

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

---

CADENA DE CARACTERES Y ARRAYS.....	2
1. INTRODUCCIÓN A LAS ESTRUCTURAS DE ALMACENAMIENTO.....	2
2. CADENAS DE CARACTERES. ....	3
2.1 OPERACIONES AVANZADAS CON CADENAS DE CARACTERES .....	4
2.2 EXPRESIONES REGULARES .....	11
3. CREACIÓN DE ARRAYS. ....	15
3.1 USO DE ARRAYS UNIDIMENSIONALES.....	17
3.2 INICIACIÓN.....	18
3.3 MOSTRAR CONTENIDO DE UN ARRAY.....	20
4. ARRAYS MULTIDIMENSIONALES.....	23
4.1 USO DE LOS ARRAYS MULTIDIMENSIONALES .....	24
4.2 INICIACIÓN DE ARRAYS MULTIDIMENSIONALES .....	26
4.3 MOSTRAR CONTENIDO DE UN ARRAY MULTIDIMENSIONAL.....	27
ANEXO I.....	34
ANEXO II: SABER MÁS ACERCA DE ARRAYS.....	36

## CADENA DE CARACTERES Y ARRAYS

---

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

Lo mismo ocurre en otros casos. Para poder realizar ciertas aplicaciones se necesita poder manejar datos que van más allá de meros **datos simples** (números y letras). A veces, los datos que tiene que manejar la aplicación son **datos compuestos**, es decir, datos que están compuestos a su vez de varios datos más simples. Por ejemplo, una persona está compuesta por varios datos, los datos podrían ser el nombre del cliente, la dirección donde vive, la fecha de nacimiento, etc.

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

Más adelante veremos esas soluciones adicionales. En este tema nos centraremos en las cadenas de caracteres y en los vectores o arrays.

## 1. INTRODUCCIÓN A LAS ESTRUCTURAS DE ALMACENAMIENTO.

---

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

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

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

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

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

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

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

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

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

## 2. CADENAS DE CARACTERES.

Probablemente, una de las herramientas que más utilizarás cuando estés trabajando con cualquier lenguaje de programación son las cadenas de caracteres. Las cadenas de caracteres son estructuras de almacenamiento que permiten guardar una secuencia de caracteres de casi cualquier longitud. Y la pregunta ahora es, ¿qué es un carácter?

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

### Para saber más

Si quieres, puedes profundizar en la codificación de caracteres leyendo el siguiente artículo de la Wikipedia. [Codificación de caracteres.](#)

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

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

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

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

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

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



### Reflexiona

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

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

Mostrar retroalimentación

Pues en realidad hay 4 instancias. La primera instancia es la que se crea con el literal de cadena "Ejemplo de cadena 1". El segundo literal, "Ejemplo de cadena 2", da lugar a otra instancia diferente a la anterior. El tercer literal, "Ejemplo de cadena 3", es también nuevamente otra instancia de *String* diferente. Y por último, al crear una nueva instancia de la clase *String* a través del operador *new*, se crea un nuevo objeto *String* copiando para ello el contenido de la cadena que se le pasa por parámetro, con lo que aquí tenemos la cuarta instancia del objeto *String* en solo una línea.



### Recomendación

Nota de rendimiento:

Para conservar memoria, Java trata a todas las literales de cadena con el mismo contenido como un solo objeto *String* que tiene muchas referencias.

## 2.1 OPERACIONES AVANZADAS CON CADENAS DE CARACTERES

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

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

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

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

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

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

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

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

El método **toString()** es un método disponible en todas las clases de Java. Su objetivo es simple: permitir la conversión de una instancia de clase en cadena de texto, de forma que se pueda convertir a texto el contenido de la instancia. Pero era conversión no siempre es tan sencilla. Hay clases fácilmente convertibles a texto, como por ejemplo la clase **Integer**. Sin embargo hay otras en las que el objeto no tiene una conversión trivial a texto, y el método **toString()** tendrá que ser implementado para que genere algún tipo de representación "textual" del contenido del objeto.

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

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

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



### Recomendación

#### Observación de ingeniería de software

No es necesario copiar un objeto **String** existente. Los objetos **String** son inmutables: su contenido de caracteres no puede modificarse una vez que son creados, ya que la clase **String** no proporciona métodos que permitan modificar el contenido de un objeto **String**.

#### Error común de programación


Intentar acceder a un carácter que se encuentra fuera de los límites de un objeto **String** (es decir, un índice menor que 0 o un índice mayor o igual a la longitud del objeto **String**), se produce una excepción **StringIndexOutOfBoundsException**.

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

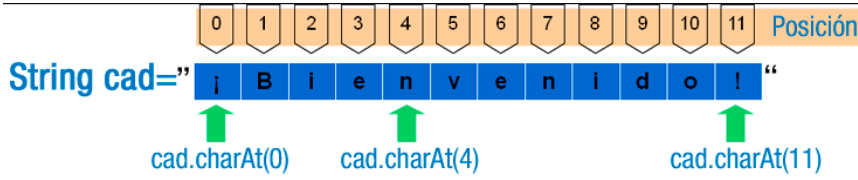
- **int length()**. Retorna un número entero que contiene la longitud de una cadena, resultado de contar el número de caracteres por la que esta compuesta. Recuerda que un espacio es también un carácter.

**String cad="¡ B i e n v e n i d o !"**



Longitud = 12

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

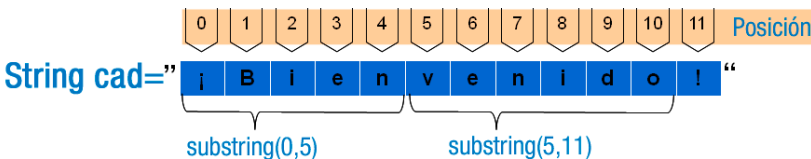


**String cad="¡ B i e n v e n i d o !"**

cad.charAt(0)      cad.charAt(4)      cad.charAt(11)

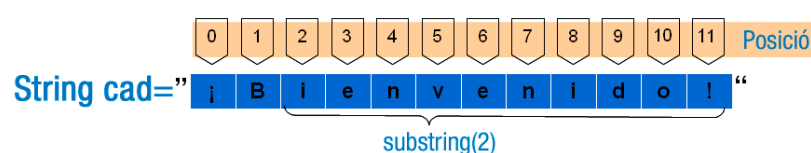
```
char t = cad.charAt(5);
System.out.println(t);
```

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



**String cad="¡ B i e n v e n i d o !"**

substring(0,5)      substring(5,11)



**String cad="¡ B i e n v e n i d o !"**

substring(2)

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

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

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

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

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

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

El resultado del código anterior es que se mostrará por pantalla "**Número cinco: 5**", y no dará ningún error. Esto es posible gracias a que Java convierte el número 5 a su clase envoltorio<sup>1</sup> (wrapper class) correspondiente (**Integer**, **Float**, **Double**, etc.), y después ejecuta automáticamente el método `toString()` de dicha clase.



### Reflexiona

¿Cuál crees que será el resultado de poner `System.out.println("A"+5f)`?

Pruébalo y recuerda: no olvides indicar el tipo de literal (f para los literales de tipo `float`, y d para los literales de tipo `double`), así obtendrás el resultado esperado y no algo diferente.



