

Aplicación de las estructuras de almacenamiento.

Caso práctico



Ana ha recibido un pequeño encargo de parte de su tutora, **María**. Se trata de que realice un pequeño programita, muy sencillo pero fundamental.

-Hola **Ana** -dice **María**-, hoy tengo una tarea especial para ti.

-¿Sí? -responde **Ana**-. Estoy deseando, últimamente no hay nada que se me resista, llevo dos semanas en racha.

-Bueno, quizás esto se te resista un poco más, es fácil, pero tiene cierta complicación. Un cliente para el que hicimos una aplicación, nos ha dicho si podemos ayudarle. El cliente tiene una aplicación para gestionar los pedidos que recibe de sus clientes. Normalmente, recibe por correo electrónico los pedidos en un formato concreto que todos sus clientes llevan tiempo usando. El cliente suele transcribir el pedido desde el correo electrónico a la aplicación, copiando dato a dato, pero nos ha preguntado si es posible que copie todo el pedido de golpe, y que la aplicación lo procese, detectando posibles errores en el pedido.

-¿Qué? -dice **María** con cierta perplejidad.

-Me alegra que te guste -dice **María** esbozando una sonrisa picara-, sé que te gustan los retos.

-Pero, ¿eso cómo se hace? ¿Cómo compruebo yo si el pedido es válido? ¿Y si el cliente ha puesto un producto que no existe?

-No mujer, se trata de comprobar si el pedido tiene el formato correcto, y transformarlo de forma que se genere un documento XML con los datos del pedido. Un documento XML luego se puede incorporar fácilmente a los otros pedidos que tenga el cliente en su base de datos. De la verificación de si hay algún producto que no existe, o de si hay alguna incoherencia en el pedido se encarga otra parte del software, que ya he realizado yo.

-Bueno -dice **María** justo después de resoplar tres veces seguidas-, empieza por contarme que es el formato XML.

Cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables. Manejar los datos de un único pedido en una aplicación puede ser relativamente sencillo, pues un pedido está compuesto por una serie de datos y eso simplemente se traduce en varias variables. Pero, ¿qué ocurre cuando en una aplicación tenemos que gestionar varios pedidos a la vez?

Lo mismo ocurre en otros casos. Para poder realizar ciertas aplicaciones se necesita poder manejar datos que van más allá de meros **datos simples** (números y letras). A veces, los datos que tiene que manejar la aplicación son **datos compuestos**, es decir, datos que están compuestos a su vez de varios datos más simples. Por ejemplo, un pedido está compuesto por varios datos, los datos podrían ser el cliente que hace el pedido, la dirección de entrega, la fecha requerida de entrega y los artículos del pedido.



Los datos compuestos son un tipo de estructura de datos, y en realidad ya los has manejado. **Las clases son un ejemplo de estructuras de datos que permiten almacenar datos compuestos**, y el objeto en sí, la instancia de una clase, sería el dato compuesto. Pero, a veces, los datos tienen estructuras aún más complejas, y son necesarias soluciones adicionales.

Aquí podrás aprender esas soluciones adicionales. Esas soluciones consisten básicamente en la capacidad de poder manejar varios datos del mismo o diferente tipo de forma dinámica y flexible.



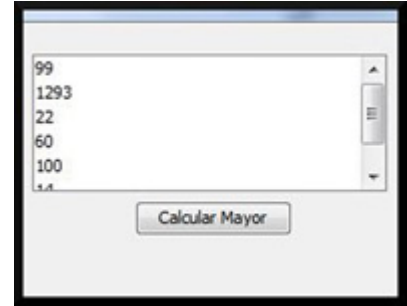
Materiales formativos de F.P. Online propiedad del Ministerio de Educación, Cultura y Deporte.

[Aviso Legal](#)

1.- Introducción a las estructuras de almacenamiento.

¿Cómo almacenarías en memoria un listado de números del que tienes que extraer el valor máximo? Seguro que te resultaría fácil. Pero, ¿y si el listado de números no tiene un tamaño fijo, sino que puede variar en tamaño de forma dinámica? Entonces la cosa se complica.

Un listado de números que aumenta o decrece en tamaño es una de las cosas que aprenderás a utilizar aquí, utilizando estructuras de datos. Pasaremos por alto las clases y los objetos, pues ya los has visto con anterioridad, pero debes saber que las clases en sí mismas son la evolución de un tipo de estructuras de datos conocidas como datos compuestos (también llamadas **registros**). Las clases, además de aportar la ventaja de agrupar datos relacionados entre sí en una misma estructura (característica aportada por los datos compuestos), permiten agregar métodos que manejen dichos datos, ofreciendo una herramienta de programación sin igual. Pero todo esto ya lo sabías.



Las estructuras de almacenamiento, en general, se pueden clasificar de varias formas. Por ejemplo, atendiendo a si pueden almacenar datos de diferente tipo, o si solo pueden almacenar datos de un solo tipo, se pueden distinguir:

- ✓ Estructuras con capacidad de **almacenar varios datos del mismo tipo**: varios números, varios caracteres, etc. Ejemplos de estas estructuras son los arrays, las cadenas de caracteres, las listas y los conjuntos.
- ✓ Estructuras con capacidad de **almacenar varios datos de distinto tipo**: números, fechas, cadenas de caracteres, etc., todo junto dentro de una misma estructura. Ejemplos de este tipo de estructuras son las clases.

Otra forma de clasificar las estructuras de almacenamiento va en función de si pueden o no cambiar de tamaño de forma dinámica:

- ✓ **Estructuras cuyo tamaño se establece en el momento de la creación o definición** y su tamaño no puede variar después. Ejemplos de estas estructuras son los **arrays** y las matrices (arrays multidimensionales).
- ✓ **Estructuras cuyo tamaño es variable (conocidas como estructuras dinámicas)**. Su tamaño crece o decrece según las necesidades de forma dinámica. Es el caso de las listas, árboles, conjuntos y, como veremos también, el caso de algunos tipos de cadenas de caracteres.

Por último, atendiendo a la forma en la que los datos se ordenan dentro de la estructura, podemos diferenciar varios tipos de estructuras:

- ✓ **Estructuras que no se ordenan de por sí**, y debe ser el programador el encargado de ordenar los datos si fuera necesario. Un ejemplo de estas estructuras son los arrays.
- ✓ **Estructuras ordenadas**. Se trata de estructuras que al incorporar un dato nuevo a todos los datos existentes, este se almacena en una posición concreta que irá en función del orden. El orden establecido en la estructura puede variar dependiendo de las necesidades del programa: alfabético, orden numérico de mayor a menor, momento de inserción, etc.

Todavía no conoces mucho de las estructuras, y probablemente todo te suena raro y extraño. No te preocupes, poco a poco irás descubriéndolas. Verás que son sencillas de utilizar y muy cómodas.



Autoevaluación

El tamaño de las estructuras de almacenamiento siempre se determina en el momento de la creación. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

2.- Cadenas de caracteres.

Caso práctico

Ana está asustada, le acaban de asignar una nueva tarea y le acaban de enviar por correo el formato del archivo que tiene que procesar. El archivo tiene un formato concreto, un tanto extraño. Primero tiene una cabecera con información general del pedido, y luego tiene los artículos del pedido, cada uno en una línea diferente. ¿Podrá realizar la tarea encomendada? Seguro que sí. En el siguiente enlace puedes ver como es el formato del archivo:

La empresa recibe los pedidos siguiendo un modelo normalizado. En realidad se trata de una pequeña aplicación web que al rellenar un formulario, envía un correo electrónico con la información siguiendo una estructura fija. A continuación puedes ver un ejemplo:

```
## PEDIDO ##
Número de pedido: { 20304 }
Cliente: { Muebles Bonitos S.A. }
Código del cliente: { 00293 }
Dirección de factura: { C/ De en frente, 11 }
} Dirección de entrega: { C/ De al lado, 22 }
Nombre del contacto: { Elias }
Teléfono del contacto: { 987654321 }
Correo electrónico del contacto: { mail@mail1234.com }
Fecha preferente de entrega: { 19/11/2012 }
Forma de pago: { Transferencia }
## ARTICULOS ##
{ Código Artículo | Descripción | Cantidad }
{ 0001231 | Tuercas tipo 1 | 200 }
{ 0001200 | Tornillos tipo 1 | 200 }
{ 0002200 | Arandelas tipo 2 | 200 }
## FIN ARTICULOS ##
## FIN PEDIDO ##
```



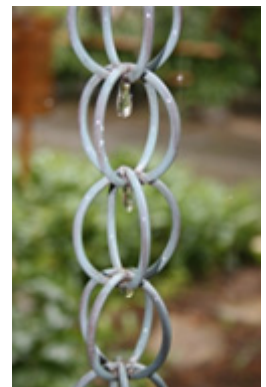
Como puedes observar, el pedido tiene una estructura fija, y solamente cambian los datos que están entre llaves. El resto del texto es fijo, dado que como se ha dicho antes, el pedido lo genera una aplicación web, y lo envía por correo electrónico al cliente.

Cuando un cliente solicita un pedido a esta empresa, no puede poner en ningún campo los símbolos "{", "}", ni "|", porque entonces entraría en conflicto con el formato enviado por correo y daría problemas, así que se optó por impedir que el cliente pudiera poner dichos caracteres extraños. Esto garantiza que cada campo del pedido se va a poder leer adecuadamente.

Lo más conflictivo, es la sección de artículos del pedido, en su primera parte, tiene una especie de cabecera que ayuda a identificar que es cada una de las columnas. Y después, tiene tantas líneas como artículos tenga el pedido, si tiene 100 artículos, tendrán 100 líneas que incluirán: código del artículo, descripción del artículo y la cantidad.

Probablemente, una de las cosas que mas utilizarás cuando estés programando en cualquier lenguaje de programación son las cadenas de caracteres. Las cadenas de caracteres son estructuras de almacenamiento que permiten almacenar una secuencia de caracteres de casi cualquier longitud. Y la pregunta ahora es, ¿qué es un carácter?

En Java y en todo lenguaje de programación, y por ende, en todo sistema informático, los caracteres se **codifican** como secuencias de bits que representan a los símbolos usados en la comunicación humana. Estos símbolos pueden ser letras, números, símbolos matemáticos e incluso **ideogramas** y **pictogramas**.



Para saber más

Si quieres puedes profundizar en la codificación de caracteres leyendo el siguiente artículo de la Wikipedia.

[Codificación de caracteres.](#)

La forma más habitual de ver escrita una cadena de caracteres es como un literal de cadena. Consiste simplemente en una secuencia de caracteres entre comillas dobles, por ejemplo: **"Ejemplo de cadena de caracteres"**.

En Java, los literales de cadena son en realidad instancias de la clase **String**, lo cual quiere decir que, por el mero hecho de escribir un literal, se creará una instancia de dicha clase. Esto da mucha flexibilidad, puesto que permite crear cadenas de muchas formas diferentes, pero obviamente consume mucha memoria. La forma más habitual es crear una cadena partiendo de un literal:

```
String cad="Ejemplo de cadena";
```

En este caso, el literal de cadena situado a la derecha del igual es en realidad una instancia de la clase **String**. Al realizar esta asignación hacemos que la variable **cad** se convierta en una referencia al objeto ya creado. Otra forma de crear una cadena es usando el operador **new** y un constructor, como por ejemplo:

```
String cad=new String ("Ejemplo de cadena");
```

Cuando se crean las cadenas de esta forma, se realiza una copia en memoria de la cadena pasada por parámetro. La nueva instancia de la clase **String** hará referencia por tanto a la copia de la cadena, y no al original.

Reflexiona

Fijate en el siguiente línea de código, ¿cuántas instancias de la clase **String** ves?

```
String cad="Ejemplo de cadena 1"; cad="Ejemplo de cadena 2"; cad=new  
String("Ejemplo de cadena 3");
```

2.1.- Operaciones avanzadas con cadenas de caracteres (I).

¿Qué operaciones puedes hacer con una cadena? Muchas más de las que te imaginas. Empezaremos con la operación mas sencilla: la concatenación. La concatenación es la unión de dos cadenas, para formar una sola. En Java es muy sencillo, pues sólo tienes que utilizar el operador de concatenación (signo de suma):

```
String cad = "¡Bien"+"venido!";
System.out.println(cad);
```

En la operación anterior se esta creando una nueva cadena, resultado de unir dos cadenas: una cadena con el texto "**¡Bien**", y otra cadena con el texto "**venido!**". La segunda línea de código muestra por la [salida estándar](#) el resultado de la concatenación. El resultado de su ejecución será que aparecerá el texto "**¡Bienvenido!**" por la pantalla.

Otra forma de usar la concatenación, que ilustra que cada literal de cadena es a su vez una instancia de la clase **String**, es usando el método **concat** del objeto **String**:

```
String cad="¡Bien".concat("venido!");
System.out.printf(cad);
```

Fíjate bien en la expresión anterior, pues genera el mismo resultado que la primera opción y en ambas participan tres instancias de la clase **String**. Una instancia que contiene el texto "**¡Bien**", otra instancia que contiene el texto "**venido!**", y otra que contiene el texto "**¡Bienvenido!**". La tercera cadena se crea nueva al realizar la operación de concatenación, sin que las otras dos hayan desaparecido. Pero no te preocupes por las otras dos cadenas, pues se borrarán de memoria cuando el [recolector de basura](#) detecte que ya no se usan.

Fíjate además, que se puede invocar directamente un método de la clase **String**, posponiendo el método al literal de cadena. Esto es una señal de que al escribir un literal de cadena, se crea una instancia del [objeto inmutable String](#).

Pero no solo podemos concatenar una cadena a otra cadena. Gracias al método **toString()** podemos concatenar cadenas con literales numéricos e instancias de otros objetos sin problemas.

El método **toString()** es un método disponible en todas las clases de Java. Su objetivo es simple, permitir la conversión de una instancia de clase en cadena de texto, de forma que se pueda convertir a texto el contenido de la instancia. Lo de convertir, no siempre es posible, hay clases fácilmente convertibles a texto, como es la clase **Integer**, por ejemplo, y otras que no se pueden convertir, y que el resultado de invocar el método **toString()** es información relativa a la instancia.

La gran ventaja de la concatenación es que el método **toString()** se invocará automáticamente, sin que tengamos que especificarlo, por ejemplo:

```
Integer i1=new Integer (1223); // La instancia i1 de la clase Integer contiene el número 1223
System.out.println("Número: " + i1); // Se mostrará por pantalla el texto "Número: 1223"
```

En el ejemplo anterior, se ha invocado automáticamente **i1.toString()**, para convertir el número a cadena. Esto se realizará para cualquier instancia de clase concatenada, pero cuidado, como se ha dicho antes, no todas las clases se pueden convertir a cadenas.





Autoevaluación

¿Qué se mostrará como resultado de ejecutar el siguiente código `System.out.println(4+1+"-"+4+1)`; ?

- ☐ Mostrará la cadena "5-41".
- ☐ Mostrará la cadena "41-14".
- ☐ Esa operación dará error.

2.1.1.- Operaciones avanzadas con cadenas de caracteres (II).

Vamos a continuar revisando las operaciones que se pueden realizar con cadenas. Como verás las operaciones a realizar se complican un poco a partir de ahora. En todos los ejemplos la variable **cad** contiene la cadena "¡Bienvenido!", como se muestra en las imágenes.

- ✓ `int length()`. Retorna un número entero que contiene la longitud de una cadena, resultado de contar el número de caracteres por la que esta compuesta. Recuerda que un espacio es también un carácter.

String cad="¡Bienvenido!"
Longitud = 12

- ✓ `char charAt(int pos)`. Retorna el carácter ubicado en la posición pasada por parámetro. El carácter obtenido de dicha posición será almacenado en un tipo de dato `char`. Las posiciones se empiezan a contar desde el 0 (y no desde el 1), y van desde 0 hasta longitud - 1. Por ejemplo, el código siguiente mostraría por pantalla el carácter "v":

String cad="¡Bienvenido!"
0 1 2 3 4 5 6 7 8 9 10 11 Posición
cad.charAt(0) cad.charAt(4) cad.charAt(11)

```
char t = cad.charAt(5);
System.out.println(t);
```

- ✓ `String substring(int beginIndex, int endIndex)`. Este método permite extraer una subcadena de otra de mayor tamaño. Una cadena compuesta por todos los caracteres existentes entre la posición `beginIndex` y la posición `endIndex - 1`. Por ejemplo, si pusieramos `cad.substring(0,5)` en nuestro programa, sobre la variable `cad` anterior, dicho método devolvería la subcadena "¡Bien" tal y como se muestra en la imagen.

String cad="¡Bienvenido!"
0 1 2 3 4 5 6 7 8 9 10 11 Posición
substring(0,5) substring(5,11)
String cad="¡Bienvenido!"
0 1 2 3 4 5 6 7 8 9 10 11 Posición
substring(0)

- ✓ `String substring (int beginIndex)`. Cuando al método `substring` solo le proporcionamos un parámetro, extraerá una cadena que comenzará en el carácter con posición `beginIndex` e irá hasta el final de la cadena. En el siguiente ejemplo se mostraría por pantalla la cadena "ienvenido!":

```
String subcad = cad.substring(2);
System.out.println(subcad);
```

Otra operación muy habitual es la conversión de número a cadena y de cadena a número. Imagínate que un usuario introduce su edad. Al recoger la edad desde la interfaz de usuario, capturarás generalmente una cadena, pero, ¿cómo comprobamos que la edad es mayor que 0? Para poder realizar esa comprobación tienes que pasar la cadena a número. Empezaremos por ver como se convierte un número a cadena.

Los números generalmente se almacenan en memoria como números binarios, es decir, secuencias de unos y ceros con los que se puede operar (sumar, restar, etc.). No debes confundir los tipos de datos que contienen números (**int**, **short**, **long**, **float** y **double**) con las secuencias de caracteres que representan un número. No es lo mismo 123 que "123", el primero es un número y el segundo es una cadena formada por tres caracteres: '1', '2' y '3'.

Convertir un número a cadena es fácil desde que existe, para todas las clases Java, el método **toString()**. Gracias a ese método podemos hacer cosas como las siguientes:

```
String cad2="Número cinco: " + 5;
System.out.println(cad2);
```


El resultado del código anterior es que se mostrará por pantalla "**Número cinco: 5**", y no dará ningún error. Esto es posible gracias a que Java convierte el número 5 a su [clase envoltorio](#) (wrapper class) correspondiente (**Integer**, **Float**, **Double**, etc.), y después ejecuta automáticamente el método **toString()** de dicha clase.

Reflexiona

¿Cuál crees que será el resultado de poner **System.out.println("A"+5f)**? Pruébalo y recuerda: no olvides indicar el tipo de literal (f para los literales de números flotantes, y d para los literales de números dobles), así obtendrás el resultado esperado y no algo diferente.

2.1.2.- Operaciones avanzadas con cadenas de caracteres (III).

¿Cómo comprobarías que la cadena "3" es mayor que 0? No puedes comparar directamente una cadena con un número, así que necesitarás aprender cómo convertir cadenas que contienen números a tipos de datos numéricos (**int**, **short**, **long**, **float** o **double**). Esta es una operación habitual en todos los lenguajes de programación, y Java, para este propósito, ofrece los métodos **valueOf**, existentes en todas las clases envoltorio descendientes de la clase **Number**: **Integer**, **Long**, **Short**, **Float** y **Double**.

