Manejo de ficheros.

Caso práctico

Ana está empezando a cursar la Formación en Centros de Trabajo (FCT).

Ya ha tenido unas reuniones con **Juan** y **María**, para saber cómo se trabaja en BK programación. Aunque está haciendo el módulo de FCT en esta empresa, ya sabe que a veces tendrá que salir a otras empresas acompañada de sus tutores para ver los requisitos de los sistemas que la empresa tenga que informatizar y, en ocasiones, **Antonio**, quizás también se apunte para echar una mano.

Ana está nerviosa, también ilusionada, y tiene muchas ganas de conocer de cerca la realidad de lo que ha estudiado en clase.

Ahora verá el uso de los conocimientos adquiridos de diferentes módulos, y buscará respuestas a posibles dudas que se vayan planteando.

En clase les habían explicado la importancia de los ficheros en el acceso a datos. -Es importante repasar los conceptos -piensa Ana.

1.- Introducción.

Si estás estudiando este módulo, es probable que ya hayas estudiado el de programación, por lo que no te serán desconocidos muchos conceptos que se tratan en este tema.

Ya sabes que cuando apagas el ordenador, los datos de la memoria RAM se pierden. Un ordenador utiliza ficheros para guardar los datos. Piensa que los datos de las canciones que oyes en mp3, las películas que ves en formato avi, o mp4, etc., están, al fin y al cabo, almacenadas en ficheros, los cuales están grabados en un soporte como son los discos duros, DVD, pendrives, etc.

Se llama a los datos que se guardan en ficheros **datos persistentes**, porque persisten más allá de la ejecución de la aplicación que los trata. Los ordenadores almacenan los ficheros en unidades de almacenamiento secundario como discos duros, discos ópticos, etc. En esta unidad veremos, entre otras cosas, cómo hacer con Java las operaciones de crear, actualizar y procesar ficheros.

A las operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como Entrada/Salida (E/S).

Las operaciones de E/S en Java las proporciona el paquete estándar de la API de Java denominado `java.io` que incorpora interfaces, clases y excepciones para acceder a todo tipo de ficheros.

La **librería** `java.io` contiene las clases necesarias para gestionar las operaciones de entrada y salida con Java. Estas clases de E/S las podemos agrupar fundamentalmente en:

- Clases para leer entradas desde un flujo de datos.
- Clases para escribir entradas a un flujo de datos.
- Clases para operar con ficheros en el sistema de ficheros local.
- Clases para gestionar la serialización de objetos.

`java.nio` **(Non- Blocking I/O)** fue introducido en el API de Java desde la versión 1.4 como extensión eficiente a los paquetes `java.io` y `java.net`. Java NIO ofrece una forma diferente de trabajar con IO que las API de IO estándar. Se basa en el "Buffer" y el "Channel".

`java.io` **vs** `java.nio`

- Depende de lo que necesitemos pueden ser complementarios y utilizaremos conjuntamente ambos paquetes.
- `java.nio` permite manejar múltiples canales (archivos o conexiones de red) con uno o unos pocos hilos.
- En `java.nio` el procesamiento de datos es más complicado que usar los streams bloqueantes de java.io
- `java.nio` es la opción si necesito manejar cientos de conexiones (canales) abiertas y en cada una manejar una pequeña cantidad de datos.
- java.io es la opción si voy a manejar pocas conexiones con un alto ancho de banda (envío mucha información a la vez)
- Hasta JSE7 `java.io.File` era la clase utilizada para realizar operaciones I/O con archivos
- Esta clase tenía limitaciones como:
  - Muchos métodos no lanzaban excepciones por lo que era muy complicado detectar y resolver errores
  - No permitía manejar enlaces simbólicos
  - Presenta problemas de escalabilidad. Directorios largos provocaban problemas de memoria e incluso de denegación de servicio
- El paquete `java.nio.file` incorporado a partir de JSE7 resuelve estos problemas.
- Es el que debemos usar para trabajar con archivos independientemente de si realizamos I/O con streams (`java.io`) ó con buffers y channels (`java.nio`)

Autoevaluación

Indica si la afirmación es verdadera o falsa:

Los datos persistentes perduran tras finalizar la ejecución de la aplicación que los trata.

○ Verdadero ○ Falso

Verdadero
La idea de persistencia de datos, es seguir existiendo tras la ejecución de la aplicación.

2.- Clases asociadas a las operaciones de gestión de ficheros y directorios.

Caso práctico

Ana le comenta a **Antonio** -Por lo que nos han comentado, vamos a tener que utilizar bastante los ficheros. ¿Cómo te manejas tú con ellos?

-Pues tan bien como tú -responde **Antonio**, -yo también estudié el módulo de Programación. ¿Es que no recuerdas que en el módulo estudiamos un tema sobre ficheros?

-Sí, pero es que, como había tantos métodos para listar, renombrar archivos, etc., ya casi no me acuerdo; y eso que hace poco que lo estudiamos -contesta **Ana**.

Antonio intenta tranquilizar a **Ana** y le dice que no se preocupe, que en cuanto se les presente la ocasión de tener que programar con ficheros, seguro que no tienen problema y refrescan los conceptos que aprendieron en su día.

En efecto, tal y como dicen Ana y Antonio, hay bastantes métodos involucrados en las clases que en Java nos permiten manipular ficheros y carpetas o directorios.

Vamos a ver la clase `File` que nos permite hacer unas cuantas operaciones con ficheros, también veremos cómo filtrar ficheros, o sea, obtener aquellos con una característica determinada, como puede ser que tengan la extensión **.odt**, o la que nos interese, y por último, en este apartado también veremos como crear y eliminar ficheros y directorios.

2.1.- Clase File.

¿Para qué sirve esta clase, qué nos permite? La clase `File` proporciona una representación abstracta de ficheros y directorios.

Esta clase, permite examinar y manipular archivos y directorios, independientemente de la plataforma en la que se esté trabajando: Linux, Windows, etc.

Las instancias de la clase `File` representan nombres de archivo, no los archivos en sí mismos.

El archivo correspondiente a un nombre puede ser que no exista, por esta razón habrá que controlar las posibles **excepciones**.

Un objeto de clase `File` permite examinar el nombre del archivo, descomponerlo en su rama de directorios o crear el archivo si no existe, pasando el objeto de tipo `File` a un constructor adecuado como `FileWriter(File f)`, que recibe como parámetro un objeto `File`.

Para archivos que existen, a través del objeto `File`, un programa puede examinar los atributos del archivo, cambiar su nombre, borrarlo o cambiar sus permisos. Dado un objeto `File`, podemos hacer las siguientes operaciones con él:

- **Renombrar** el archivo, con el método `renameTo()`. El objeto `File` dejará de referirse al archivo renombrado, ya que el `String` con el nombre del archivo en el objeto `File` no cambia.
- **Borrar** el archivo, con el método `delete()`. También, con `deleteOnExit()` se borra cuando finaliza la ejecución de la máquina virtual Java.
- **Crear** un nuevo fichero con un nombre único. El método estático `createTempFile()` crea un fichero temporal y devuelve un objeto `File` que apunta a él. Es útil para crear archivos temporales, que luego se borran, asegurándonos tener un nombre de archivo no repetido.
- **Establecer** la fecha y la hora de modificación del archivo con `setLastModified()`. Por ejemplo, se podría hacer: `new File("prueba.txt").setLastModified(new Date().getTime());` para establecerle la fecha actual al fichero que se le pasa como parámetro, en este caso prueba.txt.
- **Crear** un directorio, mediante el método `mkdir()`. También existe `mkdirs()`, que crea los directorios superiores si no existen.
- **Listar** el contenido de un directorio. Los métodos `list()` y `listFiles()` listan el contenido de un directorio. `list()` devuelve un vector de `String` con los nombres de los archivos, `listFiles()` devuelve un vector de objetos `File`.
- **Listar** los nombres de archivo de la raíz del sistema de archivos, mediante el método estático `listRoots()`.

La clase proporciona los siguientes constructores para crear objetos `File`:

- `public File(String nombreFichero|path);`
- `public File(String path, String nombreFichero|path);`
- `public File(File path, String nombreFichero|path);`

La ruta o `path` puede ser absoluta o relativa.
Ejemplos utilizando el primer constructor:

- `File f = new File("personas.dat");`
  Crea un Objeto `File` asociado al fichero personas.dat que se encuentra en el directorio de trabajo. En este caso no se indica `path`. Se supone que el fichero se encuentra en el directorio actual de trabajo.
- `File f = new File("ficheros/personas.dat");`
  Crea un Objeto `File` asociado al fichero personas.dat que se encuentra en el directorio ficheros dentro del directorio actual. En este caso se indica la ruta relativa tomando como base el directorio actual de trabajo. Se supone que el fichero personas.dat se encuentra en el directorio ficheros. A su vez el directorio ficheros se encuentra dentro del directorio actual de trabajo.
- `File f = new File("c:/ficheros/personas.dat");`
  Crea un Objeto `File` asociado al fichero personas.dat dando la ruta absoluta:

Ejemplos utilizando el segundo constructor:
En este caso se crea un objeto `File` cuya ruta (absoluta o relativa) se indica en el primer String.

- `File f = new File("ficheros", "personas.dat" );`
  Crea un Objeto `File` asociado al fichero personas.dat que se encuentra en el directorio ficheros dentro del directorio actual. En este caso se indica la ruta relativa tomando como base el directorio actual de trabajo.

Ejemplos utilizando el tercer constructor:
Este constructor permite crear un objeto `File` cuya ruta se indica a través de otro objeto `File`.

- `File ruta = new File("ficheros");`
  `File f = new File(ruta, "personas.dat" );`
  Crea un Objeto `File` asociado al fichero personas.dat que se encuentra en el directorio ficheros dentro del directorio actual.

Debemos tener en cuenta que crear un objeto `File` no significa que deba existir el fichero o el directorio o que el `path` sea correcto.
Si no existen no se lanzará ningún tipo de excepción ni tampoco serán creados.
Métodos
Alguno de los métodos de la clase `File` son los siguientes:

Método	Descripción
`boolean canRead()`	Devuelve true si se puede leer el fichero
`boolean canWrite()`	Devuelve true si se puede escribir en el fichero
`boolean createNewFile()`	Crea el fichero asociado al objeto File. Devuelve true si se ha podido crear. Para poder crearlo el fichero no debe existir. Lanza una excepción del tipo **IOException**.
`boolean delete()`	Elimina el fichero o directorio. Si es un directorio debe estar vacío. Devuelve true si se ha podido eliminar.
`boolean exists()`	Devuelve true si el fichero o directorio existe
`String getName()`	Devuelve el nombre del fichero o directorio
`String getAbsolutePath()`	Devuelve la ruta absoluta asociada al objeto File.
`String getCanonicalPath()`	Devuelve la ruta única absoluta asociada al objeto File. Puede haber varias rutas absolutas asociadas a un File pero solo una única ruta canónica. Lanza una excepción del tipo **IOException**.
`String getPath()`	Devuelve la ruta con la que se creó el objeto File. Puede ser relativa o no.
`String getParent()`	Devuelve un String conteniendo el directorio padre del File. Devuelve null si no tiene directorio padre.

`File getParentFile()`	Devuelve un objeto File conteniendo el directorio padre del File. Devuelve null si no tiene directorio padre.
`boolean isAbsolute()`	Devuelve true si es una ruta absoluta
`boolean isDirectory()`	Devuelve true si es un directorio válido
`boolean isFile()`	Devuelve true si es un fichero válido
`long lastModified()`	Devuelve un valor en milisegundos que representa la última vez que se ha modificado (medido desde las 00:00:00 GMT, del 1 de Enero de 1970). Devuelve 0 si el fichero no existe o ha ocurrido un error.
`long length()`	Devuelve el tamaño en bytes del fichero. Devuelve 0 si no existe. Devuelve un valor indeterminado si es un directorio.
`String[] list()`	Devuelve un array de String con el nombre de los archivos y directorios que contiene el directorio indicado en el objeto File. Si no es un directorio devuelve null. Si el directorio está vacío devuelve un array vacío.
`String[] list(FilenameFilter filtro)`	Similar al anterior. Devuelve un array de String con el nombre de los archivos y directorios que contiene el directorio indicado en el objeto File que cumplen con el filtro indicado.
`boolean mkdir()`	Crea el directorio. Devuelve true si se ha podido crear.
`boolean mkdirs()`	Crea el directorio incluyendo los directorios no existentes especificados en la ruta *padre* del directorio a crear. Devuelve true si se ha creado el directorio y los directorios no existentes de la ruta padre.
`boolean renameTo(File dest)`	Cambia el nombre del fichero por el indicado en el parámetro dest. Devuelve true si se ha realizado el cambio.

Autoevaluación

Señala si la afirmación es verdadera o falsa:

Podemos establecer la fecha de modificación de un archivo mediante el método **`renameTo()`**.

○ Verdadero ○ Falso

Falso

Ese no es el método que hay que usar, se usaría **`setLastModified()`**.

2.1.1.- Existencia y listado de ficheros y carpetas.

Mediante la clase **`File`**, podemos ver si un fichero cualquiera, digamos por ejemplo **texto.txt**, existe o no. Para ello, nos valemos del método **`exists()`** de la clase **`File`**. Hacemos referencia a ese fichero en concreto con el código siguiente:

```
File f = new File("C:\texto.txt");
```

En el siguiente recurso puedes descargar una pequeña aplicación en la que se usa **`File`**. Como verás, permite listar el contenido de la carpeta que pongas en un campo de texto, introduciendo el resultado en un **`JList`**.

Listar archivos. (0.01 MB)

Pero, ¿qué pasa si nos interesa copiar un fichero, cómo lo haríamos?

Con la clase **`File`** no es suficiente, necesitamos saber más, en concreto, necesitamos hablar de los flujos, como vamos a ver más adelante.

Para saber más

En este enlace puedes ver ejemplos para obtener las propiedades de los ficheros, usando la clase **`File`**:

Propiedades de los ficheros

Ejemplo creando carpetas o directorios en Java

Autoevaluación

Señala la opción correcta. Con la clase `File` podemos:

- ○ Crear ficheros temporales.
- ○ Crear directorios.
- ○ Renombrar un archivo.
- ○ Todas son correctas.

No es correcto del todo.

Incorrecto.

No es la respuesta correcta.

Muy bien, efectivamente así es.

Solución

1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

2.1.2.- Creación y eliminación de ficheros y directorios.

