MEDAC
Instituto Oficial de Formación Profesional

PROGRAMACIÓN

Introducción a las estructuras de almacenamiento

4

ÍNDICE

OBJETIVOS

Conocer el concepto de array.

Comprender el concepto de array multidimensional.

Realizar operaciones con arrays.

Aplicar expresiones regulares.

Operar con cadenas de caracteres.

Diferenciar entre array y cadenas de caracteres.

/ 1. Introducción y contextualización práctica

En este tema, hablaremos sobre el concepto de array, cuál es su sintaxis, y cómo podemos operar con ellos.

Veremos las operaciones más habituales que se pueden hacer con arrays, declararlos, acceder a sus elementos, modificarlos, ordenarlos y realizar búsquedas en ellos.

También hablaremos sobre los arrays multidimensionales, más conocidos como matrices, y veremos cuáles son las operaciones básicas que se pueden realizar con ellos.

Por último, aprenderemos cómo trabajar con cadenas de caracteres.

Todos estos conceptos los iremos reforzando en todos los temas restantes, así que no te preocupes, que tenemos mucho tiempo por delante.

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

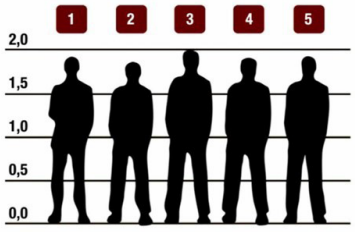


Fig. 1. En un array se almacenan valores del mismo tipo.

Audio Intro. "¿Cuántos objetos necesito?"
https://bit.ly/2X1WFEZ

/ 2. Concepto de array

Estamos acostumbrados, de unidades anteriores, a realizar programas donde siempre vamos a saber la cantidad de variables u objetos que vamos a necesitar, pero ¿qué ocurre cuando necesitamos crear un programa que pueda manipular muchas variables, pero no sepamos cuántas? Este problema suele entrañar verdaderas dificultades si no se conocen los arrays. El concepto de array en programación se puede definir como **un área de memoria conjunta formada por muchas variables del mismo tipo[1].**

Los arrays nos permitirán almacenar una cantidad limitada de elementos del mismo tipo y poder trabajar con ellos.

Las operaciones más usuales que vamos a realizar con arrays son las siguientes:

- Creación e inicialización.
- Agregar elementos.
- Eliminar elementos.
- Realizar búsquedas de elementos.
- Ordenar el array.

En los *arrays*, podremos tener datos variables de tipo primitivo, como objetos.

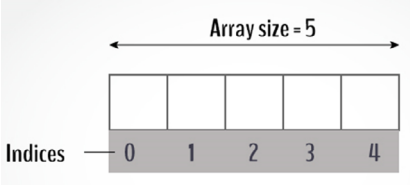


Fig. 2. Los arrays tienen un tamaño fijo y a cada elemento se accede por su posición (índice).

/ 3. Composición y representación en memoria

Los *arrays* se **componen** de:

- Un tipo, del que serán todos los elementos que almacena el *array*.
- Una serie de valores, uno por cada elemento.
- Un tamaño, que será el que nos indique el máximo de elementos que podremos almacenar en el *array*.
- Una dirección de memoria, donde estará almacenado el *array* en la memoria RAM.

Los elementos que componen el *array* estarán **almacenados en memoria de forma contigua**, es decir, uno detrás del otro. Esto implica que cada elemento tendrá una posición dentro del *array*, lo que se conoce como **índice.** Es importantísimo recordar que los índices de los elementos en los *arrays* empiezan desde el 0, esto quiere decir que, si tenemos un *array* de tamaño 5 los índices, o posiciones dentro del *array*, de los elementos serán 0, 1, 2, 3 y 4.

Un array es como una estantería: aunque esté vacío, el espacio estará reservado en memoria. Cuando queramos almacenar un elemento tendremos que indicar el índice (el estante) que le corresponde.

