

# DESARROLLO DE APLICACIONES WEB

## Unidad 6

### E s t r u c t u r a s   d e   D a t o s E s t á t i c o s

*1r DAW*

*IES La Mola de Novelda*

*Departament d'informàtica*

# ÍNDIX

1.- Introducció	3
2.- Arrays	3
2.1.- Definició	3
2.2.- Visualizació gràfica de un <i>array</i> unidimensional	4
2.3.- Arrays unidimensionals	4
2.4.- Operador new	6
2.5.- Manipulació de los datos dentro de un <i>array</i>	9
2.6.- Recorrer un <i>array</i>	11
2.7.- Clase Arrays	11
2.7.1.- Obtener el tamaño de un <i>array</i>	12
2.7.2.- Visualizar un <i>array</i>	12
2.7.3.- Inicialización de un <i>array</i>	12
2.7.4.- Comparación de dos <i>arrays</i>	13
2.7.5.- Ordenar un <i>array</i>	13
2.7.6.- Buscar dentro de un <i>array</i>	14
2.7.7.- Copiar arrays	14
2.8.- Arrays multidimensionales	15
2.8.1.- Arrays bidimensionales	16
2.8.2.- Arrays tridimensionales	18
2.8.3.- Arrays n-dimensionales	18
3.- Cadena de caracteres	20
3.1.- Inicialización de cadenas	20
3.2.- Comparaciones	21
3.2.1.- Igualdad	21
3.2.2.- Comparación alfabética	22
3.2.3.- Concatenación	23
3.2.4.- Obtención de caracteres	24
3.2.5.- Longitud de una cadena	25
3.2.6.- Búsqueda	25
3.2.7.- Comprobaciones	26
3.2.8.- Conversión	27
3.3.- Cadenas i <i>arrays</i> de caracteres	28
3.4.- Listado completo de métodos	29

## Unidad 6: ESTRUCTURAS DE DATOS ESTÁTICOS

### 1.- INTRODUCCIÓN

Los tipos de datos estructurados surgen de la necesidad de poder contestar a las siguientes preguntas:

- Cuántos valores se pueden almacenar en una variable?
- Queremos analizar las notas finales de un conjunto de estudiantes. Queremos saber la media de cada uno de ellos, qué estudiantes han aprobado, quién han superado un umbral de nota determinada, hacer gráficas de rendimiento, etc.... Suponemos que tenemos 678 estudiante. Cuántas variables harían falta para almacenar las notas de estos estudiantes?

Para poder solucionar estos tipos de problemas donde tenemos que utilizar un conjunto muy grande de variables que almacenan datos del mismo tipo, haremos uso de las estructuras de datos .

El tipo estructura de datos, a diferencia del tipo primitivo, es aquel que permite almacenar más de un valor dentro de una única variable, es decir, es una colección de datos (normalmente del tipo simple).

Las estructuras de datos se pueden clasificar, según la variabilidad de su tamaño durante la ejecución del programa, en **estáticos** y **dinámicos**. En este tema trataremos las estructuras de datos estáticas.

### 2.- ARRAYS

#### 2.1.-DEFINICIÓN

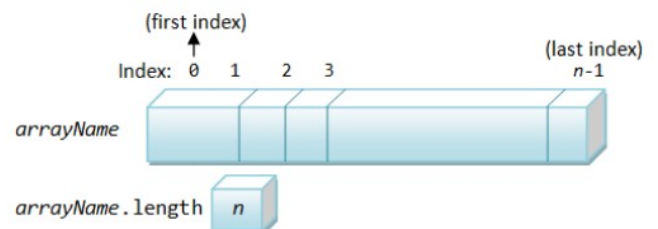
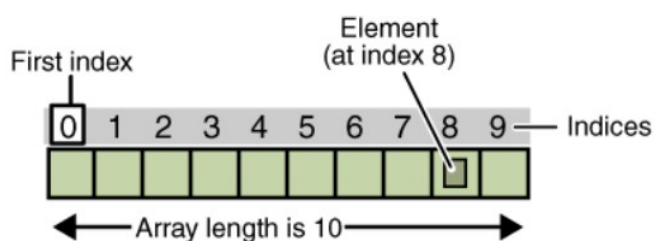
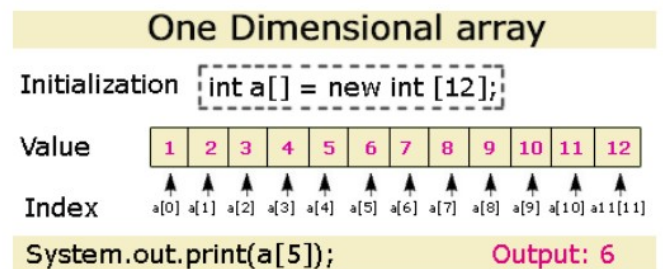
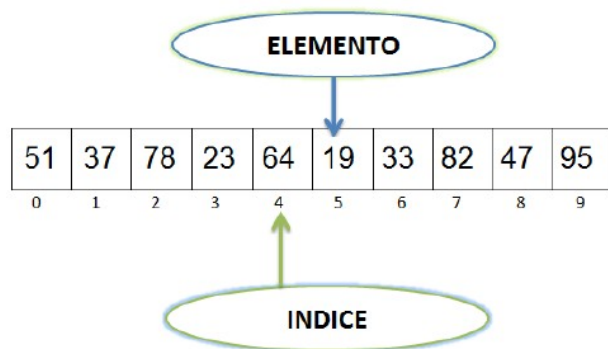
El tipo de datos compuesto **array** permite almacenar, en forma de secuencia, una cantidad predeterminada de valores pertenecientes al **mismo tipo de datos**.

Por lo tanto, un *array* es una colección de valores de un mismo tipo agrupado en la misma variable, de forma que se puede acceder a cada valor independientemente mediante un **índice**. Gracias al *array*, podemos generar un conjunto de variables con el mismo nombre, y será el **índice** del *array* el que diferenciará a cada variables. El **índice** siempre empieza por el valor **0**. Eso significa que el primer elemento de un *array* se encuentra en la posición cero.

Para Java, además, un *array* es un objeto que tiene propiedades las cuales se pueden

manipular.

## 2.2.-VISUALIZACIÓN GRÁFICA DE UN ARRAY UNIDIMENSIONAL



## 2.3.-ARRAYS UNIDIMENSIONALES

La declaración de un *array* unidimensional se hace utilizando las siguientes sintaxis (las dos son equivalentes):

```
tipo nom[ ];
```

```
tipo[ ] nom;
```

Una vez declarado hay que crearlo. La creación consistirá en indicar el tamaño o dimensión del *array*, es decir, la cantidad de elementos del mismo tipo que contendrá. Para ello utilizaremos el operador **new**. La dimensión siempre tiene que ser un número entero positivo mayor que cero.

```
nom = new tipo[dimensión];
```

También se puede hacer la declaración y creación en un solo paso:

```
tipo[ ] nom = new tipo[dimensión];
```

**Una vez creado el array, este ya no se podrá cambiar de tamaño.**

```
int edat[];
edat = new int[10];
```

A partir del siguiente ejemplo, modifica el tipo para comprobar a que se inicializa el *array* una vez declarado y creado. Utiliza tipos primitivos o básicos.