Cuando queramos **crear un fichero**, podemos proceder del siguiente modo:

```
try {
// Creamos el objeto que encapsula el fichero
File fichero = new File("c:\\prueba\\miFichero.txt");
// A partir del objeto File creamos el fichero físicamenteif (fichero.createNewFile())
System.out.println("El fichero se ha creado correctamente");
else
System.out.println("No ha podido ser creado el fichero");
} catch (Exception ioe) {
ioe.getMessage();
}
```

Para **borrar un fichero** podemos usar la clase **`File`**, comprobando previamente si existe el fichero, del siguiente modo:

```
// Borrar fichero
File fichero = new File("c:\\prueba\\miFichero.txt");
if (fichero.exists ())
fichero.delete();
```

Para **crear un directorio**, lo realizaremos del siguiente modo:

```
Try {
// Declaración de variables
      String directorio = "C:\\prueba";
      String varios = "carpeta1/carpeta2/carpeta3";
// Crear un directorio
      boolean exito = (new File(directorio)).mkdir();
         if (exito)
         System.out.println("Directorio: " + directorio + " creado");
// Crear varios directorios
         exito = (new File(varios)).mkdirs();
              if (exito)
              System.out.println("Directorios: " + varios + " creados");
          }catch (Exception e){
     System.err.println("Error: " + e.getMessage());
}
```

Para **borrar un directorio** con Java, tendremos que borrar cada uno de los ficheros y directorios que este contenga. Al poder almacenar otros directorios, se podría recorrer recursivamente el directorio para ir borrando todos los ficheros.

Podemos listar el contenido de un directorio e ir borrando con:

```
File [] ficheros = directorio.listFiles ();
```

Si el elemento es un directorio, lo sabemos mediante el método **`isDirectory ().`**

Después de vaciar el directorio, este, se puede borrar importando la librería **`FileUtils`** y después escribir **`FileUtils.deleteDirectory(newFile(destination));`**

2.2.- Interface FilenameFilter.

Hemos visto como obtener la lista de ficheros de una carpeta o directorio. A veces, nos interesa ver no la lista completa, sino los archivos que encajan con un determinado criterio.

Por ejemplo, nos puede interesar un filtro para ver los ficheros modificados después de una fecha, o los que tienen un tamaño mayor del que el que indiquemos, etc.

El interface **`FilenameFilter`** se puede usar para crear filtros que establezcan criterios de filtrado relativos al nombre de los ficheros. Una clase que lo implemente debe definir e implementar el método:

```
boolean accept(File dir, String nombre)
```

Este método devolverá verdadero en el caso de que el fichero cuyo nombre se indica en el parámetro **`nombre`** aparezca en la lista de los ficheros del directorio indicado por el parámetro **`dir`**.

En el siguiente ejemplo vemos cómo se listan los ficheros de la carpeta **c:\datos** que tengan la extensión .txt. Usamos **`try`** y **`catch`** para capturar las posibles excepciones, como que no exista dicha carpeta.

```
import java.io.File;
import java.io.FilenameFilter;


public class Filtrar implements FilenameFilter {
String extension;
   // Constructor
   Filtrar(String extension){
           this.extension = extension;
    }
public boolean accept(File dir, String name){
     return name.endsWith(extension);
     }
public static void main(String[] args) {
try {
     // Obtendremos el listado de los archivos de ese directorio
     File fichero=new File("c:\\datos\\.");
     String[] listadeArchivos = fichero.list();
      // Filtraremos por los de extension .txt
      listadeArchivos = fichero.list(new Filtrar(".txt"));
      // Comprobamos el número de archivos en el listado
      int numarchivos = listadeArchivos.length ;
      // Si no hay ninguno lo avisamos por consola
      if (numarchivos < 1)
          System.out.println("No hay archivos que listar");
          // Y si hay, escribimos su nombre por consola.
          else
       {
             for(int conta = 0; conta < listadeArchivos.length; conta++)
             System.out.println(listadeArchivos[conta]);
     }
}
catch (Exception ex) {
System.out.println("Error al buscar en la ruta indicada"); }
}
```

En el ejemplo anterior se utiliza la función **`endsWith`**. Por si no sabes para que se emplea, y para ver otras más sobre tratamiento de cadenas, sigue este enlace: https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

Para saber más

En el ejemplo anterior se utiliza la función **`endsWith`**. Por si no sabes para que se emplea, y para ver otras más sobre tratamiento de cadenas, sigue este enlace:

Operaciones con cadenas.

2.3.- Rutas de los ficheros.

En los ejemplos que vemos en el tema, estamos usando la ruta de los ficheros tal y como se usan en MS-DOS, o Windows, es decir, por ejemplo:

```
C:\\datos\Programacion\fichero.txt
```

Cuando operamos con rutas de ficheros, el carácter separador entre directorios o carpetas suele cambiar dependiendo del sistema operativo en el que se esté ejecutando el programa.

Para evitar problemas en la ejecución de los programas cuando se ejecuten en uno u otro sistema operativo y, por tanto, persiguiendo que nuestras aplicaciones sean lo más portables posibles, se recomienda usar en Java: `File.separator`.

Podríamos hacer una función que, al pasarle una ruta, nos devolviera la adecuada según el separador del sistema actual, del siguiente modo:

Código de separador de rutas.

Autoevaluación

Cuando trabajamos con fichero en Java, no es necesario capturar las excepciones, el sistema se ocupa automáticamente de ellas.

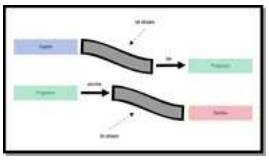
○ Verdadero ○ Falso

Falso
Las excepciones en Java hay que tratarlas adecuadamente para evitar malos funcionamientos de la aplicación.

3.- Flujos.

Caso práctico

Ana y **Antonio** saben que van a tener que ayudar a **Juan** y a **María** en labores de programación de ficheros en Java. Así que, además, de la clase `File`, van a necesitar utilizar otros conceptos relacionados con la entrada y salida: los flujos o streams. Ana recuerda que hay dos tipos de flujos: flujos de caracteres y flujos de bytes.



Un programa en Java, que necesita realizar una operación de entrada/salida (en adelante E/S), lo hace a través de un flujo o `stream`.

Un **flujo** es una **abstracción de todo aquello que produce o consume información.**

La vinculación de este flujo al dispositivo físico la hace el sistema de entrada y salida de Java.

Las clases y métodos de E/S que necesitamos emplear son las mismas independientemente del dispositivo con el que estemos actuando. Luego, el núcleo de Java sabrá si tiene que tratar con el teclado, el monitor, un sistema de archivos o un socket de red; liberando al programador de tener que saber con quién está interactuando.

Java define dos tipos de flujos en el paquete `java.io`:

- **Byte streams** (8 bits): proporciona lo necesario para la gestión de entradas y salidas de bytes y su uso está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene determinado por dos clases abstractas que son `InputStream` y `OutputStream` Estas dos clases definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan `read()`y `write()` que leen y escriben bytes de datos respectivamente.
- **Character streams** (16 bits): de manera similar a los flujos de bytes, los flujos de caracteres están determinados por dos clases abstractas, en este caso: `Reader` y `Writer`. Dichas clases manejan flujos de caracteres Unicode. Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados los métodos `read()` y `write()` que leen y escriben caracteres de datos respectivamente.

3.1.- Flujos basados en bytes.

Para el tratamiento de los flujos de bytes, hemos dicho que Java tiene dos clases abstractas que son **`InputStream`** y **`OutputStream`**.

Los archivos binarios guardan una representación de los datos en el archivo, es decir, cuando guardamos texto no guardan el texto en si, sino que guardan su representación en un código llamado UTF-8.

Las clases principales que heredan de **`OutputStream`**, para la escritura de ficheros binarios son:

- **`FileOutputStream`**: escribe bytes en un fichero. Si el archivo existe, cuando vayamos a escribir sobre él, se borrará. Por tanto, si queremos añadir los datos al final de éste, habrá que usar el constructor **`FileOutputStream(String filePath, boolean append)`**, poniendo **`append`** a **`true`**.
- **`ObjectOutputStream`**: convierte objetos y variables en vectores de bytes que pueden ser escritos en un **`OutputStream`**.
- **`DataOutputStream`**, que da formato a los tipos primitivos y objetos **`String`**, convirtiéndolos en un flujo de forma que cualquier **`DataInputStream`**, de cualquier máquina, los pueda leer. Todos los métodos empiezan por "**`write`**", como **`writeByte()`**, **`writefloat()`**, etc.

De **InputStream**, para lectura de ficheros binarios, destacamos:

- **`FileInputStream`**: lee bytes de un fichero.
- **`ObjectInputStream`**: convierte en objetos y variables los vectores de bytes leídos de un InputStream.

La escritura de un fichero Binario se hace como se muestra en el siguiente documento.

Resumen textual alternativo

Ejemplo: escritura a fichero.

En el siguiente ejemplo se puede ver cómo se escribe a un archivo binario con **`DataOutputStream`**:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class CopiaFicheros {
   public static void main(String args[]) {
      // Copiar ficheros
      File origen = new File("origen.txt");
      File destino = new File("destino.txt");
      try {
         InputStream in = new FileInputStream(origen);
         OutputStream out = new FileOutputStream(destino);
         byte[] buf = new byte[1024];
         int len;
         while ((len = in.read(buf)) > 0) {
           out.write(buf, 0, len);
         }
         in.close();
         out.close();
      } catch (IOException ioe) {
        ioe.printStackTrace();
      }
   }
}
```

Autoevaluación

Señala si la afirmación es verdadera o falsa:

Podemos escribir datos binarios a ficheros utilizando el método **`append`** de la clase **`DataOutputStream`**.

○ Verdadero ○ Falso

Falso

`append` es un parámetro para indicar que se va a añadir a un fichero, no es un método.

Internacionalización

Autoevaluación

Mediante las clases que proporcionan buffers se pretende que se hagan lecturas y escrituras físicas a disco, lo antes posible y cuantas más mejor.

○ Verdadero ○ Falso

Falso
La idea es que los accesos a disco sean los menos posibles.

4.- Formas de acceso a un fichero.

Caso práctico

El momento de programar ha llegado, y **Antonio** se pregunta qué opción será mejor para el acceso a los ficheros: si **acceso secuencial o aleatorio.** ¿Qué uso se le va a dar a estos ficheros?, ¿para qué van a servir cuando la aplicación informática esté en funcionamiento? Esa es la cuestión clave, piensa **Antonio.**

Hemos visto que en Java puedes utilizar **dos tipos de ficheros (de texto o binarios) y dos tipos de acceso a los ficheros (secuencial o aleatorio).** Si bien, y según la literatura que consultemos, a veces se distingue una tercera forma de acceso denominada concatenación, tuberías o pipes.

- **Acceso aleatorio:** los archivos de acceso aleatorio, al igual que lo que sucede usualmente con la memoria (RAM=Random Access Memory), permiten acceder a los datos en forma no secuencial, desordenada. Esto implica que el archivo debe estar disponible en su totalidad al momento de ser accedido, algo que no siempre es posible.
- **Acceso secuencial:** En este caso los datos se leen de manera secuencial, desde el comienzo del archivo hasta el final (el cual muchas veces no se conoce a priori). Este es el caso de la lectura del teclado o la escritura en una consola de texto, no se sabe cuándo el operador terminará de escribir.
- **Concatenación (tuberías o "pipes"):** Muchas veces es útil hacer conexiones entre programas que se ejecutan simultáneamente dentro de una misma máquina, de modo que lo que uno produce se envía por un "tubo" para ser recibido por el otro, que está esperando a la salida del tubo. Las tuberías cumplen esta función.

Citas para pensar

El futuro tiene muchos nombres. Para los débiles es lo inalcanzable. Para los temerosos, lo desconocido. Para los valientes es la oportunidad.

Víctor Hugo.

4.1.- Operaciones básicas sobre ficheros de acceso secuencial.

Como **operaciones más comunes en ficheros de acceso secuencial**, tenemos el acceso para:

- Crear un fichero o abrirlo para grabar datos.
- Leer datos del fichero.
- Borrar información de un fichero.
- Copiar datos de un fichero a otro.
- Búsqueda de información en un fichero.
- Cerrar un fichero.

Debes conocer

En este enlace puedes ver un ejemplo con buffer.

Acceder a web con ejemplo de flujo de salida con Buffer tipo secuencial

Ahora vamos a ver como buscar en un archivo secuencial. La idea es que al ser un fichero secuencial tenemos que abrirlo e ir leyendo hasta encontrar el dato que buscamos, si es que lo encontramos. Esto se muestra a continuación:

```
String busqueda = jTexField1.getText();
try {
// Declarar variable
DataInputStream archivo = null;
// Abrir el archivo
archivo = new DataInputStream ( new FileInputStream ("c:\\secuencial.dar") );
// Leer archivo
while (seguir) {
// Leer el nombre
nombre = archivo.readUTF();
// Si el nombre es el que buscamos
if (busqueda.equals (nombre) {
System.out.println ("encontrado");
seguir = false;
jLabel2.setText ("¡¡Registro encontrado!!");
}
// Leer los otros campos
apellidos = archivo.readUTF ();
edad = archivo.readInt ();
}
// Cerrar fichero
archivo.close ();
}
catch (FileNotFoundException fne) { / * Archivo no encontrado * / }
catch (IOException ioe) { / * Error al escribir * / }
```

Cuando se trabaja con ficheros de texto se recomienda usar las clases `Reader`, para entrada o lectura de caracteres, y `Writer` para salida o escritura de caracteres. Estas dos clases están optimizadas para trabajar con caracteres y con texto en general, debido a que tienen en cuenta que cada carácter Unicode está representado por dos bytes.

Las subclases de `Writer` y `Reader` que permiten trabajar con ficheros de texto son:

- `FileReader`, para lectura desde un fichero de texto. Crea un flujo de entrada que trabaja con caracteres en vez de con bytes.
- `FileWriter`, para escritura hacia un fichero de texto. Crea un flujo de salida que trabaja con caracteres en vez de con bytes.

También se puede montar un buffer sobre cualquiera de los flujos que definen estas clases:

- `BufferedWriter` se usa para montar un buffer sobre un flujo de salida de tipo `FileWriter`.
- `BufferedReader` se usa para montar un buffer sobre un flujo de entrada de tipo `FileReader`.

Autoevaluación

Señala si la afirmación es verdadera o falsa: La clase `Reader` está optimizada para trabajar con ficheros binarios.

○ Verdadero ○ Falso

Falso
Está optimizada para ficheros de texto.

4.2.- Operaciones básicas sobre ficheros de acceso aleatorio.

A menudo, no necesitas leer un fichero de principio a fin, sino simplemente acceder al fichero como si fuera una base de datos, donde se salta de un registro a otro; cada uno en diferentes partes del fichero. Java proporciona una clase **`RandomAccessFile`** para este tipo de entrada/salida.

Esta clase:

- Permite leer y escribir sobre el fichero, no es necesario dos clases diferentes.
- Necesita que le especifiquemos el modo de acceso al construir un objeto de esta clase: sólo lectura o bien lectura y escritura.
- Posee métodos específicos de desplazamiento como **`seek(long posicion)`** o **`skipBytes(int desplazamiento)`** para poder movernos de un registro a otro del fichero, o posicionarnos directamente en una posición concreta del fichero.

Por esas características que presenta la clase, un archivo de acceso directo tiene sus registros de un tamaño fijo o predeterminado de antemano.

La clase posee dos constructores:

- **`RandomAccessFile(File file, String mode).`**
- **`RandomAccessFile(String name, String mode).`**

En el primer caso se pasa un objeto **`File`** como primer parámetro, mientras que en el segundo caso es un String. El modo es: "r" si se abre en modo lectura o "rw" si se abre en modo lectura y escritura.

A continuación puedes ver una presentación en la que se muestra cómo abrir y escribir en un fichero de acceso aleatorio. También, en el segundo código descargable, se presenta el código correspondiente a la escritura y localización de registros en ficheros de acceso aleatorio.

Ejemplo:

Vamos a ver un pequeño ejemplo, **`Log.java`**, que añade una cadena a un fichero existente, lo crea en caso de que no exista.

