MEDAC
Instituto Oficial de Formación Profesional

PROGRAMACIÓN

Aplicación de estructuras de almacenamiento

5

ÍNDICE

OBJETIVOS

Diferenciar entre diferentes tipos de colecciones.

Diferenciar cuándo utilizar los diferentes tipos de colecciones.

Utilizar iteradores.

Procesar documentos XML.

Conocer la estructura de un documento XML.

Utilizar diferentes tipos de colecciones.

/ 1. Introducción y contextualización práctica

Uno de los principales problemas que tienen los arrays es que una vez creados no pueden modificar su tamaño, por lo tanto, si creamos un array de un tamaño dado y posteriormente se nos queda pequeño, necesitaremos crear otro nuevo de mayor capacidad y copiar los elementos en él. Otra opción es crear un array de un tamaño que estemos seguros que va a ser suficiente, sin embargo, sería un gran desperdicio de memoria. Lo ideal sería tener una estructura que adapte su tamaño conforme se vayan añadiendo y eliminando elementos. Esto es lo que hacen las colecciones.

En Java existen multitud de colecciones y no sería viable estudiarlas todas en este tema, no obstante, vamos a aprender y profundizar sobre las más utilizadas, y cómo recorrerlas de forma segura.

Por último, vamos a ver qué es un fichero XML y nos traeremos el concepto al entorno de Java. Veremos cómo podemos validarlo y cómo podemos crearlos desde Java. Conceptos que puede que hayas aprendido, o lo vayas a hacer, en otras asignaturas, pero en este tema, los estudiaremos desde un punto de vista de aplicación en Java.

Fig. 1. Una colección.

Escucha el siguiente audio en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.

Audio Intro. "Un formato desconocido".
https://bit.ly/32YwSBx

/ 2. Colecciones

Cuando hablamos de colecciones en programación, nos referimos a un conjunto de elementos que están almacenados de forma conjunta en una misma estructura de datos.

Estos elementos van a potenciar la reusabilidad de código, ya que, al poder crear colecciones con una misma estructura, pero con diferentes tipos, podremos aplicar lo que se conoce como métodos genéricos, o lo que es lo mismo, métodos que van a pertenecer a la colección y que se podrán ejecutar de forma independiente del tipo de dato que contenga dicha colección.

La única restricción que tienen las colecciones es que los objetos que almacenan en su interior han de ser del mismo tipo, o de tipos relacionados[1].

Las colecciones están disponibles en muchos lenguajes de programación, como C++, Python, entre otros.

En Java, para poder utilizar las colecciones necesitamos la clase *Collection* y el siguiente *import*:

```
import java.util.Collection;
```

Los métodos genéricos que utilizaremos son los de la tabla 1. En los siguientes apartados veremos los distintos tipos de colecciones.

Método	Descripción
int size()	Devuelve la cantidad de elementos que tiene la colección.
void add(Object ob)	Agrega a la colección el elemento que se pasa por parámetro.
void addAll(Collection c)	Agrega a la colección todos los elementos de la colección que se pasa por parámetro.
boolean remove(Object ob)	Elimina de la colección el elemento que se pasa por parámetro.
boolean removeAll(Collection c)	Elimina todos los elementos de la colección que se encuentran en la colección que se pasa por parámetro.
boolean isEmpty()	Indica si la colección está vacía o no.
void clear()	Elimina todos los elementos de la colección.

Tabla 1. Algunos métodos de las colecciones.

1. Esto lo veremos con más profundidad en futuras unidades.

/ 3. Listas

Las listas van a contener una sucesión de elementos, como ya ocurría con los *arrays* que hemos visto en unidades anteriores.