Que el array tenga un tamaño determinado no implica que se utilicen todos los huecos, si no que ese será el máximo de elementos que podremos almacenar. Sin embargo, la memoria se estará ocupando incluso en las posiciones vacías.

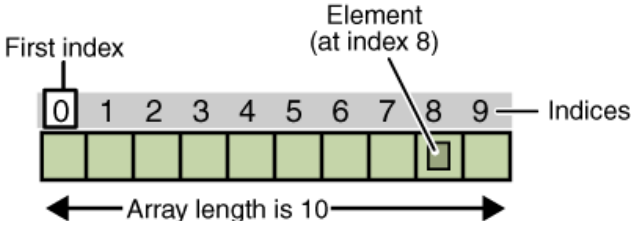


Fig. 3. Representación en memoria de un array. Fuente: http://www.codexion.com/tutorialesjava/java/nutsandbolts/arrays.html

1. Es muy importante tener claro que no vamos a poder tener arrays con elementos de diferente tipo.

Esto quiere decir que tendremos una cantidad de elementos útiles (huecos utilizados) y un máximo de elementos (capacidad del array). Estos elementos útiles no podrán ser mayores que el máximo. Cuando los elementos útiles y el máximo sean iguales tendremos el array lleno.

/ 4. Caso práctico 1: "¿Cómo represento una cantidad desconocida de variables?"

Planteamiento: Pilar y José están realizando un programa en el que tienen que calcular la nota media de todas las notas de las asignaturas de un alumno.

Todo iría bien de no ser porque tienen que poder calcular esa media de una cantidad de exámenes que no conocen, es decir, primero deberán saber cuántos exámenes hay y luego calcular su nota media.

Nudo: ¿Cómo crees que pueden solucionarlo? ¿Podrían utilizar *arrays* para ello o declarando variables sería suficiente?

Desenlace: Este ejercicio es el típico caso de uso de *arrays*, ya que deberemos operar con muchos valores para realizar un cálculo sobre ellos, y lo que es más importante, no sabemos cuántos de esos datos vamos a necesitar leer. Siempre que se cumplan esta serie de condiciones vamos a poder aplicar *arrays* para resolverlo. Al tener que averiguar la cantidad de elementos con los que vamos a operar en primer lugar (en este caso sería la cantidad de exámenes), podemos montar una estructura que nos permita ir 'pidiendo' datos, hasta que se nos suministren todos, y con ello, vamos a poder definir un *array* de ese tamaño en cuestión.

De esta forma, siempre vamos a tener la cantidad adecuada de datos que necesitamos. No nos debe importar no conocer de primeras este dato, ni hay que tenerlo en cuenta de cara a la declaración de variables, ya que, con el *array*, podremos leerlos todos, operar con ellos, y obtener un resultado de su manipulación.

En la figura a continuación se muestra el código Java para calcular la media de los elementos de un array:

```
double[] array = {19, 12.89, 16.5, 200, 13.7};
double total = 0;

for (int i = 0; i < array.length; i++) {
    total = total + array[i];
}

double average = total / array.length;
```

Código 1. Cálculo de la media de los de un array.

/ 5. Arrays unidimensionales

Dentro de los *arrays*, podemos encontrar los denominados *arrays* unidimensionales, que a priori son los que menos dificultades entrañan de entendimiento.

Podemos imaginarlos como una serie de valores, del mismo tipo, en fila unos detrás de otros, es decir: **Los *arrays* unidimensionales tendrán una única dimensión, de ahí que se puedan imaginar como una fila india.**

Para poder crear un array deberemos indicar su tipo, su nombre y su tamaño, de la siguiente forma:

```
tipo[] nombrevariable = new tipo[ tamaño ];
```

El tamaño de los *arrays* será siempre un número entero positivo mayor que cero.

Para acceder a un elemento del *array* basta con indicar su índice, recordando siempre que empezamos en 0. Con el acceso a los elementos del *array* podremos realizar las siguientes operaciones:

- Asignación de un valor.
- Modificación de un valor.
- Lectura por teclado de un valor.
- Mostrar por pantalla el contenido de una posición.

