7 Estructuras de almacenamiento

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer las diferencias entre el almacenamiento en memoria y el persistente en disco ya visto.
- ✓ Comprender la importancia y utilidad de las estructuras de almacenamiento en memoria.
- ✓ Resolver problemas utilizando arrays.
- ✓ Comprender y saber aplicar los algoritmos de ordenación.
- ✓ Estudiar los vectores de objetos.

7.1 ARRAYS O VECTORES

Hasta ahora, en todos los ejercicios que se han ido viendo durante los temas anteriores no había una necesidad muy grande de almacenar muchos valores, se utilizaban variables para almacenar el color, la edad, la cantidad, etc. Imaginemos que nos piden realizar un programa que almacene la temperatura de 100 ciudades españolas y luego saque la temperatura media nacional. No parece operativo tener 100 variables en nuestro programa, nada más que escribir los nombres en el código el número de líneas se dispara. ¿Y si nos dicen que se va a aumentar el número de ciudades a 200? La solución a esto se llama arrays o vectores (son sinónimos).

Recuerda
En Java se pueden crear vectores o arrays de tipos básicos (*boolean*, *int*, *byte*, etc.) y también arrays de objetos. De esa manera se pueden almacenar varios valores en cada posición de memoria.

Un array se compone de una serie de posiciones consecutivas en memoria.

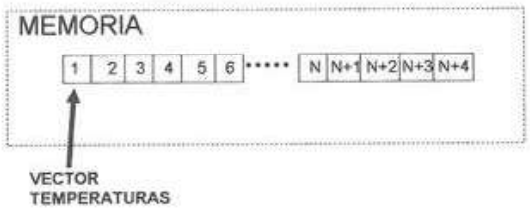

Figura 7.1. Vector temperaturas

A los vectores se accede mediante un subíndice, si por ejemplo nuestro vector anterior se llama temperaturas y se quiere acceder a la posición N, habrá que escribir temperaturas[N] en el programa para obtener la información de esa posición de memoria. N puede ser una variable o bien un valor concreto.

7.1.1 DECLARACIÓN DE VECTORES

En Java se pueden declarar vectores de dos formas diferentes. Para declarar nuestro vector de temperaturas se puede realizar de las siguientes formas:

```
byte[]  temperaturas;
byte temperaturas[];
```

Como puede observarse, en ningún momento se ha dado el tamaño de la matriz, lo único que se ha especificado es el tipo de los elementos que va a albergar dicha matriz.

7.1.2 CREACIÓN DE VECTORES

Java trata los vectores como si fuesen objetos, por lo tanto la creación de nuestro vector *temperaturas* será del siguiente modo:

```
temperaturas = new byte[100];
```

Lo que implica reservar en memoria 100 posiciones de tipo byte.

Recuerda
En los vectores, cuando se reservan N posiciones de memoria, los datos se almacenarán en las posiciones 0, 1, •••••• N-1.

El tamaño también puede asignársele mediante una variable de la siguiente forma:

```
int v=100;
byte[]  temperaturas;
temperaturas = new byte[v];
```

También es muy común ver en los programas este tipo de declaraciones:

```
int v=100;
byte[]  temperaturas = new byte[v];
```

Como se puede ver se funden la segunda y tercera línea del código anterior en una sola.

7.1.3 INICIALIZACIÓN DE VECTORES

Si no se especifica ningún valor, los elementos de un vector se inicializan automáticamente a unos valores predeterminados (variables numéricas a 0, objetos a *null*, *booleanas* a *false* y caracteres a '\u0000').

También es posible inicializarlo con los valores que desee el programador:

```
byte[]  temperaturas={10,11,12,11,10,9,18,19,14,13,15,15};
```

En este código anterior se ha creado un vector de 12 posiciones del tipo byte con los valores especificados.

7.1.4 MÉTODOS DE LOS VECTORES

Como se ha dicho anteriormente, Java maneja los vectores como si fueran objetos, por lo tanto existen una serie de métodos heredados de la clase *Object* que está en el paquete java.lang:

- **equals**. Permite discernir si dos referencias son el mismo objeto.
- **clone**. Duplica un objeto.

Un ejemplo de utilización de estos métodos es el siguiente:

```
byte[]  temperaturas1={10,11,12,11,10,9,18,19,14,13,15,15};
byte[]  temperaturas2=(byte[])temperaturas1.clone();
byte[] temperaturas3=temperaturas1;

if (temperaturas1.equals(temperaturas2)){
            System.out.println("temperaturas1==temperaturas2");
}else{
            System.out.println("temperaturas1!=temperaturas2");
}
if (temperaturas1.equals(temperaturas3)){
            System.out.println("temperaturas1==temperaturas3");
}else{
            System.out.println("temperaturas1!=temperaturas3");
}
```

En el ejemplo anterior, el programa mostrará los siguientes literales "temperaturas1!=temperaturas2" y "temperaturas1==temperaturas3" porque en el primer caso, aunque los datos son los mismos, el objeto es diferente y en el segundo caso, al asignar temperaturas3=temperaturas1 hace que temperaturas3 referencie al mismo objeto (apunta al mismo lugar en la memoria, no se duplican los datos), y en ese caso el método *equals* si da como resultado *true*.