Del mismo modo que hemos hecho con las variables, los *arrays* los podemos inicializar, tanto en su definición como en su declaración. Fijaos en los siguientes ejemplos:

```
int[] arraysEnters1 = {3,6,34,12};
int arraysEnters2[] = new int[] {3,6,34,12};
```

Los dos *arrays* tienen la misma dimensión y están inicializados con los mismos valores

```
int dades[]; //Declaración array
dades = {1,2,4}; //No permitido.
//Sólo se puede hacer en la misma
//instrucción donde se declara
```

La inicialización de un *array* sólo se puede realizar en la misma instrucción donde se declara

Un *array* se puede inicializar las veces que hagan falta, pero tenemos que tener en cuenta que se perderá el contenido anterior.

```
int[] arraysEnters = {3,6,34,12};
arraysEnters = new int[25];
```

Un *array* es una referencia a valores que se almacena en la memoria mediante el operador **new**. Si éste lo volvemos a usar en la misma referencia, el anterior contenido queda sin referenciar y por lo tanto se pierde.

Inicialmente, arraysEnters es de dimensión 4 y se ha inicializado a 3, 6, 34 y 12. En la siguiente inicialización, arraysEnters es de dimensión 25 y se ha inicializado a 0 ya que es un *array* de enteros.

```
int notas[] = {1,3};
for(int i=0;i<2;i++)
    System.out.print(notas[i]+" ");
System.out.println();
```

```
int ejemplo[] = new int[] {2,4,6,8};
for(int i=0;i<4;i++)
    System.out.print(ejemplo[i]+" ");
System.out.println();
```

```
notas = ejemplo;
```

```
for(int i=0;i<4;i++)
    System.out.print(notas[i]+" ");
System.out.println();
```

SALIDA POR CONSOLA DEL CÓDIGO

```
1 3
2 4 6 8
2 4 6 8
```

Otra manera de inicializar un *array* es asignar a cada posición un valor determinado.

```
char[] v; //Declaración de la variable array
v = new char[4]; //Creación del array

//Asignación de valores al array
v[0] = 'm';
v[1] = 'a';
v[2] = 'm';
v[3] = 'a';
```

Para asignar un valor a cada posición del *array* utilizaremos el índice del *array*.

## 2.4.-OPERADOR NEW

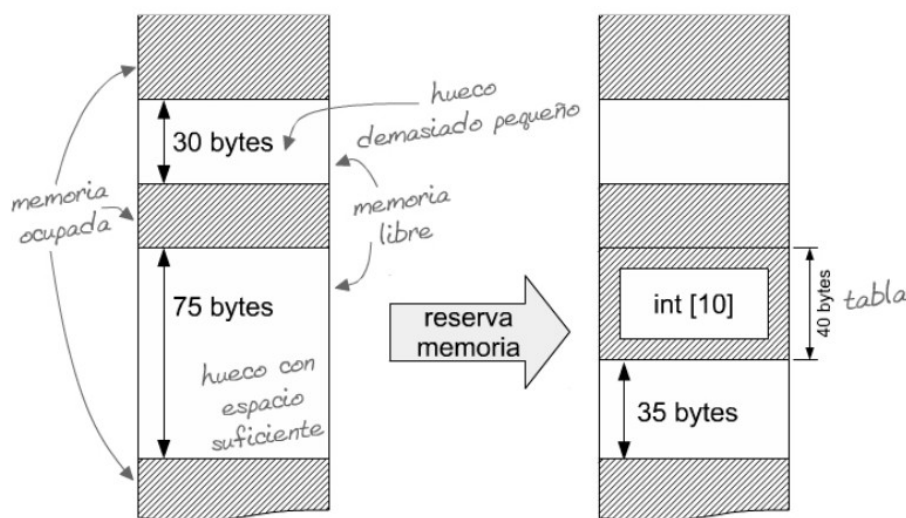
Cómo hemos visto anteriormente, una vez declarado un *array*, lo crearemos con el operador **new** (también se puede declarar y crear sin el operador **new** asignado-le valores: `int a[] = {1,2,3}`).

El operador **new** crea un *array* inicializando todas sus posiciones a un valor por defecto dependiente del tipo de datos primitivo que almacena (en el caso un *array* de enteros inicializa a cero todas las posiciones).

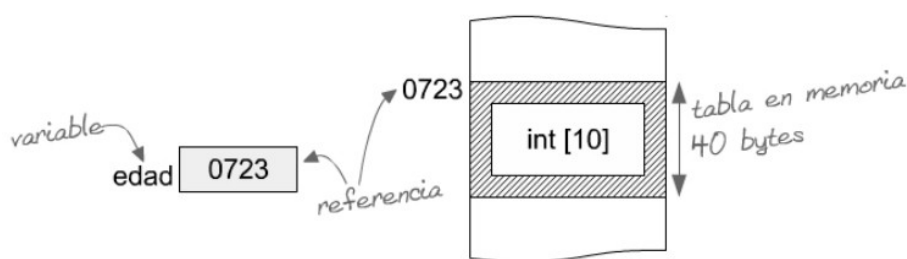
Para ver como funciona el operador **new**, vamos a suponer la siguiente instrucción:

```
int edad[ ];
edad = new int[10];
```

- En primer lugar calcula el tamaño físico que ocupará el *array*, es decir, el número de bytes que ocupará un *array* de 10 enteros. Recordamos que un entero ocupa 4 bytes, por lo tanto: 10 posiciones x 4 B = 40 B que ocupará el *array* de identificador *edad*.
- Se busca un espacio en memoria principal donde poder almacenar esos 40 B, de tal manera que los 10 enteros estén consecutivos.



- Una vez encontrado un espacio mayor o igual a 40 B, se reserva esa memoria marcándola como ocupada
- A continuación, se recorre cada una de esas 10 posiciones y se inicializa a 0 ya que es un *array* de enteros (cada 4 B será un 0). Si fuera de booleanos se pondría *false*.
- Por último, tenemos que referenciar el identificador *edad* al principio de la memoria reservada. Cada posición de memoria en un ordenador tiene una dirección que lo identifica. En Java, cada **dirección de memoria** se denomina **referencia**. Por lo tanto, al identificador del *array* **edad** se le asigna la primera posición de memoria donde empieza la memoria reservada para el *array*, como se puede ver a la siguiente figura:



- Fíjate en el que pasa si imprimimos la variable o identificador del *array* *edad*:

<pre>int edad[]; edad = new int[10]; System.out.println(edad);</pre>	<pre>[I@7344699f</pre>
--	------------------------

- Podemos ver que se obtiene la referencia guardada en la variable o identificador del *array* *edad*:



- Decimos que la variable **edad** referencia a un *array* de 10 enteros.

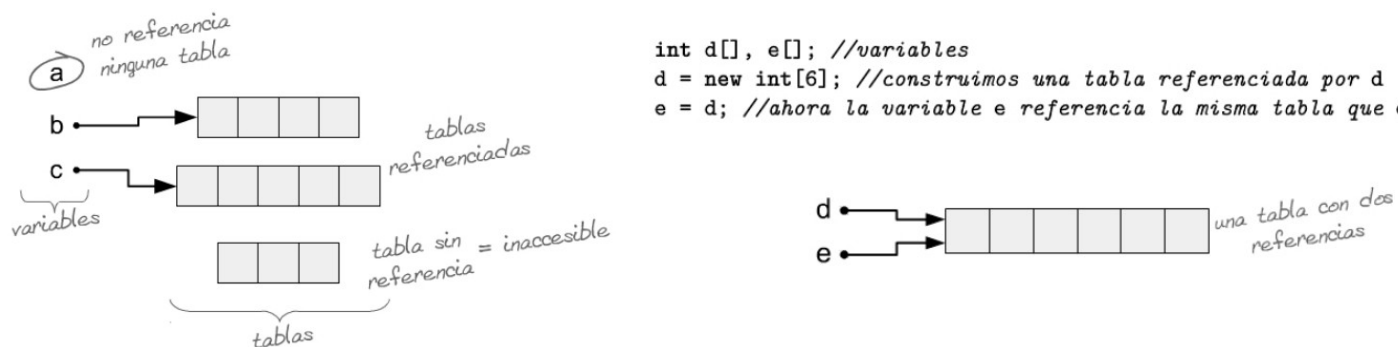
Una vez que ya hemos visto cómo funcionan las referencias, podemos plantear los siguientes escenarios:

- **int a[ ]** → Declaración de un *array* pero sin crearlo. La variable **a** no es operativa ya que no referencia a ninguna posición de la memoria principal.
- **new int [10]** → Creación del *array*, y por tanto reserva de memoria principal para esos 10

enteros, pero no se almacena la posición inicial de esa reserva en ninguna variable. Por lo tanto, la referencia a ese *array* se pierde. Esto quiere decir que no hay ninguna manera de poder utilizar esa memoria reservada.

