ejemplo
CREATE TABLE alumno (
    id NUMBER,
    nombre VARCHAR2 (20),
    telefonos tabla_telefono)
    NESTED TABLE telefonos STORE AS t_telefono;
```

Ahora la *tabla_telefono* ya no es un tipo colección, es un tipo tabla.

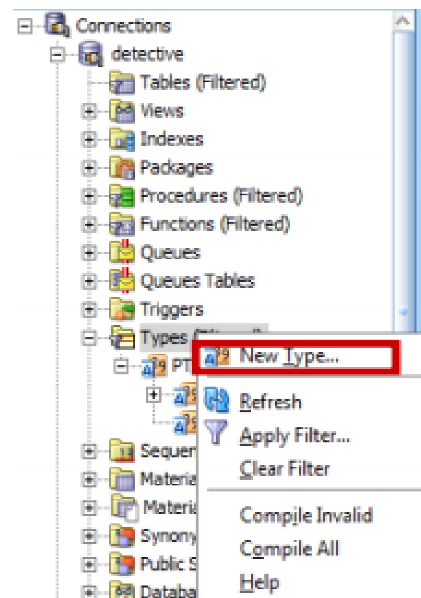
Las columnas que son tablas anidadas junto con los atributos que son tablas de objetos necesitan de una tabla independiente donde almacenar las filas de dichas tablas. Para especificar esta tabla se utiliza la cláusula *NESTED TABLE...STORE AS...*.

Para consultar, insertar, borrar o actualizar información en tablas anidadas, se realiza igual que con las colecciones.

1.7. Declaración e inicialización de Objetos

Para declarar un objeto en la base de datos Oracle tenemos que crear un nuevo tipo de datos. Podemos llevarlo a cabo de dos formas diferentes:

- Mediante **SQL Developer** podemos conectarnos a una base de datos ya creada en Oracle y, a partir de ahí, seleccionamos en el panel de conexiones, *Types*.



- De una forma más directa, utilizando la instrucción *CREATE TYPE*, cuya sintaxis es:

CÓDIGO:

```
-- Sintaxis
CREATE [OR REPLACE] TYPE [schema.] type_name
[OID 'object_identifier']
[object_type
| | {IS | AS} {varray_type_def | nested_table_type_def}
];
```

De forma que la definición de *object_type* se corresponde con:

CÓDIGO:

```
-- Sintaxis
 [inviker_rights_clause]
{{IS | AS} OBJECT
| UNDER [schema.]supertupe
}[sqlj_object_type]
[({attribute datatype [ sqlj_object_type_attr]}…
[, element_spec [, element_spec]…]
)
][[NOT] FINAL] [ [NOT] INSTANTIABLE]
```

Si nos basamos en SQL Developer, el programa nos genera de forma automática la estructura básica para la creación de un objeto. Aunque dentro de la misma tenemos que definir las propiedades principales del objeto en cuestión, como podemos apreciar en el siguiente código:

CÓDIGO:

```
-- Creación y asignación de propiedades
CREATE OR REPLACE TYPE OBJPRODUCTO AS OBJECT
(
codigo varchar2(20), nombre varchar2(60), familia varchar2(60),
altura float, anchura float, longitude float, peso float, color varchar2(20),
precioCompra float, precioVenta float, fechaCr date
);
```

1.8. Sentencia SELECT. Inserción, modificación y borrado de objetos

- **Inserción**

Cuando tengamos que insertar objetos en una tabla, utilizaremos el comando *INSERT*:

CÓDIGO:

```
-- Ejemplo
BEGIN
   INSERT INTO person
   VALUES ('Juan', 'Mata');
END;
```

Aunque también tenemos la posibilidad de poder utilizar el constructor para el objeto en cuestión:

CÓDIGO:

```
-- Ejemplo
BEGIN
   INSERT INTO person
   VALUES (Person ('Juan', 'Mata'));
END;
```

A continuación vamos a ver un script con la tabla relacional departamento. Tiene una columna tipo *person* y va a insertar una fila en esa tabla:

CÓDIGO:

```
-- Ejemplo
CREATE TABLE departamento (
    dept_nombre VARCHAR2(20),
    jefe_dept Person,
    localidad VARCHAR2(20)
);
-- Inserción de datos
INSERT INTO departamento
VALUES ('Contabilidad', Person ('Adrián', 'Tormo'), 'Lleida');

--Consultamos el tipo de dato
SELECT departamento.person.apellido FROM departamento;
```

- **Modificación**

A la hora de modificar atributos de un objeto determinado en una tabla de objetos vamos a utilizar la sentencia *UPDATE*. Por ejemplo:

CÓDIGO:

```
-- Ejemplo
BEGIN
   UPDATE person p SET p. direccion = 'c/larga, 7'
   WHERE p. apellido = 'López';
   UPDATE person p SET p = Person ('Eva', 'García')
   WHERE p. apellido = 'López';
END;
```

- **Borrado**

Para conseguir eliminar objetos (filas) de una tabla, utilizaremos el comando *DELETE*. Si tenemos que hacer un borrado selectivo añadiremos la cláusula *WHERE*. Por ejemplo:

CÓDIGO:

```
-- Ejemplo
BEGIN
   DELETE FROM person p
   WHERE p.direccion = 'c/larga, 7';
END;
```

Bibliografía

Alfonso González (2010). *Gestión de Bases de Datos*. Madrid.

Arturo Mora (2014). *Bases de Datos. Diseño y Gestión*. Vallehermoso, Madrid.

Iván López, Manuel de Castro, John Ospino (2014). *Bases de Datos.* Madrid.

Luis Hueso Ibáñez (2012). *Bases de datos*. Madrid

ILERNA Online