|  |
| --- |
| cifp Juan de colonia |
| Acceso a Datos |
| UT1: Manejo de ficheros |
|  |
| **Alexis López Briongos Dam2t** |
| **02/11/2023** |

|  |
| --- |
|  |

*Índice*

[FICHERO 2](#_Toc155881508)

[CLASES JAVA 2](#_Toc155881509)

[FLUJO O STREAM 4](#_Toc155881510)

[ACCESO A FICHEROS 5](#_Toc155881511)

[OPERACIONES SOBRE FICHEROS 6](#_Toc155881512)

[OPERACIONES SOBRE REGISTROS DE UN FICHERO 6](#_Toc155881513)

[CLASES PARA LA GESTIÓN DE STREAM DE DATOS 7](#_Toc155881514)

[FICHEROS BINARIOS 8](#_Toc155881515)

[OBJETOS EN FICHEROS BINARIOS 9](#_Toc155881516)

[ACCESO ALEATORIO FICHEROS 9](#_Toc155881517)

[FICHEROS XML 10](#_Toc155881518)

[Acceso a los ficheros XML mediante DOM 11](#_Toc155881519)

[Excepciones 11](#_Toc155881520)

[JAXB 12](#_Toc155881521)

[Paso de esquema XML a clase Java 13](#_Toc155881522)

[RESULTSET 13](#_Toc155881523)

# FICHERO

¿Qué es un fichero?

* Conjunto de bits (información) almacenados en un dispositivo.

Características:

* No son volátiles.
* Se ubican en directorios o carpetas.
* Dentro de su directorio tienen un nombre (identificador) único.
* Cada fichero suele tener una extensión la cual indica el tipo de datos que contiene.

Compuesto por:

* Registros
  + Campos
* La manera en que se agrupan los datos dentro de un fichero depende del propio diseño del fichero.

# CLASES JAVA

¿Qué es java.io?

* Es la clase que contiene todo lo necesario para tratar ficheros con java

¿Qué es File?

* **File**: es una clase que se utiliza para la manipulación de ficheros.
* **List()**: devuelve un array de strings con los nombres de los ficheros y directorios de un objeto File.
* **ListFiles()** : devuelve un array de files con los ficheros y directorios que existen en ese directorio.
* **getName()**: devuelve el nombre de un fichero o directorio.
* **getPath()**: devuelve la ruta relativa del fichero o directorio que le pasemos.
* **getAbsolutePath()**: devuelve la ruta absoluta del fichero o directorio que le pasemos.
* **exists()**: devuelve un booleano si existe o no un fichero o directorio.
* **canWrite()**: devuelve un booleano si el fichero se puede escribir o no.
* **canRead()**: devuelve un booleano si el fichero se puede leer o no.
* **isFile()**: devuelve un booleano si es fichero o no
* **isDirectory()**: devuelve un booleano si es directorio o no
* **length()**: devuelve un longitud de un fichero en bytes
* **mkdir()**: crea un directorio si no existe y devuelve un booleano si existe o no dicho directorio
* **renameTo(File, “nuevoNombre”)**: renombra el fichero y devuelve booleano si el fichero contiene , o no existe , o no tienes permisos de escritura.
* **delete():** elimina el directorio si está vacío y devuelve un booleano si ha podido hacerlo.
* **createNewFile():** crea un fichero vacío si se puede.
* **getParent():** devuelve el nombre del directorio padre (string) si no devuelve null.