7.1.5 UTILIZACIÓN DE LOS VECTORES

Un ejemplo de utilización de los vectores es el siguiente programa:

```
public class temperaturas {
    private static int[]  temperaturas1;
    final static int POS=10; //número de posiciones del array
    public static void main(String[] args) {
    int dato=0;
    int media=0;
    temperaturas1 = new int[POS];
    for (int i=0;i<POS;i++){ //leer los valores de temperatura
          try{
                System.out.println("Introduzca Temperatura:");
                String sdato = System.console().readLine();
                dato = Integer.parseInt(sdato);
          }catch(Exception e){
                System.out.println("Error en la introducción de datos");
          }
          temperaturas1[i]=dato;
    }
    for (int i=0;i<POS;i++){//hacer la media
```

```
            media = media + temperaturas1[i];
        }
        media = media / POS;
        System.out.println("La media de temperaturas es "+media);
        }
}
```

En el programa anterior se leen las temperaturas de una serie de ciudades por teclado y luego se muestra la media de temperaturas. Como se puede observar, se crea la constante POS, la cual contiene el número de temperaturas a registrar. En el ejemplo está definida con valor 10 pero se puede aumentar o disminuir su valor y el programa funcionará sin modificar más el código.

7.2 ARRAYS MULTIDIMENSIONALES O MATRICES

Tratar con matrices en Java es parecido a tratar con vectores. Por ejemplo, una matriz de enteros de dos dimensiones con 5 filas y 8 columnas se crearía de la siguiente manera:

```
int[][] matriz = new int[5][8];
```

La inicialización del array en el momento de la declaración se hará del siguiente modo:

```
int [][] matriz = {{1,4,5},{6,2,5}};
```

En el código anterior se ha creado un array de 2 filas y tres columnas.

```
System.out.println(matriz.length);
System.out.println(matriz[0].length);
```

El código anterior muestra en la primera línea el número de filas de la matriz creada anteriormente (2) y la segunda línea el número de columnas de la fila 0 de la matriz (3).

	COLUMNAS							
FILAS	M[0][0]	M[0][1]	M[0][2]	M[0][3]	M[0][4]	M[0][5]	M[0][6]	M[0][7]
	M[1][0]	M[1][1]	M[1][2]	M[1][3]	M[1][4]	M[1][5]	M[1][6]	M[1][7]
	M[2][0]	M[2][1]	M[2][2]	M[2][3]	M[2][4]	M[2][5]	M[2][6]	M[2][7]
	M[3][0]	M[3][1]	M[3][2]	M[3][3]	M[3][4]	M[3][5]	M[3][6]	M[3][7]
	M[4][0]	M[4][1]	M[4][2]	M[4][3]	M[4][4]	M[4][5]	M[4][6]	M[4][7]

Figura 7.2. Matriz de datos

El acceso a la matriz se haría igual que cuando se ha trabajado con vectores (matriz[fila][columna]). Imaginemos que queremos almacenar en cada celda de la matriz la suma de la posición de la columna y la fila. El resultado sería el siguiente:

```
for (int i=0;i<5;i++){
      for (int j=0;j<8;j++){
            matriz[i][j]=i+j;
      }
}
```

A FONDO

LOS FICHEROS JAR

De momento los programas o aplicaciones que hemos desarrollado a lo largo de este libro son bastante sencillas, uno o varios ficheros de clases con algún fichero de datos si cabe. No obstante, las aplicaciones más profesionales suelen contener múltiples ficheros, de ahí que para distribuirlas se haga en un fichero JAR. Este formato permite empaquetar múltiples ficheros (clases, datos, sonido, imágenes, etc.) en un solo archivo y tiene muchas ventajas. Dado que están comprimidos, se pueden descargar las aplicaciones de una sola vez, proporciona mecanismos de seguridad y la portabilidad que ofrece al ser un estándar muy común de la plataforma Java.

Las posibilidades de la herramienta son muchas más de las que vamos a ver en esta sección. No obstante, aquí se van a repasar todos los pasos desde crear un JAR de la nada hasta ejecutar una aplicación contenida en un fichero JAR. Obviamente este formato incluye muchas funcionalidades, entre otras la firma electrónica mediante la cual podemos firmar nuestros ficheros JAR y demostrar ante un tercero que el contenido del mismo pertenece a nosotros y no a otra persona que intente suplantarnos.

Las características básicas que vamos a ver son las siguientes:

1. Crear un fichero JAR.
2. Ver el contenido del JAR.
3. Extraer los ficheros de un JAR.
4. Ejecutar la aplicación contenida en un JAR.