Este tipo de colección admite elementos repetidos en su interior, y añade las siguientes funcionalidades:

- **Acceso a los elementos por posición:** Se permite acceder a un elemento en concreto indicando su posición, , como en un array.
- **Búsqueda de elementos:** Las listas permiten buscar elementos en su interior.
- **Operaciones con rangos:** Con las listas también se posibilita la opción de realizar operaciones indicando un rango de tamaño.
- **Iteración sobre los elementos:** Se permite iterar (recorrer los elementos mientras se hacen operaciones con ellos) sobre los elementos de una forma mucho más eficiente.

Hay dos tipos de listas:

- **ArrayList:** Este tipo de lista es la opción más sencilla. Muy parecido al concepto de array, pero con una serie de métodos ya implementados que nos ayudarán en gran medida con el manejo de estructuras de datos. Lo estudiaremos en profundidad a continuación.
- **LinkedList:** Este tipo de lista mejora su eficiencia en ciertas operaciones, como el agregar elementos, borrarlos y realizar una búsqueda. Está implementada mediante una lista doblemente enlazada.

Usar un tipo u otro de lista lo va a determinar la situación a resolver que se nos presente, aunque vamos a tender a usar la lista *ArrayList*, por su sencillez.

```
public static void main(String[] args) {
    LinkedList<Integer> lista = new LinkedList<>();

    for (int i = 0; i < 10; i++) {
        lista.add(i);
    }

    boolean estael14 = lista.contains(14);
    lista.remove(4);

    System.out.println(lista);
}
```

Código 1. Ejemplo de uso de LinkedList.

Si quieres aprender más sobre las diferencias entre ambos tipos de listas puedes revisar el siguiente enlace: https://www.geeksforgeeks.org/arraylist-vs-linkedlist-java/

/ 4. Caso práctico 1: "¿Usar *arrays* o Arraylist?"

Planteamiento: Pilar y José están realizando una práctica en la que tienen que utilizar arrays.

En ella tienen que realizar varias operaciones con éstos, entre otras: agregar elementos, buscarlos, ordenar el *array* según diferentes criterios...

Ya conocen la existencia de la clase *ArrayList*, y según han visto en clase, tiene un gran potencial, pero como con todo lo nuevo y desconocido, son un poco reacios a utilizarla y creen que es mejor el uso de *arrays* convencionales, ya los dominan mejor y, a fin y al cabo, "podemos obtener el mismo resultado, ¿no?". Se dicen entre ellos.

Nudo: ¿Qué piensas sobre esto? ¿Crees que tienen razón y es mejor utilizar *arrays* convencionales o utilizar la clase *ArrayList*?

Desenlace: Lo más normal cuando no conocemos algo es ser un poco reacios a utilizarlo, pero no por ello debemos cerrarnos a su uso. En este caso en concreto, el uso de la clase *ArrayList* frente a los arrays, merece mucho la pena. La clase *ArrayList*, como clase que es, ya tiene implementada una serie de funciones y una estructura interna totalmente probada y que funciona correctamente (como veremos a continuación). Esto va a hacer que un problema complicado de resolver mediante *arrays* convencionales, si conocemos las funciones de la clase *ArrayList*, sea mucho más fácil de implementar. Entre las acciones que podríamos hacer de una forma mucho más cómoda con *ArrayList*, podrían ser: ordenar el array, recorrerlo, buscar un elemento dentro de él, etc. En definitiva, es muy aconsejable la utilización de *ArrayList* siempre que la situación lo permita, y optar por los *arrays* convencionales para cuando no haya otra alternativa factible para la resolución del problema.

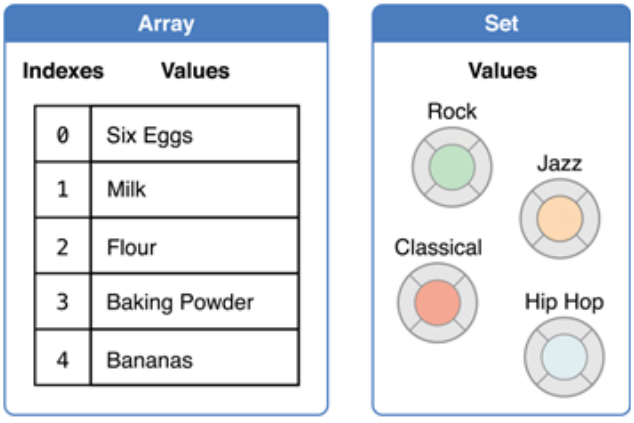

Fig. 2. No siempre la mejor opción son las colecciones, dependerá del tipo de problema.

