UNIDAD DIDÁCTICA 6

APLICACIÓN DE LAS ESTRUCTURAS DE ALMACENAMIENTO

MÓDULO PROFESIONAL: PROGRAMACIÓN

CESUR
Tu Centro Oficial de FP

Índice

RESUMEN INTRODUCTORIO

En esta unidad se va a tratar en primer lugar el concepto de array de una dimensión y de más de una dimensión. Posteriormente, se tratará el uso de la clase `String` en Java, como crearlas y algunos de los métodos más utilizados de esta clase.

A continuación, veremos el tratamiento de archivos XML en Java utilizando DOM y SAX. Finalizaremos revisando una de las estructuras dinámicas de datos más usadas como son las listas. Para cada uno de los apartados veremos ejemplos en el lenguaje de programación Java.

A continuación, seguiremos con el concepto de lista, para comprender como un programador podría implementarla y los tipos de listas ordinales existentes como son las pilas y colas.

Una vez se ha entendido el concepto de lista y su funcionamiento interno, conoceremos los distintos tipos de colecciones que ya hay implementadas en Java y que el desarrollador puede utilizar sin tener que crearlas desde cero. Entre las colecciones más importantes están las listas, los conjuntos y los mapas, cada uno de ellos tiene unas ventajas y características, por tanto, dependerá del tipo de aplicación y de las necesidades software, escoger una u otra.

Profundizaremos un poco sobre el concepto de genericidad en el momento de crear una colección, un método o una clase. Este concepto se refiere al uso de tipos genéricos o parámetros de tipo en la definición y uso de clases, interfaces y métodos.

Y, para finalizar, trataremos las expresiones regulares y su uso, ya que es importante conocerlas y saber cómo utilizarlas porque muchos lenguajes de programación las soportan.

INTRODUCCIÓN

En el mundo de la programación en Java, existen una variedad de elementos y técnicas que son fundamentales para el desarrollo de aplicaciones robustas y eficientes. Cada uno de estos elementos ofrece ventajas específicas y se adapta a diferentes situaciones, permitiendo a los desarrolladores abordar diversos desafíos de manera efectiva. En esta unidad, exploraremos algunas de las estructuras y conceptos más importantes que Java tiene para ofrecer y las razones por las cuales son cruciales en el desarrollo de software.

Una estructura de datos es una forma de organizar una serie de datos ya sean simples o compuestos. En Java tenemos estructuras de datos “estáticas” como pueden ser los arrays y estructuras de datos “dinámicas” como son las colecciones y los mapas. Dentro de las colecciones en Java también podemos encontrar las listas dinámicas.

Los arrays en Java son estructuras de datos esenciales que ofrecen eficiencia en el acceso y almacenamiento de elementos del mismo tipo. Aprenderás por qué son útiles, especialmente cuando se conoce de antemano el número exacto de elementos que se necesitarán.

En cuanto al tratamiento de los archivos XML en Java, nos presentan dos posibles formas de parsearlos (o interpretarlos) que son SAX y DOM. Java nos proporciona estas herramientas poderosas para el manejo de documentos XML y HTML. Exploraremos el DOM, que permite la manipulación completa de documentos, y SAX, que destaca por su eficiencia en el procesamiento de documentos XML grandes y su bajo consumo de memoria.

Para realizar el almacenamiento de cadenas de texto en Java, se pueden usar clases como `String`, `StringBuffer` o `StringBuilder`, y su funcionamiento no es tan simple.

Las listas dinámicas en Java, como ArrayList y LinkedList, son estructuras de datos flexibles que pueden crecer o reducirse dinámicamente. Descubrirás cómo aprovechar su flexibilidad y su capacidad de acceso eficiente por índice.

Las colecciones en Java proporcionan abstracciones de alto nivel, como List, Set y Map, que permiten trabajar con diferentes estructuras de datos de manera uniforme. Aprenderás a utilizar la estructura Map para almacenar y recuperar datos de manera eficaz utilizando pares clave-valor, donde se requiere una rápida búsqueda y recuperación de valores.

Otro aspecto muy importante a destacar en Java, es la genericidad, que nos permitirá escribir código genérico que proporciona seguridad de tipos en tiempo de compilación y promueve la reutilización de código en diferentes contextos.

Por último, conocerás cómo las expresiones regulares son herramientas versátiles para buscar patrones de texto y validar la estructura de cadenas, lo que es esencial para el procesamiento y validación de datos.

CASO INTRODUCTORIO

La aplicación que estáis desarrollando en la empresa dónde trabajas maneja una gran cantidad de información. Habéis analizado los requisitos y llegado a la conclusión de que antes de almacenar toda esta información en una base de datos, vais a utilizar algunas estructuras de datos para organizarla. Para ello, te piden no solo estructuras de datos, sino también algunos archivos con información en formato XML que tendrás que parsear.

Al finalizar el estudio de la unidad, sabrás aplicar las estructuras de almacenamiento de datos estáticas y dinámicas, serás capaz de programar en Java dichas estructuras y realizar operaciones básicas con ellas, conocerás el tratamiento que hace Java de XML y la forma de implementarlo, entenderás los mecanismos que ofrece Java para utilizar métodos genéricos, comprenderás la importancia del uso de tipos genéricos y sus restricciones y sabrás aplicar expresiones regulares en la búsqueda de patrones en cadenas de texto.

1. ARRAYS

Te acaban de asignar la categoría de Desarrollador Back-End y serás el encargado de todo lo que tenga que ver con el almacenamiento en la aplicación que tu equipo y tú estáis desarrollando en este momento.

Aunque la información se va a almacenar de forma persistente en una base de datos, también serán necesarias algunas estructuras de almacenamiento para almacenar datos temporales o intermedios que nos permitan guardar información que podamos acceder de forma rápida y sin conexión a la base de datos. Por tanto, la utilización de arrays sería una opción válida para un acceso más eficiente de recuperación de información que normalmente es estática, que no va a cambiar muy a menudo y que no es necesario registrar en la base de datos.

El array es una de las estructuras de datos más ampliamente utilizada por su flexibilidad para derivar en complejas estructuras de datos y su simplicidad. Empezaremos con una definición: *un **array** es una secuencia de elementos, donde cada **elemento** (un grupo de bytes de memoria que almacenan un único ítem de datos) se asocia con al menos un **índice** (entero no negativo).* Esta definición da lugar a cuatro puntos interesantes:

- Cada elemento ocupa el **mismo número de bytes**; el número exacto depende del tipo de datos del elemento.
- Todos los elementos son del **mismo tipo**.
- Se tiende a pensar que los elementos de un array ocupan **localizaciones de memoria consecutivas**. Cuando se vean s los arrays bidimensionales se descubrirá que **no siempre es así**.
- El número de índices asociados con cada elemento es la ***dimensión*** del array.

RECUERDA

Esta sección se enfoca exclusivamente en **arrays de una y dos dimensiones** porque los arrays de más dimensiones no se utilizan de forma tan frecuente.

ENLACE DE INTERÉS

Para ampliar información sobre qué es un array y cómo se utilizan vista este enlace, donde, además, encontrarás ejemplos en Java.

1.1 Arrays de una dimensión

El tipo de array más simple tiene una sola dimensión: cada elemento se asocia con un único índice. El lenguaje Java proporciona tres técnicas para crear un array de una dimensión:

- usar sólo un inicializador
- usar sólo la palabra clave new
- utilizar la palabra clave new con un inicializador

Utilizar sólo un Inicializador.

Utilizando un inicializador se puede utilizar cualquiera de estas dos sintaxis:

```
type variable_name '[' ']' [ '=' initializer ] ';'
type '[' ']' variable_name [ '=' initializer ] ';'
```

Donde el inicializador tiene la siguiente sintaxis:

'{' initial_value1 ',' initial_value2 ',' ... '}'

El siguiente fragmento ilustra cómo crear un array de animales:

```
String animales[] = { "Tigre", "Cebra", "Canguro" };
```

Utilizar sólo la Palabra Clave "new".

Utilizando la palabra clave new se puede utilizar cualquiera de estas dos sintaxis:

```
type variable_name '[' ']' '=' 'new' type '[' integer_expression ']' ';'
type '[' ']' variable_name '=' 'new' type '[' integer_expression ']' ';'
```

Para ambas sintaxis:

- **`variable_name`** especifica el nombre de la variable del array unidimensional.

- **type** especifica el tipo de cada elemento. Como la variable del array unidimensional contiene una referencia a un array unidimensional, el tipo es `type[]`.

- **La palabra clave new seguida por type** y seguida por `integer_expression` entre corchetes cuadrados ([]) especifica el número de elementos. `new` asigna la memoria para los elementos del array unidimensional y pone ceros en todos los bits de los bytes de cada elemento, lo que significa que cada elemento contiene un valor por defecto que se interpreta según su tipo.

- = asigna la referencia al array unidimensional a la variable `variable_name`.

RECUERDA

Los desarrolladores Java normalmente sitúan los corchetes cuadrados después del tipo (`int[] test_scores`) en vez de escribirlos después del nombre de la variable (`int test_scores[]`) cuando declaran una variable array. Mantener toda la información del tipo en un único lugar mejora la lectura del código.

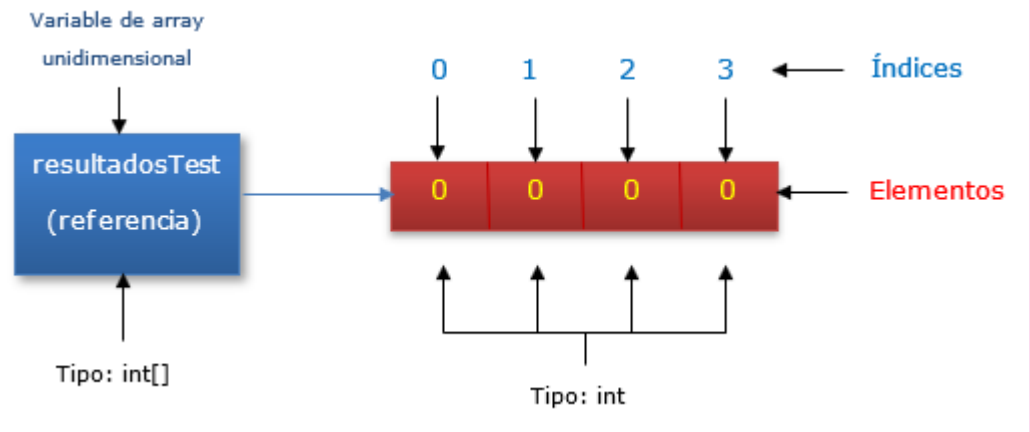
EJEMPLO PRÁCTICO

El siguiente fragmento de código utiliza sólo la palabra clave new para crear un array unidimensional que almacena datos de un tipo primitivo:

```
int[] resultadosTest = new int[4];
```

`int[] resultadosTest` declara una variable array unidimensional (`resultadosTest`) junto con su tipo de variable (`int[]`). El tipo de referencia `int[]` significa que cada elemento debe contener un ítem del tipo primitivo entero. **`new int[4]`** crea un array unidimensional asignando memoria para cuatro elementos enteros consecutivos. Cada elemento contiene un único entero y se inicializa a cero. El operador *igual a* (=) asigna la referencia del array unidimensional a `resultadosTest`.

La siguiente figura ilustra los elementos y la variable array unidimensional resultante:



RECUERDA

Cuando se crea un array unidimensional basado en un tipo primitivo, el compilador requiere que aparezca la palabra clave que indica el tipo primitivo en los dos lados del operador igual-a. De otro modo, el compilador lanzará un error. Por ejemplo

int[] resultadosTest= new long[20];

es ilegal porque las palabras claves int y long representan tipos primitivos incompatibles.

Los arrays unidimensionales de tipos primitivos almacenan datos que son valores primitivos. Por el contrario, los arrays unidimensionales del tipo referencia almacenan datos que son referencias a objetos.

EJEMPLO PRÁCTICO

El siguiente fragmento de código utiliza la palabra clave new para crear un par de arrays unidimensionales que almacenan datos basados en tipo referencia:

```
Reloj[] c1 = new Reloj[3];
Reloj[] c2 = new RelojAlarma[3];
```

`Reloj [] c1 = new Reloj [3];` declara una variable array unidimensional, `c1` de tipo `Reloj[]`, asigna memoria para un array unidimensional `Reloj` que consta de tres elementos consecutivos, y asigna la referencia del array `Reloj` a `c1`. Cada elemento debe contener una referencia a un objeto `Reloj` (asumiendo que Reloj es una clase concreta) o un objeto creado desde una subclase de Reloj y lo inicializa a null.

`Reloj [] c2 = new RelojAlarma [3];` se asemeja a la declaración anterior, excepto en que se crea un array unidimensional `RelojAlarma`, y su referencia se asigna a la variable `Reloj[]` de nombre `c2`. (Asume `RelojAlarma` como subclase de `Reloj`).

VÍDEO DE INTERÉS

En este vídeo se explica cómo utilizar los arrays en Java:

Utilizar la palabra clave "new" y un Inicializador.

Utilizar la palabra clave new con un inicializador requiere la utilización de alguna de las siguientes sintaxis:

```
type variable_name> '[' ']' '='  'new' type '[' ']' initializer
                                ';'
 type '[' ']' variable_name '='  'new' type '[' ']' initializer
                                ';'
```

Donde `initializer` tiene la siguiente sintaxis:

```
'{' [ initial_value [ ',' ... ] ] '}'
```

- **`variable_name`** especifica el nombre de la variable del array unidimensional

- **`type`** especifica el tipo de cada elemento. Como la variable del array unidimensional contiene una referencia a un array unidimensional, el tipo es `type[]`

- La palabra clave **`new`** seguida por `type` y seguida por corchetes cuadrados ([]) vacíos, seguido por `initializer`. No se necesita especificar el número de elementos entre los corchetes cuadrados porque el compilador cuenta el número de entradas en el inicializador. new asigna la memoria para los elementos del array unidimensional y asigna cada una de las entradas del inicializador a un elemento en orden de izquierda a derecha.

- **=** asigna la referencia al array unidimensional a la variable `variable_name`.

Un array unidimensional (o de más dimensiones) creado con la palabra clave new con un inicializador algunas veces es conocido como un array anónimo.

EJEMPLO PRÁCTICO

El siguiente fragmento de código utiliza la palabra clave new con un inicializador para crear un array unidimensional con datos basados en tipos primitivos:

```
int[] resultadosTest = new int[] { 70, 80, 20, 30 };
```

`int[] resultadosTest` declara una variable de array unidimensional `resultadosTest` junto con su tipo de variable (`int []`). El código **`new int[] { 70, 80, 20, 30 }`** crea un array unidimensional asignando memoria para cuatro elementos enteros consecutivos; y almacena `70` en el primer elemento, `80` en el segundo, `20` en el tercero, y `30` en el cuarto. La referencia del array unidimensional se asigna a `resultadosTest`.

RECUERDA

No especificar una expresión entera entre los corchetes cuadrados del lado derecho de la igualdad. De lo contrario, el compilador lanzará un error. Por ejemplo, new int[3] {70, 80, 20, 30} hace que el compilador lance un error, porque puede determinar el número de elementos partiendo del inicializador. Además, la discrepancia está entre el número 3 que hay en los corchetes y las cuatro entradas que hay en el inicializador.

La técnica de crear arrays unidimensionales con la palabra clave new y un inicializador también soporta la creación de arrays que contienen referencias a objetos.

EJEMPLO PRÁCTICO

El siguiente fragmento de código utiliza esta técnica para crear una pareja de arrays unidimensionales que almacenan datos del tipo referencia:

```
Reloj[] c1 = new Reloj[3] { new Reloj() };
Reloj[] c2 = new RelojAlarma[3] { new RelojAlarma() };
```

`Reloj[] c1 = new Reloj[3];` declara una variable de array unidimensional `c1` del tipo `Reloj[]`, asigna memoria para un array `Reloj` que consta de un solo elemento, crea un objeto `Reloj` y asigna su referencia a este elemento, y asigna la referencia del array `Reloj` a `c1`. El código **`Reloj[] c2 = new RelojAlarma[3];`** se parece a la declaración anterior, excepto en que crea un array unidimensional de un sólo elemento `RelojAlarma` que inicializa un objeto del tipo `RelojAlarma`.

1.1.1 Trabajar con un array unidimensional

Después de crear un array unidimensional, hay que almacenar y recuperar datos de sus elementos. Con la siguiente sintaxis se realiza esta tarea:

```
variable_name '[' integer_expression ']'
```

`integer_expression` identifica un índice de elemento y debe evaluarse como un entero entre 0 y uno menos que la longitud del array unidimensional (que devuelve `variable_name.length`). Un índice menor que 0 o mayor o igual que la longitud causa que se lance una excepción del tipo `ArrayIndexOutOfBoundsException`.

ENLACE DE INTERÉS

En este enlace se encuentra un resumen de arrays unidimensionales en Java

EJEMPLO PRÁCTICO

Este fragmento de código ilustra accesos legales e ilegales a un elemento:

```
public static void main(String[] args) {
      String [] meses = new String [] { "Ene", "Feb", "Mar", "Abr",
            "May", "Jun","Jul", "Ago", "Sep", "Oct", "Nov", "Dic"
      };
      System.out.println (meses [0]); // Salida: Ene
      // La siguiente instrucción provoca una excepción del tipo
      // ArrayIndexOutOfBoundsException porque el índice es mayor
      // que la longitud del array menos uno
      System.out.println (meses [meses.length]);
      System.out.println (meses [meses.length - 1]); // Salida: Dic
      // La siguiente instrucción provoca una excepción del tipo
      // ArrayIndexOutOfBoundsException porque el índice es < que 0
      System.out.println (meses [-1]);
}
```

Ocurre una situación interesante cuando se asigna la referencia de una subclase de un array a una variable de array de la superclase, porque un subtipo de array es una clase del supertipo de array. Si intenta asignar una referencia de un objeto de la superclase a los elementos del array de la subclase, se lanza una excepción del tipo `ArrayStoreException`.

EJEMPLO PRÁCTICO

Este fragmento demuestra esto:

```
public static void main(String[] args) {
        RelojAlarma[] ac = new RelojAlarma[1];
        Reloj[] c = ac;
        c[0] = new Reloj();
}
```

El compilador no dará ningún error porque todas las líneas son legales. Sin embargo, durante la ejecución, `c[0] = new Reloj();` resulta que lanza una excepción del tipo `ArrayStoreException`. Esta excepción ocurre porque se puede intentar acceder a un miembro específico de `RelojAlarma` mediante una referencia a un objeto `Reloj`.
Por ejemplo, supongamos que `RelojAlarma` contiene un método `public void suenaAlarma()`, `Reloj` no lo tiene, y el fragmento de código anterior se ejecuta sin lanzar una excepción del tipo `ArrayStoreException`. Un intento de ejecutar a `c[0].suenaAlarma();` bloquea la JVM *Máquina Virtual Java* porque estamos intentando ejecutar este método en el contexto de un objeto `Reloj` (que no incorpora un método `suenaAlarma()`).

Tenga cuidado cuando acceda a los elementos de un array, porque podría recibir una `ArrayIndexOutOfBoundsException` o una `ArrayStoreException`.

1.1.2 Algoritmo de búsqueda-lineal, búsqueda-binaria y ordenación de burbuja

Los desarrolladores normalmente escriben código para buscar datos en un array y para ordenar ese array. Hay tres algoritmos muy comunes que se utilizan para realizar estas tareas.

Búsqueda Lineal.