Veamos un ejemplo de su uso para un número de doble precisión, para el resto de las clases es similar:

```
String c="1234.5678";
double n;
try {
    n=Double.valueOf(c).doubleValue();
} catch (NumberFormatException e)
{ /* Código a ejecutar si no se puede convertir */ }
```



Fijate en el código anterior, en él puedes comprobar cómo la cadena *c* contiene en su interior un número, pero escrito con dígitos numéricos (caracteres). El código escrito esta destinado a transformar la cadena en número, usando el método **valueOf**. Este método lanzará la excepción **NumberFormatException** si no consigue convertir el texto a número. En el siguiente archivo, tienes un ejemplo más completo, donde aparecen también ejemplos para los otros tipos numéricos:

Ejemplo de conversión de cadena de texto a número.

Seguro que ya te vas familiarizando con este embrollo y encontrarás interesante todas estas operaciones. Ahora te planteamos otro reto: imagina que tienes que mostrar el precio de un producto por pantalla. Generalmente, si un producto vale, por ejemplo 3,3 euros, el precio se debe mostrar como "**3,30 €**", es decir, se le añade un cero extra al final para mostrar las centésimas. Con lo que sabemos hasta ahora, usando la concatenación en Java, podemos conseguir que una cadena se concatene a un número flotante, pero el resultado no será el esperado. Prueba el siguiente código:

```
float precio=3.3f;
System.out.println("El precio es: "+precio+"€");
```

Si has probado el código anterior, habrás comprobado que el resultado no muestra "**3,30 €**" sino que muestra "**3,3 €**". ¿Cómo lo solucionamos? Podríamos dedicar bastantes líneas de código hasta conseguir algo que realmente funcione, pero no es necesario, dado que Java y otros lenguajes de programación (como C), disponen de lo que se denomina [formateado de cadenas](#). En Java podemos "formatear" cadenas a través del método estático **format** disponible en el objeto **String**. Este método permite crear una cadena proyectando los argumentos en un formato específico de salida. Lo mejor es verlo con un ejemplo, veamos cuál sería la solución al problema planteado antes:

```
float precio=3.3f;
String salida=String.format ("El precio es: %.2f €", precio));
System.out.println(salida);
```

El formato de salida, también denominado "cadena de formato", es el primer argumento del método **format**. La variable **precio**, situada como segundo argumento, es la variable que se proyectará en la salida siguiendo un formato concreto. Seguro que te preguntarás, ¿qué es "**%.2f**"? Pues es un [especificador de formato](#), e indica cómo se deben formatear o proyectar los argumentos que hay después de la cadena de formato en el método **format**.

Debes conocer

Es necesario que conozcas bien el método `format` y los especificadores de formato. Por ese motivo, te pedimos que leas atentamente el siguiente documento:

[Formateado de cadenas en Java.](#)

2.1.3.- Operaciones avanzadas con cadenas de caracteres (IV).

¿Cómo puedo comprobar si dos cadenas son iguales? ¿Qué más operaciones ofrece Java sobre las cadenas? Java ofrece un montón de operaciones más sobre . En la siguiente tabla puedes ver las operaciones más importantes. En todos los ejemplos expuestos, las variables **cad1**, **cad2** y **cad3** son cadenas ya existentes, y la variable **num** es un número entero mayor o igual a cero.



Métodos importantes de la clase String.

Método.	Descripción
<code>cad1.compareTo(cad2)</code>	Permite comparar dos cadenas entre sí lexicográficamente. Retornará 0 si son iguales, un número menor que cero si la cadena (<code>cad1</code>) es anterior en orden alfabético a la que se pasa por argumento (<code>cad2</code>), y un número mayor que cero si la cadena es posterior en orden alfabético.
<code>cad1.equals(cad2)</code>	Cuando se comparan si dos cadenas son iguales, no se debe usar el operador de comparación "=", sino el método <code>equals</code> . Retornará <code>true</code> si son iguales, y <code>false</code> si no lo son.
<code>cad1.compareToIgnoreCase(cad2)</code> <code>cad1.equalsIgnoreCase(cad2)</code>	El método <code>compareToIgnoreCase</code> funciona igual que el método <code>compareTo</code> , pero ignora las mayúsculas y las minúsculas a la hora de hacer la comparación. Las mayúsculas van antes en orden alfabético que las minúsculas, por lo que hay que tenerlo en cuenta. El método <code>equalsIgnoreCase</code> es igual que el método <code>equals</code> pero sin tener en cuenta las minúsculas.
<code>cad1.trim()</code>	Genera una copia de la cadena eliminando los espacios en blanco anteriores y posteriores de la cadena.
<code>cad1.toLowerCase()</code>	Genera una copia de la cadena con todos los caracteres a minúscula.
<code>cad1.toUpperCase()</code>	Genera una copia de la cadena con todos los caracteres a mayúsculas.
<code>cad1.indexOf(cad2)</code> <code>cad1.indexOf(cad2,num)</code>	Si la cadena o carácter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1. Opcionalmente se le puede indicar la posición a partir de la cual buscar, lo cual es útil para buscar varias apariciones de una cadena dentro de otra.
<code>cad1.contains(cad2)</code>	Retornará <code>true</code> si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará <code>false</code> .
<code>cad1.startsWith(cad2)</code>	Retornará <code>true</code> si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .
<code>cad1.endsWith(cad2)</code>	Retornará <code>true</code> si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .

Método.	Descripción
<code>cad1.replace(cad2,cad3)</code>	Generará una copia de la cadena <code>cad1</code> , en la que se reemplazarán todas las apariciones de <code>cad2</code> por <code>cad3</code> . El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "zzz" por "xx" en la cadena "zzzzz" generará "xxzz" y no "zzxx".



Autoevaluación

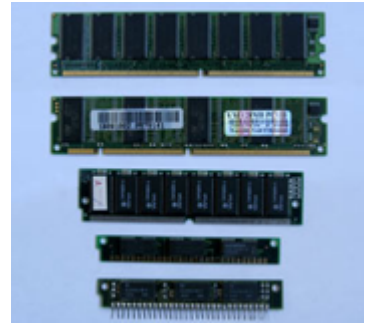
¿Cuál será el resultado de ejecutar `cad1.replace("l","j").indexOf("ja")` si `cad1` contiene la cadena "hojalata"?

- ☐ 2.
- ☐ 3.
- ☐ 4.
- ☐ -1.

2.1.4.- Operaciones avanzadas con cadenas de caracteres (V).

¿Sabes cuál es el principal problema de las cadenas de caracteres? Su alto consumo de memoria. Cuando realizamos un programa que realiza muchísimas operaciones con cadenas, es necesario optimizar el uso de memoria.

En Java, **String** es un objeto inmutable, lo cual significa, entre otras cosas, que cada vez que creamos un **String**, o un literal de **String**, se crea un nuevo objeto que no es modificable. Java proporciona la clase **StringBuilder**, la cual es un **mutable**, y permite una mayor optimización de la memoria. También existe la clase **StringBuffer**, pero consume mayores recursos al estar pensada para aplicaciones **multi-hilo**, por lo que en nuestro caso nos centraremos en la primera.



Pero, ¿en que se diferencian **StringBuilder** de la clase **String**? Pues básicamente en que la clase **StringBuilder** permite modificar la cadena que contiene, mientras que la clase **String** no. Como ya se dijo antes, al realizar operaciones complejas se crea una nueva instancia de la clase **String**.

Veamos un pequeño ejemplo de uso de esta clase. En el ejemplo que vas a ver, se parte de una cadena con errores, que modificaremos para ir haciéndola legible. Lo primero que tenemos que hacer es crear la instancia de esta clase. Se puede inicializar de muchas formas, por ejemplo, partiendo de un literal de cadena:

```
StringBuilder strb=new StringBuilder ("Hoal Muuundo");
```

Y ahora, usando los métodos **append** (insertar al final), **insert** (insertar una cadena o carácter en una posición específica), **delete** (eliminar los caracteres que hay entre dos posiciones) y **replace** (reemplazar los caracteres que hay entre dos posiciones por otros diferentes), rectificaremos la cadena anterior y la haremos correcta:

1. `strb.delete(6,8)`; Eliminamos las 'uu' que sobran en la cadena. La primera 'u' que sobra está en la posición 6 (no olvides contar el espacio), y la última 'u' a eliminar está en la posición 7. Para eliminar dichos caracteres de forma correcta hay que pasar como primer argumento la posición 6 (posición inicial) y como segundo argumento la posición 8 (posición contigua al último carácter a eliminar), dado que la posición final no indica el último carácter a eliminar, sino el carácter justo posterior al último que hay que eliminar (igual que ocurría con el método `substring`).
2. `strb.append ("!")`; Añadimos al final de la cadena el símbolo de cierre de exclamación.
3. `strb.insert (0,"¡")`; Insertamos en la posición 0, el símbolo de apertura de exclamación.
4. `strb.replace (3,5,"la")`; Reemplazamos los caracteres 'al' situados entre la posición inicial 3 y la posición final 4, por la cadena 'la'. En este método ocurre igual que en los métodos `delete` y `substring`, en vez de indicar como posición final la posición 4, se debe indicar justo la posición contigua, es decir 5.

StringBuilder contiene muchos métodos de la clase **String** (`charAt`, `indexOf`, `length`, `substring`, `replace`, etc.), pero no todos, pues son clases diferentes con funcionalidades realmente diferentes.

Debes conocer

En la siguiente página puedes encontrar más información (en inglés) sobre como utilizar la clase **StringBuilder**.

[Uso de la clase StringBuilder.](#)



Autoevaluación

Rotar una cadena es poner simplemente el primer carácter al final, y retroceder el resto una posición. Después de unas cuantas rotaciones la cadena queda igual. ¿Cuál de las siguientes expresiones serviría para hacer una rotación (rotar solo una posición)?

- ☐ `stb.delete (0,1); strb.append(stb.charAt(0));`

- ☐ `strb.append(strb.charAt(0));strb.delete(0, 1);`
- ☐ `strb.append(strb.charAt(0));strb.delete(0);`
- ☐ `strb.append(strb.charAt(1));strb.delete(1,2);`

2.2.- Expresiones regulares (I).

¿Tienen algo en común todos los números de DNI y de NIE? ¿Podrías hacer un programa que verificara si un DNI o un NIE es correcto? Seguro que sí. Si te fijas, los números de DNI y los de NIE tienen una estructura fija: X1234567Z (en el caso del NIE) y 1234567Z (en el caso del DNI). Ambos siguen un **patrón** que podría describirse como: una letra inicial opcional (solo presente en los NIE), seguida de una secuencia numérica y finalizando con otra letra. ¿Fácil no?



Pues esta es la función de las expresiones regulares: **permitir comprobar si una cadena sigue o no un patrón preestablecido**. Las expresiones regulares son un mecanismo para describir esos patrones, y se construyen de una forma relativamente sencilla. Existen muchas librerías diferentes para trabajar con expresiones regulares, y casi todas siguen, más o menos, una sintaxis similar, con ligeras variaciones. Dicha sintaxis nos permite indicar el patrón de forma cómoda, como si de una cadena de texto se tratase, en la que determinados símbolos tienen un significado especial. Por ejemplo "[01]+" es una expresión regular que permite comprobar si una cadena conforma un número binario. Veamos cuáles son las reglas generales para construir una expresión regular:

- ✓ Podemos indicar que una cadena contiene un conjunto de símbolos fijo, simplemente poniendo dichos símbolos en el patrón, excepto para algunos símbolos especiales que necesitarán un carácter de escape como veremos más adelante. Por ejemplo, el patrón "aaa" admitirá cadenas que contengan tres aes.
- ✓ "[xyz]". Entre corchetes podemos indicar opcionalidad. Solo uno de los símbolos que hay entre los corchetes podrá aparecer en el lugar donde están los corchetes. Por ejemplo, la expresión regular "aaa[xy]" admitirá como válidas las cadenas "aaax" y la cadena "aaay". **Los corchetes representan una posición de la cadena que puede tomar uno de varios valores posibles.**
- ✓ "[a-z]" "[A-Z]" "[a-zA-Z]". Usando el guión y los corchetes podemos indicar que el patrón admite cualquier carácter entre la letra inicial y la final. Es importante que sepas que se diferencia entre letras mayúsculas y minúsculas, no son iguales de cara a las expresiones regulares.
- ✓ "[0-9]". Y nuevamente, usando un guión, podemos indicar que se permite la presencia de un dígito numérico entre 0 y 9, cualquiera de ellos, pero solo uno.

Con las reglas anteriores podemos indicar el conjunto de símbolos que admite el patrón y el orden que deben tener. Si una cadena no contiene los símbolos especificados en el patrón, en el mismo orden, entonces la cadena no encajará con el patrón. Veamos ahora como indicar repeticiones:

- ✓ "a?". Usaremos el interrogante para indicar que un símbolo puede aparecer una vez o ninguna. De esta forma la letra "a" podrá aparecer una vez o simplemente no aparecer.
- ✓ "a*". Usaremos el asterisco para indicar que un símbolo puede aparecer una vez o muchas veces, pero también ninguna. Cadenas válidas para esta expresión regular serían "aa", "aaa" o "aaaaaaaa".
- ✓ "a+". Usaremos el símbolo de suma para indicar que otro símbolo debe aparecer al menos una vez, pudiendo repetirse cuantas veces quiera.
- ✓ "a{1,4}". Usando las llaves, podemos indicar el número mínimo y máximo de veces que un símbolo podrá repetirse. El primer número del ejemplo es el número 1, y quiere decir que la letra "a" debe aparecer al menos una vez. El segundo número, 4, indica que como máximo puede repetirse cuatro veces.
- ✓ "a{2,}". También es posible indicar solo el número mínimo de veces que un carácter debe aparecer (sin determinar el máximo), haciendo uso de las llaves, indicando el primer número y poniendo la coma (no la olvides).
- ✓ "a{5}". A diferencia de la forma anterior, si solo escribimos un número entre llaves, sin poner la coma detrás, significará que el símbolo debe aparecer un número exacto de veces. En este caso, la "a" debe aparecer exactamente 5 veces.
- ✓ "[a-z]{1,4}[0-9]+". Los indicadores de repetición se pueden usar también con corchetes, dado que los corchetes representan, básicamente, un símbolo. En el ejemplo anterior se permitiría de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.

Ejercicio resuelto

Con lo visto hasta ahora ya es posible construir una expresión regular capaz de verificar si una cadena contiene un DNI o un NIE, ¿serías capaz de averiguar cuál es dicha expresión regular?

2.2.1.- Expresiones regulares (II).

¿Y cómo uso las expresiones regulares en un programa? Pues de una forma sencilla. Para su uso, Java ofrece las clases **Pattern** y **Matcher** contenidas en el paquete **java.util.regex.***. La clase **Pattern** se utiliza para procesar la expresión regular y "compilarla", lo cual significa verificar que es correcta y dejarla lista para su utilización. La clase **Matcher** sirve para comprobar si una cadena cualquiera sigue o no un patrón. Veámoslo con un ejemplo:



```
Pattern p=Pattern.compile("[01]+");
Matcher m=p.matcher("00001010");
if (m.matches()) System.out.println("Si, contiene el patrón");
else System.out.println("No, no contiene el patrón");
```

En el ejemplo, el método estático **compile** de la clase **Pattern** permite crear un patrón, dicho método compila la expresión regular pasada por parámetro y genera una instancia de **Pattern** (**p** en el ejemplo). El patrón **p** podrá ser usado múltiples veces para verificar si una cadena coincide o no con el patrón, dicha comprobación se hace invocando el método **matcher**, el cual combina el patrón con la cadena de entrada y genera una instancia de la clase **Matcher** (**m** en el ejemplo). La clase **Matcher** contiene el resultado de la comprobación y ofrece varios métodos para analizar la forma en la que la cadena ha encajado con un patrón:

- ✓ **m.matches()**. Devolverá **true** si toda la cadena (de principio a fin) encaja con el patrón o **false** en caso contrario.
- ✓ **m.lookingAt()**. Devolverá **true** si el patrón se ha encontrado al principio de la cadena. A diferencia del método **matches()**, la cadena podrá contener al final caracteres adicionales a los indicados por el patrón, sin que ello suponga un problema.
- ✓ **m.find()**. Devolverá **true** si el patrón existe en algún lugar la cadena (no necesariamente toda la cadena debe coincidir con el patrón) y **false** en caso contrario, pudiendo tener más de una coincidencia. Para obtener la posición exacta donde se ha producido la coincidencia con el patrón podemos usar los métodos **m.start()** y **m.end()**, para saber la posición inicial y final donde se ha encontrado. Una segunda invocación del método **find()** irá a la segunda coincidencia (si existe), y así sucesivamente. Podemos reiniciar el método **find()**, para que vuelva a comenzar por la primera coincidencia, invocando el método **m.reset()**.

Veamos algunas construcciones adicionales que pueden ayudarnos a especificar expresiones regulares más complejas:

- ✓ **"[^abc]"**. El símbolo **"^"**, cuando se pone justo detrás del corchete de apertura, significa "negación". La expresión regular admitirá cualquier símbolo diferente a los puestos entre corchetes. En este caso, cualquier símbolo diferente de **"a"**, **"b"** o **"c"**.
- ✓ **"^[01]+\$"**. Cuando el símbolo **"^"** aparece al comienzo de la expresión regular, permite indicar comienzo de línea o de entrada. El símbolo **"\$"** permite indicar fin de línea o fin de entrada. Usándolos podemos verificar que una línea completa (de principio a fin) encaje con la expresión regular, es muy útil cuando se trabaja en **modo multilinea** y con el método **find()**.
- ✓ **"."**. El punto simboliza cualquier carácter.
- ✓ **"\\d"**. Un dígito numérico. Equivale a **"[0-9]"**.
- ✓ **"\\D"**. Cualquier cosa excepto un dígito numérico. Equivale a **"[^0-9]"**.
- ✓ **"\\s"**. Un espacio en blanco (incluye tabulaciones, saltos de línea y otras formas de espacio).
- ✓ **"\\S"**. Cualquier cosa excepto un espacio en blanco.
- ✓ **"\\w"**. Cualquier carácter que podrías encontrar en una palabra. Equivale a **"[a-zA-Z_0-9]"**.



Autoevaluación

¿En cuáles de las siguientes opciones se cumple el patrón **"A.\\d+"**?

- ☐ **"GA-99"** si utilizamos el método **find**.
- ☐ **"GAX99"** si utilizamos el método **lookingAt**.
- ☐ **"AX99-"** si utilizamos el método **matches**.
- ☐ **"A99"** si utilizamos el método **matches**.

Mostrar Información

2.2.2.- Expresiones regulares (III).

¿Te resultan difíciles las expresiones regulares? Al principio siempre lo son, pero no te preocupes. Hasta ahora has visto como las expresiones regulares permiten verificar datos de entrada, permitiendo comprobar si un dato indicado sigue el formato esperado: que un DNI tenga el formato esperado, que un email sea un email y no otra cosa, etc. Pero ahora vamos a dar una vuelta de tuerca adicional.



Los paréntesis, de los cuales no hemos hablado hasta ahora, tienen un significado especial, permiten indicar repeticiones para un conjunto de símbolos, por ejemplo: "(#**[01]**){**2,3**}". En el ejemplo anterior, la expresión "**# [01]**" admitiría cadenas como "**#0**" o "**#1**", pero al ponerlo entre paréntesis e indicar los contadores de repetición, lo que estamos diciendo es que la misma secuencia se tiene que repetir entre dos y tres veces, con lo que las cadenas que admitiría serían del estilo a: "**#0#1**" o "**#0#1#0**".