/ 5. Arraylist

La colección *ArrayList* quizás sea la más **utilizada**, por su **implementación mejorada de los arrays** y por su extrema sencillez. Esta colección representa exactamente lo mismo que teníamos con los *arrays* convencionales, solo que, al ser una clase ya tiene implementado la estructura interna de la misma, permitiendo tener **arrays dinámicos** de una forma extremadamente sencilla.

Otra de las ventajas es que ya tiene una multitud de métodos implementados que van a facilitarnos las cosas de forma considerable. Algunos de los métodos más relevantes de la clase *ArrayList* son:

Método	Descripción
int size()	Devuelve el número de elementos (int)
add(Object ob)	Añade el objeto X al final. Devuelve true.
add(int pos, Object ob)	Inserta el objeto X en la posición indicada.
Object get(int pos)	Devuelve el elemento que está en la posición indicada.
Object remove(int pos)	Elimina el elemento que se encuentra en la posición indicada. Devuelve el elemento eliminado.
Boolean remove(Object ob)	Elimina la primera ocurrencia del objeto X. Devuelve true si el elemento está en la lista.
clear()	Elimina todos los elementos.
set(int pos, Object ob)	Sustituye el elemento que se encuentra en la posición indicada por el objeto X.
Boolean contains(Object ob)	Comprueba si la colección contiene al objeto X. Devuelve true o false.
int indexOf(Object ob)	Devuelve la posición del objeto X. Si no existe devuelve -1.
boolean isEmpty()	Devuelve True si el *ArrayList* está vacio. Si no Devuelve False.
int lastIndexOf(Object ob)	Devuelve la última posición del objeto X. Si no existe devuelve -1.

Tabla 2. Métodos más comunes de ArrayList.

/ 6. Conjuntos (Set)

Los conjuntos, o set en inglés, representan un tipo de colección en la que no puede haber elementos repetidos.

Esta clase no tiene ninguna funcionalidad nueva, teniendo que utilizar el siguiente import para poder utilizarla:

```
import java.util.set;
```

Una cosa a tener muy en cuenta para que esta clase funcione correctamente, es que los elementos con los que se vaya a utilizar deberán tener de forma obligatoria implementados los métodos equals y hashCode. Es decir, si creamos un Set de Persona, la clase Persona deberá tener esos dos métodos implementados.

Existen tres tipos deferentes de conjuntos:

- **HashSet:** Este tipo de conjunto utiliza tablas hash[2] para su implementación, por lo que será bastante eficiente a la hora a de trabajar, aunque hay que destacar que no va a guardar los elementos siguiendo algún tipo de orden. Un inconveniente de este tipo de conjunto es que para que funcione correctamente necesita mucha memoria reservada.

- **TreeSet:** Este tipo de conjunto almacena los elementos ordenándolos por su valor, lo cual, hará que sea bastante más lento que el HashSet. Los elementos con los que se vaya a utilizar deberán implementar la interfaz Comparable.