En este caso, 40 B no son significativos, pero si durante la ejecución de un programa se hace reserva de grandes cantidades de memoria principal sin ser referenciadas por ninguna variable, éstas sería inaccesibles. Esto puede pasar por un mal diseño de la aplicación o bien de forma malintencionada. Java soluciona este problema ejecutando periódicamente el **recolector de basura** (*Garbage Collection*), el cual comprueba todas las reservas de memoria que hay, y si alguna está inaccesible, la destruye, dejando ese espacio libre para que pueda ser utilizado.

- Lo más común, como hemos dicho anteriormente, es **combinar** la **declaración** con la **creación** de los *arrays*.



La única condición para que una variable pueda referenciar a un *array* es que el tipo de variable y el de *array* coincidan.

Hasta ahora hemos visto la forma en que se asigna a una variable una referencia a un *array*, directamente con el operador **new** o bien mediante otra variable. Pues bien, hay otra manera de hacer el contrario, es decir, hacer que una variable que referencia a un *array* no referencie a nada, utilizando el literal de **null**, el cual significa vacío. Fíjate en la siguiente imagen:

```

int t1[], t2[]; //variables de tipo tabla entera

t1 = new int[100]; //t1 referencia una tabla de 100 elementos
t2 = t1; //ahora t2 también referencia la misma tabla

t1 = null; //anulamos t1: no referencia a ninguna tabla
           //la tabla con 100 elementos sigue siendo accesible desde t2

t2 = null; //anulamos t2: tampoco hace referencia a nada
           //la tabla es inaccesible: el recolector de basura se encargará de ella
    
```



## 2.5.-MANIPULACIÓN DE LOS DATOS DENTRO DE UN ARRAY

Los *arrays* son un poco especiales a la hora de ser manipulados, ya que no disponen de ninguna operación. No es posible usar el identificador del *array* directamente para invocar operaciones y así manipular los datos contenidos, tal como se puede hacer con las variables de otros tipos.

```
int a[] = {1,3,5};  
int b[] = {6,8,10};  
int c[] = a+b;
```

```
Prova.java:7: error: bad operand types for binary operator '+'  
    int c[] = a+b;  
                ^  
    first type: int[]  
    second type: int[]  
1 error
```

Para hacer operaciones con los datos almacenados dentro de los *arrays* tendremos que manipularlos de manera individual, posición por posición. Cada posición de un *array* tiene el mismo comportamiento que una variable. Además, cada posición viene identificada por un **índice**, el cual indica el orden de ese valor dentro de la estructura. Para acceder a cada posición tendremos que utilizar la siguiente sintaxis:

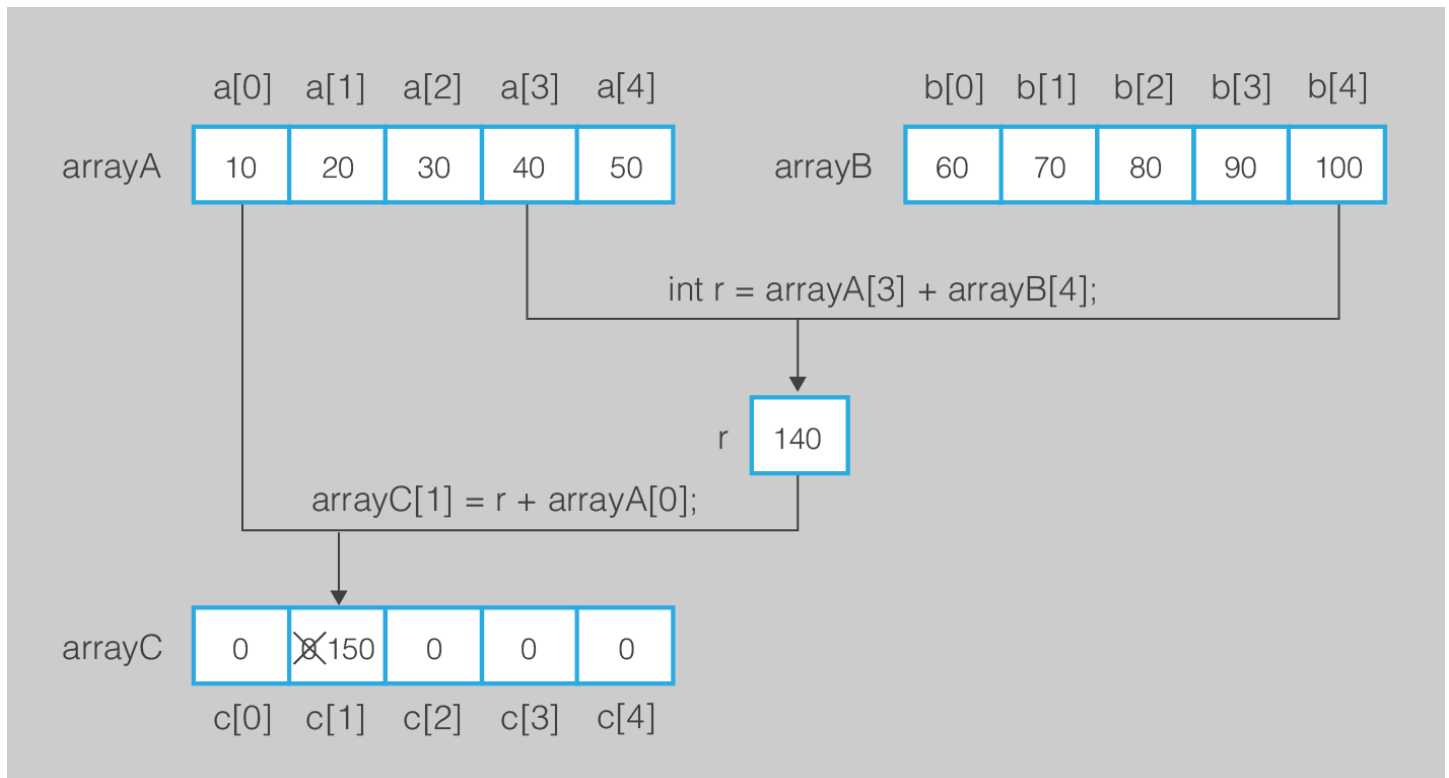
identificadorArray[índice]

El **rango de los índices** puede variar según el lenguaje de programación. En el caso de Java, estos van de **0**, para la primera posición, a (**mida – 1**), para la última. Por ejemplo, para acceder a las posiciones de un *array* de cinco posiciones usaríamos los índices 0, 1, 2, 3 y 4.

```
int a[] = {1,3,5};  
int b[] = {6,8,10};  
int c = a[1]+b[2];
```

Estamos sumando la **posición segunda** del *array* con identificador **a** y la **tercera posición** del *array* con identificador **b**. El resultado de sumar 3 + 10 se almacena en la variable entera **c**.