Crear un fichero JAR

Para crear el fichero JAR utilizaremos los parámetros *cf* de la herramienta JAR. Para mostrar más información hemos añadido la opción *v*.

```
C:\Archivos de programa\Geany>jar cfv hola.jar holamundoswing.class
manifest agregado
agregando: holamundoswing.class (entrada = 1450) (salida = 882) (desinflado
39%)
```

Ver el contenido del JAR

El contenido del JAR se muestra utilizando los parámetros *tf*.

```
C:\Archivos de programa\Geany>jar tfv hola.jar holamundoswing.class
1450 Tue Jan 04 17:59:10 CET 2011 holamundoswing.class
```

Extraer los ficheros de un JAR

Se pueden extraer todos o alguno de los ficheros contenidos en un JAR. El siguiente comando extrae todos los ficheros del JAR:

```
C:\Archivos de programa\Geany>jar xfv hola.jar
 creado: META-INF/
 inflado: META-INF/MANIFEST.MF
 inflado: holamundoswing.class
```

Y el siguiente comando extraerá solamente el archivo *holamundoswing.class*:

```
C:\Archivos de programa\Geany>jar xfv hola.jar holamundoswing.class
inflado: holamundoswing.class
```

Ejecutar la aplicación contenida en un JAR

Para ejecutar la aplicación contenida en un JAR utilizaremos los parámetros *cp*.

```
C:\Archivos de programa\Geany>java -cp hola.jar holamundoswing
(en algunas ocasiones se utiliza jre -cp en vez de java -cp)
```

La siguiente tabla muestra un resumen de los comandos vistos de la herramienta JAR.

Tabla 7.1. Comando JAR

Comando	Descripción
jar cf fichero_jar fichero/s	Crear un fichero JAR.
jar tf fichero_jar	Ver el contenido de un fichero JAR.
jar xf fichero_jar	Extraer el contenido de un fichero JAR.
jar xf fichero_jar fichero/s	Extraer ficheros de un fichero JAR.
java -cp fichero_jar.jar Clase_main o jre -cp fichero_jar.jar Clase_main	Ejecutar una aplicación empaquetada en un fichero JAR.

¿Qué es el manifest o manifiesto de un JAR?

El manifiesto contiene información sobre los ficheros contenidos en el fichero JAR y es un fichero especial. Este fichero puede adaptarse para utilizar los ficheros JAR para múltiples propósitos.

El manifiesto del JAR creado anteriormente es muy simple, tan solo contendría los siguientes datos:

```
Manifest-Version: 1.0
Created-By: 1.6.0_19 (Sun Microsystems Inc.)
```

Como se puede observar contiene la versión de Java con el que ha sido creado y poco más. En el caso de que utilicemos funcionalidades más avanzadas de la herramienta JAR el contenido de este fichero cambiaría sustancialmente.

7.3 CADENAS DE CARACTERES

Recuerda
Las cadenas de caracteres en Java se tratan como objetos de la clase *String*.

Una cadena de caracteres es un vector o array de elementos de tipo *char*.

```
char[] nombre1={'p','e','p','e'};
char[] nombre2={112,101,112,101};
char[] nombre3=new char[4];
```

En el código anterior las variables nombre1 y nombre2 contienen exactamente lo mismo dado que internamente Java almacena los caracteres con sus símbolos ASCII correspondientes (a la 'p' le corresponde el 112 y a la 'e' el 101). La variable nombre3 se ha creado como una cadena de 4 caracteres pero todavía no se ha inicializado y, por tanto, sus 4 posiciones contendrán el valor '\0'.

Recuerda
Para comprobar que las cadenas de caracteres en Java se tratan como objetos de la clase *String* prueba a hacer lo siguiente:

```
System.out.println("HOLA".length());
```

La anterior línea muestra la longitud del string/cadena de caracteres que en este caso sería 4.

7.3.1 LA CLASE STRING

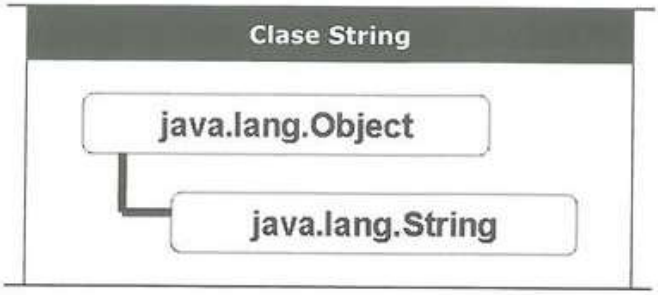

Figura 7.3. Jerarquía de la clase String

La clase *String* pertenece al paquete java.lang y proporciona todo tipo de operaciones con cadenas de caracteres. Esta clase ofrece métodos de conversión a cadena de números, conversión a mayúsculas, minúsculas, reemplazamiento, concatenación, comparación, etc.

