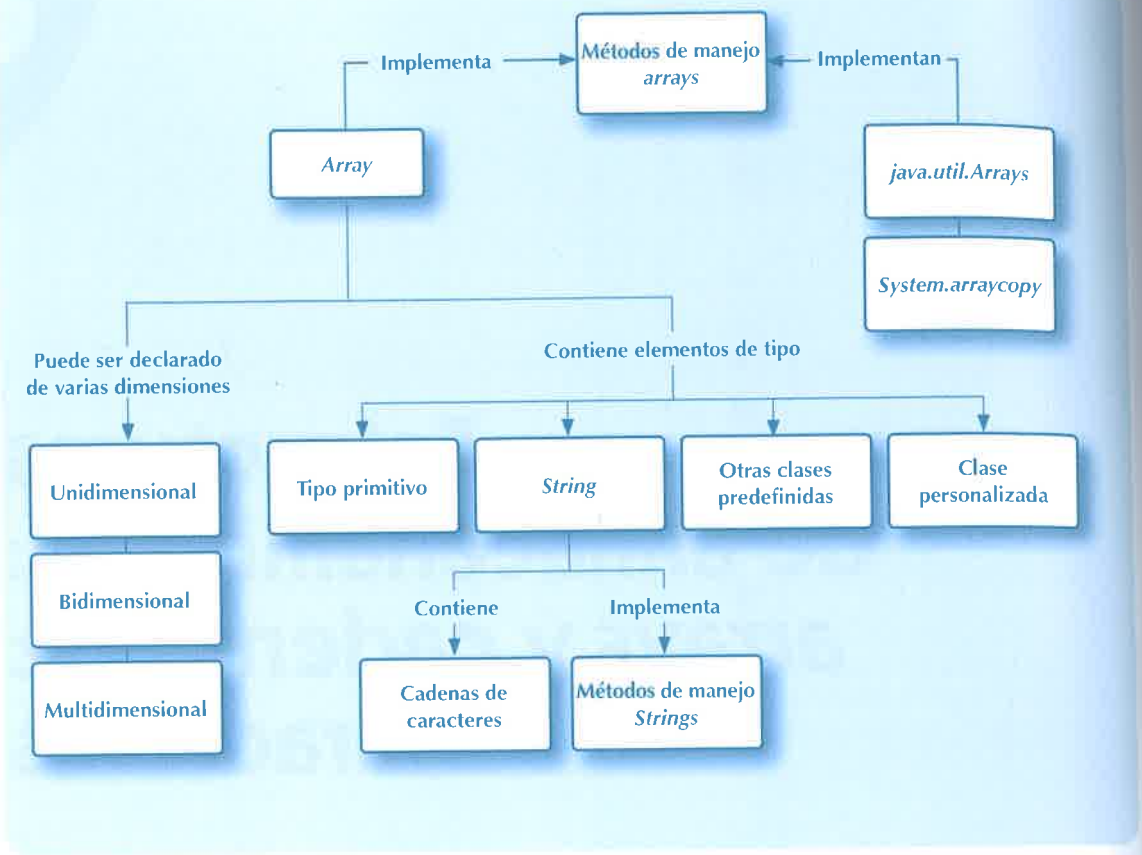
8

Estructuras de almacenamiento: *arrays* y cadenas de caracteres

Objetivos

- ✓ Familiarizarse con las diferentes estructuras de almacenamiento empleadas en Java.
- ✓ Comprender la estructura interna de almacenamiento de un *array* estático.
- ✓ Profundizar en el tipo de datos empleado para almacenar cadenas de caracteres.
- ✓ Conocer los diferentes métodos que permiten manejar los datos de un *array* y de una cadena de caracteres.

Mapa conceptual



Glosario

Array. También conocido como *arreglo* o *matriz* en caso de tener 2 dimensiones, es un conjunto de datos de un tipo determinado, primitivo o no, que se denominan *elementos*. Estos son almacenados juntos y son accedidos a través de un número entero denominado índice.

Cheatsheet. Referencia o guía rápida que permite consultar de forma ágil los contenidos de una temática determinada.

Concatenación. Unión de dos o más elementos.

DNI. Abreviatura de documento nacional de identidad, cuyo identificador se compone de un número formado por 8 dígitos y una letra.

Elemento. Cada uno de los valores contenidos en las posiciones de un *array*. Todos los elementos de un *array* serán del mismo tipo, pudiendo ser este una clase o un tipo primitivo.

Expresión regular. Secuencia de caracteres cuya disposición y orden representa un patrón de búsqueda. Son empleadas a menudo para validar campos en formularios HTML o datos que deben ser procesados y almacenados solo en caso de ser acordes a un determinado formato.

Lexicográfico. Se dice orden lexicográfico como el orden presentado por los caracteres de un determinado alfabeto.

Null. Valor empleado en programación para representar la nada. Cuando un objeto es declarado, pero no inicializado, se dice que apunta a null.

Shallow copy. Copia de un *array* origen hacia un *array* destino en la que únicamente se clona la estructura de almacenamiento y las referencias a los datos en ella incluidas, sin duplicar los espacios de memoria en los que se almacenan los datos en sí. También se conoce como *copia superficial*.

Unicode. Sistema de codificación universal que asocia una secuencia de caracteres a cada carácter a codificar.

