

Mantenimiento de la persistencia de los objetos.

Caso práctico



Antonio esta mañana está muy contento, y todo es porque una aplicación que estaba haciendo, le ha funcionado. Es una aplicación sencilla que le ha pedido **Juan**, destinada a organizar la lista de clientes de un pequeño taller mecánico. La aplicación permite introducir los datos de los conductores y las conductoras que acuden al taller, recopilando sus direcciones de correo electrónico y sus teléfonos.

De momento la aplicación no hace nada más, de hecho, el cliente no necesita ninguna funcionalidad extra, aunque deja la puerta abierta a que en el futuro se pueda añadir más información (como por ejemplo las matriculas de los coches de cada conductor o conductora). Hoy va a enseñarle a Juan la aplicación:

—Hola Juan. Hoy he traído la aplicación del taller que te dije, para que me des tu opinión —dice Antonio.

—Muy bien -contesta Juan-, seguro que será una aplicación estupenda.

—Bueno, todavía me falta mucho por hacer, pero ya se puede utilizar. ¡Mira aquí está!

Después de un rato, en el que Antonio le enseña como funciona la aplicación, Juan hace un comentario inesperado:

—Está muy bien la verdad, ya solamente te queda hacer los datos persistentes —dice Juan.

—Sí, supongo que sí —contesta Antonio.

Antonio responde que sí, pero realmente no sabe muy bien a qué se refiere con lo de datos persistentes, por lo que se queda un poco desconcertado.

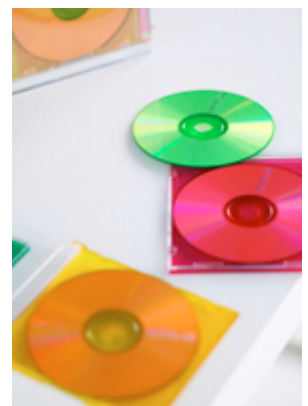
¿Utilizarías un procesador de textos que no te da la opción de guardar el documento que estás editando? Como es obvio, a nadie se le ocurre hacer un programa así. De hecho, casi todos los programas hoy día tienen la opción de guardar los datos, sean o no procesadores de texto.

Hasta ahora, ya conoces como abrir un archivo y utilizarlo como “almacén” para los datos que maneja tu aplicación. Utilizar un archivo para almacenar datos es la forma más sencilla de persistencia, porque en definitiva, **la persistencia es hacer que los datos perduren en el tiempo.**

Hay muchas formas de hacer los datos de una aplicación persistentes, y muchos niveles de persistencia. Cuando los datos de la aplicación solo están disponibles mientras la aplicación se esta ejecutando, tenemos un nivel de persistencia muy bajo, y ese era el caso de Antonio: su aplicación no almacenaba los datos en ningún lado, y en posteriores ejecuciones los datos no podían ser utilizados, pues solo estaban disponibles mientras no cerraras la aplicación.

Lo deseable es que los datos de nuestra aplicación tengan un nivel de persistencia lo mayor posible. Tendremos un mayor nivel de persistencia si los datos “sobreviven” varias ejecuciones, o lo que es lo mismo, si nuestros datos se guardan y luego son reutilizables con posterioridad. Tendremos un nivel todavía mayor si “sobreviven” varias versiones de la aplicación, es decir, si guardo los datos con la versión 1.0 de la aplicación y luego puedo utilizarlos cuando este disponible la versión 2.0.

Pero lo verdaderamente interesante de esta unidad, es la forma de hacer persistentes los datos. En esta unidad te vamos a proponer un enfoque totalmente diferente a almacenar los datos en meros archivos: vamos a utilizar bases de datos orientadas a objetos para hacer los datos de tu aplicación persistentes.



Pero para hacer esto, primero tienes que aprender qué son las bases de datos orientadas a objetos, en adelante llamadas BDOO, y aprender a usarlas en tu aplicación. Comprobarás que con las BDOO hacer los datos persistentes es más fácil y práctico de lo que imaginabas.



Materiales formativos de FP Online propiedad del Ministerio de Educación, Cultura y Deporte.

[Aviso Legal](#)

1.- Introducción a las BDOO.

Caso práctico

Antonio está un poco desconcertado, pensando a qué se puede referir Juan con eso de “hacer persistentes los datos”. Después de un rato, **Antonio** se cruza con **Ana** y le pregunta sin dudarlo:

—Oye, ¿tú sabes en qué consiste la persistencia de datos? — pregunta Antonio.

—Claro, la persistencia de datos es simplemente almacenar los datos de forma que luego puedan volverse a utilizar después, por ejemplo, usando una base de datos. ¿Por qué lo preguntas?

—Por nada, ya lo entiendo, o sea, que se refiere a usar bases de datos.

—Bueno, en realidad es algo más que eso, existen multitud de mecanismos para realizar la persistencia de datos. Si buscas por Internet encontrarás cientos de páginas con información al respecto.



Hoy en día, uno de los retos más interesantes del mundo de la informática, es simplificar la gestión de datos. Seguro que estarás de acuerdo en que es imprescindible guardar los datos de la aplicación, pero hay que reconocer que es una de las tareas más engorrosas, sobre todo si se trata de ir guardando los datos en archivos.

Las técnicas de persistencia persiguen, básicamente, un objetivo muy noble: hacer la vida más fácil al programador o a la programadora, simplificando para ello los mecanismos para guardar los datos. A continuación vamos a revisar algunas de las técnicas existentes para realizar la persistencia, valorando como simplifican el acceso a los datos:



- ✓ **Almacenamiento directo en archivos.** Almacenar los datos directamente en archivos es, como ya sabes, una de las técnicas de persistencia más usada, pero no es sencilla de implementar. Implementar la persistencia en archivos suele ser costoso y de difícil mantenimiento.
- ✓ **Sistema gestor de bases datos (SGDB).** Usar un sistema SGDB es una de las soluciones más recurridas. **Los datos son gestionados por el SGDB, garantizando consistencia y seguridad en los mismos.** Desde la aplicación se accede a los datos usando una [API](#) específica, como [JDBC](#), y usando un lenguaje de consulta de datos, como [SQL](#). Aunque los SGDB facilitan la gestión de datos, todavía se buscan soluciones de programación más cómodas.
- ✓ **Mapeado de objetos.** El mapeado de objetos simplifica bastante la utilización de las bases de datos. Se trata básicamente de técnicas que permiten almacenar objetos de un lenguaje de programación, como Java, directamente en la base de datos. Generalmente, necesitan de lo que denominamos [motor de persistencia](#).
- ✓ **Extensiones de lenguajes de programación tradicionales para facilitar el acceso a datos.** Este tipo de técnicas amplían la funcionalidad de los lenguajes de programación tradicionales, como Cobol, C, [C++](#), [C#](#) o Java, para hacer más sencillo el acceso a los datos. [SQLJ](#) o [PRO*C](#) son algunos ejemplos de extensiones de los lenguaje Java y C respectivamente. Algunas de estas extensiones, como [LINQ](#), están pensadas para varios lenguajes de programación.

Para saber más

En el siguiente enlace encontrarás más información acerca de LINQ.



Autoevaluación

¿Cuál de las siguientes opciones es una extensión de un lenguaje de programación tradicional que facilita el uso de bases de datos?

- ☐ SQL.
- ☐ SQLJ.
- ☐ Mapeado de objetos.

1.1.- ¿Qué son las BDOO? (I)



Después de tantas unidades de programación vistas, que en esta unidad te hablen de objetos te va a resultar un poco repetitivo, ¿verdad? No obstante, vamos a hacer un pequeño recordatorio sobre lo que entendemos por objeto: **un objeto en programación es una entidad que contiene datos y operaciones sobre dichos datos.**

Seguro que el concepto de objeto ya lo sabías, ahora la pregunta es, ¿qué entendemos por BDOO? **Las BDOO son bases de datos que almacenan objetos**, así de simple. Y al igual que los objetos que utilizas en Java, los objetos almacenados en una BDOO encapsulan igualmente datos y operaciones en una misma entidad. Como ya sabrás, los datos de un objeto los llamamos **atributos** y

las operaciones sobre dichos datos las denominamos **métodos**.

Una BDOO es solamente un “almacén de objetos”. Al software que proporciona la facilidad de almacenar objetos en ese almacén lo conocemos como Sistema Gestor de Bases de Datos Orientadas a Objetos (SGBDOO). En muchos libros los llaman Sistemas Gestores de Bases de Objetos (SGBO), lo cual es equivalente.

Desde nuestra aplicación solicitaremos al SGBDOO que almacene un objeto, o que lo busque dentro de todos los que tiene almacenados, de esta forma, lograremos hacer los datos de nuestra aplicación persistentes, aunque a partir de ahora podremos decir que vamos a hacer los objetos de nuestra aplicación persistentes.

Cabe decir que las BDOO no son las más utilizadas hoy día. El modelo relacional, usado por los Sistemas Gestores de Bases de Datos Relacionales (SGBDR), es el modelo de base de datos más utilizado. Los SGBDR almacenan la información en relaciones o tablas, formadas por un conjunto de filas y columnas. A pesar de que los SGBDR son los más usados, en esta unidad se abordan los SGBDOO porque tienen características que los hacen más flexibles.

Además de los SGBDR y los SGBDOO, existen lo que denominamos Sistemas Gestores de Bases de Datos Objeto-Relacionales (SGBDOR). La mayor parte de los SGDB comerciales son de tipo Objeto-Relacional, y son bases de datos que implementan el modelo relacional, y que además incorporan conceptos de orientación a objetos. Algunas bases de datos de este tipo solo incorporan algunos conceptos de orientación a objetos, como la herencia. Otras implementan muchos más conceptos de orientación a objetos, convirtiéndose en sistemas verdaderamente flexibles.

En inglés las bases de datos relacionales se denominan RDBMS (Relational Database Management System) y las orientadas a objetos ODBMS (Object Database Management System). Como podrás deducir, a las bases de datos objeto-relacionales se las denomina ORDBMS (Object-Relational Database Management System).

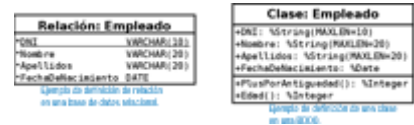
A continuación, se muestra una pequeña relación de SGDB clasificados por tipo (hay muchos más):

- ✓ Orientados a objetos: DB4O, Versant Object Database, InterSystems Caché, ObjectStore, Objectivity DB, MATTISE, etc.
- ✓ Objeto-relacionales: Oracle, IBM DB2, y PostgreSQL, etc.
- ✓ Relacionales: MySQL, Microsoft SQL Server, Microsoft Access, Interbase SMP, etc.

1.1.1.- ¿Qué son las BDOO? (II)

Al igual que en ocurre en Java, para poder usar un objeto cualquiera, primero tienes que definirlo, creando la clase pertinente. Definir un objeto consiste en hacer una descripción de cómo va a ser, indicando qué atributos y métodos van a tener todos los objetos de un mismo tipo.

Tanto en las bases de datos relacionales, como en las BDOO, es necesario definir cómo van a ser los elementos que se van a almacenar. En la imagen de la derecha tienes un ejemplo gráfico de definición de elementos para ambos casos.



Obviamente en las bases de datos relacionales, lo que hay que definir es cómo son las tablas, que es donde se almacenarán los datos. De forma equivalente, en las BDOO hay que definir cómo son los objetos que se van a almacenar. En ambos casos, se utilizará un lenguaje propio de la base de datos (los cuales suelen ser bastante diferentes a los lenguajes de programación de uso general). Usando el lenguaje de la base de datos especificaremos qué atributos van a tener todos los objetos de un determinado tipo y qué métodos implementan.

Una vez que ya hemos definido cómo van a ser los objetos a almacenar, podremos introducir objetos en la base de datos. Esto da lugar a que en la base de datos tengamos:

- ✓ La **definición de los objetos** a almacenar. Dicha de definición se suele denominar “clase” o “tipo de dato objeto”, dependiendo de la base de datos. Normalmente, la expresión “clase” se utiliza en SGBDOO y la expresión “tipo de dato objeto” en SGBDOR.
- ✓ Y un **conjunto de instancias**. Una instancia es un objeto con datos reales, es decir, se da un valor real a cada atributo de la clase. A una instancia también se le suele llamar “ejemplar”, o simplemente “objeto”.



Diferenciamos aquí por tanto lo que es la definición de objeto, de lo que sería un objeto concreto, es decir, su instancia o ejemplar. El SGBDOO proveerá de mecanismos tanto para definir los objetos, como para insertar, buscar, modificar y borrar objetos de la base de datos.

Además de lo anterior, como puedes imaginar, los SGBDOO y los SGBDOR permite la ejecución de métodos. Piensa en el ejemplo expuesto en las imágenes anteriores, en el que hay un conjunto de instancias de una clase llamada **Empleado**. En dichas instancias, el método **Edad** calcularía la edad del empleado partiendo de su fecha de nacimiento. En estas bases de datos podríamos buscar una instancia concreta dentro de todas las instancias, partiendo por ejemplo del DNI del empleado, y después ejecutar el método **Edad** para la instancia buscada.

Una de las cosas que más trabajo cuesta comprender es saber dónde se almacenan y ejecutan los métodos. Tienes que tener claro que los métodos se almacenan en el SGBD y no en la aplicación que usa los datos. Es el SGBD el que se encargará de ejecutar los métodos, a petición de la aplicación que hace uso de los objetos almacenados.

Los métodos se programan en lenguajes de alto nivel. Algunos ejemplos de estos lenguajes son PL/SQL, usado generalmente en los SGBDOR, Java u ObjectScript. No todas las bases de datos usan los mismos lenguajes.

Para saber más

En el siguiente enlace, perteneciente a la organización ODBMS, podrás encontrar más información sobre las bases de datos orientadas a objetos y sobre otras tecnologías denominadas bases de datos NoSQL.

[ODBMS.](#)



Autoevaluación

¿Cuál de las siguientes frases es verdadera?

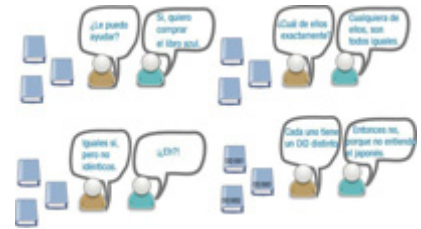
- ☐ En los sistemas gestores de bases de datos relacionales no se pueden definir objetos.
- ☐ Los objetos en los SGBDOO se definen en lenguajes de uso general como Java.

1.2.- Características de las BDOO. (I)

Las características de las BDOO son muchas y difíciles de resumir, por lo que aquí solo vamos a destacar algunas de las características más importantes. ¿Sabes cuál es la principal característica? La primera característica que tienes que recordar es que **las BDOO son sistemas orientados a objetos**, y todos los sistemas orientados a objetos tienen características comunes entre sí.

En este apartado vamos a abordar las características que tienen las BDOO, por el hecho de ser sistemas orientados a objetos. Seguro que muchos de estos conceptos ya te suenan:

- ✓ **Encapsulamiento y ocultación.** Encapsular atributos y métodos en la misma entidad, ocultando detalles de implementación. A los objetos almacenados los llamamos instancias, ejemplares o simplemente objetos, y al molde que siguen todos los objetos del mismo tipo lo llamamos “clase” (SGBDOO) o “tipo de datos objeto” (SGBDOR). De ahora en adelante utilizaremos el término “clase” para referirnos a cualquiera de los dos casos anteriores.
- ✓ **Clases y tablas de objetos.** En los SGBDOO, de cara a la programación, una clase es también un contenedor que almacena todos los objetos del mismo tipo, es decir, es un contenedor para todas las instancias o ejemplares que siguen el mismo molde. En los SGBDOR se le suele denominar “tabla de objetos” y almacena ejemplares de un tipo de dato objeto.
- ✓ **Estado.** El estado de una instancia o ejemplar esta formado por el valor de sus atributos.
- ✓ **Identidad única e inmutable de los objetos (OID).** En los SGBDOO, todos los objetos suelen tener asociado un identificador de objeto, llamado OID (en inglés, OID), que permite diferenciar a dos instancias aunque tengan el mismo estado, es decir, aunque tengan todos los atributos con el mismo valor. Decimos que **dos objetos son idénticos si tienen el mismo OID y son iguales si tienen el mismo estado**.
- ✓ **Relaciones entre objetos.** En los SGBDOO los objetos pueden relacionarse en diversas formas. Una de esas formas es a través de relaciones de herencia, donde un objeto se construye a partir de otro objeto ya existente (ampliando, limitando o concretando la funcionalidad del primero). Otras formas de relación entre objetos son las [asociaciones](#), [agregaciones](#) y [composiciones](#), donde un objeto tiene una dependencia de algún tipo sobre otros objetos.
- ✓ **Extensibilidad.** Los atributos de un objeto pueden ser de dos tipos: tipos de datos predefinidos o simples (como números enteros, cadenas de texto, etc.) o tipos de datos creados por nosotros. Los tipos de datos creados por nosotros son a su vez clases nuevas de nuestra base de datos, pudiendo usar dichas clases como tipo para los atributos.
- ✓ **Sobrecarga y anulación (polimorfismo).** Al igual que ocurre en Java, en muchas BDOO es posible hacer que un mismo método tenga comportamientos diferentes dependiendo del contexto. Existen dos tipos de polimorfismos: de [anulación](#) y de [sobrecarga](#).



Autoevaluación

¿Cuál de las siguientes afirmaciones sobre BDOO es falsa?

- ☐ Si dos ejemplares tienen el mismo OID son iguales.
- ☐ Los atributos de un objeto pueden tipos de datos simples o clases definidas por nosotros.
- ☐ Una tabla de objetos es un lugar donde se almacenan ejemplares de un mismo tipo de objeto.
- ☐ La relación de herencia permite construir objetos a partir de uno ya existente.

1.2.1.- Características de las BDOO. (II)

En el apartado anterior se analizaron las características de las BDOO desde la perspectiva de los sistemas orientados a objetos, y en este apartado, se van a analizar las características de las BDOO como bases de datos que son. Lo primero que tienes que tener en cuenta es que **las BDOO son bases de datos**, y como tales, también tienen características comunes a todas las bases de datos. Algunas de estas características son:

- ✓ **Persistencia de los datos.** El concepto de persistencia ya fue abordado antes, y como recordarás, cuando se habla de persistencia la idea es hacer que los datos estén disponibles después de cerrar la aplicación o de apagar el ordenador. **Las bases de datos son sistemas donde los datos son persistentes y solo se borrarán cuando se le pida a la base de datos que los borre.**
- ✓ **Reducen la impedancia por desajuste de modelos** (en inglés, impedance mismatch). La forma en la que se manejan los datos en la aplicación y la forma en la que se manejan los datos en la base de datos son, por lo general, bastante diferentes. Cuando una aplicación necesita almacenar datos, debe ajustar sus datos para que puedan ser almacenados en la base de datos. Si utilizamos un lenguaje orientado a objetos, como es Java, y una base de datos con un modelo muy diferente, como pueden ser el relacional, la impedancia será alta, pues habrá que adaptar los datos del lenguaje orientado a objetos al modelo relacional. En cambio, si en vez de usar el una base de datos relacional, usamos una base de datos orientada a objetos, la impedancia será menor, porque en ambas partes se utilizará el modelo orientado a objetos.
- ✓ **Lenguajes de consulta declarativos.** Prácticamente todos los SGBD, incluidos los SGBDOO, permiten manipular la información almacenada usando un [lenguaje declarativo](#).
- ✓ **Acceso concurrente y fiable a los datos.** El acceso concurrente y fiable permite que varias aplicaciones puedan acceder simultáneamente a la base de datos, sin que ello provoque incongruencias en los datos. No todas las bases de datos permiten accesos concurrentes, pero si la gran mayoría.
- ✓ **Gestión del almacenamiento secundario.** Gestionar dónde y cómo se almacenan los datos de forma física (en disco u otro soporte), cómo se organizan para dar un mejor rendimiento o permitir que existan varias replicas de la información, son capacidades típicas de las bases de datos actuales.
- ✓ **Seguridad.** Los datos de una base de datos son información importante que hay que preservar tanto de fisgones, como de modificaciones no autorizadas. Casi todos los SGBD actuales permiten configurar diferentes niveles de acceso, con protección de usuario y contraseña.
- ✓ **Interfaces de programación de aplicaciones (API).** A través de la API, las aplicaciones de usuario pueden acceder a la base de datos para guardar o rescatar los datos. Todos los SGBD tienen una API que permite hacer programas que usan la base de datos.
- ✓ **Utilidades de mantenimiento y optimización.** Casi todos los SGBD actuales incluyen herramientas de mantenimiento y optimización que facilitan el trabajo de los administradores de bases de datos. Imagínate que tienes que hacer una copia de seguridad de los datos de tu base de datos, pues para facilitar esa tarea, y otras del estilo, los SGBD suelen incluir utilidades de mantenimiento.



Autoevaluación

Señala, de las siguientes características, cuales crees que son imprescindibles para que un SGBDOO pueda ser utilizado en una aplicación realizada por ti:

- ☐ Debe almacenar los datos de forma persistente.
- ☐ Debe permitir el acceso concurrente a los datos.
- ☐ Debe tener un API para la programación de aplicaciones.
- ☐ Debe tener un lenguaje de consulta declarativo.

Mostrar Información

Para saber más

En el siguiente enlace podrás encontrar más información sobre los lenguajes declarativos:

[Artículo de la wikipedia sobre los lenguajes de programación declarativos.](#)

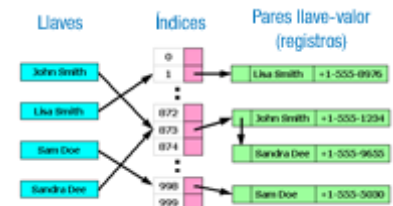
1.3.- Almacenamiento de la información.

¿Dónde y cómo se almacena la información en un SGBD? La forma de almacenar la información es similar (pero no igual) entre los diferentes SGBD. Normalmente los modelos objeto-relacionales y orientados a objetos utilizan técnicas más complejas que los relacionales.

El aspecto más visible del almacenamiento de datos en un SGBD es lo que se denomina Gestión del Almacenamiento Secundario. El almacenamiento primario sería la memoria principal del sistema (la cual se borra al apagar el ordenador), y el secundario correspondería a dispositivos de almacenamiento persistentes (disco duro, por ejemplo). Veamos algunas características de los SGBD en lo que a gestión del almacenamiento secundario se refiere:

- ✓ **SGBD Local.** Cuando tenemos un SGBD local, los datos se almacenan en el mismo ordenador donde se ejecuta la aplicación.
- ✓ **SGBD Remoto.** Cuando tenemos un SGBD remoto, los datos se almacenan en un ordenador diferente al ordenador donde se ejecuta la aplicación. Al sistema donde se ejecuta el SGBD se le denomina servidor.

- ✓ **Gestión de índices.** ¿Alguna vez has consultado el índice de un libro? Los índices permiten acceder a la información de forma rápida sin tener que revisar todas las páginas del libro una tras otra. En una base de datos los índices actúan de forma similar, son estructuras complementarias de la base de datos que facilitan la búsqueda de datos concretos dentro de la base de datos. La utilización de índices mejora el rendimiento de la base de datos notablemente. Cada base de datos, dependiendo del modelo implementado, y de sus peculiaridades, tiene una forma diferente de trabajar con los índices, pero en general se puede hablar de los siguientes tipos de índices:



- **Índice primario (primary index).** Imagina que tienes que diseñar una clase para almacenar los coches arreglados en un taller, ¿podrías decir que dato de cada coche sería el más importante? Seguro que tu respuesta ha sido la matrícula. ¿Y sabes por qué es tan importante? Porque la matrícula es única para cada coche, y permite diferenciar un coche de cualquier otro. Con los datos de la matrícula podría crearse lo que denominamos un índice primario, pues determina de forma unívoca a cada coche y permite organizar la información internamente en la base de datos con cierto orden. Cualquier atributo cuyo valor sea único para cada objeto puede usarse para construir un índice primario.
- **Índice secundario (secondary index).** Los índices primarios son siempre únicos (los valores no se repiten), pero los secundarios no tienen porque serlo. Si en la aplicación la forma más habitual de buscar los coches es a través de la marca, para mejorar la búsqueda por marca podríamos crear un índice secundario.
- **Índice compuesto (compound index).** Los índices compuestos son índices sobre más de un atributo que se crean concatenando sus valores, pueden ser **únicos** o **no únicos**. Si en la aplicación, la forma más habitual de buscar coches es a través de la marca y el modelo, podríamos crear un índice compuesto para tal fin.
- ✓ **Data Clustering y replicación de datos.** Son técnicas destinadas a mejorar el acceso a datos y a protegerlos de imprevistos. Por ejemplo, imagina que el equipo donde está instalada la base de datos se estropea. Pues manteniendo una replica de los datos en otro ordenador, podríamos seguir usando la base de datos. En este tipo de técnicas los datos están distribuidos en diferentes sistemas que almacenan y procesan la información de forma independiente, dando más rapidez y eficiencia a la base de datos.

Llave o clave primaria es un concepto muy importante en los SGBD relacionales. La clave primaria está formada por una o varias columnas que siempre tendrán valor y que de forma conjunta identificarán de forma unívoca a cada fila de la tabla. Toda tabla del modelo relacional debe tener una clave primaria. La clave primaria es automáticamente un índice primario (si está formada solo por una columna) o un índice compuesto único (si está formada por dos o más columnas)



Autoevaluación

Los índices compuestos determinan siempre de forma unívoca cada objeto, dado que es un índice primario y secundario a la vez. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

2.- Instalación de un SGDBOO.

Caso práctico



Antonio se ha informado y ya conoce un poco más que es la persistencia y como influyen las bases de datos en todo ese meollo. Después de investigar e investigar, ha pensado que va a optar por los SGBDOO para añadir persistencia a su aplicación.

Esto supone todo un reto para él, pues él ya conocía los SGBD relacionales, pero jamás se había enfrentado a un SGBDOO. Ha preguntado a Juan y a Ana, en busca de consejo. No sabe que SGBDOO usar.

¿Qué SGBDOO usarías tú? En esta sección vamos a abordar la instalación de un SGBD con soporte para almacenar objetos. Como ya sabes, para almacenar objetos se pueden usar tanto bases de datos objeto-relacionales como orientadas a objetos. Aquí vamos a explorar la base de datos Intersystems Caché, la cual es orientada a objetos, aunque haremos referencias continuas a Oracle, la cual es objeto-relacional. Pondremos ejemplos para ambas bases de datos, por lo que si quieres puedes instalar las dos.