# FLUJO O STREAM

* Para manejar la entrada salida entre una fuente y un destino se utiliza los flujos o streams. Es decir, cualquier programa que necesite obtener información de cualquier fuente necesita abrir un stream y viceversa (enviar).
* La vinculación entre el stream y el dispositivo fuente la realiza el propio sistema de e/s de java (java.io).
* Tipos de streams
  + Streams de bytes (8 bits)
    - Se utilizan para operaciones de e/s de bytes. Es decir, para la lectura y escritura de datos binarios.
    - Todas las clases de streams de bytes descienden de las clases InputStream/OutputStream y cada una de ellas tiene varias subclases dependiendo del dispositivo que se vaya a utilizar.
  + Streams de caracteres (16 bits)
    - Se implementaron porque los streams 8 bits se quedaron cortos.
    - Se utilizan para operaciones de e/s de caracteres y para ello se emplean las clases: 🡺Manejan streams de caracteres UNICODE
      * Reader
      * Writer
* Clases de streams de bytes(8 bits):
  + InputStream: clases que producen entradas de distintas fuentes
    - ByteArrayInputStream permite utilizar un espacio de almacenamiento intermedio (buffer) en memoria.
    - StringBufferInputStream: convierte un string en un InputStream
    - PipedInputStream: almacena en tuberías
    - FilterInputStream: da funcionalidades a otras clases de InputStream
      * LineNumberInputStream
      * DataInputStream
      * BufferedInputStream
      * PushBackInputStream
    - SequenceInputStream: convierte dos o más objetos InputStream en un InputStream único.
    - FileInputStream: realiza un flujo de entrada desde un fichero.
  + OutputStream: clases que producen salidas de distintas fuentes.
    - ByteArrayOutputStream: permite utilizar un espacio de almacenamiento intermedio(buffer) en memoria.
    - PipedOutputStream: se asocia al PipedInputStream y la información que se graba en el PipedOutputStream acaba automáticamente como entrada al PipedInputStream asociado.
    - FilterOutputStream: da funcionalidades a otras clases de OutputStream
      * LineOutputStream
      * BufferedOutputStream
      * PrintStream
    - FileOutputStream: realiza un flujo de salida hacia un fichero.
* Clases de streams de 16 bits
  + Reader
    - BufferedReader (BufferedInputStream)
      * LineNumberReader
    - CharArrayReader (ByteArrayInputStream)
    - InputStreamReader: convierte un InputStream en un Reader
      * FileReader (FileInputStream)
    - FilterReader(FilterInputStream)
      * PushBackReader (PushBackInputStream)
    - PipeReader (PipedInputStream)
    - StringReader (StringBufferedInputStream)
  + Writer
    - BufferedWriter (BufferedOutputStream)
    - CharArrayWriter (ByteArrayOutputStream)
    - OutputStreamWriter
      * FileWriter (FileOutputStream)
    - FilterWriter (FilterOutputStream)
    - PipedWriter (PipedOutputStream)
    - StringWriter (StringOutputStream)
    - PrintWriter (PrintOutputStream)

# ACCESO A FICHEROS

* Existen dos maneras para acceder a ficheros
  + Acceso Secuencial en el cual los datos o registros del fichero se leen y se escriben siempre en orden secuencial (de principio a fin) el nuevo registro ira siempre al final del fichero a parir del último fichero.
    - Se suelen utilizar en procesos por lotes (Batch).
    - Clases para acceder de forma secuencial a los ficheros
      * FileOutputStream
      * FileInputStream
      * FileWriter
      * FileReader
    - Ventajas
      * Son rápidos para acceder a los registros.
      * Aprovechan mejor el espacio.
      * Son sencillos de usar.
    - Inconvenientes
      * No se puede acceder directamente a un registro.
      * Son muy complicados de actualizar.
  + Acceso Aleatorio (ficheros aleatorios) se permite acceder directamente a un dato o registro sin haber pasado antes por los anteriores.
    - Los datos tienen que estar almacenados en registros de tamaño fijo.
    - Clases que se utilizan para acceso aleatorio en ficheros.
      * RandomAccessFile.
    - Los ficheros aleatorios utilizan direcciones relativas, no absolutas.
    - Ventajas
      * Rapidez con la que se accede a un registro.
    - Inconvenientes:
      * Establecer posición de cada registro
      * Se desaprovecha el espacio, puede haber huecos.

# OPERACIONES SOBRE FICHEROS

* Independientemente de cuál sea su forma de acceso, las operaciones que se puedan realizar sobre un fichero son:
  + Creación: de un fichero con un nombre único en el directorio que nos permite acceder a él. Es un proceso que solo se debe hacer una vez.
  + Apertura: es la primera operación que hay que hacer para que un programa pueda operar con el fichero.
  + Lectura: transferir información desde el fichero hacia la memoria (programa).
  + Escritura: transferir información desde la memoria hacia el fichero.
  + Cierre: cuando no se vaya a usar más y suele ser siempre la ultima instrucción del programa.
  + Eliminación