8.1. Introducción

Hasta llegar a este capítulo se ha trabajado con tipos de datos primitivos o bien con tipos de datos compuestos de tipos primitivos u otros tipos compuestos a su vez, pero en cualquier caso se ha estado manejando un único dato por cada variable declarada. Estos tipos, que puede parecer suficientes para programas básicos, no disponen de la capacidad necesaria para desarrollos de programas de mayor complejidad. Un ejemplo muy sencillo de entender es el de un programa que tenga que gestionar un número elevado de datos de un mismo tipo para trabajar con ellos de forma recurrente. La alternativa basada en esos tipos primitivos obligaría a declarar tantas variables como datos sea necesario almacenar y a acceder a ellas como variables independientes que serán (el número de variables a declarar será directamente proporcional al número de datos a manejar). Si, por el contrario, es posible emplear un tipo que permita almacenar a todos estos datos juntos, únicamente será necesario declarar una variable.

En Java, al igual que en otros lenguajes de programación modernos, se dispone de tipos especiales que actúan como contenedores de otros elementos. Este tipo se denomina *array* y permite representar a un objeto contenedor de datos de un mismo tipo. Cada ítem del *array* es denominado *elemento*.

8.2. Estructuras

Una estructura es un tipo de dato definido por el programador y que permite agrupar datos relacionados entre sí. Algunos lenguajes como C o Algol, ofrecen formas específicas de definir estos tipos de datos (*struct*). Java, por el contrario, no ofrece esta posibilidad; sin embargo, es posible simular dichas estructuras mediante la declaración de clases para construir tipos compuestos.

INVESTIGA

Busca información sobre los tipos *struct* y *union* que pueden utilizarse en el lenguaje de programación C. ¿Cuál sería la diferencia fundamental entre ambos?

8.3. *Arrays*

Tal como se ha indicado en la introducción, el tipo *array* es un objeto que representa a un contenedor de datos del mismo tipo. Cada ítem del *array* es denominado *elemento*.

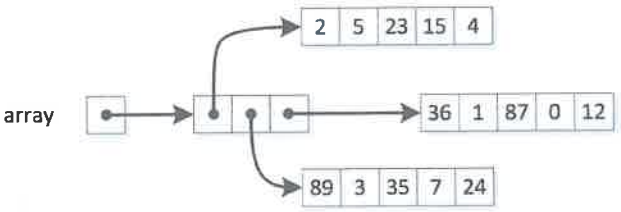
Un *array* puede ser *estático* o *dinámico*. La diferencia radica, conceptualmente hablando, en que en un *array estático* el número de elementos es fijo desde el momento en el que se declara y se reserva el espacio en memoria, mientras que en un *array dinámico* es variable en tiempo de ejecución. En este capítulo únicamente se trabajará con *arrays estáticos*; los *arrays dinámicos* y otras colecciones serán tratados en el capítulo 9 (Colecciones y tipos abstractos de datos).

8.3.1. Dimensiones de un *array*

Un *array* puede almacenar datos en varias dimensiones. En función de ello se distingue entre:

1. Array *unidimensional*, que maneja datos en una única dimensión y por tanto sus elementos pueden representarse de forma lógica, que no física, linealmente:

2. Array *multidimensional*, que se trata de un *array* cuyos elementos son a su vez *arrays*. Si se utilizan dos dimensiones, se puede hablar de *array bidimensional* o *matriz*:



8.3.2. Declaración y creación

Un *array* será declarado indicando *tipo* de los elementos, dimensión y nombre de la variable (los espacios son opcionales). Cada pareja de corchetes ([]) indicará una dimensión. Es posible alternar el orden de la declaración de dimensión y nombre de la variable. En el ejemplo mostrado a continuación se han declarado 5 *arrays* de diferentes dimensiones:

```
short [] edades;      // array unidimensional
int []pesos;          // array unidimensional
int precios [];       // array unidimensional (sintaxis heredada C/C++)
int [][] matriz;      // array bidimensional
byte [][][] arrayTridimensional;     // array tridimensional
```

Una vez declarado es necesario reservar el espacio en memoria de los datos que almacenará:

```
edades = new short[10];
pesos = new int[50];
precios = new int[100];
matriz = new int[4][2];
arrayTridimensional = new byte[4][3][10];
```

8.3.3. Inicialización y acceso

Los *arrays* deberán inicializarse con datos mediante una simple asignación. El acceso a cada elemento de *array* se realizará mediante un índice que marca la posición de este. En un *array* de *N* elementos el primer elemento del *array* será accedido con índice *0*, el segundo con índice *1*... y así sucesivamente hasta el último elemento que será accedido con índice *N-1*. En caso de intentar acceder a elementos que no estén comprendidos en este rango, el sistema lanzará la excepción *java.lang.ArrayIndexOutOfBoundsException*. En el ejemplo siguiente se muestra cómo declarar, inicializar y acceder a los diferentes elementos de un *array*:

```
edades [0]=1;
pesos [1]=3;
precios [1]=2;
matriz[2][1]=200;
arrayTridimensional[2][2][0]=127;
System.out.println(pesos[1]);
System.out.println(matriz[2][1]);
System.out.println(arrayTridimensional[2][2][0]);
```

SABÍAS QUE...