Ejemplo de manipulación de datos dentro de un *array*:



Lo más importante para acceder a una posición de un *array* es indicar un **índice correcto**. El índice nunca puede ser negativo y no puede superar su tamaño. Como hemos dicho anteriormente, el tamaño de un *array* es siempre fijo y no se puede cambiar. Por lo tanto, siempre sabremos que si la medida de un *array* es **n**, tendremos que utilizar un índice desde la **posición 0** hasta la **posición n-1**. Aun así, Java dispone de una herramienta que nos ayuda a controlar si un índice está dentro del rango admisible. Éste es el atributo **length**. Su sintaxis es:

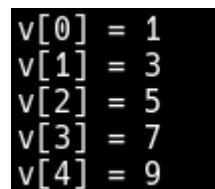
`identificadorArray.length`

El resultado de esta instrucción es el  
tamaño del *array*

## 2.6.-RECORRER UN ARRAY

Visitar todos o parte (subconjunto) de los elementos que conforma un *array* para realizar cualquier tipo de operación, es una operación muy habitual. La estructura de control más apropiada es el **for**.

```
int v[] = {1,3,5,7,9};  
  
for(int i=0;i<v.length;i++)  
    System.out.println("v["+i+"] = "+v[i]);  
System.out.println();
```



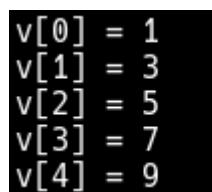
```
v[0] = 1  
v[1] = 3  
v[2] = 5  
v[3] = 7  
v[4] = 9
```

Un error muy frecuente cuando se hace recorrer un *array* es sobrepasarlo, es decir, acceder a posiciones del *array* que están fuera de su rango.

Para poder recorrer un *array* de una manera más práctica y simple, sin preocuparnos de los límites del índice, podemos utilizar una sintaxis alternativa al **for** conocida como **for-each**. En este formato tan solo tenemos que indicar el nombre del *array* que queremos recorrer y en qué variable irá colocándose el elemento que obtenemos en cada iteración del bucle. No tenemos que especificar en qué índice empieza y en cuál acaba. De esto se encarga Java.

En el siguiente ejemplo podemos ver como se va sacando uno a uno los elementos del *array* **v**, los cuales van depositándose en la variable **valor** que es del tipo **int**.

```
int v[] = {1,3,5,7,9};  
int i = 0;  
  
for(int valor:v){  
    System.out.println("v["+i+"] = "+valor);  
    i++;  
}
```



```
v[0] = 1  
v[1] = 3  
v[2] = 5  
v[3] = 7  
v[4] = 9
```

## 2.7.-CLASE ARRAYS

Cómo hemos visto en temas anteriores, la API de Java proporciona herramientas que nos facilita el trabajo de programación. Respecto a los *arrays*, la biblioteca de clases de Java incorpora la clase auxiliar **java.util.Arrays** que incluye algunas de las tareas que se suelen hacer con los *arrays*. De esa forma nos ahorramos tiempo de tener que implementarlas, además de tener la seguridad de que están probadas y son seguras de usar.

### 2.7.1.- Obtener el tamaño de un array

nombreVariable.length

### 2.7.2.- Visualizar un array

Cómo hemos visto anteriormente, si queremos visualizar el contenido de un *array* tendremos que recorrerlo mostrando cada uno de sus elementos con un **for** o con un **for-each**. Pero con **Arrays.toString (identificadorArray)** podemos obtener una cadena de texto que representa el contenido del *array*.

```
int v[] = {1,3,5,7,9};
```

```
System.out.println(Arrays.toString(v));
```

```
[1, 3, 5, 7, 9]
```

### 2.7.3.- Inicialización de un array

Por defecto un *array* se inicializa con valores específicos (0 para numéricos y *false* para booleanos). Si queremos inicializar un *array* con valores diferentes, tendremos que recorrer el *array* y asignar a cada elemento un valor. Pues bien, en la clase *Arrays* podemos utilizar el siguiente método para inicializar un *array* con identificador **v** a un valor **val** diferente al que Java tiene por defecto:

Array.fill (v,val)

```
int v[] = new int[5];
```

```
Arrays.fill(v,-1);
```

```
System.out.println(Arrays.toString(v));
```

```
[-1, -1, -1, -1, -1]
```

Pero puede ser que sólo nos interese inicializar elementos de un *array* con identificador **v** desde una posición inicial hasta una posición final con un valor **val** determinado.

Array.fill (v,inicial,final,val)

```
int v[] = new int[5];
```

```
Arrays.fill(v,2,4,-1);
```

```
System.out.println(Arrays.toString(v));
```

```
[0, 0, -1, -1, 0]
```

### 2.7.4.- Comparación de dos arrays

Dos *arrays* no se pueden comparar directamente con el operador `==` ya que este operador compara las referencias de los *arrays*, y no los elementos de los *arrays*. Podemos comparar si dos *arrays* con identificadores **v** y **w** son iguales mediante **for** o bien con el método de la clase *Arrays*:

`Arrays.equals(v,w)`

```
int v[] = new int[5];  
int w[] = new int[5];
```

```
System.out.println("v = "+Arrays.toString(v));  
System.out.println("w = "+Arrays.toString(w));  
System.out.println(Arrays.equals(v,w));  
Arrays.fill(v,-1);  
System.out.println("v = "+Arrays.toString(v));  
System.out.println(Arrays.equals(v,w));
```

```
v = [0, 0, 0, 0, 0]  
w = [0, 0, 0, 0, 0]  
true  
v = [-1, -1, -1, -1, -1]  
false
```

### 2.7.5.- Ordenar un array

La clase *Arrays* nos facilita un método para ordenar de forma ascendente los elemento de un *array*. Para ordenar el *array* con identificador **v** utilizaremos los siguiente métodos:

`Arrays.sort(v)`

```
int v[] = {2,9,4,8,3,2,5,1,3,8};  
System.out.println(Arrays.toString(v));  
Arrays.sort(v);  
System.out.println(Arrays.toString(v));
```

```
[2, 9, 4, 8, 3, 2, 5, 1, 3, 8]  
[1, 2, 2, 3, 3, 4, 5, 8, 8, 9]
```

También podemos ordenar sólo un subconjunto de elementos del *array* **v** indicando desde dónde (posición **inicial** incluida) hasta dónde (posición **final** excluida) (podéis ir al API de Java y ver su sintaxis y explicación):

`Arrays.sort(v, inicial, final)`

```
int v[] = {2,9,4,8,3,2,5,1,3,8};  
System.out.println(Arrays.toString(v));  
Arrays.sort(v,3,8);  
System.out.println(Arrays.toString(v));
```

```
[2, 9, 4, 8, 3, 2, 5, 1, 3, 8]  
[2, 9, 4, 1, 2, 3, 5, 8, 3, 8]
```

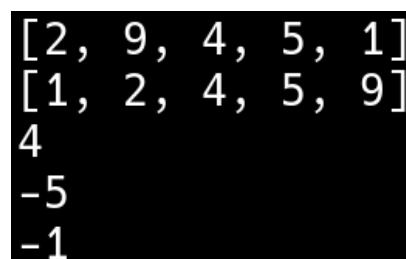
### 2.7.6.- Buscar dentro de un array

Si queremos encontrar un elemento dentro de un array, nos podremos encontrar con dos escenarios diferentes:

- Si el array está ordenado

La clase *Arrays* nos proporciona un método para poder buscar un valor **val** dentro de un array **v** (si el resultado es negativo, entonces no se ha encontrado el valor y me indica la posición que ocuparía en ese array):

```
int v[] = {2,9,4,5,1};
int pos;
System.out.println(Arrays.toString(v));
Arrays.sort(v);
System.out.println(Arrays.toString(v));
System.out.println(Arrays.binarySearch(v,9));
System.out.println(Arrays.binarySearch(v,8));
System.out.println(Arrays.binarySearch(v,-1));
```



```
[2, 9, 4, 5, 1]
[1, 2, 4, 5, 9]
4
-5
-1
```

- Si el array está desordenado

Una primera opción es ordenar el array y después utilizar el método anterior. Pero si no lo queremos utilizar por el motivo *que* sea, tendremos que hacer una búsqueda secuencial, la cual consiste en recorrer el array y comprobar el valor a encontrar con el contenido de todos los elementos del array.