Pero los paréntesis tienen una función adicional, y es la de permitir definir grupos. Un grupo comienza cuando se abre un paréntesis y termina cuando se cierra el paréntesis. Los grupos permiten acceder de forma cómoda a las diferentes partes de una cadena cuando esta coincide con una expresión regular. Lo mejor es verlo con un ejemplo (seguro que te resultará familiar):

```
Pattern p=Pattern.compile("([XY]?)([0-9]{1,9})([A-Za-z])");
Matcher m=p.matcher("X123456789Z Y00110011M 999999T");
while (m.find())
{
    System.out.println("Letra inicial (opcional):"+m.group(1));
    System.out.println("Número:"+m.group(2));
    System.out.println("Letra NIF:"+m.group(3));
}
```

Usando los grupos, podemos obtener por separado el texto contenido en cada uno de los grupos. En el ejemplo anterior, en el patrón hay tres grupos: uno para la letra inicial (grupo 1), otro para el número del DNI o NIE (grupo 2), y otro para la letra final o letra NIE (grupo 3). Al ponerlo en grupos, usando el método **group()**, podemos extraer la información de cada grupo y usarla a nuestra conveniencia.

Ten en cuenta que el primer grupo es el 1, y no el 0. Si pones **m.group(0)** obtendrás una cadena con toda la ocurrencia o coincidencia del patrón en la cadena, es decir, obtendrás la secuencia entera de símbolos que coincide con el patrón.

En el ejemplo anterior se usa el método **find**, éste buscará una a una, cada una de las ocurrencias del patrón en la cadena. Cada vez que se invoca, busca la siguiente ocurrencia del patrón y devolverá **true** si ha encontrado una ocurrencia. Si no encuentra en una iteración ninguna ocurrencia es porque no existen más, y retornará **false**, saliendo del bucle. Esta construcción **while** es muy típica para este tipo de métodos y para las iteraciones, que veremos más adelante.

Lo último importante de las expresiones regulares que debes conocer son las secuencias de escape. Cuando en una expresión regular necesitamos especificar que lo que tiene que haber en la cadena es un paréntesis, una llave, o un corchete, tenemos que usar una secuencia de escape, dado que esos símbolos tienen un significado especial en los patrones. Para ello, simplemente antepondremos **\"** al símbolo. Por ejemplo, **\"(\"** significará que debe haber un paréntesis en la cadena y se omitirá el significado especial del paréntesis. Lo mismo ocurre con **\"[\"**, **\"\\\"**, **\"\\\"**, etc. Lo mismo para el significado especial del punto, éste, tiene un significado especial (¿Lo recuerdas del apartado anterior?) salvo que se ponga **\\.**, que pasará a significar "un punto" en vez de "cualquier carácter". **La excepción son las comillas, que se pondrían con una sola barra: **\"****

Para saber más

Con lo estudiado hasta ahora, ya puedes utilizar las expresiones regulares y sacarles partido casi que al máximo. Pero las expresiones regulares son un campo muy amplio, por lo que es muy aconsejable que amplíes tus conocimientos con el siguiente enlace:

[Tutorial de expresiones regulares en Java \(en inglés\).](#)

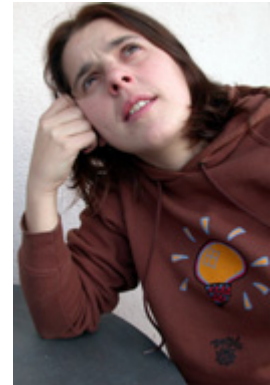
3.- Creación de arrays.

Caso práctico

Después del susto inicial, al recibir el archivo con el formato de pedido, **Ana** se puso a pensar. Vio que lo más adecuado era procesar el archivo de pedido línea a línea, viendo con que corresponde cada línea a través de expresiones regulares.

Para identificar si hay un inicio o fin de sección dentro del pedido, así como el inicio o fin del listado de artículos, ha decidido usar la siguiente expresión regular: `"^##[]*(FIN){0,1}[]*(PEDIDO|ARTICULOS)[]*##$"`.

Y para identificar cada dato del pedido (nombre, dirección, etc.) va a utilizar la siguiente expresión regular: `"^(.+):.*\\{(.*)\\}$"`, que al usar grupos le permitirá separar el nombre del campo (dirección por ejemplo) de su valor (dirección concreta). La expresión regular le ha costado mucho trabajo y ha tenido que pedir ayuda, pero después, una vez que la ha conseguido, se ha dado cuenta de que ha sido relativamente fácil.

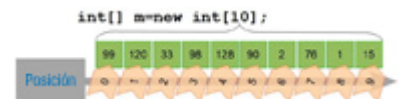


De momento solo está haciendo pruebas, pero le están saliendo muy bien, mira como lo lleva:

[Procesado del pedido, primera parte. \(3 KB\)](#)

Ya tiene parte del trabajo hecho, al programa que ha realizado se le puede pasar el archivo de pedido en texto plano, con codificación **UTF-8** y ya es capaz de procesarlo casi entero. Pero la parte de artículos todavía no la tiene terminada, básicamente tiene dudas sobre donde almacenar la información procesada, en especial, la lista de artículos.

¿Y dónde almacenarías tú la lista de artículos del pedido? Una de las soluciones es usar arrays, puede que sea justo lo que necesita Ana, pero puede que no. Todos los lenguajes de programación permiten el uso de arrays, veamos como son en Java.



Los arrays permiten almacenar una colección de objetos o datos del mismo tipo. Son muy útiles y su utilización es muy simple:

- ✓ **Declaración del array.** La declaración de un array consiste en decir "esto es un array" y sigue la siguiente estructura: "tipo[] nombre;". El tipo será un tipo de variable o una clase ya existente, de la cual se quieran almacenar varias unidades.
- ✓ **Creación del array.** La creación de un array consiste en decir el tamaño que tendrá el array, es decir, el número de elementos que contendrá, y se pone de la siguiente forma: "nombre=new tipo[dimensión]", donde dimensión es un número entero positivo que indicará el tamaño del array. Una vez creado el array este no podrá cambiar de tamaño.

Veamos un ejemplo de su uso:

```
int[] n; // Declaración del array.
n = new int[10]; // Creación del array reservando para el un espacio en memoria.
int[] m=new int[10]; // Declaración y creación en un mismo lugar.
```

Una vez hecho esto, ya podemos almacenar valores en cada una de las posiciones del array, usando corchetes e indicando en su interior la posición en la que queremos leer o escribir, teniendo en cuenta que la primera posición es la cero y la última el tamaño del array menos uno. En el ejemplo anterior, la primera posición sería la 0 y la última sería la 9.



Autoevaluación

¿Cuáles de las siguientes opciones permitiría almacenar más de 50 números decimales?

- ☐ `int[] numeros; numeros=new int[51];`
- ☐ `int[] numeros; numeros=new float[52];`
- ☐ `double[] numeros; numeros=new double[53];`
- ☐ `float[] numeros=new float[54];`

Mostrar Información

3.1.- Uso de arrays unidimensionales.

Ya sabes declarar y crear de arrays, pero, ¿cómo y cuando se usan? Pues existen tres ámbitos principalmente donde se pueden usar, y los tres son muy importantes: modificación de una posición del array, acceso a una posición del array y paso de parámetros.

La modificación de una posición del array se realiza con una simple asignación. Simplemente se especifica entre corchetes la posición a modificar después del nombre del array. Veámoslo con un simple ejemplo:

```
int[] Numeros=new int[3]; // Array de 3 números (posiciones del 0 al 2).
Numeros[0]=99; // Primera posición del array.
Numeros[1]=120; // Segunda posición del array.
Numeros[2]=33; // Tercera y última posición del array.
```



El acceso a un valor ya existente dentro de una posición del array se consigue de forma similar, simplemente poniendo el nombre del array y la posición a la cual se quiere acceder entre corchetes:

```
int suma=Numeros[0] + Numeros[1] + Numeros[2];
```

Para nuestra comodidad, los arrays, como objetos que son en Java, disponen de una propiedad pública muy útil. La propiedad `length` nos permite saber el tamaño de cualquier array, lo cual es especialmente útil en métodos que tienen como argumento un array.

```
System.out.println("Longitud del array: "+Numeros.length);
```

El tercer uso principal de los arrays, como se dijo antes, es en el paso de parámetros. Para pasar como argumento un array a una función o método, esta debe tener en su definición un parámetro declarado como array. Esto es simplemente que uno de los parámetros de la función sea un array. Veamos un ejemplo:

```
int sumaarray (int[] j) {
    int suma=0;
    for (int i=0; i<j.length;i++)
        suma=suma+j[i];
    return suma;
}
```

En el método anterior se pasa como argumento un array numérico, sobre el cual se calcula la suma de todos los números que contiene. Es un uso típico de los arrays, fíjate que especificar que un argumento es un array es igual que declarar un array, sin la creación del mismo. Para pasar como argumento un array a una función, simplemente se pone el nombre del array:

```
int suma=sumaarray (Numeros);
```

En Java las variables se pasan por copia a los métodos, esto quiere decir que cuando se pasa una variable a un método, y se realiza una modificación de su valor en dicho método, el valor de la variable en el método desde el que se ha realizado la invocación no se modifica. Pero cuidado, eso no pasa con los arrays. Cuando dicha modificación se realiza en un array, es decir, se cambia el valor de uno de los elementos del array, si que cambia su valor de forma definitiva. Veamos un ejemplo que ilustra ambas cosas:

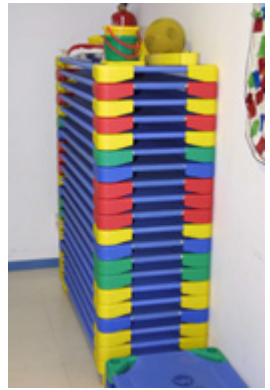
```
public static void main(String[] args) {  
    int j=0; int[] i=new int(1); i[0]=0;  
    modificaArray(j,i);  
    System.out.println(j+"-"+i[0]); /* Mostrará por pantalla "0-1", puesto que el  
        modificado en la función, y aunque la variable j también se modifica, se  
        dejando el original intacto */  
}  
  
int modificaArray(int j, int[] i); {  
    j++; i[0]++; /* Modificación de los valores de la variable, solo afectará a  
}
```



3.2.- Inicialización.

Rellenar un array, para su primera utilización, es una tarea muy habitual que puede ser rápidamente simplificada. Vamos a explorar dos sistemas que nos van a permitir inicializar un array de forma cómoda y rápida.

En primer lugar, una forma habitual de crear un array y rellenarlo es simplemente a través de un método que lleve a cabo la creación y el rellenado del array. Esto es sencillo desde que podemos hacer que un método retorne un array simplemente indicando en la declaración que el valor retornado es tipo[], donde tipo de dato primitivo (**int**, **short**, **float**,...) o una clase existente (**String** por ejemplo). Veamos un ejemplo:



```
static int[] ArrayConNumerosConsecutivos (int totalNumeros) {
    int[] r=new int[totalNumeros];
    for (int i=0;i<totalNumeros;i++) r[i]=i;
    return r;
}
```

En el ejemplo anterior se crea un array con una serie de números consecutivos, empezando por el cero, ¿sencillo no? Este uso suele ahorrar bastantes líneas de código en tareas repetitivas. Otra forma de inicializar los arrays, cuando el número de elementos es fijo y sabido a priori, es indicando entre llaves el listado de valores que tiene el array. En el siguiente ejemplo puedes ver la inicialización de un array de tres números, y la inicialización de un array con tres cadenas de texto:

```
int[] array = {10, 20, 30};
String[] diassemmana= {"Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"}
```

Pero cuidado, la inicialización solo se puede usar en ciertos casos. La inicialización anterior funciona cuando se trata de un tipo de dato primitivo (**int**, **short**, **float**, **double**, etc.) o un **String**, y algunos pocos casos más, pero no funcionará para cualquier objeto.

Cuando se trata de un array de objetos, la inicialización del mismo es un poco más liosa, dado que el valor inicial de los elementos del array de objetos será **null**, o lo que es lo mismo, crear un array de objetos no significa que se han creado las instancias de los objetos. Hay que crear, para cada posición del array, el objeto del tipo correspondiente con el operador **new**. Veamos un ejemplo con la clase **StringBuilder**. En el siguiente ejemplo solo aparecerá **null** por pantalla:

```
StringBuilder[] j=new StringBuilderStringBuilder[10];
for (int i=0; i<j.length;i++) System.out.println("Valor" +i + "=" +j[i]); // Imprimirá null pa
```

Para solucionar este problema podemos optar por lo siguiente, crear para cada posición del array una instancia del objeto:

```
StringBuilder[] j=new StringBuilder[10];
for (int i=0; i<j.length;i++) j[i]=new StringBuilder("cadena "+i);
```

Reflexiona

Para acceder a una propiedad o a un método cuando los elementos del array son objetos, puedes usar la notación de punto detrás de los corchetes, por ejemplo: **diassemmana[0].length**. Fijate bien en el

array **diassemana** anterior y piensa en lo que se mostraría por pantalla con el siguiente código:

```
System.out.println(diassemana[0].substring(0,2));
```



Autoevaluación

¿Cuál es el valor de la posición 3 del siguiente array: `String[] m=new String[10]`?

- ☐ `null`.
- ☐ Una cadena vacía.
- ☐ Daría error y no se podría compilar.

4.- Arrays multidimensionales.

Caso práctico



Ana sigue dándole vueltas a como almacenar los diferentes datos del pedido, y sobre todo, sobre como procesar los artículos, dado que los artículos del pedido pueden ser más de uno. Ha pensado en crear primero una clase para almacenar los datos del pedido, llamada **Pedido**, donde cada dato del pedido sea un atributo de la clase. Pero claro, los artículos son más de uno y no puede saber cuantos atributos necesitará.

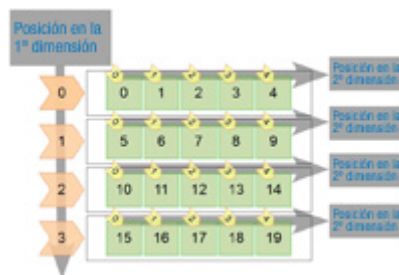
Para almacenar los artículos ha pensado en crear una clase llamada **Artículo** donde se meterían todos los datos de cada artículo, y después en la clase **Pedido** poner un array de artículos. Pero no lo tiene claro y ha decidido preguntar a su tutora, María.

¿Qué estructura de datos utilizarías para almacenar los **píxeles** de una imagen digital? Normalmente las imágenes son cuadradas así que una de las estructuras más adecuadas es la **matriz**. En la matriz cada valor podría ser el color de cada píxel. Pero, ¿qué es una matriz a nivel de programación? Pues es un array con dos dimensiones, o lo que es lo mismo, un array cuyos elementos son arrays de números.

Los arrays multidimensionales están en todos los lenguajes de programación actuales, y obviamente también en Java. La forma de crear un array de dos dimensiones en Java es la siguiente:

```
int[][] a2d=new int[4][5];
```

El código anterior creará un array de dos dimensiones, o lo que es lo mismo, creará un array que contendrá 4 arrays de 5 números cada uno. Veámoslo con un ejemplo gráfico:



Al igual que con los arrays de una sola dimensión, los arrays multidimensionales deben declararse y crearse. Podremos hacer arrays multidimensionales de todas las dimensiones que queramos y de cualquier tipo. En ellos todos los elementos del array serán del mismo tipo, como en el caso de los arrays de una sola dimensión. La declaración comenzará especificando el tipo o la clase de los elementos que forman el array, después pondremos tantos corchetes como dimensiones tenga el array y por último el nombre del array, por ejemplo:

```
int [][][] arrayde3dim;
```

La creación es igualmente usando el operador **new**, seguido del tipo y los corchetes, en los cuales se especifica el tamaño de cada dimensión:

```
arrayde3dim=new int[2][3][4];
```

Todo esto, como ya has visto en un ejemplo anterior, se puede escribir en una única sentencia.



Autoevaluación

Completa con los números que faltan:

```
int[][][] k=new int[10][11][12];
```

El array anterior es de dimensiones, y tiene un total de números enteros.

4.1.- Uso de arrays multidimensionales.

¿Y en que se diferencia el uso de un array multidimensional con respecto a uno de una única dimensión? Pues en muy poco la verdad. Continuaremos con el ejemplo del apartado anterior:

```
int[][] a2d=new int[4][5];
```

Para acceder a cada uno de los elementos del array anterior, habrá que indicar su posición en las dos dimensiones, teniendo en cuenta que los índices de cada una de las dimensiones empieza a numerarse en 0 y que la última posición es el tamaño de la dimensión menos 1.

Puedes asignar un valor a una posición concreta dentro del array, indicando la posición en cada una de las dimensiones entre corchetes:

```
a2d[0][0]=3;
```

Y como es de imaginar, puedes usar un valor almacenado en una posición del array multidimensional simplemente poniendo el nombre del array y los índices del elemento al que deseas acceder entre corchetes, para cada una de las dimensiones del array. Por ejemplo:

```
int suma=a2d[0][0]+a2d[0][1]+a2d[0][2]+a2d[0][3]+a2d[0][4];
```

Como imaginarás, los arrays multidimensionales pueden también ser pasados como parámetros a los métodos, simplemente escribiendo la declaración del array en los argumentos del método, de forma similar a como se realizaba para arrays de una dimensión. Por ejemplo:

```
static int sumaarray2d(int[][] a2d) {
    int suma = 0;
    for (int i1 = 0; i1 < a2d.length; i1++)
        for (int i2 = 0; i2 < a2d[i1].length; i2++) suma += a2d[i1][i2];
    return suma;
}
```

Del código anterior, fíjate especialmente en el uso del atributo **length** (que nos permite obtener el tamaño de un array). Aplicado directamente sobre el array nos permite saber el tamaño de la primera dimensión (**a2d.length**). Como los arrays multidimensionales son arrays que tienen como elementos arrays (excepto el último nivel que ya será del tipo concreto almacenado), para saber el tamaño de una dimensión superior tenemos que poner el o los índices entre corchetes seguidos de **length** (**a2d[i1].length**).

Para saber al tamaño de una segunda dimensión (dentro de una función por ejemplo) hay que hacerlo así y puede resultar un poco engorroso, pero gracias a esto podemos tener arrays multidimensionales irregulares.

Una matriz es un ejemplo de array multidimensional regular, ¿por qué? Pues porque es un array que contiene arrays de números todos del mismo tamaño. Cuando esto no es así, es decir, cuando los arrays de la segunda dimensión son de diferente tamaño entre sí, se puede decir que es un array multidimensional irregular. En Java se puede crear un array irregular de forma relativamente fácil, veamos un ejemplo para dos dimensiones.

```
int[][] m=new int[4][];
0 0 1 2 3 m[0]=new int[4];
1 5 6 7 8 9 m[1]=new int[5];
2 10 11 12 m[2]=new int[3];
3 15 16 17 18 19 m[3]=new int[5];
```

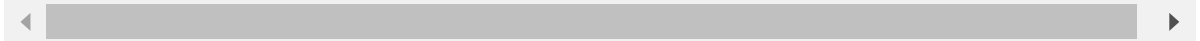
- ✓ **Declaramos y creamos el array pero sin especificar la segunda dimensión.** Lo que estamos haciendo en realidad es crear simplemente un array que contendrá arrays, sin decir como son de grandes los arrays de la siguiente dimensión: `int[][] irregular=new int[3][];`

- ✓ Después creamos cada uno de los arrays unidimensionales (del tamaño que queramos) y lo asignamos a la posición correspondiente del array anterior: `irregular[0]=new int[7]; irregular[1]=new int[15]; irregular[2]=new int[9];`

Recomendación

Cuando uses arrays irregulares, por seguridad, es conveniente que verifiques siempre que el array no sea **null** en segundas dimensiones, y que la longitud sea la esperada antes de acceder a los datos:

```
if (irregular[1]!=null && irregular[1].length>10) {...}
```



4.2.- Inicialización de arrays multidimensionales.