Es posible realizar la declaración, creación e inicialización del *array* al mismo tiempo. Para ello se puede emplear la *sintaxis abreviada:*

```
char[] letras={'a','b','c'};
byte[][] arrayBidimensional = {
    {1, 2, 3},
    {4, 5}
};
```

Gracias a este tipo de declaración se pueden declarar *arrays* multidimensionales cuyos elementos pueden ser *arrays* de distintas dimensiones entre sí.

8.3.4. *Arrays* y métodos

Un *array* puede, por el simple hecho de ser un objeto, ser enviado como argumento a un método o ser devuelto como valor de retorno del método; para ello deberá ser declarado con la misma sintaxis que se empleó para declarar un objeto de tipo *array*:

```
    // invocación a método
    int [][] datosRetorno = metodo(edades,pesos,precios,arrayTridimensional);
}
// declaración de método
public static int[][] metodo (short [] array1, int []array2,
        int[]array3, byte [][][] array4){
    // ...
    int [][] arrayRetorno = new int[4][2];
    // ...
    return arrayRetorno;
}
```

Actividad propuesta 8.1

Escribe un método que retorne un *array* de 100 números enteros generados aleatoriamente.

8.3.5. Operaciones para realizar sobre un *array* estático

A) *Recorrido del* array

Dado que todo *array* tiene un campo *length* que indica la longitud con la que ha sido creado, es fácil recorrer los elementos mediante un bucle *for*:

```
for (int i=0; i<edades.length;i++) {  // for incremental
    System.out.println(edades[i]);
}
for (int j=edades.length; j>0;j--) {  // for decremental
    System.out.println(edades[j-1]);
}
```

Otra alternativa para recorrer los elementos del *array* es mediante un bucle *for each*:

```
for (int elemento: edades) {  // for each
    System.out.println(elemento);
}
```

Actividad propuesta 8.2

Escribe un método que reciba el *array* generado en la actividad propuesta 8.1 y lo recorra en orden decreciente, calculando el sumatorio de todos los números del *array* y mostrando finalmente dicho valor por pantalla.

B) Búsqueda de elementos en un array

La tecnología Java proporciona la clase *java.util.Arrays* que contiene métodos estáticos y sobrecargados para varios tipos primitivos de datos, con los que poder buscar la posición de un elemento contenido en un *array*:

```
String [] alumnos = {"Paco","Ana","Luis","Rosa","Beatriz"};
int posicion= java.util.Arrays.binarySearch(alumnos, "Rosa");
System.out.println("Encontrado en posición: " + posicion);
```

```
run:
Encontrado en posición: 3
```

C) Impresión de elementos de un array

A pesar de que un *array* puede ser recorrido incrementalmente para conocer los elementos en él contenidos, la clase *java.util.Arrays* proporciona el método *toString*, que permite obtener la secuencia de caracteres que representa al *array* en un formato legible. Como argumento es necesario pasar un objeto de tipo *array*:

```
char A []  = {'a','b','c'};
System.out.println(Arrays.toString(A));
```

```
run:
[a, b, c]
```

D) Ordenación de elementos de un array

La clase *java.util.Arrays* también permite ordenar ascendentemente los elementos del *array*:

```
int [] numeros = {121,12,33,1,55};
java.util.Arrays.sort(numeros);
System.out.println("Array ordenado: " + Arrays.toString(numeros));
```

```
run:
Array ordenado: [1, 12, 33, 55, 121]
```

E) Comparación arrays

Además de los métodos anteriormente descritos de la clase *java.util.Arrays*, existe también el método *equals* que permiten comparar el contenido de dos *arrays*:

```
int array1[] = {4,7,14,9};
int array2[] = array1;
int arrayClonado[] = array1.clone();
System.out.println(array1 == arrayClonado);
System.out.println(array1 == array2);
System.out.println(java.util.Arrays.equals(array1, arrayClonado));
```

```
run:
false
true
true
```

NOTA: Es importante destacar que la comparación realizada con el operador binario de igualdad "==" compara si los objetos de tipo *array* apuntan a la misma dirección de memoria, mientras que el método *equals* compara si ambos *arrays* tienen los mismos elementos.

F) Inicialización masiva de elementos de un array

El último de los métodos mostrados de la clase *java.util.Arrays* es el médodo *fill* que permite rellenar un *array* con los valores pasados como argumentos:

```
int [][] array = new int[3][6];
java.util.Arrays.fill(array[1],1);
java.util.Arrays.fill(array[2],2);
System.out.println("Array[0]: " + Arrays.toString(array[0]));
System.out.println("Array[1]: " + Arrays.toString(array[1]));
System.out.println("Array[2]: " + Arrays.toString(array[2]));
```

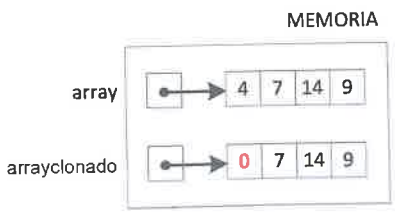
```
run:
Array[0]: [0, 0, 0, 0, 0, 0]
Array[1]: [1, 1, 1, 1, 1, 1]
Array[2]: [2, 2, 2, 2, 2, 2]
```

G) Copia de arrays

Existen varias formas de realizar copias de un *array*. En este apartado se exponen y describen las principales, haciendo especial énfasis en los diferentes resultados en función del tipo de datos de los elementos contenidos.