El algoritmo de búsqueda lineal busca en un array unidimensional un dato específico. La búsqueda primero examina el elemento con el índice 0 y continúa examinando los elementos sucesivos hasta que se encuentra el ítem o hasta que no quedan más elementos que examinar.

EJEMPLO PRÁCTICO

El siguiente código Java demuestra este algoritmo:

```
public static void main(String[] args) {
	int i=0;
	int buscado = 72;
	int x[] = {20, 15, 12, 30, -5, 72, 456};
	boolean encontrado=false;
	while (!encontrado) {
		if (x[i] == buscado) {
			System.out.println("Encontrado: " + x[i]);
			encontrado=true;
		}
		i++;
	}
	if (!encontrado)
		System.out.println("No encontrado: " + buscado);
}
```

La salida será:

```
Encontrado: 72
```

Dos de las ventajas de la búsqueda lineal son la **simplicidad** y la habilidad de buscar tanto arrays **ordenados** como **desordenados**. Su única **desventaja** es el **tiempo** empleado en examinar los elementos. El número medio de elementos examinados es la **mitad de la longitud del array**, y el máximo número de elementos a examinar es la longitud completa. Por ejemplo, un array unidimensional con 20 millones de elementos requiere que una búsqueda lineal examine una media de 10 millones de elementos y un máximo de 20 millones.

Como este tiempo de examen podría afectar seriamente al rendimiento, utilice la búsqueda lineal para arrays unidimensionales con relativamente **pocos elementos**.

ARTÍCULO DE INTERÉS

Lee este interesante artículo que te explica de manera clara y concisa la búsqueda lineal en arrays utilizando el lenguaje de programación Java. Se describen las condiciones, además de explicar los algoritmos antes de codificarlos en Java. Muy interesante para entender cómo funciona la búsqueda:

Búsqueda Binaria.

Al igual que en la búsqueda lineal, el algoritmo de búsqueda binaria busca un dato determinado en un array unidimensional. Sin embargo, al contrario que la búsqueda lineal, la búsqueda binaria **divide** el array en sección inferior y superior calculando el índice central del array. Si el dato se encuentra en ese elemento, la búsqueda binaria termina. Si el dato es numéricamente menor que el dato del elemento central, la búsqueda binaria calcula el índice central de la mitad inferior del array, ignorando la sección superior y repite el proceso. La búsqueda continúa hasta que se encuentre el dato o se exceda el límite de la sección (lo que indica que el dato **no existe** en el array).

EJEMPLO PRÁCTICO

Este código representa el equivalente Java del pseudocódigo anterior:

```
public static void main(String[] args) {
      int[] x = { -5, 12, 15, 20, 30, 72, 456 };
      int indiceInferior = 0;
      int indiceSuperior = x.length-1;
      int indiceMedio;
      int buscado = 72;
      boolean fin=false;
      while ((indiceInferior <= indiceSuperior) && (!fin)) {
         indiceMedio = (indiceInferior + indiceSuperior) / 2;
         if (buscado > x[indiceMedio])
             indiceInferior = indiceMedio + 1;
         else if (buscado < x[indiceMedio])
             indiceSuperior = indiceMedio - 1;
         else
              fin=true;
      }
      if (indiceInferior>indiceSuperior)
        System.out.println (buscado + " no encontrado");
      else
        System.out.println (buscado + " encontrado");
}
```

La **única ventaja** de la búsqueda binaria es que **reduce el tiempo** empleado en examinar elementos. El número máximo de elementos a examinar es $\log_2 n$ (donde n es la longitud del array unidimensional). Por ejemplo, un array unidimensional con 1.048.576 elementos requiere que la búsqueda binaria examine un máximo de 20 elementos. La búsqueda binaria tiene dos **inconvenientes**; el **incremento de complejidad** y la necesidad de **preordenar** el array.

VÍDEO DE INTERÉS

Otro de los algoritmos utilizados para la búsqueda sería el binario. Amplia información sobre ello en este vídeo:

Ordenación de Burbuja.

Cuando entra en juego la ordenación de datos, la ordenación de burbuja es uno de los algoritmos **más simples**. Este algoritmo hace varios pases sobre un array unidimensional. Por cada pase, el algoritmo compara datos adyacentes para determinar si numéricamente es mayor o menor. Si el dato es mayor (para ordenaciones ascendentes) o menor (para ordenaciones descendientes) los datos se intercambian y se baja de nuevo por el array. En el último pase, el dato mayor (o menor) se ha movido al final del array. Este efecto "burbuja" es el origen de su nombre.

La siguiente figura muestra una ordenación de burbuja ascendente de un array unidimensional de cuatro elementos. Hay tres pasos, el paso 0 realiza tres comparaciones y dos intercambios, el paso 1 realiza dos comparaciones y un intercambio y el paso 2 realiza una comparación y un intercambio.

Pass 0:					Pass 1:					Pass 2:				
	14	12	82	-3		12	14	-3	82		12	-3	14	82
	12	14	82	-3		12	14	-3	82		-3	12	14	82
	12	14	82	-3		12	-3	14	82					
	12	14	-3	82										

EJEMPLO PRÁCTICO

En este listado se presenta el algoritmo programado en Java:

```
public static void main(String[] args) {
	int i, pass;
	int[] x={ 20, 15, 12, 30, -5, 72, 456 };
	for (pass=0;pass<=x.length-2;pass++)
		for (i=0;i<=x.length-pass-2;i++)
			if (x[i]>x[i + 1]) {
				int temp=x[i];
				x[i]=x[i+1];
				x[i+1]=temp;
			}
	for (i=0;i<x.length;i++)
		System.out.println(x[i]);
```

Aunque la ordenación de burbuja es uno de los algoritmos de ordenación más simples, también es uno de los más **lentos**. Entre los algoritmos más **rápidos** se incluyen la **ordenación rápida** y la **ordenación de pila**.

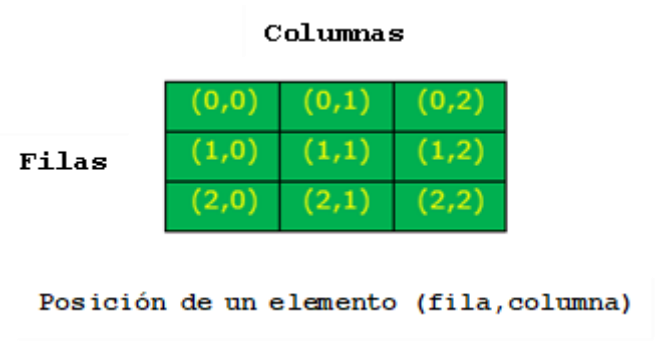
¿SABÍAS QUE...?

Otro algoritmo muy utilizado para arrays unidimensionales copia los elementos de un array fuente en otro array de destino. En vez de escribir su propio código para realizar esta terea puede utilizar el método `public static void array copy(Object src, int srcindex, Object dst, int dstindex, int length)` de la clase `java.lang.System`, que es la forma más rápida de realizar la copia.

1.2 Arrays de dos dimensiones

Un array de dos dimensiones, también conocido como ***tabla*** o ***matriz,*** donde cada elemento se asocia con una **pareja de índices**, es otro array simple. Se conceptualiza un array bidimensional como una **cuadrícula rectangular** de elementos divididos en **filas** y **columnas**, y se utiliza la notación `(fila, columna)` para identificar a un elemento específico.

La siguiente figura ilustra esta visión conceptual y la notación específica de los elementos:



Java proporciona tres técnicas para crear un array bidimensional:

Utilizar sólo un Inicializador.

Esta técnica requiere una de estas sintaxis:

```
type variable_name '[' ']' '[' ']' '='   '{' [ rowInitializer [ ',' ... ] ] '}' ';'
type '[' ']' '[' ']' variable_name '='  '{' [ rowInitializer [ ',' ... ] ] '}' ';'
```

Donde *rowInitializer* tiene la siguiente sintaxis:

```
'{' [ initial_value [ ',' ... ] ] '}'
```

Para ambas sintaxis:

- ***variable_name*** especifica el nombre de la variable del array bidimensional.

- ***type*** especifica el tipo de cada elemento. Como una variable de array bidimensional contiene una referencia a un array bidimensional, su tipo es `type[ ][ ]`.

- Especifica cero o más inicializadores de filas entre los corchetes `({ })`. Si no hay inicializadores de filas, el array bidimensional está vacío. Cada inicializador de fila especifica **cero o más valores iniciales** para las entradas de las columnas de esa fila. Si no se especifican valores para esa fila, la fila está **vacía**.

- `=` se utiliza para asignar la referencia del array bidimensional a `variable_name`.

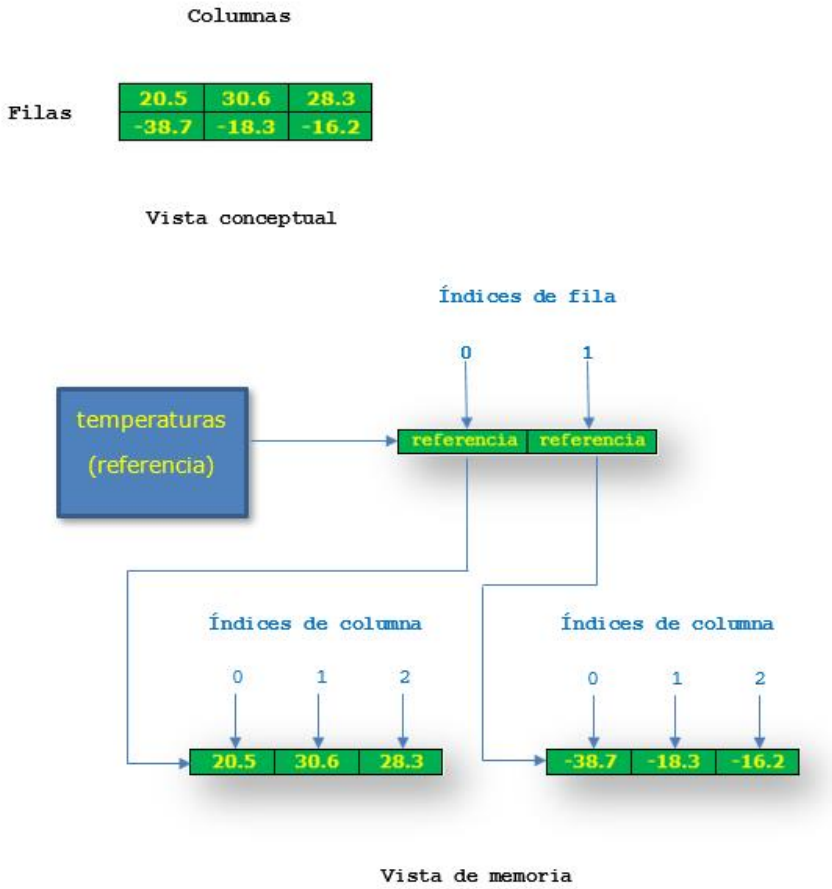
EJEMPLO PRÁCTICO

En este código se usa sólo un inicializador para crear un array bidimensional que almacena datos basados en un tipo primitivo:

```
   double[][] temperaturas = {{20.5,30.6,28.3},
{-38.7,-18.3,-16.2 }}; // Temperaturas Celsius
```

`double[][] temperaturas` declara una variable de array bidimensional (`temperaturas`) junto con su tipo de variable (`double[][]`). El tipo de referencia `double[][]` significa que cada elemento debe contener datos del tipo primitivo `double`. **`{{20.5,30.6,28.3},{-38.7,-18.3,-16.2 }}`** especifica un array bidimensional de **dos filas** por **tres columnas**, donde la primera fila contiene los datos `20.5`, `30.6`, y `28.3`, y la segunda fila contiene los datos `-38.7`, `-18.3`, y `-16.2`. El operador igual-a (=) asigna la referencia del array bidimensional a `temperaturas`.

La siguiente figura ilustra el array bidimensional resultante desde un punto de vista conceptual y de memoria.



Utilizar sólo la palabra clave "new"

Esta técnica requiere cualquiera de estas sintaxis:

```
type variable_name '[' ']' '[' ']' '=' 'new' type '[' integer_expression ']' '[' ']'
                                                ';'
type '[' ']' '[' ']' variable_name '=' 'new' type '[' integer_expression ']' '[' ']'
                                                ';'
```

En ambas sintaxis:

- ***variable_name*** especifica el nombre de la variable del array bidimensional.

- ***type*** especifica el tipo de cada elemento. Como es una variable de array bidimensional contiene una referencia a un array bidimensional, su tipo es `type[ ][ ]`.

- La palabra clave **new seguida por type** y por una **expresión entera entre corchetes cuadrados**, que identifica el número de filas. La instrucción new asigna memoria para las filas del array unidimensional de filas y pone a cero todos los bytes de cada elemento, lo que significa que cada elemento contiene una referencia nula. Debe crear un array unidimensional de columnas separado y asignarle su referencia cada elemento fila.

- = se utiliza para asignar la referencia del array bidimensional a `variable_name`.

EJEMPLO PRÁCTICO

En este fragmento de código se usa sólo la palabra clave new para crear un array bidimensional que almacena datos basados en un tipo primitivo:

```
public static void main(String[] args) {
      double[][] temperaturas = new double[2][];
      // La matriz tiene 2 filas
      temperaturas[0]=new double[3];
      // La primera fila tiene 3 columnas
      temperaturas[1]=new double[3];
      // La segunda fila tiene 3 columnas
      temperaturas[0][0]=20.5; // Se asignan valores para la fila 0
      temperaturas[0][1]=30.6;
      temperaturas[0][2]=28.3;
      temperaturas[1][0]=-38.7; //Se asignan valores para la fila 1
      temperaturas[1][1]=-18.3;
      temperaturas[1][2]=-16.2;
}
```

Utilizar la palabra clave "new" y un inicializador.

Esta técnica requiere una de estas sintaxis:

```
type variable_name '[' ']' '[' ']' '='
    'new' type '[' ']' '[' ']' '{' [ rowInitializer [ ',' ... ] ] '}' ';'

type '[' ']' '[' ']' variable_name '='
    'new' type '[' ']' '[' ']' '{' [ rowInitializer [ ',' ... ] ] '}' ';'
```

Donde *rowInitializer* tiene la siguiente sintaxis:

```
'{' [ initial_value [ ',' ... ] ] '}'
```

En ambas sintaxis:

- ***variable_name*** especifica el nombre de la variable del array bidimensional.

- ***type*** especifica el tipo de cada elemento. Como es una variable de array bidimensional contiene una referencia a un array bidimensional, su tipo es `type[ ][ ]`.

- La palabra clave ***new*** seguida por ***type*** y por dos parejas de corchetes cuadrados vacíos, y cero o más inicializadores de filas entre un par de corchetes cuadrados. Si no se especifica ningún inicializador de fila, el array bidimensional está vacío. Cada inicializador de fila especifica cero o más valores iniciales para las columnas de esa fila.

- `=` se utiliza para asignar la referencia del array bidimensional a `variable_name`.

EJEMPLO PRÁCTICO

En este fragmento de código se usa la palabra clave new y un inicializador para crear un array bidimensional que almacena datos basados en un tipo primitivo:

```
double[ ][ ] temperaturas = new double[ ][ ] { {20.5,30.6,28.3},
       {-38.7,-18.3,-16.2 } };
```

VÍDEO DE INTERÉS

Los arrays multidimensionales son más difíciles de entender. Visualiza en este vídeo cómo trabajar con matrices en la práctica:

1.2.1 Trabajar con arrays bidimensionales

Después de crear un array bidimensional, se querrá almacenar y recuperar datos de sus elementos. Se puede realizar esta tarea con la siguiente sintaxis:

```
variable_name '[' integer_expression1 ']' '[' integer_expression2 ']'
```

integer_expression1 identifica el índice de fila del elemento y va de cero hasta la longitud del array menos uno. Igual ocurre con `integer_expression2`, pero para el índice de columna. Como todas las filas tienen la misma longitud, se puede encontrar conveniente especificar `variable_name[0].length` para especificar el número de columnas de cualquier fila.

EJEMPLO PRÁCTICO

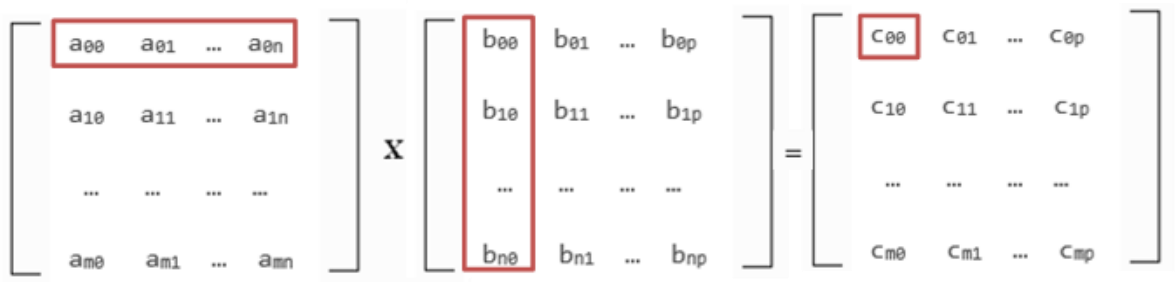
Este fragmento de código almacena y recupera datos de un elemento de un array bidimensional:

```
	public static void main(String[] args) {
		double[][] temperaturas = { {20.5,30.6,28.3},
					    {-38.7,-18.3,-16.2} };
		temperaturas[0][1] = 18.3; // Reemplaza 30.6 por 18.3
		System.out.println(temperaturas[1][2]); // Salida: -16.2
	}
}
```

1.2.2 Algoritmo de multiplicación de matrices

Multiplicar una matriz por otra es una operación común en el trabajo con gráficos, con datos económicos, o con datos industriales. Los desarrolladores normalmente utilizan el algoritmo de multiplicación de matrices para completar esa multiplicación. ¿Cómo funciona ese algoritmo? Dejemos que 'A' represente una matriz con 'm' filas y 'n' columnas. De forma similar, 'B' representa un matriz con 'p' filas y 'n' columnas.
Multiplicar A por B produce una matriz C obtenida de multiplicar todas las entradas de A por su correspondencia en B.

La siguiente figura ilustra estas operaciones:



$$c_{00} = a_{00}b_{00} + a_{01}b_{10} + \ldots + a_{0n}b_{n0}$$

RECUERDA

La multiplicación de matrices requiere que el **número de columnas** de la matriz de la izquierda (A) sea igual al **número de filas** de la matriz de la derecha (B). Por ejemplo, para multiplicar una matriz A de cuatro columnas por fila por una matriz B (como en A x B), B debe contener exactamente cuatro filas.

ENLACE DE INTERÉS

Para entender cómo multiplicar matrices en Java es preciso tener claro en qué consiste este proceso matemáticamente hablando. Consulta este enlace para entenderlo mejor:

EJEMPLO PRÁCTICO

Este método realiza la multiplicación de matrices en Java:

```
static int[][] multiplicaMatrices(int[][] a, int[][] b) {
    // Si el nº de columnas de la matriz a es distinto
    // del nº de filas de la matriz b nos muestra un error
    if (a[0].length != b.length) {
    System.err.println("El nº de columnas de la matriz a es \r\n" +
                       " distinto del nº de filas de la matriz b ");
          return null;
    }
    int[][] result = new int[a.length][];
    for (int i=0; i<result.length;i++)
          result[i]=new int[b[0].length];
    // Realiza la multiplicación y la suma
    for (int i=0;i<a.length;i++)
          for (int j=0;j<b[0].length;j++)
                for (int k=0;k<a[0].length;k++)
                      result[i][j] += a[i][k] * b[k][j];
    // Devuelve la matriz resultante
    return result;
}
```

Se requieren tres bucles FOR para realizar la multiplicación. El bucle más interno multiplica un elemento de la fila de la matriz a por un elemento de la columna de la matriz b y añade el resultado a una sola entrada de la matriz `result`.