Intersystems tiene una versión gratuita de su producto Caché, que puedes utilizar para esta unidad. Caché es un completo SGBDOO con muchas más funciones de las que nosotros vamos a utilizar realmente. La página web del producto es la siguiente, en ella podrás descargar Caché de forma gratuita:



[Página web de Intersystems Caché en Español.](#)

Para descargar Caché primero tienes que registrarte, después podrás descargar la base de datos e instalarla sin problemas. Instalar esta base de datos es relativamente sencillo. Veamos los pasos de instalación en el sistema operativo Windows. El proceso se puede resumir como sigue:

- ✓ **Descargar el instalador y ejecutarlo.**
- ✓ **Aceptar los términos de licencia.** La licencia nos permite usar esta base de datos para evaluación.
- ✓ **Seleccionar la carpeta donde se instalará.** Por defecto se instalará en "C:\Intersystems\Cache", es conveniente apuntar la carpeta donde se va a instalar, pues de ahí tendremos que coger algunos archivos importantes.
- ✓ **Instalación.** Suele durar entre 15 y 20 minutos aproximadamente, dependiendo de las características de tu ordenador. Durante este proceso Caché copia los archivos y se configura.
- ✓ **Mostrar la página de preparación.** Después de la instalación, dará la opción de mostrar una página con documentación para empezar rápidamente a usar Caché, toda esta documentación y mucha más, viene instalada con caché. El único problema es que la documentación viene en inglés. Para hacer uso de esta documentación necesitas un navegador web y tener iniciado Caché.
- ✓ **Después de la instalación,** podremos encontrar herramientas en el menú de inicio para arrancar y parar Caché, así como para acceder a la documentación (a través del navegador web), acceder a la herramienta de administración (también a través de la web) y acceder a la herramienta Caché Studio que usaremos más adelante.

Debes conocer

En la siguiente animación puedes ver con más detalle el proceso de instalación para el sistema operativo Windows.

[Resumen textual alternativo](#)

En el siguiente enlace se describe el proceso de instalación de InterSystems Caché para sistemas operativos basados en Linux y Unix:

[Instalación de InterSystems Caché en Linux y Unix.](#)

Si deseas instalar Oracle y probar los ejemplos propuestos para dicha base de datos, a continuación encontrarás el enlace oficial para descargar la base de datos Oracle XE (versión Express Edition de la base de datos Oracle). La versión de Oracle más sencilla de utilizar es la XE.

[Web de descarga de Oracle XE 11g.](#)

En los siguientes enlaces vienen instrucciones detalladas para instalar y probar Oracle XE. Si decides instalarla, no olvides apuntar la contraseña que del usuario "SYSTEM" (te la pedirá en el proceso de instalación):

[Tutorial de instalación de Oracle XE 11g en español para el sistema operativo Windows.](#)

[Tutorial de instalación de Oracle XE 11g en inglés para Linux.](#)

[Tutorial de instalación de Oracle XE 11g en inglés para Windows.](#)



3.- Primeros pasos con el SGDBOO. (I)

Caso práctico



Después de darle muchas vueltas, **Antonio** se ha decidido por probar uno de los muchos SGDBOO existentes. Guiado por los consejos de sus compañeros y compañeras, especialmente por los consejos de Juan, se ha atrevido a experimentar con la base de datos Intersystems Caché. La ha instalado, pero ahora no sabe muy bien por donde empezar. Se acerca a Juan y le pide ayuda, pero Juan está tremendamente atareado y no le puede ayudar justo en ese momento, por lo que decide empezar a investigar por su cuenta. “¿Qué es lo primero que tengo que hacer?”, se pregunta. Empezar es siempre lo más difícil.

¿Y por dónde empiezo? Los SGDBOO modernos incorporan una gran cantidad de opciones para su administración y utilización en fases de desarrollo (mientras se está desarrollando una aplicación) y de producción (cuando la aplicación ya desarrollada se está usando plenamente). Aquí nos vamos a centrar en la parte de desarrollo, es decir, creación y utilización de bases de bases de datos orientadas a objetos en un proyecto software.

Hoy día, las bases de datos orientadas a objetos disponen de varios tipos de interfaces, de fácil uso, para administrar el servidor y poder hacer todas las tareas necesarias. Veamos lo que obtenemos tras una instalación de InterSystems Caché:



- ✓ **Portal de Gestión del Sistema.** Es un servidor web que dispone de una potente herramienta de configuración del SGDB a través de la web. La URL de acceso a dicha herramienta es normalmente “http://localhost:57772”, siempre que estés usando el mismo ordenador en el que has instalado la base de datos.
- ✓ **Herramientas para iniciar y detener el SGDB.** Si has optado por desactivar el inicio automático de Caché, explicado en la animación del apartado anterior, no olvides iniciar la base de datos antes de ponerte a trabajar.
- ✓ **Documentación.** Caché incorpora una extensa documentación, accesible a través de web. La URL de acceso a la documentación, siempre que accedas desde el mismo ordenador en el que has instalado la base de datos, es “http://localhost:57772/csp/docbook/DocBook.UI.HomePageZen.cls”. Obviamente tienes que tener Caché arrancado para que funcione.
- ✓ **Caché Studio.** Studio es una potente herramienta que permite diseñar cómodamente tu base de datos.

No obstante, en una instalación típica de Caché, y dependiendo del sistema operativo usado, podrás encontrar en el menú de inicio un acceso directo a las herramientas anteriores.

Otros servidores de bases de datos, como por ejemplo Oracle XE, tienen también una avanzada interfaz web, y obviamente muchas de las características antes comentadas. Para Oracle XE existe la herramienta SQL Developer, equivalente a la herramienta Caché Studio.

Debes conocer

En la siguiente animación se muestra como crear un usuario nuevo en Caché, lo cual nos permitirá acceder a la base de datos desde Java y desde la herramienta Studio, usando dicho usuario. Si aún así no consigues crear el usuario, siempre puedes usar el usuario “_SYSTEM” con contraseña “SYS”. Dicho usuario se crea al instalar Caché.

[Resumen textual alternativo](#)

Para saber más

En el siguiente enlace puedes encontrar una completa guía para comenzar a usar Oracle XE desde cero:

[Guía sobre como empezar a usar Oracle XE.](#)

3.1.- Primeros pasos con el SGDBOO. (II)

En Caché usaremos principalmente la herramienta Caché Studio, cuyo uso se describe en apartados posteriores. No obstante, dentro de la interfaz web de Caché podemos encontrar una cosa que nos será muy útil. Se trata de una pequeña interfaz web destinada a la ejecución de sentencias SQL. Las sentencias SQL de Caché las veremos más adelante, pero de momento, veamos como acceder a esa pequeña interfaz web. Esto nos va a servir para familiarizarnos con el entorno de trabajo de Intersystems Caché. Para acceder a dicha interfaz, procedemos de la siguiente forma:

1. Accedemos al portal de gestión de Caché y hacemos clic en el enlace titulado SQL de la sección “Gestión de datos”. Eso cambiará la interfaz de modo que se mostrarán más opciones sobre SQL.
2. Después seleccionamos el espacio de nombres en el que vamos a trabajar. Los espacios de nombres aparecerán a la izquierda, y no te preocupes, más adelante se explica que es un espacio de nombres.
3. Después hacemos clic en la opción “Ejecutar sentencia SQL”, lo cual volverá a cambiar la interfaz de trabajo, mostrando ahora un área donde escribir las sentencias SQL y un botón para ejecutar las sentencias.
4. En el área de texto se escribirían las sentencias SQL.
5. Y una vez escritas se ejecutarían a través del botón “Ejecutar consulta”.

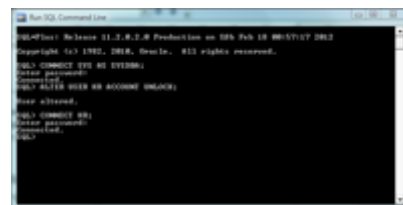


En esta unidad, como se dijo antes, se muestran ejemplos tanto para Caché como para Oracle. Así podrás comparar una base de datos orientada a objetos, y una base de datos objeto-relacional. Para poder probar los ejemplos propuestos en Oracle, te recomendamos usar Oracle XE. Oracle XE crea una base de datos básica, pero suficiente para probar todo lo que se comenta en los contenidos de esta unidad.

Y para manejar Oracle XE te recomendamos SQL*Plus. SQL*Plus aparecerá en el menú de aplicaciones como “Run SQL Command Line” en el caso de sistemas operativos como Windows. En el caso de sistemas basados en Linux, después de instalar Oracle XE dispondrás del comando “sqlplus” ejecutable desde un terminal.

Oracle XE viene con un usuario precreado, el usuario “HR”, el cual viene bloqueado por defecto. Para desbloquearlo y poder utilizarlo para probar los ejemplos aquí propuestos, tienes que seguir un pequeño proceso:

1. Inicia SQL*Plus y accede a la base de datos con el usuario SYS o el usuario SYSTEM, usados para administrar el sistema. En el proceso de instalación, Oracle XE te solicita que indiques la contraseña de dichos usuarios (la misma para ambos), y ahora es el momento de usarla, teclea lo siguiente e introduce la contraseña para conectarte:
CONNECT SYS AS SYSDBA;
2. Después, desbloquea el usuario “HR” o ponle una contraseña. Para ello, debes alterar el estado del usuario, ejecutando una de las siguientes sentencias, la primera para desbloquear la cuenta, o la segunda para agregar una contraseña al usuario (sustituye XXX por la contraseña deseada):
Opción a) **ALTER USER HR ACCOUNT UNLOCK;**
Opción b) **ALTER USER HR IDENTIFIED BY XXX;**
3. Comprueba que puedes conectarte a la base de datos usando el usuario HR. Para conectarte usa el comando “CONNECT” seguido del nombre de usuario:
CONNECT HR



Autoevaluación

¿Cuál de las siguientes herramientas es una herramienta usada ejecutar sentencias SQL en Oracle?

- ☐ HR.
- ☐ SYSDBA.

- ☐ SQL*PLUS.
- ☐ Studio.



4.- Creación de la base de datos.

Caso práctico

Juan se levanta después de un buen rato trabajando. Quiere dar una vuelta y despejarse, sabe que no es bueno tirarse muchas horas sentado delante del ordenador sin hacer estiramientos y fijando continuamente la vista en la pantalla, por lo que decide ir a tomar un poco de agua. **Antonio**, que lo ve, y que está un poco desesperado, decide ir a preguntarle acerca de la base de datos que está probando:

—Hola Juan —dice Antonio—, ¿tienes un minuto para explicarme cómo funciona la base de datos Caché que me recomendaste?

—Pues la verdad es que no, estoy muy atareado —responde Juan—. Si tan difícil te resulta, utiliza otro SGDB que conozcas, aunque sea relacional. Oracle, por ejemplo, ¿lo conoces?

—Sí, pero la verdad es que quiero aprender cosas nuevas que me abran el mercado laboral, por lo que estoy interesado en aprender a usar Caché.

—Vale, dado que estás tan interesado, podemos quedar esta tarde, después de que termine unas cuantas cosas urgentes que tengo pendientes. ¿De acuerdo?



¿Y cómo creo una nueva base de datos? De cara al SGBD, una base de datos será una estructura de disco que nos permitirá almacenar instancias de objetos de diferente tipo. Crear una base de datos nueva en el SGBD Caché, almacenada en tu disco duro local, se hace a través de la interfaz web de administración. El proceso es sencillo, y se resume en los siguientes pasos:

- ✓ **Creación de la base de datos.** Al crear la base de datos crearemos las estructuras internas necesarias y le asociaremos un directorio de disco, lugar donde se almacenarán las instancias de los objetos.
- ✓ **Creación de un espacio de nombres o namespace.** Un espacio de nombres es un espacio de trabajo lógico, que vincula una o varias estructuras de almacenamiento (bases de datos del punto anterior) con una o varias localizaciones (diferentes directorios o servidores), de tal forma que se puede cambiar por ejemplo el directorio donde están los datos de una base de datos, o hacer réplicas para garantizar el acceso a los datos.



De cara a trabajar con la base de datos Caché, lo que usaremos es el namespace o espacio de nombres. Por ejemplo, si tenemos que conectar con Caché desde Java, usaremos el espacio de nombres para indicar donde están guardados nuestros objetos. Y si queremos diseñar nuestra base de datos a través de la herramienta Studio, tendremos que indicar cual es el espacio de nombres donde se alojarán nuestro datos.

Debes conocer

En la siguiente animación se explica la forma de crear una base de datos nueva y su correspondiente espacio de nombres.

[Resumen textual alternativo](#)

Si no consigues crear la base de datos y el espacio de nombres, no te preocupes, puedes usar el espacio de nombres "USER", que encontrarás creado y listo para ser usado después de instalar Caché. Este espacio de nombres, junto con el usuario "_SYSTEM" con contraseña "SYS" te permitirán empezar a usar Caché sin configurarlo.

Y si has decidido probar también con Oracle XE, no tienes que preocuparte por la creación de la base de datos, dado que Oracle XE va con una base de datos creada por defecto, cuya instancia se llama "XE". No es así en otras versiones de Oracle, la versión eXpress Edition es una versión limitada para la realización de pruebas y aprendizaje en la que muchas cosas ya van preconfiguradas.



Autoevaluación

En Caché una base de datos es un espacio lógico que vincula uno o varios espacios de nombres. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

5.- Esquema de la base de datos.

Caso práctico

Antonio ha quedado con **Juan** para que le explique el funcionamiento de la base de datos Caché, y no quiere llegar tarde a la cita. Lleva su portátil, con la base de datos recién instalada, para que le explique cómo funciona, y una libreta para tomar anotaciones. **Juan** aparece justo a tiempo y se ponen de lleno a trabajar. Después de un rato, **Juan** hace el siguiente comentario:

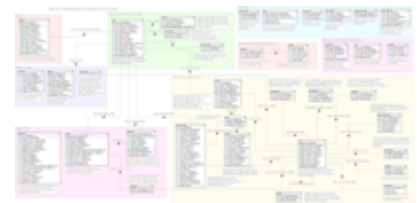


—**Antonio**, ¿qué te pasa? Te veo un poco perdido, ¿quieres que te repita algo? —dice **Juan**.

—No, que va. Me estoy enterando de todo perfectamente, no te preocupes. Lo que pasa es que estoy intentando comprender cómo se crearía el esquema de la base de datos y me estoy liando un poco. A ver, dices que tendría que crear una clase para poder almacenar objetos, y la pregunta es, ¿cuándo creo el esquema de la base de datos? ¿Antes o después? —responde Antonio.

—Me parece que no me has entendido. Definir una clase en Caché es parte de la creación del esquema de la base de datos. Crear el esquema consiste en crear las clases que permitirán almacenar información. Si no creas las clases, Caché no sabrá cómo son los objetos a almacenar.

¿Qué es el esquema de la base de datos? Antes de poder almacenar datos hay que decirle al SGBDOO como son los datos que se van a almacenar en la base de datos, incluyendo las relaciones entre los diferentes objetos que se almacenan. Para esta tarea, el SGBD dispone de un lenguaje específico, llamado lenguaje de definición de objetos (ODL), que nos permite definir como serán los datos a almacenar. El lenguaje de definición de objetos, por tanto, nos permite definir el esquema de la base de datos.



Para los SGBDOO, el lenguaje de definición de objetos varía de una base de datos a otra. Por desgracia, aunque ha habido intentos de estandarizar este tipo de lenguajes por parte de la organización ODMG (de forma que en todos los SGBDOO sea similar), no se ha llegado buen puerto. Hay pocos SGBDOO que cumplan con las especificaciones propuestas por la desaparecida ODMG. Caché usa un ODL que en cierta manera se parece al ODL de ODMG, aunque a pesar de todo, sigue siendo bastante diferente.

Para los SGBDOR, el lenguaje de definición de objetos suele ser básicamente el lenguaje estándar SQL. Es el caso de Oracle y otras bases de datos. A groso modo podríamos decir que el lenguaje SQL está compuesto de varios sublenguajes: uno para definir datos y objetos (equivalente a un ODL), otro para consulta de datos, y otro para manipulación de datos. Aquí nos referimos al primero de los tres, los otros dos los veremos más adelante.

En el caso de otras bases de datos, como DB4O, no es necesario usar un lenguaje diferente para crear el esquema de la base de datos. DB4O almacena directamente objetos Java en la base de datos, por lo que solo hay que preocuparse de crear las clases en Java, dado que después se pueden almacenar las instancias directamente en la base de datos. Interesante, ¿verdad?

Pero como nuestro foco está puesto en InterSystems Caché, veremos cómo es el ODL de dicha base de datos, comparándolo con el ODL de Oracle (SQL al fin y al cabo), para que así no pierdas la perspectiva de los sistemas objeto-relacionales. De momento, hagamos un pequeño acercamiento a la herramienta Caché Studio, que nos permitirá crear nuestro esquema de una forma sencilla y rápida.

Debes conocer

En la siguiente animación se hace un pequeño acercamiento a la herramienta Studio.

[Resumen textual alternativo](#)



Autoevaluación

A continuación, se muestran varias afirmaciones sobre los esquemas de las bases de datos. Marca aquellas que consideres falsas:

- ☐ Al lenguaje que nos permite definir el esquema de la base de datos en Caché lo denominamos DB4O.
- ☐ Todas las bases de datos tienen un ODL diferente, ya sean relacionales u orientadas a objetos.
- ☐ Definir las clases forma parte de la creación del modelo de base de datos.
- ☐ En Oracle, no es necesario definir el esquema de la base de datos, dado que se almacenan instancias de clases Java directamente.

[Mostrar Información](#)

5.1.- El lenguaje de definición de objetos. (I)

¿Y cómo son los lenguajes de definición de objetos de las diferentes bases de datos? Como se comentó en el apartado anterior existen diferentes ODL para el ámbito de los SGBDOO, y cada uno es, cuanto menos, ligeramente diferente del resto. El ODL de Caché permite crear clases persistentes con un lenguaje propio.

Veamos como sería una clase de ejemplo usando el ODL de Caché, un primer acercamiento:



```
Class User.Conductor Extends %Persistent
{
  // Nombre del conductor o de la conductora, tipo cadena de texto.
  Property Nombre As %String;
  // Apellidos del conductor o de la conductora, tipo cadena de texto.
  Property Apellidos As %String;
  // Fecha de nacimiento del conductor, tipo fecha, formato DD/MM/AAAA.
  Property FechaNacimiento As %Date (FORMAT = 4);

  // Calcula la edad y retorna un número entero.
  Method edad () as %Integer
  {
    Set e = ($PIECE($NOW(),"",1)-$PIECE(..FechaNacimiento,"",1))/365
    Quit e
  }
}
```

Veamos qué se puede sacar en conclusión después de ver el código anterior:

- ✓ Se usa la palabra reservada `Class` para comenzar a declarar una clase, seguida del nombre de la clase.
- ✓ El nombre de la clase es `"User.Conductor"` que incluiría el paquete (`"User"`) donde está la clase (`"Conductor"`).
- ✓ La clase `Conductor` extiende (a través del termino reservado `extends`, como en Java) la clase `%Persistent`, la cual es una clase de sistema (al igual que `%String` o `%Date`). Las clases de sistema llevan el `"%"` delante, y se pueden escribir como `"%Library.Persistent"` o abreviadamente como `"%Persistent"`. Extender la clase `%Persistent` hace que las instancias de dichas clases puedan almacenarse, es decir, que puedan ser persistentes.
- ✓ Las propiedades o atributos de la clase se definen usando la palabra reservada `Property`, seguida del nombre del atributo, la palabra reservada `as` e indicando después el tipo. En el ejemplo hay tres variables, dos de tipo cadena de caracteres (`%String`) y una tipo fecha (`%Date`).
- ✓ Se declara un método, el método `edad()`, que calcula la edad del conductor o de la conductora. El método está construido en un lenguaje muy diferente a Java, se trata de `ObjectScript`, el principal lenguaje que usa Caché (puede usar, como veremos, varios lenguajes para la programación de métodos). Si decides probar el ejemplo anterior, ten cuidado con los espacios, `ObjectScript` es muy caprichoso con los espacios. En `ObjectScript` cualquier sentencia debe llevar como mínimo un espacio delante, salvo contadas excepciones, y procura dejar solo un único espacio entre elementos del lenguaje, hay situaciones en las que dejar más de un espacio da error (es el caso de los comandos).

Como ves, el ODL de Caché es moderadamente complejo. Se parece a Java en algunos, más bien pocos, aspectos, pero claramente es un lenguaje orientado a objetos. Una vez descrita la clase en el ODL de Caché, hay que compilarla para comprobar que no tiene errores y para insertarla en la base de datos, operaciones que se pueden hacer desde la herramienta Studio. Una vez creada y compilada la clase, ya podemos almacenar instancias de dicha clase en Caché.

Para saber más

En el siguiente enlace puedes acceder a la guía de orientación para programación en Caché. Un buen lugar donde comenzar, para aprender como se crean las clases de Caché, y como se programan los métodos, entre otras muchas cosas.

[Guía de orientación para la programación en Caché.](#)

5.1.1.- El lenguaje de definición de objetos. (II)

Y, ¿cómo es el lenguaje de definición de objetos de Oracle? Para empezar, podemos decir que en Oracle no existe el concepto de clase, se sustituye en su lugar por el concepto de tipo de dato objeto, el cual volveremos a explicar más adelante, de momento diremos que en Oracle se crean tipos de dato objeto.



Verás que el ODL de Oracle es muy diferente al de Caché. Lo cual es en algunos aspectos negativo y en otros bastante positivo. Negativo porque obliga a aprender un lenguaje nuevo, pero positivo porque si ya sabes SQL, te costará poco trabajo aprenderlo. Veamos un ejemplo, el mismo ejemplo mostrado para Caché vamos recrearlo para Oracle:

```
CREATE TYPE tipo_conductor AS OBJECT (  
    nombre VARCHAR(20), apellidos VARCHAR(20), fecha_nacimiento DATE,  
    MEMBER FUNCTION edad RETURN INTEGER  
);  
/
```

En el ejemplo anterior se crea simplemente un tipo de dato objeto nuevo, llamado **tipo_conductor** que tiene tres variables y un método. Las variables son por un lado, **nombre** y **apellidos**, que son dos cadenas de texto de longitud máxima 20 caracteres ("**VARCHAR2(20)**"), y por otro **fecha_nacimiento** que será un dato de tipo fecha ("**DATE**"), y que como es obvio contendrá una fecha de nacimiento.

Fíjate en la barra que aparece al final ("/"), es necesario ponerla cuando usamos SQL*Plus para crear un nuevo tipo de dato objeto, y se pone en una nueva línea, sin nada delante. Si usas otra interfaz para acceder a Oracle (interfaz web por ejemplo), no es necesario poner la barra. Tampoco es necesario poner la barra en otro tipo de sentencias que veremos.

El siguiente paso sería definir el método **edad**, es decir, escribir el código del mismo:

```
CREATE TYPE BODY tipo_conductor AS  
    MEMBER FUNCTION edad RETURN INTEGER IS  
        v_edad INTEGER := 0;  
    BEGIN  
        v_edad := FLOOR(MONTHS_BETWEEN(CURRENT_DATE(), fecha_nacimiento)/12);  
        RETURN v_edad;  
    END;  
END;  
/
```

Una vez definido el código de los métodos, entonces pasamos a crear una tabla que permitirá almacenar instancias de dichos objetos. Esto es diferente a Caché, donde este paso no es necesario. En Caché, el hecho de crear y compilar una clase permite almacenar instancias de dicha clase (siempre que la clase no tenga errores), pero en Oracle no:

```
CREATE TABLE conductores AS tipo_conductor;
```

La sentencia usada para crear una tabla que permita almacenar objetos es **"CREATE TABLE"** donde se especifica el nombre de la tabla, seguido de la palabra reservada **"AS"** y el tipo de dato objeto en el que se basará la tabla (en este caso **tipo_conductor**).

Si creas un tipo o una tabla y deseas borrarlos, puedes usar la sentencia **"DROP TABLE"** (para las tablas de objetos) y **"DROP TYPE"** para los tipos de dato objeto. Tendrías que borrar primero la tabla de objetos, después el cuerpo del tipo de dato objeto, y después el tipo, en ese orden. Por ejemplo:

```
DROP TABLE conductores;  
DROP TYPE BODY tipo_conductor;  
DROP TYPE tipo_conductor;
```

Puedes ejecutar estas sentencias, y las anteriores, desde SQL*Plus. Si una tabla o tipo ya existe en Oracle, no te dejará volver a crearlos, tendrás que borrarlos primero.

Para saber más

En el siguiente documento puedes ver cómo es la sintaxis usada para crear objetos en Oracle, eso sí, tendrás que ir al apartado 5, porque los capítulos anteriores hablan de las bases de datos orientadas a objetos.

[Bases de datos orientadas a objetos.](#) (195.52 KB)




Autoevaluación

En Caché, las clases que extienden la clase %Library.Persistent son almacenables en la base de datos. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

5.2.- Tipo de datos básicos y estructurados. (I)

¿Hay tantos tipos de datos en Caché como en Java? Los tipos de datos que se pueden usar en Caché determinan obviamente que datos puede contener cada atributo o que datos puede retornar un método. La respuesta sería casi que sí, podríamos decir que tienen prácticamente los mismos tipos de datos. En Caché una propiedad o atributo puede ser:



```
// Nombre del conductor, tipo de dato de texto.  
Property Nombre As String Required {}  
// Apellido del conductor, tipo de dato de texto.  
Property Apellido As String Required {}  
// Fecha de nacimiento del conductor, fecha, formato DD/MM/AAAA.  
Property FechaNacimiento As DateTime = # {} Required {}
```

- ✓ Un tipo de dato básico (cadena de caracteres, número, etc.).
- ✓ Una instancia de otra clase, es decir, se puede usar otra clase creada como tipo para un atributo.
- ✓ Una colección de objetos.