1. Método *clone*

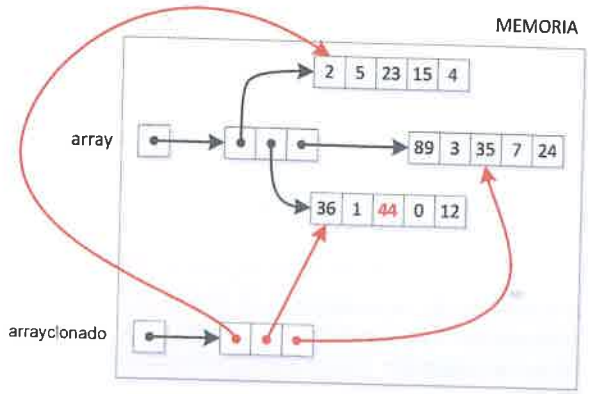
El método *clone* devuelve una copia exacta del *array* desde el cual se realiza la invocación. Hay que tener en cuenta que el resultado de la invocación será distinto dependiendo del tipo de los datos contenidos en el *array*. Si se trata de *tipos primitivos*, la invocación de *clone* devolverá un nuevo *array* cuyos datos serán totalmente independientes de los originales (los cambios en los datos del *array original* no afectarán al *array clonado*). En el siguiente ejemplo puede comprobarse cómo se lleva a cabo la copia de un *array* de números enteros:



```
int array[] = {4,7,14,9};
int arrayClonado[] = array.clone();
System.out.println(array == arrayClonado);
arrayClonado[0] = 0;
System.out.println(array [0]);
System.out.println(arrayClonado [0]);
```

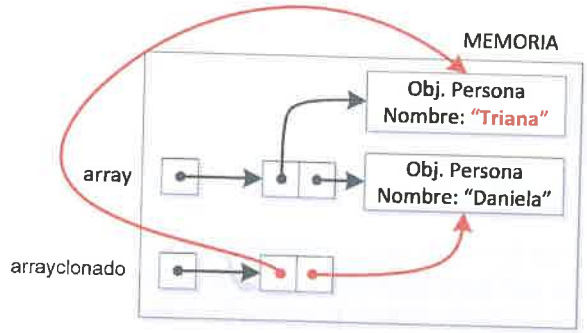
```
run:
false
4
0
```

Sin embargo, si los *tipos son no primitivos*, no se obtendrá una copia exacta sino una *Shallow copy*. Esto es debido a que los datos contenidos en el *array* original son referencias a los objetos que contienen los datos reales. Al duplicarse, lo que se está clonando es el *array* de referencias, pero estas seguirán apuntando a los mismos datos. A continuación, se muestran dos ejemplos, uno con un *array bidimensional* y otro con un *array* contenedor de objetos de la clase *Persona*:




```
int array[][] = {{2,5,23,15,4}, {36,1,87,0,12}, {89,3,35,7,24}};
int arrayClonado[][] = array.clone();
System.out.println(array == arrayClonado);
System.out.println(array [1][2]);
arrayClonado[1][2] = 44;
System.out.println(array [1][2]);
System.out.println(arrayClonado [1][2]);
```

```
run:
false
87
44
44
```




```
Persona array[] = {new Persona("Juan"), new Persona("Daniela")};
Persona arrayClonado[] = array.clone();
System.out.println(array == arrayClonado);
System.out.println(array [0].nombre);
arrayClonado[0].nombre = "Triana";
System.out.println(array [0].nombre);
System.out.println(arrayClonado [0].nombre);
```

```
run:
false
Juan
Triana
Triana
```

2. Métodos *Arrays.copyOf* y *Arrays.copyOfRange*

Los métodos *copyOf* y *copyOfRange* de la clase *java.util.Arrays* retornarán una copia total o parcial del *array* pasado como argumento (no será necesario reservar el espacio en el *array* destino antes). La sintaxis de uso es la siguiente:

```
type[] Arrays.copyOfRange (arrayOrigen, posInicio, posFinal);
type[] Arrays.copyOf (arrayOrigen, length);
```

donde:

- ✓ *type* es el tipo del dato del *array*.
- ✓ *arrayOrigen* será el *array* desde el que se quiere copiar los datos.
- ✓ *posInicio* el primer elemento desde el que comenzar la copia.
- ✓ *posFinal* el último elemento a copiar.
- ✓ *length* el número de elementos a copiar desde la primera posición.

A continuación, se puede ver un ejemplo de uso de ambos métodos:

```
int[] a = {1,2,3,4,5};
int[] b = Arrays.copyOfRange(a, 0, a.length); // copia completa de a en b
int[] c = Arrays.copyOfRange(a, a.length-2, a.length); // copia los 2 últimos elementos
int[] d = Arrays.copyOf(a, a.length); // copia completa de a en d
int[] e = Arrays.copyOf(a, 2); // copia los 2 primeros elementos
System.out.println("Array a: " + Arrays.toString(a));
System.out.println("Array b: " + Arrays.toString(b));
System.out.println("Array c: " + Arrays.toString(c));
System.out.println("Array d: " + Arrays.toString(d));
System.out.println("Array e: " + Arrays.toString(e));
```

```
run:
Array a: [1, 2, 3, 4, 5]
Array b: [1, 2, 3, 4, 5]
Array c: [4, 5]
Array d: [1, 2, 3, 4, 5]
Array e: [1, 2]
```

Al igual que ocurría al emplear el método *clone*, si los tipos no son primitivos se obtendrá una *Shallow copy*.