2. CLASE STRING: REPRESENTANDO UNA CADENA

En la aplicación en la que te encuentras trabajando, la clase String va a desempeñar un papel crucial en la comunicación con los usuarios, ya que los mensajes entre este y la interfaz serán objetos de tipo String combinados con información dinámica. Será una herramienta esencial para representar y manipular texto en la pantalla, así como para almacenar datos textuales necesarios en diferentes funcionalidades que se estén desarrollando en la aplicación.

Una cadena de texto no deja de ser más que la sucesión de un conjunto de caracteres alfanuméricos, signos de puntuación y espacios en blanco con más o menos sentido.

Se puede encontrar desde la archiconocida cadena “Hola Mundo” y la no menos “Mi primera cadena de texto”, pasando por las cadenas de texto personalizadas “Pepe”, “Pepe López”, ... hasta las inclasificables “asdf” o incluso tener una cadena vacía, con espacios en blanco o saltos de línea.

Todas ellas serán representadas en java con la clase **String.** Para encontrar la clase **String** dentro de las librerías de Java tendremos que ir a **java.lang.String.** Cuando hacemos referencia a String no nos referimos a un tipo primitivo sino a una clase que dispone de unos métodos que podremos usar para manipular cadenas de texto.

ENLACE DE INTERÉS

Conoce más información sobre la clase String:

RECUERDA

El valor de una variable de tipo String siempre debe ir entre comillas dobles.

Podemos decir que un String o cadena es un array unidimensional cuyos elementos son caracteres. Por tanto, es posible manipular cadenas como simples arrays unidimensionales de caracteres. Pero lo común es hacerlo mediante la clase String

EJEMPLO PRÁCTICO

```
Ej: public class ArrayDeCaracteres {

	public static void main(String[] args) {
		char cadena[] = new char[4];
		cadena[0] = 'H';
		cadena[1] = 'o';
		cadena[2] = 'l';
		cadena[3] = 'a';
		System.out.println(cadena);
	}
}
```

La salida por pantalla sería:

Hola

2.1 Creación de una cadena

Para crear una cadena existen dos opciones:

Crear explícitamente.

Instanciar o ejemplarizar la clase String, que sería una **creación explicita** de la clase

```
String sMiCadena = new String("Cadena de Texto");
```

Crear implícitamente la cadena de texto. Java crea un objeto de la clase String, automáticamente y le asigna el valor al objeto.

```
String sMiCadena = "Cadena de Texto";
```

En este caso, Java, creará un objeto String llamado `sMiCadena` para tratar esta cadena. Esta forma es la más eficiente, ya que si lo hacemos de forma explícita (utilizando new) se realiza la llamada al constructor dos veces.

RECUERDA

String es una clase, no es un tipo primitivo y la primera letra va en mayúsculas.

2.2 Crear una cadena vacía

Se puede tener la necesidad de crear una cadena vacía. Puede darse el caso de que no siempre se sepa lo que se va a poner de antemano en la cadena de texto. Muchas veces la cadena de texto la proporcionará el usuario, otro sistema, ...

Para poder crear la cadena vacía bastará con asignarle el valor "", o bien, utilizar el constructor vacío.

```
String sMiCadena = "";
String sMiCadena = new String();
```

Constructores String.

Se puede resumir que se tienen dos tipos de constructores principales de la clase String:

- **`String()`**, que construirá un objeto `String` sin inicializar.
- **`String(String original)`**, construye una clase `String` con otra clase `String` que recibirá como argumento.

Volcado de una cadena de texto a la consola.

Solo quedará saber cómo volcar una cadena por pantalla. Esto se hará con la clase `System.out.println` que recibirá como parámetro el objeto `String`.

Por ejemplo:

```
System.out.println("Mi Cadena de Texto");
```

o

```
String miCadena = new String("Mi Cadena de Texto");
```

```
System.out.println(miCadena);
```

Para concatenar cadenas concatenar cadenas de forma simple podemos usar el operador + .

EJEMPLO PRÁCTICO

En este ejemplo puedes ver como concatenar de forma simple varias cadenas de texto.

```
public class ConcatenacionStringConOperador {

	public static void main(String[] args) {
		String cadena2 = "me llamo ";
		String nombre = "Maricarmen";

		String resultado = "Hola " + cadena2 + nombre;
	System.out.println(resultado);
	}
}
```

El resultado en pantalla sería:

Hola me llamo Maricarmen.

Los espacios entre las cadenas se pueden introducir bien, dentro de la cadena o bien concatenando con una cadena que contenga los espacios que se deseen, como se muestra a continuación.

```
String resultado = "Hola" + " " + cadena2 + nombre;
System.out.println(resultado);
```

El resultado en pantalla al ejecutar la aplicación sería el mismo que en el ejemplo práctico anterior:

Hola me llamo Maricarmen

También se podrían concatenar cadenas con otro tipo de dato primitivo pero el resultado sería una nueva cadena de tipo texto.

Si tenemos tres variables num1, num2 y suma que son de tipo entero y la variable suma almacena la suma de los números contenidos en la variable num1 y num2. Al concatenar

la variable `suma` con la cadena de texto "El resultado es: "se obtendrá una nueva cadena de texto con el valor "El resultado es: 35", por este motivo la variable `resultado` es declarada como variable de tipo `String`.

```
public class ConcatenacionStringNumeroConOperador {
	public static void main(String[] args) {
		int num1 = 25;
		int num2 = 10;
		int suma = num1 + num2;
		String resultado = "El resultado es: " + suma;
		System.out.println(resultado);
	}
}
```

Otro caso puede ser cuando no tenemos una variable que almacena la suma y queremos que se realice la operación antes de la concatenación. Fíjate en el siguiente ejemplo.

```
public class ConcatenacionStringNumeroConOperador2 {
	public static void main(String[] args) {
		int num1 = 25;
		int num2 = 10;
		int suma = num1 + num2;
		String resultado = "El resultado es: " + num1 + num2;
		System.out.println(resultado);
	}
}
```

Ahora la salida sería un resultado incorrecto:

El resultado es: 2510

Para conseguir que se sume antes de la concatenación habría que añadir paréntesis.

La instrucción quedaría de la siguiente forma:

```
String resultado = "El resultado es: " + (num1 + num2);
```

Ahora la salida sería un resultado correcto:

El resultado es: 35

RECUERDA

Si al usar el operador **+**, alguno de los operandos es una cadena entonces en vez de actuar como operador aritmético actúa como operador de concatenación dando lugar a una nueva cadena de texto.

NOTA DE INTERÉS

Prueba el último ejemplo, pero ahora multiplica num1 y num2 de la siguiente manera:

```
String resultado = "El resultado es: " + num1 * num2;
```

¿Dará error de ejecución? ¿Por qué crees que habrá error o no? Prueba también a realizar otras operaciones aritméticas.

La **inmutabilidad** es una propiedad importante de los objetos de tipo `String` en Java. Significa que una vez que se crea un objeto String, su contenido no puede cambiar, esto implica que cualquier operación que modifique un `String`, como por ejemplo una concatenación, reemplazo o conversión, etc. en realidad crea un nuevo objeto String en lugar de modificar el original.

Esto puede ocasionar gran cantidad de objetos temporales en memoria gestionados por el recolector de basura que podría afectar al rendimiento.

En estos casos, es recomendable considerar el uso de la clase `StringBuilder` o `StringBuffer` para crear y manipular cadenas que cambian constantemente(mutables). La diferencia entre ambos es que `StringBuffer` tiene todos sus métodos sincronizados para "soporte de escritura concurrente", mientras que `StringBuilder`(que complementa a la anterior y ha aparecido en nuevas versiones de Java) no los tiene. Por tanto, StringBuilder es más eficiente siempre que no se requiera trabajar con múltiples hilos(threads) en Java.

EJEMPLO PRÁCTICO

En este ejemplo puedes ver cómo crear una cadena con la clase `StringBuilder` y añadir a esta cadena otra cadena nueva. Si quieres probar este ejemplo con la clase `StringBuffer` solo tienes que cambiar el nombre de la clase.

```
public class EjemploStringBuilder {

	public static void main(String[] args) {
		StringBuilder stringBuilder = new StringBuilder();

		stringBuilder.append("Hola, ");
		stringBuilder.append("¿cómo estás?");

		String resultado = stringBuilder.toString();

		System.out.println(resultado);
		// Imprime "Hola, ¿cómo estás?"
	}
}
```


ENLACE DE INTERÉS

Uno de los puntos más difíciles de la programación Java es la gestión de cadenas. En este enlace podrás obtener más información sobre las operaciones con cadenas en Java

2.3 Métodos más usados de la clase String

Se van a ver los métodos que permiten manipular una cadena de texto:
Información básica de la cadena.

- **`length()`**- Devuelve el tamaño que tiene la cadena.
- **`char charAt(int index)`**- Devuelve el carácter indicado como índice. El primer carácter de la cadena será el del índice 0. Junto con el método **`length()`** se pueden recuperar todos los caracteres de la cadena de texto.

Hay que tener cuidado. Ya que si se intenta acceder a un índice de un carácter que no existe nos devolverá una excepción del tipo `IndexOutOfBoundsException`.

EJEMPLO PRÁCTICO

Crea una cadena y que se muestre en pantalla su longitud, el primer carácter y el último de la misma. Ten en cuenta que los espacios dentro de una cadena cuentan como un carácter más.

```
public class MetodosDeString1
{
          public static void main(String[] args) {
              String frase = "Hola, mundo!";

              // Obtener la longitud de la cadena
              int longitud = frase.length();
              System.out.println("Longitud de la cadena: " +
                                                 longitud);

              // Acceder a caracteres individuales usando charAt()
              char primerCaracter = frase.charAt(0);
              char ultimoCaracter = frase.charAt(longitud - 1);

              // Mostrar primer y último caracter
              System.out.println("Primer caracter: " +
                                                 primerCaracter);
              System.out.println("Último caracter: " +
                                                 ultimoCaracter);
          }
      }
```

La salida por pantalla será:
Longitud de la cadena: 12
Primer carácter: H
Último carácter: !

Comparación de Cadenas.

Los métodos de comparación sirven para comparar si dos cadenas de texto son iguales o no. Dentro de los métodos de comparación están los siguientes:

`boolean equals(Object anObject)` - Permite comparar si dos cadenas de texto son iguales. En el caso de que sean iguales devolverá como valor "true", y en caso contrario devolverá "false".

Este método tiene en cuenta si los caracteres van en mayúsculas o en minúsculas. Si se quiere omitir esta validación se tienen dos opciones. La primera de las opciones es convertir las cadenas a mayúsculas o minúsculas con los métodos **`toUpperCase()`** y **`toLowerCase()`** respectivamente.

La segunda opción es utilizar el método **`equalsIgnoreCase()`** que omite si el carácter está en mayúsculas o en minúsculas.

EJEMPLO PRÁCTICO

En este ejemplo se comparan tres cadenas usando el método `equals`.
Este método distingue entre mayúsculas y minúsculas por tanto "Hola" no será igual a "hola".

```
public class MetodosDeStringEquals {
	public static void main(String[] args) {
		String cadena1 = "Hola";
		String cadena2 = "hola";
		String cadena3 = "Hola";

		// Comparación de cadenas usando equals()

		// false
		boolean primeraComparacion = cadena1.equals(cadena2);

		// true
		boolean segundaComparacion = cadena1.equals(cadena3);

		System.out.println("¿Es cadena1 igual a cadena2? "
					+ primeraComparacion);
		System.out.println("¿Es cadena1 igual a cadena3? "
					+ segundaComparacion);
	}
}
```

La salida por pantalla será:

¿Es cadena1 igual a cadena2? false
¿Es cadena1 igual a cadena3? true

`boolean equalsIgnoreCase(String anotherString)` - Compara dos cadenas de caracteres omitiendo si los caracteres están en mayúsculas o en minúsculas.

`int compareTo(String anotherString)` - Este método es un poco más avanzado que el anterior, el cual, solo indicaba si las cadenas eran iguales o diferentes. En este caso se comparan las cadenas léxicamente. Para ello se basa en el valor Unicode de los caracteres. Se devuelve un **entero menor de 0** si la cadena sobre la que se parte es léxicamente **menor** que la cadena pasada como argumento. Si las dos cadenas son **iguales** léxicamente se devuelve un **0**. Si la cadena es **mayor** que la pasada como argumento se devuelve un **número entero positivo**. Pero qué significa "mayor, menor o igual léxicamente".

Para describirlo se verá con un pequeño ejemplo en Java:

```
String s1 = "Cuervo";
String s2 = "Cuenca";
s1.compareTo(s2);
```

Se comparan las dos cadenas. Los tres primeros caracteres son iguales "Cue". Cuando el método llega al carácter de índice 3 (4º carácter) tiene que comparar entre la r minúscula y la n minúscula. Si utiliza el código Unicode llegará a la siguiente conclusión.

```
r(114) > n(110)
```

Y devolverá la resta de sus valores. En este caso un 4.

Se debe tener cuidado, porque este método no tiene en cuenta las mayúsculas y minúsculas. Y dichos caracteres, aun siendo iguales, tienen diferentes códigos. En la siguiente comparación:

```
String s1 = "CueRvo";
String s2 = "Cuervo";
s1.compareTo(s2);
```

Nuevamente los tres caracteres iniciales son iguales. Pero el cuarto es distinto. Por un lado, se tiene la r minúscula y por otro la r mayúscula. Así:

```
R(82) < r(114)
```

`int compareToIgnoreCase(String str)` - Este método se comportará igual que el anterior, pero ignorando las mayúsculas. Todo un alivio por si se nos escapa algún carácter en mayúsculas.

Búsqueda de caracteres.

Se tiene un conjunto de métodos que permiten buscar caracteres dentro de cadenas de texto. Y es que no hay que olvidar que la cadena de caracteres no es más que eso: una suma de caracteres.

`int indexOf(int ch)` - Devuelve la posición de la primera ocurrencia de un carácter dentro de la cadena de texto. En el caso de que el carácter buscado no exista devolverá -1. Si lo encuentra devuelve un número entero con la posición que ocupa en la cadena.

`int indexOf(int ch, int fromIndex)` - Realiza la misma operación que el anterior método, pero en vez de hacerlo a lo largo de toda la cadena lo hace desde el índice (fromIndex) que se le indique.

`int lastIndexOf(int ch)` - Indica cuál es la última posición que ocupa un carácter dentro de una cadena. Si el carácter no está en la cadena devuelve -1.

`int lastIndexOf(int ch, int fromIndex)` - Lo mismo que el anterior, pero a partir de una posición indicada como argumento.

EJEMPLO PRÁCTICO

Aquí tienes un ejemplo más completo en el cual se puede ver cómo tratar el valor devuelto por el método `compareTo()`.

```
public class MetodosDeStringCompareTo {
       public static void main(String[] args) {
         String s1 = "CueRvo";
         String s2 = "Cuervo";

         // Comparar s1 con s2 usando compareTo()
         int comparacion = s1.compareTo(s2);

         if (comparacion == 0) {
             System.out.println("Las cadenas s1 y s2 son iguales.");
         } else if (comparacion < 0) {
             System.out.println("La cadena s1 viene antes
                                 que la cadena s2 en orden
                                 lexicográfico.");
         } else {
             System.out.println("La cadena s1 viene después que
                                la cadena s2 en orden
                                 lexicográfico.");
         }
     }
}
```

La salida por pantalla será:

La cadena s1 viene antes que la cadena s2 en orden lexicográfico.

EJEMPLO PRÁCTICO

Realiza un programa que muestre la longitud de una cadena, la posición de cada una de sus letras y busque la primera ocurrencia de un determinado carácter a partir de una posición. Por último, que muestre la última ocurrencia de un carácter dentro de la cadena.

```
public class MetodosDeString2 {
	public static void main(String[] args) {

		String frase = "Hola, cómo estás? Estoy bien, gracias.";
		System.out.println("Longitud de la cadena: "
							+ frase.length()
							+ "\n"
							);

		/*mostrar en pantalla el índice de cada letra dentro de
		la cadena*/
		for (int i = 0; i < frase.length(); i++) {
		char caracter = frase.charAt(i);
		System.out.println("Carácter #" + i + ": " + caracter);
	 }

		/* Buscar la primera ocurrencia de la letra 'o' después
		del índice 10*/
		int primeraOcurrenciaDespuesDe = frase.indexOf('o', 10);
		System.out.println("\nPrimera ocurrencia de 'o' después
				del índice 10: " + primeraOcurrenciaDespuesDe);

		// Buscar la última ocurrencia de la letra 'e'
		int ultimaOcurrenciaDeE = frase.lastIndexOf('e');
		System.out.println("Última ocurrencia de 'e': "
							+ ultimaOcurrenciaDeE);
	}
}
```

La salida por pantalla será:

Longitud de la cadena: 38

Carácter #0: H
Carácter #1: o
Carácter #2: l
…
…
…
Carácter #36: s

```
Carácter #37: .

Primera ocurrencia de 'o' después del índice 10: 21
Última ocurrencia de 'e': 26
```

Búsqueda de subcadenas.

Este conjunto de métodos es, probablemente, el más utilizado para el manejo de cadenas de caracteres. Van a permitir buscar cadenas dentro de cadenas, así como conocer la posición en la que se encuentran en la cadena origen para poder acceder a la subcadena.

Dentro de este conjunto se encuentran:

`int indexOf(String str)` - Busca la primera ocurrencia de una cadena dentro de la cadena origen. Devuelve un entero con el índice a partir del cual está la cadena localizada. Si no encuentra la cadena devuelve un -1.

`int indexOf(String str, int fromIndex)` - Misma funcionalidad que `indexOf(String str)`, pero a partir de un índice indicado como argumento del método.

`int lastIndexOf(String str)` - Si la cadena que se busca se repite varias veces en la cadena origen, se puede utilizar este método que indicará el índice donde empieza la última repetición de la cadena buscada.

`lastIndexOf(String str, int fromIndex)` - Lo mismo que el anterior, pero a partir de un índice pasado como argumento.

`boolean startsWith(String prefix)` – Probablemente se haya encontrado con el problema de saber si una cadena de texto empieza con un texto específico. La verdad es que este método se podía obviar y utilizar el método `indexOf()`, con el cual, en el caso de que devolviese un 0, se sabe que es el inicio de la cadena.

`boolean startsWith(String prefix, int toffset)` - Más elaborado que el anterior, y quizás, con un poco menos de significado que el anterior.

`boolean endsWith(String suffix)` – Para resolver el problema de conocer si una cadena de texto acaba con otra. De igual manera que sucedía con el método `startsWith()` se puede utilizar una mezcla entre los métodos `indexOf()` y `length()` para reproducir el comportamiento de `endsWith()`.