**Realiza esta actividad.**

### 2.7.7.- Copiar arrays

El procedimiento para hacer una copia de un array en otro es:

- Creación de un nuevo array llamado destino del mismo tipo y tamaño.
- Recorrer el array original, copiando cada valor de él en la misma posición del array de destino.

**Realiza esta actividad.**

Pero la clase `Arrays` nos proporciona un método para copiar el **array origen** al **array destino**. En este método se construye un **array** de tamaño **longitud** y copia en ella el contenido del **array** origen. Si el tamaño de la copia es menor que el original, se copian los elementos que quepan. En el caso de que el nuevo **array** sea mayor que el original, se copia todo su contenido inicializando el resto de los elementos. El método es:

`destino = Arrays.copyOf(origen, longitud)`

```
int v[] = {2,9,4,5,1};
int a[],b[];

a = Arrays.copyOf(v,3);
b = Arrays.copyOf(v,10);
System.out.println(Arrays.toString(v));
System.out.println(Arrays.toString(a));
System.out.println(Arrays.toString(b));
```

```
[2, 9, 4, 5, 1]
[2, 9, 4]
[2, 9, 4, 5, 1, 0, 0, 0, 0, 0]
```

Hay una otro método que nos copia los elementos del **array origen** pero desde un índice **inicial** (no se incluye) hasta un índice **final** (se incluye). El método es:

`destino = Arrays.copyOfRange(origen, inicial, final)`

```
int v[] = {21,2,9,4,34,5,1};
int a[],b[];

a = Arrays.copyOfRange(v,2,4);
b = Arrays.copyOfRange(v,1,5);
System.out.println(Arrays.toString(v));
System.out.println(Arrays.toString(a));
System.out.println(Arrays.toString(b));
```

```
[21, 2, 9, 4, 34, 5, 1]
[9, 4]
[2, 9, 4, 34]
```

## 2.8.-ARRAYS MULTIDIMENSIONALES

Un **array multidimensional** es aquel en que para acceder a una posición concreta, en lugar de usar un solo valor como índice, se usa una secuencia de varios valores. Cada índice sirve como coordenada para una dimensión diferente. El más usado es el **array bidimensional**, de dos dimensiones, pero también se puede utilizar de cualquier dimensión, es decir, **arrays n-dimensionales**.

### 2.8.1.- Arrays bidimensionales

Un *array* bidimensional es un *array* de *arrays*. Un *array* bidimensional recibe el nombre de **matriz** y en este caso el *array* tendrá dos índices para recorrerlos, las filas y las columnas.

	0	1	2	3		Column 0	Column 1	Column 2
0	1	3	6	2	Row 0	x[0][0]	x[0][1]	x[0][2]
1	8	5	9	1	Row 1	x[1][0]	x[1][1]	x[1][2]
2	4	7	3	0	Row 2	x[2][0]	x[2][1]	x[2][2]

Su declaración se hace de la siguiente manera:

tipo nombre[ ][ ];

tipo[ ][ ] nombre;

Fijaos en el siguiente ejemplo donde se define un *array* bidimensional de enteros, se hace su lectura y a continuación se imprime por pantalla:

```
Scanner sc = new Scanner(System.in);
int[][] v = new int[3][2];
int files = v.length;
int columnes = v[0].length;

//LECTURA
for(int i = 0; i < files; i++){
    for(int j = 0; j < columnes; j++){
        System.out.print("v["+i+"]["+j+"] = ");
        v[i][j] = sc.nextInt();
    }
}

//ESCRITURA
for(int i = 0; i < files; i++){
    for(int j = 0; j < columnes; j++){
        System.out.printf("%4d", v[i][j]);
    }
    System.out.println();
}
```

```
v[0][0] = 1
v[0][1] = 2
v[1][0] = 3
v[1][1] = 4
v[2][0] = 5
v[2][1] = 6
    1    2
    3    4
    5    6
```



Los *arrays* bidimensional o multidimensionales no tienen porque tener el mismo número de columnas a las filas. Fijaos en el siguiente ejemplo donde se ha creado un *array* bidimensional de una forma creativa:

```
Scanner sc = new Scanner(System.in);
//3 arrays d'enters
int[][] notes = new int[3][];

//El primer array és de 2enters
notes[0] = new int [2];
notes[1] = new int [4];
notes[2] = new int [3];

//LECTURA
for(int i = 0; i < notes.length; i++){
    for(int j = 0; j < notes[i].length; j++){
        System.out.print("v["+i+"]["+j+"] = ");
        notes[i][j] = sc.nextInt();
    }
}

//ESCRIPURA
for(int i = 0; i < notes.length; i++){
    for(int j = 0; j < notes[i].length; j++){
        System.out.printf("%4d", notes[i][j]);
    }
    System.out.println();
}
```

```
v[0][0] = 1
v[0][1] = 2
v[1][0] = 3
v[1][1] = 4
v[1][2] = 5
v[1][3] = 6
v[2][0] = 7
v[2][1] = 8
v[2][2] = 9

    1    2
  3    4    5    6
  7    8    9
```

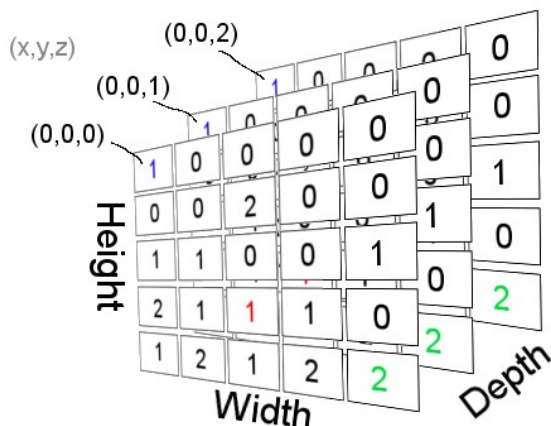
```
Scanner sc = new Scanner(System.in);
int[][] v = {{1,2,3},{4,5,6},{7,8,9}};
int[][] w = {{1,2},{4},{7,8,9}};

//ESCRIPURA v
for(int i = 0; i < v.length; i++){
    for(int j = 0; j < v[i].length; j++){
        System.out.printf("%4d", v[i][j]);
    }
    System.out.println();
}
System.out.println("=====");
//ESCRIPURA w
for(int i = 0; i < w.length; i++){
    for(int j = 0; j < w[i].length; j++){
        System.out.printf("%4d", w[i][j]);
    }
    System.out.println();
}
```

```
    1    2    3
    4    5    6
    7    8    9
=====
    1    2
    4
    7    8    9
```

### 2.8.2.- Arrays tridimensionales

Si añadimos una nueva dimensión a un *array* bidimensional podremos crear matrices con anchura, altura y profundidad, es decir, matrices tridimensionales. En este caso, se utilizarán tres índices  $[x][y][z]$ .



### 2.8.3.- Arrays n-dimensionales

Dibujar una matriz n-dimensional es complicado. Nosotros vivimos en un mundo 4-dimensional, con tres dimensiones para localizar un objeto en el espacio, y una cuarta dimensión, el tiempo, para ver la evolución de un objeto en movimiento.