3. Método *System.arraycopy*

El método *arraycopy* también permite copiar los datos de un *array* a otro de forma total o parcial. A diferencia de los métodos anteriores, para utilizar este método es necesario reservar el espacio en el *array* destino antes de realizar la copia. La sintaxis de uso es la siguiente:

```
arraycopy (Object src, int srcPos, Object dest, int destPos, int length)
```

en donde:

- ✓ *src* hace referencia al *array* origen.

- ✓ *dest* hace referencia al *array* destino.
- ✓ *srcPos* hace referencia al elemento del *array* origen desde el que se comienza la copia.
- ✓ *destPos* hace referencia a la posición en la que se comenzará copia en el destino.
- ✓ *length* hace referencia al número de elementos a copiar desde el *array* origen.

A continuación, se muestra un ejemplo de uso del método *arraycopy*:

```
int[] array1 = {1,2,3,4,5};
int[] array2 = new int[5];
System.arraycopy(array1, 2, array2, 1, 2);
for (int i=0;i<array2.length;i++)
    System.out.println(array2 [i]);
```

```
run:
0
3
4
0
0
```

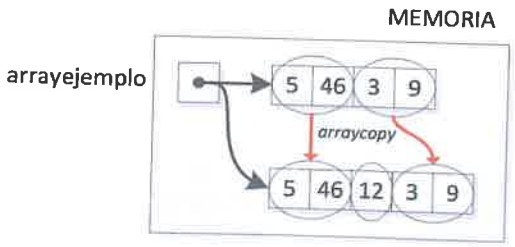
Al igual que ocurría al emplear el método *clone*, si los tipos no son primitivos, se obtendrá una *Shallow copy*.

PARA SABER MÁS

Además de los métodos expuestos en este capítulo existen otros mecanismos para llevar a cabo la copia de un *array* como la API *Stream* o la *Serialización*.

H) Insertar elementos en un array

Como ya se explicó anteriormente, uno de los inconvenientes que presentan los *arrays* estáticos es la rigidez que ofrecen para modificar su tamaño. Es por ello por lo que, si se quiere insertar un nuevo elemento en un *array* estático, será necesario declarar un nuevo *array* de dimensiones mayores al original y sobre este realizar la inserción y copiar los elementos ya existentes, sobrescribiendo finalmente la referencia al *array* original por la referencia al nuevo. Gracias a las utilidades expuestas en el apartado anterior, esta tarea es sumamente sencilla tal como puede comprobarse en el siguiente ejemplo:

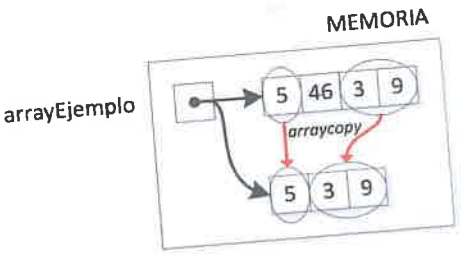


```
public static int[] insertarElemento (int[] array, int elemento, int posicion){
    int [] nuevoArray = new int [array.length+1];
    System.arraycopy(array, 0, nuevoArray, 0, posicion);
    nuevoArray[posicion] = elemento;
    System.arraycopy(array, posicion, nuevoArray, posicion+1, array.length-posicion);
    return nuevoArray;
}
public static void main(String[] args) {
    int[] arrayEjemplo = {5, 46, 3, 9};
    int posicion = 2;
    int nuevoElemento = 12;
    System.out.println("Array original: "+Arrays.toString(arrayEjemplo));
    arrayEjemplo = insertarElemento (arrayEjemplo, nuevoElemento, posicion);
    System.out.println("Array con nuevo elemento: "+Arrays.toString(arrayEjemplo));
}
```

```
run:
Array original: [5, 46, 3, 9]
Array con nuevo elemento: [5, 46, 12, 3, 9]
```

I) Eliminar elementos de un array

La tarea opuesta a la inserción se puede llevar a cabo de forma similar mediante el empleo de las utilidades ya mencionadas. A continuación se muestra un ejemplo de implementación de un programa que contiene un método para eliminar elementos del *array*:




```
public static int[] eliminarElemento (int[] array, int posicion){
    int [] nuevoArray = new int [array.length-1];
    System.arraycopy(array, 0, nuevoArray, 0, posicion);
    System.arraycopy(array, posicion+1, nuevoArray, posicion, nuevoArray.length-posicion);
    return nuevoArray;
}
public static void main(String[] args) {
    int[] arrayEjemplo = {5, 46, 3, 9};
    int posicion = 1;
    System.out.println("Array original: "+Arrays.toString(arrayEjemplo));
    arrayEjemplo = eliminarElemento (arrayEjemplo, posicion);
    System.out.println("Array con nuevo elemento: "+Arrays.toString(arrayEjemplo));
}
```

```
run:
Array original: [5, 46, 3, 9]
Array con nuevo elemento: [5, 3, 9]
```

Actividad propuesta 8.3

Desarrolla un programa que pida por consola el número de alumnos de una clase y que a continuación solicite los *N* nombres para almacenarlos en un *array*. A continuación, implementa los métodos necesarios para eliminar a un alumno del *array* a partir de su nombre, para añadir un alumno nuevo al *array* de alumnos y para ordenar el listado de alumnos.