Existen otros casos, pero nos vamos a ceñir a estos. En Caché los tipos de datos básicos están recogidos como clases de sistema, y siempre les precede el símbolo “%” (como ya se comentó antes). Hay muchos, pero los más importantes son:

- ✓ %Boolean. Valor lógico o boolean, cuyos valores posibles son 0 (falso) o 1 (verdadero).
- ✓ %Date y %Time. Almacenan la fecha y la hora en el formato interno de Caché (formato \$HOROLOG), el cual es sencillo de entender. El formato \$HOROLOG es una cadena que contiene el número de días desde el 31 de diciembre de 1840 y los segundos desde el comienzo del día, separados por una coma, tendría un aspecto como el que sigue: “ddddd,sssss.sssss” (la parte de los segundos va con decimales). Por ejemplo 62200,65310.804992 corresponde con el 19/4/2011 a las 18:08. La parte de los días, sería por tanto la fecha, y la parte de segundos sería la hora.
- ✓ %TimeStamp. Es un sello temporal. El sello temporal se usa generalmente para almacenar un momento exacto en el tiempo, por ejemplo, cuando se produce una venta, dado que contiene tanto la fecha como la hora en formato legible por el ser humano. El sello temporal tiene la forma “YYYY-MM-DD HH:MM:SS” (cuatro dígitos para el año, dos para el mes y dos para el día, separados por guiones, y dos dígitos para la hora, dos para los minutos y dos para los segundos, separados por dos puntos).
- ✓ %Integer que corresponde con número entero.
- ✓ %Numeric que corresponde con un número con decimales almacenado internamente en coma fija. Si queremos almacenar un número donde el número de decimales es fijo (dos decimales como máximo por ejemplo), deberíamos usar este tipo.
- ✓ %Double que corresponde con un número almacenado internamente en coma flotante.
- ✓ %String, que corresponde con una cadena de caracteres.

Los tipos %Boolean, %Integer, %Float, %Double y %Numeric serían tipos de datos simples, dado que almacenan un valor simple, es decir, un único dato. Mientras que los tipos %Date, %Time, %TimeStamp y %String podrían ser considerados como tipos de datos estructurados, puesto que almacenan valores compuestos a su vez de varios datos (una cadena, por ejemplo, almacena varios caracteres). Sin embargo, como se trata de tipos proporcionados de forma nativa por la base de datos, los podemos considerar también como tipos de datos simples.

Entendemos por dato estructurado aquel que nos permiten almacenar datos relacionados juntos (el día, el mes y el año de una fecha, por ejemplo). Si necesitas almacenar un dato compuesto de varias partes, por ejemplo, la dirección de una persona (calle, bloque, número, código postal, etc.) tienes que crear una clase y usar dicha clase como tipo de dato, tal y como se verá en los próximos apartados. También existen otros tipos de datos estructurados, como las colecciones, que veremos también más adelante.

Debes conocer

En el siguiente enlace podrás ampliar información sobre los tipos de datos disponibles en Caché.

[Tipos de datos básicos de Caché.](#)

Y en el siguiente enlace podrás obtener más información acerca del formato usado para fecha y hora en Caché.

[Formato \\$HOROLOG y formas de manejarlo.](#)

5.2.1.- Tipo de datos básicos y estructurados. (II)

¿Y qué tipos de datos hay en Oracle? ¿Tantos como en Caché? La respuesta es que hay muchos, pero son muy diferentes a Java y Caché. En Oracle, un atributo puede ser también un tipo básico, un objeto o una colección de objetos. Veamos a modo comparativo, cuales son los tipos de datos básicos más usados de Oracle:

```
create or replace
TYPE tipo_conductor AS OBJECT
(
  ddi VARCHAR2(10),
  nombre VARCHAR2(20),
  apellidos VARCHAR2(40),
  fecha_nacimiento DATE,
  mails array_mails,
  telefonos tabla_telefonos,
  puntos_carnet INTEGER,
  direccion tipo_direccion,
  MEMBER FUNCTION edad RETURN INTEGER
);
```

- ✓ **CHAR(n)**. Cadenas de caracteres de tamaño fijo, donde n es el número de caracteres.
- ✓ **VARCHAR2(n)** y **VARCHAR(n)**. Cadenas de caracteres de tamaño variable, donde n es el tamaño máximo. VARCHAR es un alias de VARCHAR2, pero se recomienda usar VARCHAR2.
- ✓ **NCHAR(n)**. Equivalente a CHAR(n) pero con codificación **unicode**.
- ✓ **NVARCHAR2(n)**. Equivalente a VARCHAR2 pero con codificación unicode.
- ✓ **NUMBER** o **NUMBER(p,s)**. Permite almacenar un número en coma fija o flotante. Se puede especificar la precisión (p), que será el número de dígitos máximo que tendrá, y la escala (s), que será el número máximo de decimales. Si queremos almacenar un número donde el número de decimales es fijo (dos decimales como máximo por ejemplo), deberíamos usar este tipo.
- ✓ **FLOAT**. Tipo de dato pensado para almacenar un número en coma flotante.
- ✓ **INTEGER**. Tipo de dato pensado para almacenar un número entero.
- ✓ **DATE**. Almacena la hora y la fecha en un formato interno de Oracle. Para convertir una cadena que contiene una fecha al formato interno de Oracle debemos usar la función TO_DATE, y para pasar una fecha a formato cadena (legible por un ser humano) usaremos la función TO_CHAR. Como veremos más adelante, estas funciones están también disponibles en Caché.
- ✓ **TIMESTAMP**. Es un sello de tiempo y es una extensión del tipo DATE. Permite almacenar el momento exacto en el que algo ocurrió. Igual que existe la función TO_DATE para el tipo DATE, para el tipo TIMESTAMP existe la función TO_TIMESTAMP, que permite convertir una fecha en formato legible por el ser humano a un sello temporal almacenado en la base de datos.

Como puedes ver, hay equivalencias entre los tipos de datos de Caché, de Oracle y de Java, pero no hay una correspondencia exacta. Para almacenar un String de Java, podríamos usar un **NVARCHAR2** en Oracle, o un **%String** en Caché, el único inconveniente es que en Oracle hay que especificar el tamaño máximo, lo cual se debe tener en cuenta a la hora de programar nuestra aplicación en Java, haciendo las oportunas comprobaciones.

Para saber más

En el siguiente enlace podrás encontrar más información acerca de los tipos de datos de Oracle.

[Tipos de datos usados en Oracle SQL.](#)



Autoevaluación

¿Cuáles de los siguientes tipos empleados en Oracle y Caché se podrían usar, si fuera necesario, para almacenar un número negativo sin decimales?

- ☐ FLOAT de Oracle.
- ☐ NVARCHAR de Oracle.
- ☐ %Boolean de Caché.
- ☐ %Double de Caché.

Mostrar Información

5.3.- Tipos de datos objeto. (I)

¿Tipo de datos objeto o clases? ¿Son lo mismo? Como ya se comentó con anterioridad, en los SGBDOR, como pasa en Oracle, no existen clases como tal, sino que lo que se usa son los tipos de dato objeto, y esto, ¿qué quiere decir? En los SGBDOR la estructura de almacenamiento lógico utilizada es la tabla, y eso no encaja del todo con lo que sería un modelo de objetos puro.

En SGBD como Oracle la creación de objetos se produce a través de la creación de nuevos tipos de datos, a través de sentencias SQL. Existe un tipo de dato llamado tipo de dato objeto, que es básicamente un tipo de dato compuesto que permite asociar datos y procedimientos. Los objetos serán en este caso un nuevo tipo de dato especial que podrá ser usado de las siguientes formas:

- ✓ Un tipo de dato objeto puede ser usado como tipo para las columnas de una tabla, donde puede haber columnas que no sean de tipo objeto. Esto quiere decir que las instancias de los objetos se almacenarán dentro de una columna en una fila de la tabla. En esta unidad no se abordará esta forma de trabajar con objetos en Oracle.
- ✓ Crear tablas basadas en tipos de datos objeto, es decir, que solo contendrían instancias de un tipo de objeto. Este planteamiento es posiblemente el que más se acerca a las bases de datos orientadas a objetos y es el que abordaremos en esta unidad.



Veamos primero como es la sintaxis usada para crear clases en Caché, aunque ya se introdujo en apartados anteriores, después haremos un acercamiento a Oracle. Para empezar, tienes que tener en cuenta que Caché es extremadamente extenso y tiene muchas opciones (lo mismo que Oracle), aquí nos centraremos en algunas pocas realmente, justo las que necesitamos para hacer los datos de nuestra aplicación mínimamente persistentes. Veamos ahora como sería la estructura general de una clase en Caché:

```
///Descripción de la clase.  
Class User.Conductor extends %Persistent [ClassType = persistent, Final]{  
    ///Comentario.  
    ///Aquí irían las definiciones de métodos y clases.  
}
```

El formato, por lo menos al principio, es simple. Ya se vio en un ejemplo anterior, ahora vamos a profundizar un poco en él. Vamos a analizarlo:

- ✓ La primera línea puede ser una descripción de la clase. Para incluir la descripción se utilizan tres barras inclinadas ("///") y detrás se escribiría la descripción.
- ✓ Justo después de la descripción, debe ir la definición de la clase. Para ello se usa la palabra `Class` (como ya se vio en apartados anteriores). El nombre de la clase ("Conductor") incluye el nombre del paquete ("User"), lo cual se indica poniendo el nombre del paquete seguido de la clase separado por un punto "Paquete.Clase". Un inconveniente, es que en el nombre de la clase no se puede usar el guión bajo (solo caracteres y dígitos numéricos). Bueno, en general, en Caché no se puede usar el guión bajo ("_") para nombres de clases, variables, propiedades, etc.
- ✓ Se pueden poner comentarios, dentro de la clase (fuera no es posible), usando la doble barra ("//"), tal y como se hace en Java. También se puede usar la notación de comentario multilínea de java: "/* Comentario */".
- ✓ En la definición de la clase se utilizan los modificadores `ClassType` y `Final`. Los modificadores se indican entre corchetes separados por comas, algunos necesitan una asignación y otros no ("[ClassType = persistent, Final]").
- ✓ El primer modificador, `ClassType`, permite indicar el tipo de la clase. En este caso es una clase persistente ("ClassType = persistent"), es decir, que las instancias de los objetos de esta clase se podrán almacenar en la base de datos de forma persistente. Poner este modificador no es suficiente para hacer la clase persistente, hay que poner también que se extiende la clase `%Persistent` ("extends %Persistent").
- ✓ El segundo modificador, "Final", permite indicar que esta clase no podrá tener clases hijas, es decir, que no podrá ser heredada con "extends". Si queremos que la clase pueda ser heredada, no debemos ponerlo. En contraposición a "Final" está el modificador "Abstract", que indica que no se podrán crear instancias de la clase, y que será necesario que dicha clase sea heredada por alguna subclase para poder crear instancias.

5.3.1.- Tipos de datos objeto. (II)



Y, ¿esto no se puede hacer más fácilmente? Pues sí. Te recomendamos en todo momento usar el asistente de creación de clases de la herramienta Studio de Caché para evitar problemas con el lenguaje. Sea como sea, tanto en Caché como en Oracle las clases y los tipos de dato objeto creados pueden ser reutilizados como propiedades o atributos de otros objetos. Esto permite materializar las relaciones de composición entre clases (permitiendo que un objeto esté compuesto de otros). Por ejemplo, la siguiente clase, pensada para contener los datos de una dirección, puede ser usada como atributo de otra clase:

```
Class User.Direccion Extends %SerialObject [ ClassType = serial ]
{
// Tipo de la vía (calle, paseo, paraje, etc.)
Property tipovia As %String;
// Nombre de la vía
Property nombrevia As %String;
// Número.
Property numero As %Integer;
// Bloque, planta, puerta y otro tipo de información.
Property dirinterior As %String;
Property codigopostal As %String;
Property localidad As %String;
Property provincia As %String;
Property region As %String;
Property pais As %String;
}
```

La clase **User.Direccion** podría usarse como propiedad en otra clase del mismo paquete, simplemente indicando como tipo el nombre de la clase al declarar la propiedad (recuerda que Caché distingue entre mayúsculas y minúsculas, no es lo mismo **User.Direccion** que **user.direccion**):

```
Class User.Conductor Extends %Persistent [ClassType = persistent]
{
Property direccion As Direccion;
}
```

Las clases de tipo serial (indicado con el modificador “**ClassType = serial**” y extendiendo la clase “**%SerialObject**”), son clases que no pueden ser persistentes por sí mismas, solo pueden ser persistentes cuando se incrustan o embeben en otra clase, es el caso de la clase “**User.Direccion**”. Para facilitar las

explicaciones, en esta unidad solo se usarán clases tipo serial como atributos de otras clases, puesto que su uso es más sencillo, por lo que te recomendamos que si piensas embeber una clase en otra, uses clases tipo serial para tal fin. No obstante, una clase persistente puede también ser usada como propiedad en otra clase, aunque aquí no lo veamos.

Para saber más

En el siguiente enlace encontrarás información abundante acerca de la definición de clases en Caché: propiedades, métodos, índices y mucho más.

[Usando objetos Caché.](#)



Autoevaluación

En Caché, ¿cuáles de los siguientes tipos de clases no pueden tener instancias por si solas?

- ☐ Persistent.
- ☐ Serial.
- ☐ Abstract.
- ☐ Final.

Mostrar Información

5.3.2.- Tipos de datos objeto. (III)

¿Y cómo sería la sintaxis de las clases en Oracle? ¡Pregunta trampa! Recuerda, no son clases lo que se usa en Oracle y en otras bases de datos objeto-relacionales, sino tipos de dato objeto, ¿recuerdas? Más o menos ya se introdujo con anterioridad la sintaxis SQL de Oracle para crear tipos de datos objeto. Para acercarnos un poco más a esta sintaxis, vamos a recrear el mismo ejemplo propuesto para Caché, esta vez en Oracle.

Primero crearemos el tipo de dato objeto nuevo, **tipo_direccion**, pensado para almacenar los datos de una dirección. Veamos como sería el código SQL de Oracle:



```
CREATE TYPE tipo_direccion AS OBJECT(  
    tipovia VARCHAR2(10),  
    nombrevia VARCHAR2(20),  
    numero INTEGER,  
    dirinterior VARCHAR2(30),  
    codigopostal CHAR(5),  
    localidad VARCHAR(20),  
    provincia VARCHAR(20),  
    region VARCHAR(20),  
    pais VARCHAR(15));
```

La sintaxis para crear un tipo nuevo siempre tiene más o menos la misma forma (ya iremos viendo algunos detalles adicionales más adelante). Comienza por "**CREATE TYPE**", a lo que sigue el nombre del tipo de dato ("**tipo_direccion**"), después indicamos que lo que vamos a crear es un tipo de dato objeto, y para ello ponemos "**AS OBJECT**". Después, entre paréntesis y separados por comas se indica la lista de atributos y métodos. En este caso, como no hay métodos, no hay que declarar el cuerpo de la clase.

Después de ejecutar el código SQL anterior, el tipo de dato objeto **tipo_direccion** se habrá creado y se podrá usar como tipo en otro objeto, lo cual es muy simple: para crear un atributo de dicho tipo se pone el nombre del atributo seguido del tipo. ¿Fácil no? Veamos como sería:

```
CREATE TYPE tipo_conductor AS OBJECT (  
    nombre VARCHAR(20),  
    apellidos VARCHAR(20),  
    fecha_nacimiento DATE,  
    direccion tipo_direccion);
```

Después, para poder almacenar objetos, debemos crear una tabla de objetos de ese tipo, esto en Oracle se hace con la siguiente sintaxis SQL siguiente, explicada ya en apartados anteriores:

```
CREATE TABLE conductores OF tipo_conductor;
```

Como se dijo antes, y aunque nosotros no lo usaremos, los objetos de Oracle pueden ser usados también en una tabla normal del modelo relacional, como si de un atributo se tratara:

```
CREATE TABLE empleados (  
    -- ...
```

```
identificacion VARCHAR2(20),  
nombre VARCHAR2(20),  
sueldo      NUMBER(7,2),  
direccion tipo_direccion);
```

Sin entrar en detalles, dado que este enfoque es menos interesante para la persistencia de objetos, podemos decir que el ejemplo anterior crea una tabla con datos de empleados y empleadas, en la que cada fila contendrá una identificación (el DNI o el NIE por ejemplo), el nombre del empleado o la empleada, su sueldo y además, contendrá su dirección almacenada en una estructura tipo objeto.

Para saber más

En el siguiente enlace podrás aprender un poco más sobre los objetos de Oracle.

[Introducción a los objetos de Oracle.](#)

5.4.- Atributos. (I)

¿Qué es lo más importante de una clase? Sus atributos. Está claro que sin los atributos una clase no es nada, ni siquiera los métodos de una clase son nada sin los atributos de una clase. En Caché se llaman propiedades, aunque llamarlo atributos o propiedades es indiferente. Ya se ha visto más o menos en apartados anteriores como añadir una propiedad, ahora vamos a profundizar un poco más. Nuevamente, tienes que tener en cuenta que Caché tiene múltiples opciones, y que aquí solo veremos las más importantes.

```
// Apellido del conductor, tipo cadena de texto.  
Property Apellido As String [ Required ];  
// Fecha de nacimiento del conductor, tipo fecha, formato DD/MM/AAAA.  
Property FechaNacimiento As %Date(FORMAT = 4) [ Required ];  
// Lista de mails.  
Property Mails As List Of String;  
// Array con los teléfonos.  
Property Telefonos As array Of String;  
// Calcula la edad y retorna un número entero.  
Property Edad As Integer(MINVAL = 15, MAXVAL = 100);
```

Una propiedad se crea simplemente usando la palabra reservada “**Property**”, seguida del nombre que deseamos que tenga de la propiedad, y seguida del tipo. El tipo de la propiedad se indica a través de la palabra reservada “**As**”, a la que debe seguir el tipo deseado. Finalizamos la declaración con punto y coma no lo olvides, veamos un ejemplo:

```
Property FechaNacimiento As %Date;
```

A la hora de crear una propiedad, y aquí viene la cuestión clave, podemos especificar opcionalmente algunos parámetros que modificarán el comportamiento del tipo de dato. Los parámetros se ponen entre paréntesis, justo detrás del tipo de dato. Veamos un ejemplo, en el siguiente ejemplo puedes ver los parámetros más importantes:

```
Property FechaNacimiento As %Date(FORMAT = 4);  
Property HorasSemanales As %Numeric(MINVAL=1,MAXVAL=40,SCALE=2);  
Property Identificacion As %String(MINLEN=6,MAXLEN=10);
```

El parámetro **FORMAT** usado con el tipo de dato **%Date** permite especificar el formato de la fecha. El formato número 4 (“**FORMAT=4**”) corresponde con el formato de fecha europeo (“DD/MM/AAAA”, dos dígitos para el día, seguidos de otros dos para el mes y cuatro para el año). Los parámetros **MINVAL** y **MAXVAL** permiten especificar, en tipos de datos numéricos, el valor mínimo y el máximo admitido. El parámetro **SCALE**, en determinados tipos, como el tipo “**%Numeric**”, permite especificar el número de decimales. Y los parámetros **MINLEN** y **MAXLEN** permiten indicar a las cadenas de texto el número mínimo y máximo de caracteres que admite.

Cada tipo de dato tiene sus propios parámetros, incluso podemos definir parámetros para nuestras propias clases, aunque no vamos a entrar en ese aspecto. Pero aparte de poder acotar el funcionamiento de un tipo de dato a través de sus parámetros, podemos imponer o establecer opciones de funcionamiento a una propiedad. Las opciones pueden ser de muchos tipos y son independientes del tipo de dato, es decir, todos los tipos de datos pueden tener estas opciones. Las más comunes son: obligatoriedad de que una propiedad tenga valor (no pudiendo estar vacía) y definir un valor inicial. Veamos un ejemplo:

```
Property HorasSemanales As %Integer (MINVAL=1,MAXVAL=40,SCALE=2) [Required, InitialExpression=0];
```

Este tipo de opciones se indican entre corchetes, algunas pueden tener una asignación de un valor y otras no. “**Required**” es la primera de las opciones, e indica que la propiedad tiene que tener un valor obligatoriamente, no puede estar vacía. “**InitialExpression**” es la segunda opción y permite indicar que la propiedad tomará ese valor en caso de que no se le asigne uno en el momento de creación de la clase, es digamos, el valor por defecto.



Autoevaluación

¿Cuál de los siguientes parámetros permite indicar el número de decimales de algunos tipos numéricos?

- ☐ FORMAT.
- ☐ DECIMALS.
- ☐ SCALE.
- ☐ MINVAL.



5.4.1.- Atributos. (II)

¿Y qué podemos decir de los atributos en Oracle? La creación de atributos ya se perfiló antes. Indicar que un objeto tiene un atributo concreto, se hace a través del ODL de Oracle, y consiste en indicar simplemente el nombre que queremos para el atributo seguido del tipo de dato, todo ello por supuesto se realiza en la definición del tipo de dato objeto:



```
CREATE TYPE tipo_conductor AS OBJECT (  
    nombre VARCHAR(20),  
    apellidos VARCHAR(20),  
    fecha_nacimiento DATE,  
    permiso VARCHAR(5),  
    puntoscarnet INTEGER);
```

Ahora bien, al igual que pasaba con el caso anterior, con los atributos de Caché, un atributo puede tener “acotaciones” que modifican su comportamiento. Algunas de estas acotaciones, como indicar el número máximo de caracteres de una cadena, o la cantidad de decimales de un número, se hacen a través del tipo, como se vio en el apartado de tipos de datos. Por ejemplo, el siguiente atributo tendría 10 dígitos como máximo y dos decimales:

```
numero number(10,2)
```

Pero otras restricciones más profundas, como por ejemplo indicar que un atributo solo admite un rango de valores, o que un atributo no puede estar vacío, o que una cadena de caracteres tenga al menos una longitud máxima y mínima, habría que indicarlas al crear la tabla de objetos, y no sobre el tipo de dato objeto. Por ejemplo:

```
CREATE TABLE conductores OF tipo_conductor (  
    CHECK (nombre IS NOT NULL AND apellidos IS NOT NULL AND  
        LENGTH(permiso)>2 AND puntoscarnet IS NOT NULL AND  
        puntoscarnet>0 AND puntoscarnet<=15));
```

Para incluir restricciones en una tabla de objetos se puede usar la cláusula **CHECK**. La cláusula **CHECK** se puede poner junto a otras restricciones y se pondrá a continuación de la definición de la tabla de objetos, entre paréntesis. Y entre paréntesis también, se indican todas las restricciones a comprobar, por ejemplo: que un campo tenga valor obligatoriamente (“**nombre IS NOT NULL**”), que la longitud de una cadena contenga un mínimo o un máximo de caracteres (“**LENGTH(permiso)>2**”) o que un número esté entre un rango de valores (“**puntoscarnet>0**”). Cada comprobación se une con el resto con **AND** (equivalente al operador lógico “&&” de Java) o con **OR** (equivalente a operador lógico “|” de Java), pudiendo usar paréntesis para agrupar comparaciones.

Para saber más

Si quieres saber un poco más acerca de las restricciones de Oracle, puedes consultar el siguiente documento. Las restricciones tipo **CHECK** están a partir de la página 8, pero todas son interesantes. Cuando lo leas, ten en cuenta que en el ámbito de los objetos se usan las restricciones a nivel de

tabla, no a nivel de columna. Fijate también en el uso de las sentencias **ALTER** para modificar las restricciones de una tabla.

[Restricciones en Oracle.](#) (46.42 KB)



Autoevaluación

La palabra reservada **CHECK** permite indicar condiciones que tienen que cumplir los objetos almacenados en Caché y en Oracle. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

5.4.2.- Atributos. (III)

¿Y qué otras restricciones podemos imponer a los atributos de una clase o de un tipo de dato objeto? Tanto en Caché como en Oracle hay varias restricciones para los atributos que son especialmente importantes. Se trata de las restricciones de llave primaria y de unicidad.

Uno o varios atributos pueden ser llave primaria, lo cual significa que ese atributo, o esos atributos si se trata de más de uno, identifican de forma unívoca a cada objeto, puesto que no habrá otro objeto que tenga los mismos datos en ese o esos atributos. Por ejemplo, el DNI valdría como llave primaria para objetos que almacenan datos de personas, dado que no habrá dos personas con el mismo DNI. ¿Verdad? De esa forma, conociendo el DNI, podríamos localizar a cualquier persona sin error.



El concepto de unicidad es similar al de llave primaria, pero diferente en algunos aspectos. En un objeto no puede haber más de una llave primaria (puede ser una llave primaria compuesta de varios atributos, pero no puede haber más de una). La unicidad consiste en que no puede haber dos objetos con el mismo valor en un determinado atributo. Un ejemplo de atributo único podría ser el correo electrónico, dado que no habría dos personas que en principio compartieran el mismo correo electrónico.

Para indicar que uno o varios atributos son llave primaria, o que un atributo es único, en Caché se realizaría a través de los índices. Cuidado, no confundas los índices primarios con las llaves primarias, son conceptos diferentes, aunque relacionados entre sí (las llaves primarias y los atributos únicos se usan normalmente para crear índices primarios). Los índices son indicaciones realizadas a la base de datos, en la que le decimos cuales son los campos más buscados, de forma que se puedan optimizar las búsquedas. Veamos como sería:

```
Class User.Empleado Extends %Persistent
{
    Property DNI As %String;
    Property email As %String;
    Index DNIidx On DNI [PrimaryKey];
    Index emailidx On email [Unique];
}
```

En este caso se utilizaría la palabra reservada **Index**, seguida de un nombre que le asignamos al índice. Después se pondría la palabra reservada **On** seguida del atributo o atributos sobre los que se aplica el índice (cuando son más de uno se ponen entre paréntesis y separados por comas). A continuación, se indica entre corchetes si se trata de una llave primaria (**PrimaryKey**) o de un atributo único (**Unique**).

En Oracle, la llave primaria y los atributos únicos se establecerían en la sentencia de creación de la tabla de objetos y no en la definición del tipo de dato objeto. Sería de la siguiente forma:

```
CREATE TYPE tipo_empleado AS OBJECT(
    dni VARCHAR(10),
    email VARCHAR(50));

CREATE TABLE empleados OF tipo_empleado
(CONSTRAINT dni_c PRIMARY KEY (dni), CONSTRAINT email_c UNIQUE (email));
```

La palabra reservada **CONSTRAINT** permite indicar una restricción. A continuación de la misma se indica el nombre de la restricción (**dni_c** y **email_c**) y después la restricción en sí. En el ejemplo aparece por un lado la restricción de clave primaria y por otro la de unicidad. La restricción de clave primaria se pone con la palabra reservada **PRIMARY KEY** seguida del atributo o atributos a los que se aplica entre paréntesis (si fueran más de uno se

pondrían separados por comas). La restricción de unicidad se pondría de forma similar, pero con la palabra reservada **UNIQUE**.

Oracle crearía un índice primario automáticamente para las claves primarias y los atributos únicos.



Autoevaluación

En una clase de Caché o en una tabla de objetos solo puede haber una llave primaria y un atributo único. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

5.5.- Métodos. (I)

¿Y cómo se programan los métodos en las bases de datos? A la hora de programar los métodos en las bases de datos normalmente no se utiliza el lenguaje de programación que se utiliza para programar la aplicación final. Esto es por un lado, un problema, dado que al cambiar el lenguaje de programación es necesario aprender nuevos lenguajes, pero por otro lado es necesario, dado que los requisitos de programación en una base de datos distan bastante de los requisitos de programación de un lenguaje de uso general.



