

Estructuras de almacenamiento en Java



Objetivos

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

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

En esta unidad, el alumnado se enfrenta a la utilización de estructuras de datos de capacidad estática para almacenar información. También se usarán algoritmos de ordenación y búsqueda en dichas estructuras.

1.- Introducción.

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

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

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

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

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

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

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



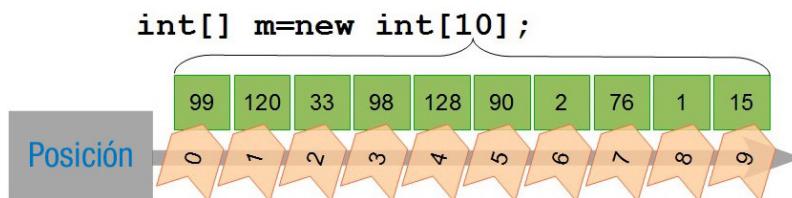
Autoevaluación

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

- Verdadero.
- Falso.

2.- Creación de arrays.

Todos los lenguajes de programación permiten el uso de arrays, veamos como son en Java.



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

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

Veamos un ejemplo de su uso:

```

int[] n;           // Declaración del array. Un array de enteros
n = new int[10];   // Creación del array reservando para él un espacio en memoria.

int[] m=new int[10]; // Declaración y creación en una línea.

```

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



Autoevaluación

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

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

2.1.- Uso de arrays unidimensionales.

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

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

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

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

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

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

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

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

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

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

```

    int suma=sumaArray (numeros);

```

Para **recorrer** un array podemos usar un bucle for o while. En las últimas versiones de Java se introdujo una nueva forma de uso del for, a la que se denomina “for extendido” o “**for each**”. Esta forma de uso del for, que ya existía en otros lenguajes, facilita el recorrido de objetos existentes en un array sin necesidad de definir el número de elementos a recorrer. Veamos un ejemplo:

```

public static void main(String[] args) {
    int [] numeros= new int[10];
    //bucle for para cargar el array
    for (int i=0; i<numeros.length; i++)
        numeros[i]=i;
    //bucle for each para recorrer el array
    for(int numero: numeros)
        System.out.println(numero);
}

```

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

```

public static void main(String[] args) {
    int j=0;
    int[] i=new int[1];
    i[0]=0;
    modificaArray(j,i);
    System.out.println(j+"-"+i[0]);
    /* Mostrará por pantalla "0-1", puesto que el contenido del array es
       modificado en el método, y aunque la variable j también se modifica, se modifica una copia
       de la misma, dejando el original intacto */
}

int modificaArray(int j, int[] i) {
    j++;
    i[0]++; /* Modificación de los valores de la variable, solo afectará al array, no a j */
}

```

2.2.- Inicialización.

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

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

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

En el ejemplo anterior se crea un array con una serie de números consecutivos, empezando por el cero, ¿sencillo no? Este uso suele ahorrar bastantes líneas de código en tareas repetitivas.

Otra forma de inicializar los arrays, cuando el número de elementos es fijo y sabido a priori, es indicando entre llaves el listado de valores que tiene el array. En el siguiente ejemplo puedes ver la inicialización de un array de tres números, y la inicialización de un array con tres cadenas de texto:

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

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

Cuando se trata de un array de objetos, la inicialización del mismo es un poco más liosa, dado que el valor inicial de los elementos del array de objetos será **null**, o lo que es lo mismo, crear un array de objetos no significa que se han creado las instancias de los objetos. Hay que crear, para cada posición del array, el objeto del tipo correspondiente con el operador **new**. Veamos un ejemplo con la clase **StringBuilder** (que estudiaremos en el apartado 5.5.). En el siguiente ejemplo solo aparecerá **null** por pantalla:

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

```
j[i]=new StringBuilder("cadena "+i);
```