```
import java.io.IOException;
import java.io.RandomAccessFile;
public class RandomEjemplo {
public static void main( String args[] ) throws IOException {
    RandomAccessFile miRAFile;
    String s = "linea a añadir al final del fichero";
    // Abrimos el fichero de acceso aleatorio
    miRAFile = new RandomAccessFile( "java.log","rw" );
    // Nos vamos al final del fichero
    miRAFile.seek( miRAFile.length() );
    // Incorporamos la cadena al fichero
    miRAFile.writeBytes( s );
    // Cerramos el fichero
    miRAFile.close();
    }
}
```

Autoevaluación

Indica si la afirmación es verdadera o falsa:

Un objeto de la clase **`RandomAccessFile`** necesita el modo de acceso al crear el objeto.

○ Verdadero ○ Falso

Verdadero
Como acabamos de ver, hay que darle el modo de acceso “r” para lectura o bien “rw” para lectura y escritura.

3.2.- Flujos basados en caracteres.

Para los flujos de caracteres, Java dispone de dos clases abstractas: **`Reader`** y **`Writer`**.

Si se usan sólo **`FileInputStream`**, **`FileOuputStream`**, **`FileReader`** o **`FileWriter`**, cada vez que se efectúa una lectura o escritura, se hace físicamente en el disco duro. Si se leen o escriben pocos caracteres cada vez, el proceso se hace costoso y lento por los muchos accesos a disco duro.

Los **`BufferedReader`**, **`BufferedInputStream`**, **`BufferedWriter`** y **`BufferedOutputStream`** añaden un buffer intermedio. Cuando se lee o escribe, esta clase controla los accesos a disco. Así, si vamos escribiendo, se guardarán los datos hasta que haya bastantes datos como para hacer una escritura eficiente.

Al leer, la clase leerá más datos de los que se hayan pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez físicamente. Esta forma de trabajar hace los accesos a disco más eficientes y el programa se ejecuta más rápido.

Además **`FileReader`** no contiene métodos que nos permitan leer líneas completas, pero sí **`BufferedReader`**. Afortunadamente, podemos construir un **`BufferedReader`** a partir del **`FileReader`** de la siguiente manera:

```
File archivo = new File ("C:\\archivo.txt");

FileReader fr = new FileReader (archivo);

BufferedReader br = new BufferedReader(fr);

...

String linea = br.readLine();
```

Ejemplo: lectura de un fichero

En este ejemplo leemos caracteres hasta llegar al fin de archivo (EOF o End Of File) del flujo.

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class LeerFichEOF {
    public static void main (String args[]) throws IOException {
            BufferedReader br = new BufferedReader(new FileReader("origen.txt")) ;
            int codigo = br.read();//lee el primer caracter
            char caracter;
        //mientras el código no sea -1 (EOF) continuo leyendo
            while (codigo != -1) {
                caracter = (char) codigo; //casting
                System.out.print(caracter);
                codigo = br.read(); //lee un caracter
                    }
        }
}
```

Debes conocer

Vídeo sobre el paquete java.io.

Vídeo sobre `java.io`.

Conoce mejor un paquete:...

Resumen textual alternativo

Para saber más

En este enlace puedes aprender más sobre internacionalización y Unicode.

4.3.- Enlaces de interés sobre ficheros.

Java IO (I)

Java IO(II)

Mejoras de Java IO

5.- Java NIO.

Una de las tareas más importante que realizan algunas aplicaciones es el manejo de la entrada y salida ya sea al sistema de ficheros o a la red. Desde las versiones iniciales de Java se ha mejorado soporte añadiendo programación asíncrona de E/S, permitir obtener información de atributos propios del sistema de archivos, reconocimiento de enlaces simbólicos y facilitado de algunas operaciones básicas.

Hemos usado durante muchos años java.io para trabajar con ficheros en el mundo Java . Se trata de un API muy potente y flexible que nos permite realizar casi cualquier tipo de operación. Sin embargo es un API complicada de entender. `Java NIO` (New IO) es un nuevo API disponible desde Java7 que nos permite mejorar el rendimiento así como simplificar el manejo de muchas cosas.

`Java.nio` **define interfaces y clases para que la máquina virtual Java tenga acceso a archivos, atributos de archivos y sistemas de archivos.** Aunque dicho API comprende numerosas clases, solo existen unas pocas de ellas que sirven de puntos de entrada al API, lo que simplifica considerablemente su manejo

`Java NIO`: **canales y búferes**

En la API de IO estándar, trabajas con secuencias de bytes y secuencias de caracteres. En NIO, trabaja con canales y búferes. Los datos siempre se leen de un canal a un búfer, o se escriben desde un búfer a un canal.

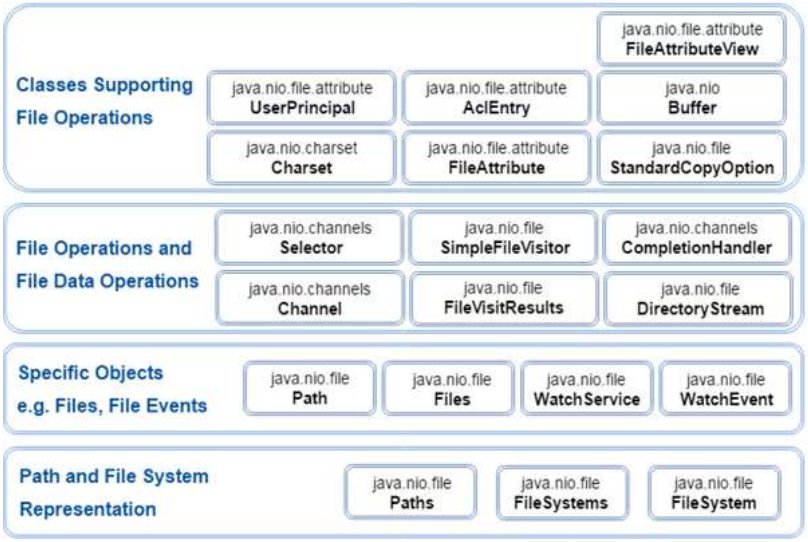
`Java NIO`: **IO sin bloqueo**

`Java NIO` le permite hacer IO sin bloqueo. Por ejemplo, un hilo puede pedirle a un canal que lea datos en un búfer. Mientras el canal lee datos en el búfer, el hilo puede hacer otra cosa. Una vez que se leen los datos en el búfer, el hilo puede continuar procesándolo. Lo mismo es cierto para escribir datos en canales.

`Java NIO`: **Selectores**

`Java NIO` contiene el concepto de "selectores". Un selector es un objeto que puede gestionar múltiples canales para eventos (como: conexión abierta, datos recibidos, etc.). Un Selector permite que un solo hilo maneje múltiples canales.

El estudio de la API NIO completa, excede el modulo, por lo que veremos lo mas básico. Vamos a estudiar en los subcapítulos siguiente las interfaces `Path` y `Files` que son las clases básicas para acceder a los ficheros.
En la imagen puedes ver las clases de las que se dispone en java.nio.



- La interfaz `java.nio.file.Path` representa un path y las clases que implementen esta interfaz puede utilizarse para localizar ficheros en el sistema de ficheros. Nos permite manejar rutas al estilo GNU/Linux y rutas al estilo Windows dependiendo del SO en el que estemos trabajando.
- La clase `java.nio.file.Files` es el otro punto de entrada a la librería de ficheros de Java. Es la que nos permite manejar ficheros reales del disco desde Java.

Pregunta Verdadero-Falso

Java NIO no contiene concepto de selectores

○ Verdadero ○ Falso

Falso
Java NIO entre otras características contiene selectores.

5.1.- Java NIO Path.

La interfaz `java.nio.file.Path` representa un path y las clases que implementen esta interfaz puede utilizarse para localizar ficheros en el sistema de ficheros .
Una ruta puede señalar a un archivo o un directorio. Un camino puede ser absoluto o relativo. Una ruta absoluta contiene la ruta completa desde la raíz del sistema de archivos hasta el archivo o directorio al que apunta. Una ruta relativa contiene la ruta al archivo o directorio relativo a alguna otra ruta.

La forma mas sencilla de construir un objeto que cumpla la interfaz Path es a partir de la clase **`java.nio.file.Paths`**, que **tiene métodos estáticos que retornan objetos Path a partir de una representación tipo String del path** deseado, por ejemplo:

```
Path p = Paths.get("/home/ad/mi_fichero");
```

Por supuesto, **no es necesario que los ficheros existan de verdad en el disco duro para que se puedan crear los objetos Path correspondientes**: La representación y manejo de paths en Java no está restringida por la existencia de esos ficheros o directorios en el sistema de ficheros.

El interfaz Path declara numerosos métodos que resultan muy útiles para el manejo de paths, como por ejemplo, obtener el nombre corto de un fichero, obtener el directorio que lo contiene, resolver paths relativos, etc.

Una instancia de tipo Path refleja el sistema de nombrado del sistema operativo subyacente, por lo que objetos path de diferentes sistemas operativos no pueden ser comparados fácilmente entre si.

Operaciones con Path

- Recuperar partes de una ruta
- Eliminar redundancias de una ruta
- Convertir una ruta
- Unir dos rutas
- Crear una ruta relativa a otra dada
- Comparar dos rutas

Ejemplo:

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class PathEjamplo {
public static void main(String args[]) {
    Path path = Paths.get("C:/Users/alumno/PathEjemplo");
    System.out.println(" path = " + path);
    System.out.println(" is absoute ? = " + path.isAbsolute());
    System.out.println(" file short name = " + path.getFileName());
    System.out.println(" parent = " + path.getParent());
    System.out.println(" uri = " + path.toUri());
    path = Paths.get("/home/PathEjemplo");
    System.out.println(" path = " + path);
    System.out.println(" is absoute ? = " + path.isAbsolute());
    System.out.println(" file short name = " + path.getFileName());
    System.out.println(" parent = " + path.getParent());
    System.out.println(" uri = " + path.toUri());
    }
}
```

Clase `FileSystem`

El concepto de **`FileSystem` define un sistema de ficheros completo.** Mientras que por otro lado el concepto de **Path hace referencia a un directorio, fichero o link que tengamos dentro de nuestro sistema de ficheros.** El siguiente código hace uso de **`FileSystem`** y Path para obtener el nombre de un fichero así como la carpeta padre en la que se encuentra ubicado.

Ejemplo:

```
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.util.Iterator;
    public class EjemploPath2 {
    public static void main(String args[]) {
    FileSystem sistemaFicheros = FileSystems.getDefault();
    Path rutaFichero = sistemaFicheros.getPath("C:\\Users\\alumno\\PathEjemplo");
    System.out.println(rutaFichero.getFileName());
    System.out.println(rutaFichero.getParent().getFileName());
    Path rutaDirectorio = sistemaFicheros.getPath("C:\\Users\\alumno");
    Iterator<Path> it = rutaDirectorio.iterator();
    while (it.hasNext()) {
         System.out.println(it.next().getFileName());
         }
```

```
    }
}
```

5.2.- Clases Java NIO Files.

La clase **`java.nio.file.Files`** es el otro punto de entrada a la librería de ficheros de Java. Es la que nos permite manejar ficheros reales del disco desde Java.

Esta clase tiene métodos estáticos para el manejo de ficheros, los métodos de la clase Files trabajan sobre objetos Path.

Las operaciones principales a realizar con archivos y directorios son:

- Verificación de existencia y accesibilidad
- Borrar un archivo o directorio
- Copiar un archivo o directorio
- Mover un archivo o directorio

Veamos cómo se realizan algunas de estas operaciones.

Ejemplo 1: Existencia y comprobación de permisos

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
public class FileEjemplo {
public static void main(String args[]) {
    Path path = Paths.get("C:\\Users\\alumno\\FileEjamplo\\hola.txt");
    System.out.println(" path = " + path);
    System.out.println(" exists = " + Files.exists(path));
    System.out.println(" readable = " + Files.isReadable(path));
    System.out.println(" writeable = " + Files.isWritable(path));
    System.out.println(" executeable = " + Files.isExecutable(path))
```

Ejemplo2: Creación y borrado de directorios

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
// Crea un nuevo fichero o directorio o lo borra si ya existe
public class FileEjemplo2 {
public static void main(String args[]) {
    Path path = Paths.get("C:\\Users\\alumno\\prueba");
    try {
    if (Files.exists(path))
         Files.delete(path);
         else
         Files.createFile(path);
      } catch (IOException e) {
     System.err.println(e);
      System.exit(1);
    }
}
}
```

El método **delete(Path)** borra el fichero o directorio o lanza una excepción si el borrado falla. El siguiente ejemplo muestra como capturar y gestionar las excepciones que pueden producirse en el borrado Si el fichero o directorio no existe la excepción que se produce es NoSuchFileException. Los sucesivos cath permiten determinar por que ha fallado el borrado:

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // File permission problems are caught here.
    System.err.println(x);
}
```

El metodo deleteIfExists(Path) tambien borra el fichero o directorio, pero no lanza ningun error en caso de que el fichero o directorio no exista.