EJEMPLO PRÁCTICO

Practica el ejemplo anterior, pero buscando una subcadena en vez de buscar un solo carácter y prueba también con el método `starsWith`.

```
public class MetodosDeString3 {

	public static void main(String[] args) {
		String frase = "what you see is what you get";

		System.out.println("Longitud de la cadena: " +
					frase.length() + "\n");

		/*mostrar en pantalla el índice de cada letra dentro de
		la cadena*/
		for (int i = 0; i < frase.length(); i++) {
		  char caracter = frase.charAt(i);
		  System.out.println("Carácter #" + i + ": "
					+ caracter);
		   }
		// Usando indexOf para buscar "what"
		int primeraOcurrencia = frase.indexOf("what");
		System.out.println("\nPrimera ocurrencia de \"what\": "
					+ primeraOcurrencia);

		// Usando lastIndexOf para buscar "what"
		int ultimaOcurrencia = frase.lastIndexOf("what");
		System.out.println("Última ocurrencia de \"what\": "
					+ ultimaOcurrencia);

		/* Usando startsWith para verificar si la frase
		  comienza con "Java" */
		boolean comienzaCon = frase.startsWith("what");
		System.out.println("\n¿La frase comienza con \"what\"? "
					+ comienzaCon);
	}
}
```

La salida por pantalla será:

```
Longitud de la cadena: 28

Carácter #0: w
Carácter #1: h
Carácter #2: a
Carácter #3: t
```

```
Carácter #4:
…
…
…
Carácter #25: g
Carácter #26: e
Carácter #27: t

Primera ocurrencia de "what": 0
Última ocurrencia de "what": 16

¿La frase comienza con "what"? true
```


ENLACE DE INTERÉS

En este enlace se puede ver un tutorial sobre la gestión de cadenas en C, que es otro lenguaje muy utilizado.

Métodos con subcadenas.

Ahora que se sabe cómo localizar una cadena dentro de otra se va a ver cómo se substrae de donde está.

`String substring(int beginIndex)` - Este método devolverá la cadena que se encuentra entre el índice pasado como argumento (`beginIndex`) y el final de la cadena origen. Así, si se tiene la siguiente cadena:

```
String s = "Víctor Cuervo";
```

El método…

```
s.substring(7);
```

Devolverá “Cuervo”.

`String substring(int beginIndex, int endIndex)` - Si se da el caso de que la cadena que se desea recuperar no llega hasta el final de la cadena origen, que será lo normal, se puede utilizar este método indicando el índice inicial y final del cual queremos obtener la cadena. Se debe tener en cuenta que el valor proporcionado a `endIndex` se interpretará como `endIndex-1`.

Así, si se parte de la cadena...

```
String s = "En un lugar de la mancha....";
```

El método...

```
s.substring(6,11);
```

Devolverá la palabra "lugar", ya que tiene en cuenta desde el carácter que se encuentra en la posición 6 hasta el carácter de la posición 10 (=11-1).

¿SABÍAS QUE...?

Los métodos de la clase String se pueden utilizar directamente sobre literales, es decir, con cadenas entre comillas.
Ej: "Esto es una cadena de texto".length()

3. TRATAMIENTO DE XML EN JAVA (LECTURA Y ESCRITURA)

En la última reunión con tu cliente, te ha comunicado la necesidad de realizar informes personalizados por lo que planteas al equipo la utilización de una API DOM o SAX para que se puedan obtener dichos informes basados en plantillas XML. Utilizando una API de este tipo las plantillas puedan ser cargadas y reemplazadas las variables para generar los informes finales. Además, debido a que muchos servicios web utilizan XML para intercambiar datos, las API DOM y SAX pueden ayudarte a procesar las respuestas XML de los servicios web y preparar solicitudes XML para enviarlas a estos servicios.

En este apartado se explicará cómo procesar ficheros en formato XML, las API más importantes que se usan en Java y las diferencias básicas entre ellas. Estas dos API van a ser **DOM** y **SAX**.

Los ficheros XML se usan básicamente para tratar datos, ya sea para estructurarlos, para enviar y recibir datos o para utilizarlos como base de datos. La principal idea de los ficheros XML es que son portables, e independientes del lenguaje de programación que usemos para procesarlos, además de ser simples de editar a mano y fáciles de comprender.

Existen dos formas de procesar los ficheros XML en Java, básicamente. Por una parte, se tiene el modelo **DOM** (Document Object Model) y por otra parte el modelo **SAX** (Simple API for XML).

ENLACE DE INTERÉS

Amplia información sobre las librerías para el tratamiento de XML en Java:

3.1 DOM

A la hora de procesar un documento XML con DOM, la representación que se tiene va a ser la de un árbol jerárquico en memoria. Esto implica varias cosas que se detallan a continuación:

- Se puede leer **cualquier parte del árbol** (todo es un nodo), de forma que se puede procesar de arriba a abajo pero también se puede volver atrás.
- Se puede **modificar cualquier nodo** del árbol.
- Al estar cargado en **memoria**, se puede tener una **falta** de ésta. Con ficheros XML pequeños no se tienen problemas, pero si el árbol es muy grande entonces se tendrá una falta de `'heap space'`.

Comentadas las características de DOM, se va a explicar cómo se van a procesar documentos con DOM. El árbol en memoria va a ser un `Document`. Para crear un objeto de esta clase se usarán las factorías de `Document`, de la siguiente manera:

```
try {
	DocumentBuilderFactory dbf =
				DocumentBuilderFactory.newInstance();
	DocumentBuilder db = dbf.newDocumentBuilder();
	Document doc = db.newDocument();
} catch (ParserConfigurationException e) {
	e.printStackTrace();
}

// Forma compacta
try {
	Document doc = DocumentBuilderFactory.newInstance()
				.newDocumentBuilder().newDocument();
} catch (ParserConfigurationException e) {
	e.printStackTrace();
}
```

Se tendrá un `Document` en memoria que representa al árbol del que saldrá el fichero XML. Sin embargo, este árbol no tiene ni siquiera un nodo raíz, así que el siguiente paso es crearlo:

```
Element root = doc.createElement("Raiz");
```

Se da por hecho que la variable `doc` ya existe porque se ha creado en el paso previo. El parámetro `String` que recibe la función `createElement()` es el texto de la etiqueta. Cada vez que se quiera crear un nuevo nodo, se deberá llamar a esta función.

En caso de querer añadir el texto correspondiente a un nodo, se usará la función `createTextElement()`, que recibe un `String` que será el texto que contenga.

```
Element nodo = doc.createElement("NombreElemento");
Text texto = doc.createTextNode("Texto del elemento");
```

Existe una cosa más en los ficheros XML, y son los **atributos**. Un **nodo** (una etiqueta), puede tener una serie de atributos a los cuales se les asigna nombre y valor, o puede no tener ninguno. De esta forma se podría agregar al nodo raíz el atributo `autor` con valor `elAutor`, de la siguiente forma:

```
root.setAttribute("autor", "yoAutor");
```

En cualquiera de los casos, en DOM **todo es un nodo**, de modo que la forma de agregar nodos es la misma, independientemente del tipo de nodo que sea. Sabiendo esto, solamente faltaría agregar cada nodo a su nodo padre:

```
nodo.appendChild(texto);
root.appendChild(nodo);
doc.appendChild(root);
```

Con esto hemos conseguido tener un `Document` creado y cargado en memoria. Sin embargo, habría que darle formato para posteriormente escribirlo en algún lugar. En este caso se escribe en un fichero de texto:

```
// Meta @ elAutor
try {
	// Se vuelca el XML al fichero
	TransformerFactory transFact =
				TransformerFactory.newInstance();
	// Se añade el sangrado y la cabecera de XML
	transFact.setAttribute("indent-number", new Integer(3));
	Transformer trans = transFact.newTransformer();
	trans.setOutputProperty(OutputKeys.INDENT, "yes");
	trans.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
		// Se hace la transformación
		StringWriter sw = new StringWriter();
		StreamResult sr = new StreamResult(sw);
		DOMSource domSource = new DOMSource(dom);
		trans.transform(domSource, sr);
	} catch(Exception ex) {
		ex.printStackTrace();
	}
```

Esto deja en la variable `sw` la representación XML de este documento, con su sangrado correspondiente y listo para ser tratado. Ahora se podría escribir por pantalla, por

fichero, mandarlo por un Socket… en este caso se agrega un poco de código para escribirlo en un fichero:

```
// Meta @ elAutor
try {
	// Se crea el fichero para escribir en modo texto
	PrintWriter writer = new PrintWriter(new
                                        FileWriter("test.xml"));
	// Se escribe todo el árbol en el fichero
	writer.println(sw.toString());
	// Se cierra el fichero
	writer.close();
} catch (IOException e) {
	e.printStackTrace();
}
```

Teniendo esto se habrá conseguido un fichero de texto XML a partir de un árbol cuya representación será de la siguiente forma:

```
<Raiz autor="elAutor">
	<NombreElemento>Texto del elemento</NombreElemento>
</Raiz>
```

Por otra parte, si se quiere leer un fichero XML y procesarlo de alguna manera, primero se necesita crear un `Document` en memoria a partir de un fichero XML bien formado:

```
// Meta @ elAutor
try {
	DocumentBuilderFactory dbf =
                        DocumentBuilderFactory.newInstance();
	DocumentBuilder db = dbf.newDocumentBuilder();
	Document doc = db.parse(new File("test.xml"));
	doc.getDocumentElement().normalize();
} catch (ParserConfigurationException e) {
	e.printStackTrace();
} catch (SAXException e) {
	e.printStackTrace();
} catch (IOException e) {
	e.printStackTrace();
}
```

Es importante tener en cuenta que se puede crear el `Document` a partir de una sola línea de código, con todos los constructores anidados.

Si se tiene un fichero XML mal formado se recibirá una excepción. La función `normalize()` **elimina** nodos de texto vacíos y combina los adyacentes (en caso de que los hubiera).

Para poder acceder al nodo raíz del documento, se va a utilizar la función **`getDocumentElement()`** del `Document`. Y a partir de aquí se puede empezar a recorrer el árbol. Cuando se tiene un nodo, por ejemplo, el nodo raíz, se pueden obtener todos los nodos que cuelgan de él con la función **`getChildNodes()`**. Sin embargo, esta función puede dar lugar a errores, y se va a ver en un ejemplo.

Se tiene el fichero generado en la parte anterior, y se lee con el código que se acaba de poner. Si se pidieran los hijos de nuestro nodo raíz, se esperaría obtener sólo un nodo, que en este caso será el nodo con nombre `NombreElemento`. Pero lo obtenido es diferente:

```
System.out.println(doc.getDocumentElement().getChildNodes().getLength());
```

// salida: 3

Se ve qué nodos son estos:

```
NodeList nodosRaiz = doc.getDocumentElement().getChildNodes();
for(int i = 0; i < nodosRaiz.getLength(); i++) {
      System.out.println(nodosRaiz.item(i).getNodeName());
}
```

Además, si se miran esos nodos de tipo `#text` se verá que no son nada. De modo que para evitarlos se pueden filtrar así:

```
if(!nodosRaiz.item(i).getNodeName().equals("#text")…
```

o también se podrían seleccionar los nodos que se deseen por el nombre, puesto que se sabe el nombre de los nodos de cada nivel:

```
NodeList nodosRaiz =
      doc.getDocumentElement().getElementsByTagName("NombreElemento");
System.out.println(nodosRaiz.getLength());
System.out.println(nodosRaiz.item(0).getNodeName());
```

Es decir, la manera de ir leyendo los diferentes hijos de cada nodo es usando o bien **`getChildNodes()`** e ir filtrando por nombres (con un `case`, por ejemplo), o bien usar varios **`getElementsByTagName()`** en caso de tener etiquetas de nombres diferentes en el mismo nivel. Hay que recordar que el nodo raíz tenía un atributo.

Se supone que se quiere sacar el valor por alguna razón. El código sería el siguiente:

```
System.out.println(doc.getDocumentElement().getAttribute("autor"));
```

En caso de no ser el nodo raíz no se pondría **`doc.getDocumentElement()`** sino el `Element` correspondiente.

EJEMPLO PRÁCTICO

Aquí puedes ver un ejemplo completo de.Archivo XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<Pedidos>
   <Pedido numeroPedido="1">
      <Cliente>
         <Nombre>Thomas</Nombre>
         <Apellido>Anderson</Apellido>
         <User>Neo</User>
      </Cliente>
      <Productos>
         <Producto>
            <Nombre>Matrix 7.0</Nombre>
            <FechaAlta>1307542390468</FechaAlta>
         </Producto>
         <Producto>
            <Nombre>Gravedad 2.0</Nombre>
            <FechaAlta>1307542390468</FechaAlta>
         </Producto>
      </Productos>
   </Pedido>
   <Pedido numeroPedido="2">
      <Cliente>
         <Nombre>Mr.</Nombre>
         <Apellido>Regan</Apellido>
         <User>Cifra</User>
      </Cliente>
      <Productos>
         <Producto>
            <Nombre>Reinsercion 1.0</Nombre>
            <FechaAlta>1307542390468</FechaAlta>
         </Producto>
      </Productos>
   </Pedido>
 </Pedidos>
```

EJEMPLO PRÁCTICO

Aquí puedes ver un ejemplo completo de Listado Clase Java

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.util.ArrayList;
import java.util.Date;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;
import org.xml.sax.SAXException;

public class PedidosXML_DOM {

        private Document dom;

        public PedidosXML_DOM() {
                dom = null;
        }

        private void addCliente(Usuario cliente, Element nodoCliente) {
                Element nombre = dom.createElement("Nombre");
                Text textoNombre =
                dom.createTextNode(cliente.getNombre());
                nombre.appendChild(textoNombre);

                Element apellido = dom.createElement("Apellido");
                Text textoApellido =

                dom.createTextNode(cliente.getApellido());
                apellido.appendChild(textoApellido);
```

```
            Element user = dom.createElement("User");
            Text textoUser = dom.createTextNode(cliente.getUser());
            user.appendChild(textoUser);

            nodoCliente.appendChild(nombre);
            nodoCliente.appendChild(apellido);
            nodoCliente.appendChild(user);
    }


    private void addProducto(Producto producto, Element nodoProducto) {
            Element nombre = dom.createElement("Nombre");
            Text textoNombre =
            dom.createTextNode(producto.getNombre());
            nombre.appendChild(textoNombre);

            Element fechaAlta = dom.createElement("FechaAlta");
            Text textoFechaAlta =
            dom.createTextNode(String.
               valueOf(producto.getFechaAlta().getTime()));
            fechaAlta.appendChild(textoFechaAlta);

            nodoProducto.appendChild(nombre);
            nodoProducto.appendChild(fechaAlta);
    }

    private void addProductos(ArrayList<Producto> listaProductos,
                    Element nodoListaProductos) {
            for(int i = 0; i < listaProductos.size(); i++) {
                    Element producto = dom.createElement("Producto");
                    addProducto(listaProductos.get(i), producto);
                    nodoListaProductos.appendChild(producto);
            }
    }

    private void addPedido(Pedido pedido) {
            // seleccionamos la raiz
            Element root = dom.getDocumentElement();

            // Se crea un nuevo elemento con el atributo del
            // número de producto
            Element unPedido = dom.createElement("Pedido");
            unPedido.setAttribute("numeroPedido",
                            String.valueOf(pedido.getNumeroPedido()));

            // Se crea un nuevo elemento para el cliente
            Element cliente = dom.createElement("Cliente");
            addCliente(pedido.getCliente(), cliente);

            // Se crea un nuevo elemento para los productos
            // de los que consta el pedido
```

```
Element productos = dom.createElement("Productos");
addProductos(pedido.getListaProductos(), productos);

// Se inserta el cliente y los productos
// en el elemento del pedido
unPedido.appendChild(cliente);
unPedido.appendChild(productos);

// Se inserta el pedido en la raíz
root.appendChild(unPedido);
/*
 * Escribe en el fichero la representación del árbol XML
 */
private void write(StringWriter sw, String path) {
        try {
                // Se crea un fichero para escribir en
                //modo texto
                PrintWriter writer = new PrintWriter(
                                new FileWriter(path));

                // Se escribe todo el árbol en XML
                writer.println(sw.toString());

                // Se cierra el fichero
                writer.close();
        } catch (IOException e) {
                e.printStackTrace();
        }
}

/*Transforma el árbol, agregando la cabecera
  y añadiendo sangrados*/
private void toFile(String ruta) {
        try {
                // Se vuelca el XML al fichero
                TransformerFactory transFact =
                   TransformerFactory.newInstance();

                // Se añade el sangrado
                transFact.setAttribute("indent-number",
                                new Integer(3));
                Transformer trans =
                     transFact.newTransformer();
                /* Se incluye la cabecera XML
                y el sangrado*/
                trans.setOutputProperty(
                   OutputKeys.OMIT_XML_DECLARATION, "no");
                trans.setOutputProperty(OutputKeys.INDENT,
                                                "yes");

                // Se hace la transformación
                StringWriter sw = new StringWriter();
```

```
                StreamResult sr = new StreamResult(sw);
                DOMSource domSource = new DOMSource(dom);
                trans.transform(domSource, sr);

                // Se escribe en el fichero
                write(sw, ruta);
            } catch(Exception ex) {
                ex.printStackTrace();

            }
        }
        /*
         * Se crea un fichero XML a partir de una lista de pedidos
         */
        public void pedidosToXML(ArrayList<Pedido> pedidos,
                                 String ruta) {
            // Se crea un nuevo Document donde se va a guardar
            // toda la estructura
            try {
                dom =
                  DocumentBuilderFactory.newInstance().
                     newDocumentBuilder().newDocument();
                Element root =
                dom.createElement("Pedidos");
                dom.appendChild(root);
            } catch (ParserConfigurationException e) {
                e.printStackTrace();
            }

            for(int i = 0; i < pedidos.size(); i++)
                addPedido(pedidos.get(i));

            // Se vuelca por pantalla
            toFile(ruta);
        }

        // Lectura de XML con DOM
        private Usuario readUsuario(Node nodoUsuario) {
            Element elementoUsuario = (Element)nodoUsuario;
            String nombre = elementoUsuario.
                 getElementsByTagName("Nombre").item(0).
                   getTextContent();
            String apellido = elementoUsuario.
                  getElementsByTagName("Apellido").item(0).
                     getTextContent();
            String user = elementoUsuario.
                  getElementsByTagName("User").item(0).
                    getTextContent();

            return new Usuario(nombre, apellido, user);
        }
```

```
                private Producto readProducto(Node nodoProducto) {
                        Element elementoProducto = (Element)nodoProducto;
                        String nombre = elementoProducto.
                             getElementsByTagName("Nombre").item(0).
                                     getTextContent();
                        long fechaAlta = Long.valueOf(elementoProducto.
                            getElementsByTagName("FechaAlta").item(0).
                                 getTextContent());

                        return new Producto(nombre, new Date(fechaAlta));
                }
                private Pedido readPedido(Node nodoPedido) {
                        Element elementoPedido = (Element)nodoPedido;

                        int numeroPedido = Integer.valueOf(

elementoPedido.getAttribute("numeroPedido"));
                        Usuario user = readUsuario(elementoPedido.

getElementsByTagName("Cliente").item(0));
                        Pedido pedido = new Pedido(numeroPedido, user);

                        NodeList productos = elementoPedido.

getElementsByTagName("Productos");
                        for(int i = 0; i < productos.getLength(); i++) {
                                pedido.addProducto(readProducto

((Element)productos.item(0)));
                        }
                        return pedido;
                }

                /*
                 * Se crea una lista de pedidos procesando un fichero XML
                 */
                public ArrayList<Pedido> XMLtoPedidos(String ruta) {
                        ArrayList<Pedido> pedidos = new
ArrayList<Pedido>();

                        try {
                                DocumentBuilderFactory dbf =

DocumentBuilderFactory.newInstance();
                                DocumentBuilder db =
dbf.newDocumentBuilder();
                                dom = db.parse(new File(ruta));
                                dom.getDocumentElement().normalize();

                                // Se seleccionan todos los pedidos y se
van leyendo
```

```
                                NodeList listaPedidos =
dom.getDocumentElement().

getElementsByTagName("Pedido");
                                for(int i = 0; i <
listaPedidos.getLength(); i++) {

pedidos.add(readPedido(listaPedidos.item(i)));
                                }
                        } catch (SAXException e) {
                                e.printStackTrace();
                        } catch (IOException e) {
                                e.printStackTrace();
                        } catch (ParserConfigurationException e) {
                                e.printStackTrace();
                        }

                        return pedidos;
                }
        }
```

3.2 SAX

Al contrario que con DOM, al procesar en SAX **no se va a tener la representación completa del árbol en memoria**, pues SAX funciona con **eventos**. Esto implica:

- Al no tener el árbol completo **no se puede volver atrás**, pues se va leyendo secuencialmente.
- La **modificación** de un nodo es mucho **más compleja** (y la **inserción** de nuevos nodos).
- Como no se tiene el árbol en memoria es mucho más **`memory friendly`**, de modo que es la **única opción viable** para casos de **ficheros muy grandes**, pero demasiado complejo para ficheros pequeños.
- Al ser orientado a eventos, el **procesado** se vuelve **bastante complejo**.