- **LinkedHashSet:** Este tipo de conjunto utiliza para almacenar los elementos tablas hash y listas enlazadas (LinkedList). Haciendo uso de las tablas hash se conseguirá que sea rápido; con las listas enlazadas se conseguirá que los elementos se almacenen según el orden de inserción.

```
public static void main(String[] args) {
    Set hashset = new HashSet();
    for (int i = 0; i < 100000; i++) {
        hashset.add(i);
    }

    System.out.println(hashset);
    Set treeset = new TreeSet();
    for (int i = 0; i < 100000; i++) {
        treeset.add(i);
    }

    System.out.println(treeset);
    Set linkedhashset = new LinkedHashSet(1_000_000);
    for (int i = 0; i < 100000; i++) {
        linkedhashset.add(i);
    }
    System.out.println(linkedhashset);
}
```

Código 2. Ejemplo de uso de conjuntos.

/ 7. Mapas

Los mapas, o *map* en inglés, son un tipo de colección que admite dos valores por elemento y que no puede contener elementos repetidos.

Los mapas se basan en elementos de tipo clave-valor, esto quiere decir, que cada elemento que contenga el mapa tendrá tanto una clave como un valor, siendo el valor de la clave el identificador del valor.

Cada clave solo puede tener un elemento asociado, aunque si utilizamos un *ArrayList* (por ejemplo) en el campo de valor podremos tener más de un valor, respetando que una clave solo puede tener un valor.

Hay diferentes tipos de mapas:

- **HashMap:** Este tipo de mapa almacena sus elementos en una tabla hash, con lo que será muy rápida a la hora de realizar las operaciones. No hay ningún tipo de orden a la hora de almacenar los elementos en los HashMap. En el momento de crearlo habrá que definir su tamaño.

- **TreeMap:** Este tipo de mapa almacena los elementos ordenándolos por su valor, lo cual hará que sea bastante más lento que el HashMap. Los elementos con los que se vaya a utilizar deberán implementar la interfaz Comparable.

2. Para más información sobre las tablas hash pinchar aquí

- **LinkedTreeMap:** En este caso, se utiliza para almacenar los elementos tablas hash y listas enlazadas, teniendo los mismos efectos que para los conjuntos, es decir, que haciendo uso de las tablas hash se conseguirá que sea rápido y con las listas enlazadas se conseguirá que los elementos se almacenen según el orden de inserción.

A la hora de crear un mapa habrá que indicarle el tipo de elemento que tendrá tanto la clave como el valor.

```
public static void main(String[] args) {
    Map<Integer, String> hashmap = new HashMap<>();
    hashmap.put(1, "Valor uno");
    System.out.println(hashmap);

    Map<Integer, String> treemap = new TreeMap<>();
    treemap.put(1, "Valor uno");
    System.out.println(treemap);

    Map<Integer, String> linkedhashmap = new LinkedHashMap<>();
    linkedhashmap.put(1, "Valor uno");
    System.out.println(linkedhashmap);
}
```

Código 3. Ejemplo de uso de mapas donde la clave es un Integer y el valor un String.

/ 8. Iteradores

Los iteradores nos van a permitir **acceder a los elementos de las colecciones**, ya sean *ArrayList*, Set, Map...

El uso de iteradores para recorrer los elementos es obligatorio en mapas y conjuntos, mientras que en ArrayList no, pero son recomendables. Esto se debe a que no tienen orden, por lo que no se pueden recorrer con un for, como se hace con los arrays. Solo ArrayList dispone de un método para acceder a los elementos por su índice.

La forma de declararlos es la siguiente:

```
Iterator<tipo> nombrevariable = colección.iterator();
```

Los iteradores nos van a proporcionar tres métodos:

- **next():** Este método devuelve el valor del elemento en el que está el iterador.
- **hasNext():** Este método indica si hay un elemento siguiente en el orden de iteración.
- **remove():** Este método eliminará un el elemento que tenga el iterador de la colección.

Los iteradores se pueden usar de forma transparente al usuario mediante el bucle *for-each*.

Este tipo de bucle recorre una colección utilizando iteradores, pero sin que el programador tenga que definirlos.

Su sintaxis es la siguiente:

```
for( tipo nombrevariable : Colecction )
```

Donde:

- Tipo es el tipo de dato que contiene la colección.
- *Nombrevariable* es el nombre que le daremos a la variable iteradora.
- *Colecction* es la colección que vamos a recorrer.