Recuerda

Aparte de todos los siguientes métodos, el propio lenguaje java ofrece el operador de concatenación +. Un ejemplo de utilización es:

```
System.out.println("Longitud de la cadena HOLA: "+"HOLA".length());
```

- **String(String dato)**. Constructor de la clase *String*

```
String cad1 = "Pepe";
String cad2 = new String("Lionel");
String cad3 = new String(cad2);
```

Las tres líneas de código anterior crean objetos de la clase *String*. Nótese como el objeto *cad3* está creado a partir del objeto *cad2* y contendrá los mismos datos "Lionel".

- **int length()**. Muestra la longitud de un objeto de la clase *String*.

```
String cad1 = "CHELO";
System.out.println(cad1.length());
```

El código anterior muestra la longitud del objeto *string cad1* (5).

- **String concat(String s)**. Devuelve un objeto fruto de la concatenación/unión de un objeto *String* con otro.

```
String cad1 = "Andy";
cad1=cad1.concat(" Rosique");
System.out.println(cad1);
```

El código anterior concatena las cadenas "Andy" y "Rosique", y muestra por pantalla el resultado de la concatenación ("Andy Rosique").

- **String toString()**. Devuelve el propio *String*.

```
String cad1 = "Emilio";
String cad2 = " Anaya";
System.out.println(cad1.toString()+cad2.toString());
```

El código anterior aprovecha el operador concatenación "+" para mostrar por pantalla la cadena "Emilio Anaya".

- **int compareTo(String s)**. Compara el objeto *String* con el objeto *String* pasado como parámetro y devuelve un número:
  - **< 0** Si es **menor** el *string* desde el que se hace la llamada al *String* pasado como parámetro.
  - **= 0** Si es **igual** el *string* desde el que se hace la llamada al *String* pasado como parámetro.
  - **> 0** Si es **mayor** el *string* desde el que se hace la llamada al *String* pasado como parámetro.

El método va comparando letra a letra ambos *String* y si encuentra que una letra u otra es mayor o menor que otra deja de comparar.

```
String cad1 = "EMMA";
String cad2 = "MARIA";
System.out.println(cad1.compareTo("emma"));
System.out.println(cad1.compareTo("EMMA"));
System.out.println(cad1.compareTo("EMMA MORENO"));
System.out.println(cad2.compareTo("MARIA AMPARO"));
System.out.println(cad2.compareTo("MAREA"));
```

El anterior código mostrará por pantalla los siguientes datos: -32, 0, -7, -7 y 4.

¡Cuidado!
El método *compareTo* distingue mayúsculas de minúsculas. Las mayúsculas están antes por orden alfabético que las minúsculas, por lo tanto 'A' es menor que 'a'.

- **boolean equals()**. Este método sirve para comparar el contenido de dos objetos del tipo *String*.

```
String cad1="EMMA";
String cad2=new String("EMMA");
if (cad1.equals(cad2)){
  System.out.println("SON IGUALES");
}else{
  System.out.println("SON DIFERENTES");
};
```

El código anterior mostrará por pantalla "SON IGUALES".

A FONDO

DIFERENCIA ENTRE EL MÉTODO EQUALS Y EL OPERADOR ==

```
String cad1="EMMA";
String cad2=new String("EMMA");
if (cad1.equals(cad2)){
System.out.println("SON IGUALES");
}else{
System.out.println("SON DIFERENTES");
};
if (cad1==cad2){
System.out.println("SON IGUALES");
}else{
System.out.println("SON DIFERENTES");
};
```

El código anterior mostrará por pantalla las siguientes cadenas: "SON IGUALES", "SON DIFERENTES". En el primer caso los dos objetos son iguales porque contienen la misma secuencia de caracteres y el segundo de los casos son diferentes porque son **diferentes objetos**. Para que en el segundo caso muestre la cadena "SON IGUALES" debería de cambiarse la línea de código:

```
String cad2=new String("EMMA");
```

Por la siguiente otra:

```
String cad2=cad1;
```

En este último caso es importante recalcar que *cad2* y *cad1* apuntan al mismo objeto.

- **String trim()**. Elimina los espacios en blanco que contenga el objeto *String* al principio y final del mismo.

```
String cad1 = "    MAYKA    ",cad2 = cad1.toUpperCase();
System.out.println(cad2.toString());
```

El código anterior mostrará por pantalla la cadena "MAYKA".

- **String toLowerCase()**. Convierte las letras mayúsculas del objeto *String* en minúsculas.

```
String cad1 = "PEDRO ruiz",cad2 = cad1.toLowerCase();
System.out.println(cad2.toString());
```

El código anterior mostrará por pantalla la cadena "pedro ruiz".

- **String toUpperCase()**. Convierte las letras minúsculas del objeto *String* en mayúsculas.

```
String cad1 = "JUAN serrano",cad2 = cad1.toUpperCase();
System.out.println(cad2.toString());
```

El código anterior mostrará por pantalla la cadena "JUAN SERRANO".