Ejemplo3: crear un directorio

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.FileAlreadyExistsException;
```

```
import java.nio.file.Files;
import java.io.IOException;
// crea una nueva carpeta
public class FileEjemplo4
{
public static void main(String args[]) {
Path path = Paths.get("C:\\Users\\alumno\\newdir");
try {
    Path newDir = Files.createDirectory(path);
} catch(FileAlreadyExistsException e){
    // el directorio ya existe
} catch (IOException e) {
    //error I/O
              e.printStackTrace();
}
}
}
```

Ejemplo4: copiar directorios

Se puede copiar un archivo o directorio usando el método copy(Path, Path, CopyOption...). La copia falla si el archivo de destino existe, a menos que se especifique la opción REPLACE_EXISTING.

Se puede copiar directorios Aunque, los archivos dentro del directorio no se copian, por lo que el nuevo directorio está vacío incluso cuando el directorio original contiene archivos.

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.FileAlreadyExistsException;
import java.nio.file.Files;
import java.io.IOException;
//import java.nio.file.FileAlreadyExistsException; //en caso de querer sobrescribir el directorio destino
public class FileEjemplo5 {
public static void main(String args[]) {
    Path sourcePath = Paths.get("C:\\Users\\alumno\\origen");
    Path destinationPath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino");
    try {
    Files.copy(sourcePath, destinationPath);
                    //Files.copy(sourcePath, destinationPath, StandardCopyOption.REPLACE_EXISTING);
    } catch (FileAlreadyExistsException e) {
    System.out.println("el fichero existe");
     } catch (IOException e) {
     e.printStackTrace();
       }
}
}
```

Ejemplo 5: Copiar ficheros

```
import java.io.IOException;
import java.nio.file.FileAlreadyExistsException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
public class FileEjemplo7 {
public static void main(String args[]) {
    Path sourcePath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\hola.txt");
    Path destinationPath =Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino\\hola.txt");
    try {
       Files.copy(sourcePath, destinationPath, StandardCopyOption.REPLACE_EXISTING);
      } catch (FileAlreadyExistsException e) {
       System.out.println("el destino existe");
     } catch (IOException e) {
     e.printStackTrace();
  }
 }
}
```

Ejemplo 6: Mover ficheros y directorios, cambiando el nombre

```
import java.io.IOException;
import java.nio.file.FileAlreadyExistsException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
public class FileEjemplo7 {
```

```
public static void main(String args[]) {
Path sourcePath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\hola.txt");
Path destinationPath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino\\OtroNombre.txt");
try {
Files.move(sourcePath, destinationPath, StandardCopyOption.REPLACE_EXISTING);
} catch (FileAlreadyExistsException e) {
System.out.println("el destino existe");
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

5.3.- Escribir contenido en un fichero.

Modos de acceso, el parámetro `OpenOptions`

A la hora de utilizar un fichero en Java se puede restringir el acceso que tenemos al mismo desde el propio lenguaje, haciendo más estrictos los permisos de acceso que dicho fichero ya tiene en el sistema de ficheros.

Por ejemplo, si el usuario tiene permisos de lectura y escritura sobre un fichero, un programa Java que solo quiera leerlo puede abrir el fichero solo en modo lectura, lo que ayudará a evitar bugs desde el propio lenguaje.

A tal efecto, en java se definen una serie de modos de acceso a un fichero a través del parámetro **`OpenOptions`**. La forma más cómoda de utilizar este parámetro es a través del **`enum StandardOpenOptions`** que puede tomar los siguientes valores (hay más):

- **`WRITE`**: habilita la escritura en el fichero
- **`APPEND`**: todo lo escrito al fichero se hará al final del mismo
- **`CREATE_NEW`**: crea un fichero nuevo y lanza una excepción si ya existía
- **`CREATE`**: crea el fichero si no existe y simplemente lo abre si ya existía
- **`TRUNCATE_EXISTING`**: si el fichero existe, y tiene contenido, se ignora su contenido para sobreescribirlo desde el principio.

Los métodos que se muestran en las siguientes ejemplos utilizan este parámetro, en la descripción de cada método en el API se explica cual es el comportamiento por defecto en caso de no utilizarse este parámetro.

Escritura desde arrays de bytes

La escritura a ficheros mediante arrays es la forma más sencilla (y limitada) de escritura de ficheros, y se realiza mediante el método **`java.nio.file.Files.write()`.**

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
public class FileEjemplo10 {
    public static void main(String args[]) {
        Path inputFile = Paths.get("C:\\Users\\alumno\\FileEjamplo\\origen\\hola.txt");
        Path outputFile = Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino\\hola.txt");
        try {
            byte[] contents = Files.readAllBytes(inputFile);
            Files.write(outputFile, contents, StandardOpenOption.WRITE,
            StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
       } catch (IOException e) {
            System.err.println(" ERROR : " + e);
            System.exit(1);
     }
  }
}
```

Escritura desde buffers
La escritura desde buffers resulta mucho más eficiente que utilizando arrays de bytes para ficheros grandes.

El siguiente programa Java copia ficheros, accediendo al fichero original una
vez por línea y escribiendo en el fichero destino una línea cada vez. Utiliza las clases de **`java.io: BufferedReader`** y **`BufferedWriter:`**

```
import java .nio. file. Path;
import java .nio. file. Paths ;
import java .nio. file. Files ;
import java .io. IOException;
import java .nio. charset . Charset ;
import java .io. BufferedReader;
import java .io. BufferedWriter;
import java .nio. file. StandardOpenOption;
public class FileEjemplo11 {
// Copy a file
public static void main( String args []) {
Path input = Paths . get( "C:\\Users\\alumno\\FileEjamplo\\origen\\hola.txt") ;
Path output = Paths . get("C:\\Users\\alumno\\FileEjamplo\\destino\\hola.txt") ;
try {
        BufferedReader inputReader = Files . newBufferedReader(input , Charset .
        defaultCharset());
         BufferedWriter outputWriter = Files . newBufferedWriter(output , Charset .
         defaultCharset() ,  StandardOpenOption. WRITE , StandardOpenOption.
         CREATE ,  StandardOpenOption. TRUNCATE_EXISTING);
         String line;
         while ( ( line = inputReader. readLine ()) != null ) {
                outputWriter. write (line , 0, line. length ());
                outputWriter. newLine () ;
         }
         inputReader. close ();
         outputWriter. close ();
         } catch ( IOException e) {
         System .err. println (" ERROR : " + e);
```

```
            System . exit (1);
        }
    }
}
```

Enlaces de interés

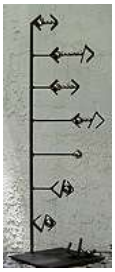
Diferencias entre Java NIO e IO

Serialización de objetos en Java

6.- Trabajo con ficheros XML: analizadores sintácticos (parser) y vinculación (binding).

Caso práctico

María y **Juan** están trabajando en un proyecto común, una aplicación para "Farmacia Manolín de Benidorm". Hay una parte de la aplicación que no hicieron ellos, sino el hermano del farmacéutico, que tiene ciertos conocimientos, por su interés autodidacta de la informática. Parece ser que Arturín, el hermano de Manolín, utilizó ficheros XML para guardar información de la aplicación. Juan y María están contemplando la posibilidad de aprovechar lo que hay desarrollado, estudiando la aplicación y viendo si podrían analizar los ficheros XML que ya existen para obtener la información que desean.



Probablemente hayas estudiado ya XML, bien porque hayas cursado el módulo **Lenguajes de marcas y sistemas de gestión de información**, o bien porque lo conozcas por tu cuenta. Si no conoces XML, te recomendamos que te familiarices con él, hay múltiples tutoriales y cursos en Internet.

El metalenguaje XML se crea para evitar problemas de interoperabilidad entre plataformas y redes. Con él se consigue un soporte estándar para el intercambio de datos: no sólo los datos están en un formato estándar sino también la forma de acceder a ellos. Y entre las ventajas de su uso destacamos:

- **Facilita el intercambio de información** entre distintas aplicaciones ya que se basa en estándares aceptados.
- **Proporciona una visión estructurada de la información**, lo que permite su posterior tratamiento de forma local.

6.1.- Conceptos previos.

¿Cómo se trabaja con datos XML desde el punto de vista del desarrollador de aplicaciones?

Una aplicación que consume información XML debe:

- Leer un fichero de texto codificado según dicho estándar.
- Cargar la información en memoria y, desde allí...
- Procesar esos datos para obtener unos resultados (que posiblemente también almacenará de forma persistente en otro fichero XML).

El proceso anterior se enmarca dentro de lo que se conoce en informática como **"parsing"** o **análisis léxico-sintáctico**, y los programas que lo llevan a cabo se denominan "parsers" o analizadores (léxico-sintácticos). Más específicamente podemos decir que el "parsing XML" es el proceso mediante el cual se lee y se analiza un documento XML para comprobar que está bien formado para, posteriormente, pasar el contenido de ese documento a una aplicación cliente que necesite consumir dicha información.

Schema: Un esquema (o schema) es una especificación XML que dicta los componentes permitidos de un documento XML y las relaciones entre los componentes. Por ejemplo, un esquema identifica los elementos que pueden aparecer en un documento XML, en qué orden deben aparecer, qué atributos pueden tener, y qué elementos son subordinados (esto es, son elementos hijos) para otros elementos. Un documento XML no tiene por qué tener un esquema, pero si lo tiene, debe atenerse a ese esquema para ser un documento XML válido.

Para saber más

En el siguiente enlace de la wikipedia puedes ver el concepto de esquema y algún que otro ejemplo:

Esquemas XML

Autoevaluación

Di si la afirmación es verdadera o falsa:

XML hace más fácil el intercambio de información entre sistemas.

○ Verdadero ○ Falso

Verdadero
XML estructura muy bien la información por lo que facilita el intercambio de información.

6.3.- "Parser" o analizador XML.

Introducción

Cuando se quieren almacenar datos que deban ser leídas por aplicaciones ejecutadas en múltiples plataformas será necesario recurrir a formatos más estandarizados, como los lenguajes de marcas.

Los documentos XML consiguen estructurar la información intercalando una serie de marcas denominadas etiquetas. En XML , las marcas o etiquetas tienen cierta similitud con un contenedor de información. Así, una etiqueta puede contener otras etiquetas o información textual. De este modo, conseguiremos subdividir la información estructurando de forma que pueda ser fácilmente interpretada.

Como toda la información es textual, no existe el problema de representar los datos de diferente manera. Cualquier dato, ya sea numérica o booleana, habrá transcribirla en modo texto, de modo que cualquiera que sea el sistema de representación de datos será posible leer e interpretar correctamente la información contenida en un archivo XML .

Es cierto que los caracteres se pueden escribir usando también diferentes sistemas de codificación, pero XML ofrece diversas técnicas para evitar que esto sea un problema. Por ejemplo, es posible incluir en la cabecera del archivo que codificación se ha utilizado durante el almacenamiento, o también se pueden escribir los caracteres de código ASCII superior a 127, utilizando entities de carácter , una forma universal de codificar cualquier símbolo .

XML consigue estructurar cualquier tipo de información jerárquica. Se puede establecer cierta similitud con la forma como la información se almacena en los objetos de una aplicación y la forma como se almacenaría en un documento XML . La información, en las aplicaciones orientadas a objeto, estructura, agrupa y jerarquiza en clases, y en los documentos XML se estructura, organiza y jerarquiza en etiquetas contenidas unas dentro de otras y atributos de las etiquetas

Analizador XML

Dado que XML es un lenguaje utilizado ámpliamente en el desarrollo de la World Wide Web, existen ya herramientas y estándares de programación para leer documentos XML. Las herramientas o programas que leen el lenguaje XML y comprueban si el documento es válido sintácticamente, se denominan analizadores o "parsers".

Un parser XML es un módulo, biblioteca o programa que se ocupa de analizar, clasificar y transformar un archivo de XML en una representación interna, extrayendo la información contenida en cada una de las etiquetas y relacionándola de acuerdo con su posición en la jerarquía.

En el caso de XML, como el formato siempre es el mismo, no necesitamos crear un parser cada vez que hacemos un programa, sino que existen un gran número de parsers o analizadores sintácticos disponibles que pueden averiguar si un documento XML cumple con una determinada gramática. Estos analizadores o parsers pueden ser secuenciales como SAX o STAX o jerarquicos como DOM.

Analizadores secuenciales

Los analizadores secuenciales, que permiten extraer el contenido a medida que se van descubriendo las etiquetas de apertura y cierre, se denominan analizadores sintácticos. Son analizadores muy rápidos, pero presentan el problema de que cada vez que se necesita acceder a una parte del contenido necesario releer todo el documento de arriba a abajo.

En Java hay dos analizadores secuenciales: SAX, que es el acrónimo de Simple API for XML . Es una analizador muy usado en varias bibliotecas de tratamiento de datos XML , pero no suele usarse en aplicaciones finales y STAX, Streaming API for XML, posterior a SAX y que lo ha superado.

Analizadores jerárquicos

Generalmente, las aplicaciones finales que necesitan trabajar con datos XML suelen usar analizadores jerárquicos, porque además de realizar un análisis secuencial que les permite clasificar el contenido, se almacenan en la memoria RAM siguiendo la estructura jerárquica detectada en el documento. Esto facilita mucho las consultas que haya que repetir varias veces, dado que las estructuras jerárquicas de la memoria RAM tienen un rendimiento de acceso parcial a los datos muy eficiente.

Los analizadores jerárquicos guardan todos los datos del XML en memoria dentro de una estructura jerárquica. Son ideales para aplicaciones que requieran una consulta continua de los datos. El formato de la estructura donde se almacena la información en la memoria RAM ha sido especificado por el organismo internacional W3C (World Wide Web Consortium) y se suele conocer como DOM (Document Object Model). Es una estructura que HTML y javascript han popularizado mucho y se trata de una especificación que Java materializa en forma de interfaces. La principal se denomina Documento y representa todo un documento XML . Al tratarse de una interfaz, puede ser implementada por varias clases.

Comparativa analizadores

CARACTERÍSTICA	STAX	SAX	DOM	TRAX
Tipo de API	Pull, streaming	Push, streaming	En memoria	Regla XSLT
Facilidad de uso	Alta	Media	Alta	Media
Capacidad XPath	No	No	Si	Si
Eficiencia de CPU y memoria	Buena	Buena	Varia	Varia
Solo hacia adelante	Si	Si	No	No

Lee XML	Si	Si	Si	Si
Escribe XML	Si	No	Si	Si
Crear, Leer, Modificar, Borrar	No	No	Si	No

6.4.- DOM.

DOM (Document Object Model) es una recomendación oficial del World Wide Web Consortium (W3C). Define una interfaz que permite a los programas acceder y actualizar el estilo, la estructura y el contenido de los documentos XML. Los analizadores XML compatibles con DOM implementan esta interfaz.