¿En qué se diferencia la inicialización de arrays unidimensionales de arrays multidimensionales? En muy poco. La inicialización de los arrays multidimensionales es igual que la de los arrays unidimensionales.

Para que una función retorne un array multidimensional, se hace igual que en arrays unidimensionales. Simplemente hay que poner el tipo de dato seguido del número de corchetes correspondiente, según el número de dimensiones del array. Eso claro, hay que ponerlo en la definición del método:



```
int[][] inicializarArray (int n, int m)
{
    int[][] ret=new int[n][m];
    for (int i=0;i<n;i++)
        for (int j=0;j<m;j++)
            ret[i][j]=n*m;
    return ret;
}
```

También se puede inicializar un array multidimensional usando las llaves, poniendo después de la declaración del array un símbolo de igual, y encerrado entre llaves los diferentes valores del array separados por comas, con la salvedad de que hay que poner unas llaves nuevas cada vez que haya que poner los datos de una nueva dimensión, lo mejor es verlo con un ejemplo:

```
int[][] a2d={{0,1,2},{3,4,5},{6,7,8},{9,10,11}};
int[][][] a3d={{0,1},{2,3}},{0,1},{2,3}};
```

El primer array anterior sería un array de 4 por 3 y el siguiente sería un array de 2x2x2. Como puedes observar esta notación a partir de 3 dimensiones ya es muy liosa y normalmente no se usa. Utilizando esta notación también podemos inicializar rápidamente arrays irregulares, simplemente poniendo separados por comas los elementos que tiene cada dimensión:

```
int[][] i2d={{0,1},{0,1,2},{0,1,2,3},{0,1,2,3,4},{0,1,2,3,4,5}};
int[][][] i3d={ { {0,1},{0,2} } , {0,1,3} } , {0,3,4},{0,1,5} } };
```

Es posible que en algunos libros y en Internet se refieran a los arrays usando otra terminología. A los arrays unidimensionales es común llamarlos también **arreglos** o **vectores**, a los arrays de dos dimensiones es común llamarlos directamente **matrices**, y a los arrays de más de dos dimensiones es posible que los veas escritos como **matrices multidimensionales**. Sea como sea, lo más normal en la jerga informática es llamarlos arrays, término que procede del inglés.



Autoevaluación

Dado el siguiente array: `int[][][] i3d={{0,1},{0,2}},{0,1,3},{0,3,4},{0,1,5}};`
¿Cuál es el valor de la posición `[1][1][2]`?

- ☐ 3
☐ 4
☐

5

☐ Ninguno de los anteriores.

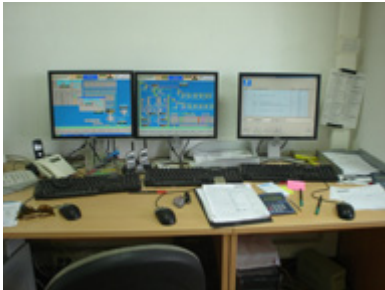
Para saber más

Las matrices tienen asociadas un amplio abanico de operaciones (transposición, suma, producto, etc.). En la siguiente página tienes ejemplos de como realizar algunas de estas operaciones, usando por supuesto arrays:

[Operaciones básicas con matrices.](#)

5.- Clases y métodos genéricos (I).

Caso práctico



María se acerca a la mesa de **Ana**, quiere saber cómo lo lleva:

-¿Qué tal? ¿Cómo vas con la tarea? -pregunta María.

-Bien, creo. Mi programita ya sabe procesar el archivo de pedido y he creado un par de clases para almacenar los datos de forma estructurada para luego hacer el volcado XML, pero no se como almacenar los artículos del pedido, porque son varios -comenta Ana.

-Pero, ¿cuál es el problema? Eso es algo sencillo.

-Pues que tengo que crear un array para guardar con los artículos del pedido, y no se como averiguar el número de artículos antes de empezar a procesarlos. Es necesario saber el número de artículos para crear el array del tamaño adecuado.

-Pues en vez de utilizar un array, podrías utilizar una lista.

¿Sabes porqué se suele aprender el uso de los genéricos? Pues porque se necesita para usar las listas, aunque realmente los genéricos son una herramienta muy potente y que nos puede ahorrar tareas de programación repetitivas.

Las clases y los métodos genéricos son un recurso de programación disponible en muchos lenguajes de programación. Su objetivo es claro: facilitar la reutilización del software, creando métodos y clases que puedan trabajar con diferentes tipos de objetos, evitando incómodas y engorrosas conversiones de tipos. Su inicio se remonta a las plantillas (templates) de C++, un gran avance en el mundo de programación sin duda. En lenguajes de más alto nivel como Java o C# se ha transformado en lo que se denomina "genéricos". Veamos un ejemplo sencillo de como transformar un método normal en genérico:



Versiones genérica y no genérica del método de compararTamaño.

Versión no genérica	Versión genérica del método
<pre>public class util { public static int compararTamaño(Object[] a, Object[] b) { return a.length- b.length; } }</pre>	<pre>public class util { public static <T> int compararTamaño (T[] a, T[] b) { return a.length- b.length; } }</pre>

Los dos métodos anteriores tienen un claro objetivo: permitir comprobar si un array es mayor que otro. Retornarán 0 si ambos arrays son iguales, un número mayor de cero si el array **b** es mayor, y un número menor de cero si el array **a** es mayor, pero uno es genérico y el otro no. La versión genérica del módulo incluye la expresión "<T>", justo antes del tipo retornado por el método. "<T>" es la definición de una **variable o parámetro formal de tipo** de la clase o método genérico, al que podemos llamar simplemente **parámetro de tipo o parámetro genérico**. Este parámetro genérico (**T**) se puede usar a lo largo de todo el método o clase, dependiendo del ámbito de definición, y hará referencia a cualquier clase con la que nuestro algoritmo tenga que trabajar. Como veremos más adelante, puede haber más de un parámetro genérico.

Utilizar genéricos tiene claras ventajas. Para invocar un método genérico, sólo hay que realizar una **invocación de tipo genérico**, olvidándonos de las conversiones de tipo. Esto consiste en indicar qué clases o interfaces concretas se utilizarán en lugar de cada parámetro genérico ("<T>"), para después, pasándole los argumentos correspondientes, ejecutar el algoritmo. Cada clase o interfaz concreta, la podemos denominar **tipo** o **tipo base** y se da por sentado que **los argumentos pasados al método genérico serán también de dicho tipo base**.

Supongamos que el tipo base es **Integer**, pues para realizar la invocación del método genérico anterior basta con indicar el tipo, entre los símbolos de menor que y mayor que ("**<Integer>**"), justo antes del nombre del método.

Invocaciones de las versiones genéricas y no genéricas de un método.

Invocación del método no genérico.	Invocación del método genérico.
<pre>Integer []a={0,1,2,3,4}; Integer []b={0,1,2,3,4,5}; util.compararTamano ((Object[])a, (Object[])b);</pre>	<pre>Integer []a={0,1,2,3,4}; Integer []b={0,1,2,3,4,5}; util.<Integer>compararTamano (a, b);</pre>

5.1.- Clases y métodos genéricos (II).

¿Crees qué el código es más legible al utilizar genéricos o que se complica? La verdad es que al principio cuesta, pero después, el código se entiende mejor que si se empieza a insertar conversiones de tipo.

Las clases genéricas son equivalentes a los métodos genéricos pero a nivel de clase, permiten definir un parámetro de tipo o genérico que se podrá usar a lo largo de toda la clase, facilitando así crear clases genéricas que son capaces de trabajar con diferentes tipos de datos base. Para crear una clase genérica se especifican los parámetros de tipo al lado del nombre de la clase:



```
public class Util<T> {
    T t1;
    public void invertir(T[] array) {
        for (int i = 0; i < array.length / 2; i++) {
            t1 = array[i];
            array[i] = array[array.length - i - 1];
            array[array.length - i - 1] = t1;
        }
    }
}
```

En el ejemplo anterior, la clase **Util** contiene el método **invertir** cuya función es invertir el orden de los elementos de cualquier array, sea del tipo que sea. Para usar esa clase genérica hay que crear un objeto o instancia de dicha clase especificando el tipo base entre los símbolos menor que ("**<**") y mayor que ("**>**"), justo detrás del nombre de la clase. Veamos un ejemplo:

```
Integer[] numeros={0,1,2,3,4,5,6,7,8,9};
Util<Integer> u= new Util<Integer>();
u.invertir(numeros);
for (int i=0;i<numeros.length;i++) System.out.println(numeros[i]);
```

Como puedes observar, el uso de genéricos es sencillo, tanto a nivel de clase como a nivel de método. Simplemente, a la hora de crear una instancia de una clase genérica, hay que especificar el tipo, tanto en la definición (**Util <integer> u**) como en la creación (**new Util<Integer>()**).

Los genéricos los vamos a usar ampliamente a partir de ahora, aplicados a un montón de clases genéricas que tiene Java y que son de gran utilidad, por lo que es conveniente que aprendas bien a usar una clase genérica.

Los parámetros de tipo de las clases genéricas solo pueden ser clases, no pueden ser jamás tipos de datos primitivos como **int**, **short**, **double**, etc. En su lugar, debemos usar sus clases envoltorio **Integer**, **Short**, **Double**, etc.

Debes conocer

Todavía hay un montón de cosas más sobre los métodos y las clases genéricas que deberías saber. En la siguiente animación se muestran algunos usos interesantes de los genéricos:



Adobe Flash Player ya no está disponible

[Resumen textual alternativo](#)



Autoevaluación

Dada la siguiente clase, donde el código del método prueba carece de importancia, ¿podrías decir cuál de las siguientes invocaciones es la correcta?

```
public class Util {  
    public static <T> int prueba (T t) { ... }  
};
```

- ☐ `Util.<int>prueba(4);`
- ☐ `Util.<Integer>prueba(new Integer(4));`
- ☐ `Util u=new Util(); u.<int>prueba(4);`

6.- Introducción a las colecciones (I).

Caso práctico

A **Ana** las listas siempre se le han atragantado, por eso no las usa. Después de darle muchas vueltas, ha pensado que no le queda más remedio y que tendrá que usarlas para almacenar los artículos del pedido. Además, ha concluido que es la mejor forma de gestionar un grupo de objetos, aunque sean del mismo tipo.

No sabe si lo más adecuado es usar una lista u otro tipo de colección, así que ha decidido revisar todos los tipos de colecciones disponibles en Java, para ver cuál se adecua mejor a sus necesidades.



¿Qué consideras una colección? Pues seguramente al pensar en el término se te viene a la cabeza una colección de libros o algo parecido, y la idea no va muy desencaminada. **Una colección a nivel de software es un grupo de elementos almacenados de forma conjunta en una misma estructura.** Eso son las colecciones.

Las colecciones definen un conjunto de interfaces, clases genéricas y algoritmos que permiten manejar grupos de objetos, todo ello enfocado a potenciar la reusabilidad del software y facilitar las tareas de programación. Te parecerá increíble el tiempo que se ahorra empleando colecciones y cómo se reduce la complejidad del software usándolas adecuadamente. Las colecciones permiten almacenar y manipular grupos de objetos que, a priori, están relacionados entre sí (aunque no es obligatorio que estén relacionados, lo lógico es que si se almacenan juntos es porque tienen alguna relación entre sí), pudiendo trabajar con cualquier tipo de objeto (de ahí que se empleen los genéricos en las colecciones).



Además las colecciones permiten realizar algunas operaciones útiles sobre los elementos almacenados, tales como búsqueda u ordenación. En algunos casos es necesario que los objetos almacenados cumplan algunas condiciones (que implementen algunas interfaces), para poder ser hacer uso de estos algoritmos.

Las colecciones son en general elementos de programación que están disponibles en muchos lenguajes de programación. En algunos lenguajes de programación su uso es algo más complejo (como es el caso de C++), pero en Java su uso es bastante sencillo, es algo que descubrirás a lo largo de lo que queda de tema.

Las colecciones en Java parten de una serie de interfaces básicas. Cada interfaz define un **modelo** de colección y las operaciones que se pueden llevar a cabo sobre los datos almacenados, por lo que es necesario conocerlas. La interfaz inicial, a través de la cual se han construido el resto de colecciones, es la interfaz **java.util.Collection**, que define las operaciones comunes a todas las colecciones derivadas. A continuación se muestran las operaciones más importantes definidas por esta interfaz, ten en cuenta que **Collection** es una interfaz genérica donde "<E>" es el parámetro de tipo (podría ser cualquier clase):

- ✓ Método `int size()`: retorna el número de elementos de la colección.
- ✓ Método `boolean isEmpty()`: retornará verdadero si la colección está vacía.
- ✓ Método `boolean contains (Object element)`: retornará verdadero si la colección tiene el elemento pasado como parámetro.
- ✓ Método `boolean add(E element)`: permitirá añadir elementos a la colección.
- ✓ Método `boolean remove (Object element)`: permitirá eliminar elementos de la colección.
- ✓ Método `Iterator<E> iterator()`: permitirá crear un iterador para recorrer los elementos de la colección. Esto se ve más adelante, no te preocupes.
- ✓ Método `Object[] toArray()`: permite pasar la colección a un array de objetos tipo `Object`.
- ✓ Método `containsAll(Collection<?> c)`: permite comprobar si una colección contiene los elementos existentes en otra colección, si es así, retorna verdadero.
- ✓ Método `addAll (Collection<? extends E> c)`: permite añadir todos los elementos de una colección a otra colección, siempre que sean del mismo tipo (o deriven del mismo tipo base).
- ✓ Método `boolean removeAll(Collection<?> c)`: si los elementos de la colección pasada como parámetro están en nuestra colección, se eliminan, el resto se quedan.
- ✓ Método `boolean retainAll(Collection<?> c)`: si los elementos de la colección pasada como parámetro están en nuestra colección, se dejan, el resto se eliminan.

✔ Método `void clear()`: vaciar la colección.

Más adelante veremos como se usan estos métodos, será cuando veamos las implementaciones (clases genéricas que implementan alguna de las interfaces derivadas de la interfaz **Collection**).

7.- Conjuntos (I).

Caso práctico

Ana se toma un descanso, se levanta y en el pasillo se encuentra con Juan, con el que entabla una conversación bastante amena. Una cosa lleva a otra y al final, Ana saca el tema que más le preocupa:

—¿Cuántos tipos de colecciones hay? ¿Tu te los sabes? —pregunta Ana.

—¿Yo? ¡Que va! Normalmente consulto la documentación cuando los voy a usar, como todo el mundo. Lo que sí creo recordar es que había cuatro tipos básicos: los conjuntos, las listas, las colas y alguno más que no recuerdo. ¡Ah sí!, los mapas, aunque creo que no se consideraban un tipo de colección. ¿Por qué lo preguntas?

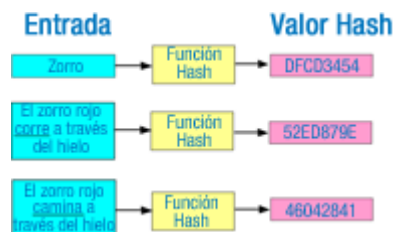
—Pues porque tengo que usar uno y no sé cuál.



¿Con qué relacionarías los conjuntos? Seguro que con las matemáticas. Los conjuntos son un tipo de colección que **no admite duplicados**, derivados del concepto matemático de conjunto.

La interfaz `java.util.Set` define cómo deben ser los conjuntos, y extiende la interfaz `Collection`, aunque no añade ninguna operación nueva. Las implementaciones (clases genéricas que implementan la interfaz `Set`) más usadas son las siguientes:

- ✓ `java.util.HashSet`. Conjunto que almacena los objetos usando **tablas hash**, lo cual acelera enormemente el acceso a los objetos almacenados. Inconvenientes: necesitan bastante memoria y **no almacenan los objetos de forma ordenada** (al contrario pueden aparecer completamente desordenados).
- ✓ `java.util.LinkedHashSet`. Conjunto que almacena objetos combinando tablas hash, para un acceso rápido a los datos, y **listas enlazadas** para conservar el orden. **El orden de almacenamiento es el de inserción**, por lo que se puede decir que es una estructura ordenada a medias. Inconvenientes: necesitan bastante memoria y es algo más lenta que `HashSet`.
- ✓ `java.util.TreeSet`. Conjunto que almacena los objetos usando unas estructuras conocidas como árboles rojo-negro. Son más lentas que los dos tipos anteriores, pero tienen una gran ventaja: **los datos almacenados se ordenan por valor**. Es decir, que aunque se inserten los elementos de forma desordenada, internamente se ordenan dependiendo del valor de cada uno.



Poco a poco, iremos viendo que son las listas enlazadas y los árboles (no profundizaremos en los árboles rojo-negro, pero si veremos las estructuras tipo árbol en general). Veamos un ejemplo de uso básico de la estructura **HashSet** y después, profundizaremos en los **LinkedHashSet** y los **TreeSet**.

Para crear un conjunto, simplemente creamos el **HashSet** indicando el tipo de objeto que va a almacenar, dado que es una clase genérica que puede trabajar con cualquier tipo de dato debemos crearlo como sigue (no olvides hacer la importación de `java.util.HashSet` primero):

```
HashSet<Integer> conjunto=new HashSet<Integer>();
```

Después podremos ir almacenando objetos dentro del conjunto usando el método **add** (definido por la interfaz `Set`). Los objetos que se pueden insertar serán siempre del tipo especificado al crear el conjunto:

```
Integer n=new Integer(10);
if (!conjunto.add(n)) System.out.println("Número ya en la lista.");
```

Si el elemento ya está en el conjunto, el método **add** retornará **false** indicando que no se pueden insertar duplicados. Si todo va bien, retornará **true**.



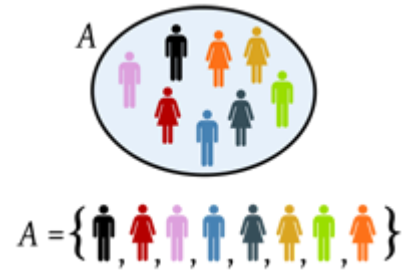
Autoevaluación

¿Cuál de las siguientes estructuras ordena automáticamente los elementos según su valor?

- ☐ **HashSet.**
- ☐ **LinkedHashSet.**
- ☐ **TreeSet.**

7.1.- Conjuntos (II).