- **String replace(char** car, **char** newcar). Reemplaza cada ocurrencia del carácter car por el carácter *newcar*.

```
String cad1 = "JUAN SUAREZ",cad2 = cad1.replace('U','O');
System.out.println(cad2.toString());
```

El código anterior mostrará por pantalla la cadena "JOAN SOAREZ".

- **String substring(int** i, **int** f). Este método devuelve un nuevo objeto *String* que será la subcadena que comienza en el carácter i y termina en el carácter *f* (el carácter *f* no se muestra). Si no se especifica el segundo parámetro devolverá hasta el final de la cadena.

```
String cad1 = "JUAN CARLOS MORENO";
System.out.println(cad1.substring(5,11));
System.out.println(cad1.substring(12));
```

El código anterior mostrará por pantalla las cadenas "CARLOS" y "MORENO".

- **boolean startsWith(String** cad). Este método devuelve *true* si el objeto *String* comienza con la cadena *cad*, en caso contrario devuelve *false*.

```
String cad1 = "MAYKA MORENO";
System.out.println(cad1.startsWith("JUAN"));
System.out.println(cad1.startsWith("MAY"));
```

El código anterior mostrará por pantalla *false* y *true*.

- **boolean endsWith(String** cad). Este método devuelve *true* si el objeto *String* termina con la cadena *cad*, en caso contrario devuelve *false*.

```
String cad1 = "MARIA AMPARO";
System.out.println(cad1.endsWith("paro"));
System.out.println(cad1.endsWith("PARO"));
System.out.println(cad1.endsWith("ARIA"));
```

El código anterior mostrará por pantalla *false*, *true* y *false*. La primera vez muestra *false* porque aunque 'p' y 'P' son la misma letra, Java las trata de manera diferente al ser dos símbolos ASCII distintos.

- **char charAt(int** pos). Devuelve el carácter del objeto *String* que se especifica en el parámetro *pos*.

```
String cad1 = "AMPARO HEREDIA";
System.out.println(cad1.charAt(0)+" "+cad1.charAt(7));
```

El código anterior mostrará por pantalla la cadena "A H".

¡Cuidado!
Si en la función *charAt* se utiliza un índice que no está entre los valores 0 y length()-1, Java lanzará una excepción.

- **int indexOf(int c)** o **int indexOf(String s)**. Este método admite dos tipos de parámetros y nos permite encontrar la primera ocurrencia de un carácter o una subcadena dentro de un objeto del tipo *String*. En el caso de que no sea encontrado el carácter o la subcadena este método devolverá el valor -1.

```
String cad1 = "EMMA MORENO";
System.out.println(cad1.indexOf('M'));
System.out.println(cad1.indexOf('J'));
System.out.println(cad1.indexOf("MO"));
System.out.println(cad1.indexOf("MI"));
```

El código anterior mostrara por pantalla el siguiente resultado: 1, -1, 5 y -1.

- **char[] toCharArray()**. Este método devuelve un vector o array de caracteres a partir del propio objeto *String*.

```
String cad1 = "LORO FELIPE";
char cad2[]=cad1.toCharArray();
```

El código anterior creará un array de caracteres *cad2* que contendrá la cadena "LORO FELIPE" contenida en el objeto *String cad1*.

- **String valueOf(int** dato). Convierte un número a un objeto *String*. La clase *String* es capaz de convertir los tipos primitivos *int*, *long*, *float* y *double*.

```
int edad1=6;
String str=String.valueOf(edad1);
float edad2=6;
str=String.valueOf(edad2);
long edad3=6;
str=String.valueOf(edad3);
double edad4=6.5;
str=String.valueOf(edad4);
```

A FONDO

CONVERTIR UN STRING EN UN NÚMERO

```
String snumero="  6  ";
int numero=Integer.parseInt(snumero.trim());
//int numero=Integer.parseInt(snumero);
```

Con el código anterior es posible convertir un objeto String en un número entero. La tercera línea de código está comentada porque lanzaría una excepción dado que no se han limpiado los espacios a derecha e izquierda de la cadena y contiene caracteres no numéricos.

Si el número es un decimal y no se quieren perder los decimales se utilizaría el siguiente código:

```
String snumero="  6.5  ";
double numero=Double.valueOf(snumero).doubleValue();
```

7.3.2 LA CLASE STRINGBUFFER

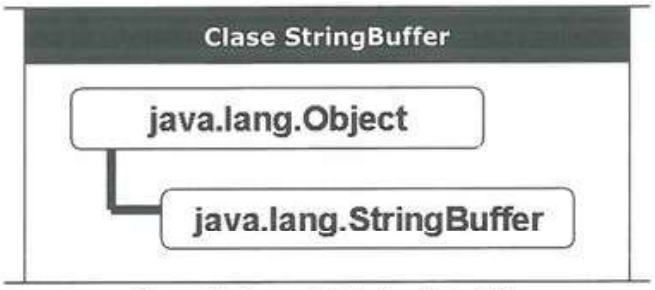

Figura 7.4. Jerarquía de la clase StringBuffer

Recuerda
Los objetos de la clase *String NO* son modificables sino que los métodos que actúan sobre los objetos devuelven un objeto nuevo con las modificaciones realizadas. En cambio, los objetos StringBuffer SÍ son modificables.