# OPERACIONES SOBRE REGISTROS DE UN FICHERO

* Altas: insertar nuevos registros
  + Si el acceso es secuencial el registro ira al final del fichero.
  + Si el acceso aleatorio el registro irá se aplica la función designada.
* Bajas: eliminar registros existentes:
  + Si el acceso es secuencial el borrado será física.
    - Se emplea algún campo que cambia de valor para indicar que ese registro está borrado lógicamente.
    - Se realizan las con un fichero auxiliar de manera que se leen todos los registros y se copian en otro fichero leyendo todos los registros y obviando el que tiene registro modificado, se elimina el fichero original y se renombra el fichero auxiliar con el nombre del fichero original.
  + Si el acceso es aleatorio el borrado será lógica del fichero.
    - Se emplea algún campo que cambia de valor para indicar que ese registro está borrado lógicamente.
* Modificaciones: cambiar valores de los registros.
  + Si el acceso es secuencial la modificación será lógica.
    - Se emplea algún campo que cambia de valor para indicar que ese registro está borrado lógicamente.
    - Se realizan con un fichero auxiliar de manera que se leen todos los registros y se copian en otro fichero leyendo todos los registros y añadiendo el registro que tiene el campo modificado pero con el valor modificado, se elimina el fichero original y se renombra el fichero auxiliar con el nombre del fichero original.
  + Si el acceso es aleatorio la modificación mediante la clave.
    - Con el campo clave buscar el registro que queremos modificar y cambiar el valor del registro.
* Consultas: accedemos y obtenemos información de los registros:
  + Si el acceso es secuencial consulta será desde el primero hasta el registro a encontrar.
  + Si el acceso es aleatorio se recogerá el valor de los registros por la clave introducida.

# CLASES PARA LA GESTIÓN DE STREAM DE DATOS

* Ficheros de texto manejan caracteres alfanuméricos (ASCII, UTF-8).
* En ellos se emplean las clases FileReader y FileWriter.
* Cada vez que se lee o escribe en un fichero de este tipo, hay que hacerlo en un bloque try catch dado que pueden generar excepciones.
  + En el caso de lectura 🡺 FIleNotFoundException
  + En el caso de escritura 🡺 IOException
* FILEREADER métodos: estos métodos devolverán siempre los caracteres leídos o -1 si se ha llegado a final del fichero o si no se puede leer más (no hay más datos, corruptos).
  + Int Read(): lee un carácter y lo devuelve
  + Int Read(char[] buf): leer hasta buf.length caracteres de una matriz de caracteres buf desde el principio.
  + Int Read(char[]buf, int despl, int n): lee hasta n caracteres de la matriz buf, comenzando por buf[despl] indicando el despl (desplazamiento) .
  + Pasos con los que trabajaremos con los ficheros.
    - Creamos fichero, creamos el flujo de entrada con el FileReader, realizamos operaciones de lectura, cerramos el fichero con el método close().
* FILEWRITER métodos:
  + Void write(int c): escribi
  + Void write(char[]buf)
  + Void write(char[]buf, int despl, int n): escribir n caracteres de la matriz buf comenzando desde int despl.
  + Void Write(String str): escribir una cadena de caracteres
  + Void append(Char c): añadir el carácter al final del fichero
* La clase FileReader no tenia métodos para leen líneas completas pero BufferedReader si con el método readLine();
* BUFFEREDREADER métodos:
  + readLine(): lee líneas completas o null si no hay nada o es fin de fichero
  + read(): lee caracteres o null si no hay nada o es fin de fichero
* BUFFEREDWRITER métodos:
  + Write():
* PRINTWRITER:
  + Print(string): imprime la cadena
  + Println(string) :imprime la cadena y un salto de línea