Y ahora te preguntará, ¿cómo accedo a los elementos almacenados en un conjunto? Para obtener los elementos almacenados en un conjunto hay que usar iteradores, que permiten obtener los elementos del conjunto uno a uno de forma secuencial (no hay otra forma de acceder a los elementos de un conjunto, es su inconveniente). Los iteradores se ven en mayor profundidad más adelante, de momento, vamos a usar iteradores de forma transparente, a través de una estructura **for** especial, denominada bucle "for-each" o bucle "para cada". En el siguiente código se usa un bucle for-each, en él la variable **i** va tomando todos los valores almacenados en el conjunto hasta que llega al último:



```
for (Integer i: conjunto) {
    System.out.println("Elemento almacenado:"+i);
}
```

Como ves la estructura for-each es muy sencilla: la palabra **for** seguida de "(tipo variable:colección)" y el cuerpo del bucle; tipo es el tipo del objeto sobre el que se ha creado la colección, variable pues es la variable donde se almacenará cada elemento de la colección y colección pues la colección en sí. Los bucles for-each se pueden usar para todas las colecciones.

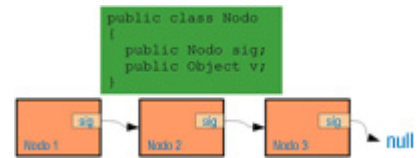
Ejercicio resuelto

Realiza un pequeño programita que pregunte al usuario 5 números diferentes (almacenándolos en un **HashSet**), y que después calcule la suma de los mismos (usando un bucle for-each).

7.2.- Conjuntos (III).

¿En qué se diferencian las estructuras **LinkedHashSet** y **TreeSet** de la estructura **HashSet**? Ya se comentó antes, y es básicamente en su funcionamiento interno.

La estructura **LinkedHashSet** es una estructura que internamente funciona como una lista enlazada, aunque usa también tablas hash para poder acceder rápidamente a los elementos. Una lista enlazada es una estructura similar a la representada en la imagen de la derecha, la cual está compuesta por nodos (elementos que forman la lista) que van enlazándose entre sí. Un nodo contiene dos cosas: el dato u objeto almacenado en la lista y el siguiente nodo de la lista. Si no hay siguiente nodo, se indica poniendo nulo (**null**) en la variable que contiene el siguiente nodo.

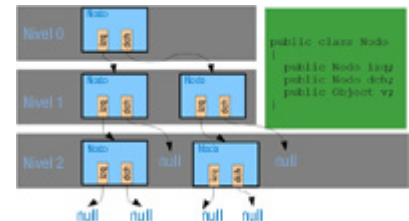


Las listas enlazadas tienen un montón de operaciones asociadas en las que no vamos a profundizar: eliminación de un nodo de la lista, inserción de un nodo al final, al principio o entre dos nodos, etc. Gracias a las colecciones podremos utilizar listas enlazadas sin tener que complicarnos en detalles de programación.

La estructura **TreeSet**, en cambio, utiliza internamente árboles. Los árboles son como las listas pero mucho más complejos. En vez de tener un único elemento siguiente, pueden tener dos o más elementos siguientes, formando estructuras organizadas y jerárquicas.

Los nodos se diferencian en dos tipos: nodos padre y nodos hijo; un nodo padre puede tener varios nodos hijo asociados (depende del tipo de árbol), dando lugar a una estructura que parece un árbol invertido (de ahí su nombre).

En la figura de la derecha se puede apreciar un árbol donde cada nodo puede tener dos hijos, denominados izquierdo (**izq**) y derecho (**dch**). Puesto que un nodo hijo puede también ser padre a su vez, los árboles se suelen visualizar para su estudio por niveles para entenderlos mejor, donde cada nivel contiene hijos de los nodos del nivel anterior, excepto el primer nivel (que no tiene padre).



Los árboles son estructuras complejas de manejar y que permiten operaciones muy sofisticadas. Los árboles usados en los **TreeSet**, los árboles rojo-negro, son árboles auto-ordenados, es decir, que al insertar un elemento, este queda ordenado por su valor de forma que al recorrer el árbol, pasando por todos los nodos, los elementos salen ordenados. El ejemplo mostrado en la imagen es simplemente un árbol binario, el más simple de todos.

Nuevamente, no se va a profundizar en las operaciones que se pueden realizar en un árbol a nivel interno (inserción de nodos, eliminación de nodos, búsqueda de un valor, etc.). Nos aprovecharemos de las colecciones para hacer uso de su potencial. En la siguiente tabla tienes un uso comparado de **TreeSet** y **LinkedHashSet**. Su creación es similar a como se hace con **HashSet**, simplemente sustituyendo el nombre de la clase **HashSet** por una de las otras. Ni **TreeSet**, ni **LinkedHashSet** admiten duplicados, y se usan los mismos métodos ya vistos antes, los existentes en la interfaz **Set** (que es la interfaz que implementan).

Ejemplos de utilización de los conjuntos **TreeSet** y **Link**

	Conjunto TreeSet .	
Ejemplo de uso	<pre>TreeSet <Integer> t; t=new TreeSet<Integer>(); t.add(new Integer(4)); t.add(new Integer(3)); t.add(new Integer(1)); t.add(new Integer(99)); for (Integer i:t) System.out.println(i);</pre>	<pre>LinkedHashS t=new Linke t.add(new I t.add(new I t.add(new I t.add(new I for (Intege</pre>

	Conjunto TreeSet.	
Resultado mostrado por pantalla	1 3 4 99 (el resultado sale ordenado por valor)	4 3 1 99 (los valores salen ord



Autoevaluación

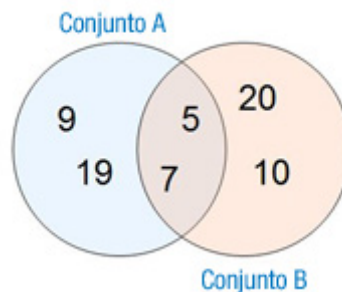
Un árbol cuyos nodos solo pueden tener un único nodo hijo, en realidad es una lista. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

7.3.- Conjuntos (IV).

¿Cómo podría copiar los elementos de un conjunto de uno a otro? ¿Hay que usar un bucle **for** y recorrer toda la lista para ello? ¡Qué va! Para facilitar esta tarea, los conjuntos, y las colecciones en general, facilitan un montón de operaciones para poder combinar los datos de varias colecciones. Ya se vieron en un apartado anterior, aquí simplemente vamos a poner un ejemplo de su uso.

Partimos del siguiente ejemplo, en el que hay dos colecciones de diferente tipo, cada una con 4 números enteros:



```
TreeSet<Integer> A= new TreeSet<Integer>();
A.add(9); A.add(19); A.add(5); A.add(7); // Elementos del conjunto A: 9, 19, 5 y 7
LinkedHashSet<Integer> B= new LinkedHashSet<Integer>();
B.add(10); B.add(20); B.add(5); B.add(7); // Elementos del conjunto B: 10, 20, 5 y 7
```

En el ejemplo anterior, el literal de número se convierte automáticamente a la clase envoltorio **Integer** sin tener que hacer nada, lo cual es una ventaja. Veamos las formas de combinar ambas colecciones:

Tipos de combinaciones.

Combinación.	Código.	Elementos finales del conjunto A.
Unión. Añadir todos los elementos del conjunto B en el conjunto A.	<code>A.addAll(B)</code>	<p>Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están: 5, 7, 9, 10, 19 y 20.</p>
Diferencia. Eliminar los elementos del conjunto B que puedan estar en el conjunto A.	<code>A.removeAll(B)</code>	<p>Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19.</p>
Intersección. Retiene los elementos comunes a ambos conjuntos.	<code>A.retainAll(B)</code>	<p>Todos los elementos del conjunto A, que también están en el conjunto B: 5 y 7.</p>

Combinación.	Código.	Elementos finales del conjunto A.
		

Recuerda, estas operaciones son comunes a todas las colecciones.

Para saber más

Puede que no recuerdes cómo era eso de los conjuntos, y dada la íntima relación de las colecciones con el álgebra de conjuntos, es recomendable que repases cómo era aquello, con el siguiente artículo de la Wikipedia.

[Álgebra de conjuntos.](#)



Autoevaluación

Tienes un `HashSet` llamado `vocales` que contiene los elementos "a", "e", "i", "o", "u", y otro, llamado `vocales_fuertes` con los elementos "a", "e" y "o". ¿De qué forma podríamos sacar una lista con las denominadas vocales débiles (que son aquellas que no son fuertes)?

- ☐ `vocales.retainAll(vocales_fuertes);`
- ☐ `vocales.removeAll(vocales_fuertes);`
- ☐ No es posible hacer esto con `HashSet`, solo se puede hacer con `TreeSet` o `LinkedHashSet`.

7.4.- Conjuntos (V).

Por defecto, los **TreeSet** ordenan sus elementos de forma ascendente, pero, ¿se podría cambiar el orden de ordenación? Los **TreeSet** tienen un conjunto de operaciones adicionales, además de las que incluye por el hecho de ser un conjunto, que permite entre otras cosas, cambiar la forma de ordenar los elementos. Esto es especialmente útil cuando el tipo de objeto que se almacena no es un simple número, sino algo más complejo (un artículo por ejemplo). **TreeSet** es capaz de ordenar tipos básicos (números, cadenas y fechas) pero otro tipo de objetos no puede ordenarlos con tanta facilidad.

Para indicar a un **TreeSet** cómo tiene que ordenar los elementos, debemos decirle cuándo un elemento va antes o después que otro, y cuándo son iguales. Para ello, utilizamos la interfaz genérica **java.util.Comparator**, usada en general en algoritmos de ordenación, como veremos más adelante. Se trata de crear una clase que implemente dicha interfaz, así de fácil. Dicha interfaz requiere de un único método que debe calcular si un objeto pasado por parámetro es mayor, menor o igual que otro del mismo tipo. Veamos un ejemplo general de cómo implementar un comparador para una hipotética clase "Objeto":

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

```
class ComparadorDeObjetos implements Comparator<Objeto> {
    public int compare(Objeto o1, Objeto o2) { ... }
}
```

La interfaz **Comparator** obliga a implementar un único método, es el método **compare**, el cual tiene dos parámetros: los dos elementos a comparar. Las reglas son sencillas, a la hora de personalizar dicho método:

- ✓ Si el primer objeto (o1) es menor que el segundo (o2), debe retornar un número entero negativo.
- ✓ Si el primer objeto (o1) es mayor que el segundo (o2), debe retornar un número entero positivo.
- ✓ Si ambos son iguales, debe retornar 0.

A veces, cuando el orden que deben tener los elementos es diferente al orden real (por ejemplo cuando ordenamos los números en orden inverso), la definición de antes puede ser un poco liosa, así que es recomendable en tales casos pensar de la siguiente forma:

- ✓ Si el primer objeto (o1) debe ir antes que el segundo objeto (o2), retornar entero negativo.
- ✓ Si el primer objeto (o1) debe ir después que el segundo objeto (o2), retornar entero positivo.
- ✓ Si ambos son iguales, debe retornar 0.

Una vez creado el comparador simplemente tenemos que pasarlo como parámetro en el momento de la creación al **TreeSet**, y los datos internamente mantendrán dicha ordenación:

```
TreeSet<Objeto> ts=new TreeSet<Objeto>(new ComparadorDeObjetos());
```

Ejercicio resuelto

¿Fácil no? Pongámoslo en práctica. Imagínate que Objeto es una clase como la siguiente:

```
class Objeto {
    public int a;
    public int b;
}
```

Imagina que ahora, al añadirlos en un **TreeSet**, estos se tienen que ordenar de forma que la suma de sus atributos (a y b) sea descendente, ¿cómo sería el comparador?

8.- Listas (I).

Caso práctico

Juan se queda pensando después de que Ana le preguntara si sabía los tipos de colecciones que había en Java. Obviamente no lo sabía, son muchos tipos, pero ya tenía una respuesta preparada:

—Bueno, sea lo que sea, siempre puedes utilizar una lista para almacenar lo que sea. Yo siempre las uso, pues te permiten almacenar cualquier tipo de objeto, extraer uno de las lista sin tener que recorrerla entera, buscar si hay o no un elemento en ella, de forma cómoda. Son para mi el mejor invento desde la rueda —dijo Juan.

—Ya, supongo, pero hay dos tipos de listas que me interesan, **LinkedList** y **ArrayList**, ¿cuál es mejor? ¿Cuál me conviene más? —respondió Ana.



¿En qué se diferencia una lista de un conjunto? Las listas son elementos de programación un poco más avanzados que los conjuntos. Su ventaja es que amplían el conjunto de operaciones de las colecciones añadiendo operaciones extra, veamos algunas de ellas:

- ✓ Las listas si pueden almacenar duplicados, si no queremos duplicados, hay que verificar manualmente que el elemento no esté en la lista antes de su inserción.
- ✓ Acceso posicional. Podemos acceder a un elemento indicando su posición en la lista.
- ✓ Búsqueda. Es posible buscar elementos en la lista y obtener su posición. En los conjuntos, al ser colecciones sin aportar nada nuevo, solo se podía comprobar si un conjunto contenía o no un elemento de la lista, retornando verdadero o falso. Las listas mejoran este aspecto.
- ✓ Extracción de sublistas. Es posible obtener una lista que contenga solo una parte de los elementos de forma muy sencilla.



En Java, para las listas se dispone de una interfaz llamada **java.util.List**, y dos implementaciones (**java.util.LinkedList** y **java.util.ArrayList**), con diferencias significativas entre ellas. Los métodos de la interfaz List, que obviamente estarán en todas las implementaciones, y que permiten las operaciones anteriores son:

- ✓ `E get(int index)`. El método `get` permite obtener un elemento partiendo de su posición (`index`).
- ✓ `E set(int index, E element)`. El método `set` permite cambiar el elemento almacenado en una posición de la lista (`index`), por otro (`element`).
- ✓ `void add(int index, E element)`. Se añade otra versión del método `add`, en la cual se puede insertar un elemento (`element`) en la lista en una posición concreta (`index`), desplazando los existentes.
- ✓ `E remove(int index)`. Se añade otra versión del método `remove`, esta versión permite eliminar un elemento indicando su posición en la lista.
- ✓ `boolean addAll(int index, Collection<? extends E> c)`. Se añade otra versión del método `addAll`, que permite insertar una colección pasada por parámetro en una posición de la lista, desplazando el resto de elementos.
- ✓ `int indexOf(Object o)`. El método `indexOf` permite conocer la posición (índice) de un elemento, si dicho elemento no está en la lista retornará -1.
- ✓ `int lastIndexOf(Object o)`. El método `lastIndexOf` nos permite obtener la última ocurrencia del objeto en la lista (dado que la lista si puede almacenar duplicados).
- ✓ `List<E> subList(int from, int to)`. El método `subList` genera una sublista (una vista parcial de la lista) con los elementos comprendidos entre la posición inicial (incluida) y la posición final (no incluida).

Ten en cuenta que los elementos de una lista empiezan a numerarse por 0. Es decir, que el primer elemento de la lista es el 0. Ten en cuenta también que **List** es una interfaz genérica, por lo que **<E>** corresponde con el tipo base usado como parámetro genérico al crear la lista.



Autoevaluación

Si **M** es una lista de números enteros, ¿sería correcto poner `"M.add(M.size(), 3);"`?

- ☐ Sí.
- ☐ No.

8.1.- Listas (II).

Y, ¿cómo se usan las listas? Pues para usar una lista haremos uso de sus implementaciones **LinkedList** y **ArrayList**. Veamos un ejemplo de su uso y después obtendrás respuesta a esta pregunta.

Supongo que intuirás como se usan, pero nunca viene mal un ejemplo sencillo, que nos aclare las ideas. El siguiente ejemplo muestra como usar un **LinkedList** pero valdría también para **ArrayList** (no olvides importar las clases **java.util.LinkedList** y **java.util.ArrayList** según sea necesario). En este ejemplo se usan los métodos de acceso posicional a la lista:



```
LinkedList<Integer> t=new LinkedList<Integer>(); // Declaración y creación del LinkedList de
t.add(1); // Añade un elemento al final de la lista.
t.add(3); // Añade otro elemento al final de la lista.
t.add(1,2); // Añade en la posición 1 el elemento 2.
t.add(t.get(1)+t.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo agrega al
t.remove(0); // Elimina el primer elementos de la lista.
for (Integer i: t) System.out.println("Elemento:" + i); // Muestra la lista.
```

En el ejemplo anterior, se realizan muchas operaciones, ¿cuál será el contenido de la lista al final? Pues será 2, 3 y 5. En el ejemplo cabe destacar el uso del bucle for-each, recuerda que se puede usar en cualquier colección.

Veamos otro ejemplo, esta vez con **ArrayList**, de cómo obtener la posición de un elemento en la lista:

```
ArrayList<Integer> al=new ArrayList<Integer>(); // Declaración y creación del ArrayList de er
al.add(10); al.add(11); // Añadimos dos elementos a la lista.
al.set(al.indexOf(11), 12); // Sustituimos el 11 por el 12, primero lo buscamos y luego lo re
```

En el ejemplo anterior, se emplea tanto el método **indexOf** para obtener la posición de un elemento, como el método **set** para reemplazar el valor en una posición, una combinación muy habitual. El ejemplo anterior generará un **ArrayList** que contendrá dos números, el 10 y el 12. Veamos ahora un ejemplo algo más difícil:

```
al.addAll(0, t.subList(1, t.size()));
```

Este ejemplo es especial porque usa sublistas. Se usa el método **size** para obtener el tamaño de la lista. Después el método **subList** para extraer una sublista de la lista (que incluía en origen los números 2, 3 y 5), desde la posición 1 hasta el final de la lista (lo cual dejaría fuera al primer elemento). Y por último, se usa el método **addAll** para añadir todos los elementos de la sublista al **ArrayList** anterior.

Debes saber que las operaciones aplicadas a una sublista repercuten sobre la lista original. Por ejemplo, si ejecutamos el método **clear** sobre una sublista, se borrarán todos los elementos de la sublista, pero también se borrarán dichos elementos de la lista original:

```
al.subList(0, 2).clear();
```

Lo mismo ocurre al añadir un elemento, se añade en la sublista y en la lista original.



Debes conocer

Las listas enlazadas son un elemento muy recurrido y su funcionamiento interno es complejo. Te recomendamos el siguiente artículo de la wikipedia para profundizar un poco más en las listas enlazadas y los diferentes tipos que hay.

[Listas enlazadas.](#)



Autoevaluación

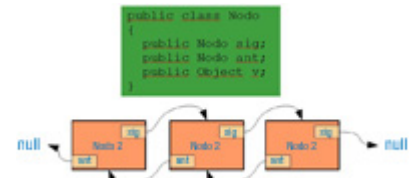
Completa con el número que falta.
Dado el siguiente código:

```
LinkedList<Integer> t=new LinkedList<Integer>();  
t.add(t.size()+1); t.add(t.size()+1); Integer suma = t.get(0) + t.get(1);
```

El valor de la variable suma después de ejecutarlo es .

8.2.- Listas (III).

¿Y en qué se diferencia un **LinkedList** de un **ArrayList**? Los **LinkedList** utilizan listas doblemente enlazadas, que son listas enlazadas (como se vio en un apartado anterior), pero que permiten ir hacia atrás en la lista de elementos. Los elementos de la lista se encapsulan en los llamados nodos. Los nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento. Tener un doble enlace significa que en cada nodo se almacena la información de cuál es el siguiente nodo y además, de cuál es el nodo anterior. Si un nodo no tiene nodo siguiente o nodo anterior, se almacena null o nulo para ambos casos.



No es el caso de los **ArrayList**. Estos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente a nosotros, no nos enteramos cuando se produce, pero eso redundaría en una diferencia de rendimiento notable dependiendo del uso. Los **ArrayList** son más rápidos en cuanto a acceso a los elementos, acceder a un elemento según su posición es más rápido en un array que en una lista doblemente enlazada (hay que recorrer la lista). En cambio, eliminar un elemento implica muchas más operaciones en un array que en una lista enlazada de cualquier tipo.