Vamos a echar un vistazo a los diferentes métodos de programación en los diferentes SGBD que estamos explorando:

- ✓ Caché permite una gran variedad de lenguajes de programación para métodos: ObjectScript (lenguaje similar al lenguaje M) y Basic. También permite otros lenguajes, como Java, pero con el inconveniente de que los métodos en Java no se ejecutan ni compilan por el SGBD, sino que se ejecutan y compilan por y en la aplicación cliente. Esto quiere decir que los métodos Java de una clase Caché, se ejecutan en tu programa cuando se invocan desde tu programa, y los métodos ObjectScript y Basic se ejecutan en el servidor cuando tu programa invoca un método remoto.
- ✓ Oracle permite crear métodos de dos tipos: procedimientos y funciones. Los métodos en Oracle se programan principalmente en PL/SQL, aunque también es posible programarlos en Java. El proceso de programación en Java es bastante más liso que en PL/SQL, por lo que este último suele ser la opción principal. En ambos casos los métodos se ejecutan en el servidor.

A la hora de programar en una u otra base de datos, hay que tener en cuenta que Caché no permite sobrecarga de métodos (métodos con el mismo nombre pero con diferentes argumentos), mientras que Oracle si. Veamos ahora como son los mecanismos para crear un método en ambas bases de datos.

La definición de un método dentro de una clase de Caché, se suele hacer de la siguiente forma:

```
Method edad(t1 As %String, t2 As %String) As %String [Language = cache]
{
    Set e = ($PIECE($NOW(),",",1)-$PIECE(..FechaNacimiento,",",1))/365
    Quit t1_e_t2
}
```

Un método puede ser mucho más complejo que el ejemplo anterior pero con esto es suficiente para el propósito de esta unidad. El ejemplo anterior muestra un ejemplo de método escrito en el lenguaje ObjectScript. La palabra reservada **"Method"** se utiliza para indicar que se va a crear un método. El nombre del método es **"edad"**, el cual tiene dos parámetros, **t1** y **t2**, que son de tipo cadena de texto (**%String**).

Como resultado, el método retornará una cadena, detalle que se indica poniendo **"As"** y el tipo a devolver detrás de los paréntesis con los argumentos (**"As %String"**). Es opcional indicar que el método devolverá algo, no tiene porqué devolver nada. No obstante, el uso que en esta unidad se hará de los métodos implica que devuelvan algo.

Opcionalmente se puede indicar, detrás del tipo a devolver por el método, el lenguaje de programación del método, que por defecto será el lenguaje propio de Caché, es decir, ObjectScript. Esto se indica poniendo la palabra clave **"Language"**, un signo de igual, y el lenguaje a usar después, todo entre corchetes. Los lenguajes que se pueden utilizar son **"cache"** (ObjectScript), **"basic"** y **"java"** entre otros. Entre los corchetes podemos definir muchas más palabras clave separadas por comas, pero no es necesario que de momento las conozcas.

Y entre llaves iría obviamente todo el código del método. Si se usa ObjectScript, hay que tener cuidado con el uso de los espacios. Como se comento antes, hay que dejar al menos un espacio al comienzo de cada sentencia y conviene dejar un único espacio entre elementos del lenguaje (dejar más podría dar error en algunos casos).

Debes conocer

En el siguiente documento encontrarás un resumen orientativo de como es la sintaxis de ObjectScript, comparada con la sintaxis de Java. Ten en cuenta que el objetivo principal de ese documento no es que aprendas un nuevo lenguaje, sino que sepas más o menos, como es la sintaxis para poder defenderte y aprender más por tu cuenta.

[Resumen de la sintaxis de ObjectScript.](#)

5.5.1.- Métodos. (II)

¿Y cómo son los métodos de Oracle? Como se comentó en el apartado anterior, los métodos en Oracle se programan principalmente en PL/SQL y tienen dos fases: definición en el tipo de dato objeto y declaración, en lo que se denomina, cuerpo del tipo de dato objeto. En Oracle tenemos dos tipos de métodos: procedimientos y funciones. Los procedimientos, en principio, no retornan un valor y las funciones sí. Aunque se explican los dos tipos, nosotros usaremos las funciones.



En primer lugar tendríamos que crear los tipos de datos objeto indicando que van a tener uno o varios métodos. Esto es relativamente sencillo, dado que consiste simplemente en indicar la cabecera del método. Veamos un ejemplo:

```
CREATE TYPE tipo_coordenada AS OBJECT (  
    x      NUMBER,  
    y      NUMBER,  
    MEMBER PROCEDURE moverPunto(incx NUMBER, incy NUMBER),  
    MEMBER FUNCTION distanciaCon (x2 NUMBER, y2 NUMBER) RETURN NUMBER  
);
```

El tipo de dato objeto anterior está pensado para poder almacenar un par de coordenadas cartesianas y en el se han declarado dos métodos: un procedimiento que permite desplazar un punto, indicando el incremento en la dirección "x" y en la dirección "y" (procedimiento **moverPunto**), y una función que permite calcular la distancia con otro punto (función **distanciaCon**). Para declarar un método, simplemente se usa la palabra reservada "**MEMBER**" seguida de "**PROCEDURE**" o "**FUNCTION**" dependiendo de si se trata de un procedimiento o de una función respectivamente. Después se indica el nombre del método, seguido de los argumentos del método entre paréntesis y separados por comas. En el caso de que el método sea tipo función, se deberá indicar además el tipo a retornar, con la palabra reservada **RETURN** seguida del tipo de dato.

Los parámetros del método pueden ser uno, varios o ninguno, y van separados por comas, y se indican simplemente poniendo el nombre de la variable seguida del tipo (**incx NUMBER**, por ejemplo). Siempre existe el parámetro **SELF**, aunque no se indique manualmente. **SELF** es una referencia a la propia instancia de objeto desde la que se invoca el método. "**SELF.x**" sería el atributo x del objeto, aunque en principio no es necesario poner "**SELF**" delante.

Pero lo anterior, es solo la creación del tipo, ahora tenemos que crear el cuerpo y definir los métodos. Veamos como sería:

```
CREATE TYPE BODY tipo_coordenada AS  
    MEMBER PROCEDURE moverPunto(incx NUMBER, incy NUMBER) IS  
    BEGIN  
        x:=x+incx;  
        y:=y+incy;  
    END;  
    MEMBER FUNCTION distanciaCon (x2 NUMBER, y2 NUMBER) RETURN NUMBER IS  
        distancia NUMBER;  
    BEGIN  
        distancia:=SQRT(POWER(x2-x,2)+POWER (y2-y,2));  
        RETURN distancia;  
    END;  
END;
```

La creación del cuerpo del tipo de dato objeto, o lo que es lo mismo, la definición de los métodos, comienza por **"CREATE TYPE BODY"**, seguido del nombre del tipo dato objeto en cuestión, y la palabra reservada **"AS"**, a continuación de la cual se pondrá la definición de cada método. La definición de cada método comienza por repetir la cabecera declarada al crear el tipo de dato objeto, seguida de un bloque de declaración de variables (entre **"IS"** y **"BEGIN"**), donde se declararán las variables que se usarán en el método, y del código del método (entre **"BEGIN"** y **"END;"**). En las funciones se utilizará el término reservado **"RETURN"** obligatoriamente, seguido del valor a retornar, para obviamente indicar el valor devuelto por la función.

Para saber más

En el siguiente enlace puedes ampliar tus conocimientos sobre la sintaxis de PL/SQL cuando se usa con objetos. En esta unidad no se profundizará en esta sintaxis, dado que no es el objetivo principal, pero aquí puedes encontrar más información:

[Uso de PL/SQL y objetos.](#)

Y en el siguiente enlace encontrarás un completo tutorial sobre PL/SQL, que no está enfocado a su uso con objetos, pero que te servirá para conocer mejor su sintaxis.

[Tutorial de PL/SQL.](#)

5.6.- Herencia. (I)

¿Y para qué sirve la herencia en una base de datos? Pues exactamente para lo mismo que en Java. Tanto Caché como Oracle permiten implementar herencia en su base de datos.

En Caché la sintaxis para indicar que una clase hereda las propiedades y los métodos de otra clase es igual que en Java, usando el operador **extends**, por lo que no te costará ningún trabajo. La única salvedad es que caché permite **herencia múltiple** de la cual en principio no vamos a hacer uso. Supongamos que tenemos la siguiente clase:

```
CLASS USER.CONDUCTOR EXTENDS %PERSISTENT
{
  // DNI es un documento de identificación (DNI, NIE u otro tipo)
  Property DNI As String ( Required )
  Index DNI On DNI ( PrimaryKey, Unique )
  // Nombre del conductor, tipo cadena de texto.
  Property Nombre As String ( Required )
  // Apellidos del conductor, tipo cadena.
  Property Apellidos As String ( Required )
  // Fecha de nacimiento del conductor.
  Property FechaNacimiento As Date ( Required )
  // Puntos de carnet.
  Property PuntosCarnet As Integer ( MINVAL=0, MAXVAL=15 )
  // Lista de
  Property Maletas As Array ( String )
  // Array colapsado.
  Property Ticks As Array ( String )
  // Salidas y entradas en el aeropuerto.
}
```

```
Class User.Persona extends %Persistent {
  Property nombre As %String;
  Property apellidos As %String; }
```

Esta clase ya hereda la clase **%Persistent**, usando el operador **extends** para ello. Se trata de una clase que contiene los datos mínimos de cualquier persona, su nombre y sus apellidos. Veamos ahora como sería para crear una clase que herede o extienda la clase anterior:

```
Class User.Conductor Extends User.Persona{
  Property PuntosCarnet As %Integer(MINVAL=0,MAXVAL=15);
}
```

En el ejemplo se crea la clase **Conductor** que extiende la clase **Persona** añadiendo el atributo **PuntosCarnet** a las propiedades anteriores. Si la clase está en el mismo paquete (ambas clases anteriores están en el paquete **User**), no es necesario poner “**Extends User.Persona**”, basta con poner “**Extends Persona**”. La clase **Conductor** hereda también la capacidad de persistencia de la clase **Persona**, dado que dicha clase ya había extendido la clase **%Persistent**. Si la clase **Persona** no hubiera extendido la clase **%Persistent**, podríamos indicar que nuestra nueva clase **Conductor** extiende o hereda varias clases, simplemente indicando el listado de clases entre paréntesis y separado por comas:

```
Class User.Socio Extends (Persona,%Persistent) {
  Property PuntosCarnet As %Integer(MINVAL=0,MAXVAL=15); }
```

La herencia en Caché no afecta particularmente a los métodos. Si en la clase **User.Persona** existiese un método que es necesario sobrescribir en la clase **Conductor**, simplemente se crearía el método en la clase **Conductor**, usando el mismo nombre de método, con exactamente los mismos argumentos y exactamente retornando el mismo tipo de dato. Si no se usan los mismos argumentos (en cuanto número y tipo), dará error, dado que Caché no permite la sobrecarga de métodos.

Veamos un ejemplo de sobreescritura de métodos, donde un mismo método “**getNombreApellidos**” se comporta de forma diferente en un caso y en otro:

```
Class User.Persona Extends %Persistent {
  Property nombre As %String;
  Property apellidos As %String;
  Method getNombreApellidos () as %String {
    Quit ..apellidos_, "_.nombre }
```

```
}
```

```
Class User.Conductor Extends (Persona,%Persistent) {  
Property PuntosCarnet As %Integer(MINVAL=0,MAXVAL=15);  
Method getNombreApellidos () as %String {  
    Quit ..nombre_"_"..apellidos }  
}
```



Autoevaluación

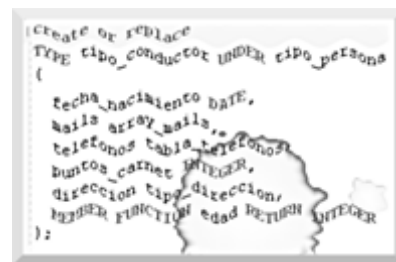
¿Cuál de las siguientes afirmaciones sobre la herencia en la base de datos Caché es falsa?

- ☐ Caché permite la sobrecarga de métodos heredados.
- ☐ Caché permite sobrescribir métodos heredados.
- ☐ Caché permite la herencia múltiple.

5.6.1.- Herencia. (II)

¿Y qué ocurre en Caché con las propiedades heredadas? ¿Se pueden sobrescribir cambiándole el tipo por ejemplo? En cuanto a las propiedades de las clases heredadas, tienes que tener en cuenta que no puedes volver a definir una propiedad ya existente en una clase heredada (esto es así tanto en Oracle como en Caché).

En Oracle la herencia se hace efectiva a través de subtipos, es decir, creando lo que se denomina subtipos de tipos de datos objeto ya existentes. A diferencia de Caché, donde cualquier clase puede heredarse salvo que se marque como final, los tipos de objeto Oracle no pueden heredarse, porque son siempre finales, salvo que se indique lo contrario. Esto significa que en los objetos de Oracle, para poder heredarlos, hay que marcarlos como “no finales”. Veamos un ejemplo:



```
CREATE TYPE tipo_persona AS OBJECT (  
    nombre          VARCHAR2(20),  
    apellidos       VARCHAR2(40),  
    MEMBER FUNCTION getNombreYApellidos RETURN VARCHAR2  
) NOT FINAL;  
  
CREATE TYPE BODY tipo_persona AS  
    MEMBER FUNCTION getNombreYApellidos RETURN VARCHAR2 IS  
    BEGIN  
        RETURN apellidos || ', ' || nombre;  
    END;  
END;
```

En el ejemplo anterior se crea el tipo objeto “**tipo_persona**” equivalente al ejemplo propuesto para Caché. Fíjate que en la declaración del tipo de dato objeto **tipo_persona** se añade al final “**NOT FINAL**”, con eso se consigue que el tipo de dato objeto sea no final, y que por lo tanto, pueda heredarse. Para hacer un subtipo podríamos hacerlo de la siguiente forma:

```
CREATE TYPE tipo_conductor UNDER tipo_persona (  
    puntoscarnet INTEGER,  
    OVERRIDING MEMBER FUNCTION getNombreYApellidos RETURN VARCHAR2)  
NOT FINAL;  
  
CREATE TYPE BODY tipo_conductor AS  
    OVERRIDING MEMBER FUNCTION getNombreYApellidos RETURN VARCHAR2 IS  
    BEGIN  
        RETURN nombre || ' ' || apellidos;  
    END;  
END;
```

Para crear un subtipo se utiliza el término reservado **UNDER** a continuación del cual hay que indicar el tipo objeto del cual se heredarán métodos y propiedades, este será conocido como el supertipo.

Para sobrescribir un método existente se utiliza la palabra reservada **OVERRIDING**. Dicha palabra reservada se debe poner tanto a la hora de declarar el método, como a la hora de volver a escribir su código en el cuerpo del tipo de dato objeto.

Resumamos las características de la herencia tanto en Caché, como en Oracle:

- ✓ En Caché se permite herencia múltiple, mientras que en Oracle no.
- ✓ Tanto en Caché como en Oracle se pueden sobrescribir métodos ya existentes.
- ✓ Ni en Caché, ni en Oracle se pueden redefinir atributos que ya existan en una clase padre o supertipo.



Autoevaluación

¿Para qué sirve la palabra reservada **OVERRIDING** de Oracle?

- ☐ Para sobrecargar un atributo.
- ☐ Para sobrescribir un método.
- ☐ Para usar poder usar los métodos de los supertipos en los subtipos.



5.7.- Constructores.

¿Sabes qué es un constructor en programación? Seguro que si, dado que Java usa constructores, y es una de las técnicas usadas en la programación orientada a objetos.

Los constructores en un SGBDOO o en un SGBDOR permiten crear e inicializar una instancia de objeto partiendo de una serie de valores pasados como argumentos, garantizando que esa inicialización será lo primero que se hará al crear la instancia. Ahora te preguntará, ¿cómo se pone un constructor en Caché? ¿Y en Oracle? Pues Caché no permite la creación de constructores, lo cual es un inconveniente en algunos aspectos, pero Oracle sí. De hecho, en Oracle son muy importantes, dado que los constructores se pueden usar para introducir una nueva instancia de objeto en la base de datos.

Veamos pues cómo son los constructores en Oracle. Todo tipo de dato objeto de Oracle tiene un constructor por defecto.

Imagina la siguiente tabla de objetos:

```
CREATE TYPE tipo_persona AS OBJECT (  
    nombre          VARCHAR2(20),  
    apellidos       VARCHAR2(40)  
) NOT FINAL;  
  
CREATE TYPE tipo_conductor UNDER tipo_persona (  
    puntoscarnet INTEGER)  
    NOT FINAL;
```

El constructor por defecto sería el siguiente:

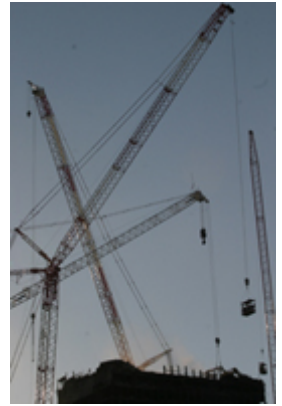
```
tipo_conductor('Pepe', 'García Rodríguez', 12)
```

En Oracle todo tipo de dato objeto tiene un constructor, cuyos argumentos son todos los atributos del objeto, puestos en el orden que se han puesto en la declaración del tipo objeto. Ese constructor se usa, por ejemplo, a la hora de guardar un objeto nuevo en la base de datos:

```
CREATE TABLE conductores OF tipo_conductor;  
INSERT INTO conductores VALUES (tipo_conductor('Pepe', 'García Rodríguez', 12));
```

En el ejemplo anterior, se inserta un nuevo **tipo_conductor** en la tabla **conductores**. Eso se realiza a través de la expresión **INSERT INTO** de SQL, en la que se indica, por un lado la tabla de objetos donde se va a insertar el objeto (**conductores** en este caso), y por otro, los valores a insertar. La palabra reservada **VALUES**, permite indicar que valores se insertarán, después de ponerla se indican los valores a insertar en la tabla encerrados entre paréntesis. En este caso, el valor a insertar se obtiene a partir del constructor del tipo de dato objeto **tipo_conductor**, por tanto, lo que hay que poner como valor a insertar es simplemente el constructor.

Los constructores también se utilizan en los procedimientos y las funciones programadas con PL/SQL, para crear por ejemplo una instancia de un objeto.



Para saber más

Oracle permite personalizar, es decir, crear a medida tus propios constructores. En la siguiente página, en inglés, puedes profundizar en ese aspecto:

[Creación de constructores a medida para tipos objeto de Oracle.](#)



Autoevaluación

Toda clase de Caché tiene un constructor por defecto, el cual contiene todos los atributos de la clase, en el orden en el que se han puesto en la definición de la clase. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

5.8.- Tipo de datos colección. (I)

¿Qué es una colección? Una colección, como ya sabrás de unidades anteriores, es una estructura que permite almacenar varios datos, generalmente del mismo tipo, de forma que sean fácilmente accesibles. En Java, por ejemplo, son muy usadas las listas y los mapas.



De cara a una base de datos, las colecciones son estructuras que nos permiten almacenar varios datos, generalmente del mismo tipo, y esto se materializa en atributos que son capaces de contener varios objetos del mismo tipo. En las bases de datos, las colecciones son bastante diferentes a las colecciones en Java, son menos flexibles y hay menos tipos. Veamos qué colecciones tenemos en las bases de datos que estamos explorando:

- ✓ **Caché.** En Caché una propiedad puede ser de dos tipos de colecciones: **listas y arrays asociativos** (similares a los mapas en Java). La diferencia entre ambas es que las listas son simplemente un conjunto de elementos a los que se accede por su posición, mientras que los arrays son de tipo asociativo, es decir, permiten acceder a un elemento por una clave que puede ser un número o una cadena. El número de elementos que podemos almacenar en una lista y un array de Caché es ilimitado.
- ✓ **Oracle.** En Oracle DB las colecciones pueden ser de dos tipos: tablas anidadas y arrays (conocidos como VARRAYS). La principal diferencia entre ambas estructuras, es que las tablas anidadas permiten almacenar un número dinámico de elementos, digamos sin límite lógico, mientras que los VARRAYS almacenan un número fijo de elementos, aunque este puede ser cambiado. Las tablas anidadas se gestionarán como una tabla que está dentro de otra tabla, mientras que los VARRAYS serán digamos una propiedad con múltiples valores del mismo tipo.

Veamos cómo se usan las colecciones de objetos. Empezaremos por Caché, donde se podrá crear una nueva propiedad tipo array o tipo lista de la siguiente forma:

```
Class User.Conductor Extends (Persona)
{
    Property PuntosCarnet As %Integer(MINVAL=0,MAXVAL=15);
    Property Mails As List Of %String;
    Property Telefonos As Array of %String;
}
```

En el ejemplo anterior se añaden dos propiedades tipo colección a la clase **Conductor**, una lista de correos electrónicos y un array de teléfonos. Los tipos base para la lista y el array son cadenas de texto (**%String**), pero se podría haber utilizado cualquier otro tipo o clase creada por nosotros. Para declarar estas propiedades se pone **"As List Of"** o **"As Array Of"**, seguido del tipo de dato a almacenar en la lista, dependiendo de si se trata de una lista (primer caso) o de si se trata de un array (segundo caso). Las listas en Caché están implementadas por la clase **%ListOfObjects**, mientras que los arrays están implementados por la clase **%ArrayOfObjects**, esto significa que cuando creamos una lista o un array, esta realmente se traduce en una de esas clases. Dichas clases proveen de sendos métodos para gestionar los elementos que tienen en su interior.

Los correos electrónicos del ejemplo anterior se almacenarían como una lista. En la lista se accede a los elementos por la posición que ocupan, simplemente hay que indicar el número correspondiente a su posición. Pero los teléfonos se almacenarían como un array asociativo, y eso implica que para acceder no se indica su posición, sino una cadena de texto o número que actúa de llave. Los teléfonos se almacenarían asociados a una llave, lo cual es muy útil. Por ejemplo, podríamos decir que el teléfono asociado a la llave "FAX" es el teléfono "12345678", y que el asociado a la llave "MÓVIL" es otro diferente. Los arrays de caché son el equivalente a los mapas de Java (**HashMap** por ejemplo).

Debes conocer

No te pierdas el siguiente código de ejemplo. En él puedes ver como se usan las listas y los arrays asociativos de Caché, usando ObjectScript. Hay métodos para eliminar, modificar, extraer, etc. elementos de listas y arrays, verás que realmente es muy sencillo:

[Ejemplo de uso de listas y arrays en Caché.](#)

Y en el siguiente archivo encontrarás las clases anteriores, importables a Caché a través de la herramienta Studio.

[Paquete de ejemplo con usos de las listas y arrays en Caché.](#) (3 KB)

En los siguientes enlaces podrás ver con detalle los métodos implementados en las clases **%ListOfObjects** y **%ArrayOfObjects**:

[Detalles de la clase %ListOfObjects.](#)

[Detalles de la clase %ArrayOfObjects.](#)

5.8.1.- Tipo de datos colección. (II)

¿Y cómo sería para Oracle? Pues bastante diferente, dado que no hay correlación entre los tipos de colecciones de Oracle y los de Caché. El caso del array de teléfonos podríamos resolverlo en Oracle a través de tablas anidadas, pero primero hay que crear un tipo de dato objeto nuevo (**tipo_telefono** por ejemplo). Crearemos un tipo nuevo que contendrá el número de teléfono y el tipo de teléfono asociado (móvil, fax, etc.):



```
CREATE TYPE tipo_telefono AS object (  
    tipo varchar2(10),  
    numero varchar2(30)  
    ) NOT FINAL;
```

Después tendríamos que crear un tipo de dato nuevo, no visto hasta ahora, partiendo de **tipo_telefono**. Ese tipo de dato nuevo sería un **tipo de dato tabla**. Una vez hayamos creado este nuevo tipo de dato, podremos poner una tabla de objetos **tipo_telefono**, como atributo en otra tabla, permitiendo así crear una tabla anidada (por eso se llama tabla anidada, porque se usa como atributo de otro objeto). Veamos por tanto como se crea un tipo de dato tabla, que podrá contener objetos del tipo **tipo_telefono**:

```
CREATE TYPE tabla_telefonos AS TABLE OF tipo_telefono;
```

La sintaxis es bastante sencilla. Comenzaríamos por poner “**CREATE TYPE**” seguido del nombre del tipo. Y para decir que se trata de un tipo de dato tabla pondríamos “**AS TABLE OF**”, y después, habría que indicar el nombre del tipo de dato objeto en el que se basaría dicho tipo de datos tabla. Una vez creado el tipo **tabla_telefonos**, tenemos que usarlo como atributo. Para ello, nos vamos a la declaración del tipo objeto **tipo_conductor**, lugar donde vamos a introducir los teléfonos, e indicamos que uno de los atributos será la tabla anidada anterior:

```
CREATE TYPE tipo_conductor UNDER tipo_persona (  
    puntoscarnet INTEGER,  
    telefonos tabla_telefonos) NOT FINAL;
```

Para el caso de los mails, podemos hacer uso de la estructura VARRAY, presuponiendo por ejemplo que no tendremos más de 5 mails, como mucho, para un usuario normal (dado que los VARRAYs son de tamaño fijo). Para crear un VARRAY que contenga dichos mails debemos empezar creando también un tipo de dato nuevo, que será de forma efectiva un array de cadenas que contendrá correos electrónicos. Si bien, aunque los elementos del VARRAY en este caso son un tipo básico (**VARCHAR2**), es obviamente posible utilizar tipos de dato objeto. Veamos como sería:

```
CREATE TYPE array_mails AS VARRAY(5) OF VARCHAR2(40);
```

En el caso anterior se crea un nuevo tipo de dato, no visto hasta ahora, con el que se define un VARRAY. La creación del mismo se realiza poniendo **CREATE TYPE**, seguido del nombre del tipo de dato, y de la expresión “**AS VARRAY (5) OF**”, donde se debe especificar el número de elementos del VARRAY (el ejemplo propuesto tiene 5 elementos). Al final, no debemos olvidar poner el tipo de dato a usar para cada elemento del array (**VARCHAR2** u otro tipo). Ahora ya podemos completar nuestro **tipo_conductor**, añadiendo el varray para los mails:

```
CREATE TYPE tipo_conductor UNDER tipo_persona (  
    puntoscarnet INTEGER,  
    mails array_mails,  
    telefonos tabla_telefonos) NOT FINAL;
```



Autoevaluación

¿Cuál de las siguientes afirmaciones es incorrecta?