# FICHEROS BINARIOS

* Ficheros que almacenan secuencias de dígitos binarios que no son legibles directamente.
* Ventajas
  + Ocupan menos espacio que los de caracteres.
  + Para almacenar mucha información son mejores.
* Clases que se utilizan para estos ficheros
  + FileInputStream
  + FileOutputStream
* FILEINPUTSTREAM métodos:
  + Int read(): lee un byte(8bits) y lo devuelve
  + Int Read(byte[] b): leer la matriz b hasta su longitud máxima y los devuelve.
  + Int read[byte[] b, int index, int n): lee n bytes(8bits) de la matriz b, comenzando por index.
* FILEOUTPUTSTREAM métodos:
  + Void write(int b):
  + Void write(byte[] b)
  + Void write(byte [] b, int index,int n) : escribe n bytes, de la matriz b, comenzando desde el index.
* Para leer y escribir tipos de datos primitivos, (int, float,etc) se usan los siguientes métodos
* DataInputStream métodos:
  + Boolean readBoolean();
  + Byte readByte();
  + Int readUnsignedByte();
  + Int readUnsignedShort();
  + Shor readShort();
  + Char readChar();
  + Int readint();
  + Long readLong();
  + Float readFloat();
  + Double readDouble();
  + String readUTF();
* DataOutputStream métodos:
  + void writeBoolean(boolean b);
  + void writeByte(int v);
  + void writeBytes(string s)
  + void writeShort(int v);
  + void writeChar(int v);
  + void writeChars(String s)
  + void writeInt(int v);
  + void writeLong(long v);
  + void writeFloat(float v);
  + void writeDouble(doublé v);
  + void writeUTF(String str);

# OBJETOS EN FICHEROS BINARIOS

* Serializable tiene métodos para leer y guardar objetos en ficheros binarios.
* La serialización de objetos permite tomar cualquier objeto implementado por la interfaz serializable y convertirlo en una secuencia de bits, que pueden ser leídas o guardadas.
* Utilizaremos las clase ObjectOutputStream y ObjectInputStream.
* OBJECTINPUTSTREAM:
  + Object readObject();
    - Pueden lanzar excepciones:
      * IOException: error de entrada/salida
      * ClassNotFoundException
* OBJECTOUTPUTSTREAM:
  + void writeObject(Object obj);
    - Pueden lanzar excepciones:
      * IOException: error de entrada/salida
  + Void writeStreamHeader(): este método sirve para escribir la cabecera (es importante modificar este método creando una clase que extienda ObjectOutputStream para modificarlo y poder sobreescribir un fichero.

# ACCESO ALEATORIO FICHEROS

* RANDOMACCESFILE: esta clase tendrá métodos para acceder a un fichero de acceso aleatorio
  + RandomAccessFile(String nombreFich,String modoAcceso)
    - modoAcceso=“-r” se abre en modo lectura y el fichero debe existir
      * Puede dar exepción IOException
    - modoAcces=”-rw” se abre en modo lectura/escritura y si no existe, se crea
  + RandomAccesFile(File objetofile, String modoAcceso)
    - Pueden lanzar la excepción FileNotFoundException
  + Funcionamiento
    - Se tiene un puntero interno que señala la posición actual dentro del fichero. Se coloca siempre en la posición 0 cuando el fichero se crea y se va moviendo según la cantidad de Bytes escritos o leídos
  + RandomAccessFile métodos:
    - Se puede utilizar los métodos de DataInputSttream y DataOutputStream para leer y escribir.
    - Long getFilePointer(); devuelve la posición actual del puntero
    - Void seek(long posición); coloca el puntero en la posición que le indiquemos.
    - Long lenght; devuelve la longitud de fichero en bytes.
    - Int skipBytes(int despl); desplaza el puntero desde la posición actual un numero de bytes que le indiquemos (despl).

# FICHEROS XML

* XML es un metalenguaje no es un lenguaje de programación que permite estructurar y jerarquizar la información describiendo los contenidos dentro del propio documento.
* Los ficheros XML son ficheros de texto que tienen unas marcas especiales utilizadas para delimitar la estructura del documento, de tal manera que cada etiqueta irá delimitada por los símbolos antes < y > y tendrá un nombre <nombre>. Además podrá contener o no una serie de atributos.
* Se pueden usar para:
  + Proporcionar datos a una base de datos.
  + Almacenar copias de parte del contenido de una base de datos.
  + Ficheros de configuración de algunos programas.
  + Protocolo SOAP(Simple Object Access Protocol), sirve para gestionar servidores en modo remoto.
* Para leer los ficheros XML en java y acceder a su contenido se usan:
  + Procesadores XML (Parsers): son independientes al del lenguaje de programación.
    - DOM (Document Object Model): se almacena toda la estructura del documento en memoria en forma de árbol, es decir, con nodos padre, nodos hijos y nodos finales. Una vez creado el árbol, se recorrerán los diferentes nodos y se verá a que tipo pertenece cada uno. Este procesamiento de la información requiere el uso de mucha memoria y tiempo sobre todo cuando los ficheros XML son grandes.
    - SAX(Simple API for XML): se lee el fichero XML de forma secuencial y se produce una serie de eventos en función de la lectura:
      * Inicio y fin del documento
      * Inicio y fin de la etiqueta
      * Etc.
      * Cada evento producido invoca a un método definido por el programador.
      * Este procesamiento consume muy poca memoria a diferencia del DOM, pero no se tiene una visión global del documento.