¿Y esto que quiere decir? **Que si se van a realizar muchas operaciones de eliminación de elementos sobre la lista, conviene usar una lista enlazada (LinkedList), pero si no se van a realizar muchas eliminaciones, sino que solamente se van a insertar y consultar elementos por posición, conviene usar una lista basada en arrays redimensionados (ArrayList).**

LinkedList tiene otras ventajas que nos puede llevar a su uso. Implementa las interfaces **java.util.Queue** y **java.util.Deque**. Dichas interfaces permiten hacer uso de las listas como si fueran una cola de prioridad o una pila, respectivamente.

Las colas, también conocidas como colas de prioridad, son una lista pero que aportan métodos para trabajar de forma diferente. ¿Tú sabes lo que es hacer cola para que te atiendan en una ventanilla? Pues igual. Se trata de que el que primero llega es el primero en ser atendido (**FIFO**). Simplemente se aportan tres métodos nuevos: meter en el final de la lista (**add** y **offer**), sacar y eliminar el elemento más antiguo (**poll**), y examinar el elemento al principio de la lista sin eliminarlo (**peek**). Dichos métodos están disponibles en las listas enlazadas **LinkedList**:

- ✓ boolean add(E e) y boolean offer(E e), retornarán true si se ha podido insertar el elemento al final de la LinkedList.
- ✓ E poll() retornará el primer elemento de la LinkedList y lo eliminará de la misma. Al insertar al final, los elementos más antiguos siempre están al principio. Retornará null si la lista está vacía.
- ✓ E peek() retornará el primer elemento de la LinkedList pero no lo eliminará, permite examinarlo. Retornará null si la lista está vacía.

Las pilas, mucho menos usadas, son todo lo contrario a las listas. Una pila es igual que una montaña de hojas en blanco, para añadir hojas nuevas se ponen encima del resto, y para retirar una se coge la primera que hay, encima de todas. En las pilas el último en llegar es el primero en ser atendido. Para ello se proveen de tres métodos: meter al principio de la pila (**push**), sacar y eliminar del principio de la pila (**pop**), y examinar el primer elemento de la pila (**peek**, igual que si usara la lista como una cola).

Las pilas se usan menos y haremos menos hincapié en ellas. Simplemente ten en mente que, tanto las colas como las pilas, son una lista enlazada sobre la que se hacen operaciones especiales.



Autoevaluación

Dada la siguiente lista, usada como si fuera una cola de prioridad, ¿cuál es la letra que se mostraría por la pantalla tras su ejecución?

```
LinkedList<String> tt=new LinkedList<String>();
tt.offer("A"); tt.offer("B"); tt.offer("C");
System.out.println(tt.poll());
```

- ☐ A.
- ☐ C.

☐ D.

8.3.- Listas (IV).

A la hora de usar las listas, hay que tener en cuenta un par de detalles, ¿sabes cuáles? Es sencillo, pero importante.

No es lo mismo usar las colecciones (listas y conjuntos) con objetos inmutables (**Strings**, **Integer**, etc.) que con objetos mutables. Los objetos inmutables no pueden ser modificados después de su creación, por lo que cuando se incorporan a la lista, a través de los métodos **add**, se pasan por copia (es decir, se realiza una copia de los mismos). En cambio los objetos mutables (como las clases que tú puedes crear), no se copian, y eso puede producir efectos no deseados.



Imaginate la siguiente clase, que contiene un número:

```
class Test
{
    public Integer num;
    Test (int num) { this.num=new Integer(num); }
}
```

La clase de antes es mutable, por lo que no se pasa por copia a la lista. Ahora imagina el siguiente código en el que se crea una lista que usa este tipo de objeto, y en el que se insertan dos objetos:

```
Test p1=new Test(11); // Se crea un objeto Test donde el entero que contiene vale 11.
Test p2=new Test(12); // Se crea otro objeto Test donde el entero que contiene vale 12.
LinkedList<Test> lista=new LinkedList<Test>(); // Creamos una lista enlazada para objetos tip
lista.add(p1); // Añadimos el primero objeto test.
lista.add(p2); // Añadimos el segundo objeto test.
for (Test p:lista) System.out.println(p.num); // Mostramos la lista de objetos.
```

¿Qué mostraría por pantalla el código anterior? Simplemente mostraría los números 11 y 12. Ahora bien, ¿qué pasa si modificamos el valor de uno de los números de los objetos test? ¿Qué se mostrará al ejecutar el siguiente código?

```
p1.num=44;
for (Test p:lista) System.out.println(p.num);
```

El resultado de ejecutar el código anterior es que se muestran los números 44 y 12. El número ha sido modificado y no hemos tenido que volver a insertar el elemento en la lista para que en la lista se cambie también. Esto es porque en la lista no se almacena una copia del objeto **Test**, sino un apuntador a dicho objeto (solo hay una copia del objeto a la que se hace referencia desde distintos lugares).

Citas para pensar

"Controlar la complejidad es la esencia de la programación."



Autoevaluación

Los elementos de un ArrayList de objetos Short se copian al insertarse al ser objetos mutables.
¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

9.- Conjuntos de pares clave/valor.

Caso práctico

Juan se quedó pensativo después de la conversación con **Ana**. **Ana** se fue a su puesto a seguir trabajando, pero él se quedó dándole vueltas al asunto...

—Sí que está bien preparada **Ana**, me ha puesto en jaque y no sabía qué responder.

El hecho de no poder ayudar a Ana le frustró un poco.

De repente, apareció María. Entonces Juan aprovecha el momento para preguntar con más detalle acerca del trabajo de Ana. María se lo cuenta y de repente, se le enciende una bombilla a Juan... dice: —Vale, creo que puedo ayudar a Ana en algo, le aconsejaré usar mapas y le explicaré cómo se usan.



¿Cómo almacenarías los datos de un diccionario? Tenemos por un lado cada palabra y por otro su significado. Para resolver este problema existen precisamente los **arrays asociativos**. Un tipo de array asociativo son los mapas o diccionarios, que permiten almacenar pares de valores conocidos como clave y valor. La clave se utiliza para acceder al valor, como una entrada de un diccionario permite acceder a su definición.

En Java existe la interfaz **java.util.Map** que define los métodos que deben tener los mapas, y existen tres implementaciones principales de dicha interfaz: **java.util.HashMap**, **java.util.TreeMap** y **java.util.LinkedHashMap**.

¿Te suenan? Claro que sí. Cada una de ellas, respectivamente, tiene características similares a **HashSet**, **TreeSet** y **LinkedHashSet**, tanto en funcionamiento interno como en rendimiento.

Los **mapas** utilizan **clases genéricas** para dar extensibilidad y flexibilidad, y permiten definir un tipo base para la clave, y otro tipo diferente para el valor. Veamos un ejemplo de como crear un mapa, que es extensible a los otros dos tipos de mapas:



```
HashMap<String,Integer> t=new HashMap<String,Integer>();
```

El mapa anterior permite usar cadenas como llaves y almacenar de forma asociada a cada llave, un número entero. Veamos los métodos principales de la interfaz **Map**, disponibles en todas las implementaciones. En los ejemplos, **V** es el tipo base usado para el valor y **K** el tipo base usado para la llave:

Métodos principales de los mapas.

Método.	Descripción.
V put(K key, V value);	Inserta un par de objetos llave (key) y valor (value) en el mapa. Si la llave ya existe en el mapa, entonces retornará el valor asociado que tenía antes, si la llave no existía, entonces retornará null.
V get(Object key);	Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, retornará null.
V remove(Object key);	Elimina la llave y el valor asociado. Retorna el valor asociado a la llave, por si lo queremos utilizar para algo, o null, si la llave no existe.
boolean containsKey(Object key);	Retornará true si el mapa tiene almacenada la llave pasada por

Método.	Descripción.
	parámetro, <code>false</code> en cualquier otro caso.
<code>boolean containsValue(Object value);</code>	Retornará <code>true</code> si el mapa tiene almacenado el valor pasado por parámetro, <code>false</code> en cualquier otro caso.
<code>int size();</code>	Retornará el número de pares llave y valor almacenado en el mapa.
<code>boolean isEmpty();</code>	Retornará <code>true</code> si el mapa está vacío, <code>false</code> en cualquier otro caso.
<code>void clear();</code>	Vacía el mapa.



Autoevaluación

Completa el siguiente código para que al final se muestre el número 40 por pantalla:

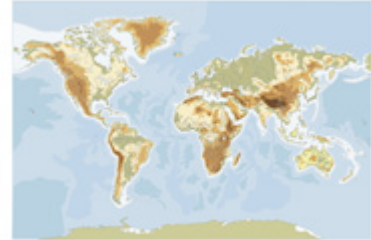
```
HashMap< String,  > datos=new  < String,String >();  
datos. ("A","44");  
System.out.println(Integer. (datos. (""))-);
```

10.- Iteradores (I).

Caso práctico

Juan se acercó a la mesa de Ana y le dijo:

—María me ha contado la tarea que te ha encomendado y he pensado que quizás te convendría usar mapas en algunos casos. Por ejemplo, para almacenar los datos del pedido asociados con una etiqueta: nombre, dirección, fecha, etc. Así creo que te será más fácil generar luego el XML.



—La verdad es que pensaba almacenar los datos del pedido en una clase especial llamada Pedido. No tengo ni idea de que son los mapas -dijo Ana-, supongo que son como las listas, ¿tienen iteradores?

—Según me ha contado María, no necesitas hacer tanto, no es necesario crear una clase específica para los pedidos. Y respondiendo a tu pregunta, los mapas no tienen iteradores, pero hay una solución... te explico.

¿Qué son los iteradores realmente? Son un mecanismo que nos permite recorrer todos los elementos de una colección de forma sencilla, de forma secuencial, y de forma segura. Los mapas, como no derivan de la interfaz Collection realmente, no tienen iteradores, pero como veremos, existe un truco interesante.

Los iteradores permiten recorrer las colecciones de dos formas: **bucles for-each** (existentes en Java a partir de la versión 1.5) **y a través de un bucle normal creando un iterador**. Como los bucles for-each ya los hemos visto antes (y ha quedado patente su simplicidad), nos vamos a centrar en el otro método, especialmente útil en versiones antiguas de Java. Ahora la pregunta es, ¿cómo se crea un iterador? Pues invocando el método "**iterator()**" de cualquier colección. Veamos un ejemplo (en el ejemplo **t** es una colección cualquiera):



```
Iterator<Integer> it=t.iterator();
```

Fíjate que se ha especificado un parámetro para el tipo de dato genérico en el iterador (poniendo "**<Integer>**" después de **Iterator**). Esto es porque los iteradores son también clases genéricas, y es necesario especificar el tipo base que contendrá el iterador. Si no se especifica el tipo base del iterador, igualmente nos permitiría recorrer la colección, pero retornará objetos tipo **Object** (clase de la que derivan todas las clases), con lo que nos veremos obligados a forzar la conversión de tipo.

Para recorrer y gestionar la colección, el iterador ofrece tres métodos básicos:

- ✓ `boolean hasNext()`. Retornará true si le quedan más elementos a la colección por visitar. False en caso contrario.
- ✓ `E next()`. Retornará el siguiente elemento de la colección, si no existe siguiente elemento, lanzará una excepción (`NoSuchElementException` para ser exactos), con lo que conviene chequear primero si el siguiente elemento existe.
- ✓ `remove()`. Elimina de la colección el último elemento retornado en la última invocación de `next` (no es necesario pasárselo por parámetro). Cuidado, si `next` no ha sido invocado todavía, saltará una incomoda excepción.

¿Cómo recorreríamos una colección con estos métodos? Pues de una forma muy sencilla, un simple bucle mientras (**while**) con la condición **hasNext()** nos permite hacerlo:

```
while (it.hasNext()) // Mientras que haya un siguiente elemento, seguiremos en el bucle.
```

```
{  
    Integer t=it.next(); // Escogemos el siguiente elemento.  
    if (t%2==0) it.remove(); //Si es necesario, podemos eliminar el elemento extraído  
}
```

¿Qué elementos contendría la lista después de ejecutar el bucle? Efectivamente, solo los números impares.

Reflexiona

Las listas permiten acceso posicional a través de los métodos **get** y **set**, y acceso secuencial a través de iteradores, ¿cuál es para tí la forma más cómoda de recorrer todos los elementos? ¿Un acceso posicional a través un bucle `"for (i=0;i<lista.size();i++)"` o un acceso secuencial usando un bucle `"while (iterador.hasNext())"`?

10.1.- Iteradores (II).

¿Qué inconvenientes tiene usar los iteradores sin especificar el tipo de objeto? En el siguiente ejemplo, se genera una lista con los números del 0 al 10. De la lista, se eliminan aquellos que son pares y solo se dejan los impares. En el ejemplo de la izquierda se especifica el tipo de objeto del iterador, en el ejemplo de la derecha no, observa el uso de la conversión de tipos en la línea 6.

Comparación de usos de los iteradores, con o sin especificación de tipo

Ejemplo indicando el tipo de objeto de iterador	
<pre> ArrayList <Integer> lista=new ArrayList<Integer>(); for (int i=0;i<10;i++) lista.add(i); Iterator<Integer> it=lista.iterator(); while (it.hasNext()) { Integer t=it.next(); if (t%2==0) it.remove(); } </pre>	<pre> ArrayList lista=new ArrayList(); for (int i=0;i<10;i++) lista.add(i); Iterator it=lista.iterator(); while (it.hasNext()) { Integer t=it.next(); if (t%2==0) it.remove(); } </pre>

Un iterador es seguro porque esta pensado para no sobrepasar los límites de la colección, ocultando operaciones más complicadas que pueden repercutir en errores de software. Pero realmente se convierte en inseguro cuando es necesario hacer la operación de conversión de tipos. Si la colección no contiene los objetos esperados, al intentar hacer la conversión, saltará una incomoda excepción. Usar genéricos aporta grandes ventajas, pero usándolos adecuadamente.

Para recorrer los mapas con iteradores, hay que hacer un pequeño truco. Usamos el método `entrySet` que ofrecen los mapas para generar un conjunto con las entradas (pares de llave-valor), o bien, el método `keySet` para generar un conjunto con las llaves existentes en el mapa. Veamos como sería para el segundo caso, el más sencillo:



```

HashMap<Integer,Integer> mapa=new HashMap<Integer,Integer>test();
for (int i=0;i<10;i++) mapa.put(i, i); // Insertamos datos de prueba en el mapa.
for (Integer llave:mapa.keySet()) // Recorremos el conjunto generado por keySet, contendrá la
{
    Integer valor=mapa.get(llave); //Para cada llave, accedemos a su valor si es neces
}

```

Lo único que tienes que tener en cuenta es que el conjunto generado por **keySet** no tendrá obviamente el método **add** para añadir elementos al mismo, dado que eso tendrás que hacerlo a través del mapa.

Recomendación

Si usas iteradores, y piensas eliminar elementos de la colección (e incluso de un mapa), debes usar el método **remove** del iterador y no el de la colección. Si eliminas los elementos utilizando el método **remove** de la colección, mientras estás dentro de un bucle de iteración, o dentro de un bucle `for-each`, los fallos que pueden producirse en tu programa son impredecibles. ¿Logras adivinar porqué se pueden producir dichos problemas?

Los problemas son debidos a que el **método** `remove` del iterador elimina el elemento de dos sitios: de la colección y del iterador en sí (que mantiene interiormente información del orden de los elementos). Si usas el método **`remove`** de la colección, la información solo se elimina de un lugar, de la colección.



Autoevaluación

¿Cuándo debemos invocar el método `remove()` de los iteradores?

- ☐ En cualquier momento.
- ☐ Después de invocar el método `next()`.
- ☐ Después de invocar el método `hasNext()`.
- ☐ No es conveniente usar este método para eliminar elementos, es mejor usar el de la colección.

11.- Algoritmos (I).

Caso práctico

Ada se acercó a preguntar a **Ana**. **Ada** era la jefa y **Ana** le tenía mucho respeto. **Ada** le preguntó cómo llevaba la tarea que le había encomendado **María**. Era una tarea importante, así que prestó mucha atención.

Ana le enseñó el código que estaba elaborando, le dijo que en un principio había pensado crear una clase llamada Pedido, para almacenar los datos del pedido, pero que Juan le recomendó usar mapas para almacenar los pares de valor y dato. Así que se decantó por usar mapas para ese caso. Le comentó también que para almacenar los artículos si había creado una pequeña clase llamada Artículo. Ada le dio el visto bueno:

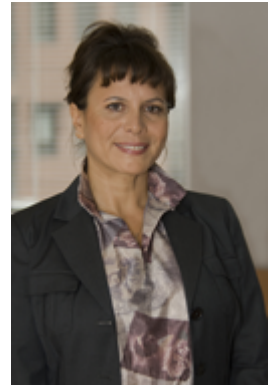
—Pues Juan te ha recomendado de forma adecuada, no vas a necesitar hacer ningún procesamiento especial de los datos del pedido, solo convertirlos de un formato específico a XML. Eso sí, sería recomendable que los artículos del pedido vayan ordenados por código de artículo —dijo Ada.

—¿Ordenar los artículos? Vaya, que jaleo -respondió Ana.

—Arriba ese ánimo mujer, si has usado listas es muy fácil, déjame ver tu código y te explicaré cómo hacerlo.

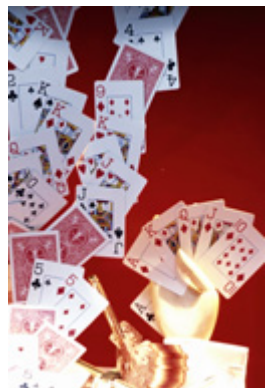
—Sí, aquí está, es el siguiente, espero que te guste:

[Procesar archivo de pedido, 2ª Fase.](#) (0.01 MB)



La palabra algoritmo seguro que te suena, pero, ¿a qué se refiere en el contexto de las colecciones y de otras estructuras de datos? Las colecciones, los arrays e incluso las cadenas, tienen un conjunto de operaciones típicas asociadas que son habituales. Algunas de estas operaciones ya las hemos visto antes, pero otras no. Veamos para qué nos pueden servir estas operaciones:

- ✓ Ordenar listas y arrays.
- ✓ Desordenar listas y arrays.
- ✓ Búsqueda binaria en listas y arrays.
- ✓ Conversión de arrays a listas y de listas a array.
- ✓ Partir cadenas y almacenar el resultado en un array.



Estos algoritmos están en su mayoría recogidos como métodos estáticos de las **clases** **java.util.Collections** y **java.util.Arrays**, salvo los referentes a cadenas obviamente.

Los algoritmos de ordenación ordenan los elementos en orden natural, siempre que Java sepa como ordenarlos. Como se explico en el apartado de conjuntos, cuando se desea que la ordenación siga un orden diferente, o simplemente los elementos no son ordenables de forma natural, hay que facilitar un mecanismo para que se pueda producir la ordenación. Los tipos "ordenables" de forma natural son los enteros, las cadenas (orden alfabético) y las fechas, y por defecto su orden es ascendente.

La clase **Collections** y la clase **Arrays** facilitan el método **sort**, que permiten ordenar respectivamente listas y arrays. Los siguientes ejemplos ordenarían los números de forma ascendente (de menor a mayor):

Ordenación natural en listas y arrays.