De esta forma, no se va a explicar cómo escribir o modificar ficheros con SAX debido a su complejidad, sino cómo leerlos y procesarlos.

ENLACE DE INTERÉS

En este enlace encontrarás librerías para el tratamiento de XML en Java:

PARA SABER MÁS

Conoce más sobre como leer un XML con SAX:

Se va a partir de que se tiene ya el fichero XML anterior:

```
<Raiz autor=" ">
      <NombreElemento>Texto del elemento</NombreElemento>
</Raiz>
```

Para poder procesar un fichero XML con SAX la clase lectora va a necesitar heredar de la clase `DefaultHandler` (se recuerda que, como buena práctica de programación, los datos deben estar separados de la entrada/salida). Además, se va a necesitar un objeto de la clase `XMLReader`, el cual va a usar la propia clase como `ContentHandler` y `ErrorHandler`. El esqueleto de la clase entonces sería algo así:

```
public class LectorXML_SAX extends DefaultHandler {
      private XMLReader reader;
      public LectorXML_SAX() {
            try {
                  reader = XMLReaderFactory.createXMLReader();
```

```
                reader.setContentHandler(this);
                reader.setErrorHandler(this);
            } catch (SAXException e) {
                e.printStackTrace();
            }
        }
    }
```

Se ha dicho que SAX funciona por eventos. Pero ¿qué eventos son esos?

- **`startDocument()`**: llamado cuando empieza el documento.
- **`endDocument()`**: llamado cuando acaba el documento.
- **`startElement()`:** llamado cuando empieza un nodo (por ejemplo, al llegar al < en <Raiz>).
- **`characters()`:** llamado al acabar el evento **`startElement()`**. Sirve para leer el contenido de una etiqueta (por ejemplo, el texto Texto del elemento de la etiqueta `NombreElemento`).
- **`endElement()`:** llamado al llegar al final de una etiqueta (por ejemplo, al llegar al </ en `</NombreElemento>`).

ARTÍCULO DE INTERÉS

En este artículo se puede ver una comparativa de las tecnologías para XML de Java:

Ahora se creará el método para la lectura, de la siguiente manera:

```
    public void leeXML(String path) {
        try {
            reader.parse(path);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (SAXException e) {
            e.printStackTrace();
        }
    }
```

Al ejecutar esto se puede observar que no se obtiene nada. Esto es así porque las funciones explicadas más arriba para los eventos están vacías y se necesita redefinirlas. En este caso se va a volcar el contenido del fichero por pantalla, sin sangrado:

```
	@Override
	public void startElement(String uri, String localName, String name,
Attributes atts) {
		System.out.println("<" + localName + ">");
	}

	@Override
	public void characters(char[] cadena, int inicio, int length) {
	    if(String.valueOf(cadena, inicio, length).trim().length() != 0)
		System.out.println(String.valueOf(cadena, inicio, length));
	}

	@Override
	public void endElement(String uri, String name, String qName) {
		System.out.println("");
	}
```

Si se crease un objeto de esta clase y se invocase al método **`leerXML()`**, se obtendría lo siguiente por la consola:

```
<Raiz><NombreElemento>Texto del elemento</NombreElemento></Raiz>
```

A partir de esto se podría, en lugar de escribir por pantalla los valores, procesarlos y crear atributos de objetos de clases que se hayan definido.

EJEMPLO PRÁCTICO

Visualiza un ejemplo completo de. Archivo XML:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<Pedidos>
   <Pedido numeroPedido="1">
      <Cliente>
         <Nombre>Thomas</Nombre>
         <Apellido>Anderson</Apellido>
         <User>Neo</User>
      </Cliente>
      <Productos>
         <Producto>
            <Nombre>Matrix 7.0</Nombre>
            <FechaAlta>1307542390468</FechaAlta>
         </Producto>
         <Producto>
            <Nombre>Gravedad 2.0</Nombre>
            <FechaAlta>1307542390468</FechaAlta>
         </Producto>
      </Productos>
   </Pedido>
   <Pedido numeroPedido="2">
      <Cliente>
         <Nombre>Mr.</Nombre>
         <Apellido>Regan</Apellido>
         <User>Cifra</User>
      </Cliente>
      <Productos>
         <Producto>
            <Nombre>Reinsercion 1.0</Nombre>
            <FechaAlta>1307542390468</FechaAlta>
         </Producto>
      </Productos>
   </Pedido>
 </Pedidos>
```

EJEMPLO PRÁCTICO

Visualiza un ejemplo de Listado de la clase Java

```
import java.io.IOException;
import java.util.ArrayList;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;

public class PedidosXML_SAX extends DefaultHandler {

	private XMLReader reader;

	public PedidosXML_SAX() {
		try {
			reader = XMLReaderFactory.createXMLReader();
			reader.setContentHandler(this);
			reader.setErrorHandler(this);
		} catch (SAXException e) {
			e.printStackTrace();
		}
	}

	public ArrayList<Pedido> XMLtoPedidos(String ruta) {
		ArrayList<Pedido> pedidos = new ArrayList<Pedido>();

		try {
			reader.parse(ruta);
		} catch (IOException e) {
			e.printStackTrace();
		} catch (SAXException e) {
			e.printStackTrace();
		}

		return pedidos;
	}

	@Override
	public void startElement(String uri, String localName, String name,
                                                        Attributes
atts) {
		System.out.println("<" + localName + ">");
	}
```

```
        @Override
        public void characters(char[] cadena, int inicio, int length) {
                if(String.valueOf(cadena, inicio, length).trim().length()
!= 0)
                        System.out.println(String.valueOf(cadena, inicio,
                             length));
        }

        @Override
        public void endElement(String uri, String name, String qName) {
                System.out.println("</" + name +">");
        }
}
```

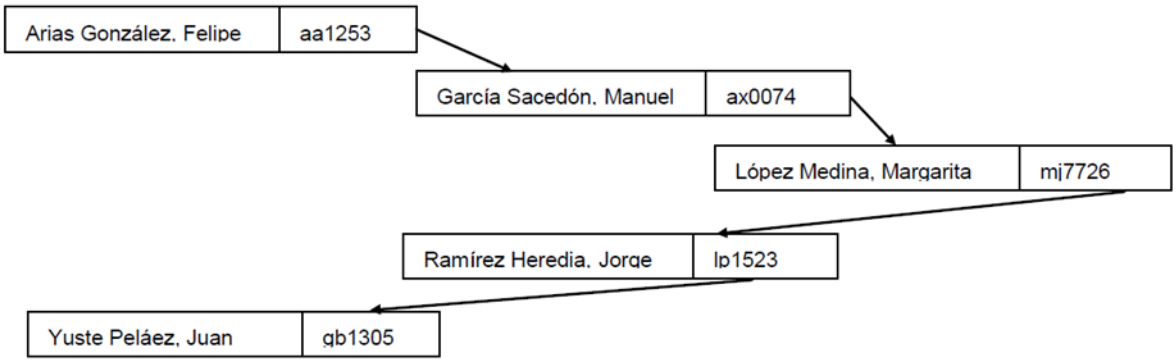
4. CONCEPTO DE LISTA

En tu equipo de trabajo han entrado un grupo de desarrolladores en prácticas. No han trabajado mucho con estructuras dinámicas de datos, hasta ahora, solo habían trabajado con arrays. Por tanto, crees necesario impartir una clase magistral sobre el concepto de lista y como implementarla desde cero. Aunque ya existen listas implementadas en la API de Java, es importante que todo desarrollador entienda, por lo menos, su funcionamiento interno antes de utilizar las que ya están predefinidas.

Una lista es una estructura de datos lineal que se puede representar simbólicamente como un conjunto de nodos enlazados entre sí.

Las listas permiten modelar diversas entidades del mundo real como, por ejemplo, los datos de los alumnos de un grupo académico, los datos del personal de una empresa, los programas informáticos almacenados en un disco magnético, etc.

La figura muestra un ejemplo de lista correspondiente a los nombres y apellidos de un conjunto de alumnos con su código de matrícula.



Tal vez resulte conveniente identificar a los diferentes elementos de la lista (que normalmente estarán configurados como una estructura de registro) mediante uno de sus campos (clave) y en su caso, se almacenará la lista respetando un criterio de ordenación (ascendente o descendente) respecto al campo clave.

Una **definición formal** de lista es la siguiente:

"Una lista es una secuencia de elementos del mismo tipo, de cada uno de los cuales se puede decir cuál es su siguiente (en caso de existir)."

Existen dos criterios generales de calificación de listas:

- **Por la forma de acceder** a sus elementos.

    - **Listas densas**. Cuando la estructura que contiene la lista es la que determina la posición del siguiente elemento. La localización de un elemento de la lista es la siguiente:

        - Está en la posición 1 si no existe elemento anterior.
        - Está en la posición N si la localización del elemento anterior es la posición (N-1).

    - **Listas enlazadas**: La localización de un elemento es:

        - Estará en la dirección k, si es el primer elemento, siendo k conocido.
        - Si no es el primer elemento de la lista, estará en una dirección, j, que está contenida en el elemento anterior.

- **Por la información** utilizada para acceder a sus elementos:

    - **Listas ordinales**. La posición de los elementos en la estructura la determina su orden de llegada.
    - **Listas calificadas**. Se accede a un elemento por un valor que coincide con el de un determinado campo, conocido como clave. Este tipo de listas se pueden clasificar a su vez en ordenadas o no ordenadas por el campo clave.

ENLACE DE INTERÉS

Visita este enlace donde obtendrás más información sobre las estructuras de datos:

4.1 Implementación de listas

El concepto de lista puede implementarse en soportes informáticos de diferentes maneras.

Mediante estructuras estáticas.

Con toda seguridad resulta el mecanismo más intuitivo. Una simple *matriz* resuelve la idea

0	Arias González, Felipe	aa1253
1	García Sacedón, Manuel	ax0074
2	López Medina, Margarita	mj7726
3	Ramírez Heredia, Jorge	lp1523
4	Yuste Peláez, Juan	gb1305

Implementación de una lista densa mediante una estructura estática (matriz).

El **problema** de esta alternativa es el derivado de las operaciones de **inserción** y **modificación**.

En efecto, la declaración de una lista mediante una matriz implica **conocer de antemano** el número (o al menos el orden de magnitud) de elementos que va a almacenar, pudiendo darse las circunstancias de que si se declara con un número pequeño podría desbordarse su capacidad o, en caso contrario, declararlo desproporcionadamente con un número elevado provocaría un decremento de eficiencia.

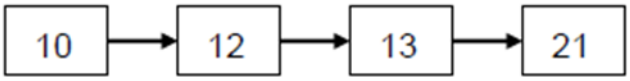
Otro problema asociado es el tratamiento de los **elementos eliminados**. Dado que, en el caso de no informar, de alguna manera, de la inexistencia de dicho elemento el nodo previamente ocupado (y ahora no válido) quedaría como no disponible.

Adicionalmente, si se desea trabajar con listas ordenadas el algoritmo de inserción debería alojar a los nuevos elementos en la posición o con la referencia adecuada.

Algunas soluciones, más o menos ingeniosas, permiten tratar estas circunstancias. La figura siguiente muestra un ejemplo basado en una matriz de registros.

0	1	2	3	4	5	6	7	8
1	10	77	12	26	21	11	13	18
2	3	4	7	6	0	8	5	0

Otra posible representación de esta lista sería la siguiente:



VÍDEO DE INTERÉS

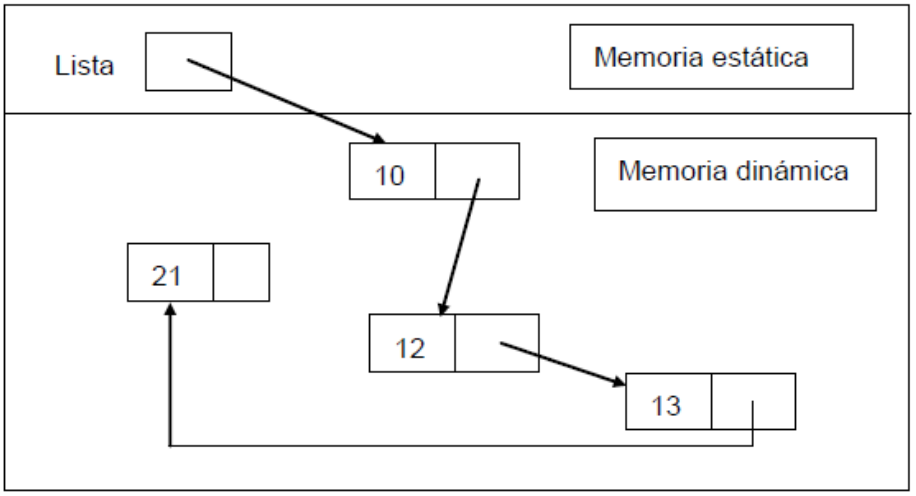
Conoce el funcionamiento de las listas enlazadas en Java en este vídeo :

Mediante estructuras dinámicas.

Sin duda se trata de **la mejor alternativa** para la construcción de listas. Se trata de hacer uso de la correspondiente tecnología que implica el uso de referencias.

La idea consiste en declarar una **referencia a un nodo** que será el primero de la lista. Cada nodo de la lista (en la memoria dinámica) contendrá tanto la propia información **del mismo** como una referencia al **nodo siguiente**. Se deberá establecer un convenio para identificar el **final de la lista**.

La figura muestra un ejemplo de implementación dinámica de la lista de la figura 3.



Lo explicado hasta el momento consiste en la solución más sencilla: cada nodo de la lista "apunta" al siguiente, con las excepciones del **último elemento** de la lista (su "apuntador", o "**referencia**" es un valor especial, por ejemplo, *null*, en Java) y del **primero**, que es "apuntado" por la referencia declarada en la memoria estática. Esta tecnología se identifica como "Listas enlazadas o unidireccionales". Existen otras posibilidades entre las que cabe mencionar: listas bidireccionales o doblemente enlazadas, listas circulares, listas con cabecera, listas con centinela o cualquier combinación de ellas.

4.2 Tratamiento de listas en Java

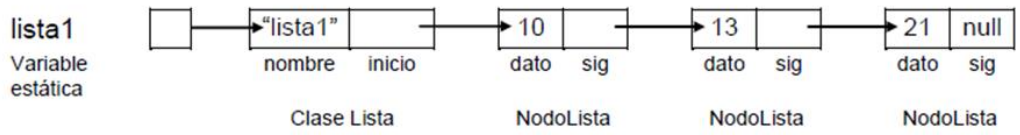
Para la utilización de listas es necesario definir la clase *NodoLista* utilizando la siguiente sintaxis:

```
public class NodoLista {
	public int dato;
	public NodoLista sig;

	public NodoLista(int x, NodoLista n) {
```

Así como la clase Lista:

```
public class Lista {
	public Lista(String nombreLista) {
		inicio = null;
		nombre = nombreLista;
	}

	public NodoLista inicio;
	public String nombre;
}
```

La representación gráfica de una variable de la clase `Lista` llamada *lista1* que contenga los elementos 10, 13 y 21 sería:



4.3 Algoritmos básicos con listas

Los algoritmos que implican el recorrido, parcial o total, de la lista pueden implementarse tanto de forma recursiva como iterativa.

Recorrido completo.

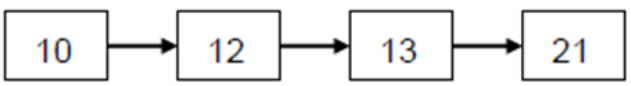
El recorrido de una lista de manera **`recursiva`** se puede realizar por medio de un método ***`static`*** invocado desde un programa que utilice la clase `Lista`. Esto implica utilizar el tipo `NodoLista` para avanzar por la lista y ver el contenido del campo clave. Merecen una consideración especial:

- La llamada inicial donde *lista* es el valor de la variable estática.
- El final del recorrido, que se alcanza cuando la lista está vacía.

En el siguiente método estático (`escribirLista`), se recorre una lista (`nodoLista`) mostrando en la pantalla el contenido de sus campos clave. Se utiliza un método de llamada (`escribirListaCompleta`), que recibe como argumento un objeto de la clase *Lista* (*lista*):

```
static void escribirLista(NodoLista nodoLista) {
	if (nodoLista != null) {
		System.out.print(nodoLista.clave + " ");
		escribirLista(nodoLista.sig);
	} else
		System.out.println(" FIN");
}

static void escribirListaCompleta(Lista lista) {
	if (lista != null) {
		System.out.print(lista.nombre + ": ");
		escribirLista(lista.inicio);
	} else
		System.out.println("Lista vacía");
}
```

Si se aplica el algoritmo anterior a la lista de la figura



el resultado sería la secuencia:

lista1: 10 13 21 FIN.

La ejecución de *escribirListaCompleta* implicaría una llamada inicial al método *escribirLista,* pasando como argumento *lista.inicio* (la referencia al nodo de clave 10) y 3 llamadas recursivas al método *escribirLista*:

- En la primera llamada recursiva, *nodoLista* es el contenido del campo *sig* del nodo de clave 10. Es decir, una referencia al nodo de clave 13.

- En la segunda, *nodoLista* es el contenido del campo *sig* del nodo de clave 13. Es decir, una referencia al nodo de clave 21.

- En la tercera, *nodoLista* es el contenido del campo *sig* del nodo de clave 21, es decir, *null*. Cuando se ejecuta esta tercera llamada se cumple la condición de finalización y, en consecuencia, se inicia el proceso de "vuelta". Ahora *nodoLista* toma sucesivamente los valores:

  - Referencia al nodo de clave 21 (campo *sig* del nodo de clave 13).
  - Referencia al nodo de clave 13 (campo *sig* del nodo de clave 10).
  - Referencia al nodo de clave 10 (el primer elemento de la lista).

El **recorrido de una lista** es una operación necesaria en los algoritmos de **eliminación** y, en muchos casos, también en los de **inserción**. La condición de finalización "**pesimista**" consiste en alcanzar el final de la lista (*nodoLista == null*). No obstante, normalmente se produce una **terminación anticipada** que se implementa *no realizando nuevas llamadas recursivas*.

En los siguientes apartados se van a presentar los diferentes tipos de listas, sobre las cuales se explican algunos algoritmos básicos: *inicialización, búsqueda, inserción y eliminación.*

4.4 Listas ordinales

En las listas ordinales el **orden** dentro de la estructura lo establece la **llegada** a la misma. A diferencia de las listas calificadas, en este tipo de listas no existe ningún elemento que identifique el nodo, y, por lo tanto, los valores **se pueden repetir**. El criterio de inserción resulta específico en cada caso (se podría insertar por el principio, o bien por el final).

ENLACE DE INTERÉS

Conoce dos ejemplos de listas ordinales, las pilas y las colas, para entender mejor de en qué consisten:

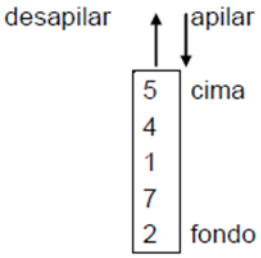
Veremos a continuación dos ejemplos de listas ordinales que ya hemos tratado como TADs (*Tipos Abstractos de Datos*): las pilas y las colas.

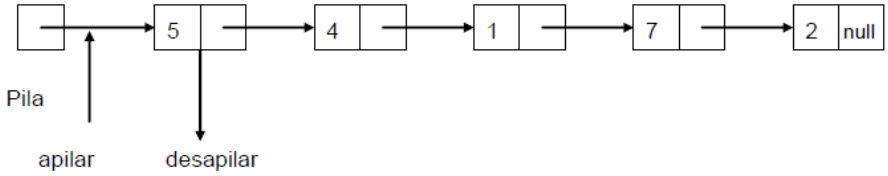
Pilas.

Una pila es una agrupación de elementos de determinada naturaleza o tipo (datos de personas, números, procesos informáticos, automóviles, etc.) entre los que existe definida una relación de orden (**estructura de datos**). En función del tiempo, algunos elementos de dicha naturaleza pueden llegar a la pila o salir de ella (**operaciones / acciones**). En consecuencia, el estado de la pila varía.

En una pila (comportamiento **LIFO** -*Last In First Out*-) se establece el criterio de ordenación en **sentido inverso al orden de llegada**. Así pues, el último elemento que llegó al conjunto será el primero en salir del mismo, y así sucesivamente.

Las figuras siguientes ilustran respectivamente el concepto de pila de números enteros y su implementación mediante una lista dinámica.





ENLACE DE INTERÉS

Una de las clases más importantes en Java para la implementación de Pilas es la clase Stack. Amplíe información visitando este enlace:

La estructura de datos de la pila y el constructor utilizado sería:

```
package tadPila;

//En esta clase se define el nodo:
class NodoPila {
	// Atributos accesibles desde otras rutinas del paquete
		int dato;
		NodoPila siguiente;

		// Constructor
	NodoPila(int elemento, NodoPila n) {
		dato = elemento;
		siguiente = n;
	}

}
```

Y la interfaz utilizada sería la siguiente:

```
package tadPila;

import java.io.*;

public interface Pila {
	void inicializarPila();
	boolean pilaVacia();
	void eliminarPila();
	int cima() throws PilaVacia;
	void apilar(int x);
	int desapilar() throws PilaVacia;
	void decapitar() throws PilaVacia;
	void imprimirPila();
	void leerPila() throws NumberFormatException, IOException;
	int numElemPila();
}
```

A continuación, se muestra la clase TadPila, con el constructor y los algoritmos correspondientes a las operaciones *pilaVacia*, *inicializarPila*, así como *apilar* y *desapilar* que operan al principio de la lista por razones de eficiencia.

```
package tadPila;

public class TadPila implements Pila {
	protected NodoPila pila;

	public TadPila() {
		pila = null;
	}

	public void inicializarPila() {
		pila = null;
	}

	public boolean pilaVacia() {
		return pila == null;
	}

	public void apilar(int dato) {
		NodoPila aux = new NodoPila(dato, pila);
		pila = aux;
	}

	public int desapilar() throws PilaVacia {
		int resultado;
		if (pilaVacia())
			throw new PilaVacia("Desapilar: La pila está
```

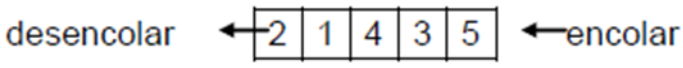
```
                                                    vacía");
                resultado = pila.dato;
                pila = pila.siguiente;
                return resultado;
            }
        }
```

Colas.

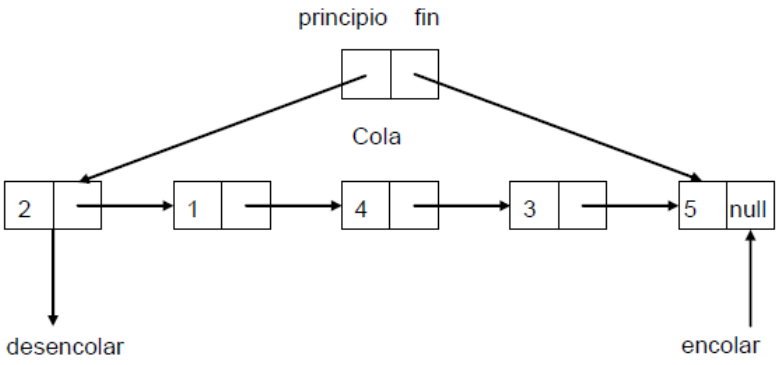
Una cola es una agrupación de elementos de determinada naturaleza o tipo (datos de personas, números, procesos informáticos, automóviles, etc.) entre los que existe definida una **relación de orden**. En función del tiempo, pueden llegar a la cola o salir de ella algunos elementos de dicha naturaleza (**operaciones/acciones**). En consecuencia, el estado de la cola varía.

En una cola (comportamiento **FIFO** -*First In First Out*-) se respeta como criterio de ordenación el momento de la llegada: el primero de la cola será el que primero llegó a ella y, en consecuencia, el primero que saldrá, y así sucesivamente.

Las figuras siguientes ilustran respectivamente el concepto de cola de números enteros y su implementación mediante una lista dinámica.



Modelo gráfico de Cola.



ENLACE DE INTERÉS

Amplía información sobre colas en Java, visitando:

La estructura de datos de la cola y el constructor sería:

```
package tadCola;

//En esta clase se define el nodo:

class NodoCola {
	//Atributos accesibles desde otras rutinas del paquete
		int dato;
		NodoCola siguiente;
	//Constructor
	NodoCola(int elemento, NodoCola n) {
		dato = elemento;
		siguiente = n;
	}
}
```

Y la interfaz utilizada sería:

```
package tadCola;

import java.io.IOException;

public interface Cola {
	void inicializarCola();
	boolean colaVacia();
	void eliminarCola();
	int primero() throws ColaVacia;
	void encolar(int x);
	int desencolar () throws ColaVacia;
	void quitarPrimero() throws ColaVacia;
	void mostrarEstadoCola();
	void imprimirCola();
```

```
        void leerCola() throws NumberFormatException, IOException;
        int numElemCola();
        void invertirCola() throws ColaVacia;
}
```

5. COLECCIONES

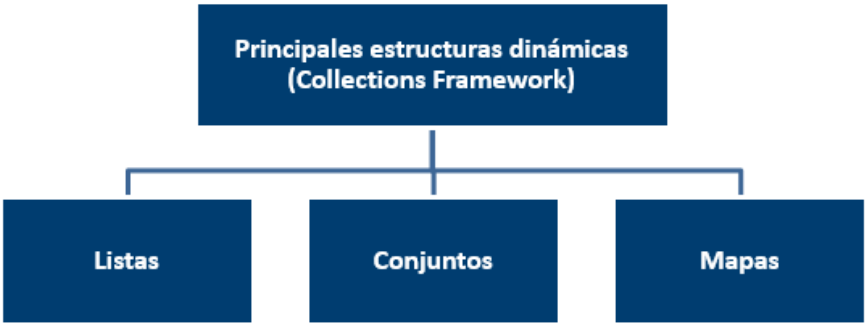
Parece que el concepto de lista ha quedado claro a todo el equipo junior y de prácticas, así como los tipos de listas que existen. Es el momento de empezar a utilizar las colecciones que proporciona Java y sus métodos. El uso de colecciones en la aplicación que estás desarrollando tendrá numerosas ventajas, entre las que cabe destacar, el dinamismo, ya que estas pueden crecer o reducirse en tamaño según sea necesario, la reutilización de código y la mejora del rendimiento que proporcionan los algoritmos de búsqueda y manipulación optimizados que ya están probados, como es el caso de conjuntos y mapas.

Hasta este punto, hemos visto arrays de una dimensión y de dos dimensiones que utilizamos como almacenes donde guardamos un conjunto de datos que están relacionados. El término colecciones en Java hace referencia a un conjunto de clases que han sido construidas con el fin de almacenar muchos objetos estruturados de algún modo. Estos arrays que hemos visto hasta ahora también son colecciones. Por ejemplo, un String[] es una colección de cadenas, en la que cada cadena se almacena en una posición específica del array, del mismo modo un Integer[] es una colección de objetos pero de tipo Integer y un Object[] es una colección de objetos de cualquier clase.

Pero los arrays convencionales (los estudiados hasta ahora) se declaran con un tamaño específico y no se puede cambiar ese tamaño durante la ejecución del programa, es decir, si declaramos un array de 10 elementos no se puede aumentar o disminuir esta cantidad en dicho array durante la ejecución de la aplicación.

Para superar esta limitación, Java proporciona otros tipos de estructuras dinámicas que se incluyen en lo que se denomina "`Java Collections Framework`", que son útiles cuando no sabemos de antemano el espacio que se va a ocupar en memoria y se pueden aumentar y disminuir en tiempo de ejecución.

Principalmente nos encontraremos con tres tipos fundamentales de estructuras dentro del `Framework Collections` que son; las listas, los conjuntos y los mapas o diccionarios.



Principales estructuras dinámicas del Framework Collections.

Fuente: Elaboración propia.

VÍDEO DE INTERÉS

Visualiza este resumen sobre colecciones en Java.

Listas.

En apartados anteriores se ha profundizado en el concepto de lista y como es su estructura interna. Las listas que podemos usar del framework Collection siguen la misma filosofía añadiendo ciertas características o funcionalidades que las diferencian a unas de otras. Son colecciones ordenadas de elementos que pueden contener duplicados. Se utilizan cuando el lugar que ocupan los elementos es importante y cuando se necesitan operaciones de acceso por índice. (nos recuerdan a los arrays, pero los arrays son estáticos y las listas son dinámicas).

Algunas de las implementaciones más comunes de la interfaz List son ArrayList y LinkedList.

Conjuntos.

Los conjuntos son colecciones que no permiten elementos duplicados. Se utilizan cuando se necesita almacenar una colección única de elementos y poder verificar la

existencia o no de elementos en el conjunto. Algunas implementaciones de la interfaz Set incluyen HashSet, TreeSet y LinkedHashSet.

Mapas.

Sirven para guardar datos identificados por claves que no se repiten (deben ser únicas). Se almacenan pares clave-valor y permiten buscar rápidamente valores por su clave. Algunas implementaciones de la interfaz Map son HashMap, TreeMap y LinkedHashMap.

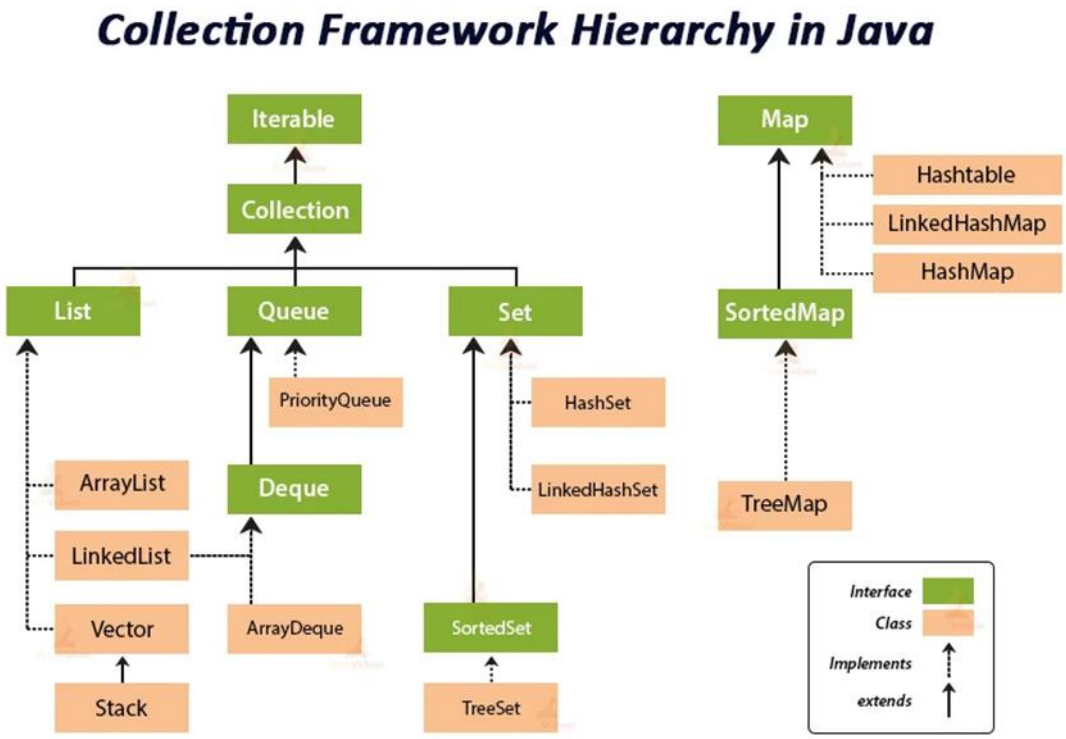
El uso de un tipo de colección u otra lo establecerán los requisitos de la aplicación que se esté desarrollando y de la eficiencia esperada.

El conjunto de interfaces y clases del marco de trabajo Collection se encuentran en el paquete `java.util`. Aquí podemos encontrar dos grupos, las que derivan de la interfaz `Collection` y las que derivan de la interfaz `Map`.

La interface **`Collection`** declara una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección, etc. que definen operaciones básicas que deben incluir las colecciones que la implementan. Partiendo de la interfaz genérica `Collection` extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.

La interface **`Map`** se utiliza para representar colecciones de pares clave-valor. Los métodos definidos en la interfaz `Map` se centran en operaciones relacionadas con la gestión y búsqueda de valores a través de sus claves. Por este motivo no tiene relación con la interface `Collection` pero se suelen incluir en el grupo de las colecciones.

En resumen, las interfaces `Collection` y `Map` representan dos tipos diferentes de estructuras de datos dinámicas, cada una con sus propias características y métodos específicos. La falta de herencia entre `Collection` y `Map` refleja esta distinción en sus propósitos y funcionalidades.



Resumen de Interface Collection, Map y sus derivaciones.
Recuperado de https://techvidvan.com/tutorials/java-collection-framework

ENLACE DE INTERÉS

En este enlace puedes repasar los tipos de interfaces y clases que forman parte del marco de trabajo Collection(Collections Framework)

5.1 Listas

Las listas son una de las estructuras de datos más comunes y versátiles en la programación. En Java, las listas dinámicas son implementadas por la interfaz `List` y tienen varias implementaciones concretas, siendo las más comunes `ArrayList` y `LinkedList`(lista enlazada, cada nodo tiene la dirección del elemento siguiente y el anterior).

Características de las listas:

- Permiten almacenar una colección de elementos en un orden específico, se respeta el orden de inserción.
- Ofrecen una variedad de operaciones para agregar, eliminar, modificar y acceder a elementos.
- Permiten la duplicación de elementos, es decir, podemos tener elementos idénticos en posiciones distintas.
- Tienen un tamaño dinámico, lo que significa que pueden crecer o reducirse según la necesidad.
- Los elementos de una lista se pueden ordenar fácilmente utilizando el método sort().
- Podemos usar genéricos para especificar el tipo de elementos que contendrá la lista, lo que proporciona seguridad en tiempo de compilación.

`ArrayList` y `LinkedList` proporcionan los mismos métodos y funcionalidades porque ambas implementan la interfaz `List`. La diferencia entre usar una u otra afecta levemente al rendimiento, de modo que podríamos utilizar tanto una como otra. `LinkedList`, además, implementa métodos de la interfaz Deque, esto significa que podemos utilizar una LinkedList para trabajar con colas.

`ArrayList` es más rápida en operaciones que impliquen recorrer la lista para lectura o modificación de elementos, sin embargo, `LinkedList` tiene mejor rendimiento en las operaciones de inserción y eliminación de nodos, pero no tanta leyendo elementos.

Como se ha comentado anteriormente, dependiendo del uso que se vaya a hacer de la colección se usará una u otra, por ejemplo, si se van a realizar muchos accesos y pocas eliminaciones o inserciones sería mejor el uso de un `ArrayList`, en caso contrario, si va a haber más inserciones o eliminaciones que búsquedas, entonces la opción más apropiada sería un LinkedList.

Sintaxis general para construir un ArrayList:

Para construir cualquier tipo de colección dinámica, normalmente, solo hay que cambiar las interfaces y clases implicadas, a continuación, puedes ver el formato para construir un ArrayList.

```
ArrayList<tipo_de_dato> nombreDeLaLista = new ArrayList<>();
```

Aquí estás declarando directamente el tipo concreto **`ArrayList`** como el tipo de la variable **`nombreDeLaLista`**. Esto significa que estás atando específicamente tu código a la implementación `ArrayList`.

o también

//se recomienda usar esta

```
List<tipo_de_dato> nombreDeLaLista = new ArrayList<>();
```

En esta declaración, estás utilizando la **interfaz `List`** como el tipo de la variable *nombreDeLaLista*. Esto te permite cambiar fácilmente la implementación subyacente a cualquier clase que implemente la interfaz List, como por ejemplo…`ArrayList`, `LinkedList`, etc.

Si **no se incluye** el *tipo_de_dato,* la lista podrá contener o almacenar objetos de cualquier tipo (los tipos de datos básicos los convertirá de forma automática a su clase envolvente o wrapper para tratarlos como objetos antes de añadirlos al ArrayList) pero **si lo incluimos** (el tipo_de_dato), la lista solo contendrá datos del *tipo_de_dato* indicado.

Más adelante trataremos más a fondo el uso de la **genericidad** en Java, que se refiere a la capacidad de crear clases, interfaces y métodos que puedan trabajar con tipos de datos genéricos.

Si se incluye *el <tipo_de_dato>*, este debe ser una clase, es decir, no puede ser un tipo primitivo, por ejemplo, si queremos que la lista almacene solo enteros (int), entonces utilizaremos **`Integer`** como tipo_de_dato, que es la clase envolvente de **`int`**.

ENLACE DE INTERÉS

Para ampliar información sobre las listas y su tratamiento en Java visite este enlace:

EJEMPLO PRÁCTICO

- Crea un **ArrayList** que almacene números enteros.
- Agrega nuevos elementos usando el método **add(),** accede a ellos con el método **get()** y para eliminarlos el método **remove().**
- Recorre el **ArrayList** y muestra los elementos antes y después de realizar inserciones y eliminaciones.

```
import java.util.ArrayList;
```

- Crea un **ArrayList** que almacene números enteros.
- Agrega nuevos elementos usando el método **add(),** accede a ellos con el método **get()** y para eliminarlos el método **remove().**
- Recorre el **ArrayList** y muestra los elementos antes y después de realizar inserciones y eliminaciones.