# Acceso a los ficheros XML mediante DOM

* Necesitaremos las clases, interfaces contenidas en:
  + org.w3c.dom
  + javax.xml.parsers
* Estas clases ofrecen métodos para cargar documentos XML desde una fuente de datos y contienen 2 clases fundamentales:
  + DocumentBuilderFactory.
  + DocumentBuilder.
* DOM no define ningún mecanismo para generar un fichero XML a partir de un árbol en memoria y para ello se utiliza el paquete:
  + java.xml.transform: Permite especificar una fuente y un resultado.
* Los programas que usen DOM necesitarán las siguientes interfaces:
  + Document: es un objeto que equivale a un documento xml y que permite crear nuevos nodos.
  + Element: es cada elemento del documento XML. Esta interfaz tendrá propiedades y métodos para manipular los elementos y los atributos de estos.
  + Node: cualquier nodo del documento.
  + NodeList: que contiene una lista de los nodos hijos de un nodo dado.
  + Attr: permite acceder a los atributos de un nodo.
  + Text: son los datos de tipo carácter de un elemento.
  + CharacterData: son los datos carácter del documento proporcionando atributos y métodos para manipular los datos de caracteres.
  + DocumentType: proporciona información contenida en la etiqueta <!DOCTYPE>

# Excepciones

* Evento que ocurre durante la ejecución de un programa interrumpiendo el flujo normal de las sentencias.
* Es decir, el programa java es ejecutado de forma secuencial en el momento que una sentencia provoca una excepción, será el propio programa quien la capture para tratarla y de no ser así, será capturada por el gestor de excepciones por defecto.
* Retorna un mensaje y detiene el programa.
* Cuando ocurre una excepción dentro de un método, es el propio método quien crea un objeto Exception derivado de la clase Throwable.
* Para capturar excepciones dentro del código utilizamos el bloque:
  + Try: capturar las sentencias donde se pueden provocar la excepción.
  + Catch: donde se trata la o las excepciones.
  + Finally (opcional): este bloque se ejecuta siempre.ç
* Si se quiere obtener más información de la excepción se utilizaría los métodos de la clase Throwable:
  + String getMessage();
  + String getLocalizedMessage();
  + String toString();
  + Void printStackTrace();
    - printStackTrace(PrintStream)
    - printStackTrace(PrintWriter)
* Los bloques trys están dentro de otros , pero si el try interno no tuviera un manejador catch, se buscaría las sentencias catch del try externo.
* Para especificar excepciones se usa la palabra reservada throws dentro de la misma función.

# JAXB

* <https://javaee.github.io/jaxb-v2/>
  + Descargar las siguientes jar:
    - Jaxb-api.jar
    - Jaxb-runtime.jar
    - Javax.activation-api.jar
    - Istack-commons-runtime.jar
* Java Arquitecture for XML Binding es una tecnología que permite mapear (serializar) clases java a representaciones XML y viceversa.
* Tiene dos funciones fundamentales:
  + Marshalling sirve para presentar(serializar) un objeto java en xml
  + Unmarshalling sirve para presentar xml un objeto java o deserealizar.
* El compilador JAXB nos permite generar clases java a partir de esquemas xml que podrán llamarse desde las aplicaciones a partir de métodos getters y setters para obtener o escribir los datos de un documento XML.
* Para crear objetos java en XML se emplean las clases JavaBeans que son clases primitivas. En estas clases que se van a mapear (serializar) se añaden las anotaciones que son indicaciones que ayudan a convertir el JavaBeans en XML. Estas anotaciones son:
  + @XmlRootElement (namespace=”nombre”) esta anotación define la raíz del XML de tal manera que si una clase va a ser raíz se define este elemento.
  + @XmlType (namespace=”nombre”) esta anotación define el orden en el que se van a escribir los elementos. Si una clase no va a ser raíz se añade esta anotación:
    - @XmlType (propOrder={“campo1”,”campo2”,…}
  + @XmlElement (namespace=”nombre”) esta anotación define el elemento xml.
* Cualquiera de ellos puede llevar el nombre de la etiqueta que se quiera que aparezca.
* Para cada atributo de la clase XML el método get correspondiente debe llevar una anotación @XmlElement.
  + Si el atributo es una colección además debe llevar una anotación @XmlElementWrapper.
  + Si el atributo fuese otra clase JavaBeans se pondría igualmente al método get correspondiente la anotación @XmlElement.
  + La clase que hace de atributo deberá llevar las correspondientes anotaciones:

# Paso de esquema XML a clase Java

* El compilador JAXB permite generar una serie de clases java a partir de esquemas XML.
* Estos esquemas XML describen la estructura de un documento XML y se conocen como XSD (XML Schema Definition)
* La manera de actuar es que JAXB compila el fichero XSD creando una serie de clases para cada uno de los tipos que se hayan especificado en el fichero XSD.
* Estas clases son clase (POJO)🡺Plain Old Java Object y se utilizan para crear y modificar documentos XML

# RESULTSET

* Métodos de ResultSet para actuar con el puntero:
  + Boolean next(): mueve el puntero una fila adelante(registro?) a partir de la posición actual.
  + Boolean first(): mueve el puntero al primer registro.
  + Boolean last(): mueve el puntero al último registro.
  + Boolean previous(): mueve el puntero al registro anterior a partir de la posición actual del puntero.
  + Void beforeFirst(): posiciona el puntero justo antes del primer registro. Inicializa el ResultSet.
  + Int getRow(): devuelve el número del registro actual.

# JDBC

* JDBC nos da herramientas para obtener información de la base de datos
* Sentencias de descripción de datos
* Se utilizan metaobjetos de los cuales la interfaz DataBaseMetaData proporciona diferentes métodos:
  + getTables(); proporciona información sobre las tablas y las vistas de la base de datos y devuelve un ResultSet.
    - Public abstract ResultSet getTables(String catalogo, String esquema, String patronDeTabla, String tipos[ ]) throws SQLException
      * Catalogo: indicamos el catalogo y si es null significa que es el catalogo actual.
      * Esquema: indicamos el esquema y si es null puede significar el esquema actual o todos los esquemas dependiendo del SGBD.
      * Patron de Tabla: es un patrón que indica el nombre de las tablas a obtener y se pueden utilizar los símbolos <\_> y <%> como comodines para realizar búsquedas menos precisas.
      * Tipos[ ]: es un array de String en el que se indican los tipos de objetos a obtener si se pone null significa todos los tipos.
        + TABLE
        + VIEW
        + SYSTEM TABLE
        + GLOBAL TEMPORARY
        + LOCAL TEMPORARY
        + ALIAS
        + SYNONYM
    - Las columnas que devuelve el ResultSet son:
      * TABLE\_CAT: nombre del catalogo
      * TABLE\_SCHEMA: nombre del esquema.
      * TABLE\_NAME: nombre de la tabla.
      * TABLE\_TYPE: tipo de objeto tabla (TABLE O VIEW).
      * REMARKS: comentarios.
      * TYPE\_CAT: tipo de catalogo
      * TYPE\_SCHEM: tipo de esquema
      * TYPE\_NAME
      * SELF\_REFERING\_COL\_NAME:
      * REF\_GENERATION
  + getColumns(); devuelve información sobre las columnas de una tabla también devuelve un ResultSet.
    - Public abstract ResultSet getColumns(String catalogo, String esquema, String patronDeTabla, String tipos[ ]) throws SQLException
    - Las columnas que devuelve el ResultSet son:
    - **TABLE\_CAT** String => table catalog (may be null)
    - **TABLE\_SCHEM** String => table schema (may be null)
    - **TABLE\_NAME** String => table name
    - **COLUMN\_NAME** String => column name
    - **DATA\_TYPE** int => SQL type from java.sql.Types
    - **TYPE\_NAME** String => Data source dependent type name, for a UDT the type name is fully qualified
    - **COLUMN\_SIZE** int => column size.
    - **BUFFER\_LENGTH** is not used.
    - **DECIMAL\_DIGITS** int => the number of fractional digits. Null is returned for data types where DECIMAL\_DIGITS is not applicable.
    - **NUM\_PREC\_RADIX** int => Radix (typically either 10 or 2)
    - **NULLABLE** int => is NULL allowed.
      * columnNoNulls - might not allow NULL values
      * columnNullable - definitely allows NULL values
      * columnNullableUnknown - nullability unknown
    - **REMARKS** String => comment describing column (may be null)
    - **COLUMN\_DEF** String => default value for the column, which should be interpreted as a string when the value is enclosed in single quotes (may be null)
    - **SQL\_DATA\_TYPE** int => unused
    - **SQL\_DATETIME\_SUB** int => unused