En cada iteración que haga el bucle *for-each*, se irá pasando automáticamente al siguiente elemento, guardándose la variable definida en su cabecera.

Los bucles *for-each* los podemos usar para recorrer todas las colecciones. Los iteradores son elementos seguros, ya que están implementados para no sobrepasar el límite de elementos de una colección.

Vídeo 1. "Ejemplo de uso de iteradores".
https://bit.ly/2ByjTem

/ 9. Documentos XML

El XML, *eXtensible Markup Language* en inglés, es un metalenguaje que nos va a permitir definir marcas para poder de esta forma almacenar datos de forma comprensible.

El XML es un estándar que es utilizado en muchos campos, desde bases de datos, hasta documentos utilizados por instituciones oficiales.

Los documentos XML tienen las siguientes ventajas:

- **Se pueden modificar**, agregando nuevas etiquetas para nueva información, de una forma muy sencilla.
- **Hay verificadores de lenguaje XML ya disponibles** para no tener que crearlos de cero.
- Su **estructura es muy sencilla,** por lo que empezar a utilizarlo es muy fácil.

Los documentos XML se componen de las siguientes partes:

- **Prólogo:** Describe la versión de XML que estamos utilizando o la codificación. Su uso no es obligatorio.
- **Cuerpo:** Es donde se va a definir el elemento raíz del XML que contendrá todos los nodos con la información.
- **Elementos:** Nodos e información del XML.
- **Atributos:** Son propiedades de los nodos. Su uso no es obligatorio.
- **Elementos predefinidos:** Código para representar unos ciertos caracteres especiales.
- **Sección CDATA:** Esta sección permitirá definir una serie de caracteres para poder utilizarlos en el XML.
- **Comentarios:** Típico comentario que sirve para aclarar qué se está haciendo.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Catalog>
   <Book id="bk101">
      <Author>Garghentini, Davide</Author>
      <Title>XML Developer's Guide</Title>
      <Genre>Computer</Genre>
      <Price>44.95</Price>
      <PublishDate>2000-10-01</PublishDate>
      <Description>An in-depth look at creating applications
      with XML.</Description>
   </Book>
   <Book id="bk102">
      <Author>Garcia, Debra</Author>
      <Title>Midnight Rain</Title>
      <Genre>Fantasy</Genre>
      <Price>5.95</Price>
      <PublishDate>2000-12-16</PublishDate>
      <Description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</Description>
   </Book>
</Catalog>
```

Fig. 4. Ejemplo de XML.

/ 10. Documentos DTD

Los documentos DTD (Definición de Tipo de Documento) son unos documentos que definen la descripción de los elementos de un fichero XML y su sintaxis.

Su función consiste en definir las estructuras de los elementos que se van a usar en el fichero XML y que también validará, consiguiendo así una consistencia en todos sus elementos, es decir, que todos los elementos del mismo tipo sean iguales.

Las definiciones de los DTD pueden hacerse en el mismo fichero XML que validarán, aunque lo más recomendable es hacerlo en un fichero aparte e indicarle posteriormente al fichero XML quién es su validador DTD.

Los DTD van a describir los siguientes elementos:

- **Los elementos:** Se indicará qué elementos están permitidos y cuáles no, además de su contenido.
- **La estructura:** Se indicará el orden de los elementos en el documento XML.
- **El anidamiento de elementos:** Se indicará qué elementos van dentro de otros.

Un ejemplo de validador DTD lo podemos ver en la siguiente figura.