Para poder tener una representación gráfica de una matriz n-dimensional, el que haremos será descomponer la matriz en otras más simples. Por ejemplo, una matriz de 5 dimensiones la podemos ver como una matriz tridimensional, donde cada elemento de la matriz es otra matriz bidimensional. De los 5 índices que hacen falta para identificar los elementos de una matriz 5-dimensional, podemos utilizar los tres primeros en la matriz tridimensional y utilizar los otros dos índices para situarnos en la segunda matriz bidimensional.

**Las matrices n-dimensionales son útiles para manipular la información atendiendo a criterios de clasificación**

- **Ejemplo:** Suponemos una máquina que procesa naranjas y necesitamos por motivos de calidad, clasificarlas y llevar la cuenta del número de frutas recogidas de cada tipo atendiendo a cinco criterios: diámetro, color, maduración, forma y peso.

Como que utilizamos 5 criterios, el más apropiado es una matriz 5-dimensional, donde cada dimensión corresponderá a un criterio de clasificación. Falta, para cada una de las dimensiones (criterios), formalizar una correspondencia entre los posibles valores reales de

un criterio (color: naranja, amarillo o verde; nivel de maduración: madura o inmadura; etc.) con los tamaños de cada dimensión, que utilizaremos como índices. Una posible correspondencia podría ser:

Diámetro:

0. Pequeño, diámetro <= 4 cm.
1. Medio, 4 cm <= diámetro <= 8 cm
2. Grande, diámetro > 8 cm

Maduración:

0. Pasada
1. Óptima
2. Ligeramente inmadura
3. Completamente inmadura

Peso:

0. Menos de 100 g
1. Entre 100 y 200 g
2. Entre 200 y 300 g
3. Entre 300 y 400 g
4. Entre 400 y 500 g
5. Más de 500 g

Color:

0. Naranja
1. Amarillo
2. Verde

Forma:

0. Redonda
1. Otra forma

Vamos a crear la variable **taronges** como una **matriz de 5 dimensiones** donde cada una de ellas representa un criterio:

taronges[diámetro][color][maduración][forma][peso]

La declaración y creación de la variable será:

```
int taronges [][][][][];  
taronges = new [3][3][5][2][6];
```

Si la máquina contabiliza 25 naranjas con las siguientes propiedades:

1. Diámetro de 11 cm: primer índice 2
2. De un color naranja intenso: segundo índice 0
3. En su punto óptimo de maduración: tercer índice 1.
4. La forma es redondeada: cuarto índice 0.
5. Pesa 285 g: quinto índice 2.

Haremos la siguiente asignación:

```
taronges[2][0][1][0][2] = 25;
```

### 3.- CADENA DE CARACTERES

Las cadenas, un conjunto secuencial de caracteres, se manipulan mediante la clase **String**. Esta clase funciona de manera dual. Por una parte, de manera general, tiene un funcionamiento no estático; pero al mismo tiempo, dispone de métodos que si que lo son.

Para definir variables del tipo **String** lo haremos de la manera habitual:

```
String cad; //cad es una variables del tipo cadena
```

Una variable del tipo **String** almacenará una cadena de caracteres, que podrá venir o de otra cadena o de un literal. Un literal cadena consiste en un texto que está entre dobles comillas ("). Ejemplos de literales:

```
"Hola \n"
```

```
"En un lugar de la mancha"
```

```
"Un corazón: \u2661" // Recorda que el número és hexadecimal
```

#### 3.1.-INICIALIZACIÓN DE CADENAS

Igual que hacemos en la clase **Scanner**, podemos utilizar **new** para crear y asignar un valor a una variable del tipo **String**. Pero también lo podemos hacer como cualquier variable:

```
//LAS DOS FORMAS SON EQUIVALENTES
String a = new String("Literal cadena");
String b = "Un altre literal cadena";
```

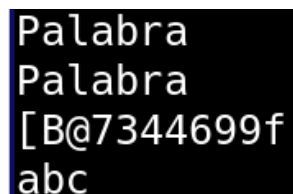
Recordad el uso de las dobles comillas dentro del literal de una cadena:

```
String cad = "Mi perro \"Perico\" es de color blanco";
System.out.print(cad);
```

Otra forma de crear **String** es a partir de *arrays* (hay que mirar la API para ver a partir de qué *arrays* se pueden crear un **String**):

```
char[] palabra = {'P','a','l','a','b','r','a'}; //Array de char
String cadena = new String(palabra);
byte[] datos = {97,98,99};
String codificada = new String (datos);

System.out.println(palabra);
System.out.println(cadena);
System.out.println(datos);
System.out.println(codificada);
```



Muchas veces queremos representar un valor del tipo primitivo en forma de cadena. Por ejemplo, el valor entero 1234 (mil doscientos treinta y cuatro) lo queremos convertir a la cadena "1234", es decir, con la cadena formada por los caracteres '1', seguido por el carácter '2' y '3' y finalizado con el carácter '4'. Para hacer esta conversión utilizaremos el método:

```
static String valueOf (tipo valor);  
  
String cad;  
cad = String.valueOf(1234); //cad = "1234"  
cad = String.valueOf(-12.34); //cad = "-12.34"  
cad = String.valueOf('C'); //cad = "C"  
cad = String.valueOf(false); //cad = "false"
```

### 3.2.-COMPARACIONES

Los operadores ==, <, <=, >, >= no están disponibles directamente para las cadenas de caracteres. En su lugar, disponemos de unos métodos que incorpora la clase **String**.

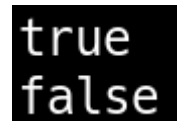
#### 3.2.1.- Igualdad

El operador == no se puede utilizar con **String** ya que es una clase y no un tipo primitivo. Para comparar dos cadenas utilizaremos los siguientes métodos:

- **boolean equals (String otraCadena);**

Compara el contenido de la cadena que invoca el método y otraCadena.

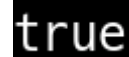
```
String cad1 = "Hola mundo";  
String cad2 = "Hola mundo";  
String cad3 = "Hola buenos días";  
boolean iguales;  
iguales = cad1.equals(cad2);  
System.out.println(iguales);  
iguales = cad1.equals(cad3);  
System.out.println(iguales);
```



- **boolean equalsIgnoreCase (String otraCadena);**

Igual que *equals* pero sin tener en cuenta las mayúsculas ni las minúsculas.

```
String cad1 = "Hola Mundo";  
String cad2 = "Hola mundo";  
  
boolean iguales;  
iguales = cad1.equalsIgnoreCase(cad2);  
System.out.println(iguales);
```

true

- **boolean regionMatches (int inicio, String otraCadena, int inicioOtra, int num);**

Compara dos fragmentos de cadenas: el primero corresponde a la cadena que invoca y empieza en el carácter con índice inicio; y el segundo corresponde a *otraCadena* y empieza en el carácter con índice inicioOtra. Los dos fragmentos tienen una longitud de num caracteres. El método devuelve *true* o *false* para indicar si las regiones coinciden.

```
boolean b;  
String cad = "Mi perro ladra mucho";  
String otra = "Un bonito perro blanco";  
  
b = cad.regionMatches(3, otra, 10, 5);  
System.out.println(b);
```

true

- **boolean regionMatches (boolean ignora, int inicio, String otraCadena, int inicioOtra, int num);**

Igual que el método anterior, pero si *ignora* es *true* entonces la comprobación se hace ignorando las mayúsculas y las minúsculas.

### 3.2.2.- Comparación alfabética

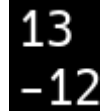
Se pueden comparar dos cadenas de caracteres alfabéticamente. La orden el marca la posición dentro del alfabeto, y las letras mayúsculas se consideran anteriores a las minúsculas. La comparación se hace mirando el primer carácter diferente a la cadena. Por ejemplo, "monitor" y "monzón", la comparación se hace mirando el cuarto carácter, con índice 3, de cada cadena, que es el primer carácter diferente.