8.4. Cadenas de caracteres

Las cadenas de caracteres ya fueron introducidas en el capítulo 4. Sin embargo, no se ha profundizado en los métodos de los que este tipo de objetos dispone. Será en este capítulo donde se hará una descripción detallada de cómo utilizar estos objetos.

8.4.1. Declaración, creación e inicialización

Después de ver en detalle qué es un *array*, posiblemente se piense en la posibilidad de manejar cadenas de caracteres como *arrays* formados de elementos tipo *char*. Sin embargo, el lenguaje Java ofrece el tipo *String*, que ha sido especialmente diseñado para manejar cadenas.

Existen varias formas de construir un *String*. A continuación se muestra un ejemplo en el que se construye a partir de una secuencia de caracteres encerrados entre comillas dobles (“”) y a través de un *array* de elementos tipo *char*:

```
String cadena1 = "Hola mundo!";
System.out.println(cadena1);

char[] arrayChar = { 'H','o','l','a',' ','m','u','n','d','o','!'};
String cadena2 = new String(arrayChar);
System.out.println(cadena2);
```

```
run:
Hola mundo!
Hola mundo!
```

SABÍAS QUE...

Una vez que se crea e inicializa un *String*, este es inmutable y no se puede modificar.

8.4.2. Operaciones

Existe un gran repertorio de métodos públicos que permiten llevar a cabo operaciones sobre objetos de tipo *String*. En el cuadro 8.1 se halla un resumen de los más destacados (algunos de ellos están sobrecargados):

CUADRO 8.1
Métodos de *String* más destacados

`char charAt(int posición)`	Retorna el carácter situado en posición.
`int compareTo(String cadena)`	Compara lexicográficamente con la cadena que se pasa como argumento basándose en los valores Unicode de cada carácter. Un *valor negativo* indica que el *String* precede lexicográficamente a la cadena. Un *valor positivo* indicará lo contrario. Un *valor 0* indica que ambos *Strings* son iguales.

[.../...]

CUADRO 8.1 (CONT.)

`int compareToIgnoreCase(String cadena)`	Similar al anterior pero sin tener en cuenta las posibles diferencias entre mayúsculas y minúsculas.
`String concat(String cadena)`	Retorna la concatenación del *String* invocante con la cadena que se pasa como argumento. El resultado es equivalente a emplear el operador '+' con ambas cadenas.
`boolean contains(String s)`	Devuelve *true* si el *String* contiene la cadena pasada como argumento, y *false* en caso contrario.
`boolean endsWith(String sufijo)`	Devuelve *true* si el *String* finaliza con la cadena pasada como argumento, y *false* en caso contrario.
`boolean equals(Object objeto)`	Devuelve *true* si el *String* pasado como argumento es igual, lexicográficamente hablando, al que realiza la invocación.
`boolean equalsIgnoreCase(String cadena)`	Similar al anterior pero sin tener en cuenta mayúsculas o minúsculas.
`int indexOf(String cadena)`	Devuelve un *número entero* que representa la posición de la primera ocurrencia de cadena dentro del *String*. Si no se encuentra devolverá *-1*.
`int indexOf(String cadena, int posición)`	Similar al anterior comenzando la búsqueda en *posición*.
`int lastIndexOf(String cadena)`	Similar pero buscando en orden inverso desde el final.
`int lastIndexOf(String str, int posición)`	Similar al anterior comenzando la búsqueda desde *posición* hacia atrás.
`boolean isEmpty()`	Devuelve *false* en caso de que el *String* no contenga caracteres. Si el objeto es *null*, retornará una excepción.
`int length()`	Devuelve el número de caracteres del *String*.
`boolean matches(String regex)`	Devuelve *true* si el *String* coincide con la expresión regular pasada como argumento.
`String replace(String cadenaBuscada, String reemplazo)`	Sustituye cada ocurrencia de cadena buscada por la cadena de reemplazo.
`String replaceAll(String regex, String reemplazo)`	Similar al anterior pero buscando secuencias que coinciden con una *expresión regular*.
`String replaceFirst(String regex, String reemplazo)`	Similar al anterior pero solo sustituye la primera ocurrencia.
`String[] split(String regex)`	Busca las coincidencias de la *expresión regular* en la cadena y retorna un *array* de las subcadenas comprendidas entre ellas.
`boolean startsWith (String prefijo)`	Devuelve *true* si el *String* comienza con la cadena pasada como argumento.
`String substring(int posComienzo)`	Retorna la subcadena comprendida desde *posComienzo* hasta el final de la cadena.

[.../...]

CUADRO 8.1 (CONT.)

`String substring(int comienzo, int final)`	Retorna la subcadena comprendida entre las posiciones comienzo y final.
`char[] toCharArray()`	Devuelve un *array* cuyos elementos son cada uno de los caracteres contenidos en el *String*.
`String toLowerCase()`	Retorna el mismo *String* con todos los caracteres en minúsculas.
`String toUpperCase()`	Similar al anterior pero convirtiéndolos en mayúsculas.
`String trim()`	Retorna una copia de la cadena en la que se han eliminado los posibles espacios en blanco del comienzo y del final.