El estándar W3C define la especificación de la clase **`DocumentBuilder`** con el propósito de poder instanciar estructuras DOM a partir de un XML . La clase **`DocumentBuilder`** es una clase abstracta, y para que se pueda adaptar a las diferentes plataformas, puede necesitar fuentes de datos o requerimientos diversos. Recuerda que las clases abstractas no se pueden instanciar de forma directa. Por este motivo, el consorcio W3 especifica también la clase **`DocumentBuilderFactory`**

Las instrucciones necesarias para leer un archivo XML y crear un objeto Documento serían las siguientes:

```
DocumentBuilderFactory dbFactory = DocumentBuilderFactory. newInstance ;
DocumentBuilder dBuilder = dbFactory. newDocumentBuilder ;
Document doc = dBuilder. parse ( new File ( "fitxer.xml" ) ) ;
```

La escritura de la información contenida en el DOM se puede secuenciar en forma de texto utilizando otra utilidad de Java llamada **`Transformer`** . Se trata de una utilidad que permite realizar fácilmente conversiones entre diferentes representaciones de información jerárquica. Es capaz, por ejemplo, de pasar la información contenida en un objeto Documento a un archivo de texto en formato XML . También sería capaz de hacer la operación inversa, pero el mismo **`DocumentBuilder`** ya se encarga de ello.

`Transformer` es también una clase abstracta y requiere de una *fábrica* para poder ser instanciada. La clase **`Transformer`** puede trabajar con multitud de contenedores de información porque en realidad trabaja con un par de tipos adaptadores (clases que hacen compatibles jerarquías diferentes) que se llaman **`Source`** y **`Result`**. Las clases que implementen estas interfaces se encargarán de hacer compatible un tipo de contenedor específico al requerimiento de la clase **`Transformer`** . Así, disponemos de las clases **`DOMSource`**, **`SAXSource`** **O** **`StreamSorce`** como adaptadores del contenedor de la fuente de información (DOM, SAX o Stream respectivamente). **`DOMResult`**, **`SAXResult`** **O** **`StreamResult`** son los adaptadores equivalentes del contenedor destino.

El código básico para realizar una transformación de DOM archivo de texto XML sería el siguiente:

```
// Creación de una instancia Transformer
Transformer trans = TransformerFactory. newInstance . newTransformer ;
// Creación de los adaptadores Source y Results a partir de un Documento
// y un File.
StreamResult result =  new StreamResult ( file ) ;
DOMSource source =  new DOMSource ( doc ) ;
trans.transform ( source, result ) ;
```

Con el fin de rebajar la complejidad, vamos a crear una clase que llamaremos **`XmlCtrlDomcon`** utilidades genéricas que nos simplifiquen el traspaso de archivos XML a DOM o viceversa.

```
public  class XmlCtrlDom {
    public  static  Document instanciarDocument
        throws ParserConfigurationException {
        Document doc = null ;
        doc = DocumentBuilderFactory. newInstance . newDocumentBuilder . newDocument ;
        return doc ;
    }
    public  static  void escriureDocumentATextXml ( Document doc, File file )
      throws TransformerException {
        Transformer trans = TransformerFactory. newInstance . newTransformer ;
        trans.setOutputProperty ( OutputKeys. indent , "yes" ) ;
        StreamResult result =  new StreamResult ( file ) ;
        DOMSource source =  new DOMSource ( doc ) ;
        trans. transform ( source, result ) ;
      }
    public  static  Document instanciarDocument ( File fXmlFile )
                            throws ParserConfigurationException,
                                   SAXException,
                                   IOException {
        Document doc = null ;
        doc = DocumentBuilderFactory. newInstance.newDocumentBuilder . parse ( fXmlFile ) ;
        doc. getDocumentElement.normalize ;
        return doc ;
    }
    ...
}
```

6.2.- Definiciones.

¿Qué es y para qué sirve **JAXB** (Java Architecture for XML Binding)? JAXB simplifica el acceso a documentos XML representando la información obtenida de los documentos XML en un programa en formato Java, o sea, proporciona a los desarrolladores de aplicaciones Java, una forma rápida para vincular esquemas XML a representaciones Java.

JAXB proporciona métodos para, a partir de documentos XML, obtener árboles de contenido (generados en código Java), para después operar con ellos o manipular los los mismos en una aplicación Java y generar documentos XML con la estructura de los iniciales, pero ya modificados.

Parsear un documento XML consiste en "escanear" el documento y dividirlo o separarlo lógicamente en piezas discretas. El contenido parseado está entonces disponible para la aplicación.

Binding: Binding o vincular un esquema (schema) significa generar un conjunto de clases Java que representan el esquema.

Compilador de esquema o schema compiler: liga un esquema fuente a un conjunto de elementos de programa derivados. La vinculación se describe mediante un lenguaje de vinculación basado en XML.

>Binding runtime framework: proporciona operaciones de unmarshalling y marshalling para acceder, manipular y validar contenido XML usando un esquema derivado o elementos de programa.

Marshalling: es un proceso de codificación de un objeto en un medio de almacenamiento, normalmente un fichero. Proporciona a una aplicación cliente la capacidad para convertir un árbol de objetos Java JAXB a ficheros XML. Por defecto, el marshaller usa codificación UTF-8 cuando genera los datos XML.

Unmarshalling: proporciona a una aplicación cliente la capacidad de convertir datos XML a objetos Java JAXB derivados.

Para saber más

Hay muchos "parsers" conocidos, como **SAX** (Simple API for XML). SAX es un API para parsear ficheros XML. Proporciona un mecanismo para leer datos de un documento XML. Otra alternativa es DOM (Document Object Model).

Tienes más información sobre DOM en la wikipedia:

DOM

En este enlace se ve un ejemplo de cómo parsear un fichero XML mediante SAX.

Parsear

Resumen textual alternativo

6.4.1.- La estructura DOM.

La estructura DOM toma la forma de un árbol, donde cada parte del XML se encontrará representada en forma de nodo. **En función de la posición en el documento XML, hablaremos de diferentes tipos de nodos. El nodo principal que representa todo el XML entero denomina documento**, y las diversas **etiquetas**, incluida la etiqueta raíz, se conocen como **nodos elemento.** El **contenido textual** de una etiqueta se instancia como **nodo de tipo** `TextElement` y los **atributos** como nodos de **tipo** `Atribute`**.** Cada nodo específico dispone de métodos para acceder a sus datos concretos (nombre, valor, nodos hijos, nodo padre, etc.).

El DOM resultante obtenido de un XML termina siendo una copia exacta del archivo, pero con una disposición distinta. Tanto el DOM como el XML tendrán información no visible, como los retornos de carro, que deben tenerse en cuenta para saber cómo procesar correctamente el contenido y evitar sorpresas un poco comprensibles.

Debes tener en cuenta que al mapear XML los retornos de carro se interpretan en DOM como un *hijo* y que el contenido textual de las etiquetas se plasma en el DOM como un nodo hijo de la etiqueta contenedora. Es decir, para obtener el texto de una etiqueta hay que obtener el primer hijo de ésta.

La interfaz `Documentcontempla` un conjunto de métodos para seleccionar diferentes partes del árbol a partir del nombre de la etiqueta o de un atributo identificador. Las partes del árbol se devuelven como objetos `Element`, los cuales representan un nodo y todos sus hijos. De este modo, podremos ir explorando partes del árbol sin necesidad de tener que pasar por todos los nodos.

interfaces DOM

DOM define varias interfaces, las mas comunes son las siguientes

- `Node` – Representa a cualquier nodo del documento.
- `Element` – expone propiedades y métodos para manipular los elementos del documento y sus atributos.
- `Attr` – Representa un atributo de un elemento.
- `Text` – Contenido de un elemento o atributo.
- `Document` – Representa al documento XML completo. Generalmente nos referiremos a el como árbol DOM. Proporciona información del documento. Permite crear nuevos nodos en el documento.
- `NodeList`. Colección de nodos a los que se puede acceder por medio de un índice.

metodos DOM mas usuales

- `Document.getDocumentElement()` – Retorna el elemento raiz del documento.
- `Node.getFirstChild()` – Retorna el primer nodo hijo del nodo.
- `Node.getLastChild()` – Retorna el ultimo nodo hijo del nodo.
- `Node.getNextSibling()` – Retorna el siguiente hermano de un nodo.
- `Node.getPreviousSibling()` – Retorna el hermano anterior de un nodo.
- `Node.getAttribute(attrName)` – Retorna el atributo del nodo con el nombre que se pasa como atributo.

Para facilitar la obtención del contenido de un `Element` ampliaremos las utilidades de la clase `XmlCtrlDom` añadiendo dos métodos más.

```
public static String getValorEtiqueta ( String etiqueta, Element elemento ) {
            Nodo nValue = elemento. getElementsByTagName ( etiqueta ) . item ( 0 ) ;
            return nValue. getChildNodes . item ( 0 ) . getNodeValue ;
}

public static Elemento getElementEtiqueta ( String etiqueta, Element elemento ) {
            return ( Element ) elemento. getElementsByTagName ( etiqueta ).item ( 0 ) ;
}
```

El primero recibe el nombre de la etiqueta y el elemento (o nodo parcial del árbol) a partir del cual se desea realizar la búsqueda. Utilizando el método `getElementsByTagName`, conseguiremos todos los nodos que tengan por nombre el valor del parámetro etiqueta. Es decir, nos devolverá una colección de nodos. Si sólo existe un único nodo con el nombre del parámetro, éste ocupará la primera posición de la lista. Por ello accedemos con el método item, indicando que nos interesa el primer elemento (posición cero).

El segundo método es muy similar al primero, pero en vez de recuperar el texto, obtendremos el nodo con todos los hijos que tenga. Es útil para aplicar a nodos intermedios (no textuales).

Los objetos Elemento disponen de métodos para añadir nuevos hijos (`appendChild`) o asignar el valor a un atributo (`setAtributte`). También permiten la consulta del valor de los atributos (`getAttributte`) o la navegación por los nodos del árbol (`getParentNode`, para obtener el padre; `getFirstChild/getLastChild`, para obtener el primer / último hijo, o `getNextSiblingpara` navegar de hermano en hermano).

El objeto Document hará de factory para cualquier nodo del documento. La creación de nuevos elementos (etiquetas) implicará el uso de `createElement`. Si queremos crear contenido textual habrá llamar al método `createTextNode`, y si lo que queremos son comentarios llamaremos `createComment`.

La creación de nodos no implica la ubicación de este dentro del árbol. Es decir, además de crearlos habrá asignarlos a un padre usando `appendChild`.

6.4.2.- Ejemplo completo.

El ejemplo siguiente muestra cómo utilizar DOM para analizar y extraer información un fichero XML.

Listamos la información contenida en el documento **`clase.xml`**, si etiquetas.

```
clase.xml
<? xml version = "1.0"?>

<clase>
   <alumno numero = "393">
      <nombre> Luis </ nombre>
      <apellido> Luna </ apellido>
      <apodo> Na </ apodo>
      <marcas> 85 </ marcas>
   </ alumno>
<alumno numero = "493">
      <nombre> Antonio </ nombre>
      <apellido> Alvarez</ apellido>
      <apodo> Avez </ apodo>
      <marcas> 95 </ marcas>
   </ alumno>
<alumno numero = "593">
      <nombre> Juan </ nombre>
      <apellido> Juz </ apellido>
     <apodo> jazz </ apodo>
      <marcas> 90 </ marcas>
   </ alumno>
</ clase>
```

Pasos a seguir...

1.- Importar paquetes XML.

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
```

2.- Crear un SAXBuilder.

```
DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

3.- Crear un documento para el flujo de datos.

```
StringBuilder xmlStringBuilder = new StringBuilder();
xmlStringBuilder.append("<?xml version="1.0"?> <class> </class>");
ByteArrayInputStream input = new ByteArrayInputStream(
   xmlStringBuilder.toString().getBytes("UTF-8"));
Document doc = builder.parse(input);
```

4.- Extraer el elemento raíz.

```
Element root = document.getDocumentElement();
```

5.- Examinar atributos.

```
//returns specific attribute
getAttribute("attributeName");

//returns a Map (table) of names/values
getAttributes();
```

6.- Examinar subelementos.

```
//returns a list of subelements of specified name
getElementsByTagName("subelementName");

//returns a list of all child nodes
getChildNodes();
```

El programa completo...

```
import java.io.File;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
public class DomParserDemo {
public static void main(String[] args) {
    try {
         File inputFile = new File("clase.xml");
         DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
         DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
         Document doc = dBuilder.parse(inputFile);
         doc.getDocumentElement().normalize();
         System.out.println("Root element :" +
         doc.getDocumentElement().getNodeName());
         NodeList nList = doc.getElementsByTagName("alumno");
         System.out.println("----------------------------");
         for (int temp = 0; temp < nList.getLength(); temp++) {
            Node nNode = nList.item(temp);
            System.out.println("\nCurrent Element :" + nNode.getNodeName());
            if (nNode.getNodeType() == Node.ELEMENT_NODE) {
                Element eElement = (Element) nNode;
                System.out.println("numero de alumno : "+ eElement.getAttribute("numero"));
                System.out.println("nombre : "+ eElement.getElementsByTagName("nombre").item(0).getTextContent());
                System.out.println("apellido :"+ eElement.getElementsByTagName("apellido").item(0).getTextContent());
                System.out.println("apodo : "+ eElement.getElementsByTagName("apodo").item(0).getTextContent());
                System.out.println("marcas : "+eElement.getElementsByTagName("marcas").item(0).getTextContent());
             }
            }
          } catch (Exception e) {
          e.printStackTrace();
       }
    }
}
```

Más ejemplos

6.4.3.- Creación de un fichero XML a partir de un documento.

La escritura de la información contenida en el DOM se puede secuenciar en forma de texto utilizando otra utilidad de Java llamada **`Transformer`** . Se trata de una utilidad que permite realizar fácilmente conversiones entre diferentes representaciones de información jerárquica. Es capaz, por ejemplo, de pasar la información contenida en un objeto Documento a un archivo de texto en formato XML .

`Transformer` es también una clase abstracta y requiere de una fábrica para poder ser instanciada. La clase **`Transformer`** puede trabajar con multitud de contenedores de información porque en realidad trabaja con un par de tipos adaptadores (clases que hacen compatibles jerarquías diferentes) que se llaman **`Source`** y **`Result`**. Las clases que implementen estas interfaces se encargarán de hacer compatible un tipo de contenedor específico al requerimiento de la clase **`Transformer`**. Así, disponemos de las clases **`DOMSource, SAXSource`** o **`StreamSorce`** como adaptadores del contenedor de la fuente de información (DOM, SAX o Stream respectivamente). **`DOMResult, SAXResult`** o **`StreamResult`** son los adaptadores equivalentes del contenedor destino.