### Debes conocer

Un error común de programación se da al comparar referencias con `==`

Se pueden producir errores lógicos ya que `==` compara las referencias para determinar si se refieren al mismo objeto, no a si dos objetos tienen el mismo contenido.

Si se comparan dos objetos idénticos con `==` pero se trata distintos objetos en memoria, es decir, diferentes referencias, el resultado será `false`. Si va a comparar objetos **para determinar si tienen el mismo contenido**, se debe utilizar el método `equals`.

<sup>1</sup> Las clases envoltorio son clases que encapsulan tipos de datos básicos, tales como `int`, `short`, `double`, etc. Sirven básicamente para poder usar los tipos básicos como si fueran objetos. Las clases envoltorio son generalmente inmutables.

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

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

```
String c="1234.5678";

double n;

try {

    n=Double.valueOf(c).doubleValue();

} catch (NumberFormatException e){ /* Código a ejecutar si no se puede convertir */}
```

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

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

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

Si has probado el código anterior, habrás comprobado que el resultado no muestra "**3,30 €**" sino que muestra "**3,3 €**". ¿Cómo lo solucionamos? Podríamos dedicar bastantes líneas de código hasta conseguir algo que realmente funcione, pero no es necesario, dado que Java y otros lenguajes de programación (como C), disponen de lo que se denomina formateado de cadenas<sup>2</sup>. En Java podemos "formatear" cadenas a través del método estático **format** disponible en el objeto **String**. Este método permite crear una cadena proyectando los argumentos en un formato específico de salida. Lo mejor es verlo con un ejemplo. Observemos cuál sería la solución al problema planteado antes:

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

---

<sup>2</sup> Operación consistente en especificar cómo queremos que se transforme un dato, generalmente numérico, en cadena. Dicha conversión permite especificar de forma precisa cosas como el número de decimales que queremos que tenga el resultado



## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

El formato de salida, también denominado "cadena de formato", es el primer argumento del método **format**. La variable **precio**, situada como segundo argumento, es la variable que se proyectará en la salida siguiendo un formato concreto. Seguro que te preguntarás, ¿qué es **"%.2f"**? Pues es un especificador de formato<sup>3</sup> e indica cómo se deben formatear o proyectar los argumentos que hay después de la cadena de formato en el método **format**.



### Debes conocer

Es necesario que conozcas bien el método **format** y los especificadores de formato. Por ese motivo, te pedimos que leas atentamente el *Anexo I: Formateado de cadenas en Java*, de la unidad.

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

### Métodos importantes de la clase String.

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

<sup>3</sup> Es una subcadena dentro de una cadena que especifica cómo se debe formatear un dato concreto que se pasa como argumento al método **format**. El especificador de formato está formado por un carácter de escape **"%"** seguido de varios símbolos, destinados a describir cómo será el formato de salida.

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

<code>cad1.contains(cad2)</code>	Retornará <code>true</code> si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará <code>false</code> .
<code>cad1.startsWith(cad2)</code>	Retornará <code>true</code> si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .
<code>cad1.endsWith(cad2)</code>	Retornará <code>true</code> si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .
<code>cad1.replace(cad2,cad3)</code>	Generará una copia de la cadena <code>cad1</code> , en la que se sustituirán todas las apariciones de <code>cad2</code> por <code>cad3</code> . El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "zzz" por "xx" en la cadena "zzzzz" generará "xxzzz" y no "zzxxz".

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

En Java, **String** es un objeto inmutable, lo cual significa, entre otras cosas, que cada vez que creamos un **String**, o un literal de **String**, se crea un nuevo objeto que no es modificable. Java proporciona la clase **StringBuilder**, la cual es un mutable<sup>4</sup>, y permite una mayor optimización de la memoria. También existe la clase **StringBuffer**, pero consume mayores recursos al estar pensada para aplicaciones multi-hilo<sup>5</sup>, por lo que en nuestro caso nos centraremos en la primera.

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

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

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

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

1. **strb.delete(6,8);** Eliminamos las 'uu' que sobran en la cadena. La primera 'u' que sobra está en la posición 6 (no olvides contar el espacio), y la última 'u' a eliminar está en la posición 7. Para eliminar dichos caracteres de forma correcta hay que pasar como primer argumento la posición 6 (posición inicial) y como segundo argumento la posición 8 (posición contigua al último carácter a eliminar), dado que la posición final no indica el último carácter a eliminar, sino el carácter justo posterior al último que hay que eliminar (igual que ocurría con el método **substring()**).
2. **strb.append("!");** Añadimos al final de la cadena el símbolo de cierre de exclamación.
3. **strb.insert (0,"i");** Insertamos en la posición 0, el símbolo de apertura de exclamación.

<sup>4</sup> Es lo contrario a un objeto inmutable. Son objetos que una vez creados se pueden modificar sin problemas.

<sup>5</sup> Se trata de aplicaciones que ejecutan varias líneas de ejecución simultáneamente y que necesitan acceder a datos compartidos. Cuando varias líneas de ejecución, cada una a su ritmo, necesitan acceder a datos compartidos, hay que tener mucho cuidado para que los datos que manejan no sean incoherentes.

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

4. **strb.replace (3,5,"la");** Reemplazamos los caracteres 'al' situados entre la posición inicial 3 y la posición final 4, por la cadena 'la'. En este método ocurre igual que en los métodos **delete()** y **substring()**, en vez de indicar como posición final la posición 4, se debe indicar justo la posición contigua, es decir 5.

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

### Debes conocer

En la siguiente página puedes encontrar más información (en inglés) sobre como utilizar la clase **StringBuilder**.  
[Uso de la clase StringBuilder.](#)



## Recomendación

### Observaciones de rendimiento

Java puede realizar ciertas optimizaciones en las que se involucran objetos **String** (como hacer referencia a un objeto **String** desde múltiples variables), ya que sabe que estos objetos no cambiarán. Si los datos no van a cambiar, debemos usar objetos **String** (y no **StringBuilder**).

En los programas que realizan la concatenación de cadenas con frecuencia, o en otras modificaciones de cadenas, a menudo es más eficiente implementar las modificaciones con la clase **StringBuilder**.

El proceso de incrementar en forma dinámica la capacidad de un objeto **StringBuilder** puede requerir una cantidad considerable de tiempo. La ejecución de un gran número de estas operaciones puede degradar el rendimiento de una aplicación. Si un objeto **StringBuilder** va a aumentar su tamaño en forma considerable, tal vez varias veces, al establecer su capacidad a un nivel alto en un principio se incrementará el rendimiento.

### Error común de programación

Al tratar de acceder a un carácter que se encuentra fuera de los límites de un objeto **StringBuilder** (es decir, con un índice menor que 0, o mayor o igual que la longitud del objeto **StringBuilder**) se produce una excepción **StringIndexOutOfBoundsException**.