Reflexiona

Para acceder a una propiedad o a un método cuando los elementos del array son objetos, puedes usar la notación de punto detrás de los corchetes, por ejemplo: `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[2]);
```



Autoevaluación

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

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

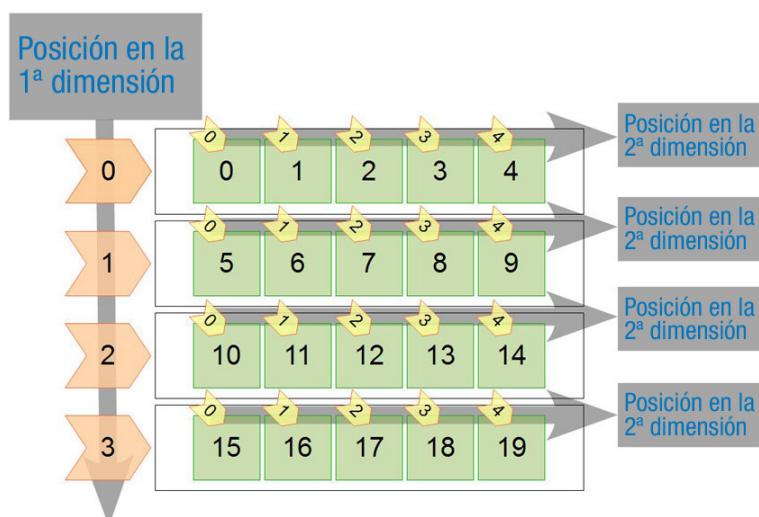
3.- Arrays multidimensionales.

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

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

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

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



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

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

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

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

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



Autoevaluación

Completa con los números que faltan:

<input type="radio"/>	int[][][] k=new int[10][11][12];
<input type="radio"/>	

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

3.1.- Uso de arrays multidimensionales.

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

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

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

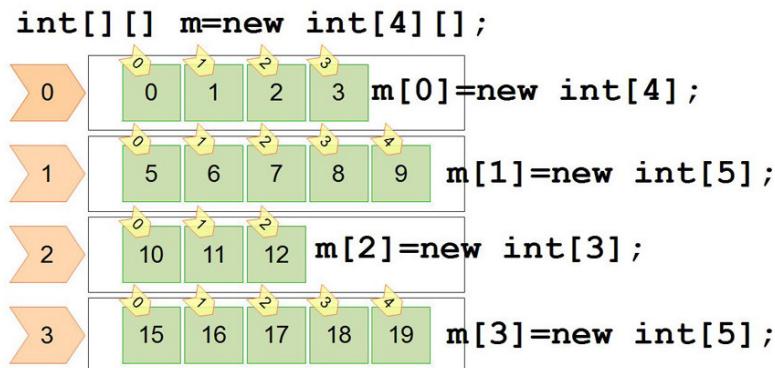
```
int suma = a2d[0][0] + a2d[0][1] + a2d[0][2] + a2d[0][3] + a2d[0][4];
```

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;
}
```

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

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



Una matriz es un ejemplo de array multidimensional **regular**, ¿por qué? Pues porque es un array que contiene arrays, 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 cómo 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) {...}
```

3.2.- Inicialización de arrays multidimensionales.

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

Para que un método retorne un array multidimensional, se hace igual que con 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:

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

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

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

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

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

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



Autoevaluación

Dado el siguiente array: `int||||| i3d={{ {0,1},{0,2} }, { {0,1,3} }, { {0,3,4},{0,1,5} } };`

¿Cuál es el valor de la posición [1][1][2]?

- 3
- 4
- 5
- Ninguno de los anteriores.

4.- Operaciones con arrays.

Cuando manejamos listas de información con muchos elementos, el hecho de que los elementos de un array estén ordenados facilita bastante las búsquedas de un elemento determinado, ya que en muchos casos no habrá que recorrer toda la lista para saber si el elemento que buscamos está o no está en ella. Por eso están ordenadas las palabras de un diccionario, o los nombres del listín telefónico, o los alumnos de una lista de clase.

Basta con que encontremos una palabra que alfabéticamente va después de la que buscamos sin que hayamos encontrado ésta, para que podamos afirmar que esa palabra no aparece en el diccionario, y abandonamos la búsqueda sin tener que leer todo el diccionario. Si las palabras no estuvieran en orden alfabético, tendríamos que revisar todo el diccionario, desde el principio hasta el final, antes de poder asegurar que esa palabra no está en el diccionario.

Lo mismo ocurre en el caso de los arrays. Y además, siempre que queramos:

- ✓ borrar,
- ✓ consultar,
- ✓ modificar los datos de uno de los elementos del array,

lo primero que tendremos que hacer es buscarlo y localizarlo dentro del array.

Si el array no está lleno, resulta útil que esté “empaquetado”, es decir,

- ✓ que todos los elementos que contenga estén juntos,
- ✓ sin huecos libres, al principio del array,
- ✓ y que todos los huecos o posiciones libres del vector estén juntos al final del array.

De esta forma, cuando buscamos un elemento y encontramos un hueco, también podemos abandonar la búsqueda, ya que a partir de ahí sólo van a quedar huecos.

Si queremos que el array esté permanentemente ordenado para aprovecharnos de búsquedas más rápidas, que harán que otras operaciones también sean más rápidas, mejorando la eficiencia de nuestra aplicación, debemos preocuparnos de mantener ese orden cada vez que insertamos un nuevo elemento y preocuparnos de que el vector quede empaquetado cada vez que borramos un elemento.

La clase `Arrays` facilita, entre otros, el método `sort` que permiten ordenar arrays. La veremos en el tema siguiente.

El siguiente ejemplo de ordenaría los números de un array de forma ascendente (de menor a mayor):

Ordenación natural en arrays.

Ejemplo de ordenación de un array de números