- ☐ Las tablas de Oracle se parecen los mapas de Java.
- ☐ Los arrays de Oracle se parecen los arrays de Java.
- ☐ Los arrays de Caché se parecen a los mapas de Java.
- ☐ Las listas de Caché se parecen a las listas de Java.



5.8.2.- Tipo de datos colección. (III)



¿Y ya está? ¿No hay que hacer nada más para crear una colección en Oracle? Todavía falta un paso más. Con lo visto en el apartado anterior ya tendrías el tipo de dato **tipo_conductor** creado. Ahora debemos crear una tabla que contendrá los conductores y las conductoras, proceso que ya se ha explicado antes, pero que cambia ligeramente al tener una tabla anidada. Para poder crear ahora la tabla de objetos deberemos indicar como se llamará la tabla anidada en la que se almacenarán los teléfonos, veamos como sería:

```
CREATE TABLE conductores OF tipo_conductor NESTED TABLE telefonos STORE AS telefonos_nt;
```

Esto crearía como ya sabes una tabla de objetos, donde cada fila es un único objeto. Los teléfonos se almacenarían en realidad en una tabla aparte, llamada "**telefonos_nt**", la cual se indica con la expresión "**NESTED TABLE telefonos STORE AS telefonos_nt**". Para rellenar la tabla podemos usar también SQL, recurriendo a la sentencia **INSERT** antes comentada y usando constructores:

```
INSERT INTO conductores VALUES (  
    tipo_conductor ( 'pepe', 'garcía', 12,  
    array_mails('mail1@direccionemail.com', 'mail2@direccionemail.com'),  
    tabla_telefonos (tipo_telefono ('fax','283847575'),tipo_telefono ('movil','28283748
```

En el ejemplo anterior, fíjate que se hace uso de varios niveles de constructores. Se comienza con el constructor para el **tipo_conductor**, que incluirá en su interior entre otras cosas, un constructor para el varray de mails, donde se han incluido dos mails (separados por comas), y un constructor para el tipo **tabla_telefonos**, en el que se insertan dos instancias de **tipo_telefono**, creadas a su vez usando el constructor del tipo de dato objeto **tipo_telefono**. Es un poco lioso, pero es solo al principio, cuando te acostumbras resulta más sencillo.



Autoevaluación

¿Cuál de las siguientes expresiones corresponde con crear un array en Caché?

- ☐ AS TABLE OF.
- ☐ AS ARRAY OF.
- ☐ AS ARRAY(5) OF.
- ☐ AS TABLE(5) OF.

Para saber más

En el siguiente documento sobre Oracle y su modelo de objetos, podrás ver un resumen de lo visto en este tema, y algunas cosas más. ¡Échale un vistazo!

[Modelo objeto-relacional en Oracle.](#) (72 KB)

6.- Mecanismos de consulta y acceso a la base de datos.

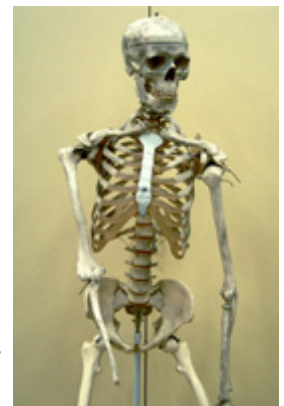
Caso práctico



Juan está muy emocionado, ya ha empezado a diseñar su base de datos, modelando los objetos de la aplicación, pero, sigue preocupado por un asunto: “Una vez diseñado el esquema de la base de datos, ¿cómo se meten datos en la base de datos? ¿Y cómo se recuperan? ¿Será muy difícil?”. No quiere molestar a Juan, porque sabe lo atareado que está, por lo que decide investigar por su cuenta los mecanismos para conectarse a la base de datos desde Java.

Bueno, ya sabes como crear una estructura que te permite almacenar los datos de tu aplicación en un almacenamiento persistente, pero, ¿cómo hacemos los datos persistentes? Hasta ahora solo tenemos un esqueleto en el que no hemos guardado nada. Ahora veremos un enfoque más práctico de las bases de datos, donde nuestra aplicación entra en contacto con la base de datos, literalmente.

Para poder almacenar y manipular objetos de la base de datos (insertar, modificar, eliminar y consultar objetos), los SGBD proporcionan un lenguaje de consulta de objetos (OQL, que en inglés significa Object Query Language) y un lenguaje de manipulación de datos (OML, que en inglés significa Object Management Language). A través de estos lenguajes realizaremos todas o casi todas las operaciones con los objetos: a través del OQL podremos recuperar los objetos almacenados, y a través del OML insertar, modificar y eliminar objetos.



Uno de los primeros intentos de definir un OQL estándar vino de la organización ODMG. El OQL propuesto por la desaparecida organización ODMG, se intentaba parecer al lenguaje SQL. OQL era por tanto similar, pero no igual, al lenguaje SQL, usado intensamente en el modelo relacional. Pero SQL evoluciono, en principio solo usado para bases de datos relacionales, mejoró para poder tener capacidades propias de sistemas orientados a objetos. Esa evolución se concreto en 1999 con SQL3.

Dado que SQL3 estaba respaldado por grandes empresas y por organismos de estandarización de gran calado (como ISO y ANSI), su expansión fue rápida y contundente, desplazando al OQL de ODMG. A todo esto había que sumar que realmente ODMG no proponía un OML propio, sino que proponía que el OML fuera una extensión del lenguaje de programación usado (C++, Java, etc.), mientras que SQL3 si tenía un OML claro y definido.

Todo esto ha repercutido en que muchos de los SGBDOO y SGBDOR actuales usan SQL de alguna forma. Quizás no se usa SQL3 o posterior de forma precisa, pero si usan una sintaxis bastante parecida. Y así ocurre en nuestro caso, Caché y Oracle usan SQL como OML y OQL, aunque el SQL de Oracle es más fiel a SQL3.

Ahora que ya sabemos con que lenguaje manipular los datos almacenados en la base de datos, necesitamos saber como acceder a la base de datos para poder manipular los objetos. Para ponerse en contacto con la base de datos y para manipular datos existen muchos mecanismos. Vayamos por partes, ¿qué mecanismos nos permite Caché? Veamos los principales:

- ✓ **Caché Java Binding.** Esta forma de manipular objetos en la base de datos es posiblemente la más cómoda. Con este mecanismo, a través de Caché, creamos una clase Java que estará vinculada directamente con una clase homóloga en Caché. En nuestra aplicación utilizaremos dicha clase Java para almacenar datos en la base de datos, sin tener que emplear SQL.
- ✓ **Caché Jalapeño.** Esta forma de almacenar objetos permite almacenar directamente objetos Java, sin tener que definirlos en Caché. Este sistema requiere de un motor de persistencia especial, capaz de transformar las clases Java, en objetos almacenables en la base de datos Caché. Para poder usarlo hay que crear las clases Java de una forma especial (con anotaciones).
- ✓ **Conexión a través de JDBC.** Esta forma de trabajo crea una conexión directa a la base de datos, a través de la cual se pueden ejecutar sentencias SQL que nos permitirán insertar, borrar, actualizar y consultar los objetos almacenados en nuestra base de datos. JDBC es propio de Java, es decir, va incluido en los paquetes JRE y JDK de Java, pero requiere de un **driver** específico para cada base de datos.

Y Oracle, ¿cómo podemos comunicarnos con Oracle desde Java? Pues tenemos varios mecanismos, veamos:

- ✔ **Conexión a través de JDBC.** Por supuesto, Oracle también permite JDBC, y por supuesto dispone de un driver para JDBC.
- ✔ **SQLJ.** Es un mecanismo que permite incrustar directamente sentencias SQL en el lenguaje Java, sin los complejos mecanismos de JDBC. Esto reduce enormemente el tiempo de desarrollo.
- ✔ **JPublisher.** Es un mecanismo equivalente a Caché Java Binding pero para Oracle.

De todos estos mecanismos, vamos a explorar Caché Java Binding, y la ejecución de sentencias SQL a través de JDBC.

7.- Almacenando objetos en la base de datos.

Caso práctico

Juan está intranquilo, está muy liado pero sabe que **Antonio** necesita ayuda. No sabe qué hacer, ¿le ayuda o no? **Juan** mira su propio escritorio, ve todo lo que tiene pendiente y se pone a sudar, pero decide ir a ver cómo lo lleva:

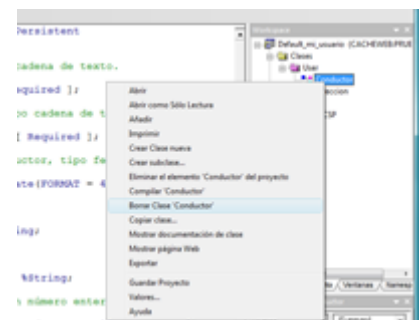
—¿Qué tal? ¿Cómo lo llevas? —dice **Juan**.

—Pues estoy muy asustado —dice **Antonio**—. **Juan**, si te soy sincero, no paro de mirar páginas y páginas, y me resulta imposible. Es muy difícil.

—No es tan difícil, lo que pasa es que es la primera vez que lo ves y siempre las primeras veces cuesta mucho. Voy a decirle a **María** que te eche una mano, creo que ella está un poco más libre que yo.



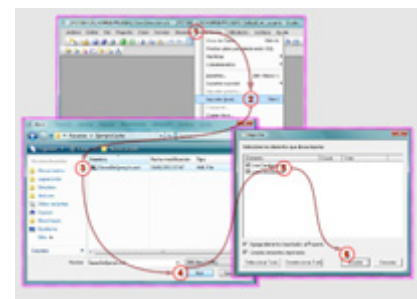
Antes de continuar, ¿recuerdas los ejemplos que hemos ido poniendo con anterioridad en los que se creaban diferentes tipos de clases y tipos de datos objeto? A lo largo de lo que queda de unidad, vamos a poner ejemplos sobre como conectarse a Caché y a Oracle, desde Java, y de como gestionar los datos almacenados. Para facilitar su comprensión, te facilitamos a continuación el esquema de la base de datos que se usará como punto de partida de dichos ejemplos, tanto para Caché como para Oracle. El esquema de ejemplo es una continuación de los ejemplos puestos antes.



Antes de continuar, si ya has creado en Caché clases llamadas "**Conductor**" o "**Direccion**" (como en los ejemplos anteriores), es conveniente borrarlas. Para borrar una clase simplemente despliega el menú contextual de la clase a borrar, desde el navegador de proyecto de la herramienta Caché Studio, y selecciona la opción "Borrar clase". Fíjate que en el navegador de proyecto hay tres pestañas, hay una para navegar por los archivos de tu proyecto (los que tienes incorporados a tu proyecto actual), y otra para navegar por las clases del namespace. Las clases pueden no estar en tu proyecto (porque no las estés usando), pero si en el namespace (almacenadas en la base de datos).

El esquema que te facilitamos para Caché, va en formato XML. Para incorporarlo a tu proyecto tienes que seguir los siguientes pasos:

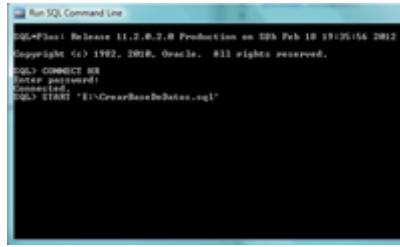
1. En la herramienta Caché Studio ve al menú Herramientas.
2. Selecciona la opción "Importar Local...". Esto te permitirá buscar el archivo XML entre los archivos de tu ordenador.
3. Busca el archivo XML que contiene el esquema de Caché.
4. Abre el archivo XML, te saldrá una nueva ventana emergente donde seleccionar las clases a importar.
5. Selecciona las clases a importar (todas en realidad).
6. Dale a la opción de importar. Como resultado las clases se incorporarán a tu proyecto y se compilarán, quedando listas para ser usadas.



Para usar el esquema de ejemplo de Oracle, puedes hacerlo desde SQL*Plus. Se incluyen dos scripts SQL, uno para borrar el esquema y otro para crearlo. Para ejecutar los scripts puedes conectarte, por ejemplo, con el usuario HR desde SQL*Plus ("**CONNECT HR**") y después puedes usar el comando **START** para ejecutar los scripts. Al comando **START** tienes que indicarle la ruta completa al script a ejecutar, por ejemplo: "**START 'E:\CrearEsquemaBaseDeDatos.sql'**".

En Oracle es conveniente borrar primero los posibles tipos existentes para poder volver a crearlos, y lo mismo pasa con las tablas. Si intentas crear una tabla o un tipo que ya existe, en general no te dejará crearlo (existen

mecanismos para reemplazar o cambiar la definición de un tipo o una tabla, pero aquí no se han visto).



```
SQL*Plus Release 11.2.0.2.0 Production on 18 FEB 18 17:35:54 2012
Copyright (c) 1982, 2010, Oracle. All rights reserved.

SQL> CONNECT HR
Connected.
SQL> START 'E:\OracleBase\Debes.sql'
```

Debes conocer

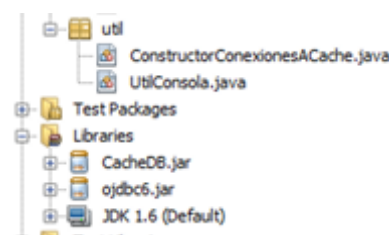
En el siguiente archivo comprimido se incluyen los archivos necesarios para recrear en tu ordenador los esquemas necesarios para seguir los ejemplos. Son esquemas sencillos que seguro no representan ninguna dificultad para ti.

[Esquemas de ejemplo para Caché y Oracle.](#) (6 KB)

7.1.- Preparando nuestro proyecto para almacenar objetos.

¿Y por dónde se empieza? ¿Cómo conecto con la base de datos? Damos por sentado que ya tienes un esquema de objetos creado en la base de datos, capaz de dar soporte a los datos de tu aplicación. Lo siguiente a realizar es conectar tu aplicación con la base de datos.

Para poder realizar la conexión, es necesario incluir en tu proyecto las librerías que te permitirán conectarte a la base de datos. Veamos cómo incorporar una librería nueva en nuestro proyecto NetBeans:



1. Abre un proyecto de aplicación Java existente o crea uno nuevo en NetBeans. Después, en el navegador de proyecto despliega la sección "Librerías" (en inglés, Libraries), del árbol de proyecto.
2. Con el botón derecho del ratón haz clic encima de la sección librerías en "Añadir Jar/Carpeta" (en inglés, Add Jar/Folder). Sacará un cuadro de diálogo que te permitirá seleccionar la librería.
3. Localizas el archivo JAR correspondiente a la librería a incorporar y le das a aceptar. Ya tienes la librería incorporada a tu proyecto.

Estos pasos son necesarios si quieres usar la base de datos Caché desde tu aplicación Java, y obviamente también para usar la base de datos Oracle. Veamos las librerías que hay que importar en cada caso:

- ✓ Para poder acceder a la base de datos Caché desde tu aplicación hay que incorporar la librería de cliente Caché en tu proyecto, llamada **CacheDB.jar**. Esta librería podrás encontrarla en la carpeta de instalación de Caché (en el apartado 2 se avisó de que había que apuntar la carpeta de instalación, ¿recuerdas?), en el subdirectorio "lib". Esta librería contendrá todo lo necesario para conectarse con tu aplicación, incluido el driver JDBC.
- ✓ Para poder acceder a Oracle (versión XE 11g) desde tu aplicación debes incorporar el driver JDBC de Oracle. El driver JDBC está disponible a través de la página web de Oracle y hay diferentes versiones. Dentro de toda la maraña de versiones, caben destacar las versiones llamadas: **ojdbc5.jar**, driver JDBC de Oracle a utilizar cuando se usa Java 5, y **odbc6.jar**, driver JDBC de Oracle a usar cuando se usa Java 6 o superior.

Para saber más

En el siguiente enlace puedes descargar los drivers JDBC de Oracle:

[Drivers JDBC de Oracle.](#)

Una vez incorporadas las librerías necesarias a nuestro proyecto, para conectarse a la base de datos hay que "preparar" cierta información. Es necesario saber la [dirección IP](#) o nombre del ordenador donde estará instalado el SGDB. Si hacemos la aplicación pensando en acceder a un servidor de bases de datos que estará instalado en el mismo ordenador que la aplicación, la dirección IP será "127.0.0.1" o "localhost". Es lo que usaremos para hacer pruebas en fase de desarrollo.

Para conectarnos a Caché, necesitaremos también saber cuál es el espacio de nombres con el que queremos trabajar (uno que hayamos preparado nosotros o bien el espacio "USER" preexistente en caché). Y en el caso de Oracle, necesitaremos el nombre de la instancia de la base de datos con la que vamos a trabajar. La base de datos se llamará "XE" en el caso de Oracle XE 11g, dado que esta versión va restringida y solo deja una base de datos.

Y por último, necesitamos también el nombre de usuario y la contraseña. Recuerda que Caché tiene por defecto el usuario "_SYSTEM" con la contraseña "SYS", lo cual es útil en el caso de que no hayas podido crear un usuario nuevo. Recuerda también que Oracle XE dispone del usuario "HR", el cual es necesario desbloquear, tal y como se dijo en apartados anteriores (si no se desbloquea, no se podrá conectar a la base de datos usando dicho usuario).




Autoevaluación

Completa con las palabras que faltan.

Para poder acceder desde Java a la base de datos usando es necesario incorporar la librería del a nuestro proyecto. También tenemos que conocer cierta información para realizar la conexión, como por ejemplo el espacio de , en el caso de Caché, o el nombre de la instancia de la base de datos, en el caso de .

7.2.- Conectando con la base de datos.

 Imagen que muestra un armario de datos con un montón de conexiones de red, y dos operarios arreglando un problema existente.

Y ahora vamos a realizar una conexión a nuestra base de datos, que nos permitirá manipular los datos almacenados. Esta conexión la haremos a través de JDBC. JDBC se verá con mayor profundidad en la siguiente unidad, por lo que aquí vamos a verlo de forma básica. Para crear una conexión que nos permita conectarnos a la base de datos lo primero que debemos hacer indicar a Java, con la sentencia import, los paquetes implicados en el proceso:

- ✓ `java.sql.*`: contiene las clases necesarias de Java para poder conectarse vía JDBC a una base de datos. Es necesario tanto para Caché como para Oracle.
- ✓ `com.intersys.jdbc.*` y `com.intersys.objects.*`: contienen las clases necesarias para establecer una conexión a Caché, vía JDBC, y para poder usar Caché Java Binding.
- ✓ `oracle.jdbc.pool.*`: contiene las clases necesarias para realizar una conexión a Oracle, a través de JDBC. También vamos a necesitar las clases del paquete `oracle.sql.*`, para poder manejar los tipos de datos colección.

Para conectar por tanto con la base de datos, debes importar los paquetes correspondientes, y luego efectuar la conexión. Para crear la conexión se usan las clases **CacheDataSource** con Caché u **OracleDataSource** con Oracle, dependiendo de lo que estemos utilizando. Tanto **CacheDataSource** como **OracleDataSource** son clases derivadas de la clase **DataSource**. La clase **DataSource** es parte de JDBC y agrupa una colección de métodos que permitirán conectarse a la base de datos.

Pero antes de usar las clases anteriores, debemos preparar lo que se denomina URL de conexión. La URL de conexión es una cadena que contiene información para conectarse a la base de datos. En nuestro caso la URL contendrá la dirección IP o nombre de la máquina en la que está instalado el SGBD. Contendrá además el [puerto](#) para conectarse (generalmente Caché usa el puerto 1972 por defecto y Oracle el puerto 1521) y el espacio de nombres (en caso de Caché) o la instancia de la base de datos (que en caso de Oracle XE será "XE").

Y una vez hallamos conectado, la información de conexión se encapsulará en una instancia de la clase **java.sql.Connection**, la cual usaremos para manipular los datos. Veamos como sería el proceso de conexión:

Ejemplos de conexión con los SGBD Cac

Conectando con Caché vía JDBC	
<pre>CacheDataSource ds = new CacheDataSource(); String ipDelServidor="127.0.0.1"; Integer puertoDelServidor=1972; String namespace="USER", String usuario="_SYSTEM", password="SYS"; String url = "jdbc:Cache://" + ipDelServidor + ":" + puertoDelServidor + "/" + namespace; ds.setURL(url); ds.setUser(usuario); ds.setPassword(password); Connection conn = ds.getConnection();</pre>	<pre>OracleD String Integer String String String ipDelSe ds.setU ds.setU Connect</pre>

En los ejemplos anteriores, primero se crea una instancia de **CacheDataSource** u **OracleDataSource** según corresponda, en ambos casos se almacena en una variable llamada **ds**. Después se construye una cadena de texto que contiene la URL (la cual tiene un formato específico en cada caso), añadiendo la información necesaria: IP o nombre de la máquina, puerto y espacio de nombres o instancia de la base de datos, según corresponda. Después se indica la URL, el usuario y la contraseña con los métodos **setURL**, **setUser** y **setPassword** respectivamente. Y por último se establece la conexión con el método **getConnection**. De todo el proceso, lo único que realmente cambia entre un SGBD y otro es el formato de la URL.

El proceso de conexión puede generar una excepción de tipo **java.sql.SQLException**, que es necesario capturar y procesar. Esta excepción ocurrirá cuando no se pueda establecer una conexión con la base de datos.



Para saber más

En el siguiente archivo comprimido se facilitan dos clases muy interesantes, de esas clases que te hacen la vida más fácil. Se trata de la clase **ConexionACache** y **ConexionAOracle**, que te permitirán cómodamente crear una conexión sin complicarte la vida.

[Clases facilitadoras de la conexión a Caché y Oracle.](#) (2.66 KB)



Autoevaluación

En Caché solo puede generarse más de una proyección por clase, e incluso, para varios lenguajes. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.



7.4.- Sentencias de inserción de datos en la base de datos.

¿Y cómo insertamos objetos en la base de datos cuando no se usan proyecciones? Para insertar objetos sin usar proyecciones lo que se usa es la sentencia **INSERT** de SQL. Esta sentencia ya se ha explicado con anterioridad para Oracle, y aquí vamos a profundizar sobre su uso en Caché. Para empezar, tienes que tener en cuenta que la sentencia **INSERT** se usa en el modelo relacional, donde se insertan filas en una tabla, y que aquí vamos a ver su utilización para insertar objetos. Oracle por ejemplo es una base de datos objeto-relacional, y la sentencia **INSERT** puede usar constructores para crear e inicializar un nuevo objeto, aunque también podrá utilizarse la sintaxis que vamos a ver aquí para Caché.



La sintaxis que usa Caché para la sentencia **INSERT** es similar a la que se usaría en una base de datos relacional, donde no se usan constructores. Caché permite manipular los objetos almacenados de una determinada clase, como si de una tabla relacional se tratase. Esto es porque mantiene de forma paralela una proyección SQL de las clases creadas, de modo que pueden ser accedidas como si se tratase de tablas. La sintaxis que se usaría para la sentencia **INSERT** en Caché sigue el siguiente formato (el cual también se puede usar en Oracle):

```
INSERT INTO clase ( lista de atributos separados por comas) VALUES (lista de valores sep
```

La sentencia **INSERT** tiene fundamentalmente dos partes. Una primera parte que se utiliza para indicar la clase y una lista de atributos, y una segunda parte que se utilizaría para indicar los valores a insertar. Entre paréntesis, después del nombre de la clase, se pueden indicar los atributos en los que se va a guardar un valor, dado que no es necesario poner a priori un valor para todos los atributos. Si no se pone la lista de atributos entonces si es necesario indicar el valor de todos los atributos. Eso si, siempre es imprescindible poner un valor para aquellos atributos que se hayan marcado como obligatorios o que no pueden estar vacíos. Veamos un ejemplo:

```
INSERT INTO Conductor (DDI, Nombre,Apellidos,FechaNacimiento,PuntosCarnet) VALUES ('1234
```

Del ejemplo anterior, se pueden extraer varias conclusiones a tener en cuenta de ahora en adelante:

- ✓ El número de atributos tiene que coincidir con el número de valores.
- ✓ Las cadenas de texto se ponen entre comillas simples ('cadena'). Esto es así tanto en Oracle como en Caché.
- ✓ Los valores indicados en la segunda parte de la sentencia **INSERT** pueden ser una expresión compleja, donde incluso se pueden usar funciones. Es el caso del valor indicado para el atributo PuntosCarnet, donde se usa una expresión matemática que suma dos números ("10+4"). También es el caso del valor indicado para la fecha de nacimiento (FechaNacimiento), que usa la función **TO_DATE**. Esto es así tanto en Caché como en Oracle.

Vamos a pararnos un poco en la función **TO_DATE**, disponible tanto en Caché como en Oracle. Esta función convierte una cadena que contiene una fecha, la cual es legible por un ser humano, al formato interno de la base de datos. Para ello, se le proporciona por un lado la fecha, y por otro, el formato de la fecha. El formato de fecha usado en Europa es **'DD/MM/YYYY'**, lo cual significa: dos dígitos para el día (**DD**), dos dígitos para el mes (**MM**) y cuatro dígitos para el año (**YYYY**).

Recomendación

Si usas la interfaz web de Caché para ejecutar sentencias SQL no pongas punto y coma al final de la sentencia. Mientras que en Oracle es normal siempre finalizar la sentencia en punto y coma, se use la

interfaz que se use, en Caché da error si pones un punto y coma al final.



Autoevaluación

Dada la clase numeros, la cual tiene dos atributos no obligatorios tipo %Integer (t1 y t2), indica cuáles de las siguientes formas de insertar objetos serían válidas para Caché usando la sentencia INSERT:

- ☐ INSERT INTO numeros VALUES (tipo_numero (10,20)).
- ☐ INSERT INTO numeros VALUES (10,20).
- ☐ INSERT INTO numeros (t1) VALUES (10).
- ☐ INSERT INTO numeros () VALUES (10,20).

Mostrar Información

7.5.- Almacenando objetos a través de JDBC. (I)

Las proyecciones son un mecanismo fácil y sencillo de almacenar, modificar y manipular objetos, pero en muchas ocasiones se quedan cortas, y es necesario recurrir a otros mecanismos. ¿Qué alternativas tenemos? Para superar las limitaciones que tienen las proyecciones podemos usar JDBC, disponible en Caché y en Oracle.