```
import java.util.ArrayList;

public class EjemploArrayList {
	public static void main(String[] args) {

		// Crear un ArrayList de enteros
		ArrayList<Integer> numeros = new ArrayList<>();

		// Agregar elementos al ArrayList
		numeros.add(5);
		numeros.add(10);
		numeros.add(15);

		// Acceder y mostrar elementos del ArrayList
		System.out.println("Elementos del ArrayList:");
```

```
            for (int i = 0; i < numeros.size(); i++) {
                  System.out.println(numeros.get(i));
            }

            // Eliminar un elemento
            numeros.remove(1);

            // Mostrar elementos actualizados
            System.out.println("Elementos después de eliminar:");
            for (Integer num : numeros) {
                  System.out.println(num);
            }
      }
}
```

La salida en pantalla será:

```
Elementos del ArrayList:
5
10
15
Elementos después de eliminar:
5
15
```

La clase Math en Java es una clase que proporciona métodos estáticos para realizar operaciones matemáticas. Es parte del paquete java.lang. Entre las operaciones que incluye está la generación de números al azar con el método random(). El rango o margen con el que trabaja el método random() oscila entre 0.0 y 1.0 (Este último no incluido).

EJEMPLO PRÁCTICO

Genera un número entero aleatorio entre 0 y 9 utilizando la clase **Math** y su método **random().**

```
public class EjemploGeneracionNumeroAleatorio {

	public static void main(String[] args) {
		//rango de valores  0.0 <= numero < 10.0
		int numero1 = (int) (Math.random () * 10);
		System.out.println(numero1);

		//rango de valores  0.0 <= numero <= 10.0
		int numero2 = (int)(Math. random()*10+1);
		System.out.println(numero2);
	}
}
```

Al multiplicar el número aleatorio obtenido por 10, el rango de valores en el que se generará un número aleatorio se convierte en: 0.0 <= numero < 10.0 (no incluye el número 10), al multiplicarlo por 10+1, el rango de valores será 0.0 <= numero <= 10.0(se incluye también el 10), por tanto, el número obtenido aleatoriamente será un número comprendido entre 0 y 10(incluido).

EJEMPLO PRÁCTICO

Te proponemos un sencillo juego, en el que se elimine a una persona de un grupo al azar. Imagina una audición de actores o cantantes en la que se debe informar de que alguien no ha sido seleccionado para continuar.

Para ello se pide crear un ArrayList que almacene los nombres de las personas y utilizando la clase `Math` y el método `random()` escojas un nombre aleatorio de la lista, lo elimines, informes por pantalla de los sucedidos y a continuación muestres todos los nombres de las personas que quedan en el grupo.

```
import java.util.ArrayList;
public class EjemploArrayList_Math
{
	public static void main(String[] args)
	{
		int indiceDelEliminado;
		String eliminado;
		ArrayList<String> grupo = new ArrayList<String>();

		grupo.add("Marta");
		grupo.add("Pablo");
		grupo.add("Sergio");
		grupo.add("Paloma");
		grupo.add("Laura");

		indiceDelEliminado = (int) (Math.random() *
                                                grupo.size());
		eliminado = grupo.remove(indiceDelEliminado);
		System.out.println("Lo siento, " + eliminado +
                    ". No has sid@ seleccionad@.\n");
		System.out.println("Personas que permanecen en el
                                               grupo:\n " + grupo);
	}
}
```

Ten en cuenta que como está planteado el ejemplo, en cada ejecución solo se eliminará una persona de la lista. Para que en la misma ejecución se eliminen mas de una, entonces sería útil la introducción de una estructura de repetición como por ejemplo un while o un do..while, que repita la eliminación en función de que se cumpla una condición, que puede ser, la confirmación de seguir o no seguir por parte del usuario o que la lista ya no tenga elementos para eliminar.

La salida por pantalla en una ejecución determinada podría ser la siguiente:

```
Lo siento, Paloma. No has sid@ seleccionad@.

Personas que permanecen en el grupo:
```

```
[Marta, Pablo, Sergio, Laura]
```

¿SABÍAS QUE...?

El método `random()` de la clase `Math` inicialmente nos devuelve un número aleatorio entre 0.0 y 0.1(esto lo podemos cambiar, como has visto), pero si queremos almacenar este número aleatorio obtenido en una variable de tipo entero debemos realizar un casting, que consistirá en escribir `(int)` delante de la llamada al método.

Ej:

```
int indiceDelEliminado
indiceDelEliminado = (int) (Math.random() * grupo.size());
eliminado = grupo.remove(indiceDelEliminado);
```

PARA SABER MÁS

Añade al ejercicio práctico el código necesario para que se pregunte al usuario si quiere seguir eliminando personas del grupo. En caso de que la respuesta sea afirmativa se elimine a otra persona, siempre al azar ,muestre el resultado y las personas que siguen en el grupo.
La ejecución finalizará bien por orden del usuario o porque ya no haya más elementos en la lista, en este caso también se debe indicar por pantalla esta situación.

EJEMPLO PRÁCTICO

- Crea una **LinkedList** que almacene cadenas.
- Añade elementos al inicio y al final utilizando los métodos **addFirst()** y **addLast().**
- Recorre la lista y muestra los elementos antes y después de las inserciones y eliminaciones.

```
import java.util.LinkedList;

public class EjemploLinkedList {
    public static void main(String[] args) {
        // Crear una LinkedList de cadenas
        LinkedList<String> nombres = new LinkedList<>();

        // Agregar elementos al inicio de la LinkedList
        nombres.addFirst("Rocío");
        nombres.addFirst("Juan");
        nombres.addFirst("Daniel");

        // Agregar elementos al final de la LinkedList
        nombres.addLast("Nuria");
        nombres.addLast("MariCarmen");

        // Acceder y mostrar elementos de la LinkedList
        System.out.println("Elementos de la LinkedList:\n");
        for (String nombre : nombres) {
            System.out.println(nombre);
        }

        // Eliminar el primer y último elemento
        nombres.removeFirst();
        nombres.removeLast();

        // Mostrar elementos actualizados
        System.out.println("\nElementos después de eliminar:\n");
        for (String nombre : nombres) {
            System.out.println(nombre);
        }
    }
}
```

La salida por pantalla será:

```
Elementos de la LinkedList:

Daniel
Juan
Rocío
```

```
Nuria
MariCarmen

Elementos después de eliminar:

Juan
Rocío
Nuria
```

La clase **Vector** en Java es una de las estructuras de datos proporcionadas por la interface **List**. Anteriormente, era una de las opciones principales para manejar arrays dinámicos antes de la introducción de otras colecciones más modernas y eficientes. Es similar a **ArrayList**,

¿SABÍAS QUE...?

Las colas son estructuras de datos que se utilizan para almacenar y administrar elementos en un orden específico.

Dos términos comunes asociados con colas son FIFO y LIFO. FIFO, significa "Primero en entrar, primero en salir" y LIFO, "Último en entrar, primero en salir".

Ejemplos:
Cola FIFO: Cola de espera en un supermercado, si llegas primero te atienden antes.
Cola LIFO: Cuando apilamos platos unos encima de otros, el último será el primero que cogeremos.

ENLACE DE INTERÉS

En este enlace puedes revisar los métodos que proporciona la interfaz Collection y List para trabajar con listas:

VÍDEO DE INTERÉS

Conoce las diferencias entre ArrayList y LinkedList en este vídeo:

5.1.1 Iteradores

Para recorrer una lista nodo a nodo podemos hacerlo con un bucle for o por medio de iteradores. Estos son objetos que apuntan sucesivamente a los nodos de la lista, comenzando por el primero. Cuando queremos utilizar un iterador con una lista, primero hay que crear el iterador. A continuación, se invoca el método iterador() desde la lista y nos devolverá un iterador asociado a ella.

Dispone de tres métodos principales, que son **`hasNext(), next() y remove().`**
Método hasNext: es un método de la interfaz Iterator que verifica si hay más elementos para recorrer en la colección.

- Devuelve **true** si hay más elementos disponibles para ser accedidos por el iterador, y **false** si no hay más elementos.
- Útil para controlar el flujo de ejecución al recorrer una colección, evitando que intentes acceder a elementos que no existen.

Método next(): Es otro método de la interfaz `Iterator` que devuelve el siguiente elemento en la colección y avanza el iterador al siguiente elemento.

- Si `hasNext()` devolvió true, `next()` devuelve el siguiente elemento en la secuencia.
- Después de llamar a `next()`, el iterador avanza a la siguiente posición en la colección.
- Si no hay más elementos para recorrer (es decir, `hasNext()` es false), llamar a `next()` puede lanzar una excepción NoSuchElementException.

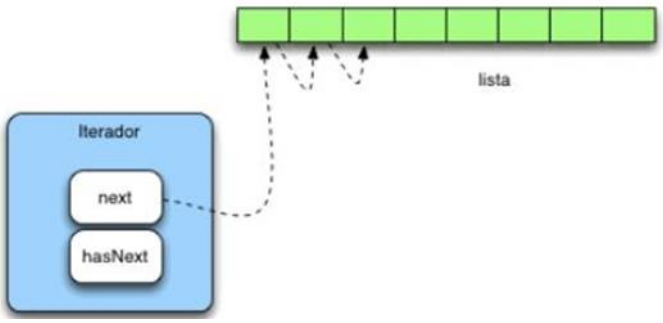
Método remove(): Es otro método importante proporcionado por la interfaz Iterator en Java

- Es un método de la interfaz Iterator que se utiliza para eliminar el elemento actual de la colección subyacente durante el proceso de iteración.

- Este método permite eliminar el elemento al que apunta el iterador en ese momento específico, pero no se puede usar antes de llamar a next().

- Generalmente se utiliza para evitar problemas al eliminar elementos mientras se recorre una colección, especialmente cuando se desea eliminar elementos de manera segura sin afectar la estructura de la colección.

- No todos los iteradores admiten la operación remove(). Por ejemplo, los iteradores generados por colecciones inmutables (no modificables) lanzarán una excepción si se intenta llamar a remove()



Iterador de una lista.
Fuente: https://www.arquitecturajava.com/java-iterator-vs-foreach/

Para entender el uso de iteradores, aquí tienes dos ejemplos que muestran el recorrido de una lista de varias formas, una sin iterador y otra con iterador, además puedes ver cómo usar un bucle `for-each` que es similar al bucle `for` pero con una estructura más simplificada.

EJEMPLO PRÁCTICO

Se crea una nueva lista llamada 'lista' que almacena elementos de tipo String mediante la implementación de un ArrayList.

Se agregan 3 elementos a la lista.

Se recorre la lista usando un sencillo bucle for.

```
import java.util.ArrayList;
import java.util.List;

	public class EjemploSinIterador {
	    public static void main(String[] args) {
	        List<String> lista = new ArrayList<>();
	        lista.add("Elemento 1");
	        lista.add("Elemento 2");
	        lista.add("Elemento 3");

	        for (int i = 0; i < lista.size(); i++) {
	            System.out.println(lista.get(i));
	        }
	    }
	}
```

En el siguiente ejemplo en vez de recorrer la lista con una estructura `for` como de costumbre, se hace uso de un `for-each` y de un `iterador`, después de añadir los elementos a la lista.

La lectura de este `for-each` sería la siguiente:

<<*Para cada elemento de tipo* `String` *que pertenece a lista mostrar elemento*, entonces la variable *elemento* irá tomando cada vez los valores de la lista '*lista*'>>

Aquí, se aplica este bucle a una lista, pero se puede usar con cualquier colección, sin embargo, hay operaciones para las que no vale, como por ejemplo eliminar nodos ya que '`elemento`' es siempre la referencia a una copia de un elemento de la lista, en este caso tendremos que usar el iterador.

A continuación, se define un iterador llamado '`it`' con el tipo String que empieza apuntando al principio de la lista 'lista'.

Justo después se vuelve a recorrer la lista pero usando el iterador que acabamos de crear que se ejecuta mientras el iterador tenga más elementos para recorrer, es decir, mientras `it.hasNext()` devuelva true. En el momento que se encuentre un elemento igual a "Elemento 2" se efectuará el borrado de dicho elemento.

```
import java.util.ArrayList;
import java.util.Iterator;
```

```
import java.util.List;

public class EjemploRemoveIterator {
	public static void main(String[] args) {
		List<String> lista = new ArrayList<>();
		lista.add("Elemento 1");
		lista.add("Elemento 2");
		lista.add("Elemento 3");
		 lista.add("Elemento 2");

		// Mostrar la lista antes de la eliminación
		System.out.println("Lista antes de la eliminación:");
		for (String elemento : lista) {
			System.out.println(elemento);
		}

		//creación del iterador
		Iterator<String> it = lista.iterator();

		//recorrer la lista
		while (it.hasNext()) {
			String elemento = it.next();
			if (elemento.equals("Elemento 2")) {
				// Elimina el elemento "Elemento 2"
				it.remove();
			}
		}

		// Mostrar la lista después de la eliminación
		System.out.println("\nLista después de la
                                        eliminación:\n" + lista);
	}
}
```

El resultado por pantalla será:

```
Lista antes de la eliminación:
Elemento 1
Elemento 2
Elemento 3
Elemento 2


Lista después de la eliminación:
[Elemento 1, Elemento 3]
```

5.2 Conjuntos

La interfaz Set en Java es parte del framework Collections.

Características de los conjuntos.

- Se utilizan para representar colecciones que no permiten elementos duplicados y en la cual el orden no es importante (no se respeta el orden de inserción). (La estructura TreeSet si es ordenada, cada vez que insertamos si mantiene el orden).

- Cada elemento en el conjunto debe ser único.

- Todos sus métodos los hereda de la interfaz Collection lo único que añade es la restricción de no permitir duplicados, por tanto, si se intenta añadir un nodo ya existente no se realizará dicha inserción (aunque no se producirá ningún aviso de error ni excepción) y el método add () devolverá false.

Java proporciona varias clases que implementan la interfaz `Set`, como `HashSet`, `TreeSet` y `LinkedHashSet`. Cada implementación tiene sus propias características y comportamientos.

- **HashSet**: Implementación que utiliza una tabla hash para almacenar elementos. Ofrece un acceso rápido y no garantiza un orden específico. Tiene un buen rendimiento.

EJEMPLO PRÁCTICO

Crea un HashSet que almacene cadenas de texto.

Agrega varios elementos e intenta añadir uno que ya hayas agregado. Recorre el HashSet y muestra el contenido. Verifica si existen algunos nombres dentro del conjunto, a continuación, elimina alguno de los existentes y vuelve a mostrar los elementos actualizados.

```
import java.util.HashSet;

public class EjemploHashSet {
	public static void main(String[] args) {
	// Crear un HashSet de cadenas
	HashSet<String> nombres = new HashSet<>();

	// Agregar elementos al HashSet
	nombres.add("Nuria");
	nombres.add("Daniel");
	nombres.add("Rocío");

	// Intento de agregar un duplicado
	nombres.add("Nuria");

	// Mostrar elementos del HashSet
	System.out.println("Elementos del HashSet:\n");
	for (String nombre : nombres) {
		System.out.println(nombre);
	}

	// Verificar si un elemento existe en el HashSet
	boolean existeAlice = nombres.contains("Nuria");
	boolean existeDavid = nombres.contains("David");

	System.out.println("\n¿Existe Nuria? " + existeAlice);
	System.out.println("\n¿Existe David? " + existeDavid);

	// Eliminar un elemento del HashSet
	nombres.remove("Daniel");

	// Mostrar elementos actualizados
	System.out.println("\nElementos después de eliminar:\n");
	for (String nombre : nombres) {
		System.out.println(nombre);
	}
  }
}
```

La salida en pantalla será la siguiente:

```
Elementos del HashSet:

Nuria
Daniel
Rocío

¿Existe Nuria? true

¿Existe David? false

Elementos después de eliminar:

Nuria
Rocío
```

- **TreeSet**: Implementación que almacena los elementos en un árbol binario de búsqueda balanceado, ordenando automáticamente los elementos. Tiene peor rendimiento para buscar o modificar que un HashSet pero garantiza el orden según el valor de los elementos insertados.

VÍDEO DE INTERÉS

En este enlace puedes visualizar un video que explica con detalle y con ejemplos la estructura TreeSet en Java.

VÍDEO DE INTERÉS

Y, para un nivel más avanzado, con TreeSet, accede a este vídeo:

- **LinkedHashSet**: Implementación que mantiene el orden de inserción de los elementos utilizando una estructura auxiliar de lista enlazada unidad a la tabla hash, además de no permitir duplicados.

VÍDEO DE INTERÉS

Visualiza este resumen sobre Set en Java.

VÍDEO DE INTERÉS

En este vídeo podrás ver un resumen sobre HashSet en Java.

5.3 Mapas o diccionarios

Los mapas realmente no son colecciones, aunque están relacionadas con ellas y están dentro del mismo entorno de trabajo, pero en la unidad los veremos dentro del concepto de colecciones.

Funcionan de forma parecida a como lo hace un diccionario.

Características de los mapas:

- Los Map no heredan de la interface Collection, por tanto, tendrán métodos diferentes a las otras colecciones vistas, pero con funcionalidades parecidas.
- No permiten elementos duplicados.
- Están compuestos por una clave y un valor.
- A la hora de crearlo debemos indicarle cual es la clave y cual el valor.
- La clave debe ser única, aunque el valor puede estar repetido.

La sintaxis general para crear un mapa es la siguiente:

```
Map<keyType,ValueType> nombreMapa = new claseImplementaMap<>();
```

Donde *KeyType* representa la clave y *ValueType* el valor asociado a la clave. *ClaseImplementaMap* será el nombre de la clase que implementa la interfaz Map, como por ejemplo HasMap, TreeMap o LinkedHashMap.

- **HashMap**

Es una implementación de Map con una tabla hash. Es una colección con tiempos relativamente cortos de búsqueda e inserción.

EJEMPLO PRÁCTICO

Dentro de una misma clase, crea una clase Empleado y una clase Main. La clase Empleado solo tendrá un atributo que será el nombre del empleado, un constructor que reciba un parámetro y los getter y setter correspondientes.

Dentro de la clase Main(en el ejemplo se llama EjemploHashMap), en el método main() debes crear un mapa y añadir al menos los datos de dos empleados(usando el constructor de la clase Empleado) con sus correspondientes claves que serán el dni de cada uno.

Elimina uno de ellos por su clave e imprime también el valor asociado a una clave específica, finalmente, recorre todas las entradas del mapa y muestra tanto la clave como el valor de cada entrada utilizando un bucle 'for-each'

```
import java.util.HashMap;
import java.util.Map;

public class EjemploHashMap {

public static void main(String [] args){
	Map<String,Empleado> plantilla = new
					HashMap<String,Empleado>();

	plantilla.put("34806509V", new Empleado("Jesús Gómez"));
	plantilla.put("34111261A", new Empleado("Vanesa Muñoz"));
	plantilla.remove("34806408V");
	System.out.println("El DNI 34806509V corresponde a "
					+ plantilla.get("34806509V"));
	//recorrer todas las entradas del mapa
	System.out.println("\nListado de empleados:");
	System.out.println("--------------------");
	for (Map.Entry<String,Empleado> entrada :
				plantilla.entrySet())
	{
		String clave = entrada.getKey();
		Empleado e = entrada.getValue();
		System.out.println("\nclave = " + clave
				+ ", valor = "
			+ e);
}
}

static class Empleado{
	private String nombre;

	public Empleado(String nombre) {
		this.nombre = nombre;
```

```
	}

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	@Override
	public String toString() {
		return	'\'' + nombre + '\'';
	}

}

}
```

La salida en pantalla será:

```
El DNI 34806509V corresponde a 'Jesús Gómez'