- **StringBuffer([arg])**. Constructor de la clase *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("Pepe");
StringBuffer apellidos = new StringBuffer(80);
StringBuffer direccion = new StringBuffer();
```

La segunda línea de código crea un objeto vacío con una capacidad para 80 caracteres. En la tercera línea de código no se especifica la capacidad pero por defecto la deja a 16.

- **int length()**. Muestra la longitud del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("Pepe");
System.out.println(nombre.length());
```

La salida por pantalla del código anterior será 4.

- **int capacity()**. Muestra la capacidad del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("Pepe");
System.out.println(nombre.capacity());
```

Curioso: La salida por pantalla del código anterior será 20 aunque su longitud era solo de 4. Eso es debido a que cuando se crea un objeto Java le otorga una capacidad igual al número de caracteres almacenados más 16.

- **StringBuffer append(argumento)**. Añade el argumento al final de la cadena de caracteres *StringBufffer*. El tipo del argumento puede ser *int*, *long*, *float*, *double*, *boolean*, *char*, *char*[ ], *String* y *Object*.

```
StringBuffer nombre = new StringBuffer("Juan Carlos");
String apellidos=new String(" Moreno Pérez");
nombre.append(apellidos);
System.out.println(nombre);
```

El código anterior muestra por pantalla la cadena "Juan Carlos Moreno Pérez".

- **StringBuffer insert(int pos, arg)**. Añade el argumento en la posición pos de la cadena de caracteres StringBufffer. El tipo del argumento puede ser *int*, *long*, *float*, *double*, *boolean*, *char*, *char*[], *String* y *Object*.

```
StringBuffer nombre = new StringBuffer("EMMA");
String apellidos=new String(" MORENO");
nombre.insert(nombre.length(),apellidos);
System.out.println(nombre);
```

El código anterior muestra por pantalla la cadena "EMMA MORENO".

- **StringBuffer reverse()**. Invierte la cadena de caracteres que contiene.

```
StringBuffer nombre = new StringBuffer("TURRION");
nombre.reverse();
System.out.println(nombre);
```

El código anterior mostrara por pantalla la cadena "NOIRRUT".

- **StringBuffer delete(int x, int y)**. Elimina los caracteres entre las posiciones *x* e *y* del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("RAUL JESUS TURRION");
nombre = nombre.delete(4,10);
System.out.println(nombre);
```

El código anterior mostrará por pantalla la cadena de caracteres "RAUL TURRION"

- **StringBuffer replace(int x, int y, String s)**. Reemplaza los caracteres entre las posiciones *x* e *y* por el *String s* del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("RAUL JESUS");
nombre = nombre.replace(5,10,"TURRION");
System.out.println(nombre);
```

El código anterior mostrará por pantalla la cadena de caracteres "RAUL TURRION".

- **String substring(int x, int y)**. Devuelve un *String* que contiene la cadena que comienza en el carácter *x* hasta el carácter *y-1* (o hasta el final si no se especifica el argumento *y*).

```
StringBuffer nombre = new StringBuffer("RAUL JESUS TURRION");
String turri = nombre.substring(0,4)+nombre.substring(10);
System.out.println(turri);
```

El código anterior mostrará por pantalla la cadena de caracteres "RAUL TURRION".

- **String toString()**. Devuelve un objeto *String* el cual es una copia del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("TURRION");
String turri = nombre.toString();
System.out.println(turri);
```

El código anterior crea un objeto *String* a partir de un objeto *StringBuffer* y muestra su contenido por pantalla.

- **char charAt(int x)**. Devuelve el carácter que está en la posición *x* del objeto *StringBuffer*.

```
StringBuffer nombre = new StringBuffer("EMMA");
System.out.println(nombre.charAt(0));
```

El código anterior mostrará por pantalla el carácter 'E'.

- **void setCharAt(int x, char c)**. Reemplaza el carácter que está en la posición *x* del objeto *StringBuffer* por el carácter *c*.

```
StringBuffer nombre = new StringBuffer("EMMA");
nombre.setCharAt(0,'e');
System.out.println(nombre.toString());
```

El código anterior mostrará por pantalla la cadena de caracteres "eMMA".

A FONDO

CLASE STRINGTOKENIZER

Esta clase permite dividir una cadena de caracteres en elementos independientes si estos están separados por un espacio en blanco, un retorno de carro (\r) o de línea (\n), un avance de página (\f) o un tabulador (\t).