El código básico para realizar una transformación de DOM archivo de texto XML sería el siguiente:

1.- Instanciar el documento DOM en memoria.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
DOMImplementation implementation = builder.getDOMImplementation();
Document document = implementation.createDocument(null, name, null);
```

2.- El documento DOM se crea en memoria creando primero el elemento raíz.

```
Element raiz = document.getDocumentElement();
```

y a partir de el añadiendo otros elementos, los métodos:

```
createElement(String tagName) //crea el elemento del tipo especificado
createTextNode(String data)//crea un Textnode dando valor
(Node newChild)
appendChild(Node newChild)
//añade un nodo hijo al final de de la lista de nodos hijos del nodo3.- Creación e instalación Transformer.
```

```
<span>Transformer trans = TransformerFactory.newInstance .newTransformer ;<br></span><span id="yui_3_17_2_1_1596016483417_136">
// Creación de los adaptadores Source y Results a partir de un Documento
<br></span><span>
// y un File.<br></span><span>StreamResult result = new StreamResult ( file ) ;
<br></span><span>DOMSource source = new DOMSource ( doc ) ;<br></span>
<span>trans. transform ( source, result ) ;</span>
```

Además

Puede encontrar otro ejemplo completo en el siguiente enlace:

Ejemplo

6.5.- SAX.

SAX (*API Simple para XML),* es una interfaz simple de aplicaciones XML, fácil e intuitiva. El parser trabaja de la siguiente forma:

1. Lee un fichero XML de forma secuencial, produciendo eventos de manera secuencial en función de los resultados de lectura.
2. Cada evento invoca a un método que ha sido realizado por el programador procesando el documento poco a poco y no consumiendo prácticamente memoria, aunque por otra parte impide tener visión general de todo el documento XML.

Permite analizar el documento XML de forma secuencial, es decir, va cargando en memoria diferentes partes del mismo, cosa contraria a lo que hace DOM que carga todo el documento de golpe en memoria.

Es conveniente utilizar SAX:

- Cuando el documento XML es considerablemente grande.
- Cuando no se requiere una modificación estructural.
- Si se quiere parsear diferentes partes del documento.

La lectura de un documento XML produce eventos los cuales invocan a métodos, los eventos son: inicio y fin de un archivo XML(`startDocument()` y `endDocument()`), inicio y fin de un elemento (`startElement()` y `endElement()`) y la información que llevan las diferentes etiquetas (`characters()`).

SAX genera un objeto procesador de XML llamado XMLReader, para después decir que objetos tienen métodos para transferir los eventos. Estos objetos implementan los siguientes interfaces:

- `ContentHandler`, el cual recibe las notificaciones de los eventos del fichero XML.
- `DTDHandler`, que recoge los eventos del DTD del fichero.
- `ErrorHandler`, crea el tratamiento de errores.
- `EntityResolver`, se utilizan siempre que se referencia a otra entidad.
- `DefaultHandler`, la cual implementa por defecto los métodos, siendo el programador quien los defina. Con esta clase se podrá crear el parser XML. Se compone de los siguientes eventos básicos:
  - `startDocument`: se llama cuando se detecta que el documento empieza. Aquí deben indicarse las acciones a realizar al inicio del documento.
  - `endDocument`: se llama cuando se detecta que el documento ha acabado. Por lo tanto, aquí deben indicarse las acciones a realizar al finalizar el documento.
  - `startElement`: Se llama cuando encuentra un nuevo elemento, nodo, etiqueta, tag, etc. Aquí debe indicarse el tratamiento que se deberá realizar sobre cada nuevo elemento, como la recogida de información de sus atributos.
  - `endElement`: Se llama una vez ha leído el elemento. Aquí se encuentra la información del nodo y del contenido del nodo por lo que es aquí donde, normalmente, se recoge la información.
  - `Characters`: se invoca para encontrar una cadena de texto.

Pregunta Verdadero-Falso

El SAX lee todo el fichero .xml y lo muestra

○ Verdadero ○ Falso

Falso

El SAX según va leyendo va mostrando toda la información poco a poco y no toda de golpe.

6.6.- StAX.

Introducción

StAX es una API basada en Java para analizar documentos XML de forma similar a como lo hace el analizador SAX. Pero hay dos diferencias principales entre las dos API:

- StAX es una API PULL, mientras que SAX es una API PUSH. Significa que en el caso del analizador StAX, una aplicación cliente necesita solicitar al analizador StAX que obtenga información de XML siempre que lo necesite. Pero en el caso del analizador SAX, se requiere una aplicación cliente para obtener información cuando el analizador SAX notifica a la aplicación cliente que la información está disponible.
- **StAX API puede leer y escribir documentos XML.** Usando SAX API, un archivo XML solo se puede leer.

StAX consta realmente de 2 distintas API:

- `API Cursor`: **Representa un cursor** con el cual se puede ir hacia adelante en un documento XML desde el principio hasta el final. Este cursor puede **apuntar un elemento a la vez** y siempre se mueve hacia adelante, nunca hacia atrás.
- `API Iterator` : **Representa un flujo** de un documento XML como un conjunto **de objetos de eventos discretos.** La aplicación saca estos eventos en el mismo orden que los proporciona el parser al leerlos del documento XML. Por cada paso de iteración se obtiene un objeto `XMLEvent`, que contiene información sobre el evento generado en el proceso de lectura de la información. Mediante la referencia a este objeto, podemos extraer la información de los eventos generados, con los métodos que proporciona esta clase.

Una de las principales diferencias entre ambos estilos es que cuando se usa la `Iterator` API es posible ir hacia atras en la jerarquia , cosa que no puede realizarse cuando utilizamos la Cursor API, ya que una vez mueves el cursor hacia el próximo evento, no se tiene información acerca del evento previo. Sin embargo la Cursor API es mas eficiente en cuanto a memoria, así que la selección de que estilo de API utilizar dependerá de las necesidades.

6.6.1.- Características de StAX.

Las siguientes son las características de StAX API:

- Lee un documento XML de arriba a abajo, reconociendo los tokens que componen un documento XML bien formado.
- Los tokens se procesan en el mismo orden en que aparecen en el documento.
- Informa sobre el programa de aplicación la naturaleza de los tokens que el analizador ha encontrado a medida que ocurren.
- El programa de aplicación proporciona un lector de "eventos" que actúa como un iterador e itera sobre el evento para obtener la información requerida. Otro lector disponible es "cursor" que actúa como un puntero a los nodos XML.
- A medida que se identifican los eventos, los elementos XML pueden recuperarse del objeto de evento y pueden procesarse más.

Debemos usar un analizador StAX cuando:

- Puede procesar el documento XML de forma lineal de arriba a abajo.
- El documento no está profundamente anidado.
- Está procesando un documento XML muy grande cuyo árbol DOM consumiría demasiada memoria. Las implementaciones DOM típicas usan diez bytes de memoria para representar un byte de XML.
- El problema a resolver involucra solo una parte del documento XML.
- Los datos están disponibles tan pronto como el analizador los vea, por lo que StAX funciona bien para un documento XML que llega a través de una transmisión.

Para poder utilizar clases e interfaces de ambas API necesitamos un origen de datos XML valido. Esto lo conseguimos con la factoria **`XMLInputFactory:`**

```
XMLInputFactory xmlif = XMLInputFactory.newInstance();
XMLStreamReader xmlsr =xmlif.createXMLStreamReader(new FileReader("books.xml"));
```

o bien

```
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
InputStream in = new FileInputStream("books.xml");
```

Como deciamos anteriormente, STAX consiste en dos API llamados lectores "Iterador" y "cursor"

¿Cómo funciona?

El analizador crea diferentes tipos de eventos a medida que avanza leyendo el documento XML de origen.

Algunos de los tipos de eventos importantes son:

1. Comience el documento
2. Elemento de inicio
3. Comentarios
4. Caracteres
5. Elemento final
6. Documento final

partimos del siguiente documento XML:

```
<? xml version = "1.0"?>
<BookCatalogue xmlns = "http://www.publishing.org">
    <Libro>
        <Titulo> Yogasana Vijnana: la ciencia del yoga </ Title>
        <ISBN> 81-40-34319-4 </ ISBN>
        <Cost currency = "INR"> 11.50 </ Cost>
    </ Book>
</ BookCatalogue>
```

Este documento genera 18 eventos entre primarios y secundarios:

#	Element/Attribute	Event
1	`version="1.0"`	`StartDocument`
2	`isCData = false data = "\n" IsWhiteSpace = true`	`Characters`
3	`qname = BookCatalogue:http://www.publishing.org attributes = null namespaces = {BookCatalogue" -> http://www.publishing.org"}`	`StartElement`
4	`qname = Book attributes = null namespaces = null`	`StartElement`
5	`qname = Title attributes = null namespaces = null`	`StartElement`
6	`isCData = false data = "Yogasana Vijnana: the Science of Yoga\n\t" IsWhiteSpace = false`	`Characters`
7	`qname = Title namespaces = null`	`EndElement`

8	qname = ISBN attributes = null namespaces = null	StartElement
9	isCData = false data = "81-40-34319-4\n\t" IsWhiteSpace = false	Characters
10	qname = ISBN namespaces = null	EndElement
11	qname = Cost attributes = {"currency" -> INR} namespaces = null	StartElement
12	isCData = false data = "11.50\n\t" IsWhiteSpace = false	Characters
13	qname = Cost namespaces = null	EndElement
14	isCData = false data = "\n" IsWhiteSpace = true	Characters
15	qname = Book namespaces = null	EndElement
16	isCData = false data = "\n" IsWhiteSpace = true	Characters
17	qname = BookCatalogue:http://www.publishing.org namespaces = {BookCatalogue" -> http://www.publishing.org"}	EndElement
18		EndDocument

Aspectos importantes:

- Los eventos se crean en el orden en que se encuentran los elementos XML correspondientes en el documento, incluida la anidación de elementos, la apertura y el cierre de elementos, el orden de los atributos, el inicio del documento y el final del documento, y así sucesivamente.
- Al igual que con la sintaxis XML adecuada, todos los elementos del contenedor tienen los correspondientes eventos de inicio y final; por ejemplo, cada `StartElement` tiene un `EndElement` correspondiente , incluso para elementos vacíos.
- `Los` eventos de `atributo` se tratan como eventos secundarios, y se accede a ellos desde su evento `StartElement` correspondiente .
- De forma similar a los eventos de `Atributo` , los eventos de `Espacio de nombres` se tratan como secundarios, pero aparecen dos veces y son accesibles dos veces en la secuencia de eventos, primero desde su `StartElement` correspondiente y luego desde su `EndElement` correspondiente .
- `Los` eventos de `caracteres` se especifican para todos los elementos, incluso si esos elementos no tienen datos de caracteres. Del mismo modo, los eventos de carácter se pueden dividir entre eventos.
- El analizador StAX mantiene una pila de espacio de nombres, que contiene información sobre todos los espacios de nombres XML definidos para el elemento actual y sus antecesores. Se puede acceder a la pila del espacio de nombres, que se expone a través de la interfaz `javax.xml.namespace.NamespaceContext` , mediante el prefijo del espacio de nombres o el URI.

Pregunta Verdadero-Falso

Los eventos de caracteres se especifican para todos los elementos XML

○ Verdadero ○ Falso

Verdadero

6.6.2.- Ejemplo de API Cursor.

El siguiente programa es un ejemplo muy sencillo de como utilizar un cursor para recorrer un documento **`XMLPartiendo`** del documento XML **`books.xml`**, listamos los títulos de los libros.

`books.xml`

```
<bookstore>
<book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
</book>
<book category="children">
     <title lang="en">Harry Potter</title>
     <author>J K. Rowling</author>
     <year>2005</year>
     <price>29.99</price>
</book>
<book category="web">
     <title lang="en">XQuery Kick Start</title>
     <author>James McGovern</author>
     <author>Per Bothner</author>
     <author>Kurt Cagle</author>
     <author>James Linn</author>
     <author>Vaidyanathan Nagarajan</author>
     <year>2003</year>
     <price>49.99</price>
</book>
<book category="web" cover="paperback">
     <title lang="en">Learning XML</title>
     <author>Erik T. Ray</author>
     <year>2003</year>
     <price>39.95</price>
</book>
</bookstore>
```

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.ArrayList;
import javax.xml.stream.FactoryConfigurationError;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;
//El programa java que recorre el documento para extraer los titulos y el atributo lang es el siguiente
 public class ListaLibros {
public static void main(String[] args) throws FileNotFoundException, XMLStreamException {
// Creamos el flujo
XMLInputFactory xmlif = XMLInputFactory.newInstance();
XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader("books.xml"));
String tag = null;
int eventType;
System.out.println("Lista de libros");
// iteramos con el cursor a lo largo del documento
   while (xmlsr.hasNext()) {
        eventType = xmlsr.next();
        switch (eventType) {
        case XMLEvent.START_ELEMENT:
             tag = xmlsr.getLocalName();
             if (tag.equals("title")) {
                 System.out.println(xmlsr.getElementText() + " idioma  " + xmlsr.getAttributeValue(0));
              }
         break;
         case XMLEvent.END_DOCUMENT:
          System.out.println("Fin del documento");
          break;
              }
        }
    }
}
```

6.6.3.- Ejemplo de API Event.