## 2.2 EXPRESIONES REGULARES

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

Pues esta es la función de las expresiones regulares: **permitir comprobar si una cadena sigue o no un patrón preestablecido**. Las expresiones regulares son un mecanismo para describir esos patrones, y se construyen de una forma relativamente sencilla. Existen muchas librerías diferentes para trabajar con expresiones regulares,

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

y casi todas siguen, más o menos, una sintaxis similar, con ligeras variaciones. Dicha sintaxis nos permite indicar el patrón de forma cómoda, como si de una cadena de texto se tratase, en la que determinados símbolos tienen un significado especial. Por ejemplo "[01]+" es una expresión regular que permite comprobar si una cadena conforma un número binario. Veamos cuáles son las reglas generales para construir una expresión regular:

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

Con las reglas anteriores podemos indicar el conjunto de símbolos que admite el patrón y el orden que deben tener. Si una cadena no contiene los símbolos especificados en el patrón, en el mismo orden, entonces la cadena no encajará con el patrón<sup>6</sup>. Veamos ahora cómo indicar repeticiones. Se llevan a cabo mediante los **operadores de cuantificación** (o **cuantificadores**). Los más habituales son la **interrogación** (símbolo "?"), la **estrella** (símbolo "\*"), el **más** (símbolo "+") y las **llaves** (símbolos "{" y "}"):

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

---

<sup>6</sup> Modelo con el que encaja la información que estamos analizando o que simplemente se ha utilizado para construir dicha información. Un patrón, en el caso de la informática, está constituido por una serie de reglas fijas que deben cumplirse o ser seguidas para formar la información. Las direcciones de correo electrónico, por ejemplo, todas tienen la misma forma, y eso es porque todas siguen el mismo patrón para ser construidas.

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

- "[a-z]{1,4}[0-9]+". Los indicadores de repetición se pueden usar también con corchetes, dado que los corchetes representan, básicamente, un símbolo. En el ejemplo anterior se permitiría de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.

Otro operador importante en las expresiones regulares es la barra (símbolo "|") para indicar una entre varias opciones. Por ejemplo, la expresión regular "o|u" encontrará cualquier "o" o "u" dentro del texto y la expresión regular "este|oeste|norte|sur" puede servir para encontrar el nombre de cualquiera de los puntos cardinales (en minúscula).



### Ejercicio resuelto

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

Mostrar retroalimentación

Una expresión regular que permite verificar si una cadena contiene un DNI o un NIE es la siguiente:

```
"([0-9]{8}[a-zA-Z])|([XxYyZz][0-9]{7}[a-zA-Z])";
```

¿Y cómo uso las expresiones regulares en un programa?

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

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

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

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

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

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



### Recomendación

#### Error común de programación

Una expresión regular puede compararse con un objeto de cualquier clase que implemente a la interfaz `CharSequence`, pero la expresión regular debe ser un objeto `String`. Si se intenta crear una expresión regular como un objeto `StringBuilder` se produce un error.

El método `matches` (de las clases `String`, `Pattern` o `Matcher`) devuelve `true` sólo si todo el objeto de búsqueda concuerda con la expresión regular. Los métodos `find` y `lookingAt` (de la clase `Matcher`) devuelven `true` si una parte del objeto de búsqueda concuerda con la expresión regular.

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

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

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

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

partes de una  
coincide con una  
Lo mejor es verlo  
(seguro que te

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

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

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

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

### Para saber más

Con lo estudiado hasta ahora, ya puedes utilizar las expresiones regulares y sacarles partido casi que al máximo. Pero las expresiones regulares son un campo muy amplio, por lo que es muy aconsejable que amplíes tus conocimientos con el siguiente enlace: [Tutorial de expresiones regulares en Java \(en inglés\)](#).

## 3. CREACIÓN DE ARRAYS.

---

¿Dónde almacenarías tú una lista de nombres de personas? Una de las soluciones es usar arrays, puede que sea justo lo que necesita Ana, pero puede que no. Todos los lenguajes de programación permiten el uso de arrays, veamos como son en Java.

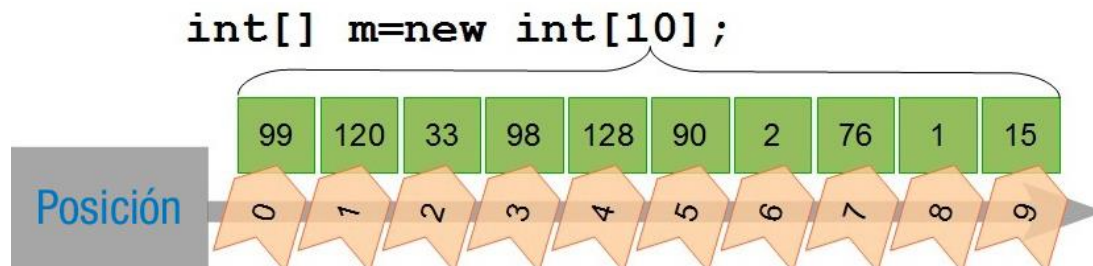
**Los arrays permiten almacenar una colección de objetos o datos del mismo tipo.** Son muy útiles y su utilización es muy simple:



#### TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

Veamos un ejemplo de su uso:



Veamos un ejemplo de su uso:

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

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





## Recomendación

### Error común de programación

Un índice debe ser un valor `int`, o un valor de un tipo que pueda promoverse a `int`, es decir, los tipos `byte`, `short` o `char`, pero no `long`. De lo contrario, se producirá un error de compilación (error semántico por "pérdida de precisión" al promover de `long` a `int`).

Al contrario que en otros lenguajes de programación, **en la declaración de un array no se debe indicar el número de elementos**. Si se especifica el número de elementos en los corchetes de la declaración (por ejemplo `int[12] c`), se produce también un error de compilación (en este caso de sintaxis).

Debes ser cuidadoso con las **declaraciones múltiples de variables de tipo array**, pues pueden llevar a confusión. Por ejemplo, en la declaración:

```
int[] a, b, c;
```

Si `a`, `b` y `c` deben ser variables de tipo array de enteros, entonces esa declaración es correcta. Al colocar corchetes directamente después del tipo, indicamos que todos los identificadores en la declaración son variables tipo array. Ahora bien, si solamente `a` debe ser una variable tipo array, mientras que `b` y `c` deben ser variables `int` individuales (escalares), entonces la declaración no sería apropiada. En tal caso la declaración

```
int a[], b, c;
```

lograría el resultado deseado.

### Buena práctica de programación

Por cuestión de legibilidad, suele ser apropiado indicar solo una variable de tipo array en cada declaración. Y si se añade un comentario que describa a la variable que se está declarando, mucho mejor.

### Indicaciones sobre el rendimiento

En muchos lenguajes de programación los arrays se pasan por referencia en los métodos por cuestiones de rendimiento. Si los arrays se pasaran por valor, se pasaría una copia de cada elemento cada vez que se invocara al método, lo cual podría ser terriblemente ineficiente (imagina un array de cientos o miles de elementos). En el caso de Java no hay problema, siempre se hace por referencia, pues un array es ya una referencia por sí mismo.

## 3.1 USO DE ARRAYS UNIDIMENSIONALES

Ya sabes declarar y crear de arrays, pero, ¿cómo y cuándo se usan?

Las operaciones habituales que se suelen realizar sobre ellos son:

- modificación del elemento ubicado en una posición del array;
- acceso al elemento ubicado en una posición del array.