Ejemplo de ordenación de un array de números	Ejemplo de ordenación en una lista
<pre>Integer[] array={10,9,99,3,5};</pre>	<pre>ArrayList<Integer> lista=new ArrayList<>(); lista.add(10); lista.add(9); lista.add(99); lista.add(3); lista.add(5);</pre>

```
Arrays.sort(array);
```

```
lista.add(3); lista.add(5);  
Collections.sort(lista);
```

11.1.- Algoritmos (II).

En Java hay dos mecanismos para cambiar la forma en la que los elementos se ordenan. ¿Recuerdas la tarea que Ada pidió a Ana? Que los artículos del pedido aparecieran ordenados por código de artículo. Imagina que tienes los artículos almacenados en una lista llamada **"articulos"**, y que cada artículo se almacena en la siguiente clase (fíjate que el código de artículo es una cadena y no un número):



```
class Articulo {
    public String codArticulo; // Código de artículo
    public String descripcion; // Descripción del artículo.
    public int cantidad; // Cantidad a proveer del artículo.
}
```

La primera forma de ordenar consiste en crear una clase que implemente la interfaz **java.util.Comparator**, y en ende, el método **compare** definido en dicha interfaz. Esto se explicó en el apartado de conjuntos, al explicar el **TreeSet**, así que no vamos a profundizar en ello. No obstante, el comparador para ese caso podría ser así:

```
class comparadorArticulos implements Comparator<Articulo>
{
    @Override
    public int compare( Articulo o1, Articulo o2) { return o1.codArticulo.compareTo(o2.codArticulo); }
}
```

Una vez creada esta clase, ordenar los elementos es muy sencillo, simplemente se pasa como segundo parámetro del método **sort** una instancia del comparador creado:

```
Collections.sort(articulos, new comparadorArticulos());
```

La segunda forma es quizás más sencilla cuando se trata de objetos cuya ordenación no existe de forma natural, pero requiere modificar la clase **Articulo**. Consiste en hacer que los objetos que se meten en la lista o array implementen la interfaz **java.util.Comparable**. **Todos los objetos que implementan la interfaz Comparable son "ordenables" y se puede invocar el método sort sin indicar un comparador para ordenarlos.** La interfaz **comparable** solo requiere implementar el método **compareTo**:

```
class Articulo implements Comparable<Articulo>{
    public String codArticulo;
    public String descripcion;
    public int cantidad;

    @Override
    public int compareTo(Articulo o) { return codArticulo.compareTo(o.codArticulo); }
}
```

Del ejemplo anterior se pueden denotar dos cosas importantes: que la interfaz **Comparable** es genérica y que para que funcione sin problemas es conveniente indicar el tipo base sobre el que se permite la comparación (en este caso, el objeto **Articulo** debe compararse consigo mismo), y que el método **compareTo** solo admite un parámetro, dado que comparará el objeto con el que se pasa por parámetro.

El funcionamiento del método **compareTo** es el mismo que el método **compare** de la interfaz **Comparator**: si la clase que se pasa por parámetro es igual al objeto, se tendría que retornar 0; si es menor o anterior, se debería retornar un número menor que cero; si es mayor o posterior, se debería retornar un número mayor que 0.

Ordenar ahora la lista de artículos es sencillo, fíjate que fácil: **"Collections.sort(articulos);"**



Autoevaluación

Si tienes que ordenar los elementos de una lista de tres formas diferentes, ¿cuál de los métodos anteriores es más conveniente?

- ☐ Usar comparadores, a través de la interfaz **java.util.Comparator**.
- ☐ Implementar la interfaz comparable en el objeto almacenado en la lista.

11.2.- Algoritmos (III).

¿Qué más ofrece las **clases** `java.util.Collections` y `java.util.Arrays` de Java? Una vez vista la ordenación, quizás lo más complicado, veamos algunas operaciones adicionales. En los ejemplos, la variable **"array"** es un array y la variable **"lista"** es una lista de cualquier tipo de elemento:

Operaciones adicionales sobre listas y arrays.

Operación	Descripción	Ejemplos
Desordenar una lista.	Desordena una lista, este método no está disponible para arrays.	<code>Collections.shuffle (lista);</code>
Rellenar una lista o array.	Rellena una lista o array copiando el mismo valor en todos los elementos del array o lista. Útil para reiniciar una lista o array.	<code>Collections.fill (lista,elemento);</code> <code>Arrays.fill (array,elemento);</code>
Búsqueda binaria.	Permite realizar búsquedas rápidas en un una lista o array ordenados. Es necesario que la lista o array estén ordenados, si no lo están, la búsqueda no tendrá éxito.	<code>Collections.binarySearch(lista,elemento);</code> <code>Arrays.binarySearch(array, elemento);</code>
Convertir un array a lista.	Permite rápidamente convertir un array a una lista de elementos, extremadamente útil. No se especifica el tipo de lista retornado (no es <code>ArrayList</code> ni <code>LinkedList</code>), solo se especifica que retorna una lista que implementa la interfaz <code>java.util.List</code> .	<code>List lista=Arrays.asList(array);</code> Si el tipo de dato almacenado en el array es conocido (Integer por ejemplo), es conveniente especificar el tipo de objeto de la lista: <code>List<Integer>lista = Arrays.asList(array);</code>
Convertir una lista a array.	Permite convertir una lista a array. Esto se puede realizar en todas las colecciones, y no es un método de la clase <code>Collections</code> , sino propio de la interfaz <code>Collection</code> . Es conveniente que sepas de su existencia.	Para este ejemplo, supondremos que los elementos de la lista son números, dado que hay que crear un array del tipo almacenado en la lista, y del tamaño de la lista: <code>Integer[] array=new Integer[lista.size()];</code> <code>lista.toArray(array)</code>
Dar la vuelta.	Da la vuelta a una lista, poniéndola en orden inverso al que tiene.	<code>Collections.reverse(lista);</code>

Otra operación que no se ha visto hasta ahora es la dividir una cadena en partes. Cuando una cadena está formada internamente por trozos de texto claramente delimitados por un separador (una coma, un punto y coma o cualquier otro), es posible dividir la cadena y obtener cada uno de los trozos de texto por separado en un array de cadenas. Es una operación sencilla, pero dado que es necesario conocer el funcionamiento de los arrays y de las expresiones regulares para su uso, no se ha podido ver hasta ahora. Para poder realizar esta operación, usaremos el método **split** de la clase **String**. El delimitador o separador es una expresión regular, único argumento del método **split**, y puede ser obviamente todo lo complejo que sea necesario:



```
String texto="Z,B,A,X,M,O,P,U";
String []partes=texto.split(",");
Arrays.sort(partes);
```

En el ejemplo anterior la cadena **texto** contiene una serie de letras separadas por comas. La cadena se ha dividido con el método **split**, y se ha guardado cada carácter por separado en un array. Después se ha ordenado el array.

¡Increíble lo que se puede llegar a hacer con solo tres líneas de código!

Para saber más

En el siguiente vídeo podrás ver en qué consiste la búsqueda binaria y cómo se aplica de forma sencilla:

[Resumen textual alternativo](#)

12.- Tratamiento de documentos estructurados XML.

Caso práctico

Ana ya ha terminado lo principal, ya es capaz de procesar el pedido y de almacenar la información en estructuras de memoria que luego podrá proyectar a un documento XML, incluso ha ordenado los artículos en base al código de artículo, definitivamente era bastante más fácil de lo que ella pensaba.

Ahora le toca la tarea más ardua de todas, o al menos así lo ve ella, generar el documento XML con la información del pedido, ¿le resultará muy difícil?



¿Qué es XML? XML es un mecanismo extraordinariamente sencillo para estructurar, almacenar e intercambiar información entre sistemas informáticos. XML define un lenguaje de etiquetas, muy fácil de entender pero con unas reglas muy estrictas, que permite encapsular información de cualquier tipo para posteriormente ser manipulada. Se ha extendido tanto que hoy día es un estándar en el intercambio de información entre sistemas.

La información en XML va escrita en texto legible por el ser humano, pero no está pensada para que sea leída por un ser humano, sino por una máquina. La información va codificada generalmente en [unicode](#), pero estructurada de forma que una máquina es capaz de procesarla eficazmente. Esto tiene una clara ventaja: si necesitamos modificar algún dato de un documento en XML, podemos hacerlo con un [editor de texto plano](#). Veamos los elementos básicos del XML:



Elementos de un documento XML.

Elemento	Descripción	Ejemplo
Cabecera o declaración del XML.	Es lo primero que encontramos en el documento XML y define cosas como, por ejemplo, la codificación del documento XML (que suele ser ISO-8859-1 o UTF-8) y la versión del estándar XML que sigue nuestro documento XML.	<code><?xml version="1.0" encoding="ISO-8859-1"?></code>
Etiquetas.	Una etiqueta es un delimitador de datos, y a su vez, un elemento organizativo. La información va entre las etiquetas de apertura (" <code><pedido></code> ") y cierre (" <code></pedido></code> "). Fíjate en el nombre de la etiqueta ("pedido"), debe ser el mismo tanto en el cierre como en la apertura, respetando mayúsculas.	<code><pedido></code> información del pedido <code></pedido></code>
Atributos.	Una etiqueta puede tener asociado uno o más atributos. Siempre deben ir detrás del nombre de la etiqueta, en la etiqueta de apertura, poniendo el nombre del atributo seguido de igual y el valor encerrado entre comillas . Siempre debes dejar al menos un espacio entre los atributos.	<code><articulo cantidad="20"></code> información <code></articulo></code>
Texto.	Entre el cierre y la apertura de una etiqueta puede haber texto.	<code><cliente></code> Muebles Bonitos S.A.

		<code></cliente></code>
Etiquetas sin contenido.	Cuando una etiqueta no tiene contenido, no tiene porqué haber una etiqueta de cierre, pero no debes olvidar poner la barra de cierre ("/") al final de la etiqueta para indicar que no tiene contenido.	<code><fecha entrega="1/1/2012" /></code>
Comentario.	Es posible introducir comentarios en XML y estos van dirigidos generalmente a un ser humano que lee directamente el documento XML.	<code><!-- comentario --></code>

El nombre de la etiqueta y de los nombres de los atributos no deben tener espacios. También es conveniente evitar los puntos, comas y demás caracteres de puntuación. En su lugar se puede usar el guión bajo ("**<pedido_enviado>** ... **</pedido_enviado>**").



Autoevaluación

Señala las líneas que no serían elementos XML válidos.

- ☐ `<cliente>Informática Elegante <cliente/>`
- ☐ `<!-- -->`
- ☐ `<pedido fechaentrega=25 />`
- ☐ `<direccion_entrega>sin especificar</direccion_entrega>`

Mostrar Información

12.1.- ¿Qué es un documento XML?

Los documentos XML son documentos que solo utilizan los elementos expuestos en el apartado anterior (declaración, etiquetas, comentarios, etc.) de **forma estructurada**. Siguen una estructura de árbol, pseudo-jerárquica, permitiendo agrupar la información en diferentes niveles, que van desde la raíz a las hojas.

Para comprender la estructura de un documento XML vamos a utilizar una terminología afín a la forma en la cual procesaremos los documentos XML. Un documento XML está compuesto desde el punto de vista de programación por nodos, por nodos que pueden (o no) contener otros nodos. Todo es un nodo:



- ✓ El par formado por la etiqueta de apertura ("`<etiqueta>`") y por la de cierre ("`</etiqueta>`"), junto con todo su contenido (elementos, atributos y texto de su interior) es un nodo llamado elemento (`Element` desde el punto de vista de programación). Un elemento puede contener otros elementos, es decir, puede contener en su interior subetiquetas, de forma anidada.
- ✓ Un atributo es un nodo especial llamado atributo (`Attr` desde el punto de vista de programación), que solo puede estar dentro de un elemento (concretamente dentro de la etiqueta de apertura).
- ✓ El texto es un nodo especial llamado texto (`Text`), que solo puede estar dentro de una etiqueta.
- ✓ Un comentario es un nodo especial llamado comentario (`Comment`), que puede estar en cualquier lugar del documento XML.
- ✓ Y por último, un documento es un nodo que contiene una jerarquía de nodos en su interior. Está formado opcionalmente por una declaración, opcionalmente por uno o varios comentarios y **obligatoriamente por un único elemento**.

Esto es un poco lioso, ¿verdad? Vamos a clarificarlo con ejemplos. Primero, tenemos que entender la diferencia entre nodos padre y nodos hijo. Un elemento (par de etiquetas) puede contener varios nodos hijo, que pueden ser texto u otros elementos. Por ejemplo:

```
<padre att1="valor" att2="valor">
  texto 1
  <ethija> texto 2 </ethija>
</padre>
```

En el ejemplo anterior, el elemento padre tendría dos hijos: el texto "**texto 1**", sería el primer hijo, y el elemento etiquetado como "**ethija**", el segundo. Tendría también dos atributos, que sería nodos hijo también pero que se consideran especiales y la forma de acceso es diferente. A su vez, el elemento "**ethija**" tiene un nodo hijo, que será el texto "**texto 2**". ¿Fácil no?

Ahora veamos el conjunto, un documento estará formado, como se dijo antes, por algunos elementos opcionales, y obligatoriamente por un único elemento (es decir, por único par de etiquetas que lo engloba todo) que contendrá internamente el resto de información como nodos hijo. Por ejemplo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<pedido>
  <cliente> texto </cliente>
  <codCliente> texto </codCliente>
  ...
</pedido>
```

La etiqueta **pedido** del ejemplo anterior, será por tanto el elemento raíz del documento y dentro de él estará toda la información del documento XML. Ahora seguro que es más fácil, ¿no?

Para saber más

En el siguiente enlace podrás encontrar una breve guía, en inglés, de las tecnologías asociadas a XML actuales. Son bastantes y su potencial es increíble.

[Breve guía de las tecnologías XML.](#)

12.2.- Librerías para procesar documentos XML (I).

¿Quién establece las bases del XML? Pues el W3C o World Wide Web Consortium es la entidad que establece las bases del XML. Dicha entidad, además de describir como es el XML internamente, define un montón de tecnologías estándar adicionales para verificar, convertir y manipular documentos XML. Nosotros no vamos a explorar todas las tecnologías de XML aquí (son muchísimas), solamente vamos a usar dos de ellas, aquellas que nos van a permitir manejar de forma simple un documento XML:



- ✓ Procesadores de XML. Son librerías para leer documentos XML y comprobar que están bien formados. En Java, el procesador más utilizado es SAX, lo usaremos pero sin percatarnos casi de ello.
- ✓ XML DOM. Permite transformar un documento XML en un modelo de objetos (de hecho DOM significa Document Object Model), accesible cómodamente desde el lenguaje de programación. DOM almacena cada elemento, atributo, texto, comentario, etc. del documento XML en una estructura tipo árbol compuesta por nodos fácilmente accesibles, sin perder la jerarquía del documento. A partir de ahora, la estructura DOM que almacena un XML la llamaremos árbol o jerarquía de objetos DOM.

En Java, estas y otras funciones están implementadas en la librería JAXP (Java API for XML Processing), y ya van incorporadas en la edición estándar de Java (Java SE). En primer lugar vamos a ver como convertir un documento XML a un árbol DOM, y viceversa, para después ver cómo manipular desde Java un árbol DOM.

Para cargar un documento XML tenemos que hacer uso de un procesador de documentos XML (conocidos generalmente como parsers) y de un constructor de documentos DOM. Las clases de Java que tendremos que utilizar son:

- ✓ `javax.xml.parsers.DocumentBuilder`: será el procesador y transformará un documento XML a DOM, se le conoce como constructor de documentos.
- ✓ `javax.xml.parsers.DocumentBuilderFactory`: permite crear un constructor de documentos, es una fábrica de constructores de documentos.
- ✓ `org.w3c.dom.Document`: una instancia de esta clase es un documento XML pero almacenado en memoria siguiendo el modelo DOM. Cuando el parser procesa un documento XML creará una instancia de esta clase con el contenido del documento XML.

Ahora bien, ¿esto cómo se usa? Pues muy fácil, en pocas líneas (no olvides importar las librerías):

```
try {
    // 1º Creamos una nueva instancia de un fabrica de constructores de documentos.
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    // 2º A partir de la instancia anterior, fabricamos un constructor de documentos,
    DocumentBuilder db = dbf.newDocumentBuilder();
    // 3º Procesamos el documento (almacenado en un archivo) y lo convertimos en un é
    Document doc=db.parse(CaminoAArchivoXml);
} catch (Exception ex) {
    System.out.println("¡Error! No se ha podido cargar el documento XML.");
}
```

Es un poco enrevesado pero no tiene mucha complicación, es un pelín más complicado para hacer el camino inverso (pasar el DOM a XML). Este proceso puede generar hasta tres tipos de excepciones diferentes. La más común, que el documento XML esté mal formado, por lo que tienes que tener cuidado con la sintaxis XML.



Autoevaluación

¿Cuál es la función de la clase `org.w3c.dom.Document`?



Procesar el documento XML.

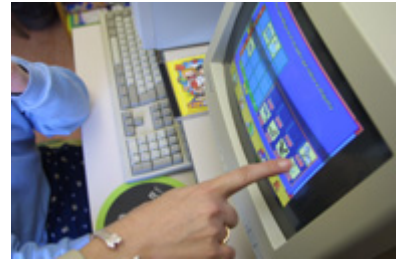
- ☐ Almacenar el documento XML en un modelo de objetos accesible desde Java.
- ☐ Fabricar un nuevo constructor de documentos.

12.2.1.- Librerías para procesar documentos XML (II).

¿Y cómo paso la jerarquía o árbol de objetos DOM a XML? En Java esto es un pelín más complicado que la operación inversa, y requiere el uso de un montón de clases del paquete **java.xml.transform**, pues la idea es transformar el árbol DOM en un archivo de texto que contiene el documento XML.

Las clases que tendremos que usar son:

- ✓ `javax.xml.transform.TransformerFactory`. Fábrica de transformadores, permite crear un nuevo transformador que convertirá el árbol DOM a XML.
- ✓ `javax.xml.transform.Transformer`. Transformador que permite pasar un árbol DOM a XML.
- ✓ `javax.xml.transform.TransformerException`. Excepción lanzada cuando se produce un fallo en la transformación.
- ✓ `javax.xml.transform.OutputKeys`. Clase que contiene opciones de salida para el transformador. Se suele usar para indicar la codificación de salida (generalmente UTF-8) del documento XML generado.
- ✓ `javax.xml.transform.dom.DOMSource`. Clase que actuará de intermediaria entre el árbol DOM y el transformador, permitiendo al transformador acceder a la información del árbol DOM.
- ✓ `javax.xml.transform.stream.StreamResult`. Clase que actuará de intermediaria entre el transformador y el archivo o String donde se almacenará el documento XML generado.
- ✓ `java.io.File`. Clase que, como posiblemente sabrás, permite leer y escribir en un archivo almacenado en disco. El archivo será obviamente el documento XML que vamos a escribir en el disco.



Esto es un poco lioso, ¿o no? No lo es tanto cuando se ve un ejemplo de cómo realizar el proceso de transformación de árbol DOM a XML, veamos el ejemplo:

```
try {
    // 1º Creamos una instancia de la clase File para acceder al archivo donde guarda
    File f=new File(CaminoAlArchivoXML);
    //2º Creamos una nueva instancia del transformador a través de la fábrica de trar
    Transformer transformer = TransformerFactory.newInstance().newTransformer();
    //3º Establecemos algunas opciones de salida, como por ejemplo, la codificación c
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
    //4º Creamos el StreamResult, intermediria entre el transformador y el archivo de
    StreamResult result = new StreamResult(f);
    //5º Creamos el DOMSource, intermediaria entre el transformador y el árbol DOM.
    DOMSource source = new DOMSource(doc);
    //6º Realizamos la transformación.
    transformer.transform(source, result);
} catch (TransformerException ex) {
    System.out.println("¡Error! No se ha podido llevar a cabo la transformación.");
}
```

Debes conocer

A continuación te adjuntamos, cortesía de la casa, una de esas clases anti-estrés... utilízala todo lo que quieras, con ella podrás convertir un documento XML a árbol DOM y viceversa de forma sencilla. El documento XML puede estar almacenado en un archivo o en una cadena de texto (e incluso en Internet, para lo que necesitas el URI). Los métodos estáticos **DOM2XML** te permitirán pasar el árbol DOM a XML, y los métodos **String2DOM** y **XML2DOM** te permitirán pasar un documento XML a un árbol DOM.

También contiene el método **crearDOMVacio** que permite crear un árbol DOM vacío, útil para empezar un documento XML de cero.

[Clase de ayuda para convertir documentos XML a árboles DOM y viceversa.](#) (0.01 MB)

12.3.- Manipulación de documentos XML (I).

Bien, ahora sabes cargar un documento XML a DOM y de DOM a XML, pero, ¿cómo se modifica el árbol DOM? Como ya se dijo antes, un árbol DOM es una estructura en árbol, jerárquica como cabe esperar, formada por nodos de diferentes tipos. El funcionamiento del modelo de objetos DOM es establecido por el organismo W3C, lo cual tiene una gran ventaja, el modelo es prácticamente el mismo en todos los lenguajes de programación.



En Java, prácticamente todas las clases que vas a necesitar para manipular un árbol DOM están en el paquete **org.w3c.dom**. Si vas a hacer un uso muy intenso de DOM es conveniente que hagas una importación de todas las clases de este paquete ("**import org.w3c.dom.*;**").

Tras convertir un documento XML a DOM lo que obtenemos es una instancia de la clase **org.w3c.dom.Document**. Esta instancia será el nodo principal que contendrá en su interior toda la jerarquía del documento XML. Dentro de un documento o árbol DOM podremos encontrar los siguientes tipos de clases:

- ✓ **org.w3c.dom.Node** (Nodo). Todos los objetos contenidos en el árbol DOM son nodos. La clase **Document** es también un tipo de nodo, considerado el nodo principal.
- ✓ **org.w3c.dom.Element** (Elemento). Corresponde con cualquier par de etiquetas ("") y todo su contenido (atributos, texto, subetiquetas, etc.).
- ✓ **org.w3c.dom.Attr** (Atributo). Corresponde con cualquier atributo.
- ✓ **org.w3c.dom.Comment** (Comentario). Corresponde con un comentario.
- ✓ **org.w3c.dom.Text** (Texto). Corresponde con el texto que encontramos dentro de dos etiquetas.

¿A qué te eran familiares? Claro que sí. Estas clases tendrán diferentes métodos para acceder y manipular la información del árbol DOM. A continuación vamos a ver las operaciones más importantes sobre un árbol DOM. En todos los ejemplos, "**doc**" corresponde con una instancia de la clase **Document**.

Obtener el elemento raíz del documento.

Como ya sabes, los documentos XML deben tener obligatoriamente un único elemento ("**<pedido></pedido>**" por ejemplo), considerado el elemento raíz, dentro del cual está el resto de la información estructurada de forma jerárquica. Para obtener dicho elemento y poder manipularlo podemos usar el método **getDocumentElement**.

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

Buscar un elemento en toda la jerarquía del documento.

Para realizar esta operación se puede usar el método **getElementsByTagName** disponible tanto en la clase **Document** como en la clase **Element**. Dicha operación busca un elemento por el nombre de la etiqueta y retorna una lista de nodos (**NodeList**) que cumplen con la condición. Si se usa en la clase **Element**, solo buscará entre las subetiquetas (subelementos) de dicha clase (no en todo el documento).

```
NodeList nl=doc.getElementsByTagName("cliente");
Element cliente;
if (nl.getLength()==1) cliente=(Element)n2.item(0);
```

El método **getLength()** de la clase **NodeList**, permite obtener el número de elementos (longitud de la lista) encontrados cuyo nombre de etiqueta es coincidente. El método **item** permite acceder a cada uno de los elementos encontrados, y se le pasa por argumento el índice del elemento a obtener (empezando por cero y acabando por longitud menos uno). Fijate que es necesario hacer una conversión de tipos después de invocar el método **item**. Esto es porque la clase **NodeList** almacena un listado de nodos (**Node**), sin diferenciar el tipo.



Autoevaluación

Dado el siguiente código XML: "<pedido><!--Datos del pedido--></pedido>" indica que elementos DOM encontramos en el mismo.

- ☐ `org.w3c.dom.Element.`
- ☐ `org.w3c.dom.Node.`
- ☐ `org.w3c.dom.Text.`
- ☐ `org.w3c.dom.Comment.`

Mostrar Información

12.3.1.- Manipulación de documentos XML (II).

¿Y qué más operaciones puedo realizar sobre un árbol DOM? Veámoslas.

Obtener la lista de hijos de un elemento y procesarla.

Se trata de obtener una lista con los nodos hijo de un elemento cualquiera, estos pueden ser un sub-elemento (sub-etiqueta) o texto. Para sacar la lista de nodos hijo se puede usar el método **getChildNodes**:

```
NodeList nl=doc.getDocumentElement().getChildNodes();
for (int i=0; i<nl.getLength();i++) {
    Node n=nl.item(i);
    switch (n.getNodeType())
    {
        case Node.ELEMENT_NODE: Element e=(Element)n;
            System.out.println("Etiqueta:" + e.getTagName());
            break;
        case Node.TEXT_NODE: Text t=(Text)n;
            System.out.println("Texto:" + t.getWholeText());
            break;
    }
}
```



En el ejemplo anterior se usan varios métodos. El método **"getNodeTypes()"** de la clase **Node** permite saber de que tipo de nodo se trata, generalmente texto (**Node.TEXT_NODE**) o un sub-elemento (**Node.ELEMENT_NODE**). De esta forma podremos hacer la conversión de tipos adecuada y gestionar cada elemento según corresponda. También se usa el método **"getTagName"** aplicado a un elemento, lo cual permitirá obtener el nombre de la etiqueta, y el método **"getWholeText"** aplicado a un nodo de tipo texto (**Text**), que permite obtener el texto contenido en el nodo.

Añadir un nuevo elemento hijo a otro elemento.

Hemos visto como mirar que hay dentro de un documento XML pero no hemos visto como añadir cosas a dicho documento. Para añadir un sub-elemento o un texto a un árbol DOM, primero hay que crear los nodos correspondientes y después insertarlos en la posición que queramos. Para crear un nuevo par de etiquetas o elemento (**Element**) y un nuevo nodo texto (**Text**), lo podemos hacer de la siguiente forma:

```
Element dirTag=doc.createElement("Direccion_entrega")
Text dirTxt=doc.createTextNode("C/Perdida S/N");
```

Ahora los hemos creado, pero todavía no los hemos insertado en el documento. Para ello podemos hacerlo usando el método **appendChild** que añadirá el nodo (sea del tipo que sea) al final de la lista de hijos del elemento correspondiente:

```
dirTag.appendChild(dirTxt);
doc.getDocumentElement().appendChild(dirTag);
```

En el ejemplo anterior, el texto se añade como hijo de la etiqueta **"Direccion_entrega"**, y a su vez, la etiqueta **"Direccion_entrega"** se añade como hijo, al final del todo, de la etiqueta o elemento raíz del documento. Aparte del método **appendChild**, que siempre insertará al final, puedes utilizar los siguientes métodos para insertar nodos dentro de un árbol DOM (todos se usan sobre la clase **Element**):

- ✓ **insertBefore** (Node nuevo, Node referencia). Insertará un nodo nuevo antes del nodo de referencia.

✔ replaceChild (Node nuevo, Node anterior). Sustituye un nodo (anterior) por uno nuevo.

Debes conocer

En el siguiente enlace encontrarás a la documentación del API de DOM para Java, con todas las funciones de cada clase.

[API de DOM para Java.](#)

12.3.2.- Manipulación de documentos XML (III).

Seguimos con las operaciones sobre árboles DOM. ¿Sabrías cómo eliminar nodos de un árbol? ¿No? Vamos a descubrirlo.

Eliminar un elemento hijo de otro elemento.

Para eliminar un nodo, hay que recurrir al nodo padre de dicho nodo. En el nodo padre se invoca el método **removeChild**, al que se le pasa la instancia de la clase **Element** con el nodo a eliminar (no el nombre de la etiqueta, sino la instancia), lo cual implica que primero hay que buscar el nodo a eliminar, y después eliminarlo. Veamos un ejemplo:



```
NodeList n13=doc.getElementsByTagName("Direccion_entrega");
for (int i=0;i<n13.getLength();i++){
    Element e=(Element)n13.item(i);
    Element parent=(Element)e.getParentNode();
    parent.removeChild(e);
}
```

En el ejemplo anterior se eliminan todas las etiquetas, estén donde estén, que se llamen **"Direccion_entrega"**. Para ello ha sido necesario buscar todos los elementos cuya etiqueta sea esa (como se explicó en ejemplos anteriores), recorrer los resultados obtenidos de la búsqueda, obtener el nodo padre del hijo a través del método **getParentNode**, para así poder eliminar el nodo correspondiente con el método **removeChild**.

No es obligatorio obviamente invocar al método **"getParentNode"** si el nodo padre es conocido. Por ejemplo, si el nodo es un hijo del elemento o etiqueta raíz, hubiera bastado con poner lo siguiente:

```
doc.getDocumentElement().removeChild(e);
```

Cambiar el contenido de un elemento cuando solo es texto.

Los métodos **getTextContent** y **setTextContent**, aplicado a un elemento, permiten respectivamente acceder al texto contenido dentro de un elemento o etiqueta. Tienes que tener cuidado, porque utilizar **"setTextContent"** significa eliminar cualquier hijo (sub-elemento por ejemplo) que previamente tuviera la etiqueta. Ejemplo:

```
Element nuevo=doc.createElement("direccion_recogida").setTextContent("C/Del Medio S/N");
System.out.println(nuevo.getTextContent());
```



Autoevaluación

De los siguientes tipos de nodos, marca aquellos en los que no se pueda ejecutar dos veces el método **"appendChild"**:

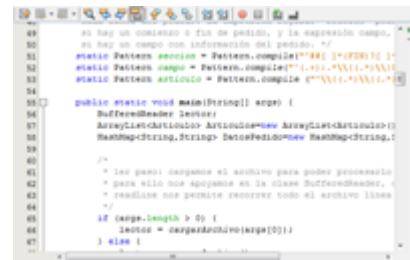
- ☐ **Document.**
- ☐ **Element.**
- ☐ **Text.**
- ☐ **Attr.**

Mostrar Información

12.3.3.- Manipulación de documentos XML (IV).

Caso práctico

El documento XML que tiene que generar Ana tiene que seguir un formato específico, para que la otra aplicación sea capaz de entenderlo. Esto significa que los nombres de las etiquetas tienen que ser unos concretos para cada dato del pedido. Esta ha sido quizás la parte que más complicada, ver como encajar cada dato del mapa con su correspondiente etiqueta en XML, pero ha conseguido resolverlo de forma elegante. De hecho, ya ha terminado su tarea, ¿quieres ver el resultado?



[Procesar archivo de pedido completo.](#) (0.01 MB)

Y ya solo queda una cosa, descubrir como manejar los atributos en un árbol DOM.

Atributos de un elemento.

Por último, lo más fácil. Cualquier elemento puede contener uno o varios atributos. Acceder a ellos es sencillísimo, solo necesitamos tres métodos: **setAttribute** para establecer o crear el valor de un atributo, **getAttribute** para obtener el valor de un atributo y **removeAttribute** para eliminar el valor de un atributo. Veamos un ejemplo:



```
doc.getDocumentElement().setAttribute("urgente", "no");
System.out.println(doc.getDocumentElement().getAttribute("urgente"));
```

En el ejemplo anterior se añade el atributo "urgente" al elemento raíz del documento con el valor "no", después se consulta para ver su valor. Obviamente se puede realizar en cualquier otro elemento que no sea el raíz. Para eliminar el atributo es tan sencillo como lo siguiente:

```
doc.getDocumentElement().removeAttribute("urgente");
```

Anexo I.- Formateado de cadenas en Java.

Sintaxis de las cadenas de formato y uso del método format.

En Java, el método estático **format** de la clase **String** permite formatear los datos que se muestran al usuario o la usuaria de la aplicación. El método **format** tiene los siguientes argumentos:

- ✓ Cadena de formato. Cadena que especifica cómo será el formato de salida, en ella se mezclará texto normal con especificadores de formato, que indicarán cómo se debe formatear los datos.
- ✓ Lista de argumentos. Variables que contienen los datos cuyos datos se formatearán. Tiene que haber tantos argumentos como especificadores de formato haya en la cadena de formato.

Los especificadores de formato comienzan siempre por "%", es lo que se denomina un carácter de escape (carácter que sirve para indicar que lo que hay a continuación no es texto normal, sino algo especial). El especificador de formato debe llevar como mínimo el símbolo "%" y un carácter que indica la conversión a realizar, por ejemplo "%d".

La conversión se indica con un simple carácter, y señala al método **format** cómo debe ser formateado el argumento. Dependiendo del tipo de dato podemos usar unas conversiones u otras. Veamos las conversiones más utilizadas:

Listado de conversiones más utilizada y ejemplos.

Tipo de conversión	Especificación de formato	Tipos de datos aplicables	Ejemplo
Valor lógico o booleano.	"%b" o "%B"	Boolean (cuando se usan otros tipos de datos siempre lo formateará escribiendo true).	<pre>boolean b=true; String d= String.format("Resultado: %b", b); System.out.println (d);</pre>
Cadena de caracteres.	"%s" o "%S"	Cualquiera, se convertirá el objeto a cadena si es posible (invocando el método toString).	<pre>String cad="hola mundo"; String d= String.format("Resultado: %s", cad); System.out.println (d);</pre>
Entero decimal	"%d"	Un tipo de dato entero.	<pre>int i=10; String d= String.format("Resultado: %d", i); System.out.println (d);</pre>
Número en notación científica	"%e" o "%E"	Flotantes simples o dobles.	<pre>double i=10.5; String d= String.format("Resultado: %E", i); System.out.println (d);</pre>
Número decimal	"%f"	Flotantes simples o dobles.	<pre>float i=10.5f;</pre>

Tipo de conversión	Especificación de formato	Tipos de datos aplicables	Ejemplo
			<pre>String d= String.format("Resultado: %f", i); System.out.println (d);</pre>
Número en notación científica o decimal (lo más corto)	"%g" o "%G"	Flotantes simples o dobles. El número se mostrará como decimal o en notación científica dependiendo de lo que sea mas corto.	<pre>double i=10.5; String d= String.format("Resultado: %g", i); System.out.println (d);</pre>

Ahora que ya hemos visto alguna de las conversiones existentes (las más importantes), veamos algunos modificadores que se le pueden aplicar a las conversiones, para ajustar como queremos que sea la salida. Los modificadores se sitúan entre el carácter de escape ("%") y la letra que indica el tipo de conversión (d, f, g, etc.).

Podemos especificar, por ejemplo, el número de caracteres que tendrá como mínimo la salida de una conversión. Si el dato mostrado no llega a ese ancho en caracteres, se rellenará con espacios (salvo que se especifique lo contrario):

`%[Ancho]Conversión`

El hecho de que esté entre corchetes significa que es opcional. Si queremos por ejemplo que la salida genere al menos 5 caracteres (poniendo espacios delante) podríamos ponerlo así:

```
String.format ("%5d",10);
```

Se mostrará el "10" pero también se añadirán 3 espacios delante para rellenar. Este tipo de modificador se puede usar con cualquier conversión.

Cuando se trata de conversiones de tipo numéricas con decimales, solo para tipos de datos que admitan decimales, podemos indicar también la precisión, que será el número de decimales mínimos que se mostrarán:

`%[Ancho][.Precisión]Conversión`

Como puedes ver, tanto el ancho como la precisión van entre corchetes, los corchetes no hay que ponerlos, solo indican que son modificaciones opcionales. Si queremos, por ejemplo, que la salida genere 3 decimales como mínimo, podremos ponerlo así:

```
String.format ("%%.3f",4.2f);
```

Como el número indicado como parámetro solo tiene un decimal, el resultado se completará con ceros por la derecha, generando una cadena como la siguiente: **"4,200"**.

Una cadena de formato puede contener varios especificadores de formato y varios argumentos. Veamos un ejemplo de una cadena con varios especificadores de formato:

```
String np="Lavadora";  
int u=10;  
float ppu = 302.4f;  
float p=u*ppu;  
String output=String.format("Producto: %s; Unidades: %d; Precio por unidad: %.2f €; Total: %.  
System.out.println(output);
```

Cuando el orden de los argumentos es un poco complicado, porque se reutilizan varias veces en la cadena de formato los mismos argumentos, se puede recurrir a los índices de argumento. Se trata de especificar la posición del argumento a utilizar, indicando la posición del argumento (el primer argumento sería el 1 y no el 0) seguido por el símbolo del dólar ("\$"). El índice se ubicaría al comienzo del especificador de formato, después del porcentaje, por ejemplo:

```
int i=10;  
int j=20;  
String d=String.format("%1$d multiplicado por %2$d (%1$dx%2$d) es %3$d",i,j,i*j);  
System.out.println(d);
```

El ejemplo anterior mostraría por pantalla la cadena **"10 multiplicado por 20 (10x20) es 200"**. Los índices de argumento se pueden usar con todas las conversiones, y es compatible con otros modificadores de formato (incluida la precisión).









Para saber más

Si quieres profundizar en los especificadores de formato puedes acceder a la siguiente página (en inglés), donde encontrarás información adicional acerca de la sintaxis de los especificadores de formato en Java:

[Sintaxis de los especificadores de formato.](#)

Anexo.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo

Recurso (1)	Datos del recurso (1)	Recurso (2)
	Autoría: Chealer. Licencia: GNU GPL. Procedencia:	
	Autoría: LatinStock. Licencia: Uso educativo para plataformas públicas de FPaD. Procedencia: LatinStock.	
	Autoría: LatinStock. Licencia: Uso educativo para plataformas públicas de FPaD. Procedencia: LatinStock.	
	Autoría: Ilustración de persona: AIGA symbol signs collection; trabajo derivado por kismalac. Licencia: CC-BY-SA. Procedencia: http://es.wikipedia.org/wiki/Archivo:PersonsSet.svg	
	Autoría: en:Joestape89. Licencia: CC-BY-SA. Procedencia:	
	Autoría: Esterab. Licencia: CC-BY-SA. Procedencia:	
	Autoría: Original photo: User:Fanghong; Derivative work: User:Gnomz007. Licencia: CC-BY-SA. Procedencia:	



Autoría: Booyabazooka.

Licencia: CC-BY-SA.

Procedencia:

http://commons.wikimedia.org/wiki/File:Rubik%27s_cube.svg