Utilizamos el documento books.XML del ejemplo de la API cursor, el listado es similar al ejemplo de API de cursor

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.util.Iterator;
import javax.xml.namespace.QName;
import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.events.Attribute;
import javax.xml.stream.events.StartElement;
import javax.xml.stream.events.XMLEvent;
public class EventReader {
    public static void main(String[] args) {
        // primero crea un nuevo XMLInputFactory
        XMLInputFactory inputFactory = XMLInputFactory.newInstance();
        // Configura un nuevo eventReader a partir del fichero XML
        InputStream in = null;
        try {
          in = new FileInputStream("books.xml");
        } catch (FileNotFoundException e1) {
          e1.printStackTrace();
          }
          try {

            XMLEventReader eventReader = InputFactory.createXMLEventReader(in);
            // repetitiva que recorre todos los eventos
            while (eventReader.hasNext()) {
              XMLEvent event = eventReader.nextEvent();
              // si el evento es el inicio del nodo titulo
              // avanzo un evento para obtener el titulo del libro
             if (event.getEventType() == MLStreamConstants.START_ELEMENT) {
                  StartElement startElement = event.asStartElement();
             if (startElement.getName().getLocalPart() == "title") {
                  Iterator iterator = ((StartElement) event).getAttributes();
                  while (iterator.hasNext())
                    { Attribute attribute = (Attribute) iterator.next();
                    QName name = attribute.getName();
                    String value = attribute.getValue();
                    System.out.println("Atributo name/valor: " ++ "/" + value); }
                    event = eventReader.nextEvent();
                    System.out.println((String) event.asCharacters().getData()); }
            } else if (event.getEventType() == XMLStreamConstants.END_DOCUMENT)
               { System.out.println("fin del documento");
            }
          }
          } catch (XMLStreamException e) {
       e.printStackTrace();
    }
  }
}
```

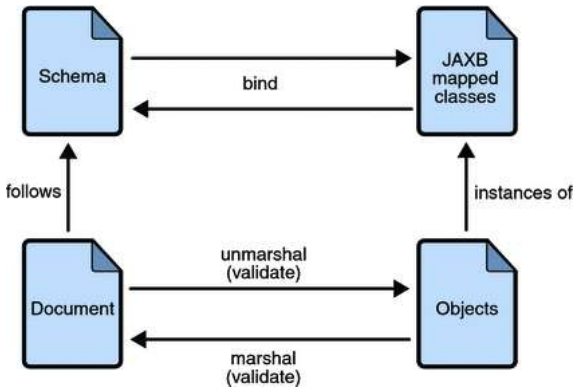
6.7.- Binding.

El *Binding* es una técnica que consiste en vincular clases Java con formatos específicos de almacenamiento de manera automatizada.

En Java existen varias bibliotecas para gestionar el *binding* , como por ejemplo *JAXB* , *JiBX* , *XMLBinding* , etc. Desde la versión 6.0 se ha incorporado en el JDK estándar *JAXB,* una potente biblioteca.

Java Architecture for XML Binding (JAXB) permite a los desarrolladores Java asignar clases de Java a representaciones XML. JAXB proporciona dos características principales: la capacidad de serializar las referencias de objetos Java a XML y la inversa, es decir, deserializar XML en objetos Java. En otras palabras, JAXB permite almacenar y recuperar datos en memoria en cualquier formato XML, sin la necesidad de implementar un conjunto específico de rutinas de carga y guardado de XML para la estructura de clases del programa.

La siguiente figura muestra lo que ocurre durante el proceso de enlace de JAXB.



6.7.1.- Configuración con anotaciones.

Las **Anotaciones pueden asociarse a un paquete, a una clase, a un atributo o incluso a un parámetro.** Estas clases especiales **se declaran** en el código de la aplicación **anteponiendo el símbolo @ en el nombre de la Anotación** . Cuando el compilador de Java detecta una Anotación crea una instancia y la inyecta en el elemento estructural afectado (paquete, clase, método, atributo, etc.). Esto hace que éstas no aparezcan como atributos o métodos propios del objeto, y por eso decimos que no interacciona con el modelo de datos, pero las aplicaciones que lo necesiten pueden obtener la instancia inyectada y usarla.

Las Anotaciones **pueden declararse con parámetros o sin ellos.** En caso de que tengan parámetros, estos pueden ser otros Anotaciones, valores constantes o valores literales, de modo que estén disponibles en tiempo de compilación (que es cuando el compilador realiza la inyección).

Anotaciones principales.

`@XmlRootElement(namespace = "namespace", name = "nombre" )`: Define la raíz del XML.
`@XmlType(propOrder = { "field2", "field1",.. }, name = "nombre")`: Permite definir en que orden se van escribir los elementos dentro del XML. Esta anotación deberán tenerla todas las clases que no mapean la raiz del XML.
`@XmlElement(name = "nombre")`: Define el elemento de XML que se va usar. Si el atributo de la clase tiene el mismo nombre que la etiqueta XML, podemos omitirlo. Tenemos que escribir esta anotación antes del método *setter* correspondiente. Mirar ejemplos en el siguiente subcapitulo.
`@XmlAttribute` se utiliza para mapear atributos a los nodos XML.
`@XmlElementWrapper`: ofrece la posibilidad de crear un contenedor alrededor de una representación XML. Este contenedor puede contener una colección de elementos.

El parámetro `name` = "nombre" permite especificar el nombre de la etiqueta XML.

Pregunta Verdadero-Falso

Las anotaciones pueden declararse solo con parámetros.

○ Verdadero ○ Falso

Falso

Se pueden declarar tanto con o sin parámetros.

6.7.2.- Ejemplos con anotaciones.

Tenemos la estructura XML siguiente:

```
<libreria>
    <ListaLibro>
        <Libro>
            <autor>XXXXXXX</autor>
            <nombre>XXXXXXX</nombre>
            <editorial>XXXXXXX</editorial>
            <isbn>XXXXXXX</isbn>
        </Libro>
         ......
</ListaLibro>
....
<lugar>XXXXXXX</lugar>
<nombre>XXXXXX</nombre>
</libreria>
```

Para mapear la raiz es necesario una clase librería con anotación @XmlRootElement.

```
  @XmlRootElement
            public class Libreria {
                  .....
            }
```

Esta clase tiene un atributo por cada etiqueta del XML incluido ListaLibro que es un array de elementos de tipo Libro:

```
            @XmlElementWrapper(name = "ListaLibro")
            @XmlElement(name = "Libro")
            private ArrayList<Libro> ListaLibro;
```

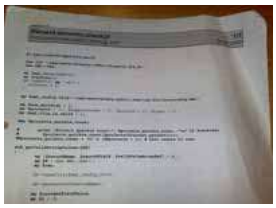
La clase librería, es la clase que mapea el XML

```
import java.util.ArrayList;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;
//Esto significa que la clases "Libreria.java" es el elemento raiz
@XmlRootElement
public class Libreria {
    //Wrapper
    @XmlElementWrapper(name = "ListaLibro")
    @XmlElement(name = "Libro")
    private ArrayList<Libro> ListaLibro;
    private String nombre;
    private String lugar;
    public ArrayList<Libro> getListaLibro() {
        return ListaLibro;
    }
    public void setListaLibro(ArrayList<Libro> ListaLibro) {
        this.ListaLibro = ListaLibro;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public void setLugar(String lugar) {
        this.lugar = lugar;
    }
    public String getNombre() {
        return nombre;
    }
    public String getLugar() {
        return lugar;
    }
}
```

La clase libro que mapea cada uno de los nodos <Libro> del XML

```java
import javax.xml.bind.annotation.XmlType;
@XmlType(propOrder = { "autor", "nombre", "editorial", "isbn" })
public class Libro {
private String nombre;
private String autor;
private String editorial;
private String isbn;
public String getNombre() {
return nombre;
}
public String getAutor() {
return autor;
}
public String getEditorial() {
return editorial;
}
public String getIsbn() {
return isbn;
}
public void setNombre(String nombre) {
this.nombre = nombre;
}
public void setAutor(String autor) {
this.autor = autor;
}
public void setEditorial(String editorial) {
this.editorial = editorial;
}
public void setIsbn(String isbn) {
this.isbn = isbn;
}
}
```

6.7.3.- Funcionamiento de JAXB.



Para construir una aplicación JAXB necesitamos tener un esquema XML.

ras obtener el esquema XML, seguimos los **siguientes pasos para construir la aplicación JAXB**:

1. **Escribir el esquema**: es un documento XML que contiene la estructura que se tomará como indicaciones para construir las clases. Estas indicaciones pueden ser, por ejemplo, el tipo primitivo al que se debe unir un valor de atributo en la clase generada.
2. **Generar los ficheros fuente de Java**: para esto usamos el compilador de esquema, ya que éste toma el esquema como entrada de información. Cuando se haya compilado el código fuente, podremos escribir una aplicación basada en las clases que resulten.
3. **Construir el árbol de objetos Java**: con nuestra aplicación, se genera el árbol de objetos java, también llamado árbol de contenido, que representa los datos XML que son validados con el esquema. Hay dos formas de hacer esto:
    1. Instanciando las clases generadas.
    2. Invocando al método `unmarshall` de una clase generada y pasarlo en el documento. El método `unmarshall` toma un documento XML válido y construye una representación de árbol de objetos.
4. **Acceder al árbol de contenido** usando nuestra aplicación: ahora podemos acceder al árbol de contenido y modificar sus datos.
5. **Generar un documento XML** desde el árbol de contenido. Para poder hacerlo tenemos que invocar al método `marshall` sobre el objeto raíz del árbol.

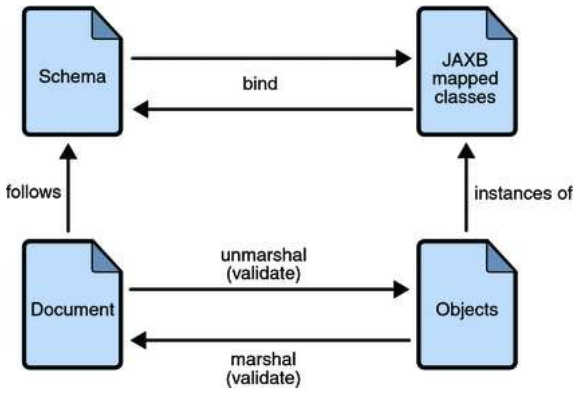
JAXB proporciona dos principales características:

- La capacidad de serializar (marshalling) objetos Java a XML.
- Lo inverso, es decir, deserializar (unmarshalling) XML a objetos Java.

O sea que **JAXB permite almacenar y recuperar datos en memoria en cualquier formato XML, sin la necesidad de implementar un conjunto específico de rutinas XML** de carga y salvaguarda para la estructura de clases del programa.

El compilador de JAXB (schema compiler) permite generar una serie de clases Java que podrán ser llamadas desde nuestras aplicaciones a través de métodos sets y gets para obtener o establecer los datos de un documento XML.

Los pasos en el proceso de enlace de JAXB están representados en la figura:



1. Generar clases: podemos escribirlas manualmente, como en el capitulo anterior o utilizar las clases generadas por el IDE JAXB automáticamente utilizando el esquema .xsd correspondiente al XML, lo veremos más adelante.
2. Deben compilarse todas las clases generadas, los archivos fuente y el código de la aplicación .
3. Aplicar Marshall a los objetos de las clases mapeadas para crear XML
4. Aplicar UnMarshall al XML para obtener objetos de las clases mapeadas .

Marshall

Para poder poder hacer esta traducción, lo primero que necesitamos es un contexto, el cual instanciaremos pasándole la clase de nuestro root:

```
 JAXBContext jaxbContext = JAXBContext.newInstance(Libreria.class);
```

Ya tenemos un contexto y ahora toca generar nuestro Marshaller

```
Marshaller marshaller = jaxbContext.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

Para obtener un resultado, debemos pasarle una salida a nuestro marshaller. Aquí tienes un par de ejemplo de como hacerlo.

```
marshaller.marshal(libreria, System.out);
```

o bien

```
File libreria-jaxb = this.getFile();
if (libreria-jaxbl != null) {
       marshaller.marshal(libreria, libreria-jaxb);
   }
```

UnMarshall
Otra parte también fácil, tenemos varios inputs y utilizaremos el fichero por ahora para poder parsear. Partiendo de que tenemos ya un contexto, el cual reutilizaremos en esta parte.

```
Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();

Libreria libreria = (Libreria) unmarshaller.unmarshal( libreria-jaxb);
```

Así de simple, recuperaremos o pasaremos de nuestro fichero xml a un objeto el cual tenemos ya **mapeado**

Para saber más

En el siguiente enlace puedes ver un ejemplo paso a paso en el que:

- Se crea el fichero de esquema XML.
- Se crea un proyecto nuevo con NetBeans.
- Se añade un JAXB Binding utilizando el fichero XSD creado anteriormente.
- Se añade un servicio web y se utiliza las clases Java JAXB Binding como un tipo de objeto.

6.7.4.- Ejemplo: Marshall, Unmarshall.

Una vez que tenemos el proyecto con las clases del apartado Ejemplo anotaciones, vamos a añadir el siguiente programa