La **modificación de un elemento en una posición del array** se realiza mediante una simple asignación. Simplemente se especifica entre corchetes la posición a modificar después del nombre del array (operador "corchete"). Veámoslo con un simple ejemplo:

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

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

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

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

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



### Recomendación

#### Buena práctica de programación

Dependiendo del lenguaje de programación, a las **variables constantes** también se les conoce como **constantes con nombre**. Este tipo de variables mejoran la **legibilidad** del código respecto al uso de valores literales. Por ejemplo, una constante con un nombre como `LONGITUD_VECTOR` indica sin duda su propósito, mientras que un valor literal (por ejemplo, 10) podría tener distintos significados, dependiendo de su contexto.

#### Error común de programación

Tratar de **asignar un valor a una variable constante después de inicializarla** produce un error de compilación.

Intentar **usar una constante antes de inicializarla** también da lugar a un error de compilación.

#### Observación para prevenir errores

Al acceder a un elemento de un array, hay que asegurarse de que el índice del array siempre sea mayor o igual a 0 y menor que su longitud. Esto ayudará a evitar excepciones del tipo `ArrayIndexOutOfBoundsException`.

## 3.2 INICIACIÓN

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

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

```
int totalNumeros = 10 ;
int[] consecutivos = new int[totalNumeros];
for (int i=0; i < totalNumeros;i++)
    consecutivos[i] = i ;
```

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

```
int[] array = {10, 20, 30};

String[] diasemana = {"Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"};
```

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

Cuando se trata de un array de objetos, la inicialización del mismo es un poco más liosa, dado que el valor inicial de los elementos del array de objetos será **null**<sup>7</sup>, o lo que es lo mismo, crear un array de objetos no significa que se han creado las instancias de los objetos. Hay que crear, para cada posición del array, el objeto del tipo correspondiente con el operador **new**. Veamos un ejemplo con la clase **StringBuilder**. En el siguiente ejemplo solo aparecerá **null** por pantalla:

```
StringBuilder[] oStrBuilds = new StringBuilderStringBuilder[10];

for (int i = 0; i < oStrBuilds.length; i++)
    System.out.println("Valor" + i + "=" + oStrBuilds[i]); // Imprimirá null para los 10 valores.
```

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

```
StringBuilder[] oStrBuilds = new StringBuilder[10];

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



### Reflexiona

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

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

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

### Ejercicio Resuelto

Ya sabemos lo que son los arrays, y cómo inicializarlos. Supongamos un array, por ejemplo de enteros, ¿se te ocurre cómo haríamos para ordenarlo?

---

<sup>7</sup> Una variable que almacena un objeto en realidad es un apuntador a la zona de memoria donde se almacena dicho objeto. Cuando dicha variable, no apunta a ningún objeto, porque el objeto no se ha creado con el operador **new**, entonces la variable apunta a **null**, para indicar que no hace referencia todavía a ningún objeto.,

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

La idea puede ser ir recorriendo el array, y para cada elemento del mismo, hacemos una segunda pasada, un segundo recorrido del array desde el siguiente elemento más uno en adelante y comparo. Así, si el elemento del array que estoy recorriendo en primer lugar es mayor que el de la segunda pasada, entonces intercambio esos elementos. Intenta resolverlo antes de mirar la solución.

### [Ordenar array ascendentemente.](#)

Ahora supón que tenemos un array de números enteros, ordenado, pero que puede contener huecos (null) donde insertar un número. Supongamos que los nulos, los huecos están al final. Queremos poder insertar un número al array, y que siga estando ordenado ascendentemente. ¿Cómo lo harías? Una idea puede ser: primero comprobar si hay hueco, y si no lo hay aviso y termino. Si lo hay, empiezo a recorrer el array y si es nulo, lo inserto sin más, si no, comparo si el elemento que quiero insertar es menor que el de la posición actual del array. Cuando detecte un número mayor que el que yo quiero insertar, tendré que desplazar los elementos del array desde esa posición en adelante para abrir hueco. ¿Eres capaz de hacerlo? ¡Adelante! Inténtalo antes de ver la solución más abajo.

### [Insertar elemento en array ordenado](#)

Por último, veamos cómo borrar un elemento de un array manteniendo el array ordenado y empaquetado (sin huecos null en medio, esto quiere decir que todos los espacios vacíos que tenga el vector van a estar siempre juntos en las últimas posiciones del vector). ¿Cómo lo harías? La idea es desplazar todos los elementos de las posiciones siguientes al elemento que queremos borrar una posición hacia abajo, para mantener el vector empaquetado, sin abrir huecos en medio. Una vez hecho esto, debemos asegurarnos de igualar a null la última persona del vector, para que no aparezca duplicada, y para que el vector tenga un hueco más de los que tenía Como siempre, antes de mirar la solución, intenta hacerlo.

### [Borrar elemento en array ordenado](#)

## 3.3 MOSTRAR CONTENIDO DE UN ARRAY.

Es posible que en nuestros programas necesitemos mostrar por pantalla el contenido completo de todos los elementos contenidos en un array. Para ello bastaría con realizar un **bucle que fuera desde el primer índice hasta el último** y fuera mostrando el contenido de cada elemento correspondiente a esa posición o índice:

```
// Mostrar el contenido del array escribiendo cada elemento en una línea diferente
for (int i = 0; i < arrayElementos.length; i++) {
    System.out.printf("El contenido de la posición: %d es %d\n", i, arrayElementos[i]);
}
```

Este ejemplo mostraría cada elemento en una línea diferente indicando su posición. Si quisiéramos, por ejemplo, mostrar todos los elementos en una misma línea, separándolos por un espacio en blanco podríamos hacer:

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

```
// Mostrar el contenido del array escribiendo cada elemento separado por un espacio
System.out.printf("El contenido del array es: ");
for (int i = 0; i < arrayElementos.length; i++) {
    System.out.printf("%d ", i, arrayElementos[i]);
}
```

Y así en general para cualquier manera que se te ocurra a la hora de presentar esa información.

También puedes, en lugar de escribir toda esa información en pantalla, generar una única cadena con todos los datos para más adelante mostrarla en pantalla con una sola sentencia o bien para que la utilice otra parte de tu programa (o incluso otro programa). En tal caso, en lugar de ir mostrando cada fragmento de información por pantalla irías concatenando todos esos fragmentos en una única cadena. Los ejemplos anteriores podrían quedar entonces así:

```
// Volcar el contenido del array en un objeto String
String contenidoArray= "";
for (int i = 0; i < arrayElementos.length; i++) {
    contenidoArray += String.format ("El contenido de la posición: %d es %d\n", i, arrayElementos[i]);
}
// Mostrar el contenido del array
System.out.printf("%s\n", contenidoArray);
```

```
// Volcar el contenido del array en un objeto String
String contenidoArray= "";
for (int i = 0; i < arrayElementos.length; i++) {
    contenidoArray += arrayElementos[i] + " ";
}
// Mostrar el contenido del array
System.out.printf("El contenido del array es: %s", contenidoArray);
```

Basándose en esa filosofía de encapsular los resultados, Java proporciona algunas herramientas para generar un **String** con una representación textual del contenido de un array. Se trata del método estático **toString** de la clase **Arrays**, que recibe como parámetro un array y devuelve un objeto **String** con una representación textual. El formato de la cadena que devuelve es del estilo:

```
[elemento_0, elemento_1, elemento_2, . . . , elemento_(n-1)]
```

donde n sería el tamaño del array.

En tal caso, el código para mostrar el contenido de un array se vería sensiblemente simplificado:

```
System.out.printf("El contenido del array es: %s\n", Arrays.toString(arrayElementos));
```

La clase **Arrays** se encuentra en el paquete **java.util**, así que recuerda que deberás hacer un **import java.util.Arrays** para poder utilizarla en tus programas.

Para saber más

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

Puedes consultar toda la información sobre el método estático **Arrays.toString** en la documentación de la API de Java:

[Método toString de la clase Arrays.](#)

Observarás que en realidad que el método está sobrecargado y que hay tantas versiones del método como tipos primitivos hay en Java.



### Ejercicio Resuelto

Escribe un programa en Java que solicite un número *n* y rellene un array de enteros de tamaño *n* con los *n* primeros números pares empezando desde 0 y terminando con  $2(n-1)$ . Una vez relleno el array, se debe mostrar su contenido por pantalla escribiendo cada elemento en una línea diferente.

Un posible resultado de la ejecución del programa podría ser:

```
ARRAY DE NÚMEROS PARES
-----
Introduzca la cantidad de elementos: 12

RESULTADO: LISTA DE PARES EN EL ARRAY
-----
Elemento 0: 0
Elemento 1: 2
Elemento 2: 4
Elemento 3: 6
Elemento 4: 8
Elemento 5: 10
Elemento 6: 12
Elemento 7: 14
Elemento 8: 16
Elemento 9: 18
Elemento 10: 20
Elemento 11: 22
```

Prueba ahora a mostrar todos los elementos en una única línea separándolos un espacio en blanco. En tal caso, el resultado debería ser algo así como:

```
0 2 4 6 8 10 12 14 16 18 20 22
```

En este caso, en lugar de escribir un avance de línea ("**\n**") después de cada elemento, podemos colocar un espacio en blanco ("") u otro separador:

```
// Mostramos el contenido del array elemento a elemento
System.out.printf("Contenido del array: ");
for (int i = 0; i < listaPares.length; i++) {
    System.out.printf("%d ", listaPares[i]);
}
```

¿Y si quisiéramos `Arrays.toString` para mostrar el contenido? ¿Cómo lo harías?

En ese caso sería incluso más sencillo, pues nos ahorramos tener que hacer un bucle ya que el método `toString` internamente lo hará por nosotros para generar un objeto `String` con la representación textual del contenido del array:

```
System.out.printf("\nContenido del array: %s ", Arrays.toString(listaPares));
```

Eso sí, tendremos que amoldarnos al formato que utiliza `Arrays.toString`:

```
Contenido del array: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22]
```

Si queremos personalizarlo de algún modo, tendremos que implementarlo nosotros "a mano".

## 4. ARRAYS MULTIDIMENSIONALES.

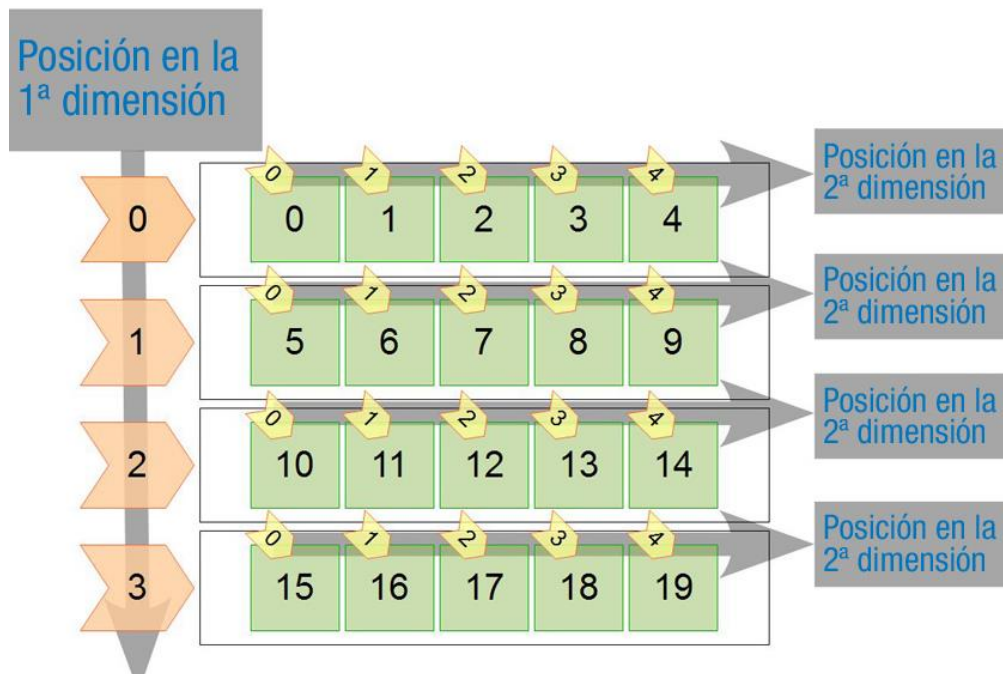
¿Qué estructura de datos utilizarías para almacenar los píxeles<sup>8</sup> de una imagen digital?

Normalmente las imágenes son cuadradas así que una de las estructuras más adecuadas es la matriz<sup>9</sup>. En la matriz cada valor podría ser el color de cada píxel. Pero, ¿qué es una matriz a nivel de programación? Pues es un array con dos dimensiones, o lo que es lo mismo, un array cuyos elementos son arrays de números.

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

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

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



Al igual que con los arrays de una sola dimensión, los arrays multidimensionales deben **declararse y crearse**. Podremos hacer arrays multidimensionales de todas las dimensiones que queramos y de cualquier tipo. En ellos todos los elementos del array serán del mismo tipo, como en el caso de los arrays de una sola dimensión.

La **declaración** comenzará especificando el tipo o la clase de los elementos que forman el array, después pondremos tantos corchetes como dimensiones tenga el array y por último el nombre del array, por ejemplo:

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

<sup>8</sup> Punto que compone una imagen digital

<sup>9</sup> Estructura matemática compuesta de datos numéricos dispuestos en filas y columnas, similar a una tabla.

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

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

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

### 4.1 USO DE LOS ARRAYS MULTIDIMENSIONALES

¿Y en que se diferencia el uso de un array multidimensional con respecto a uno de una única dimensión?

Pues en muy poco la verdad. Continuaremos con el ejemplo del apartado anterior:

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

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

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

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

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

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

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