Con los elementos del *array*, podremos realizar todas las operaciones que nos permita su tipo. Como es lógico, al acceder a un elemento del *array*, el índice será un entero positivo mayor o igual a 0. Si agregamos más elementos a un *array* de los que tiene definido como máximo recibiremos un error.

5.1. Métodos y arrays

Los *arrays* al fin y al cabo son variables que definimos, y como variables que son, podremos utilizarlos en los métodos[2].

Los *arrays* los podremos utilizar de la siguiente forma:

```java
public static void main(String[] args) {
    Scanner teclado_int = new Scanner(System.in);
    int[] array = new int[10]; // Declaración de un array

    array[0] = -9; // Pongo el valor -9 en la primera posición
    array[1] = teclado_int.nextInt(); // Leo por teclado un valor
    array[2] = array[0] + array[1]; // Suma de elementos del array

    System.out.println(array[1]); // Muestro por pantalla un valor
}
```

Código 2. Ejemplo de Array

- Los métodos podrán **devolver** un *array*.
- Los métodos podrán **tener como parámetro** un *array*.
- Los métodos podrán declarar *arrays* **dentro de su bloque de código**, siendo éstos locales al método.

Como vemos, las funciones pueden devolver un *array*, con lo cual, la restricción de que una función solo podía devolver un valor la podemos tergiversar un poco, si bien solo podemos devolver un único valor, si devolvemos un *array* podremos devolver muchos valores en él, por lo que podremos "saltarnos" esa restricción.

Para obtener la cantidad de elementos que tiene un *array* podemos usar la función *length*. Los *arrays* en los parámetros de los métodos se pasan por referencia, así que si los modificamos dentro de un método esa modificación será permanente.

```java
/**
 * Esta función rellena un array
 * @param array Array a rellenar
 */
public static void rellenarArray(int[] array)
{
    // Recorro el array, yendo desde 0 hasta su tamaño
    for (int i = 0; i < array.length; i++) {
        array[i] = 7;
    }
}

public static void main(String[] args) {
    int[] array = new int[10]; // Declaración de un array

    rellenarArray(array);

    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i]);
    }
}
```

Código 3. Arrays con métodos.

2. Recordemos que cuando nos referimos a métodos estamos incluyen tanto funciones como procedimientos.

5.2. Ordenación de arrays

Una de las operaciones más habituales que se realizan con *arrays* es la ordenación de los valores.

Como curiosidad, la primera tarea que se hizo mediante el uso de una computadora fue precisamente esa, la de ordenar *arrays* de tamaño gigantesco, de ahí que nosotros llamemos a las computadoras ordenadores.

La forma en que podemos ordenar los elementos de un *array* puede ser la que necesitemos, pero lo más normal es ordenarlos en orden ascendente o descendente.

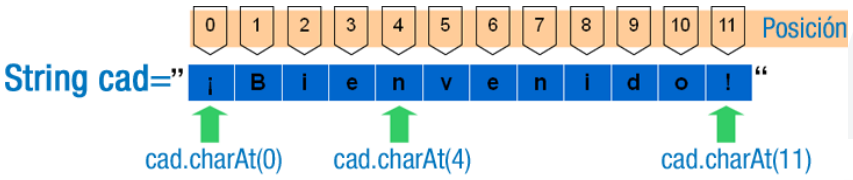
Existen **tres métodos básicos de ordenación** de *arrays*:

- Ordenación por el **método de la** burbuja.
- Ordenación por el **método de** selección.
- Ordenación por el **método de** inserción.

Estos tres métodos son los más básicos, es decir, son los más fáciles de implementar, pero son los menos eficientes en cuanto a tiempo se refiere. Si utilizamos estos métodos con *arrays* de pocos elementos, no será significativo el tiempo que se emplea cuando operamos con ellos; si los utilizamos con *arrays* con una cantidad considerable de elementos, podremos ver que tardan muchísimo en realizar las operaciones.