Los métodos disponibles para comparar alfabéticamente cadenas son:

- **int compareTo (String cadena);**

Los posibles valores que puede devolver son: **0** si las cadenas son exactamente iguales, **negativo** si la cadena que invoca es menor alfabéticamente que la cadena pasada como parámetro, **positivo** si la cadena que invoca es mayor que la cadena pasada.

```
String cad1 = "Alondra";
String cad2 = "Nutria";
String cad3 = "Zorro";
//cad2 és major que cad1
System.out.println(cad2.compareTo(cad1));
System.out.println(cad2.compareTo(cad3));
```



```
13
-12
```

- **int compareToIgnoreCase (String cadena);**

Hace la misma comparación alfabética pero sin diferenciar mayúsculas ni minúsculas.

### 3.2.3.- Concatenación

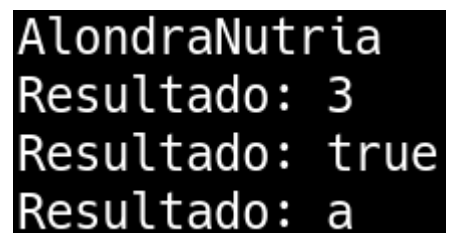
Con el operador **+** podemos unir o concatenar dos cadenas.

```
String cad1 = "Alondra";
String cad2 = "Nutria";
String cad3 = cad1 + cad2;
System.out.println(cad3);
```

```
String a,b,c;
//SON EQUIVALENTES
//a = "Resultado: " + String.valueOf(3);
a = "Resultado: " + 3;
System.out.println(a);

//SON EQUIVALENTES
//b = "Resultado: " + String.valueOf(true)
b = "Resultado: " + true;
System.out.println(b);

//SON EQUIVALENTES
//c = "Resultado: " + String.valueOf('a');
c = "Resultado: " + 'a';
System.out.println(c);
```



```
AlondraNutria
Resultado: 3
Resultado: true
Resultado: a
```



Otra forma de concatenar cadenas es con el método:

- **String concat (String cad);**

```
String cad1 = "Alondra";  
String cad2 = "Nutria";  
String cad3 = cad1.concat(cad2);  
System.out.println(cad3);
```

AlondraNutria

### 3.2.4.- Obtención de caracteres

Todos los caracteres pueden ser identificados mediante la posición que ocupa a la cadena **String** del mismo modo que se hace en un *array*. Para obtenerlos, haremos uso de métodos que nos ofrece la clase **String**.

- **char charAt (int posición);** //Obtener un carácter

```
String cad = "Alondra";  
System.out.println(cad.charAt(2));  
System.out.println(cad.charAt(20));
```

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 20  
at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:48)  
at java.base/java.lang.String.charAt(String.java:1515)  
at Prova.main(Prova.java:9)
```

- **String substring (int inicial);** //Devuelve la subcadena formada desde la posición inicial hasta el final de la cadena. Devuelve una copia y la cadena que invoca el método no se modifica.
- **String substring (int inicial, int final);** //Devuelve subcadena comprendida entre posición inicial y posición final.

```
String cad = "Alondra es mala";  
System.out.println(cad.substring(5));  
System.out.println(cad.substring(5,8));
```

ra es mala  
ra

Como no, si utilizamos un índice que esté fuera del rango, se producirá un error y finalizará la ejecución del programa.

- **String trim ();** //Devuelve una copia de la cadena eliminando desde el principio hasta el final, los caracteres que hay en blanco, así como tabuladores. La cadena que invoca el método no se modifica.



```
String cad = "    Alondra es mala    de narices    ";
System.out.println(cad.length());
String cad1 = cad.trim();
System.out.println(cad1);
System.out.println(cad1.length());
```

```
38
Alondra es mala    de narices
29
```

- **String [ ] split (String regex);** //Divide una cadena en una matriz de subcadenas con un delimitador específico regex. Nos devuelve un *array* de cadenas (**String**).

```
String strMain = "Alfa,Beta,Delta,Gamma,Sigma";
String[] arrSplit = strMain.split(",|");

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

```
Alfa
Beta
Delta
Gamma
Sigma
```

### 3.2.5.- Longitud de una cadena

- **int length ();** //Devuelve el número de caracteres o longitud de una cadena. Lo hemos utilizado al ejemplo anterior.

### 3.2.6.- Búsqueda

Dentro de una cadena hay la posibilidad de buscar un carácter o una subcadena.

- **int indexOf (int c);** //Busca la primera ocurrencia del carácter *c* desde el principio (posición 0) de la cadena que invoca el método. Se pone un *char*, a pesar de que en el parámetro de la función hay un *int*. Se hace la conversión automática entre el tipo *char* y *int*. Se devuelve el índice donde está el carácter, o un -1 si no se ha encontrado
- **int indexOf (String cadena);** //Buscar la primera ocurrencia.

```
String cad = "Mi perro se llama Perico";

System.out.println(cad.indexOf('j'));
System.out.println(cad.indexOf('e'));
System.out.println(cad.indexOf("hola"));
System.out.println(cad.indexOf("perro"));
```

```
-1
4
-1
3
```

- **int indexOf (int c, int pos);** //Busca la primera ocurrencia del carácter *c* desde la posición pos hasta el final de la cadena que invoca el método.
- **int indexOf (String cadena, int pos);** //Buscar la primera ocurrencia de cadena a partir de la posición pos de la cadena que invoca el método.

```
String cad = "Mi perro se llama Perico";

System.out.println(cad.indexOf('M',1));
System.out.println(cad.indexOf('e',5));
System.out.println(cad.indexOf("er",5));
System.out.println(cad.indexOf("er",2));
```

```
-1
10
19
4
```

Todos estos métodos hacen la busca de izquierda a derecha. Para hacer la busca de derecha a izquierda, es decir, empezando por el final, los métodos a utilizar son:

- `int lastIndexOf (int c);`
- `int lastIndexOf (String cadena);`
- `int lastIndexOf (int c, int pos);`
- `int lastIndexOf (String cadena, int pos);`

### 3.2.7.- Comprobaciones

Son métodos que devuelven un valor booleano indicando el éxito o el fracaso de la consulta.

- `boolean isEmpty ();` //Indica si la cadena está vacía.
- `boolean contains (Charsequence subcadena);` //Devuelve un *true* si en la cadena que invoca el método se encuentra esa subcadena. **CharSequence** es una clase. No nos tenemos que preocupar, porque Java realiza la conversión automática desde la clase **String**. Por lo tanto, la podemos utilizar pasándole directamente una cadena.

```
String cad1 = "", cad2 = " ";
String cad = "Pepito de los palotes";

System.out.println(cad1.isEmpty());
System.out.println(cad2.isEmpty());
System.out.println(cad.contains("los"));
System.out.println(cad.contains("loss"));
```

```
true
false
true
false
```

También podemos comprobar en las cadenas si empiezan (prefijo) o acaban (sufijo) por una subcadena. Los métodos que podemos utilizar son:

- `boolean startsWith (String prefijo);` //Comprueba si la cadena que invoca el método empieza por la cadena prefijo que se le pasa como parámetro.