```
static int sumaArray2d(int[][] a2d) {  
  
    int suma = 0;  
  
    for (int i1 = 0; i1 < a2d.length; i1++)  
  
        for (int i2 = 0; i2 < a2d[i1].length; i2++)  
            suma += a2d[i1][i2];  
  
    return suma;  
}
```



## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

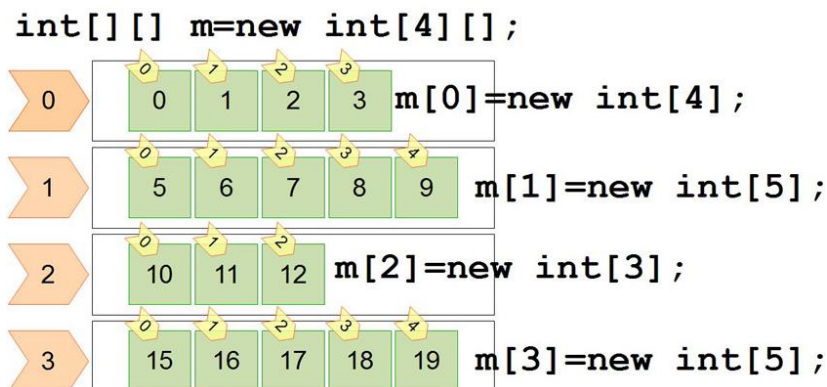
Del código anterior, fíjate especialmente en el uso del atributo **length** (que nos permite obtener el tamaño de un array). Aplicado directamente sobre el array nos permite saber el tamaño de la primera dimensión (**a2d.length**). Como los arrays multidimensionales son arrays que tienen como elementos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

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

Una matriz es un ejemplo de array multidimensional regular. ¿Por qué?

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

- **Declaramos y creamos el array pero sin especificar la segunda dimensión.** Lo que estamos haciendo en realidad es crear simplemente un array que contendrá arrays, sin decir como son de grandes los arrays de la siguiente dimensión: `int[][] irregular=new int[3][];`
- **Después creamos cada uno de los arrays unidimensionales** (del tamaño que queramos) y lo asignamos a la posición correspondiente del array anterior: `irregular[0]=new int[7]; irregular[1]=new int[15]; irregular[2]=new int[9];`



### Recomendación

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

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

### Para saber más

Puedes echar un vistazo a este vídeo donde se habla sobre arrays multidimensionales:

<https://www.youtube.com/watch?v=nfAR9QDy1IU>

## 4.2 INICIACIÓN DE ARRAYS MULTIDIMENSIONALES

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

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

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

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

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

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

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

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

### Para saber más

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

[Operaciones básicas con matrices.](#)

## 4.3 MOSTRAR CONTENIDO DE UN ARRAY MULTIDIMENSIONAL

Para mostrar por pantalla un array de más de una dimensión o simplemente para realizar una representación textual que queramos almacenar en una cadena, tendremos que construir una **estructura de bucles anidados** que vayan recorriendo cada dimensión, uno dentro de otro. Necesitaremos por tanto un bucle por cada dimensión.

Por ejemplo, para **recorrer un array de dos dimensiones (bidimensional) regular (todas las filas tienen las mismas columnas)**, podríamos hacer algo así:

```
int numFilas= tabla.length; // Tamaño de la primera dimensión (número de filas)
int numColumnas= tabla[0].length; // Tamaño de la segunda dimensión: número columnas en primera fila (igual para todas)
for (int fila=0; fila<numFilas; fila++) {
    for (int columna=0; columna<numColumnas; columna++) {
        // Mostramos en elemento colocado en la posición [fila][columna]
        // o bien vamos concatenando fragmentos en una cadena de resultado final
    }
}
```

Ahora bien, si los elementos de la segunda dimensión (columnas de cada fila) no tienen todos el mismo tamaño (cada fila puede tener un número diferente de columnas) entonces tendremos que recorrer en cada caso un número de columnas diferente y deberemos consultar ese tamaño para cada fila (**nombreArray[fila].length**):

```
int numFilas= tabla.length; // Tamaño de la primera dimensión ()
for (int fila=0; fila<numFilas; fila++) {
    int numColumnas= tabla[fila].length; // Tamaño de la segunda dimensión (potencialmente diferente para cada fila)
    for (int columna=0; columna<numColumnas; columna++) {
        // Mostramos en elemento colocado en la posición [fila][columna]
        // o bien vamos concatenando fragmentos en una cadena de resultado final
    }
}
```

Otra opción es no utilizar una variable para guardar los tamaños de cada dimensión y consultarlos con el atributo **length** cada vez que sea necesario, que suele ser lo más habitual:

```
for (int fila=0; fila<tabla.length; fila++) {
    for (int columna=0; columna<tabla[fila].length; columna++) {
        . . .
    }
}
```

#### TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

¿Y qué ocurre si decidimos usar la herramienta **Arrays.toString**? En este caso la representación textual del array que devuelve este método puede resultarnos algo confusa pues devolverá una cadena con un aspecto similar al siguiente:

```
[[I@28d93b30, [I@1b6d3586, . . . , [I@1540e19d]
```

¿Qué significado tiene esto? Ten en cuenta que cada elemento de este array es, a su vez, una referencia a un nuevo array (la siguiente dimensión). Sin embargo, para cada elemento no se nos muestra su contenido (otro array) sino el **hash** de esa referencia (una marca de identificación de la máquina virtual respecto a su almacenamiento). Si quisiéramos que esa referencia se "expandiera" y se mostrara también una representación textual de los elementos de cada subarray (segunda dimensión) podríamos usar **otro método estático** proporcionado por la clase **Arrays** llamado **deepToString**. Este método sí "profundizará" en todas las subdimensiones o subarrays y generará una representación textual de todas las subdimensiones, sean de los niveles que sean. Por ejemplo, para un array de dos dimensiones podría obtenerse un resultado de este estilo:

```
[[1, 2, 3], [4, 5, 6, 7], . . . , [50, 51, 52, 53]]
```

Podemos observar que se abre un nuevo corchete para cada "subdimensión" o "subarray" que exista. Es decir, que para llegar a un elemento final (escalar) habrá que pasar por tantos corchetes como dimensiones existan. **Para un array de tres dimensiones** habría por tanto que pasar a través de tres corchetes para poder llegar a los elementos finales almacenados. Por ejemplo:

```
[[[1, 2], [3]], [[4], [5], [6, 7]], . . . , [[50, 51], [52, 53]]]
```

Aquí se puede observar que en la posición [0][0][0] hay almacenado un 1, en la [0][0][1] un 2, en la [0][1][0] un 3, en la [1][0][0] un 4, en la [1][1][0] un 5, en la [1][2][0] un 6, y así sucesivamente tal y como se ha visto en el apartado anterior al explicar el funcionamiento de los arrays multidimensionales.

Ahora, podemos ver a continuación un ejemplo donde se rellena primero y se muestra después el contenido de un array de dos dimensiones. En cada celda de este array se almacenará el valor de la suma de las posiciones de cada dimensión. Es decir, en la posición [0][0] se almacenará un 0, en la [0][1] un 1, en la [0][2] un 2, en la [1][0] un 1, en la [1][1] un 2, en la [0][3] un 3, y así sucesivamente.

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

```
final int FILAS = 5 ;
final int COLUMNAS = 3 ;

// Declaramos el array y reservamos espacio para sus dos dimensiones
int[][] arrayBidimensional = new int[FILAS][COLUMNAS];

// Rellenamos el array
for (int i=0; i< arrayBidimensional.length; i++)
    for (int j=0; j< arrayBidimensional[i].length; j++)
        arrayBidimensional[i][j] = i + j ;