La eficiencia en orden de tiempo de un algoritmo se mide mediante la notación de complejidad O (O mayúscula). La de estos tres métodos sería O(n2), siendo n la cantidad de elementos que tenga el *array*. Por ejemplo, para un *array* de 100 elementos, la ordenación en cualquiera de estos métodos entrañaría una complejidad en torno a 100^2=10000.

Existen métodos mucho más eficientes (menos complejos) para la ordenación, como son el MergeSort o QuickSort, entre otros.



Código. 4. Cómo medir el tiempo de ejecución.

Vídeo 1. “Ordenación de un array”.
https://bit.ly/335cqP8

5.3. Búsqueda lineal

Las búsquedas de elementos en los arrays son algo fundamental. Mediante las búsquedas podremos comprobar si un elemento existe dentro de un array o no. El primer tipo de búsqueda que vamos a ver es la búsqueda lineal. Esta búsqueda es la más fácil de entender y de programar, pero, al igual que pasaba con los métodos de ordenación, al ser la más simple de implementar, será la menos eficiente, es decir, la que más tarda en encontrar el elemento.

El pseudocódigo[3] de la búsqueda es el siguiente:

- Recorro el *array*.
    - Si el elemento por donde voy es igual al que busco paro.
    - Aumento el contador de la posición y vuelvo al punto 1.

En cuanto a eficiencia en el tiempo, la búsqueda lineal es de complejidad O(n), siendo n el número de elementos del array.

En Java podremos hacer una función que implemente este método como se muestra en la siguiente figura:

```java
/**
 * Método que aplica la búsqueda lineal a un array
 *
 * @param array Array para buscar un elemento
 * @param elemento Elemento que se busca en el array
 * @return Devuelve true si el elemento está en el array, falso sino
 */
public static boolean busquedaLineal(int array[], int elemento) {
    boolean encontrado = false;

    for (int i = 0; i < array.length && !encontrado; i++) {
        if (array[i] == elemento) {
            encontrado = true;
        }
    }

    return encontrado;
}

public static void main(String[] args) {
    int[] array = new int[10]; // Declaración de un array
    // Relleno el array
    for (int i = 0; i < array.length; i++) {
        array[i] = i + 1;
    }

    // Busco el 4 en el array
    boolean estael4 = busquedaLineal(array, 4);
}
```

Código 5. Búsqueda lineal

5.4. Búsqueda binaria

La búsqueda binaria es un tipo de búsqueda en arrays mucho más eficiente que la búsqueda lineal, pero algo más compleja de implementar, y además necesita que el array esté obligatoriamente ordenado. Se basa en tomar los extremos del array y comparar el elemento central con el que estamos buscando. Si son iguales la búsqueda termina con éxito. En caso contrario se comprueba si el elemento buscado es menor o mayor que éste. Como el array está ordenado, si es menor, el elemento que buscamos estará en el lado izquierdo del array; si es mayor, en el derecho. En ambos casos se continúa la búsqueda en el lado que corresponda, actualizando los extremos y volviendo a comparar el elemento central del nuevo segmento. El proceso se repite hasta encontrar el elemento o hasta que los extremos del segmento se crucen, que implicará que el elemento no se encuentre en el array.

El orden de eficiencia de la búsqueda binaria es de complejidad O(log n), siendo n el número de elementos que tiene el array.

El algoritmo a seguir puede parecer un poco lioso de primeras, pero como vemos en la siguiente figura es bastante fácil de implementar.