Un ejemplo de utilización de esta clase es el siguiente:

```
StringTokenizer str;
str = new StringTokenizer("UNO DOS TRES PERICO JUANICO Y_ANDRES");
System.out.println("La cadena str tiene "+str.countTokens()+" elementos");
while (str.hasMoreTokens()) System.out.println(str.nextToken());
```

El código anterior mostrará que la cadena str tiene 6 elementos y los irá sacando por pantalla uno a uno.

Para utilizar esta clase se debe de importar la clase *StringTokenizer*:

```
import java.util.StringTokenizer;
```

O importar todas las clases del paquete java.util:

```
import java.util.*;
```

También es posible especificar los delimitadores dentro del constructor de la clase:

```
str = new StringTokenizer("UNO|DOS|TRES PERICO|JUANICO|Y_ANDRES","|");
```

En el ejemplo anterior se utiliza el símbolo '|' como delimitador.

7.4 ARRAYS O VECTORES DE OBJETOS STRING

Dado que ya se conoce cómo funcionan los vectores y los objetos *String*, en este apartado se va a ver un ejemplo de utilización de estos tipos de datos. En el ejemplo se va a realizar un programa que lee nombres por teclado y los va almacenando en un vector. Una vez creado el vector se mostrarán dichos nombres por teclado según la posición en que se han leído.

```
public class test {
    private static String[]  lista;
    final static int POS=10; //número de posiciones del array
    public static void muestra(){
```

```
        for (int i=0;i<POS;i++)    System.out.print(lista[i]+" ");
    }
    public static void main(String[] args) {
        lista = new String[POS];
        for (int i=0;i<POS;i++){
            String ln = System.console().readLine();
            lista[i]=ln.toString();
        }
        System.out.println("");
        muestra();
        System.out.println("");
    }
}
```

7.5 ALGORITMOS DE ORDENACIÓN

Los algoritmos de ordenación se aplican generalmente en arrays unidimensionales (también en ficheros) y su finalidad es organizar los datos en dichos arrays. Estos algoritmos de ordenación tienen gran importancia, con lo cual en muchos lenguajes de programación existen librerías que contienen funciones o procedimientos para ordenar arrays. No todos los algoritmos ordenan de la misma forma ni todos son igual de eficientes, algunos son muy rápidos cuando el vector está casi ordenado, otros lo son cuando está muy desordenado y a otros el preordenamiento no les afecta en gran medida. Dependiendo de la estrategia, algunos algoritmos son mejores que otros. Los algoritmos se distinguen por su complejidad computacional la cual clasifica los algoritmos dependiendo de su eficiencia, en esta clasificación se tiene en cuenta el peor caso, el caso promedio y el mejor de los casos.

Tabla 7.2. Algoritmos de ordenación

Algoritmo	Estrategia de ordenación
Burbuja o bubblesort	Este ordenamiento pega una serie de pasadas al vector y compara parejas de elementos adyacentes y los intercambia si no están ordenados. En la primera pasada obviamente el mayor elemento se situará el último y así sucesivamente. Cuando en una pasada no hace ningún intercambio quiere decir que el vector ya está ordenado.
Cocktail sort	El un ordenamiento por burbuja bidireccional.

Ordenación por inserción o insertion sort	La ordenación se realiza insertando los datos ordenadamente en un vector. Los datos se van insertando agrupados y cuando se inserta un nuevo dato se le hace hueco en la posición que le corresponde desplazando los demás elementos.
Por Mezcla o Merge Sort	Es un algoritmo desarrollado por Von Newmann que emplea la técnica divide y vencerás. El algoritmo va dividiendo por la mitad de forma recursiva el vector a ordenar en dos listas las cuales deberán de ser también ordenadas. El algoritmo para de dividir cuando se queda con 0 ó 1 elementos, entonces se consideran que están ya ordenados. Si se ordenan dos listas las cuales a su vez están ordenadas, el resultado será una lista ordenada.
Ordenamiento por selección o selection sort	El funcionamiento es simple, se busca el menor elemento de la lista y se coloca en el primer lugar, luego se busca el segundo y se coloca en el segundo lugar y así sucesivamente.
Ordenación rápida o Quicksort	Es el algoritmo más rápido. Utiliza la técnica divide y vencerás. Es un algoritmo recursivo que utiliza un pivote como elementos central y coloca todos los elementos menores a su izquierda y los mayores a su derecha. La lista se separa en dos sublistas y se repite el proceso de manera recursiva con las dos sublistas hasta que toda la lista se queda ordenada.

7.5.1 ORDENACIÓN POR EL MÉTODO DE LA BURBUJA

El ordenamiento por el método de la burbuja es muy utilizado, sobre todo porque es sencillo de entender e implementar. En la siguiente tabla se puede comprender claramente cómo funciona el método de la burbuja. Como se puede apreciar, al igual que las burbujas de aire en el agua suben a la superficie, los elementos mayores de la lista se irán colocando al final del vector de forma ordenada.