// Escribimos el contenido de cada celda del array usando dos bucles anidados
for (int i=0; i< arrayBidimensional.length; i++) {
    for (int j=0; j< arrayBidimensional[i].length; j++) {
        System.out.print ("[" + i + "][" + j + "] = " + arrayBidimensional[i][j] + "    ");
    }
    System.out.println() ;
}
```

### Para saber más

Puedes consultar toda la información sobre el método estático **Arrays.deepToString** en la documentación de la API de Java:

[Método deepToString de la clase Arrays.](#)



## Ejercicio Resuelto

Implementa un programa en Java que:

1. solicite al usuario el **número de filas** para un array de números enteros;
2. solicite al usuario, una por una, el **número de columnas que se desea que tenga cada fila** (cada fila no tiene por qué tener el mismo número de columnas);
3. según se vaya conociendo el rango de cada dimensión se vaya **reservando espacio** para esa dimensión o subdimensión;
4. una vez creada la estructura completa, **rellenarla con números enteros comenzando por 1** e incrementando de uno en uno;
5. una vez relleno el array completo, recorrerlo para **mostrar por pantalla todo su contenido** escribiendo el contenido de cada fila en una línea diferente;
6. mostrar también el **contenido del array** usando los **métodos estáticos** de la clase **Arrays**:
  1. **Arrays.toString**;
  2. **Arrays.deepToString**.

Aquí tienes un posible ejemplo de ejecución:

```
Introduzca el número de filas en el array: 5
Introduzca el número de columnas en la fila 0: 2
Introduzca el número de columnas en la fila 1: 3
Introduzca el número de columnas en la fila 2: 6
Introduzca el número de columnas en la fila 3: 1
Introduzca el número de columnas en la fila 4: 4

El contenido del array es:
1 2
3 4 5
6 7 8 9 10 11
12
13 14 15 16
Contenido usando Arrays.toString: [[I@28d93b30, [I@1b6d3586, [I@4554617c, [I@74a14482, [I@1540e19d]
Contenido usando Arrays.deepToString: [[1, 2], [3, 4, 5], [6, 7, 8, 9, 10, 11], [12], [13, 14, 15, 16]]
```

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

Para llevar a cabo la primera parte lo primero que habrá que hacer **será reservar espacio para la primera dimensión del array** (el número de **filas**) una vez que el usuario lo haya indicado, por ejemplo, a través de la variable `numFilas`:

```
int[][] array2D = new int[numFilas][];
```

Si te fijas, hemos declarado el array como de dos dimensiones (dos pares de corchetes `[][]`) y si embargo al hacer el `new` tan solo hemos colocado una cantidad en la primera dimensión (número de filas), pues es la única que conocemos. La variable `array2D` es una variable de tipo referencia que apunta a una zona de memoria donde estará el array (una sucesión de elementos dispuestos uno tras otro en la memoria). Cada uno de esos elementos será a su vez una nueva referencia que apuntará a otro array (en este caso un array de `int`).

**Una vez que conocemos el número de filas**, podremos ir preguntando al usuario, una por una, cuántas columnas quiere que tenga cada fila y aprovechar para ir reservando espacio para cada una de ellas:

```
// Crear la estructura de columnas para cada fila
for (int fila = 0; fila < numFilas; fila++) {
    System.out.printf("Introduzca el número de columnas en la fila %d: ", fila);
    numColumnas = teclado.nextInt();
    array2D[fila] = new int[numColumnas]; // Dimensionar las columnas de esta fila
}
```

Ten en cuenta aquí que **cada elemento** `array2D[fila]` es **una referencia que apuntará a otro array** de una cantidad de columnas determinada cuyo espacio en memoria se reservará con un nuevo `new`. La referencia que devuelva el operador `new` es lo que se almacenará en la posición `fila` del array de nombre `array2D` (`array2D[fila]`).

Una vez que ya disponemos de la estructura completa con todos los elementos reservados en memoria, podemos comenzar a recorrerla y **rellenarla de información mediante un doble bucle anidado**:

```
// Rellenar las celdas
int numero = 1;
for (int fila = 0; fila < numFilas; fila++) {
    for (int columna = 0; columna < numColumnas; columna++) {
        array2D[fila][columna] = numero;
        numero++;
    }
}
```

Una vez que la tengamos rellena, podemos proceder a **mostrar su contenido con otro doble bucle anidado** similar al anterior:

```
// salida de resultados
System.out.println("\nEl contenido del array es:");
for (int fila = 0; fila < numFilas; fila++) {
    numColumnas = array2D[fila].length; // Columnas de esta fila
    for (int columna = 0; columna < numColumnas; columna++) {
        System.out.print(array2D[fila][columna] + " ");
    }
    System.out.println(); // Una nueva línea para mostrar cada fila
}
```

Y por último **mostramos el contenido del array** usando los **métodos estáticos** que proporciona la clase no instanciable `Arrays` (métodos `Arrays.toString` y `Arrays.deepToString`):

```
System.out.printf("Contenido usando Arrays.toString: %s\n", Arrays.toString(array2D));
System.out.printf("Contenido usando Arrays.deepToString: %s\n", Arrays.deepToString(array2D));
```

Fíjate que si utilizamos simplemente `toString` nos quedamos únicamente en la primera dimensión y no se "profundiza" en el resto de dimensiones, por eso vemos simplemente las referencias a los otros arrays, pero no una representación textual de esos arrays. Sin embargo, si utilizamos `deepToString` sí obtenemos una representación textual del contenido de cada subarray.



## Ejercicio Resuelto

Implementa un programa en Java que:

1. solicite al usuario el **número de filas base** para un array de caracteres (entre 3 y 10);
2. se genere un array bidimensional de caracteres cuyo número de filas sea el doble de ese número de filas "base";
3. la primera fila contendrá 1 un solo elemento (una columna), la segunda 3 elementos (dos más que la anterior), la tercera 5 (dos más que la anterior) y así sucesivamente hasta el número de filas "base", a partir de esa fila, se repetirá otra fila con la misma cantidad de elementos y continuación se irán teniendo filas con dos elementos menos cada vez, hasta la última fila que contendrá un único elemento;
4. se rellenarán todas las posiciones de la misma fila se rellenarán con el mismo carácter: la primera fila con la letra 'A', la segunda con la 'B', la tercera con la 'C' y así sucesivamente;
5. se mostrará el contenido del array usando `Arrays.deepToString`;
6. se mostrará el contenido del array en forma de "tabla" usando un doble bucle que recorra primero filas y dentro de cada fila, cada una de las celdas (columnas).

Un ejemplo de ejecución podría ser algo así:

```
ARRAY 2D EN FORMA DE ROMBO
-----
Introduzca el número de filas "base" (3-10): 3

RESULTADO: ARRAY 2D EN FORMA DE ROMBO
-----

Contenido del array usando Arrays.deepToString: [[A], [B, B, B], [C, C, C, C, C], [D, D, D, D, D], [E, E, E], [F]]

Contenido del array mostrado elemento a elemento (doble bucle):
A
B B B
C C C C
D D D D
E E E
F
```

Nuestro array será multidimensional, en este caso concreto bidimensional (dos pares de corchetes) cuyos elementos serán de tipo `char`:

```
char arrayRombo[][];
```

La parte de entrada de datos consiste simplemente en incluir una comprobación de rango junto con la captura de una excepción `InputMismatchException` por si se introdujera algún carácter no compatible con la representación textual de un número entero:

```
numFilasBase = 0; // Valor inicial inválido
do {
    System.out.print("Introduzca el número de filas \"base\" (3-10): ");
    try {
        numFilasBase = teclado.nextInt();
    } catch (InputMismatchException ex) {
        teclado.nextLine(); // "Purgamos" lo que haya en el buffer
    }
} while (numFilasBase < 3 || numFilasBase > 10);
```

De esa manera, mientras no se introduzca un valor válido, se nos pedirá la entrada una y otra vez hasta que lo hagamos:

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

### ARRAY 2D EN FORMA DE ROMBO