3. El pseudocódigo es una representación en lenguaje humano de los pasos que realizar un algoritmo.

Puedes ver un ejemplo animado de estas búsquedas en: https://webs.um.es/ldaniel/iscyp17-18/fig/binary-and-linear-search-animations.gif

```java
/**
 * Método que aplica la búsqueda binaria a un array
 *
 * @param array Array para buscar un elemento
 * @param elemento Elemento que se busca en el array
 * @return Devuelve true si el elemento está en el array, falso sino
 */
public static boolean busquedaBinaria(int array[], int elemento) {
    boolean encontrado = false;
    int inicio = 0;
    int fin = array.length - 1;
    int posicion_buscar;

    while (inicio <= fin && !encontrado) {
        posicion_buscar = (inicio + fin) / 2;
        if (array[posicion_buscar] == elemento) {
            encontrado = true;
        } else {
            if (elemento > array[posicion_buscar]) {
                inicio = posicion_buscar + 1;
            } else {
                fin = posicion_buscar - 1;
            }
        }
    }

    return encontrado;
}
```

Código 6. Búsqueda binaria.

/ 6. Arrays multidimensionales

Hemos visto que tenemos los arrays unidimensionales, pero dentro de la tipología de arrays, también existen arrays que pueden tener más de una dimensión, los conocidos como **arrays multidimensionales.**

Este tipo de arrays pueden tener tantas dimensiones como necesitemos, pero con cada dimensión extra irán ganando un poco más de dificultad a la hora de trabajar con ellos.

Los más conocidos son los arrays de dos dimensiones, también conocidos como matrices. En matemáticas, se define una **matriz** como un rango bidimensional de valores, teniendo ésta filas y columnas.

En programación, el concepto de matriz es el mismo, y lo implementamos mediante un array con dos dimensiones. Esto implica que, para realizar todas las operaciones, tengamos que proporcionar esas dos dimensiones para poder localizar el elemento con el que queremos operar.

Las matrices se utilizan en informática para infinidad de situaciones, desde guardar los píxeles de una imagen, hasta el tablero de un juego como el ajedrez o el tres en raya.

La representación en memoria será la misma que la de un array unidimensional, solo que el número de elementos será el producto del número de filas por el número de columnas que tenga la matriz.

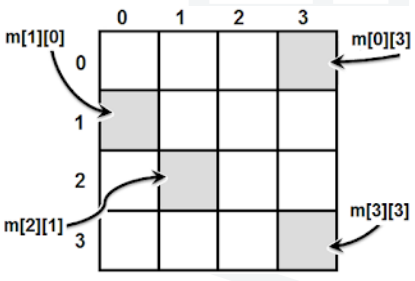


Fig. 4. Matriz de 4x4 (16 elementos)

6.1. Operaciones en arrays multidimensionales

Cuando usamos arrays multidimensionales, concretamente matrices, vamos a poder realizar las siguientes operaciones:

- **Inicialización:** Las matrices se crearán de la siguiente forma: *tipo[][] nombrevariable = new tipo[filas][columnas]*
- **Asignación de un valor:** Para asignar un valor a un elemento de una matriz tendremos que indicar tanto la fila como la columna donde se encuentra de la siguiente forma: *variable[fila][columna] = valor*
- **Lectura por teclado de un valor:** Podremos leer por teclado directamente el valor de un elemento de la matriz indicando la fila y la columna del mismo.
- **Mostrar por pantalla el contenido de una posición:** Podremos mostrar por pantalla directamente el valor de un elemento de la matriz indicando la fila y la columna del mismo.

La cantidad de filas y de columnas de las matrices serán números enteros positivos mayores que cero.

Es importante resaltar el hecho que la primera de las dimensiones que se tiene que indicar en las matrices son las filas, mientras que la segunda serán las columnas, para que así el tratamiento que hagamos de la matriz sea por filas. Al igual que con los arrays unidimensionales, podremos utilizar matrices en los métodos, pudiendo pasarlas como parámetros, crearlas localmente dentro del método y devolverlas.

Las matrices en los parámetros de los métodos se pasan por referencia, así que si las modificamos dentro de un método esa modificación será permanente.