- **boolean startsWith (String prefijo, int inicio);** //Lo mismo que el anterior pero empieza a comprobar a partir de inicio.
- **boolean endsWith (String sufijo);** //Comprueba si la cadena que invoca el método finaliza con la cadena sufijo que se le pasa como parámetro.

```
String cad = "Hola mundo cruel";  
String p = "Hola", s = "mun";  
  
System.out.println(cad.startsWith("Hola"));  
System.out.println(cad.startsWith("Hol"));  
System.out.println(cad.startsWith("hola"));  
  
System.out.println(cad.startsWith(p,5));  
System.out.println(cad.startsWith(s,5));
```

```
true  
true  
false  
false  
true
```

### 3.2.8.- Conversión

Una cadena se puede transformar sustituyendo todas las letras a minúscula o mayúscula. Es útil por ejemplo para los valores que provienen de un formulario y en que cada usuario puede escribir de un forma u otra. Disponemos de los siguientes métodos:

- **String toLowerCase ();** //Se devuelve una copia de la cadena que invoca el método donde todas las letras se han sustituido por minúsculas. Tan solo se convierten las letras; el resto de caracteres se mantienen igual.
- **String toUpperCase ();** //Similar al anterior pero convierte todas las letras a mayúsculas.
- **String replace (char car, char otro);** //Devuelve un copia de la cadena que invoca el método donde se han sustituido las ocurrencias del carácter car por el carácter otro.

```
String cad = "HoLa 23: mUnDo CrUel!...";  
String cad1 = "Hola mundo.";   
  
System.out.println(cad.toLowerCase());  
System.out.println(cad.toUpperCase());  
  
System.out.println(cad1.replace('o','\u2661'));
```

```
hola 23: mundo cruel!...  
HOLA 23: MUNDO CRUEL!...  
H♡la mund♡.
```

### 3.3.-CADENAS I ARRAYS DE CARACTERES

Hay una relación entre las cadenas de la clase **String** y los *arrays* de caracteres **char[ ]**, hasta el punto de que en algunos lenguajes de programación no existe el tipo cadena, sino *arrays* de caracteres. En Java, las dos se pueden convertir sin ningún problema unas en otras. Por ejemplo, es más fácil manipular los caracteres de una cadena si la convertimos a *arrays* de caracteres puesto que el acceso es más directo.

Tipo	Descripción	Ejemplo										
String	Cadena de caracteres	"Hola mundo"										
char []	Arrays de caracteres	<table><tr><td>'H'</td><td>'o'</td><td>'l'</td><td>'a'</td><td>' '</td><td>'m'</td><td>'u'</td><td>'n'</td><td>'d'</td><td>'o'</td></tr></table>	'H'	'o'	'l'	'a'	' '	'm'	'u'	'n'	'd'	'o'
'H'	'o'	'l'	'a'	' '	'm'	'u'	'n'	'd'	'o'			

- **char [ ] toCharArray ();** //Crea y devuelve un *array* de caracteres con el contenido de la cadena que ha invocado el método.
- **static String valueOf (char [ ] a);** //Realiza el proceso inverso al anterior. Crea una cadena de caracteres (**String**) a partir del *array* que se le pasa como argumento.
- **static String valueOf (char [ ] a, int inicio, int numero);** //Similar al anterior. Devuelve la cadena formada por un subconjunto de los caracteres del *array* **a**. El parámetro inicio es el índice del primer elemento del *array* que nos interesa y numero es el número de caracteres que incluirá a la cadena.

```
String cad = "Hola mundo";
char[] c;
int i = 0;

c = cad.toCharArray();

for(char valor:c){
    System.out.println("c["+i+"] = "+valor);
    i++;
}
System.out.println(Arrays.toString(c));

char d[] = {'H','o','l','a'};
cad = String.valueOf(d);
System.out.println(cad);

char[] e = {'a','e','i','o','u','A','E','I','O'};
cad = String.valueOf(e,2,4);
System.out.println(cad);
```

```
c[0] = H
c[1] = o
c[2] = l
c[3] = a
c[4] = 
c[5] = m
c[6] = u
c[7] = n
c[8] = d
c[9] = o
[H, o, l, a, , m, u, n, d, o]
Hola
iouA
```

### 3.4.-LISTADO COMPLETO DE MÉTODOS

método	descripción
<b>char charAt(int index)</b>	Proporciona el carácter que está en la posición dada por el entero <i>index</i> .
<b>int compareTo(String s)</b>	Compara las dos cadenas. Devuelve un valor menor que cero si la cadena <i>s</i> es mayor que la original, devuelve 0 si son iguales y devuelve un valor mayor que cero si <i>s</i> es menor que la original.
<b>int compareToIgnoreCase(String s)</b>	Compara dos cadenas, pero no tiene en cuenta si el texto es mayúsculas o no.
<b>String concat(String s)</b>	Añade la cadena <i>s</i> a la cadena original.
<b>String copyValueOf(char[] data)</b>	Produce un objeto <b>String</b> que es igual al array de caracteres <i>data</i> .
<b>boolean endsWith(String s)</b>	Devuelve <b>true</b> si la cadena termina con el texto <i>s</i>
<b>boolean equals(String s)</b>	Compara ambas cadenas, devuelve <b>true</b> si son iguales
<b>boolean equalsIgnoreCase(String s)</b>	Compara ambas cadenas sin tener en cuenta las mayúsculas y las minúsculas.
<b>byte[] getBytes()</b>	Devuelve un array de caracteres que toma a partir de la cadena de texto
<b>void getBytes(int srcBegin, int srcEnd, char[] dest, int dstBegin);</b>	Almacena el contenido de la cadena en el array de caracteres <i>dest</i> . Toma los caracteres desde la posición <i>srcBegin</i> hasta la posición <i>srcEnd</i> y les copia en el array desde la posición <i>dstBegin</i>
<b>int indexOf(String s)</b>	Devuelve la posición en la cadena del texto <i>s</i>
<b>int indexOf(String s, int primeraPos)</b>	Devuelve la posición en la cadena del texto <i>s</i> , empezando a buscar desde la posición <i>PrimeraPos</i>
<b>int lastIndexOf(String s)</b>	Devuelve la última posición en la cadena del texto <i>s</i>

método	descripción
<b>int lastIndexOf(String s, int primeraPos)</b>	Devuelve la última posición en la cadena del texto <i>s</i> , empezando a buscar desde la posición <i>PrimeraPos</i>
<b>int length()</b>	Devuelve la longitud de la cadena
<b>String replace(char carAnterior, char ncarNuevo)</b>	Devuelve una cadena idéntica al original pero que ha cambiando los caracteres iguales a <i>carAnterior</i> por <i>carNuevo</i>
<b>String replaceFirst(String str1, String str2)</b>	Cambia la primera aparición de la cadena <i>str1</i> por la cadena <i>str2</i>
<b>String replaceFirst(String str1, String str2)</b>	Cambia la primera aparición de la cadena uno por la cadena dos
<b>String replaceAll(String str1, String str2)</b>	Cambia la todas las apariciones de la cadena uno por la cadena dos
<b>String startsWith(String s)</b>	Devuelve <b>true</b> si la cadena comienza con el texto <i>s</i> .
<b>String substring(int primeraPos, int segundaPos)</b>	Devuelve el texto que va desde <i>primeraPos</i> a <i>segundaPos</i> .
<b>char[] toCharArray()</b>	Devuelve un array de caracteres a partir de la cadena dada
<b>String toLowerCase()</b>	Convierte la cadena a minúsculas
<b>String toLowerCase(Locale local)</b>	Lo mismo pero siguiendo las instrucciones del argumento <i>local</i>
<b>String toUpperCase()</b>	Convierte la cadena a mayúsculas
<b>String toUpperCase(Locale local)</b>	Lo mismo pero siguiendo las instrucciones del argumento <i>local</i>
<b>String trim()</b>	Elimina los blancos que tenga la cadena tanto por delante como por detrás
<b>Static String valueOf(tipo elemento)</b>	Devuelve la cadena que representa el valor <i>elemento</i> . Si <i>elemento</i> es booleano, por ejemplo devolvería una cadena con el valor <b>true</b> o <b>false</b>