Para empezar, debemos disponer de una conexión, ya sea a Caché o a Oracle vía JDBC. Una vez dispongamos de ella, la conexión nos facilitará la preparación de sentencias SQL y la ejecución de las mismas en el SGBD. Veamos primero algunas peculiaridades adicionales sobre el uso de la sentencia **INSERT** en la base de datos Caché. Tienes que tener en cuenta lo siguiente:



- ✓ A la hora de almacenar objetos de una determinada clase hay que tener en cuenta el paquete donde se ubica la clase. Recuerda que cuando creas una clase, ésta se ubica dentro de un paquete. Por ejemplo, la clase "User.Conductor" es la clase Conductor dentro del paquete User.
- ✓ No te confundas de espacio de nombres. No podrás acceder a las clases creadas en otro espacio de nombres, salvo que se hayan mapeado los paquetes de forma adecuada, y eso es algo que no se ha explicado.
- ✓ Caché proyecta cada clase como una tabla del mismo nombre (la clase Conductor se proyectaría como la tabla Conductor).
- ✓ Caché proyectará cada paquete de clases como un esquema SQL. Un esquema es en la práctica como un paquete, pero que contiene tablas. Ten en cuenta lo siguiente:
 - Las clases del namespace "USER" almacenadas en el paquete "User" se proyectarán en SQL dentro de un esquema llamado SQLUser, que es el esquema por defecto. Al ser el esquema por defecto no es necesario poner "SQLUser.Conductor" para acceder desde SQL a los datos, basta con poner "Conductor" para hacer referencia a la tabla proyectada en SQL.
 - Las clases de otros paquetes se proyectarán como un esquema del mismo nombre que el paquete. Por ejemplo: la clase prueba del paquete test (test.prueba) se proyectará como la tabla prueba del esquema test (test.prueba). De esta forma para acceder desde SQL a una tabla proyectada de un paquete diferente a "User" tienes que poner, en notación de punto, el paquete y la clase ("test.prueba" por ejemplo).

Una vez aclarados los puntos anteriores, ya podemos empezar con la inserción de objetos en la base de datos de Caché. Lo primero es construir una sentencia SQL de inserción acorde con nuestras necesidades, por ejemplo:

```
String SQLq="INSERT INTO Conductor (Nombre, Apellidos, FechaNacimiento,  
PuntosCarnet, direccion_nombrevia, direccion_numero, direccion_codigopostal, DDI)  
VALUES (?,?,?,?,?,?,?,?)";
```

Fijate que la sentencia anterior es parecida a las sentencias vistas hasta ahora, pero que usa interrogantes. Cada interrogante ("?",) representa un **parámetro de consulta**, y se usa para indicar que en ese lugar se va a introducir un valor desde JDBC, podríamos indicarlo manualmente pero hacerlo desde Java nos quita muchos problemas.

Un detalle que no se te debe pasar por alto es como se proyectan los atributos de la clase **Direccion**. Recuerda que la clase **Direccion** antes comentada, es tipo serial, y se utiliza como propiedad en la clase **Conductor**, dando lugar a la propiedad llamada **direccion** (en la clase Caché aparece con la d minúscula). Fijate que podemos acceder a cada atributo de la propiedad **direccion** poniendo un guión bajo y el nombre del atributo detrás. De esa forma "**direccion_nombrevia**" sería el atributo **nombrevia** de la propiedad **direccion**, y "**direccion_numero**" sería el atributo **numero** de la propiedad **direccion**, y así para todos los casos. Cuidado, esto es así en Caché, pero en Oracle no. Caché proyecta de esta forma las clases tipo serial. Si **Direccion** no fuera tipo serial se proyectaría de otra forma.

Recomendación

Ten en cuenta lo siguiente: las sentencias SQL ejecutadas a través de JDBC no deben acabar en punto y coma. En Oracle, por ejemplo, cuando la sentencia se ejecuta a través de SQL*Plus, se suele poner un punto y coma al final (o una barra "/" en una línea sola, depende de la costumbre), pero en JDBC no es así. Dependiendo de la interfaz, a veces se pone punto y coma al final, y otras no.

7.5.1.- Almacenando objetos a través de JDBC. (II)

Bien, ahora que ya tenemos nuestra sentencia SQL de inserción preparada para JDBC, procedamos a efectuar la inserción de datos. En los ejemplos propuestos a continuación "**conn**" es una instancia de la clase **Connector**, la cual contiene como se dijo en apartados anteriores toda la información relativa a la conexión.



En primer lugar preparamos la sentencia para su ejecución. Para ello usamos el método **prepareStatement** de la clase **Connector**, al que pasamos como argumento la consulta anterior, y si es necesario, algunas opciones de funcionamiento. En este caso, la opción que pasamos es **RETURN_GENERATED_KEYS**, la cual nos permitirá obtener el OID del objeto creado. El resultado de la ejecución de dicho método generará una instancia de **PreparedStatement**, que nos servirá para ejecutar la sentencia SQL:

```
PreparedStatement ps=conn.prepareStatement(SQLq,Statement.RETURN_GENERATED_KEYS);
```

El OID, por si no lo recuerdas, se vio al comienzo de la unidad, y es un identificador único que tiene toda instancia de objeto. Al almacenar el objeto en la base de datos, se le asigna un OID, y esto ocurre tanto en Caché como en Oracle. Para nosotros será muy útil, puesto que teniendo el OID podremos abrir una instancia de una clase y manipularla desde la correspondiente proyección Java (usando Caché Java Bindings).

Una vez preparada la consulta, tenemos que indicar el valor a poner en cada parámetro de la consulta, es decir, decimos que hay que poner en sustitución a cada una de las interrogantes. Para ello debemos contar la posición del interrogante con respecto al resto de interrogantes, y empezando a numerar por 1, vamos invocando los métodos "**setString**" (para poner un parámetro tipo cadena), "**setDate**" (para poner el valor en un parámetro tipo fecha), "**setInt**" (para poner un parámetro tipo numérico), etc. El primer argumento de estas funciones es la posición del parámetro del cual se quiere indicar el valor, y el segundo argumento es el valor a fijar en dicha posición.

Lo importante aquí es usar la función adecuada para cada parámetro. Si un atributo en Caché o en Oracle es de tipo número entero, deberemos usar "**setInt**" y no "**setString**". Veamos como continuaríamos nuestro código ahora que ya sabemos todo esto:

```
ps.setString(1, "Nombre de ejemplo");
ps.setString(2, "Apellidos de ejemplo");
ps.setDate(3, java.sql.Date.valueOf("2000-01-01"));
ps.setInt(4, 15);
ps.setString(5, "Calle Desconocida");
ps.setInt(6, 55);
ps.setString(7,"93939");
ps.setString (8,"12345C");
```

Fíjate en el uso de **setDate**. En el ejemplo anterior la fecha se pone a través el método estático **ValueOf** de la clase **java.sql.Date**, pasando como argumento una cadena con la fecha en formato "AAAA-MM-DD" (cuatro dígitos para el año, dos dígitos para el mes y dos dígitos para el día, separados por guiones). Esto generará una instancia de la clase **java.sql.Date** que contiene dicha fecha. Y cuidado, porque **java.sql.Date** no es igual que **java.util.Date** (el primero es para JDBC y el segundo para el sistema), aunque es posible convertir de un tipo a otro.

Ahora ya podemos ejecutar la consulta. Esto es lo más fácil de todo, para ejecutarla simplemente invocamos el método **executeUpdate** de la clase **PreparedStatement**:

```
ps.executeUpdate();
```



Autoevaluación

¿Sería correcto poner `setInt(0,99)`?

- ☐ Sí.
- ☐ No.



7.5.2.- Almacenando objetos a través de JDBC. (III)

¿Y los constructores? Adicionalmente, en Oracle, como ya se explico en apartados anteriores, se pueden usar constructores en la sentencia **INSERT**. Y esto, como ya veremos más adelante, será muy útil. De hecho, para insertar una dirección en la tabla de objetos **conductores** del esquema de datos que os hemos facilitado, habría que usar un constructor de **tipo_direccion**.



La sentencia SQL siguiente sería la equivalencia Oracle a la expuesta en apartados anteriores para Caché. Oracle no proyecta de igual forma los atributos que son tipo de dato objeto, no existiendo "**direccion_nombrevia**" por ejemplo, y existiendo la necesidad de usar constructores:

```
String SQLq = "INSERT INTO CONDUCTORES  
(nombre, apellidos, fecha_nacimiento, puntos_carnet, direccion, ddi)  
VALUES (?, ?, ?, ?, TIPO_DIRECCION(NULL, ?, ?, NULL, ?, NULL, NULL, NULL, NULL), ?);"
```

En esta sentencia SQL se usa el constructor por defecto de **tipo_direccion**, el problema es que al usar el constructor por defecto, y no crear un constructor diferente y propio, estamos obligados a poner un valor para todos los atributos, y claro, es posible que haya valores que no vayamos a rellenar, sea el motivo que sea. Para esos casos, se pone **NULL**. **NULL** significará "no hay dato".

NULL se puede poner usar tanto en Oracle, como en Caché, dado que en ambas bases de datos significará "no hay dato" (y en general, se puede usar en cualquier base de datos objeto-relacional). Si ocurre que un atributo, por ejemplo **puntos_carnet**, para el cual habíamos previsto poner un dato a través de JDBC con un parámetro de consulta (un interrogante), de repente, no tiene dato, habría que usar la función "**setNull**" de la clase **PreparedStatement** (en el ejemplo siguiente ps sigue siendo una instancia de **PreparedStatement**):

```
ps.setNull(4, java.sql.Types.INTEGER);
```

setNull obviamente se usa tanto en Oracle como en Caché, y se indica por un lado la posición del interrogante (primer argumento), y por otro, el tipo base SQL que corresponde al tipo de dato almacenado en dicha posición (**java.sql.Types.INTEGER** para enteros, **java.sql.Types.VARCHAR** para cadenas, **java.sql.Types.DATE** para fechas, etc.). De cara a JDBC, los tipos de datos almacenados en la base de datos tienen un "nombre" SQL fijo, y será el SGBD el que deberá proveer, a través del driver JDBC, la transparencia necesaria entre tipos.

Esto implica que al usar **setNull** en Caché, dado que los tipos de datos usados son diferentes a los estandarizados para SQL, debemos buscar cual es la equivalencia SQL al tipo de dato Caché usado. Por ejemplo, en un atributo tipo **%String** se usaría **java.sql.Types.VARCHAR**.

Y el resto del código en Java para insertar un objeto en la base de datos, a través de JDBC, se quedaría igual que el descrito en el apartado anterior, solo cambiaríamos, en este caso concreto, la sentencia SQL, que es ligeramente diferente entre Caché y Oracle.

Debes conocer

Échale un vistazo al siguiente documento sin dudarlo, en él podrás ver cómo insertar datos en los tipos de datos colección, tanto para Caché, como para Oracle.

Para saber más

En el siguiente archivo encontrarás un proyecto de NetBeans de ejemplo, en el que se utiliza la sentencia **INSERT** para insertar datos en la base de datos de ejemplo que te hemos proporcionado. El proyecto no incorpora las librerías CacheDB.jar ni odbc6.jar, por lo que deberás incluirlas para poder compilar el proyecto.

[Insertar conductores y conductoras a la base de datos a través de la sentencia INSERT.](#) (670 KB)

Para saber como se proyectan en SQL cada tipo de dato usado por Caché, puedes consultar la siguiente tabla:

[Analogías entre los tipos de datos SQL y los tipos de datos usados por Caché.](#)



Autoevaluación

Si el atributo T es de tipo entero en la base de datos, ¿cuál de las siguientes líneas de Java valdría para indicar que su valor es NULL con `setNull`?

- ☐ `setNull(1, java.sql.Types.INTEGER).`
- ☐ `setNull("T", java.sql.Types.INTEGER).`
- ☐ `setNull(1, java.sql.Types.VARCHAR).`
- ☐ Ninguna de la anteriores.

8.- Manipulando los objetos almacenados.

Caso práctico

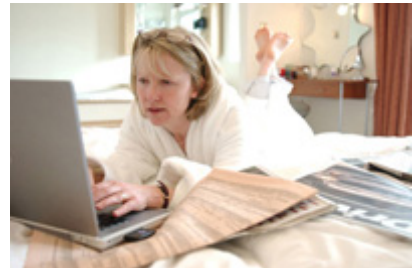


María ha estado ayudando a **Antonio**, y menos mal, porque **Antonio** se había atascado con JDBC y no conseguía avanzar. Pero ahora que ya ha conseguido solucionar sus problemas, lo ve todo mucho más sencillo, sobre todo porque para comunicarse con la base de datos se usa SQL, algo que ya conocía con anterioridad.

Conoce SQL pero tiene que repasarlo, porque hay algunas cosas que no recuerda muy bien. Es fácil olvidarse de las cosas que no se utilizan todos los días, así que siempre conviene tener documentación al lado que nos sirva de referencia.

¿Y cómo recupero, modifico o elimino los datos almacenados? Ya sabemos como insertar objetos en la base de datos a través de diferentes mecanismos. Ahora vamos a explorar de que formas podemos realizar el resto de operaciones.

Ya hemos visto la sentencia **INSERT** de SQL, la cual permitiría insertar objetos en la base de datos. Ahora veremos como recuperar objetos de la base de datos con la sentencia **SELECT**, como eliminarlos con la sentencia **DELETE** y como modificarlos con la sentencia **UPDATE**.



Para ejecutar estas sentencias podemos seguir usando JDBC. Las sentencias **DELETE** y **UPDATE** se ejecutan igual que un **INSERT**:

1. Creamos la sentencia SQL usando los interrogantes donde queramos pasar un parámetro. Por ejemplo:

```
String sqlu="UPDATE Conductor SET PuntosCarnet=PuntosCarnet-3 WHERE DDI=?";
```

2. Creamos una instancia de la clase `PreparedStatement` usando la sentencia anterior. Por ejemplo (conn una instancia de la clase `Connection`):

```
PreparedStatement ps=conn.prepareStatement(conn,sqlu);
```

3. Sustituimos los parámetros usando la función `set` de la clase `PreparedStatement` adecuada:

```
ps.setString(1,"12345C");
```

4. Ejecutamos la consulta usando el método `executeUpdate` de la clase `PreparedStatement`:

```
ps.executeUpdate();
```

En cambio, ejecutar una sentencia **SELECT** a través de JDBC es ligeramente diferente, dado que hay que recoger los resultados obtenidos tras la consulta, para después poder procesarlos uno a uno. Ten en cuenta que como

resultado de una consulta tipo **SELECT** se pueden obtener datos de más de un objeto, por ejemplo, podríamos recuperar todos los nombres de todos los conductores en una sola consulta. Veamos como sería:

1. Creamos la sentencia SQL tipo SELECT.

```
sqlq="Select Nombre,Apellidos,FechaNacimiento from Conductor";
```

2. Creamos igualmente una instancia de la clase PreparedStatement:

```
PreparedStatement ps2=conn.prepareStatement(conn,sqlq);
```

3. Indicamos los parámetros usando la función set adecuada, pero como nuestra consulta no tiene parámetros, no hace falta.
4. Ejecutamos la consulta usando el método executeQuery de la clase PreparedStatement. Esto generará una instancia de la clase ResultSet, que contendrá el resultado de la consulta:
code
5. Para extraer los datos de un objeto, y después, poder extraer los datos del siguiente objeto, tenemos que usar el método next() de la clase ResultSet. El método next(), es como un iterador, permite ir recorriendo todos los resultados obtenidos, uno por uno. Al principio, no apunta a ninguno de los resultados, su primera ejecución hará que apunte al primer resultado, su segunda ejecución hará que se apunte al segundo resultado, y así sucesivamente. Devolverá false cuando ya no queden más resultados que extraer. Podemos volver a hacer que apunte al primer resultado con el método first() de la clase ResultSet.
6. Conforme vayamos recorriendo la lista de resultados con la función next(), tendremos que ir extrayendo los datos con las funciones tipo get de la clase ResultSet. Tendremos una función get para cada tipo de dato a recuperar: getString para cadenas, getInt para enteros, getDate para fechas (en formato java.sql.Date), etc. A cada función get se le pasa como argumento la posición del dato a extraer (comenzando a numerar por 1). Veamos un ejemplo:

```
while (rs.next())
{
String Nombre=rs.getString(1);
Apellidos=rs.getString(2);
java.sql.Date fn=rs.getDate(3);
System.out.println("-->" +Nombre+" "+Apellidos+" "+fn.toString());
}
```

A JDBC se le dedica la siguiente unidad. Aquí haremos un uso básico del mismo.



8.1.- Recuperación de los objetos almacenados. (I)

¿Y cómo funciona la sentencia **SELECT**? La sentencia **SELECT** nos va a permitir recuperar información de los objetos almacenados en la base de datos, y su sintaxis es relativamente compleja, aunque aquí vamos a ver una versión simplificada de la misma.

La sentencia **SELECT** nos permitirá recuperar los datos de los objetos almacenados, pero se recuperarán como si de una tabla se tratase, organizada por filas y columnas, donde cada fila contendrá los datos de un objeto diferente y cada columna el atributo extraído del objeto. Una sentencia **SELECT** tiene la siguiente forma básica:

SELECT atributos FROM clases u tablas de objetos WHERE condiciones

La sentencia se divide de forma básica en tres partes:

- ✓ Parte **SELECT** (obligatoria): donde se indican los atributos a extraer de las clases o de las tablas de objetos.
- ✓ Parte **FROM** (obligatoria): donde se indican las clases (en Caché) o las tablas de objetos (en Oracle) sobre los que se realizará la consulta.
- ✓ Parte **WHERE** (opcional): donde se indican las condiciones que deben cumplir los elementos a extraer, si no cumplen la condición, no se extraen.



Veamos ahora, parte por parte. Empezaremos por la primera parte, la parte **SELECT** (recuerda que para ejecutar las sentencias vía JDBC no se ponen los puntos y coma finales):

- ✓ Podemos recuperar uno o más atributos a la vez. Para indicar más de un atributo simplemente se separan por comas:
 - Ejemplo para Caché: "SELECT **Apellidos, Nombre** FROM Conductor".
 - Ejemplo para Oracle: "SELECT apellidos, nombre FROM conductores;"
- ✓ Los atributos a recuperar pueden tener un alias, de forma que de cara a los resultados obtenidos después de la consulta, el nombre de la columna será el alias, y no el nombre del atributo. El alias se pone indicando el nombre del alias al lado del atributo:
 - Ejemplo para Caché: "SELECT Apellidos **APE**, Nombre **NOM** FROM Conductor".
 - Ejemplo para Oracle: "SELECT apellidos **APE**, nombre **NOM** FROM conductores;"
- ✓ Los atributos a recuperar pueden contener expresiones de diferente tipo, combinando diferentes campos entre sí. Una de las expresiones a utilizar es la concatenación de cadenas, que se realiza con el operador "||", tanto en Oracle como en Caché. Una cadena se introduciría entre comillas simples, como ya se comentó en apartados anteriores:
 - Ejemplo para Caché: "SELECT **Apellidos || ', ' || Nombre NombreYApellidos** FROM Conductor".
 - Ejemplo para Oracle: "SELECT **apellidos || ', ' || nombre NombreYApellidos** FROM conductores;"
- ✓ También es muy común usar expresiones matemáticas en vez de indicar un atributo. Las expresiones matemáticas que podemos usar pueden combinarse con atributos numéricos, y su sintaxis es igual que en Java, pudiendo usarse de paréntesis, realizar sumas, restas, multiplicaciones y divisiones (la operación módulo no está disponible):
 - Ejemplo para Caché: "SELECT **5*(PuntosCarnet+2)** FROM Conductor".
 - Ejemplo para Oracle: "SELECT **5*(puntos_carnet+2)** FROM conductores;"
- ✓ También se pueden recuperar todos los atributos de la clase o de la tabla de objetos de un golpe, sin indicarlos uno a uno. Para realizarlo se usa el símbolo "*" en vez de indicar atributo por atributo:
 - Ejemplo para Caché: "SELECT ***** FROM Conductor".
 - Ejemplo para Oracle: "SELECT ***** FROM conductores;"
- ✓ Y en Caché, toda clase esconde un atributo llamado %ID, que es el OID de cada instancia de objeto almacenada (existirá aunque no sea indicado). Es muy útil poder obtenerlo a través de una sentencia **SELECT**, dado que nos permitirá usar la proyección Java para operaciones de borrado y actualización.
 - Ejemplo para Caché: "SELECT **%ID, Nombre, Apellidos** FROM Conductor".



Autoevaluación

En la parte `SELECT` de la sentencia `SELECT` pueden indicarse todos los atributos a extraer, o ninguno si no deseamos extraer ninguno, de los objetos almacenados en la base de datos. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

8.1.1.- Recuperación de los objetos almacenados. (II)

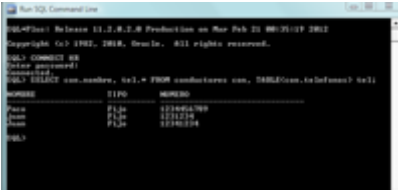
¿Y cómo sería la parte **FROM** de la sentencia **SELECT**? La parte **FROM** permite indicar sobre que tablas de objetos o clases se va a trabajar, permitiendo incluso combinar varias tablas de objetos o varias clases entre sí, pero no vamos a entrar en tanto detalle (aunque un poco sí). Veamos cómo sería:

- ✓ Podemos indicar que los datos se sacaran de una clase o tabla de objetos concreta, y podemos poner también un alias de dicha tabla poniendo el nombre del alias al lado del nombre de la tabla:
 - Ejemplo para Caché: "SELECT * FROM **Conductor** con".
 - Ejemplo para Oracle: "SELECT * FROM **conductores** con;"
- ✓ Una vez fijados los alias de las clases o de las tablas de objetos, podemos usar notación de punto (Alias.Atributo) para indicar que atributos retornaría mi consulta:
 - Ejemplo para Caché: "SELECT con.Nombre FROM **Conductor** con".
 - Ejemplo para Oracle: "SELECT con.nombre FROM **conductores** con;"

En el caso de Oracle, la parte **FROM** cobra especial relevancia cuando se trata de acceder a tablas anidadas. Una tabla anidada es una tabla dentro de otra tabla, por lo que sería un caso en el que se combinarían varias tablas. En el ejemplo siguiente se mostrarían todos los atributos de la tabla anidada teléfonos (tipo de teléfono y número), junto al nombre del conductor o la conductora:

- ✓ Ejemplo para Oracle: "SELECT con.nombre, tel.* FROM **conductores** con, **TABLE(con.telefonos)** tel;"

En el ejemplo anterior se pone **TABLE** y entre paréntesis la tabla anidada en notación de punto (con.telefono), para indicar que la tabla conductores se va a combinar con la tabla anidada. Al combinarla es como si tuviéramos una única tabla que contendría cada conductor o conductora repetido tantas veces como teléfonos tenga. Si Juan tiene un teléfono fijo y un teléfono móvil, aparecerían dos filas, con el nombre repetido, y con distinto teléfono.



```
SQL> SELECT con.nombre, tel.* FROM conductores con, TABLE(con.telefonos) tel;
```

nombre	tipo	numero
Juan	Fijo	111111111
Juan	Movil	2222222

Para acceder a los elementos almacenados en un **VARRAY** podemos usar la siguiente notación (parecida a la anterior, pero más compleja):

```
SELECT COLUMN_VALUE as mail FROM TABLE  
(SELECT mails FROM conductores WHERE DDI='99999F' AND mails IS NOT NULL);
```

En el ejemplo anterior se usa también **TABLE**, pero en su interior hay una consulta tipo **SELECT**. Dicha sub-consulta **SELECT** debe generar el **VARRAY** de un único objeto, y además su resultado debe ser no nulo (si es nulo, fallaría la consulta). El nombre de la columna con los datos se llamaría "**COLUMN_VALUE**" a la cual conviene poner un alias. Poniendo **TABLE** realmente lo que hacemos es convertir cualquier consulta en tabla, por lo que el caso anterior también se podría usar también para extraer por ejemplo los teléfonos de un único conductor:

```
SELECT tel.tipo, tel.numero FROM TABLE  
(SELECT telefonos FROM conductores  
WHERE DDI='99999F' AND mails IS NOT NULL) tel;
```

La diferencia entre este caso y el del **VARRAY**, es que en el caso de las tablas anidadas los telefonos si tienen nombres para los atributos almacenados, dado que el valor almacenado es un tipo de dato objeto, concretamente el tipo **tipo_telefono**, por lo que se puede poner "**tel.tipo**" y "**tel.telefono**", cosa que no ocurría en el caso del **VARRAY**, dado que almacenaba cadenas (**VARCHAR2**).

¿Y cómo manejo las listas y los arrays de Caché? En el caso de las colecciones de Caché (listas y arrays), lo mejor y más cómodo es manejarlas desde las proyecciones Java, dado que la sintaxis anterior no es aceptada por Caché.



Autoevaluación

Marca aquellas sentencias que se podrían ejecutar en Caché:

- ☐ `SELECT %ID, Nombre, Apellidos FROM Conductor.`
- ☐ `SELECT * FROM Conductor con.`
- ☐ `SELECT con.nombre, tel.* FROM Conductor con, TABLE(con.telefonos) tel.`

Mostrar Información

8.1.2.- Recuperación de los objetos almacenados. (III)

Otra cosa en la que Oracle y Caché son diferentes, es en lo que respecta a la forma de tratar desde SQL con aquellos atributos que son a su vez un tipo de dato objeto o una clase, como es el caso del atributo **direccion** de la clase **conductor**.

Si recuerdas de apartados anteriores, en Caché dijimos que para estas situaciones íbamos a trabajar con clases tipo serial, es decir, para embeber una clase en otra, íbamos a utilizar clases tipo serial, que son más fáciles de tratar. También dijimos que las clases tipo serial embebidas en otra clase, proyectan sus atributos poniendo el nombre de la clase, guión bajo y el nombre del atributo ("**direccion_nombrevia**"). Imagina que quieres mostrar algún dato de la dirección, por ejemplo, la calle en la que vive un conductor o una conductora. Se haría de la siguiente forma:



- ✓ Ejemplo para Caché: `"SELECT con.Nombre, con.direccion_nombrevia FROM Conductor con"`.
- ✓ Ejemplo para Oracle: `"SELECT con.nombre, con.direccion.nombrevia FROM conductores con;"`.

Fíjate que en Caché se pondría como "**direccion_nombrevia**" y en Oracle como "**direccion.nombrevia**". En Oracle se usa un punto para indicar que se trata de un atributo dentro de un tipo de dato objeto, mientras que en Caché se pone un guión bajo. Esto es así porque en Caché **Direccion** es tipo serial, si no fuera tipo serial se pondría de otra forma.