Listado de empleados:
--------------------

clave = 34806509V, valor = 'Jesús Gómez'

clave = 34111261A, valor = 'Vanesa Muñoz'
```

PARA SABER MÁS

La declaración
`Map<KeyType, ValueType> nombreMapa = new HashMap<KeyType, ValueType>(),`
se adhiere a buenas prácticas de programación al enfocarse en la interfaz Map en lugar de la implementación concreta HashMap, lo que brinda flexibilidad y abstracción en el código.

Es decir, se puede declarar un HashMap de la siguiente forma también,
`HashMap<KeyType, ValueType> nombreMapa = new HashMap<KeyType, ValueType>()`
Como se ha visto en los ejemplos, pero se recomienda la primera.
Esta puntualización es aplicable a todas las colecciones (ArrayList, HasSet, etc.)

VÍDEO DE INTERÉS

Con este video podrás obtener más información y ejemplos sobre HashMap en Java.

- **TreeMap**

Los elementos tendrán un orden. Por ejemplo, si la clave es un valor entero, se ordenará de menor a mayor y si son variables de texto, el orden será alfabético.

- **LinkedHashMap**

Los elementos se ordenan según su orden de inserción. Esta clase tiene peor rendimiento que las otras cuando se realizan búsquedas.

ENLACE DE INTERÉS

En este enlace encontrarás información sobre Map en Java.

VÍDEO DE INTERÉS

Si quieres conocer más sobre la interfaz Map, accede a este video.

6. GENERICIDAD

Una de las premisas más importantes en la aplicación que estás construyendo es reducir la posibilidad de errores en tiempo de ejecución, por ello, indicas a tu equipo la necesidad de utilizar genéricos para evitar el tener que realizar castings repetitivos. El uso de clases y métodos genéricos proporcionarán verificaciones de tipos en tiempo de compilación lo cual ayudará a detectar y prevenir errores de tipos antes de que se ejecute la aplicación.

La genericidad en Java se refiere al uso de tipos genéricos o parámetros de tipo en la definición y uso de clases, interfaces y métodos. La genericidad permite crear componentes reutilizables que pueden trabajar con diferentes tipos de datos sin necesidad de escribir código específico para cada tipo. La programación genérica surgió a partir de la versión 5.0 de Java.

VÍDEO DE INTERÉS

Accede a este vídeo donde podrás visualizar un video sobre la genericidad en Java.

Ventajas:

- Independencia entre el código y el tipo de datos utilizado lo cual reduce el impacto del cambio y el coste en el mantenimiento.
- Mejora la seguridad y la legibilidad del código.
- Reutilización y reducción de código duplicado.
- Permite definir clases y métodos que funcionan con cualquier tipo de datos sin sacrificar la seguridad de tipo.
- Comprobación del uso correcto de tipos de datos en tiempo de compilación.

La genericidad se introduce mediante el uso de paréntesis angulares ("< >") o también llamados operadores ‘diamante’ y el nombre de tipo genérico dentro de los corchetes.

En ejemplos anteriores hemos visto cómo utilizar esta característica con colecciones, pero se puede utilizar con clases, métodos y atributos.

Ejemplo en una declaración de clase:

```
public class DeclaracionClaseGenerica<T> {
 //Código y cuerpo de la clase aquí
}
```

Donde '*DeclaracionClaseGenerica*' será el nombre de la clase, <T> especifica el parámetro de tipo genérico (puedes usar cualquier identificador para representar el tipo genérico), el cuerpo de la clase incluirá los atributos, métodos y constructores de la clase. También se pueden agregar atributos y métodos que utilicen el tipo genérico 'T' dentro del cuerpo de la clase para crear componentes reutilizables que trabajen con diferentes tipos de datos.

EJEMPLO PRÁCTICO

Para ilustrar la diferencia entre usar o no usar tipos de datos genéricos vamos a crear una lista de tipo ArrayList, sin indicar el tipo de dato que va a contener.

```
package tiposGenericos;

import java.util.ArrayList;
import java.util.List;

public class ArrayListDeObjetos {

	public static void main(String[] args) {
	//Definición ArrayList
	List miLista = new ArrayList<>();
	miLista.add("hola");
	miLista.add(25);

	String primerDato = (String)miLista.get(0);
	System.out.println(primerDato);
	//Integer segundoDato = miLista.get(1);
	Integer segundoDato = (Integer)miLista.get(1);
	System.out.println(segundoDato);

	}
}
```

Como no se ha indicado el tipo, el ArrayList estará formado por cualquier tipo de objeto. En el caso del ejemplo, queremos almacenar el primer dato en una variable de tipo String y el segundo dato de la lista en una variable de tipo numérico, pero como la lista está formada por objetos, devolverá objetos y será necesario realizar un cast o casting para realizar una conversión de tipo al tipo de la variable primerDato(String) y segundoDato(Integer).

El problema surge porque tenemos que saber de qué tipo es cada uno de los elementos de la lista para realizar el casting apropiado, por tanto, habría que valorar si nos interesa establecer esta estructura según las necesidades de la aplicación

RECUERDA

Los genéricos permiten al autor de una clase o método introducir parámetros de tipo que son símbolos que pueden ser sustituidos por un tipo concreto.

EJEMPLO PRÁCTICO

Para mostrar cómo funciona realmente una clase genérica en Java, vamos a construir una clase llamada Librería que nos permitirá almacenar objetos de tipo Libro y Bolígrafo.

El principio en que se basa toda clase genérica es que puede almacenar un solo tipo de objeto, pero no diferentes a la vez, es decir, se va a adaptar al tipo de dato que sea recibido como argumento y así evitamos tener una clase por cada tipo de dato. Vamos a ver qué quiere decir esto.

```
package tiposGenericos.EjemploLibreria;

import java.util.ArrayList;

public class Libreria < T > {
	private ArrayList < T > lista = new ArrayList < T >();

	public void add(T objeto){
		lista.add(objeto);
	}
	public ArrayList<T> getProducts(){
		return lista;
	}
}
```

El parámetro de tipo T es el que maneja la clase Librería. Cuando queramos utilizar esta clase, debemos instanciarla y en ese momento se le pasará el tipo de dato que queremos que se almacene en nuestro ArrayList llamado 'lista' que tenemos declarado en la clase librería.

Como otra clase independiente tenemos la clase Libros.

```
package tiposGenericos.EjemploLibreria;

public class Libro {
	private String titulo;
	private String autor;

	public Libro(String titulo, String autor) {
		this.titulo = titulo;
		this.autor = autor;
	}
	@Override
	public String toString() {
		return "- Título: " + titulo + " - Autor: " +  autor ;
	}
}
```

Y la clase Boligrafo

```
package tiposGenericos.EjemploLibreria;

class Boligrafo {
	private String nombre;
	private String tipo;

	public Boligrafo(String nombre, String tipo) {
		this.nombre = nombre;
		this.tipo = tipo;
	}
	 @Override
	public String toString() {
	    return "- Nombre: " + nombre + " - Tipo: " + tipo;
	}
}
```

En otra clase que contenga un método main(), se instanciará la clase Librería, pasándole como tipo de dato la clase Libro y por otro lado, pasándole como tipo de dato la clase Bolígrafo, de esta forma tendremos dos listas, una de libros y otra de bolígrafos. Como son listas de tipo ArrayList(recuerda que se declaró un atributo llamado 'lista' de tipo ArrayList en la clase Librería) usaremos los métodos que provee el ArrayList, para añadir elementos a cada una de las listas.

```
package tiposGenericos.EjemploLibreria;

public class ArrayListConGenericos {

	public static void main(String[] args) {
		Libreria<Libro> listaLibros = new Libreria<Libro>();
		Libreria<Boligrafo> listaBoligrafos =
                                        new Libreria<Boligrafo>();

		listaLibros.add(new Libro("Cien años de soledad",
                                    "Gabriel García Márquez"));
		listaLibros.add(new Libro("Don Quijote de la Mancha",
                                "Miguel de Cervantes Saavedra"));

		listaBoligrafos.add(new Boligrafo("Gel", "Suave"));
		listaBoligrafos.add(new Boligrafo("Bola", "Estándar"));

		System.out.println("\nArtículos en la librería:");
		System.out.println("------------------------");

		// Recorrer y mostrar libros
		System.out.println("\nLIBROS:\n");
		for (Libro libro : listaLibros.getProducts()) {
			System.out.println(libro);
		}

		// Recorrer y mostrar chocolates
		System.out.println("\nBOLÍGRAFOS:\n");
```

```
            for (Boligrafo boli : listaBoligrafos.getProducts()) {
                  System.out.println(boli);
            }
      }
}
```

Cuando se ejecute la clase main, la salida por consola será la siguiente:

Artículos en la librería:

LIBROS:

- Título: Cien años de soledad - Autor: Gabriel García Márquez
- Título: Don Quijote de la Mancha - Autor: Miguel de Cervantes Saavedra

BOLÍGRAFOS:

- Nombre: Gel - Tipo: Suave
- Nombre: Bola - Tipo: Estándar

PARA SABER MÁS

Un cast o casting en Java nos permite realizar una conversión de un tipo. Se indica entre paréntesis con el tipo de dato al que queremos pasar.

Por ejemplo:
int a;
double b=2.5;
a=(int)b;

Si en un momento dato nos interesa almacenar en la variable 'a' el valor que está almacenado en la variable 'b' no será posible porque una variable de tipo int no puede almacenar datos de tipo double. Pero si hacemos un cast, en este caso sería posible, de forma que la variable 'a' despúes de este casting almacenará un 2 (se ha convertido 2.5 que era un double a el número 2 que es un entero)

ENLACE DE INTERÉS

En este enlace a una cuenta en GitHub puedes encontrar una gran cantidad de ejercicios sobre estructuras dinámicas y sobre otros conceptos que se verán en el módulo de programación, que podrás poner en práctica.

VÍDEO DE INTERÉS

Con este video aumentarás tus conocimientos sobre ArrayList y cómo utilizarlos con genéricos:

7. EXPRESIONES REGULARES

Estas muy contento ya que la aplicación está practicamente terminada, se ha realizado un gran trabajo en equipo, solo queda la parte de validaciones de entradas del usuario para que se cumplan formatos y restricciones impuestas por la aplicación. En común se ha decidido, usar expresiones regulares que son extremadamente potentes y flexibles para validar direcciones de correo electrónico, números de teléfono, códigos postales y contraseñas, entre otros.

Una expresión regular (regex o regexp) en Java es una secuencia de caracteres que define un patrón de búsqueda. Se utiliza para buscar y manipular texto en cadenas de caracteres, ya sea para encontrar coincidencias, reemplazar partes de una cadena, validar formatos o realizar otras operaciones relacionadas con el procesamiento de texto.

En Java, las expresiones regulares se implementan a través de la clase java.util.regex.Pattern y la clase java.util.regex.Matcher.

Las expresiones regulares tienen una amplia gama de aplicaciones en el procesamiento y manipulación de texto.

Aquí tienes algunos ejemplos comunes en los que puedes usar expresiones regulares:

Validación de formato de:

- Direcciones de correo electrónico.
- Números de teléfono en diferentes formatos.
- Códigos postales, números de identificación.
- Contraseñas, etc.

Extracción de Información de:

- Enlaces de una página web.
- Fechas, horas o elementos específicos de un texto.
- Menciones de usuarios o hashtags en redes sociales, etc.

Reemplazo y formato de:

- Caracteres o palabras concretas en un texto o cadena.
- Números, fechas en un formato específico.

Búsqueda y filtrado:

- Buscar palabras determinadas en documentos y textos largos.
- Filtrar líneas que cumplan criterios en un archivo.

Tokenización y separación:

- Dividir una cadena en palabras o frases.
- Separar elementos a partir de un carácter u otro delimintador.

ARTÍCULO DE INTERÉS

Debido a que muchos lenguajes y plataformas de programación admiten expresiones regulares, hoy día existen muchas herramientas online que permiten probar expresiones regulares antes de empezar a codificarlas en un lenguaje determinado. En este enlace tienes algunas de ellas.

Estos son solo algunos ejemplos de cómo las expresiones regulares pueden ser utilizadas en el desarrollo de software. La versatilidad de las expresiones regulares les permite ser una herramienta poderosa para tratar con una amplia variedad de tareas relacionadas con el procesamiento de texto.

La expresión regular define un patrón para buscar una cadena de caracteres. El patrón se buscará en la cadena o texto de izquierda a derecha comprobando que se cumple el patrón especificado. El patrón estará compuesto por una serie de caracteres y símbolos que indican cuáles serán las coincidencias válidas o no válidas. A continuación, tienes una tabla con algunos de estos símbolos más comunes, pero hay más.

Símbolos comunes en expresiones regulares:

Patrón	Descripción
.	Indica cualquier carácter.
*	0 ó más.
+	1 ó más.
?	0 ó 1.
{n}	Donde N será un número. Se indica que sea exactamente ese número. Ej: {3} exactamente tres.
{n,}	Indica que puede ser n ó más. Ej: {3,}, puede ser 3 ó más.
{n,m}	Indica que puede ser un valor entre n y m. Ej:{3,5},puede ser 3,4 o 5.
^expresión	El símbolo ^ indica el principio de la cadena. Con esto indicamos que la expresión debe ir al principio de la cadena.
expresión$	El símbolo $ indica el final de la cadena. En este caso el String debe contener la expresión al final.
[abc]	Los corchetes representan a una clase de caracteres. En este ejemplo la cadena debe contener las letras a ó b ó c (alguna de las tres).
[abc][56]	La cadena debe contener las letras a ó b ó c seguidas de 5 ó 6
[^abc]	El símbolo ^ dentro de los corchetes indica negación. En este caso la cadena debe contener cualquier carácter excepto a ó b ó c.
[a-z1-9]	Especifica un rango, que sería de la letra minúscula 'a' hasta la letra minúscula 'z', y los dígitos desde el 1 hasta el 9 (los extremos se incluyen en ambos casos).
A|B	El carácter | es un OR. A ó B.
AB	Concatenación. A seguida de B.
(ar)+c	Los paréntesis definen un grupo de captura. Referencia a cualquier cadena de texto que contenga uno o más ocurrencias de la secuencia "ar" seguida de la letra c.

VÍDEO DE INTERÉS

En este vídeo tienes explicaciones sobre expresiones regulares y ejercicios

Pasos para trabajar con expresiones regulares en Java:

1. Importar la clase **`java.util.regex.Pattern`** y **`java.util.regex.Matcher`**: Para trabajar con expresiones regulares, necesitas importar estas dos clases. Puedes hacerlo al principio de tu archivo Java.

2. **Definir la expresión regular:** Decide qué patrón deseas buscar o manipular en el texto. Define la expresión regular que describa ese patrón. Por ejemplo, si quieres buscar la palabra "casa", la expresión regular podría ser "casa".

3. **Compilar la expresión regular:** Utiliza el método estático Pattern.compile() para compilar la expresión regular en un objeto Pattern.

4. **Crea un objeto 'Matcher':** Utiliza un objeto 'Pattern' para crear un objeto 'Matcher' que buscará el patrón en el texto.

5. **Usar el objeto 'Matcher' para buscar coincidencias:** Utiliza los métodos como find() ó matches() en el objeto Matcher para buscar coincidencias en el texto.

PARA SABER MÁS

Aprende a utilizar la página de regexr.com que es una de las más utilizadas para practicar con expresiones regulares independientemente del lenguaje en el que vayas a programar.

Los objetos **`Pattern`** y **`Matcher`** son clases fundamentales en Java para trabajar con expresiones regulares. Juntos, permiten compilar y aplicar patrones de búsqueda en cadenas de texto. Aquí hay una descripción más detallada de cada uno:

Objeto Pattern:

La clase `Pattern` representa un patrón de expresión regular compilado. Puedes pensar en ella como una plantilla que describe un patrón que deseas buscar o manipular en una cadena de texto. Esta clase proporciona varios métodos útiles para compilar y manipular expresiones regulares. Los métodos importantes de esta clase son:

- **compile**(String regex): Compila una expresión regular en un objeto Pattern.
- **matcher**(CharSequence input): Crea un objeto Matcher para buscar el patrón en la cadena especificada.

Objeto Matcher:

La clase `Matcher` se utiliza para aplicar un patrón de expresión regular a una cadena de texto y buscar coincidencias en esa cadena. Puedes pensar en ella como una "máquina" que busca y evalúa la cadena de texto utilizando el patrón compilado. También proporciona una serie de métodos, los más destacados son:

- **matches():** Verifica si toda la cadena coincide con el patrón.
- **find():** Busca la próxima coincidencia en la cadena.

RECUERDA

El método matches() verifica si toda la cadena completa coincide con el patrón proporcionado mientras que el método find() se usa para buscar coincidencias parciales en el texto.

EJEMPLO PRÁCTICO

Realiza un pequeño programa en Java, que utilice una expresión regular para encontrar una coincidencia de una palabra específica en un texto.

```
package expresionesRegulares;

import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class MiPrimeraExpresionRegular {

	public static void main(String[] args) {
		//Definir expresión regular
		String patron = "casa";

		//compilar la expresión regular en objeto Pattern
		Pattern objetoPattern = Pattern.compile(patron);

		//crear objeto Matcher
		String texto = "La casa de la pradera.";
		Matcher objetomatcher = objetoPattern.matcher(texto);

		//Buscar coincidencias con el objeto Matcher
		if (objetomatcher.find()) {
			System.out.println("Se encontró una
					coincidencia.");
		} else {
			System.out.println("No se encontraron
					 coincidencias.");
		}
	}
}
```

La salida por pantalla será:

Se encontró una coincidencia.

EJEMPLO PRÁCTICO

Cambia el patrón para que se encuentre una cadena concreta al principio de un texto.

```
package expresionesRegulares;
```

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class CoincidenciasAlPrincipio {

	public static void main(String[] args) {
	    Pattern pat = Pattern.compile("^ABC.*");
	    String texto = "ABC es una marca de periódico.";
	     Matcher mat = pat.matcher(texto);

	     if (mat.matches()) {
	         System.out.println("Válido");
	     } else {
	         System.out.println("No Válido");
	     }
	}
}
```

La salida por pantalla será:

Válido

ENLACE DE INTERÉS

En este link tienes un resumen y ejemplos de expresiones regulares en Java:

ENLACE DE INTERÉS

Si quieres más información y ejemplos para probar en la misma página y de forma online, accede a este enlace:

RESUMEN FINAL

Cualquier programa que se desarrolle en **Java** y que tenga la necesidad de manejar una cantidad importante de datos los organizará en arrays colecciones o en archivos XML que se parsearán utilizando DOM o SAX. EL uso de la clase String en Java es importante y no trivial: hay que tener en cuenta el funcionamiento de esta clase para trabajar de forma adecuada.

En esta unidad se ha comenzado analizando el concepto de array unidimensional y de más de una dimensión como estructura de datos estática, debido a que su tamaño, una vez definido durante la creación, no puede cambiar durante la ejecución del programa. Posteriormente se ha visto el funcionamiento de la clase String, para pasar a ver el funcionamiento de la interpretación de archivos XML utilizando SAX y DOM.

Para finalizar esta unidad nos hemos adentrado en el funcionamiento de estructuras de datos dinámicas como son las listas, los conjuntos y los mapas.

Hemos aprendido que hay diferentes estructuras llamadas Collections que forman parte del API de JAVA y que hay ciertas ventajas en utilizar unas u otras, todo dependerá del tipo de funcionalidad que deseemos para que su utilización sea más o menos eficiente.

Por otro lado, se ha puesto de manifiesto que la genericidad en la programación y especialmente en Java, ofrece varias ventajas que pueden mejorar la claridad, seguridad y eficiencia de una aplicación software. Entre ellas cabe destacar la reutilización de código, la seguridad en el tipo de dato, utilización excesiva de casting y reducción de errores en tiempo de ejecución.

Se ha realizado una introducción a las expresiones regulares, que nos permiten describir patrones de texto muy complejos y específicos para reemplazar, modificar, analizar o formatear grandes cantidades de datos en poco tiempo, así como una valiosa forma de validar si cadenas de texto cumplen con ciertos formatos o patrones como direcciones de correo electrónico, números de teléfono o fechas.