    - **CHAR\_OCTET\_LENGTH** int => for char types the maximum number of bytes in the column
    - **ORDINAL\_POSITION** int => index of column in table (starting at 1)
    - **IS\_NULLABLE** String => ISO rules are used to determine the nullability for a column.
      * YES --- if the column can include NULLs
      * NO --- if the column cannot include NULLs
      * empty string --- if the nullability for the column is unknown
    - **SCOPE\_CATALOG** String => catalog of table that is the scope of a reference attribute (null if DATA\_TYPE isn't REF)
    - **SCOPE\_SCHEMA** String => schema of table that is the scope of a reference attribute (null if the DATA\_TYPE isn't REF)
    - **SCOPE\_TABLE** String => table name that this the scope of a reference attribute (null if the DATA\_TYPE isn't REF)
    - **SOURCE\_DATA\_TYPE** short => source type of a distinct type or user-generated Ref type, SQL type from java.sql.Types (null if DATA\_TYPE isn't DISTINCT or user-generated REF)
    - **IS\_AUTOINCREMENT** String => Indicates whether this column is auto incremented
      * YES --- if the column is auto incremented
      * NO --- if the column is not auto incremented
      * empty string --- if it cannot be determined whether the column is auto incremented
    - **IS\_GENERATEDCOLUMN** String => Indicates whether this is a generated column
      * YES --- if this a generated column
      * NO --- if this not a generated column
      * empty string --- if it cannot be determined whether this is a generated column
  + getPrimaryKeys(): devuelve información sobre la o las columnas que forman la clave primaria de una tabla.
    - Public abstract ResultSet getPrimaryKeys(String catalogo, String esquema, String table) throws SQLException
    - Each primary key column description has the following columns:
      * **TABLE\_CAT** String => table catalog (may be null)
      * **TABLE\_SCHEM** String => table schema (may be null)
      * **TABLE\_NAME** String => table name
      * **COLUMN\_NAME** String => column name
      * **KEY\_SEQ** short => sequence number within primary key( a value of 1 represents the first column of the primary key, a value of 2 would represent the second column within the primary key).
      * **PK\_NAME** String => primary key name (may be null)
* getExportedKeys();Devuelve información de las claves ajenas que devuelven la clave primaria.
* Public abstract ResultSet getPrimaryKeys(String catalogo, String esquema, String table) throws SQLException
* Each foreign key column description has the following columns:
  + **PKTABLE\_CAT** String => primary key table catalog (may be null)
  + **PKTABLE\_SCHEM** String => primary key table schema (may be null)
  + **PKTABLE\_NAME** String => primary key table name
  + **PKCOLUMN\_NAME** String => primary key column name
  + **FKTABLE\_CAT** String => foreign key table catalog (may be null) being exported (may be null)
  + **FKTABLE\_SCHEM** String => foreign key table schema (may be null) being exported (may be null)
  + **FKTABLE\_NAME** String => foreign key table name being exported
  + **FKCOLUMN\_NAME** String => foreign key column name being exported
  + **KEY\_SEQ** short => sequence number within foreign key( a value of 1 represents the first column of the foreign key, a value of 2 would represent the second column within the foreign key).
  + **UPDATE\_RULE** short => What happens to foreign key when primary is updated:
    - importedNoAction - do not allow update of primary key if it has been imported
    - importedKeyCascade - change imported key to agree with primary key update
    - importedKeySetNull - change imported key to NULL if its primary key has been updated
    - importedKeySetDefault - change imported key to default values if its primary key has been updated
    - importedKeyRestrict - same as importedKeyNoAction (for ODBC 2.x compatibility)
  + **DELETE\_RULE** short => What happens to the foreign key when primary is deleted.
    - importedKeyNoAction - do not allow delete of primary key if it has been imported