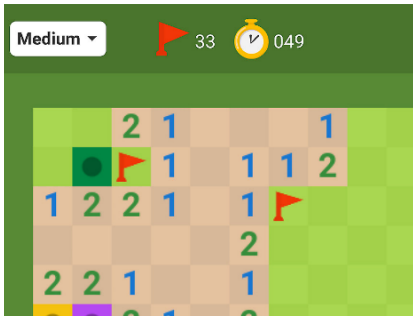


Fig. 5. Buscaminas: ejemplo de uso de una matriz.

Audio 1. "Recorrido de una matriz".
https://bit.ly/39I8zJv

/ 7. Caso práctico 2: "¿Cuándo usar un array unidimensional o multidimensional?"

Planteamiento: Pilar y José están reflexionando sobre los conceptos de *arrays* y matrices. Aún no tienen muy claro el uso de ellos y menos aún, cuándo hay que utilizar cada uno. José se apoya en la idea de que una matriz al fin y al cabo es una especie de array, pero con más elementos.

Nudo: ¿Qué piensas al respecto? ¿Se pueden utilizar siempre *arrays* y nunca matriz, o viceversa?

Desenlace: El uso diferenciado de *arrays* y matrices es muy sencillo de comprender. Imaginemos que necesitamos realizar un programa que dados una serie de números, muestre por pantalla cuál de ellos es el que más se repite.

Este, por ejemplo, es un caso muy claro de cuándo hay que usar *arrays* unidimensionales, ya que podríamos almacenar todos los números en un array e ir recorriéndolo para contar las veces que se repite cada uno de ellos.

Imaginemos ahora que necesitamos realizar el juego del Sudoku. Este, sin embargo, es un ejemplo muy claro del uso de una matriz, ya que la propia forma del tablero de Sudoku es una matriz en sí misma, por tanto, su representación e implementación será mucho más clara que si utilizásemos un array unidimensional, ya que tendríamos que idear una forma de hacer la relación entre la posición de los elementos del *array,* con las filas y las columnas de la matriz, cosa que puede ser algo complicada de hacer.

Por tanto, vemos que se nos van a presentar situaciones que claramente con *arrays* unidimensionales podremos solventarlas, pero seguro que, en otras muchas ocasiones, los *arrays* multidimensionales nos serán de gran ayuda.

Fig. 6. Una matriz es un array de arrays.

/ 8. Cadenas de caracteres

Cuando hablamos de cadenas de caracteres, nos estamos refiriendo a un array de caracteres. Éstos, aunque no te lo parezca, ya los hemos estado utilizando unidades atrás. Concretamente ha sido mediante la clase String.

Un *String* no es más que un array unidimensional de datos de tipo char. Podemos crear un array de tipo char de la siguiente forma:

```
char[] cadena = new char[10];
```

Con esta expresión, hemos declarado una cadena de caracteres de tamaño 10, pudiendo manipularla exactamente de la misma forma que hemos visto en los puntos anteriores.

Si no queremos sofisticar mucho los problemas, lo más sencillo es utilizar clase *String* para trabajar con cadenas, ya que esta clase, al ser interna de Java, ya está completamente implementada y nos ofrece una gran cantidad de métodos para poder utilizar.

La forma más habitual que vamos a encontrar para declarar y dar valor a una cadena de caracteres es la siguiente:

String cadena = *"Esto es una cadena";*

Dentro de las comillas dobles que forman la cadena podremos poner cualquier carácter sin restricción alguna.

Los *String* se conocen como cadenas dinámicas, ya que se irá ajustando su tamaño necesario automáticamente conforme se vaya modificando su valor.

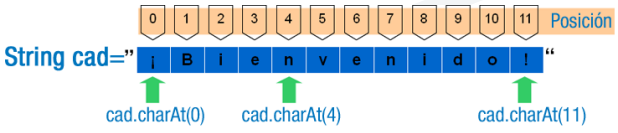


Fig. 7. Internamente, la clase String es un array de caracteres.