```
<!ELEMENT biblioteca (libro*)>
<!ELEMENT libro (ISBN, titulo, autor, editorial?)>
<!ELEMENT ISBN (#PCDATA) >
<!ELEMENT titulo (#PCDATA) >
<!ELEMENT autor (#PCDATA) >
<!ELEMENT editorial (#PCDATA)>
```

Código 5. Ejemplo de DTD.

Donde:

- **<biblioteca>** es un nombre de elemento válido. El * indica que puede haber 0 o más elementos de libro.
- **<libro>** es un nombre de elemento válido. Éste contiene obligatoriamente los elementos ISBN, titulo y autor, siendo editorial opcional por llevar el símbolo "?".
- **<ISBN>** es un nombre de elemento válido. Contiene caracteres.
- **<titulo>** es un nombre de elemento válido. Contiene caracteres.
- **<autor>** es un nombre de elemento válido. Contiene caracteres.
- **<editorial>** es un nombre de elemento válido. Contiene caracteres.

/ 11. Documentos XSD

Los documentos XSD se utilizan para definir la estructura y las restricciones de los documentos XML de una forma precisa.

El XSD es un lenguaje que está escrito en XML y proporciona una mayor potencia que los DTD vistos anteriormente.

Estos se desarrollaron como una alternativa mejor a los DTD, mucho más completa, que intenta arreglar los puntos débiles de la primera.

La definición de los XSD se basa en lo que se conoce como *namespaces* o espacios de nombre. Cada espacio de nombre va a contener una serie de elementos relacionados entre sí, siendo algo parecido a los paquetes en Java.

Una vez escrito un XSD habrá que validarlo para comprobar que no existe ningún error con él.

Un ejemplo de fichero XSD es el de la siguiente figura.

```
ml version="1.0" encoding="UTF-8"?>
d:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Libro">
      <xsd:complexType>
         <xsd:sequence>
              <xsd:element name="Título" type="xsd:string"/>
              <xsd:element name="Autores" type="xsd:string" maxOccurs="10"/>
              <xsd:element name="Editorial" type="xsd:string"/>
         </xsd:sequence>
         <xsd:attribute name="precio" type="xsd:double"/>
      </xsd:complexType>
</xsd:element>
sd:schema>
```

Código 6. Ejemplo de XSD.

En primer lugar, debemos indicar la versión de XML que vamos a usar y la codificación del mismo.

En la segunda línea, se está declarando el espacio de nombres que vamos a utilizar, del que podremos obtener los elementos necesarios para las validaciones.

Podemos ver que los elementos se tendrán que llamar “Libro” y que van a estar compuestos de:

- **Título:** Será el título del libro y es un dato de tipo cadena.
- **Autores:** Serán los autores del libro y será un dato de tipo cadena, pudieron haber 10 como máximo.
- **Editorial:** Será la editorial del libro y será un dato de tipo cadena.
- **Precio:** Será el precio del libro y será un dato de tipo real.

Podemos ver que los atributos Título, Autores y Editorial conforman un dato compuesto.

Una vez tengamos el XSD completo, tendremos que indicar en el fichero XML que ese será su validador mediante la instrucción *xmlns:xsi*.

/ 12. Caso práctico 2: “¿Qué colección utilizo?”

Planteamiento: Pilar y José están repasando todo lo visto en clase sobre las colecciones.

Tienen que hacer un ejercicio en el que probablemente tengan que poner en práctica esos conocimientos.

Pilar le comenta a José que está hecha un lío, ya que son muchas las colecciones que existen, que si *ArrayList*, mapas, conjuntos... y no sabe cuándo tiene que utilizar cada una de ellas.

José le comenta que está en la misma situación, y que por más vueltas que le dé al problema, la solución siempre la termina obteniendo utilizando los *arrays* convencionales.

Nudo: ¿Qué piensas al respecto? ¿Cómo crees que pueden Pilar y José identificar cuándo utilizar cada clase? ¿Podría resolverle todo con *arrays* simplemente?

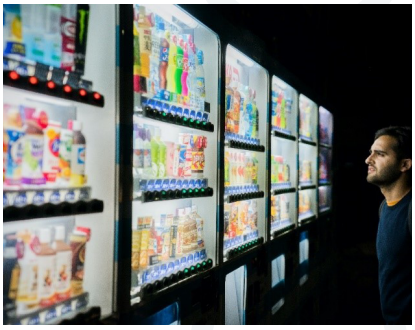

Fig. 3. Difícil elección.

Desenlace: Una de las ventajas de las colecciones es su gran número de elementos, los cuales nos pueden sacar de un aprieto si sabemos cómo utilizarlos, pero claro, esto puede ser un arma de doble filo, ya que, al haber tantas opciones para elegir, y cada una de ellas tener tantos métodos diferentes, no sepamos por cuál decidirnos, terminando así utilizando la más básica, los *arrays* convencionales.

Una muy buena práctica sería listar los distintos tipos de colecciones disponibles junto a su utilidad, para poder consultarlo y decidirse de una forma más fácil, ya que cada nuevo caso que se nos presente, se podría comparar con los ejemplos de este listado, y así establecer un símil que nos ayude a decidirnos. Un ejemplo de esto podría ser que necesitemos obtener una cantidad de números no repetidos, para eso podríamos utilizar un *HashSet*, ya que por definición no deja tener números duplicados, y así no se tendría que utilizar un *array* y tener que gestionar la no repetición de elementos. Las listas las podríamos utilizar para almacenar datos que acepten números repetidos, como por ejemplo las notas de una clase. Los mapas los podremos utilizar cuando necesitemos acceder rápidamente a un dato ligado a una clave, como por ejemplo los datos de una persona, siendo el DNI su clave.

/ 13. Biblioteca DOM

La biblioteca DOM es una biblioteca para **tratamiento de ficheros XML optimizado para el lenguaje de programación Java.** Esta biblioteca ofrece un tratamiento eficiente y transparente de ficheros XML, ya que éstos pueden llegar a formar estructuras complejas, siendo difíciles de tratar si no se impone cierto orden.

Esta biblioteca, es similar a la DOM que se utiliza en JavaScript, solo que está especialmente creada y diseñada para el lenguaje de programación Java. Es una **biblioteca interna**, si recordamos lo visto en unidades anteriores, esto quiere decir que ya está integrada en la propia JDK de Java, así que podremos utilizarla sin necesidad de agregar nada a nuestros proyectos.

Para poder utilizarla necesitaremos el siguiente *import*:

```
import org.w3c.dom.*;
```

Si bien, no utilizaremos el asterisco para importar todas las clases.

Dentro de esta biblioteca tendremos las siguientes clases:

- **Document:** Representa un documento XML.
- **Node:** Representa un nodo XML.
- **NodeList:** Representa una lista de nodos XML.
- **Element:** Representa un elemento del XML. Puede ser un nodo.
- **DOMImplementation:** Representa la configuración de XML.
- **Text:** Representa el valor de un nodo.
- **Comment:** Representa un comentario.

Audio 1. "Otras alternativas para XML en Java"

https://bit.ly/39wX33i

13.1. Operaciones con la biblioteca DOM

Con la biblioteca DOM podremos realizar las siguientes operaciones:

- Crear un fichero XML vacío.
- Cargar un fichero XML desde un fichero.
- Crear el nodo raíz del fichero XML.
- Insertar un nodo al nodo raíz.
- Agregar una propiedad a un nodo de la estructura.

- Eliminar un nodo de la estructura.
- Recorrer todos los nodos de la estructura.
- Recorrer todos los nodos de la estructura a partir de un nodo concreto.
- Insertar un comentario en un nodo.

Por ejemplo, para crear un nodo a partir de una ruta de un fichero existente utilizaremos el siguiente código:

```
DocumentBuilder documentBuilder =
DocumentBuilderFactory.newInstance().
newDocumentBuilder();