```
-----
Introduzca el número de filas "base" (3-10): x
Introduzca el número de filas "base" (3-10): 0ç
Introduzca el número de filas "base" (3-10): 11
Introduzca el número de filas "base" (3-10):
```

Una vez que disponemos del número de filas "base", ya podemos reservar espacio para las filas del array. El número de filas será el doble de la cantidad "base" que se nos introduce:

```
// Reservamos espacio para las filas del array
arrayRombo = new char[numFilasBase * 2][];
```

Si te fijas, la segunda dimensión (segundo par de corchetes) permanece en blanco porque aún no sabemos cuántos elementos va a tener cada fila. No van a ser la misma cantidad (no va a ser una tablar "rectangular").

Una vez que dispongamos de las filas, tendremos que recorrer cada una de ellas y reservar espacio para el conjunto de elementos char de cada una (reservar espacio para los elementos de cada fila). Dado que va a tener una forma de "rombo" (las primeras filas con pocos elementos y va creciendo hasta la mitad para luego ir decreciendo), podemos dividir la reserva de memoria en dos partes:

✔ la **creciente**:

```
// Reservamos espacio para cada una de las filas del array hasta numFilasBase
// (creciendo)
for (int i = 0; i < numFilasBase; i++) {
    arrayRombo[i] = new char[1 + 2 * i];
}
```

✔ la **decreciente**:

```
// Reservamos espacio para cada una de las filas del array desde numFilasBase
// (decreciendo)
for (int i = 0; i < numFilasBase; i++) {
    arrayRombo[numFilasBase + i] = new char[1 + 2 * (numFilasBase - i - 1)];
}
```

Una vez que contemos con la estructura del array completada, podemos rellenarla con los caracteres que se nos ha pedido (letras consecutivas por filas a partir de la 'A'):

```
// Rellenamos todas las posiciones del array con letras consecutivas por filas
for (int i = 0; i < arrayRombo.length; i++) {
    for (int j = 0; j < arrayRombo[i].length; j++) {
        arrayRombo[i][j] = (char) ('A' + i);
    }
}
```

Y ya solo nos quedaría mostrar por pantalla el contenido del array usando `Arrays.deepToString`:

```
// Mostramos el contenido del array con Arrays.deepToString
System.out.printf("\nContenido del array usando Arrays.deepToString: %s \n", Arrays.deepToString(arrayRombo));
```

O bien elemento a elemento mediante dos bucles anidados:



## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

```
// Mostramos el contenido del array elemento a elemento con doble bucle
System.out.println("\nContenido del array mostrado elemento a elemento (doble bucle): ");
// Recorrido de filas
for (int i = 0; i < arrayRombo.length; i++) {
    // Contenido de la fila dibujado en la misma línea
    // Recorrido de columnas
    for (int j = 0; j < arrayRombo[i].length; j++) {
        System.out.printf("%c ", arrayRombo[i][j]);
    }
    System.out.printf("\n");
}
```

Una vez que hemos conseguido tener adecuadamente relleno el array y mostrar su contenido por pantalla, ¿crees que serías capaz de mostrarlo de una manera más estética dando la apariencia de "rombo" o "diamante"? La salida por pantalla podría tener entonces un aspecto similar al siguiente:

```
A
B B B
C C C C C
D D D D D
E E E
F
```

¿Te ves capaz de intentarlo?

Mostrar retroalimentación

Una forma posible de generar esa salida podría consistir en **ir escribiendo espacios antes de cada fila**, teniendo en cuenta que **en las primeras filas habrá que escribir más espacios y en la última fila central (fila "base") no hará falta escribir ningún espacio** y a partir de ahí en sentido inverso: sin espacios al principio y creciendo el número de espacios antes de cada fila según se vaya avanzando. Si tenemos en cuenta la cantidad de elementos de cada fila que estamos pintando puede hacerse de una manera bastante sencilla.

Aquí tenéis una propuesta para intentar darle esa forma geométrica a la salida siguiendo el planteamiento anterior:

```
// Mostramos el contenido del array elemento a elemento con doble bucle y dibujando un rombo
System.out.println(
    "Contenido del array elemento a elemento (doble bucle) con inquietudes \"estéticas\": ");
for (int i = 0; i < arrayRombo.length; i++) {
    // Espacios en blanco para dar forma de rombo
    for (int j = 0; j < (arrayRombo.length - arrayRombo[i].length); j++) {
        System.out.print(" ");
    }
    // Contenido de la fila dibujado en la misma línea
    // Recorrido de columnas
    for (int j = 0; j < arrayRombo[i].length; j++) {
        System.out.printf("%c ", arrayRombo[i][j]);
    }
    System.out.printf("\n");
}
```

## ANEXO I

Sintaxis de las cadenas de formato y uso del método `format()`.

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

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

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

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

Tipo de conversión	Especificación de formato	Tipos de datos aplicables	Ejemplo	Resultado del ejemplo
Valor lógico o booleano.	<code>"%b"</code> o <code>"%B"</code>	Boolean (cuando se usan otros tipos de datos siempre lo formateará escribiendo <code>true</code> ).	<pre>boolean b=true; String d= String.format("Resultado: %b", b); System.out.println (d);</pre>	true
Cadena de caracteres.	<code>"%s"</code> o <code>"%S"</code>	Cualquiera, se convertirá el objeto a cadena si es posible (invocando el método <code>toString()</code> ).	<pre>String cad="hola mundo"; String d= String.format("Resultado: %s", cad); System.out.println (d);</pre>	hola mundo
Entero decimal	<code>"%d"</code>	Un tipo de dato entero.	<pre>int i=10; String d= String.format("Resultado: %d", i); System.out.println (d);</pre>	10
Número en notación científica	<code>"%e"</code> o <code>"%E"</code>	Flotantes simples o dobles.	<pre>double i=10.5; String d= String.format("Resultado: %E", i); System.out.println (d);</pre>	1.050000E+01
Número decimal	<code>"%f"</code>	Flotantes simples o dobles.	<pre>float i = 10.5f; String d = String.format("Resultado: %f", i); System.out.println (d);</pre>	10.500000
Número en notación científica o decimal (lo más corto)	<code>"%g"</code> o <code>"%G"</code>	Flotantes simples o dobles. El número se mostrará como decimal o en notación científica dependiendo de lo que sea mas corto.	<pre>double i=10.5; String d= String.format("Resultado: %g", i); System.out.println (d);</pre>	10.5000

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

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

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

```
%[Ancho]Conversión
```

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

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

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

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

```
%[Ancho][.Precisión]Conversión
```

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

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

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

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

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

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

## TEMA 4: CADENAS DE CARACTERES Y ARRAYS.

```
int i=10;
int j=20;

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

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

### Para saber más

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

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

## ANEXO II: SABER MÁS ACERCA DE ARRAYS

---

[La clase Arrays del API de Java.](#)

### Introducción a las Colecciones y la clase ArrayList

[Concepto de clase genérica y parametrizada: La clase ArrayList del API de Java. Listas redimensionables.](#)

[Arrays dinámicos en Java \(clase ArrayList\): ArrayList en Java, con ejemplos.](#)