Tanto Caché como Oracle disponen además de un catálogo de funciones que se pueden usar para "transformar" digamos un atributo. Funciones que permiten muchas cosas, desde convertir una fecha a un formato legible por el ser humano, a calcular el valor absoluto de un número. Veamos un caso típico y no trivial, que es precisamente convertir una fecha a formato legible por un ser humano:

- ✓ Ejemplo para Caché: `"SELECT TO_CHAR(FechaNacimiento, 'DD/MM/YYYY') FROM Conductor"`.
- ✓ Ejemplo para Oracle: `"SELECT TO_CHAR(fecha_nacimiento, 'DD/MM/YYYY') FROM conductores;"`.

El ejemplo anterior usa la función **TO_CHAR** para convertir la fecha de nacimiento en un formato concreto, concretamente el formato europeo de fecha (dos dígitos para el día, dos para el mes y cuatro para el año, separados por barras).

Para saber más

En el listado siguiente podrás encontrar un compendio de las funciones disponibles en Caché:

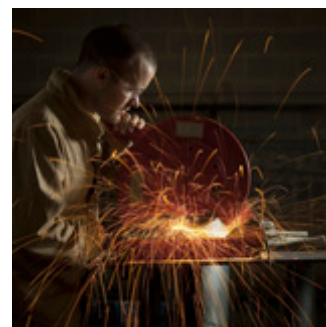
[Funciones disponibles en el SQL de Caché.](#)

Y en el siguiente enlace, tienes un compendio de funciones para el SQL de Oracle.

[Funciones disponibles en el SQL de Oracle.](#)

8.1.3.- Recuperación de los objetos almacenados. (IV)

¿Y cómo sería la parte **WHERE** de las sentencias **SELECT**? La parte **WHERE** nos permite delimitar el conjunto de objetos de los que se recuperará información. Veamos como utilizarla:



- ✓ ¿Podríamos mostrar solo los conductores y las conductoras que tuvieran más de 10 puntos? Pues si. Lo haríamos a través de un operador de comparación numérico. Podemos especificar en la sentencia **SELECT** que dicho atributo debe ser mayor (>), menor (<), igual (=), mayor o igual (>=), menor o igual (<=) o distinto (<>) que otro número o que otro atributo. Los comparadores son iguales que en Java salvo el comparador de distinto que pasa a ser "<>" en lugar de "!=". Veamos como sería:
 - Ejemplo para Caché: "SELECT Nombre,PuntosCarnet FROM Conductor WHERE PuntosCarnet>10".
 - Ejemplo para Oracle: "SELECT nombre, puntos_carnet FROM conductores where puntos_carnet>10;"
- ✓ Con las cadenas de texto se puede usar el operador igual para verificar que una cadena contiene un valor concreto (recuerda, en SQL se usan comillas simples para las cadenas):
 - Ejemplo para Caché: "SELECT Nombre, PuntosCarnet FROM Conductor where Nombre='Juan'".
 - Ejemplo para Oracle: "SELECT nombre, puntos_carnet FROM conductores where Nombre='Juan';".
- ✓ Cuando necesitamos hacer una búsqueda imprecisa de una cadena, porque solo conocemos parte de la cadena, podemos usar el comparador "like". El comparador like permite indicar que un atributo debe parecerse a un patrón. El patrón es muy simple, se pone porcentaje ("%") para decir que puede haber cualquier número de caracteres y guión bajo ("_") para decir que puede haber un único carácter:
 - Ejemplo para Caché: "SELECT Nombre, PuntosCarnet FROM Conductor where Nombre like '%uan'".
 - Ejemplo para Oracle: "SELECT nombre, puntos_carnet FROM conductores where Nombre like '%uan';".
- ✓ La fechas también se puede comparar usando los comparadores numéricos. Esto permitirá comprobar si una fecha es posterior o anterior a otra. En el ejemplo siguiente se indica que la fecha de nacimiento del conductor o de la conductora debe ser posterior a 1 de enero de 1976. La función **TO_DATE** se utiliza para convertir una fecha en formato legible por el ser humano, a formato interno de la base de datos.
 - Ejemplo para Caché: "SELECT Nombre, FechaNacimiento FROM conductor WHERE FechaNacimiento>TO_DATE('01/01/1976','DD/MM/YYYY')".
 - Ejemplo para Oracle: "SELECT nombre, fecha_nacimiento FROM conductores WHERE fecha_nacimiento>TO_DATE('01/01/1976','DD/MM/YYYY');".
- ✓ Las condiciones "is null" e "is not null" son especialmente útiles cuando lo que se desea comprobar es que un atributo contiene valor, y que no se ha dejado vacío. Su uso es muy simple, "is null" significa "está vacío" e "is not null" significa "no está vacío":
 - Ejemplo para Caché: "SELECT Nombre FROM conductor WHERE PuntosCarnet is not null;"
 - Ejemplo para Oracle: "SELECT nombre FROM conductores WHERE PuntosCarnet is not null;"

Las condiciones de la sección **WHERE** se pueden agrupar usando operadores **AND** y **OR**, equivalentes a las operaciones Y lógica ("&&") y O lógica ("||") de Java. Se pueden usar paréntesis para agrupar dichas operaciones, y además, se puede usar **NOT** como negación lógica, equivalente a la negación lógica de Java con el operador "!". Veamos un ejemplo:

```
SELECT Nombre, Apellidos, FechaNacimiento, PuntosCarnet FROM Conductor WHERE
PuntosCarnet is not null AND
(PuntosCarnet>10 OR FechaNacimiento>TO_DATE('01/01/1976','DD/MM/YYYY'))
```

En el ejemplo anterior, se recuperan de la base de datos aquellos conductores para los que hay información de puntos de en el carnet y cumplen una de las dos condiciones siguientes: o tienen más de 10 puntos de carnet, o han nacido después del 1 de enero de 1976. El ejemplo sería para Caché, pero en Oracle realizaría de forma similar.



Autoevaluación

Dada la condición " $\text{PuntosCarnet} > 10$ AND $\text{PuntosCarnet} < 5$ ". ¿Qué rango de valores admitiría?

- ☐ Todos los valores entre 10 y 5.
- ☐ Todos los valores que no estén entre 10 y 5.
- ☐ No admitiría ningún valor.

Dicho esto, la estructura de la sentencia **UPDATE** es de la siguiente forma:

En el caso de Caché, lo mejor para manipular las colecciones (listas y arrays asociativos) es usar proyecciones Java, como ya se comentó antes. Para ello, simplemente averiguamos el OID a través de una consulta SQL

(consultando el campo oculto %ID), y cargamos la proyección a través del estático método `_open`, disponible en cada proyección, pasándole como argumento el OID recuperado.



8.3.- Borrado de los objetos almacenados.

El borrado de objetos es lo más sencillo, y lo más peligroso. Para borrar un objeto de la base de datos usamos la sentencia **DELETE**. La sentencia SQL **DELETE** tiene básicamente dos partes:



`DELETE FROM clase o nombre de la tabla WHERE condicion`

Aunque la sentencia **DELETE** puede ser más complicada, el uso que vemos aquí es realmente simple. El término reservado **FROM** es opcional. Habría que indicar por un lado la tabla de objetos o la clase de la que se eliminarán objetos, y por otro, una condición. Se trata de eliminar todos aquellos registros que cumplan una condición determinada. La condición será del mismo tipo a la usada para la sentencia **SELECT**. Por ejemplo:

- ✓ Ejemplo para Caché: `"DELETE Conductor WHERE Nombre like '%uan'"`.
- ✓ Ejemplo para Oracle: `"DELETE conductores WHERE nombre like '%uan';"`.

La condición puede ser todo lo compleja que queramos, pero en el caso anterior borraría todos los registros cuyo nombre terminara en "uan". Esta forma de eliminar objetos funcionará en Caché, y obviamente también en Oracle.

Nuevamente aquí hay que tener mucho cuidado, porque es fácil borrar objetos que no queramos si usamos condiciones poco finas. Es conveniente siempre usar un atributo que sea llave primaria, para asegurar que estamos eliminando justo el objeto que queremos:

- ✓ Ejemplo para Caché: `"DELETE Conductor WHERE DNI='12345C'"`.
- ✓ Ejemplo para Oracle: `"DELETE conductores WHERE DNI='12345C';"`.

En Caché también puedes usar la proyección Java de las clases Caché para eliminar objetos de la base de datos. Su eliminación es muy sencilla. Para ello, cada proyección Java tendrá un método estático llamado `"_delete"` al que se le pasará como parámetro el OID del elemento a eliminar.



Autoevaluación

¿Cuál de estas sentencias borraría de la base de datos aquellos conductores y aquellas conductoras que tienen menos de 10 puntos en el carnet?

- ☐ `DELETE * FROM Conductor WHERE PuntosCarnet>10.`
- ☐ `DELETE Conductor WHERE PuntosCarnet<=10.`
- ☐ `DELETE FROM Conductor WHERE PuntosCarnet<10.`
- ☐ Ninguna de las anteriores.

Para saber más

En el siguiente enlace encontrarás más información acerca del SQL empleado en base de datos Caché.

[Guía del SQL usado por Caché.](#)

8.4.- Borrar y modificar objetos desde JDBC y proyecciones.

En apartados anteriores se han explicado las sentencias SQL de eliminación y modificación, veamos ahora rápidamente como sería su uso desde JDBC. Su ejecución desde JDBC es simple, se usan los interrogantes para indicar que en una posición determinada se va a pasar un parámetro a través de los métodos set de la clase **PreparedStatement**. Veamos un pequeño ejemplo de uso de la sentencia **UPDATE** a través de JDBC. El ejemplo propuesto correspondería con el esquema de ejemplo para Caché, en el caso de Oracle sería igual, pero cambiando el nombre de los atributos y de las tablas (**conn** corresponde con una instancia de la clase **JDBC Connection**):



```
SQLu = "UPDATE Conductor SET Nombre=? WHERE DDI=?";
PreparedStatement ps=conn.prepareStatement(SQLu);
ps.setString(1,"Juan"); // Establecemos el parámetro el nuevo valor del nombre a través
ps.setString(2,"12345C"); // Indicamos el DNI del conductor a modificar a través también
int objetos_modificados=ps.executeUpdate();
```

El ejemplo anterior ya te debe ser más o menos conocido. Como se dijo en apartados anteriores, se pueden usar parámetros (interrogantes) con las sentencias **UPDATE** y **DELETE**, y su ejecución correría a cargo del método **executeUpdate** de la clase **PreparedStatement**. Usarlo para sentencias tipo **DELETE** es por tanto, muy parecido. Para conocer el número de objetos afectadas por una modificación (o una eliminación) el método **executeUpdate** retornará un número entero que contendrá el número de objetos modificados (o eliminados en caso de una sentencia **DELETE**).

Un detalle importante es que en las sentencias **UPDATE**, cuando hablamos de Oracle, se pueden usar los métodos **setArray** y **setObject** para pasar como parámetro una colección entera o un objeto respectivamente. Eso nos evita tener sentencias **UPDATE** excesivamente complicadas, donde se usan muchos constructores. Si quieres recordar el uso de los métodos **setArray** y **setObject** echa un vistazo de nuevo al apartado de almacenamiento de objetos.

En el caso de Caché, la modificación y el borrado de elementos también se puede realizar a través de las proyecciones Java. Para ello, es necesario conocer de antemano el OID del objeto u objetos a modificar o eliminar. Podemos averiguar el OID a través del campo oculto **%ID**, en el cual Caché almacenará el OID de cada objeto. Veamos como sería la consulta para extraer el OID, la cual es ya conocida de apartados anteriores:

```
String SQLd="SELECT %ID FROM Conductor WHERE DDI=?";
PreparedStatement ps=conn.prepareStatement(SQLd);
ps.setString(1,"12345C");
ResultSet rs=ps.executeQuery();
```

En el código anterior se ejecuta una consulta **SELECT** para recuperar el OID de un objeto almacenado. En nuestro caso, la consulta anterior generaría un único resultado, dado que la columna **DDI** es clave primaria y no se puede repetir, pero no obstante, siempre conviene chequear que solo se haya obtenido un único OID (necesitamos exactamente uno):

```
rs.last(); // Nos movemos a la última fila de la tabla.
String oid=null;
if (rs.getRow()>1) { System.out.println("Error: existe más de un campo con el mismo DDI");
else {
    if (rs.first()) oid=rs.getString(1);
    else System.out.println("Error: No existe ningún objeto para el DDI proporcionado.'");
```



```
}
```

En el código anterior, se usa el método **last()** de la clase **ResultSet**, que permite desplazarse a la última fila del resultado obtenido. Después se usa el método **getRow()** de la misma clase, para obtener el número de fila sobre la que se está apuntando en ese momento. Si **getRow()** retorna un valor mayor de 1, después de habernos movido a la última fila, es que tenemos más de un resultado, lo cual no es válido, dado que no sabríamos que OID eliminar de entre todos los obtenidos.

Después se usa la función **first()** de la clase **ResultSet** para movernos a la primera fila. Dicha función retornará **true** si existe una primera fila, y **false** si la tabla está vacía. De esa forma nos aseguramos que efectivamente el resultado de la consulta SQL ha permitido recuperar el OID del objeto buscado. Una vez obtenido el OID, ya podemos eliminar el objeto a través de la proyección, usando el método estático “**_delete**” que encontraremos en cada proyección:

```
if (oid!=null) Conductor._delete (conn,new ID(oid));
```

Eso sí, tendrás que usar el método **_delete** de la proyección correspondiente al objeto a eliminar, no podrás usar cualquier proyección. El método **_delete** tiene exactamente los mismos argumentos que el método estático **_open**, disponible también en las proyecciones Java de los objetos Caché, pero su objetivo es diferente, dado que el método **_open** lo usamos para abrir y modificar una objeto almacenado, tal y como se explico en apartados anteriores. Veamos como sería:

```
if (oid!=null) Conductor c=(Conductor)Conductor._open(conn,new ID(oid));
```

El método estático **_open** devolverá una instancia de la clase **com.intersys.classes.RegisteredObject**, sobre la que habrá que hacer una conversión de tipos.



Anexo I.- Resumen de la sintaxis de ObjectScript.

No es el objetivo de este documento que aprendas del todo un nuevo, completo y complejo lenguaje de programación como es ObjectScript, pero sí que lo conozcas mínimamente para saber hacer algunas tareas básicas. La idea es que seas capaz de almacenar datos en una base de dato orientada a objetos, y que si necesitas hacer un pequeño método que se ejecuta en la base de datos, que puedas hacerlo sin problemas. Por lo que vamos a analizar las características principales de ObjectScript:

- ✓ Se basa en el lenguaje estándar M, pero con una sintaxis más relajada.
- ✓ Los bloques de código son definidos por llaves, como en Java.
- ✓ Dentro de un método, el uso de espacios es clave:
 - ➔ Hay que dejar al menos un espacio delante de cada sentencia.
 - ➔ Entre ciertos elementos del código hay que dejar un único espacio (detrás de los comandos). No es lo mismo poner "Set e" con un único espacio entre la palabra reservada "Set" y la variable "e", que poner dos espacios en medio. Si pones dos o más espacios en este caso dará error al compilar.
- ✓ Las líneas no acaban en punto y coma como en Java, simplemente, se pone salto de línea y listo.
- ✓ Las variables son de tipado débil, esto quiere decir que no se declara el tipo de la variable como en Java, sino que la variable asume el tipo que se le asigna en el momento de la asignación.
- ✓ Un método puede retornar o no un dato. Cuando no se retorna un dato puede finalizarse la ejecución del método en cualquier momento con la sentencia "Quit". Cuando se retorna un dato, este se debe retornar poniendo "Quit" y detrás el nombre de la variable con el dato a retornar, por ejemplo: "Quit e". Dado que ObjectScript es de tipado débil, hay que asegurarse de que la variable con el valor a retornar contiene un dato del tipo adecuado (por ejemplo, dará error al ejecutarse si se espera que retorne un número entero y la variable retornada contiene una cadena).
- ✓ Los comentarios son del estilo a Java, iguales, salvo que existen otros tipos de comentarios adicionales, que solo se pueden usar dentro del bloque de código de ObjectScript.

Veamos brevemente cómo es la sintaxis de ObjectScript, partiendo de como se diferencia con respecto a Java. Imagina que tienes la siguiente clase de partida:

```
/// Clase conductor.
Class User.Prueba Extends %Persistent [ ClassType = persistent ]
{
    Property dato As %Integer [ Required ];
    Method ponerdato()
    {
        Set ..dato=$RANDOM(10)+1
    }
    Method obtenerdato() As %Integer
    {
        Quit ..dato
    }
}
```

En el ejemplo anterior tienes dos métodos (**ponerdato** y **obtenerdato**) y una propiedad (**dato**). El resto, vamos a analizarlo a partir de ahora. Lo primero, veamos las 4 formas de poner comentarios:

```
// Modelo de comentario número 1, solo una línea.
; Modelo de comentario número 2, solo una línea.
;; Modelo de comentario número 3, solo una línea.
#; Modelo de comentario número 4, solo una línea.
/* Modelo de comentario multilínea. */
```

Como puedes observar, poner un comentario es sencillo, se hace igual que en Java, aunque hay más opciones. Ahora vamos a ver como se hace una simple asignación. La asignación se hace con el operador **Set**, el cual permite asignar un nuevo valor a una variable, y si la variable no está creada, pues la crea. No se indica el tipo, dado que es un lenguaje de tipado débil, y entre **Set** y el nombre de la variable solo puede haber un único espacio. Veamos un ejemplo:

```
Method obtenerdato() As %Integer
{
    Set a = 10
    Quit ..dato*a
}
```

Las operaciones aritméticas y el uso de paréntesis es igual que en Java, con dos excepciones: que la potencia se pone con dos asteriscos seguidos ("**"), por ejemplo **a**b** significaría **a** elevado a **b**, y que el módulo se pone con la almohadilla("#"), por ejemplo **a#b** es el módulo de la división **a** entre **b**.

```
Set b = (a+b)*c
Set b = a**2 // A elevado a 2
Set b = a#2 // A módulo 2
```

Las cadenas son iguales que en Java, salvo que para concatenar cadenas se usa el guión bajo (en vez de el símbolo de suma), y que para poner unas comillas en la cadena no usamos la barra inclinada como carácter de escape ("\ cantenar con variables numéricas, de forma que dichas variables se transformarán a cadena de texto. Veamos un ejemplo:

```
Set a = 10
Set c = "ASDF"_"ASDF" // Concatenar cadenas, resultado ASDFASDF
Set c = "ASDF""ASDF" // Poner unas comillas en una cadena, resultado ASDF"ASDF
Set d = "ASDF"_a // Concatena a ASDF la variable a, dando lugar a la cadena ASDF10
```

En Caché hay definidas un montón de funciones globales, que son muy útiles. Las funciones globales siempre deben llevar el signo del dolar("\$") antepuesto al nombre. Es el caso de la función **\$RANDOM** que genera un número aleatorio entre 0 y el número justamente anterior al que se pasa por parámetro (en el ejemplo siguiente sería un número entre 0 y 9, al que se le suma uno para que el rango pase a ser de 1 a 10):

```
Method multiplicarporealeatorio() As %Integer
{
    Set e=$RANDOM(10)+1 // Valor aleatorio entre 1 y 10
    Quit ..dato*e
}
```

Veamos ahora cómo se realiza la evaluación de condiciones en ObjectScript, esto si que es diferente a Java en algunos aspectos, veamoslo:

- ✔ **Comparación de igualdad.** En Java es con dos iguales ("==") pero ObjectScript es con un solo igual ("=").
- ✔ **Comparación menor que.** Se realiza igual que en Java, con el símbolo menor que ("<").

- ✓ **Comparación menor o igual que.** Igual que en Java, con los símbolos menor que seguido de igual (" \leq ").
- ✓ **Comparación mayor que.** Igual que en Java, con el símbolo de mayor que (" $>$ ").
- ✓ **Comparación mayor o igual que.** Igual que en Java, con los símbolos mayor que seguido del símbolo igual (" \geq ").
- ✓ **Comparación de distinto a.** Muy diferente a Java, en ObjectScript es con el símbolo de apóstrofo seguido de igual (" \neq ").
- ✓ **Comparación "no menor que".** No existe en java, al menos de esta forma, se indica poniendo un apóstrofo seguido del símbolo de menor que (" $<$ "). "No menor que" es lo mismo que decir o "mayor o igual que".
- ✓ **Comparación "no mayor que".** Igual que la anterior, no existe en java, al menos de esta misma forma. Se indicaría poniendo el apóstrofo seguido del símbolo de mayor que (" $>$ "). Decir "no mayor que" es lo mismo que decir "menor o igual que".

Y los operadores lógicos binarios, usados para combinar varias condiciones, o para negar una condición, son casi iguales a Java:

- ✓ Los operadores lógicos Y binaria (" $\&\&$ ") y O binaria (" $\|\|$ ") son iguales que en Java.
- ✓ El operador lógico negación es diferente a Java. La negación es con el apóstrofo, por ejemplo: " $(A < B \ \&\& \ B < C)$ ", mientras que en Java es con el signo de exclamación invertida (" $!(A < B \ \&\& \ B < C)$ ").
- ✓ Se pueden usar paréntesis, igual que en Java, para agrupar comparaciones.

Veamos ahora un ejemplo de cómo se usan las condiciones. El ejemplo siguiente utiliza el operador de control de flujo **If**, que es igual que en Java, salvo por el hecho de que se usa "**Elseif**" (todo junto) en lugar de "**else if**" (separado), para hacer un condicional múltiple. Y otro detalle, en ObjectScript es opcional usar paréntesis para indicar la condición a evaluar, veámoslo:

```
Set a=$RANDOM(5)*2+1 // Número impar del 0 al 10
Set b=$RANDOM(5)*2 // Número par del 0 al 10
If a<b { // Si "a" no es menor que "b", incrementamos "a"
    Set a=a+1
}
Elseif '(a<b && b>5) { // Si no se cumple que "a" sea menor que "b" y "b" mayor que 5
    Set a=a+2
}
Else { // En cualquier otro caso
    Set a=a+3
}
```

Veamos ahora cómo es la sentencia de control de flujo "mientras" en ObjectScript. El bucle mientras es igual que en Java, salvo obviamente por el hecho de que los operadores de condición (menor que, mayor que, etc.) cambian para algunos casos, detalle explicado en párrafos anteriores. Veamos un ejemplo, en el siguiente trozo de código se suman todos los números entre 1 y 10, para lo cual se usa un bucle mientras:

```
set a=10
set b=0
while (a<10)
{
    set b=b+a
    set a=a-1
}
```

Si el bucle mientras es igual que en Java, el bucle **for** es completamente diferente. Se pone con la sintaxis siguiente:

```

For variable = valorinicial:incremento:valorfinal {
    // Código a ejecutar en cada iteración
}

```

En el bucle **for** anterior **valorinicial** será el valor de comienzo, **incremento** será el incremento de valor en cada iteración del bucle, y **valorfinal** obviamente será el valor de salida del bucle. Veamos un ejemplo que hará exactamente lo mismo que el bucle mientras anterior (sumar los números del 1 al 10):

```

set b=0
For a =1:1:10 {
    write a
}

```

Ya estamos casi acabando este pequeño resumen. Es importante que sepas que cuando una variable ya no se usa, se puede eliminar con el operador "**kill**", pudiendo indicar varias variables separadas por comas:

```

Method CalcularSuma(inicio As %Integer, fin As %Integer) As %Integer
{
    Set suma=0
    Set contador=inicio
    while (contador>=inicio && contador<=fin)
    {
        set suma=suma+contador
        set contador=contador+1
    }
    Kill contador
    Quit suma
}

```

Y otra cosa muy útil, con la función **\$DATA** podemos determinar si una variable existe o no, es decir, si se ha creado antes:

```

if $DATA(a) {
    write "Variable existente"
}
else
{
    write "Variable no existe"
}

```

Y por último, veamos cómo se accede a otros métodos de la clase (es decir, como se invocan), y como se puede obtener y cambiar el valor de los atributos de la clase. Para invocar otro método de la clase, se distinguen dos

casos. Primer caso: el método no retorna nada, o simplemente no nos interesa lo que retorna. Para ejecutarlo se usa el operador “do” (después del operador **do** solo puede haber un único espacio):

```
Do ..ponerdato()
```

Fijate en la notación “punto punto”, se anteponen dos puntos (“.”) delante de los métodos y de los atributos de la clase. De esa forma Caché sabe que se trata de un método o de un atributo a nivel de clase. Veamos ahora el segundo caso: el método retorna un valor que además nos interesa usar. En ese caso no se usa el operador **do**, se realizaría de la siguiente forma:

```
Set a=a+..edad("A","B")
```

Y por último, y no menos importante, para acceder a una propiedad o atributo de la clase también se usa la notación de punto punto (como ya habrás imaginado), veamos un ejemplo:

```
Set a=a+..dato // Suma a la variable "a" el valor de la propiedad de clase "dato"
```

Este ha sido un pequeño y breve compendio de la sintaxis de ObjectScript, lo suficiente como para poder hacer tus pequeños experimentos.



Debes conocer

En el siguiente enlace podrás conocer, con un poco más de profundidad, las sintaxis de ObjectScript. Eso sí, antes de hacer clic, coge un poco de aire:

[Sintaxis de ObjectScript.](#)

Y en el siguiente enlace podrás buscar funciones preexistentes en Caché, para su uso en métodos programados con ObjectScript por ti:

[Funciones preexistentes en ObjectScript.](#)

Anexo II.- Ejemplo de utilización de colecciones en Caché.