8.1. Operaciones con cadenas de caracteres

Ya sabemos que la clase String está implementada, pero ¿qué métodos nos ofrece para trabajar con cadenas?

En la siguiente tabla podemos ver algunos de los métodos más útiles que nos ofrece esta clase:

Método	Descripción
length()	Devuelve la longitud de la cadena
indexOf('caracter')	Devuelve la posición de la primera aparición de carácter
lastIndexOf('caracter')	Devuelve la posición de la última aparición de carácter
charAt(n)	Devuelve el carácter que está en la posición n
substring(n1,n2)	Devuelve la subcadena comprendida entre las posiciones n1 y n2-1
toUpperCase()	Devuelve la cadena convertida a mayúsculas
toLowerCase()	Devuelve la cadena convertida a minúsculas
equals("cad")	Compara dos cadenas y devuelve true si son iguales
equalsIgnoreCase("cad")	Igual que equals pero sin considerar mayúsculas y minúsculas
compareTo(OtroString)	Devuelve 0 si las dos cadenas son iguales. <0 si la primera es alfabéticamente menor que la segunda ó >0 si la primera es alfabéticamente mayor que la segunda.
compareToIgnoreCase(OtroString)	Igual que compareTo pero sin considerar mayúsculas y minúsculas.
valueOf(N)	Convierte el valor N a String. N puede ser de cualquier tipo.
replace (char viejoChar, char nuevoChar)	Remplaza en el string que invoca el método, el viejoChar por el nuevoChar, todas las apariciones.
replaceAll(String viejaString, String nuevaString)	Remplaza en el string que invoca al método la viejaString por la nuevaString. Se utiliza para reemplazar subcadenas.
replaceFirst (String viejaString, String nuevaString)	Reemplaza solo la primera aparición.

Tabla 1. Algunos métodos de la clase String.

Vídeo 2. "Expresiones regulares".
https://bit.ly/3jL4jgE

/ 9. Resumen y resolución del caso práctico de la unidad

A lo largo de este tema, hemos visto el **concepto de array,** cómo se representa en memoria, cómo podemos declararlos y algunas de las operaciones que podemos hacer ellos.

Hemos comprobado que las tres formas más simples de **ordenación de *arrays*** van a tardar lo mismo en ofrecer el resultado, y que no son muy óptimas en cuanto a tiempo, cuando los *arrays* son especialmente grandes.

La **búsqueda de elementos en los *arrays*** es otra de las operaciones básicas que hemos visto, habiendo dos tipos de búsqueda, **la líneal y la binaria**, siendo la binaria mucho mejor en cuanto a ejecución en tiempo.

También hemos descubierto que existen ***arrays* de dos dimensiones,** las conocidas matrices, cómo declararlas y operar con ellas.

Por último, hemos aprendido el uso básico de las **cadenas de caracteres** mediante el uso de la case String.

Resolución del caso práctico de la unidad

Cuando estamos aprendiendo nuestro primer lenguaje de programación, es muy común llegar a estos problemas, y no sabemos al principio cómo afrontarlos:

“Si no sé cuántos objetos voy a necesitar, ¿cómo puedo representar todos los clientes que necesito? ¿y si creo 10 clientes y me hacen falta más?”

Esta situación se va repetir muchísimo en el desarrollo de software, y su solución es bastante sencilla con la utilización de *arrays*.

Utilizando un *array* podremos almacenar todos los objetos que necesitemos, aunque siempre tendremos un límite (por ahora).

/ 10. Bibliografía

Colaboradores de Wikipedia. (2020, abril 26). *Vector (informática)* - Wikipedia, la enciclopedia libre. Recuperado 21 de mayo de 2020, de https://es.wikipedia.org/wiki/Vector_(inform%C3%A1tica)

colaboradores de Wikipedia. (2020, mayo 17). *Cadena de caracteres* - Wikipedia, la enciclopedia libre. Recuperado 21 de mayo de 2020, de https://es.wikipedia.org/wiki/Cadena_de_caracteres