Document documentoXML = documentBuilder.
parse(new File("ruta del fichero XML"));
```

Código 7. Creación de nodo a partir de ruta de fichero.

Si ese mismo fichero lo quisiéramos guardar utilizaremos el siguiente código:

```
DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();

DocumentBuilder builder = factory.
newDocumentBuilder();

DOMImplementation implementation = builder.
getDOMImplementation();

documentoXML = implementation.
createDocument(null, etiquetaraiz, null);

documentoXML.setXmlVersion("1.0");
```

Código 8. Guardar fichero.

Vídeo 2. "Tratamiento de XML con DOM".

https://bit.ly/2CQYTAd

/ 14. Resumen y resolución del caso práctico de la unidad

A lo largo de esta unidad, hemos visto qué son las colecciones, las cuales nos permite almacenar dentro de un mismo contenedor muchos elementos del mismo tipo.

Hemos profundizado en las colecciones más importantes, las cuales son las **listas, mapas y conjuntos**, cada uno con una serie de propiedades que los hace únicos, llegando a resolver problemas totalmente diferentes.

También hemos aprendido el concepto de **iterador,** y que gracias a él, podemos recorrer cualquier tipo de colección, sea cual sea el tipo de dato del que esté formada, de una forma eficiente y segura, ya que implementan métodos para no sobrepasar los límites de las mismas.

Por último, hemos estudiado los **ficheros XML**, cómo están estructurados internamente y cómo podemos tratarlos desde Java. Este tratamiento es posible gracias a la **biblioteca DOM**, entre otras, que hace que la gestión de ficheros XML sea eficiente y sencilla.

Resolución del caso práctico de la unidad

En muchas ocasiones vamos a encontrar la situación que están viviendo nuestros amigos. Vamos a tener que hacer una exportación de la información a un formato concreto. El formato XML es usado es muchas situaciones para esto, debido a que su formato está estandarizado.

Debido a esta generalización, Java ya posee este estándar implementado mediante una serie de clases internas, de las que no tendremos que preocuparnos lo más mínimo de cómo se estructuran internamente.

En estos casos, en donde tendríamos que poner la atención, es en ojear la documentación para saber cómo utilizar estas clases, es por esto, que la documentación sea tan importante.

En el caso concreto que deben realizar nuestros amigos, deberemos saber cuántas calificaciones podrá haber y a qué exámenes se refieren. Conocido esto, por ejemplo, el examen del tema 1 podría ser tal que *<nota_examen1> 7.25 </nota_examen1>*, y así sucesivamente...

/ 15. Bibliografía

Amor, R. V. (2020, marzo 6). *Introducción a Colecciones en Java*. Recuperado 22 de mayo de 2020, de https://www.adictosaltrabajo.com/2015/09/25/introduccion-a-colecciones-en-java/

De Lamarca Lapuente, M. J. (s. f.). *DTDs y XML Esquema*. Recuperado 22 de mayo de 2020, de http://www.hipertexto.info/documentos/dtds.htm

Colaboradores de Wikipedia. (2020a, abril 27). *Extensible Markup Language* - Wikipedia, la enciclopedia libre. Recuperado 23 de mayo de 2020, de https://es.wikipedia.org/wiki/Extensible_Markup_Language#Ventajas_del_XML

Colaboradores de Wikipedia. (2020a, abril 17). *XML Schema* - Wikipedia, la enciclopedia libre. Recuperado 23 de mayo de 2020, de https://es.wikipedia.org/wiki/XML_Schema

Colaboradores de Wikipedia. (2020a, febrero 6). *Definición de tipo de documento* - Wikipedia, la enciclopedia libre. Recuperado 23 de mayo de 2020, de https://es.wikipedia.org/wiki/Definici%C3%B3n_de_tipo_de_documento

Colaboradores de Wikipedia. (2019, julio 30). *JDOM* - Wikipedia, la enciclopedia libre. Recuperado 23 de mayo de 2020, de https://es.wikipedia.org/wiki/JDOM