A continuación se muestra la clase Socio, que implementa dos colecciones, una lista de mails y un array de teléfonos. En el siguiente código se muestran sendos ejemplos de métodos en **ObjectScript** que utilizan las listas y los arrays de Caché:

```
Class User.Socio Extends %Persistent
{
    // Nombre del socio
    Property nombre As %String [ Required ];
    // Apellidos del socio
    Property apellidos As %String [ Required ];
    // Número de socio.
    Property NumSocio As %String [Required];
    // Lista con los mails.
    Property Mails As list Of %String;
    // Lista con los telefonos.
    Property Telefonos As array Of %String;

    /* Ejemplo A.1 de uso de colecciones tipo listas de Caché.
    Inserta un nuevo mail en la list de mails. */
    Method insertarMail(mail As %String)
    {
        /* El método Insert de las listas permite insertar un nuevo elemento del tipo
        especificado en la lista, se insertará al final, dado que las listas no son
        ordenadas. */
        do ..Mails.Insert(mail)
    }

    /* Ejemplo A.2 de uso de colecciones tipo listas de Caché.
    Hacer un recuento de los mails almacenados en la lista. */
    Method contarMails() As %Integer
    {
        /* El método Count de las listas permiten obtener el número de elementos
        que contienen */
        Quit ..Mails.Count()
    }

    /* Ejemplo A.3 de uso de colecciones tipo listas de Caché.
    Obtener el mail almacenado en una posición concreta de la lista.
    Si dicho mail no existe se retornará cadena vacia (""), también conocido en Caché como
    null string. Tienes que tener en cuenta que las posiciones se empiezan a numerar por 1
    y no por 0.*/
    Method obtenerMail (pos As %Integer) As %String
    {
        /* Primero se comprueba si la posición indicada, donde está el elemento a
        obtener, existe o no en la lista. Para ello, hacemos uso del método contarMails
        antes implementado. */
        if pos<=..contarMails() && pos>=1
        {
            /* El método GetAt permite obtener un mail almacenado en una posición concreta
```

```

        Su único argumento es la posición a obtener. Tienes que tener en cuenta
        que las posiciones se empiezan a numerar por 1, y no por 0. */
        Quit ..Mails.GetAt(pos)
    }
Else /* Si el mail no existe, se retorna cadena vacía (""), también llamado null ;
{
    Quit ""
}
}

/* Ejemplo A.4 de uso de colecciones tipo listas de Caché.
Eliminar un mail. Retornará 1 (verdadero) se ha eliminado, y 0 (false) en otro caso.
Se le pasa como parámetro la posición del elemento a eliminar, que se empieza a numerar
por 1 y no por 0.
*/
Method eliminarMail(pos As %Integer) As %Boolean
{
    Set eliminado=0
    /* Volvemos a hacer uso de la función contarMails para saber si la posición
    indicada está o no dentro de los límites de almacenamiento de la lista. */
    if pos<=..contarMails() && pos>=1
    {
        /* El método RemoveAt de las listas permite eliminar un elemento
        de una posición concreta. Se le pasa como argumento la posición a eliminar, y
        deberá empezar a contarse en 1 y no en 0. Si consigue eliminarse, entonces
        retornará el elemento eliminado, en caso contrario retornará cadena vacía ("")
        también llamado null string. */
        Set result=..Mails.RemoveAt(pos)
        if result="" { Set eliminado = 0 }
        else { Set eliminado = 1 write !,"Eliminado:",result }
    }
    Quit eliminado
}

/* Ejemplo A.5 de uso de colecciones tipo listas de Caché.
Obtener los mails separados por comas. Usando métodos antes creados se obtiene
una lista separada por comas de los mails.
*/
Method obtenerMails () As %String
{
    Set t=""
    for i=1:1:..contarMails() {
        if i'=1 { Set t=t_", " }
        Set t=t_..obtenerMail(i)
    }
    Quit t
}

/* Ejemplo A.6 de uso de colecciones tipo listas de Caché.
Vaciar la lista de mails. */
Method vaciarMails ()
{

```



```

    /* El método clear permite vaciar una lista, dejandola sin elementos. */
    do ..Mails.Clear()
}

/* Ejemplo B.1 de uso de colecciones tipo arrays de Caché.
Inserción de un telefono y su tipo en el array.
Se trata de poder almacenar el telefono, junto al tipo, por ejemplo: "fax" junto al
número de fax o "movil" junto al teléfono móvil, para así poder diferenciar el tipo
de teléfono.*/
Method insertarTlf (numero As %String, tipotlf As %String)
{
    /* Con el método SetAt se inserta un telefono asociado a una llave, que identificará
    al telefono por el propósito del mismo.*/
    Do ..Telefonos.SetAt(numero,tipotlf)
    Quit
}

/* Ejemplo B.2 de uso de colecciones tipo array de Caché.
Recuperación del telefono de un tipo determinado almacenado en el array. */
Method obtenerTlf (tipo As %String) As %String
{
    /* Se obtiene un elemento etiquetado como "tipo" a través del método GetAt,
    retornando el elemento en cuestión o "" (null string) en caso de que no
    se encuentre. */
    Quit ..Telefonos.GetAt(tipo)
}

/* Ejemplo B.3 de uso de colecciones tipo array de Caché.
Elimina un telefono (por tipo) introducido previamente en el array. */
Method eliminarTlf (tipo As %String) As %Boolean
{
    Set eliminado=0
    /* A través del método isDefined se puede averiguar si en el array hay o no un
    elemento almacenado con una clave asociada concreta. Si es así, retornará 1 (true)
    en caso contrario retornara 0 (false). */
    if ..Telefonos.IsDefined(tipo) {
        /* El método RemoveAt elimina un elemento que tenga una clave concreta,
        si se ha eliminado retornará el elemento eliminado, en caso contrario,
        por ejemplo, si no existiera el elemento, retornaría cadena vacía (""),
        también conocida como null string. */
        Set result=..Telefonos.RemoveAt(tipo)
        If result="" { Set eliminado = 0 } Else { set eliminado=1 }
    }
    Quit eliminado
}

/* Ejemplo B.4 de uso de colecciones tipo array de Caché.
Obtiene la lista de telefonos separadas por comas. Si no hay telefonos, retornará una
cadena vacía (""), también conocido como null string.
*/
Method obtenerTlfs () As %String
{

```

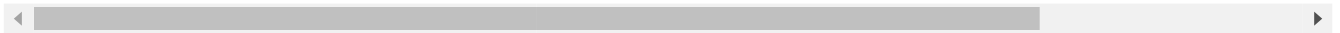
```

Set result=""
/* El método Next, de los arrays permite obtener la primera llave o clave del
primer par llave/valor almacenado, pero solo cuando se le pasa cadena vacía como
parámetro. Si no existiese, retonaría cadena vacía. */
Set tipotlf=..Telefonos.Next("")
while tipotlf="" {
    if result="" { Set result=result_", " }
    /* Con el método GetAt obtenemos el valor asociado a una llave o clave. */
    Set result=result_tipotlf_"_"..obtenerTlf(tipotlf)
    /* El método Next, permite obtener la siguiente llave o clave de la lista
    cuando se le pasa una clave, de esa forma, permite iterar sobre el array,
    como se hace con los iteradores de java. */
    Set tipotlf=..Telefonos.Next(tipotlf)
}
Quit result
}

/* Ejemplo B.5 de uso de colecciones tipo array de Caché.
Vacía la lista de teléfonos usando el método Clear, de forma similar a como se hace
con las listas.*/
Method vaciarTlfs ()
{
    do ..Telefonos.Clear()
}

}

```



Anexo III.- CACHÉ. Inserción de datos en las colecciones usando Proyecciones.

Para Caché, se recomienda utilizar las proyecciones a la hora de insertar datos en las colecciones, así como para manipularlas (modificar, eliminar y consultar los elementos de las colecciones), es mucho más cómodo. Imagina que ya has ejecutado tu consulta SQL de inserción de objetos a través de JDBC (dejando sin rellenar las colecciones obviamente), y que ahora quieres insertar elementos en las colecciones. Para realizar esta operación, recurriremos al OID del elemento recién insertado (**conn** es una instancia de la clase **Connection**):

```
String SQLq="INSERT INTO Conductor (Nombre, Apellidos, FechaNacimiento, PuntosCarnet, di
PreparedStatement ps=conn.prepareStatement(SQLq,Statement.RETURN_GENERATED_KEYS);
ps.setString(1, "Nombre de ejemplo");
ps.setString(2, "Apellidos de ejemplo");
ps.setDate(3, java.sql.Date.valueOf("2000-01-01"));
ps.setInt(4, 15);
ps.setString(5, "Calle Desconocida");
ps.setInt(6, 55);
ps.setString(7,"93939");
ps.setString (8,"12345C");
ps.executeUpdate();
```

En el caso expuesto antes, no se dice nada de las colecciones **Telefonos** ni **Mails**, creadas como array de %**String** en el primer caso, y lista de %**String** en el segundo caso. Una vez que ya se haya ejecutado el método **executeUpdate**, ya podemos obtener el OID del objeto recién almacenado, invocando al método **getGeneratedKeys** de la clase **PreparedStatement**:

```
ResultSet rs=ps.getGeneratedKeys();
rs.next(); String nuevoOID=rs.getString(1);
```

Este OID es muy útil, especialmente en Caché (aunque en realidad, para Caché lo que se obtiene con las sentencias anteriores no es el OID, sino otro tipo de identificador que al fin y al cabo nos va a servir para lo mismo). Y ahora que lo tenemos, veamos como añadir información a los atributos tipo colección usando proyecciones. Veremos primero como crear un nuevo objeto Caché a través de las proyecciones, y luego, como usar un objeto ya existente, a través de su OID.

Crear un nuevo objeto Caché, como por ejemplo un nuevo **Conductor**, a través de proyecciones Java es relativamente fácil. Imagina que has incorporado las proyecciones de **Conductor** y **Direccion** a tu proyecto, y que "**conn**" contiene una instancia válida de la clase **Connection**, la cual contiene toda la información necesaria para conectarse a Caché.

Crear un nuevo **conductor** sería tan fácil como lo siguiente:

```
Conductor c=new Conductor(CacheDatabase.getDatabase(conn)); // Creamos un nuevo objeto
c.setNombre("Nombre"); // Establecemos el valor de los atributos necesarios.
c.setApellidos("Apellidos");
c._save(); // Guardamos.
```

En el código anterior se ha creado una nueva instancia de la clase **Conductor**, invocando al constructor de la proyección Java. A dicho constructor se le pasa como parámetro una conexión a Caché, pero no exactamente una instancia de la clase **Connection**, sino algo parecido que se puede obtener fácilmente partiendo de una conexión JDBC existente. Después se establece el valor de los atributos y se guarda el objeto invocando el método estático “**_save()**”, existente en cualquier proyección.

Para recuperar un objeto almacenado en Caché y así poder usarlo a través de su proyección, necesitamos el OID del objeto. Cargar un objeto y manipularlo directamente desde su proyección facilita enormemente el trabajo con los tipos de dato colección. Veamos como podríamos ahora cargar un conductor existente y manipular rápidamente las colecciones que contienen los teléfonos y los mails:

```
Conductor c=(Conductor)Conductor._open(CacheDatabase.getDatabase(conn), new Id(0ID));
c.getTelefonos().put("Fijo","3939393");
c.getMails().add("mimail@maildeprueba");
c._save();
```

En el ejemplo anterior se recuperan los datos de un conductor partiendo de su OID a través de la función “**_open**”. El método estático **_open** devolverá una instancia de la clase **com.intersys.classes.RegisteredObject**, sobre la que habrá que hacer una conversión de tipos. Fijate que la función **_open** tiene dos parámetros, una conexión a la base de datos, y una instancia de la clase **Id**, a la que se le pasa como parámetro el OID. Después, se añade un teléfono al array de teléfonos, y eso se hace como si se tratara de un mapa de Java (**java.util.Map**). Y lo mismo pasa con la lista de mails, se utiliza como si se tratara de una lista de Java (**java.util.List**). Tan fácil como eso.

ORACLE. Inserción de datos en las colecciones usando JDBC.

Y ahora, nos ponemos de nuevo con Oracle. Comparativamente, manejar las colecciones en Oracle te va a ser un poco más difícil que en Caché, dado que en Oracle no usaremos proyecciones. ¿Cuánto de difícil? Vamos a averiguarlo.

El problema se produce, de cara a usar la sentencia **INSERT**, en el momento en el que tienes una cantidad de datos variable a almacenar en las colecciones. Cuando tienes una cantidad fija de datos, por ejemplo, exactamente dos mails, podemos usar constructores anidados, pero cuando un objeto puede tener dos mails, y otro, tres, no es tan fácil. Esto se puede resolver desde JDBC, pero es un poco lioso.

Lo primero ahora es crear una sentencia **INSERT** que permita insertar colecciones a través de JDBC (esta sentencia **INSERT** está pensada para el esquema de ejemplo que os hemos facilitado):

```
String SQLq = "INSERT INTO CONDUCTORES
VALUES (TIPO_CONDUCTOR(?,?,?,?,?,?,?,TIPO_DIRECCION
(NULL,?,?, NULL,?, NULL, NULL, NULL, NULL)))";
```

En el ejemplo anterior se usan constructores a varios niveles. Los interrogantes de las posiciones 5 y 6, corresponden con las posiciones del VARRAY de mails y de la tabla anidada de teléfonos, respectivamente.

Los valores correspondientes a tipos de datos colección, y los correspondientes a objetos (como es el caso del **tipo_direccion**), pueden pasarse directamente a través de JDBC, como si fuera un bloque único. Esto nos permitirá indicar varios teléfonos o varios mails de una tajada, es decir, podremos insertar varios elementos en las colecciones, sin tener que modificar sustancialmente la sentencia SQL.

Al igual que existen los métodos “**setString**” o “**setInt**” en la clase **PreparedStatement**, existen los métodos “**setArray**” y “**setObject**”. **setArray** permite indicar que un interrogante de la sentencia SQL será una colección (ya sea VARRAY o tabla anidada), y **setObject** permite indicar que un interrogante es un objeto (por ejemplo **tipo_telefono**). También podría haberse usado en el caso de **tipo_direccion**, pero no se ha puesto para no complicar más todavía el ejemplo.

Para indicar que un parámetro es un objeto tenemos que crear primero un descriptor de estructura del tipo de dato objeto. Esto se realiza a través del método estático “**createDescriptor**” de la clase **StructDescriptor**,

proporcionada a través del paquete **oracle.sql.*** (esta clase debe proporcionarla de forma independiente cada SGDB). Sería de la siguiente forma (**conn** sigue siendo una instancia de la clase **Connection**):

```
StructDescriptor sd_tipo_telefono = StructDescriptor.createDescriptor  
("TIPO_TELEFONO", conn);
```

La instancia anterior de **StructDescriptor** llamada **sd_tipo_telefono** contiene todo lo necesario para crear un objeto **tipo_telefono**, y así, poder guardarlo en la colección **telefonos**. Fíjate que el primer argumento de la función **createDescriptor** es el nombre del tipo de dato en la base de datos, y que el segundo es una conexión válida. Una vez conocida la estructura, ya podemos guardarlas en la base de datos a través de la clase **STRUCT**:

```
String[] s_telefono_fijo={"Fijo","123451234"};  
STRUCT telefono_fijo=new STRUCT(sd_tipo_telefono,conn,s_telefono_fijo);
```

En el código anterior se crea una instancia de la clase **STRUCT** a la que se le pasa como parámetro: el descriptor de **tipo_telefono** (**sd_tipo_telefono**), así como la conexión a la base de datos, y un array de objetos que contendrá todos los atributos (por orden) de dicho tipo objeto. Aquí tienes que tener cuidado, en este caso todos los atributos del tipo **tipo_telefono** son cadenas de texto, si fueran de otro tipo, tendrías que usar el tipo Java equivalente (**Integer**, **Float**, etc.) al usado en la base de datos. Si tienes tipos de datos diferentes (cadenas mezcladas con números), tendrías que usar un array de Objects.

Esta técnica se podría usar para indicar la dirección en la sentencia **INSERT** anterior, en lugar de poner el constructor de tipo de dato objeto **tipo_direccion**, se pondría un único parámetro que se establecería a través del método **setObject**. De esa manera se hubiera reducido el tamaño de la sentencia SQL, pero en contraposición, hubieran aumentado las líneas de código notablemente. Como imaginarás, al método **setObject** se le pasan como argumentos la posición del interrogante a reemplazar, y después, la instancia de la la clase **STRUCT** que contiene el objeto.

Continuando con el ejemplo del apartado anterior, y continuando con Oracle, veamos ahora como indicar a través de JDBC que un parámetro de la sentencia no es solo un objeto, sino una colección completa de objetos.

Partamos de que ya tenemos lo siguiente, visto en párrafos anteriores:

```
StructDescriptor sd_tipo_telefono = StructDescriptor.createDescriptor  
("TIPO_TELEFONO", conn);  
String[] s_telefono_fijo={"Fijo","123451234"};  
STRUCT telefono_fijo=new STRUCT  
(sd_tipo_telefono,conn,s_telefono_fijo);
```

Y ahora, para indicar que un parámetro de una sentencia SQL es una colección, debemos proceder a crear otro descriptor, pero en este caso, el descriptor será para los tipos de dato colección. Primero creamos lo que se denomina un descriptor de array, que será una estructura que contendrá información sobre como es la colección en la base de datos. Para crearla usamos la clase **ArrayDescriptor** (del paquete **oracle.sql.***), de la misma forma que se uso la clase **StructDescriptor**, pasándole como argumento el tipo de dato colección (con el nombre usado en la base de datos, que sería **TABLA_TELEFONOS** para los teléfonos y **ARRAY_MAILS** para los mails) y el conector:

```
ArrayDescriptor desc_tabla_telefonos = ArrayDescriptor.createDescriptor  
("TABLA_TELEFONOS", conn);
```

Una vez hecho esto, tenemos que crear una instancia de la clase **ARRAY** (también del paquete oracle.sql.*). Dicha instancia contendrá la información que se pasará definitivamente como parámetro a la sentencia SQL. Veamos como sería:

```
//Creamos el descriptor para el objeto tipo_telefono
StructDescriptor sd_tipo_telefono = StructDescriptor.createDescriptor ("TIPO_TELEFONO",

//Creamos un objeto tipo_telefono llamada tipo_fijo que ya podremos almacenar en la base
String[] s_telefono_fijo={"Fijo","123451234"};
STRUCT telefono_fijo=new STRUCT(sd_tipo_telefono,conn,s_telefono_fijo);

// Creamos el descriptor para el array de telefonos, donde meteremos el telefono anterior
ArrayDescriptor desc_tabla_telefonos = ArrayDescriptor.createDescriptor("TABLA_TELEFONOS",conn);

// Creamos un array de telefonos, que ahora si se podrá meter en la base de datos, al que le pasamos el descriptor de la tabla
STRUCT[] a_telefonos=new STRUCT[1];
a_telefonos[0]=telefono_fijo;
ARRAY nt_telefonos = new ARRAY(desc_tabla_telefonos,conn,a_telefonos);
```

En el ejemplo anterior se crea una instancia de la clase **ARRAY**, que estará basada en el descriptor del array **desc_tabla_telefonos**. Esa instancia ya podremos insertarla en la base de datos. Para crear la instancia de la clase **ARRAY**, se le pasa como argumento el descriptor de la tabla **desc_tabla_telefonos**, además de la conexión a la base de datos, y los datos que contendrá dicha colección. Los datos a almacenar en la colección se le pasarán como si fueran un array de **Object**, donde cada elemento será una instancia de **STRUCT** creada para el tipo de dato objeto **tipo_telefono**.

Y ahora, ya podemos indicar a la clase **PreparedStatement** cuales serán los datos a almacenar en la tabla anidada. Esto lo haremos con el método **"setArray"**:

```
ps.setArray(6, nt_telefonos);
```

El caso anterior contemplaba las tablas anidadas. Pero cuando se trata de un atributo que es un VARRAY se realiza de la misma forma. Veamos cómo sería para el caso de los mails. Este caso es mucho más sencillo, dado que el VARRAY contiene un tipo de dato básico (**VARCHAR2**) y no un tipo de dato objeto:

```
ArrayDescriptor array_mails=ArrayDescriptor.createDescriptor("ARRAY_MAILS",conn);
String[] s_mails={"mail1@mail","mail2@mail", "mail3@mail"};
ARRAY mails=new ARRAY(array_mails,conn,s_mails);
ps.setArray(5, mails);
```

Veamos ahora el ejemplo completo:

```
//Creamos el descriptor para el objeto tipo_telefono
StructDescriptor sd_tipo_telefono = StructDescriptor.createDescriptor ("TIPO_TELEFONO",

//Creamos un objeto tipo_telefono llamada tipo_fijo que ya podremos almacenar en la base
```

```

String[] s_telefono_fijo={"Fijo","123451234"};
STRUCT telefono_fijo=new STRUCT(sd_tipo_telefono,conn,s_telefono_fijo);

// Creamos el descriptor para el array de telefonos, donde meteremos el telefono anterior
ArrayDescriptor desc_tabla_telefonos = ArrayDescriptor.createDescriptor("TABLA_TELEFONOS",conn);

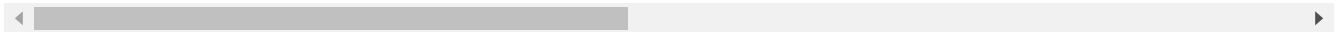
// Creamos un array de telefonos, que ahora si se podrá meter en la base de datos, al que le vamos a meter el telefono anterior
STRUCT[] a_telefonos=new STRUCT[1];
a_telefonos[0]=telefono_fijo;
ARRAY nt_telefonos = new ARRAY(desc_tabla_telefonos s,conn,a_telefonos);

// Reemplazamos el interrogante número 6 por la tabla de teléfonos.
ps.setArray(6, nt_telefonos);

// Creamos el array de mails, que ahora se podrá meter en la base de datos:
ArrayDescriptor array_mails=ArrayDescriptor.createDescriptor("ARRAY_MAILS",conn);
String[] s_mails={"mail1@mail","mail2@mail", "mail3@mail"};
ARRAY mails=new ARRAY(array_mails,conn,s_mails);


// Reemplazamos el interrogante número 5 por el array de mails.
ps.setArray(5, mails);

```



Anexo.- Licencias de recursos.

Licencias de recursos utilizados en I

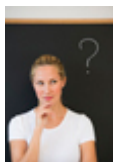
Recurso (1)	Datos del recurso (1)	
	<p>Autoría: LatinStock.</p> <p>Licencia: Uso educativo no comercial para plataformas de FpaD.</p> <p>Procedencia: LatinStock.</p>	
	<p>Autoría: dracos.</p> <p>Licencia: CC BY-SA.</p> <p>Procedencia: http://commons.wikimedia.org/wiki/File:Applications-database.svg</p>	
	<p>Autoría: LatinStock.</p> <p>Licencia: Uso educativo no comercial para plataformas de FpaD.</p> <p>Procedencia: LatinStock.</p>	
	<p>Autoría: LatinStock.</p> <p>Licencia: Uso educativo no comercial para plataformas de FpaD.</p> <p>Procedencia: LatinStock.</p>	
	<p>Autoría: Salvador Romero Villegas.</p> <p>Licencia: Uso educativo no comercial.</p> <p>Procedencia: Elaboración propia, partiendo de las siguientes imágenes:</p> <ul style="list-style-type: none"> ✓ Nombre: PROG10_CONT_AUXR09_R01_AccederALaInterfazEjecucionSQL1 Autoría: Salvador Romero Villegas para la composición. Intersystems Corporation para el diseño y la iconografía empleada en su portal de gestión del sistema. Licencia: Uso educativo no comercial para la parte correspondiente a Salvador Romero Villegas. Copyright (cita) para la parte correspondiente a Intersystems Corporation. Procedencia: Elaboración propia. ✓ Nombre: PROG10_CONT_AUXR09_R02_AccederALaInterfazEjecucionSQL2 Autoría: Salvador Romero Villegas para la composición. Intersystems Corporation para el diseño y la iconografía empleada en su portal de gestión del sistema. Licencia: Uso educativo no comercial para la parte correspondiente a Salvador Romero Villegas. Copyright (cita) para la parte correspondiente a Intersystems Corporation. Procedencia: Elaboración propia. ✓ Nombre: PROG10_CONT_AUXR09_R03_AccederALaInterfazEjecucionSQL3 Autoría: Salvador Romero Villegas para la composición. Intersystems Corporation para el diseño y la iconografía empleada en su portal de gestión del sistema. Licencia: Uso educativo no comercial para la parte correspondiente a Salvador Romero Villegas. Copyright (cita) para la parte correspondiente a Intersystems Corporation. Procedencia: Elaboración propia. 	



Autoría: LatinStock.
Licencia: Uso educativo no comercial para plataformas de FpaD.
Procedencia: LatinStock.



Autoría: LatinStock.
Licencia: Uso educativo no comercial para plataformas de FpaD.
Procedencia: LatinStock.



Autoría: LatinStock.
Licencia: Uso educativo no comercial para plataformas de FpaD.
Procedencia: LatinStock.



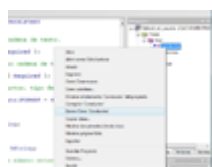
Autoría: LatinStock.
Licencia: Uso educativo no comercial para plataformas de FpaD.
Procedencia: LatinStock.



Autoría: LatinStock.
Licencia: Uso educativo no comercial para plataformas de FpaD.
Procedencia: LatinStock.



Autoría: LatinStock.
Licencia: Uso educativo no comercial para plataformas de FpaD.
Procedencia: LatinStock.



Autoría: Salvador Romero Villegas para la composición final. Intersystems Corporation para las capturas de pantalla de su producto Studio.
Licencia: Para la parte correspondiente a Salvador Romero, uso educativo no comercial, para el resto Copyright (cita).
Procedencia: Elaboración propia.

	<p>Autoría: Salvador Romero Villegas para la composición final y parte del texto escrito. Oracle Corporation para las capturas de pantalla procedentes de la herramienta SQL*Plus. Microsoft corporation para los elementos visibles propios del sistema operativo Windows.</p> <p>Licencia: Para la parte correspondiente a Salvador Romero, uso educativo no comercial, para el resto Copyright (cita).</p> <p>Procedencia: Elaboración propia.</p>	
	<p>Autoría: LatinStock.</p> <p>Licencia: Uso educativo no comercial para plataformas de FpaD.</p> <p>Procedencia: LatinStock.</p>	
	<p>Autoría: Salvador Romero Villegas para la composición final y parte del texto escrito. Intersystems Caché para las capturas de pantalla procedentes de la herramienta Studio. Microsoft corporation para los elementos visibles propios del sistema operativo Windows.</p> <p>Licencia: Para la parte correspondiente a Salvador Romero, uso educativo no comercial, para el resto Copyright (cita).</p> <p>Procedencia: Elaboración propia.</p>	
	<p>Autoría: LatinStock.</p> <p>Licencia: Uso educativo no comercial para plataformas de FpaD.</p> <p>Procedencia: LatinStock.</p>	
	<p>Autoría: LatinStock.</p> <p>Licencia: Uso educativo no comercial para plataformas de FpaD.</p> <p>Procedencia: LatinStock.</p>	
	<p>Autoría: Salvador Romero Villegas para la composición final y parte del texto escrito. Oracle Corporation para las capturas de pantalla procedentes de la herramienta SQL*Plus. Microsoft Corporation para las partes visibles de su sistema operativo Windows.</p> <p>Licencia: Para la parte correspondiente a Salvador Romero, uso educativo no comercial, para el resto Copyright (cita).</p> <p>Procedencia: Elaboración propia.</p>	
	<p>Autoría: LatinStock.</p> <p>Licencia: Uso educativo no comercial para plataformas de FpaD.</p> <p>Procedencia: LatinStock.</p>	