En el siguiente ejemplo puede verse el manejo de algunos de los métodos descritos:

```
String cadena = "  Esta es una cadena    ";
System.out.println(cadena.trim());
System.out.println(cadena.toLowerCase());
System.out.println("Caracter en posición 3: "+cadena.charAt(3));

String copiaCadena = cadena;
System.out.println("cadena --> "+cadena);
System.out.println("copiaCadena --> "+copiaCadena);
System.out.println("cadena == copiaCadena --> "+(cadena==copiaCadena));
System.out.println("cadena.equals(copiaCadena) --> "+(cadena.equals(copiaCadena)));

cadena = cadena.replace("cadena", "cadena de caracteres");

System.out.println("cadena --> "+cadena);
System.out.println("copiaCadena --> "+copiaCadena);
System.out.println("cadena == copiaCadena --> "+(cadena==copiaCadena));
System.out.println("cadena.equals(copiaCadena) --> "+(cadena.equals(copiaCadena)));
```

```
run:
Esta es una cadena
  esta es una cadena
Carácter en posición 3: s
cadena -->   Esta es una cadena
copiaCadena -->   Esta es una cadena
cadena == copiaCadena --> true
cadena.equals(copiaCadena) --> true
cadena -->   Esta es una cadena de caracteres
copiaCadena -->   Esta es una cadena
cadena == copiaCadena --> false
cadena.equals(copiaCadena) --> false
```

Actividad propuesta 8.4

Escribe un programa que sea capaz de contar el número de palabras diferentes que hay en un texto que se le pasa por argumento, sin tener en cuenta si están escritas en mayúsculas o minúsculas.

Recursos web

Las expresiones regulares son ampliamente utilizadas en programación para conseguir validar datos y realizar búsquedas de patrones en textos. En la URL https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html podrás obtener una detallada descripción de cómo pueden ser utilizadas en un programa Java. Además, hay disponibles varias herramientas web que ayudan a construir y validar las expresiones regulares (https://www.regexplanet.com/advanced/java/index.html y http://myregexp.com/ son algunas de ellas).

8.4.3. Conversiones

Por medio de las clases envoltorio que fueron expuestas en el capítulo 4, es posible realizar conversiones de dígitos a cadenas de caracteres y viceversa. En el cuadro 8.2 se muestra un resumen de los diferentes mecanismos disponibles para hacer las conversiones.

CUADRO 8.2
Conversiones de dígitos a *String* y viceversa

De *String* al tipo numérico primitivo	De *String* al tipo numérico representado por la clase envoltorio

`Byte.parseByte(cadena);`
`Short.parseShort(cadena);`
`Integer.parseInt(cadena);`
`Long.parseLong(cadena);`
`Float.parseFloat(cadena);`
`Double.parseDouble(cadena);`
	`Byte.valueOf(cadena);`
	`Short.valueOf(cadena);`
	`Integer.valueOf(cadena);`
	`Long.valueOf(cadena);`
	`Float.valueOf(cadena);`
	`Double.valueOf(cadena);`

De tipo numérico a *String*	
variableNumerica+""	*String.valueOf(variableNumerica);*
Método toString() de la clase envoltorio.	

Ejemplo de uso de conversores de tipos numéricos a *String*:

```
int num1 = 1;
Integer num2 = new Integer (2);
System.out.println(String.valueOf(num1));
System.out.println(String.valueOf(num2));
System.out.println(num1+"");
System.out.println(num2+"");
System.out.println(num2.toString());
```

```
run:
1
2
1
2
2
```

Resumen

- Los *arrays* estáticos permiten almacenar elementos de tipos primitivos o clases.
- A diferencia de otros lenguajes, Java no dispone de estructuras y simula este tipo de elementos mediante el uso de clases.
- Existen varios mecanismos para realizar copias de un *array* y cuando un *array* contiene objetos, y no datos de tipo primitivo, las copias son en realidad copias superficiales.
- Un *array* estático no puede modificar su tamaño en tiempo de ejecución.
- Las cadenas de caracteres se representan mediante objetos de la clase *String*, la cual ofrece varios métodos para trabajar con las cadenas.

Ejercicios propuestos

1. Diseña tu propio *Cheatsheet* con los métodos explicados a lo largo del capítulo.
2. Describe la diferencia existente entre utilizar el operador "+" y el método *concat* sobre objetos de tipo *String*.
3. ¿Qué método emplearías para sustituir todas las ocurrencias de correos electrónicos dentro de un texto?
4. Una vez se crea e inicializa un *String*, por ejemplo *String a="cadena";*, este es inmutable y no se puede modificar a lo largo de toda su vida. ¿Qué sucede al realizar una operación de asignación a otra cadena como la siguiente *a="nueva cadena";*? Razona tu respuesta.
5. Propón ejemplos reales en los que puede ser interesante hacer uso de los métodos *replace* y *split* de la clase *String*.
6. Investiga sobre el uso del método *trim* de la clase *String*. Además de eliminar espacios del comienzo y el final de una cadena, ¿eres capaz de eliminar tabulaciones?

Supuestos prácticos

1. Construye un programa que genere 100 números aleatorios mediante el uso de la función *Math.random* y que posteriormente ofrezca al usuario la posibilidad de:
   - Conocer el mayor de los números.
   - Conocer el menor de los números.
   - Obtener la suma de todos los números.
   - Obtener la media de los números.
   - Sustituir el valor de un elemento por otro número introducido por teclado.