```
import java.util.Arrays;
public class ejemploOrdenacion {
    public static void main(String[] args) {
        int[] array={10,9,99,3,5};
        Arrays.sort(array);
        for(int elemento:array)
            System.out.println(elemento);
    }
}
```

4.1.- Búsquedas

La búsqueda en un array consiste en comprobar si un elemento o varios del array contienen un valor determinado. En caso de que se encuentre, el algoritmo dará como resultado la posición en la que se encuentra dicho valor. Dependiendo de si el array está ordenado o no, la búsqueda se podrá realizar mediante un método u otro.

Búsqueda en arrays desordenados

Consiste en recorrer los elementos del array, y para cada uno de ellos comprobar si dicho elemento coincide con el valor a buscar. Hay dos condiciones que interrumpen la búsqueda:

- ✓ se ha localizado el elemento que se buscaba.
- ✓ se recorre todo el vector y no se ha encontrado el elemento.

Para buscar un valor en un array **unidimensional** se recorre el vector de izquierda a derecha, hasta encontrar el valor que coincide con el buscado o hasta que se acabe el vector. En el ejemplo que sigue se busca un valor *x* en un array unidimensional.

```
i=0;
while( i<TAM-1 && vector[i] != x){
    i++;
}//fin while

//Imprimo el resultado.
if ( vector[i]== x )
    System.out.println ("Elemento encontrado en la posicion de indice " + i );
else
    System.out.println ("Elemento no encontrado.");
```

La búsqueda en un array **multidimensional**, se realiza anidando varios bucles cuya finalización vendrá dada por la aparición del valor o por haber recorrido el array o tabla completamente.

Búsqueda en arrays ordenados

- ✓ Búsqueda **secuencial**: Es el mismo método visto para arrays desordenados, pero teniendo en cuenta que si el array estuviera ordenado, por ejemplo de forma ascendente, la búsqueda se realizará hasta encontrar el elemento en cuestión o hasta alcanzar un elemento con un valor superior al buscado.

```
i=0;
while ( vector[i] < x ){
    i++;
}//fin while

//Imprimo el resultado.
if (vector[i] == x )
    System.out.println ("Elemento encontrado en la posicion " + i );
else
    System.out.println ("Elemento no encontrado.");
```

- ✓ Búsqueda **dicotómica o binaria**: Existe otro método más efectivo si el array está ordenado

Llamado búsqueda binaria o dicotómica. Consiste en comparar el elemento que ocupa la posición central del array con el valor a buscar. Si coincide, la búsqueda ha terminado. Si no coincide, la búsqueda se reduce al intervalo comprendido entre el primer elemento y el que ocupa la posición central, o bien al intervalo comprendido entre el valor central y el último, dependiendo de si el valor a buscar es menor o mayor que el contenido del elemento central. Con este nuevo intervalo de búsqueda se repite el proceso, es decir se calcula cual es el nuevo elemento central y se compara su contenido con el valor a buscar. Este proceso acaba cuando el elemento que ocupa el valor central del intervalo coincide con el valor buscado o cuando los límites del intervalo coinciden, deduciéndose en este caso que el valor buscado no está en el array. Ejemplo siendo x el valor a buscar:

```
int izq=0;
int der=TAM-1;
cen=((izq+der)/2);      // se calcula el centro del vector
while(vector[cen]!= x && izq<der) {
    if(vector[cen]< x)
        izq=cen+1;      // se cambia el límite izquierdo
    else
        der=cen-1;      // se cambia el límite derecho
    cen=((izq+der)/2); // nuevo centro
} //fin while
//Imprimo el resultado.
if(vector[cen]== x)
    System.out.println ("Elemento encontrado en la posición " + cen+1);
else
    System.out.println ("Elemento no encontrado.");
```

4.2.- Ordenaciones.

La ordenación de un array consiste en organizar sus elementos con respecto a un criterio. Por ejemplo, si los elementos son enteros, podemos referirnos a una ordenación que puede ser creciente o decreciente.

Vamos a ver algunos métodos aplicados a los vectores o arrays unidimensionales puesto que estos métodos se pueden generalizar a tablas de dos o más dimensiones. Existen muchos más métodos de ordenación.

- ✓ Ordenación por inserción directa o método de la **baraja**.

Supongamos un vector V de N componentes que deseamos ordenar de manera ascendente (de menor a mayor). El método de la baraja consiste en repetir el siguiente proceso desde la segunda componente del vector hasta la última: “*Se toma la componente que toca y se inserta en el lugar que le corresponde entre las componentes situadas a su izquierda (que ya estarán ordenadas). Las componentes superiores a la tratada se desplazan un lugar a la derecha*”.

Ejemplo	3	2	4	1	2
Paso 1	2	3	4	1	2
Paso 2	2	3	4	1	2
Paso 3	1	2	3	4	2
Paso 4	1	2	2	3	4

La función Java correspondiente sería:

```

final static int DIM = 15;      // común para todas las funciones
public static void ordenarBaraja (int[ ] vector) {
    int i,j,aux;
    for (i=1;i<DIM;i++) {
        for (j=0; j<i ; j++)
            if (vector[i]<vector[j]) {
                aux=vector[j];
                vector[j]=vector[i];
                vector[i]=aux;
            }
    }
    //Llamada al método:
    ordenarBaraja(array);
}

```

- ✓ Ordenación por **selección directa**.

El método consiste en repetir el siguiente proceso desde la primera componente hasta la penúltima. Se selecciona la componente de menor valor de todas las situadas a la derecha de la tratada y se intercambia con ésta.

Ejemplo	3	2	4	1	2
Paso 1	1	2	4	3	2
Paso 2	1	2	4	3	2
Paso 3	1	2	2	3	4
Paso 4	1	2	2	3	4

La función Java correspondiente sería:

```

public static void ordenarSeleccion (int[ ] vector) {
    int i,j,aux,minimo, posicion;
    for (i=0;i<DIM-1;i++) {
        minimo=vector[i];
        for (j=i+1;j<DIM;j++)
            if (minimo>vector[j]) {
                minimo=vector[j];
                posicion=j;
            }
        if (vector[i]>minimo) {
            aux=vector[i];
            vector[i]=vector[posicion];
            vector[posicion]=aux;
        }
    }
}

```

✓ Ordenación por intercambio directo, método de la **burbuja**.

Este método tiene dos versiones basadas en la misma idea que consiste en recorrer sucesivamente el vector comparando los elementos consecutivos e intercambiándolos cuando estén descolocados. El recorrido se puede hacer de izquierda a derecha (desplazando los valores mayores hacia la derecha) o de derecha a izquierda (desplazando los valores menores hacia su izquierda), ambos casos para la clasificación ascendente.

Ejemplo	3	2	4	1	2
Paso 1	2	3	1	2	4
Paso 2	2	1	2	3	4
Paso 3	1	2	2	3	4
Paso 4	1	2	2	3	4

Recorrido de Izquierda a Derecha

Va colocando en cada pasada el mayor elemento de los tratados en la última posición, quedando colocado y por lo tanto excluido de los elementos a tratar en la siguiente pasada.

Ejemplo	3	2	4	1	2
Paso 1	1	3	2	4	2
Paso 2	1	2	3	2	4
Paso 3	1	2	2	3	4
Paso 4	1	2	2	3	4

Recorrido de Derecha a Izquierda

Esta versión va colocando en cada pasada el menor elemento de los tratados en la primera posición, quedando colocado y por lo tanto excluido de los elementos a tratar en la siguiente pasada.

La función Java correspondiente sería:

```

public static void ordenarBurbuja (int[ ] vector) {
    int i,j,aux;
    for (i=0;i<DIM-1;i++) {
        for (j=0;j<DIM-1;j++) {
            if (vector[ j ] > vector[ j+1 ] ) {
                aux=vector[j];
                vector[j]=vector[j+1];
                vector[j+1]=aux;
            }
        }
    }
}

```

5.- Cadenas de caracteres. Clase String

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

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

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

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

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

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

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

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

5.1.- Concatenación.

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

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

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

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

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

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

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

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

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

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

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

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



Autoevaluación

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

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

5.2.- Operaciones con cadenas de caracteres (I).

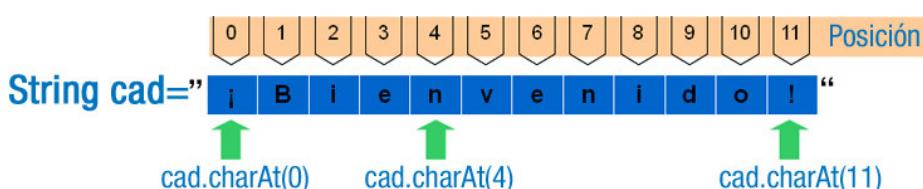
Vamos a continuar revisando las operaciones que se pueden realizar con cadenas. Como verás las operaciones a realizar se complican un poco a partir de ahora. En todos los ejemplos la variable `cad` contiene la cadena "¡Bienvenido!", como se muestra en las imágenes.

- ✓ `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.

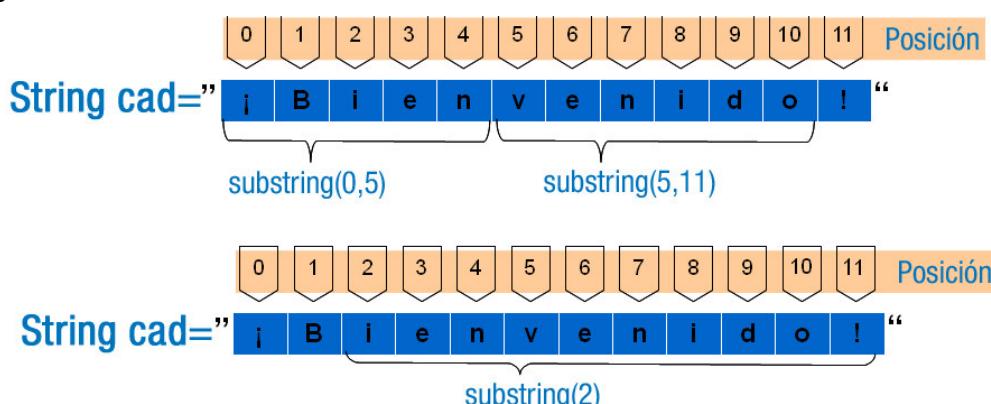


- ✓ `char charAt(int pos)`. Retorna el carácter ubicado en la posición pasada por parámetro. El carácter obtenido de dicha posición será almacenado en un tipo de dato `char`. Las posiciones se empiezan a contar desde el 0 (y no desde el 1), y van desde 0 hasta `longitud - 1`. Por ejemplo, el código siguiente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

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

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

Otra operación muy habitual es la conversión de número a cadena y de cadena a número. Imagínate que un usuario introduce su edad. Al recoger la edad desde la interfaz de usuario, capturarás generalmente

una cadena, pero, ¿cómo compruebas que la edad es mayor que 0? Para poder realizar esa comprobación tienes que pasar la cadena a número. Empezaremos por ver como se convierte un número a cadena.

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

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

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

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



Reflexiona

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

5.3.- Operaciones con cadenas de caracteres (II).

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

Fíjate en el código anterior, en él puedes comprobar cómo la cadena `c` contiene en su interior un número, pero escrito con dígitos numéricos (caracteres). El código escrito está destinado a transformar la cadena en número, usando el método `valueOf`.

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

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

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

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

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



Debes conocer

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

[Formateado de cadenas en Java.](#)

5.4.- Operaciones con cadenas de caracteres (III).

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

Métodos importantes de la clase String.

Método.	Descripción
<code>cad1.compareTo(cad2)</code>	Permite comparar dos cadenas entre sí lexicográficamente. Retornará 0 si son iguales, un número menor que cero si la cadena (<code>cad1</code>) es anterior en orden alfabético a la que se pasa por argumento (<code>cad2</code>), y un número mayor que cero si la cadena es posterior en orden alfabético.
<code>cad1.equals(cad2)</code>	Cuando se comparan si dos cadenas son iguales, no se debe usar el operador de comparación " <code>==</code> ", sino el método <code>equals</code> . Retornará <code>true</code> si son iguales, y <code>false</code> si no lo son.
<code>cad1.compareToIgnoreCase(cad2)</code> <code>cad1.equalsIgnoreCase(cad2)</code>	El método <code>compareToIgnoreCase</code> funciona igual que el método <code>compareTo</code> , pero ignora las mayúsculas y las minúsculas a la hora de hacer la comparación. Las mayúsculas van antes en orden alfabético que las minúsculas, por lo que hay que tenerlo en cuenta. El método <code>equalsIgnoreCase</code> es igual que el método <code>equals</code> pero sin tener en cuenta las minúsculas.
<code>cad1.trim()</code>	Genera una copia de la cadena eliminando los espacios en blanco anteriores y posteriores de la cadena.
<code>cad1.toLowerCase()</code>	Genera una copia de la cadena con todos los caracteres a minúscula.
<code>cad1.toUpperCase()</code>	Genera una copia de la cadena con todos los caracteres a mayúsculas.
<code>cad1.indexOf(cad2)</code> <code>cad1.indexOf(cad2,num)</code>	Si la cadena o carácter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1. Opcionalmente se le puede indicar la posición a partir de la cual buscar, lo cual es útil para buscar varias apariciones de una cadena dentro de otra.
<code>cad1.contains(cad2)</code>	Retornará <code>true</code> si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará <code>false</code> .
<code>cad1.startsWith(cad2)</code>	Retornará <code>true</code> si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .
<code>cad1.endsWith(cad2)</code>	Retornará <code>true</code> si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .
<code>cad1.replace(cad2,cad3)</code>	Generará una copia de la cadena <code>cad1</code> , en la que se reemplazarán todas las apariciones de <code>cad2</code> por <code>cad3</code> . El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "zzz" por "xx" en la cadena "zzzz" generará "xxzz" y no "zxzz".

Método.	Descripción
cad1.split(caracter)	Divide la cadena <code>cad1</code> a partir del carácter que se usa como delimitador, y devuelve un array de Strings en el que cada elemento es cada parte de la cadena. El delimitador o separador es una expresión regular, único argumento del método <code>split</code> , y puede ser todo lo complejo que sea necesario.



Autoevaluación

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

- 2.
- 3.
- 4.
- 1.

5.5.- La clase StringBuilder

¿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 mutable, y permite una mayor optimización de la memoria. También existe la clase `StringBuffer`, pero consume mayores recursos al estar pensada para aplicaciones multi-hilo, por lo que en nuestro caso nos centraremos en la primera.

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

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

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

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

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

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



Debes conocer

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

[Uso de la clase `StringBuilder`.](#)



Autoevaluación

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

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

6.- Excepciones.

A lo largo de nuestro aprendizaje de Java nos hemos topado en alguna ocasión con Errores, pero éstos suelen ser los que nos ha indicado el compilador. Un punto y coma por aquí, un nombre de variable incorrecto por allá, pueden hacer que nuestro compilador nos avise de estos descuidos. Cuando los vemos, se corrigen y obtenemos nuestra clase compilada correctamente.

Pero, ¿Sólo existen este tipo de Errores? ¿Podrían existir Errores no sintácticos en nuestros programas? Está claro que sí, un programa perfectamente compilado en el que no existen Errores de sintaxis, puede generar otros tipos de Errores que quizás aparezcan en tiempo de ejecución. A estos Errores se les conoce como **excepciones**.

Aprenderemos a gestionar de manera adecuada estas excepciones y tendremos la oportunidad de utilizar el potente sistema de manejo de Errores que Java incorpora. La potencia de este sistema de manejo de Errores radica en:

1. Que el código que se encarga de manejar los Errores, es perfectamente identificable en los programas. Este código puede estar separado del código que maneja la aplicación.
2. Que Java tiene una gran cantidad de Errores estándar asociados a multitud de fallos comunes, como por ejemplo divisiones por cero, fallos de entrada de datos, etc. Al tener tantas excepciones localizadas, podemos gestionar de manera específica cada uno de los Errores que se produzcan.

En Java se pueden preparar los fragmentos de código que pueden provocar Errores de ejecución para que si se produce una excepción, el flujo del programa es lanzado (throw) hacia ciertas zonas o rutinas que han sido creadas previamente por el programador y cuya finalidad será el tratamiento efectivo de dichas excepciones. Si no se captura la excepción, el programa se detendrá con toda probabilidad.

En Java, las excepciones están representadas por clases. El paquete **java.lang.Exception** y sus subpaquetes contienen todos los tipos de excepciones. Todas las excepciones derivarán de la clase **Throwable**, existiendo clases más específicas. Por debajo de la clase **Throwable** existen las clases **Error** y **Exception**.

- ✓ **Error** es una clase que se encargará de los Errores que se produzcan en la máquina virtual, no en nuestros programas.
- ✓ Y la clase **Exception** será la que a nosotros nos interese conocer, pues gestiona los Errores provocados en los programas.

Java lanzará una excepción en respuesta a una situación poco usual. Cuando se produce un error se genera un objeto asociado a esa excepción. Este objeto es de la clase **Exception** o de alguna de sus herederas. Este objeto se pasa al código que se ha definido para manejar la excepción. Dicho código puede manipular las propiedades del objeto **Exception**.

El programador también puede lanzar sus propias excepciones. Las excepciones en Java serán objetos de clases derivadas de la clase base **Exception**. Existe toda una jerarquía de clases derivada de la clase base **Exception**. Estas clases derivadas se ubican en dos grupos principales:

- ✓ Las excepciones en tiempo de ejecución, que ocurren cuando el programador no ha tenido cuidado al escribir su código.
- ✓ Las excepciones que indican que ha sucedido algo inesperado o fuera de control.

En la siguiente imagen te ofrecemos una aproximación a la jerarquía de las excepciones en Java.



6.1.- Captura de excepciones.

Para poder capturar excepciones, emplearemos la estructura de captura de excepciones **try-catch-finally**.

Básicamente, para capturar una excepción lo que haremos será declarar bloques de código donde es posible que ocurra una excepción. Esto lo haremos mediante un bloque **try** (intentar). Si ocurre una excepción dentro de estos bloques, se lanza una excepción. Estas excepciones lanzadas se pueden capturar por medio de bloques **catch**. Será dentro de este tipo de bloques donde se hará el manejo de las excepciones.

Su sintaxis es:

```

try {
    código que puede generar excepciones;
}
catch (Tipo_excepcion_1 objeto_excepcion) {
    Manejo de excepción de Tipo_excepcion_1;
} catch (Tipo_excepcion_2 objeto_excepcion) {
    Manejo de excepción de Tipo_excepcion_2;
}
...
finally {
    instrucciones que se ejecutan siempre
}

```

En esta estructura, la parte **catch** puede repetirse tantas veces como excepciones diferentes se deseen capturar. La parte **finally** es opcional y, si aparece, solo podrá hacerlo una sola vez.

Cada **catch** maneja un tipo de excepción. Cuando se produce una excepción, se busca el **catch** que posea el manejador de excepción adecuado, será el que utilice el mismo tipo de excepción que se ha producido. Esto puede causar problemas si no se tiene cuidado, ya que la clase **Exception** es la superclase de todas las demás. Por lo que si se produjo, por ejemplo, una excepción de tipo **AritmeticException** y el primer **catch** capture el tipo genérico **Exception**, será ese **catch** el que se ejecute y no los demás.

Por eso el último **catch** debe ser el que capture excepciones genéricas y los primeros deben ser los más específicos. Lógicamente si vamos a tratar a todas las excepciones (sean del tipo que sean) igual, entonces basta con un solo **catch** que capture objetos **Exception**.



Ejercicio resuelto

Realiza un programa en Java en el que se solicite al usuario la introducción de un número por teclado comprendido entre el 0 y el 100. Utilizando manejo de excepciones, debes controlar la entrada de dicho número y volver a solicitarlo en caso de que ésta sea incorrecta.



Autoevaluación

Si en un programa no capturamos una excepción, será la máquina virtual de Java la que lo hará por nosotros, pero inmediatamente detendrá la ejecución del programa y mostrará una traza y un mensaje de error. Siendo una traza, la forma de localizar dónde se han producido errores. ¿Verdadero o Falso?

- Verdadero
- Falso

6.2.- Manejo de excepciones.

Como hemos comentado, siempre debemos controlar las excepciones que se puedan producir o de lo contrario nuestro software quedará expuesto a fallos. Las excepciones pueden tratarse de dos formas:

- ✓ **Interrupción.** En este caso se asume que el programa ha encontrado un error irrecuperable. La operación que dio lugar a la excepción se anula y se entiende que no hay manera de regresar al código que provocó la excepción. Es decir, la operación que originó el error, se anula.
- ✓ **Reanudación.** Se puede manejar el error y regresar de nuevo al código que provocó el error.

Java emplea la primera forma, pero puede simularse la segunda mediante la utilización de un bloque `try` en el interior de un `while`, que se repetirá hasta que el error deje de existir. En la siguiente imagen tienes un ejemplo de cómo llevar a cabo esta simulación.