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
public class JavaJAXB {
    private static final String LIBRERIA_XML = "./libreria-jaxb.xml";
    public static void main(String[] args) throws JAXBException, IOException {
        // Lista de LIbros
        ArrayList<Libro> libroLista = new ArrayList<Libro>();
        // Creamos varios libros
        Libro libro1 = new Libro();
        libro1.setIsbn("978-0060554736");
        libro1.setNombre("The Game");
        libro1.setAutor("Neil Strauss");
        libro1.setEditorial("Harpercollins");
        libroLista.add(libro1);
        Libro libro2 = new Libro();
        libro2.setIsbn("978-3832180577");
        libro2.setNombre("Feuchtgebiete");
        libro2.setAutor("Charlotte Roche");
        libro2.setEditorial("Dumont Buchverlag");
        libroLista.add(libro2);
        // Se crea La libreria y se le asigna la lista de libros
        Libreria libreria = new Libreria();
        libreria.setNombre("LIbreria sin limite");
        libreria.setLugar("Barrio Obrero");
        libreria.setListaLibro(libroLista);
        // Creamos un contexto de la clase JAXB y lo intanciamos
       JAXBContext context = JAXBContext.newInstance(Libreria.class);
       Marshaller m = context.createMarshaller();
       m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
       // Lo creamos con system out
       m.marshal(libreria, System.out);
       // Escribimos en el archivo
       m.marshal(libreria, new File(LIBRERIA_XML));
       // Obtenemos las variables obtenidas del XML creado anteriormente
       System.out.println();
       System.out.println("Salida del XML: ");
       Unmarshaller um = context.createUnmarshaller();
       Libreria libreria2 = (Libreria) um.unmarshal(new FileReader(LIBRERIA_XML));
       ArrayList<Libro> lista = libreria2.getListaLibro();
       for (Libro libro : lista) {
       System.out.println("Libro: " + libro.getNombre() + " de " + libro.getAutor());
   }
}
}
```

Puedes encontrar mas ejemplos en la página:

https://www.javacodegeeks.com/2014/12/jaxb-tutorial-xml-binding.html#marshal

6.7.5.- Generación automática de clases Java a partir del esquema .xsd.

JAXB es una parte de la plataforma Java SE y una de las APIs de la plataforma Java EE, y es parte del Java Web Services Development Pack (JWSDP). También es uno de los fundamentos para WSIT. JAXB es parte de la versión 1.6 SE.

La herramienta **"xjc"** se puede utilizar para convertir un XML Schema y otros tipos de archivo de esquemas (en Java 1.6, RELAX NG, XML DTD y WSDL son compatibles experimentalmente) a representaciones de clase.
Las clases son marcadas usando anotaciones del espacio de nombres `javax.xml.bind.annotation.*`, por ejemplo, `@XmlRootElement` y `@XmlElement`. Las secuencias de listas XML se representan con atributos de tipo `java.util.List`.
Los serializadores y deserializadores se crean a través de una instancia de `JAXBContext`.

Además, JAXB incluye la herramienta "`schemagen`" que en esencia puede llevar a cabo la inversa de "xjc", creando un XML `Schema` a partir de un conjunto de clases anotadas.

La siguiente tabla muestra las asignaciones de tipos de datos XML Schema (XSD) a tipos de datos Java en JAXB:

Tipo de XML Schema	Tipo de dato Java
`xsd:string`	`java.lang.String`
`xsd:positiveInteger`	`java.math.BigInteger`
`xsd:int`	`int`
`xsd:long`	`long`
`xsd:short`	`short`
`xsd:decimal`	`java.math.BigDecimal`
`xsd:float`	`float`
`xsd:double`	`double`
`xsd:boolean`	`boolean`
`xsd:byte`	`byte`
`xsd:QName`	`javax.xml.namespace.QName`
`xsd:dateTime`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:base64Binary`	`byte[]`
`xsd:hexBinary`	`byte[]`
`xsd:unsignedInt`	`long`
`xsd:unsignedShort`	`int`
`xsd:unsignedByte`	`short`
`xsd:unsignedLong`	`java.math.BigDecimal`
`xsd:time`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:date`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:g`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:anySimpleType (for xsd:element of this type)`	`java.lang.Object`
`xsd:anySimpleType (for xsd:attribute of this type)`	`java.lang.String`
`xsd:duration`	`javax.xml.datatype.Duration`
`xsd:NOTATION`	`javax.xml.namespace.QName`

Los pasos para obtener las clases IDE Eclipse

1. Descargar la distribución:
   Download standalone distribution, descomprimimos el fichero.
2. Añadir todas las clases que estan en la carpeta `\jaxb-ri-2.3.0\jaxb-ri\lib, jaxb-api.jar, jaxb-core.jar, jaxb-impl.jar, jaxb-jxc.jar, jaxb-xjc.jar`
3. Obtener el esquema .xsd a partir del XML.
4. Crear un proyecto en Eclipse, en la carpeta scr del proyecto situar el esquema y en la carpeta raíz del proyecto situamos el fichero XML .Para generar las clases de forma automática clicamos el botón derecho del ratón sobre el esquema, generate -> JAXB Classes.

7.- Librerías para conversión de documentos XML a otros formatos.

Caso práctico

María le dice a **Juan** que una buena práctica para **Antonio** y **Ana** sería que hicieran los informes que necesitan para la aplicación de la farmacia.

-Tan solo tenemos que instruirles un poco en JasperReport e indicarles unos cuantos tutoriales, seguro que lo hacen bien -le comenta **María** a **Juan**.

María decide, además, consultar a su amiga **Blanca**, que es experta en JasperReport y trabaja en otra empresa, qué tutoriales de Internet o de cualquier otra fuente recomendaría a unos principiantes en esta materia.

En la mayoría de aplicaciones informáticas, hay que mostrar la información resultante de los procesos que se ejecutan, sobre todo en aplicaciones que generan información que implica tomar decisiones comerciales. Dicha información está almacenada normalmente en bases de datos o en archivos.

Hoy en día, XML está muy extendido, y muchas empresas guardan la información en ficheros o bases de datos con ese formato.

Hay muchos productos o herramientas informáticas que permiten convertir documentos XML a otros formatos.

En nuestro caso, vamos a optar por una herramienta que permite generar informes de todo tipo en Java de una forma sencilla: JasperReport.

Esta herramienta permite generar informes electrónicos en formato pdf, quizás el formato más usado debido a su portabilidad entre sistemas conservando la apariencia. Pero existen muchos más: `xls`, `html`, `rtf`, `csv`, `xml`, etc.

Para saber más

En el siguiente enlace puedes ver un tutorial de otra herramienta de conversión de XML a otros formatos.

XSL

Autoevaluación

JasperReport permite generar documentos en formato html.

○ Verdadero ○ Falso

Verdadero
Así es, y en otros formatos adicionales.

7.1.- Introducción a JasperReport.

En Java, durante un tiempo, la generación de informes fue uno de los puntos débiles del lenguaje, pero hoy en día, existen muchas librerías y herramientas dedicadas (varias de ellas, de código abierto) para la rápida generación de informes. JasperReports, es una de las más conocidas.

JasperReports es una herramienta que consta de un poderoso motor para la generación de informes. Está empaquetada en un archivo JAR y puede ser utilizada como una librería, la cuál podemos integrar en cualquier IDE de desarrollo en Java para desarrollar nuestras aplicaciones. Está escrita totalmente en Java, su código es abierto y es totalmente gratuita bajo los términos de la licencia GPL (Licencia Pública General).

Si visitas el siguiente enlace podrás acceder a la página de descarga de JasperReports para todas las plataformas. Encontrarás una lista con todas las versiones disponibles, no es necesario identificarte para poder bajarte la que desees.

Zona de descarga de JasperReports

Debes conocer

Documento sobre como descarga de JasperReports e integración en NetBeans.

Resumen textual alternativo

En la presentación que acabas de ver, al descomprimir el fichero de la descarga, has visto que en el mismo hay varios directorios o carpetas. Comentamos brevemente qué contiene cada una:

- **build**: es la librería JasperReports sin empaquetar, con todas las clases que incluye.
- **demo**: podemos encontrar algunos ejemplos de utilización de la librería. Estos ejemplos están preparados para ser compilados con la herramienta "**ant**". Puedes inspeccionar el código Java e intentar compilarlos y ejecutarlos.
- **dist**: es donde se encuentra realmente la librería empaquetada en un fichero JAR (`jasperreports-3.7.4.jar`) y algunos ficheros JAR que no utilizaremos. También podemos acceder a la documentación tipo javadoc.
- **docs**: es la referencia rápida en formato XML.
- **lib**: Diferentes librerías necesarias por JasperReports, como algunas para exportar a distintos formatos, para incluir gráficos, etc.
- **src**: Ficheros fuente de la librería.

Para saber más

En este enlace tienes la documentación en línea de la API de JasperReports.

API de JasperReports

7.2.- Diseñar y compilar la plantilla.

Las plantillas de los informes de JasperReports son sencillamente ficheros XML con la extensión .jrxml. Podemos hacer que NetBeans reconozca este tipo de ficheros como XML, para que cuando los editemos en el editor se muestren los mismos códigos de colores en las etiquetas y demás elementos de la sintaxis de XML.

En la imagen se ilustra cómo conseguirlo: en NetBeans pinchamos en el menú Tools, y ahí en Options. Ahí seleccionamos Miscellaneous, luego la pestaña Files. Entonces pulsamos en el botón New... para añadir la nueva extensión.

Los pasos a seguir para trabajar con JasperReport serían:

Paso 1: Diseñar la plantilla del informe: un fichero .jrxml. El documento de diseño está representado por un archivo XML que mantiene la estructura de un archivo DTD (Document Type Definition) definido por el motor de JasperReports.

La generación de un diseño implica editar un archivo XML validado mediante:

```
<!DOCTYPE jasperReport PUBLIC "-//JasperReports//DTD Report Design//EN" "http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">
```

Estos documentos XML cuentan con una estructura similar a la de cualquier documento de texto. Fundamentalmente se siguen estas secciones:

- `title` Título del informe.
- `pageHeader` Encabezado del documento.
- `columnHeader` Encabezado de las columnas.
- `detail` Detalle del documento. Cuerpo
- `columnFooter` Pie de la columna.
- `pageFooter` Pie del documento.
- `sumary` Cierre del documento.

Paso 2: Compilación: Una vez que se ha realizado el diseño, se compila antes de poder iniciar el proceso de carga de datos. La compilación se lleva a cabo a través del método `compileReport()`.

En este proceso, el diseño se transforma en un objeto serializable de tipo `net.sf.jasperreports.engine` JasperReport, que luego se guarda en disco.

Autoevaluación

¿Es correcta la afirmación siguiente?

Los documentos XML que se usan en el diseño de los informes cuentan con una serie de secciones.

○ Verdadero ○ Falso

Verdadero
Es correcto, las citadas en el anterior paso 1.

7.3.- Rellenar el informe con datos, exportar el informe.

Paso 3: Rellenar el informe con datos: mediante los métodos fillReportXXX(), se puede realizar la carga de datos del informe, pasándole como parámetros el objeto de diseño (o bien, el archivo que lo representa en formato serializado) y la conexión JDBC a la base de datos desde donde se obtendrá la información que necesitemos.

Como resultado de este proceso, se obtiene un objeto que representa un documento listo para ser impreso, un objeto serializable de tipo `JasperPrint`. Este objeto puede guardarse en disco para su uso posterior, o bien puede ser impreso, enviado a la pantalla o transformado en PDF, XLS, CSV, etc.

Paso 4: Visualización

Ahora podemos optar por mostrar un informe por pantalla, imprimirlo, o bien obtenerlo en algún tipo específico de fichero, como PDF, etc.

- Para mostrar un informe por pantalla se utiliza la clase `JasperViewer`, la cual, a través de su método `main()`, recibe el informe a mostrar.
- Para imprimir el informe usaremos los métodos `printReport()`, `printPage()` o `printPages()`, contenidos en la clase `JasperPrintManager`.
- Para exportar los datos a un formato de archivo específico podemos utilizar los métodos exportReportXXX().

En el siguiente enlace se muestra un video de como intalar JasperReports con Netbeans.

Para saber más

El principal inconveniente que puedes encontrarte al trabajar con JasperReports sin más, es sin duda el diseño del informe. Por ello, para facilitar el diseño de los mismos, y hacerlos de manera visual y cómoda se pueden usar otros productos como iReport, que es también una herramienta de software libre.

Hay mucha documentación sobre iReport en la red, aquí te adjuntamos dos:

Tutorial iReport

Crear informes con iReport

Anexo I.- Listar ficheros de una carpeta, filtrando.

```
import java.io.File;
import java.io.FilenameFilter;
public class Filtrar implements FilenameFilter {
String extension;
// Constructor
Filtrar(String extension){
this.extension = extension;
}
public boolean accept(File dir, String name){
return name.endsWith(extension);
}
public static void main(String[] args) {
try {
// Obtendremos el listado de los archivos de ese directorio
File fichero=new File("c:\\datos\\.");
String[] listadeArchivos = fichero.list();
// Filtraremos por los de extension .txt
listadeArchivos = fichero.list(new Filtrar(".txt"));
// Comprobamos el número de archivos en el listado
int numarchivos = listadeArchivos.length ;
// Si no hay ninguno lo avisamos por consola
if (numarchivos < 1)
System.out.println("No hay archivos que listar");
// Y si hay, escribimos su nombre por consola.
else
{
for(int conta = 0; conta < listadeArchivos.length;
conta++)
System.out.println(listadeArchivos[conta]);
}
}
catch (Exception ex) {
System.out.println("Error al buscar en la ruta indicada");
}
}
}
```

Anexo II.- Código de separador de rutas.

```
String substFileSeparator(String ruta){
 String separador = "\\";
        try{
     // Si estamos en Windows
           if ( File.separator.equals(separador) )
                separador = "/" ;
        // Reemplaza todas las cadenas que coinciden con la expresión
        // regular dada oldSep por la cadena File.separator
           return ruta.replaceAll(separador, File.separator);
        }catch(Exception e){
    // Por si ocurre una java.util.regex.PatternSyntaxException
           return ruta.replaceAll(separador + separador, File.separator);
        }
       }
```

Anexo III.- Código de crear un fichero.

```
try { // Creamos el objeto que encapsula el fichero
File fichero = new File("c:\\prufba\\miFichero.txt");
// A partir del objeto File creamos el fichero físicamente
if (fichero.createNewFile()) System.out.println("El fichero se ha creado correctamente");
         else System.out.println("No ha podido ser creado el fichero");
} catch (Exception ioe) { ioe.getMessage(); }
```

Anexo IV.- Código de crear un directorio.

```
try {
// Declaración de variables
       String directorio = "C:\\prueba";
       String varios = "carpeta1/carpeta2/carpeta3";
// Crear un directorio
       boolean exito = (new File(directorio)).mkdir();
       if (exito)
          System.out.println("Directorio: " + directorio + " creado");
// Crear varios directorios
      exito = (new File(varios)).mkdirs();
      if (exito)
          System.out.println("Directorios: " + varios + " creados");
}catch (Exception e){
      System.err.println("Error: " + e.getMessage());
}
```

Anexo V.- Anexo de recursos

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	Autoría: daddo83. Licencia: CC-by-nc. Procedencia: http://www.flickr.com/photos/daddo83/3406962115/		Autoría: O'Reilly & Associates. Licencia: Copyright (cita). Procedencia: http://aps2.elekta.lt/Books/oreilly/java/fclass/figs/jfc_1101.gif
	Autoría: Art3mis4. Licencia: CC-by-nc-sa. Procedencia: http://www.flickr.com/photos/art3mis4/4910243349/		Autoría: José Javier Bermúdez Hernández. Licencia: Copyright (Cita). Procedencia: Captura de pantalla del programa Explorer, propie de Microsoft.
	Autoría: Kasaa. Licencia: CC-by-nc. Procedencia: http://www.flickr.com/photos/kasaa/2693784352/		Autoría: Pedro Sousa. Licencia: CC-by-nc-sa. Procedencia: http://www.flickr.com/photos/pedrosousa/2355996
	Autoría: д§мд. Licencia: CC-by-nc. Procedencia: http://www.flickr.com/photos/asmamirza/2599581983/		Autoría: ConvenienceStoreGourmet. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/conveniencestoregourmet/4873237
	Autoría: Ornellaswouldgo. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/ornellas/4257487503/		Autoría: Identity chris is. Licencia: CC-by-nc-sa. Procedencia: http://www.flickr.com/photos/identity-c is/71240905/
	Autoría: aldoaldoz. Licencia: CC-by-nc-sa. Procedencia: http://www.flickr.com/photos/aldoaldoz/3895614433/#/		Autoría: Danard Vicente. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/danardvincente/2512148775/
	Autoría: Fartese. Licencia: CC-by-sa. Procedencia: http://www.flickr.com/photos/fartese/4214174953/		Autoría: RalphTQ. Licencia: CC-by-nc. Procedencia: http://www.flickr.com/photos/ralphtq/3157588757/
	Autoría: Roland. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/roland/4358742850/		Autoría: Freddy the Boy. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/freddy-c boy/3483094014/
	Autoría: José Javier Bermúdez Hernández. Licencia: GNU GPL v2. Procedencia: Montaje sobre Captura de pantalla del programa NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.		Autoría: Mike Baird. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/mikebaird/352740629

Dos carpetas una mas pequeña sobre otra más grande