2. Implementa un programa que pregunte por un DNI y valida si se trata de un DNI válido. El cálculo de la letra correspondiente a los dígitos se realiza mediante algoritmo descrito en http://www.interior.gob.es/web/servicios-al-ciudadano/dni/calculo-del-digito-de-control-del-nif-nie.
3. Escribe el código de un programa que pida por teclado el contenido de un texto y realiza búsquedas de una determinada palabra dentro de él. La secuencia de ejecución será: pedir texto, pedir palabra, buscar número de ocurrencias de palabra en texto, mostrar por pantalla el número de apariciones de la palabra.
4. Supón que tu empresa se va a hacer cargo del desarrollo de un programa para el registro de datos de los trabajadores de una compañía. Esta tiene varias sedes que están identificadas por el nombre de la ciudad en la que residen (hasta el momento *Madrid, Barcelona, Valencia* y *Oviedo*). En cada sede hay varios departamentos, algunos de los cuales están repetidos entre varias sedes, a los que pertenecen los diferentes empleados. Hasta el momento los departamentos existentes en todas las sedes son *Ventas, RR. HH.* y *Producción*. De cada empleado es importante almacenar el *código de empleado, DNI, nombre, primer apellido, segundo apellido, año de nacimiento (numérico)* y si tiene concedida, o no, *reducción de jornada (booleano)*. Diseña la estructura que almacenará los datos de la compañía y escribe el código fuente del programa que permita comenzar a registrar la información de los empleados.

ACTIVIDADES DE AUTOEVALUACIÓN

1. Si se ejecuta *arrayB = arrayA.clone();* y los datos contenidos son elementos de tipo char:
   - ☐ *a)* La instrucción *arrayA[1] = 'z';* será equivalente a *arrayB[1] = 'z';*
   - ☐ *b)* La instrucción *arrayA[1] = 'z';* no altera el contenido de *arrayB*
   - ☐ *c)* La instrucción *arrayA[1] = 'z';* creará un nuevo objeto tipo Character para ser almacenado en arrayA.
   - ☐ *d)* Todas las opciones son incorrectas.
2. Si se ejecuta *arrayB = arrayA;* y los datos contenidos son elementos de tipo char:
   - ☐ *a)* La instrucción *arrayA[1] = 'z';* será equivalente a *arrayB[1] = 'z';*
   - ☐ *b)* La instrucción *arrayA[1] = 'z';* no altera el contenido de *arrayB*
   - ☐ *c)* *arrayA* y *arrayB* son totalmente independientes.
   - ☐ *d)* Todas las opciones son incorrectas.

3. ¿Cuál de los siguientes métodos requiere de reserva de espacio en memoria previa para hacer la copia?
   - ☐ a) *Arrays.copyOf*
   - ☐ b) *Arrays.copyOfRange*
   - ☐ c) *System.arraycopy*
   - ☐ d) *clone*

4. Para "trocear" una cadena de texto en varias partes que vienen separadas por el carácter "#"dentro de la cadena original, se podrá emplear:
   - ☐ a) El método *split* de la clase *String*.
   - ☐ b) Una combinación de los métodos *indexOf* y *substring* de la clase *String*.
   - ☐ c) Una combinación de los métodos *lastIndexOf* y *substring* de la clase *String*.
   - ☐ d) Todas las opciones son correctas.

5. Identifica la afirmación correcta entre las siguientes:
   - ☐ a) El método *length* de un *array* devuelve el número de elementos en él contenidos.
   - ☐ b) El campo *length* de un *String* devuelve el número de caracteres en él contenidos.
   - ☐ c) A partir de un *array* se puede invocar a su método *toString* para obtener una cadena de texto que contenga los elementos en el array contenidos.
   - ☐ d) Todas las opciones son incorrectas.

6. ¿Cuál de las siguientes instrucciones no realizará una conversión a *String*?
   - ☐ a) *String.valueOf(num1);*
   - ☐ b) *String.parseString(num1);*
   - ☐ c) *num1+""+num2.*
   - ☐ d) *num1.toString()+num2.toString();*

7. ¿Cuál de los siguientes métodos permite obtener un número entero, de tipo primitivo, a partir de una cadena de caracteres?
   - ☐ a) *Integer.valueOf (cadena);*
   - ☐ b) *Integer.value (cadena);*
   - ☐ c) *Integer.parseInt (cadena);*
   - ☐ d) *Integer.parse(cadena);*

8. ¿Qué método busca coincidencias en una cadena de una expresión regular pasada como argumento y retorna un *array* de las subcadenas comprendidas entre ellas?
   - ☐ a) *matches*
   - ☐ b) *substring*
   - ☐ c) *split*
   - ☐ d) *trim*

9. ¿Cuál de las siguientes declaraciones se realiza conforme a la sintaxis del lenguaje C/C++?
   - ☐ a) *char cadena [];*
   - ☐ b) *char [] cadena;*
   - ☐ c) *char []cadena;*
   - ☐ d) *[]char cadena;*

10. ¿Qué valor tomará la variable s tras la ejecución `s = 3+2+"4"+3+2;`?

- ☐ *a)* 545.
- ☐ *b)* 5432.
- ☐ *c)* 32432.
- ☐ *d)* 14.

SOLUCIONES:

1.	a	**b**	c	d	5.	a	b	c	**d**
2.	**a**	b	c	d	6.	a	**b**	c	d
3.	a	b	**c**	d	7.	a	b	**c**	d
4.	a	b	c	**d**	8.	a	b	**c**	d

9.	**a**	b	c	d
10.	a	**b**	c	d