```
public static void main(String[] args) {
    boolean fueraDeLimites=true;
    int i; //tomará valores aleatorios de 0 a 9
    //declaro un array de 5 elementos de tipo String
    String texto[] = {"uno", "dos", "tres", "cuatro", "cinco"};

    while (fueradeLimites){
        try{
            i=(int) Math.round(Math.random()*9); //generamos un indice aleatorio
            System.out.println(texto[i]);
            fueraDeLimites=false;
        }catch(ArrayIndexOutOfBoundsException e){
            //si el indice está fuera de rango
            System.out.println("Fallo en el indice: "+e.getMessage());
        }
    }
}
```

En este ejemplo, a través de la función de generación de números aleatorios se obtiene el valor del índice `i`. Con dicho valor se accede a una posición del array que contiene cinco cadenas de caracteres. Este acceso, a veces puede generar un error del tipo `ArrayIndexOutOfBoundsException`, que debemos gestionar a través de un `catch`. Al estar el bloque `catch` dentro de un `while`, se seguirá intentando el acceso hasta que no haya error.

6.3.- Delegación de excepciones con throws.

¿Puede haber problemas con las excepciones al usar llamadas a métodos en nuestros programas? Efectivamente, si se produjese una excepción es necesario saber quién será el encargado de solucionarla. Puede ser que sea el propio método llamado o el código que hizo la llamada a dicho método.

Quizá pudiéramos pensar que debería ser el propio método el que se encargue de sus excepciones, aunque es posible hacer que la excepción sea resuelta por el código que hizo la llamada. Cuando un método utiliza una sentencia que puede generar una excepción, pero dicha excepción no es capturada y tratada por él, sino que se encarga su gestión a quién llamó al método, decimos que se ha producido **delegación de excepciones**.

Para establecer esta delegación, en la cabecera del método se declara el tipo de excepciones que puede generar y que deberán ser gestionadas por quien invoque a dicho método. Utilizaremos para ello la sentencia **throws** y tras esa palabra se indica qué excepciones puede provocar el código del método. Si ocurre una excepción en el método, el código abandona ese método y regresa al código desde el que se llamó al método. Allí se posará en el **catch** apropiado para esa excepción. Su sintaxis es la siguiente:

```
public class delegacionExcepciones {
    ...
    public int leeAño(BufferedReader lector) throws IOException, NumberFormatException{
        String linea = teclado.readLine();
        return Integer.parseInt(linea);
    }
    ...
}
```

Donde **IOException** y **NumberFormatException**, serían dos posibles excepciones que el método **leeaño** podría generar, pero que no gestiona. Por tanto, un método puede incluir en su cabecera un listado de excepciones que puede lanzar, separadas por comas.

Al redefinir un método heredado podemos modificar la declaración de las excepciones comprobadas lanzadas (**throws**). Teniendo en cuenta que:

- ✓ Sólo es posible reducir la lista de excepciones.
- ✓ No se puede incluir una excepción comprobada que no lance el método de la clase padre.
- ✓ Es posible indicar una excepción más específica que la que se hereda:

Ejemplo: en la clase padre el método lanza **IOException** y la redefinición **FileNotFoundException** que es un subtipo.

Anexo I.- Formateado de cadenas en Java.

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

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

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

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

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

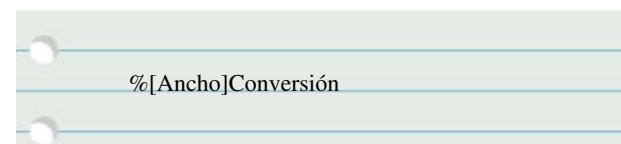
Listado de conversiones más utilizada y ejemplos.

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

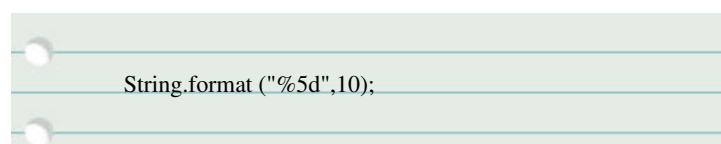
Tipo de conversión	Especificación de formato	Tipos de datos aplicables	Ejemplo
Número en notación científica	"%e" o "%E"	Flotantes simples o dobles.	<pre>double i=10.5; String d= String.format("Resultado: %E", i); System.out.println(d);</pre>
Número decimal	"%f"	Flotantes simples o dobles.	<pre>float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

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

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

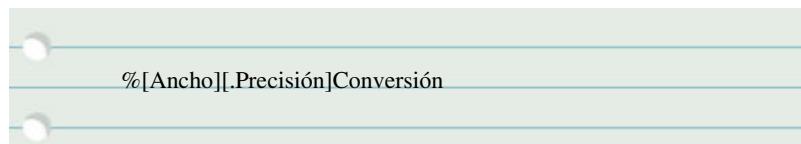


El hecho de que esté entre corchetes significa que es opcional. Si queremos por ejemplo que la salida genere al menos 5 caracteres (poniendo espacios delante) podríamos ponerlo así:

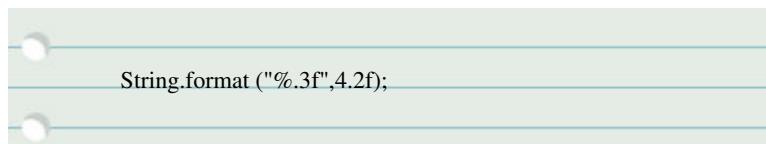


Se mostrará el "10" pero también se añadirán 3 espacios delante para llenar. Este tipo de modificador se puede usar con cualquier conversión.

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:

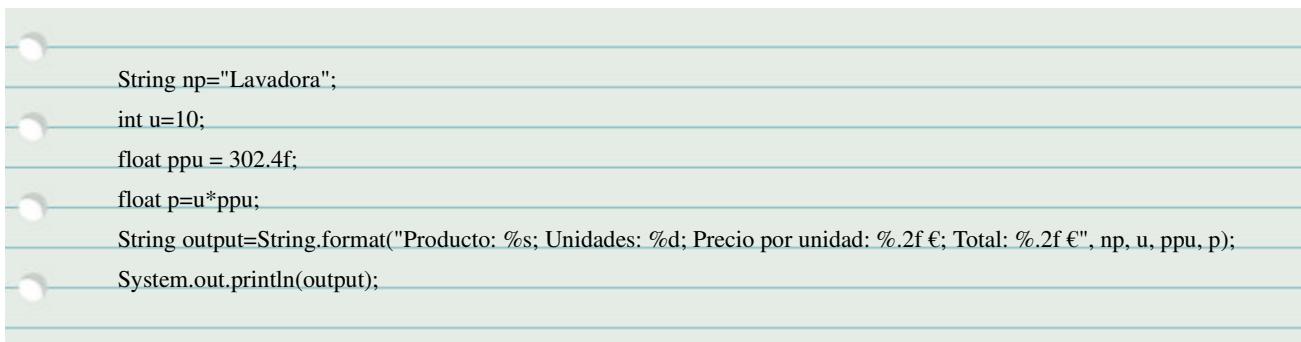


Como puedes ver, tanto el ancho como la precisión van entre corchetes, los corchetes no hay que ponerlos, solo indican que son modificaciones opcionales. Si queremos, por ejemplo, que la salida genere 3 decimales como mínimo, podremos ponerlo así:

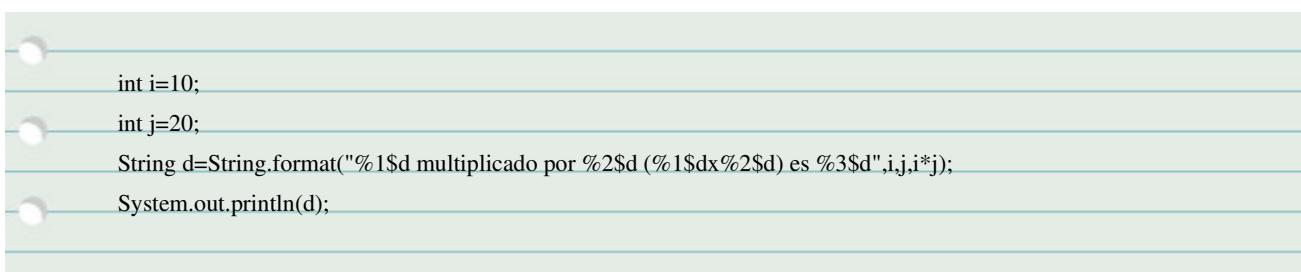


Como el número indicado como parámetro solo tiene un decimal, el resultado se completará con ceros por la derecha, generando una cadena como la siguiente: "4,200".

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



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:



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).