    - importedKeyCascade - delete rows that import a deleted key
    - importedKeySetNull - change imported key to NULL if its primary key has been deleted
    - importedKeyRestrict - same as importedKeyNoAction (for ODBC 2.x compatibility)
    - importedKeySetDefault - change imported key to default if its primary key has been deleted
  + **FK\_NAME** String => foreign key name (may be null)
  + **PK\_NAME** String => primary key name (may be null)
  + **DEFERRABILITY** short => can the evaluation of foreign key constraints be deferred until commit
    - importedKeyInitiallyDeferred - see SQL92 for definition
    - importedKeyInitiallyImmediate - see SQL92 for definition
    - importedKeyNotDeferrable - see SQL92 for definition
* getImportedKeys(); devuelve información sobre las claves ajenas existentes en una tabla.
  + Public abstract ResultSet getPrimaryKeys(String catalogo, String esquema, String table) throws SQLException
* Each primary key column description has the following columns:
  + **PKTABLE\_CAT** String => primary key table catalog being imported (may be null)
  + **PKTABLE\_SCHEM** String => primary key table schema being imported (may be null)
  + **PKTABLE\_NAME** String => primary key table name being imported
  + **PKCOLUMN\_NAME** String => primary key column name being imported
  + **FKTABLE\_CAT** String => foreign key table catalog (may be null)
  + **FKTABLE\_SCHEM** String => foreign key table schema (may be null)
  + **FKTABLE\_NAME** String => foreign key table name
  + **FKCOLUMN\_NAME** String => foreign key column name
  + **KEY\_SEQ** short => sequence number within a foreign key( a value of 1 represents the first column of the foreign key, a value of 2 would represent the second column within the foreign key).
  + **UPDATE\_RULE** short => What happens to a foreign key when the primary key is updated:
    - importedNoAction - do not allow update of primary key if it has been imported
    - importedKeyCascade - change imported key to agree with primary key update
    - importedKeySetNull - change imported key to NULL if its primary key has been updated
    - importedKeySetDefault - change imported key to default values if its primary key has been updated
    - importedKeyRestrict - same as importedKeyNoAction (for ODBC 2.x compatibility)
* **DELETE\_RULE** short => What happens to the foreign key when primary is deleted.
  + - importedKeyNoAction - do not allow delete of primary key if it has been imported
    - importedKeyCascade - delete rows that import a deleted key
    - importedKeySetNull - change imported key to NULL if its primary key has been deleted
    - importedKeyRestrict - same as importedKeyNoAction (for ODBC 2.x compatibility)
    - importedKeySetDefault - change imported key to default if its primary key has been deleted
  + **FK\_NAME** String => foreign key name (may be null)
  + **PK\_NAME** String => primary key name (may be null)
  + **DEFERRABILITY** short => can the evaluation of foreign key constraints be deferred until commit
    - importedKeyInitiallyDeferred - see SQL92 for definition
    - importedKeyInitiallyImmediate - see SQL92 for definition
    - importedKeyNotDeferrable - see SQL92 for definition
* getProdecures(); devuelve información sobre los procedimientos almacenados.
  + Public abstract ResultSet getPrimaryKeys(String catalogo, String esquema, String table) throws SQLException
  + Each procedure description has the following columns:
    - **PROCEDURE\_CAT** String => procedure catalog (may be null)
    - **PROCEDURE\_SCHEM** String => procedure schema (may be null)
    - **PROCEDURE\_NAME** String => procedure name
    - **REMARKS** String => explanatory comment on the procedure
    - **PROCEDURE\_TYPE** short => kind of procedure:
      * procedureResultUnknown - Cannot determine if a return value will be returned
      * procedureNoResult - Does not return a return value
      * procedureReturnsResult - Returns a return value
    - **SPECIFIC\_NAME** String => The name which uniquely identifies this procedure within its schema.