1ª Pasada

4	2	9	3	1
2	4	9	3	1
2	4	3	9	1
2	4	3	1	9

2ª Pasada

2	4	3	1	9
2	3	4	1	9
2	3	1	4	9

3ª Pasada

2	3	1	4	9
2	1	3	4	9

4ª Pasada

2	1	3	4	9
1	2	3	4	9

Como parece lógico, las pasadas cada vez serán más cortas dado que en cada pasada, al menos 1 elemento quedará ordenado al final de la lista. Una implementación de este código en Java sería el siguiente:

```
public static void burbuja(int array[]){
       int aux;
       for (int i=array.length;i>0;i--){
              for (int j=0;j<i-1;j++){
                     if (array[j]>array[j+1]){
                            aux = array[j+1];
                            array[j+1]=array[j];
                            array[j]=aux;
                     }
              }
       }
}
```

Este código no es del todo eficiente porque si el vector ya está ordenado el procedimiento seguirá comprobando hasta terminar sin tener en cuenta este hecho. Para conseguir esto, basta con utilizar un *flag* que marque si el vector está ordenado o no y en el caso de que el algoritmo de una pasada sin realizar ningún intercambio de elementos el proceso parará.

A FONDO

BÚSQUEDA DE DATOS DENTRO DE UN ARRAY

Las ventajas de los arrays o vectores son la versatilidad y posibilidades que ofrecen. Una de las operaciones más frecuentes que se realizan en los mismos es la búsqueda de información dentro de ellos. La forma más fácil de buscar información en un array es realizar una búsqueda secuencial hasta encontrar el dato, pero no es la más eficiente. Imaginemos un array muy grande. De media tendremos que buscar en N/2 elementos para encontrar nuestro dato. Si el array es muy grande perderemos mucho tiempo durante la búsqueda. Un método más eficiente es la **búsqueda binaria.** En la búsqueda binaria se sigue el lema "divide y vencerás". En la siguiente imagen se puede observar cómo funciona el algoritmo:

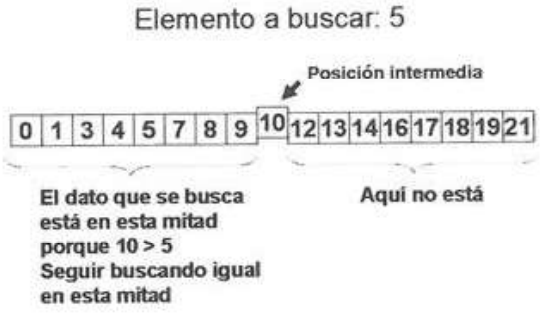

Figura 7.5. Búsqueda binaria

Importante: Los datos del array donde se va a realizar la búsqueda deberán de estar ordenados.

Como se puede ver, lo que se hace es elegir una posición intermedia y desechar de la búsqueda la mitad del array donde el dato no va a estar siguiendo la búsqueda en el lado restante.

La busqueda seguirá igual pero para la mitad del array donde debería de estar el elemento a buscar.

7.5.2 ORDENACIÓN POR EL MÉTODO DE INSERCIÓN DIRECTA

Este algoritmo funciona de la siguiente manera, se desean insertar en un array de 5 elementos los siguientes datos (4, 2, 9, 3 y1).

1ª Inserción (4)

4				

2ª Inserción (2)

4				
	4			
2	4			

3ª Inserción (9)

2	4			
2	4	9		

4ª Inserción (3)

2	4	9		
2		4	9	
2	3	4	9	

5ª Inserción (1)

2	3	4	9	
	2	3	4	9
1	2	3	4	9

A FONDO

CLASE HASHTABLE

Esta clase se encuentra en el paquete java.util. Una *Hashtable* implementa una tabla *hash* la cual crea una especie de diccionario que relaciona claves con valores.

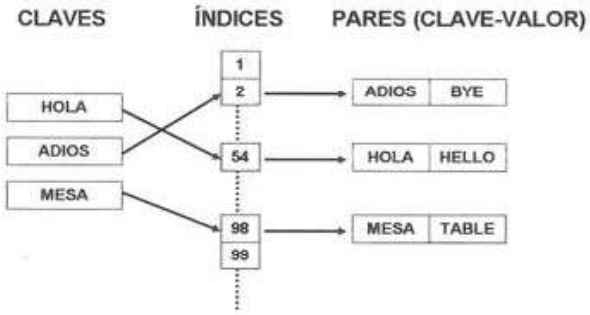

Figura 7.6. La clase Hashtable

En el siguiente ejemplo comentado se muestra cómo se crea y se trabaja con una *hashtable* como la de la figura anterior:

```
import java.util.*;
public class diccionario {
    public static void main(String[] args) {
    Hashtable dic  = new Hashtable();
    dic.put("HOLA","HELLO");
    dic.put("ADIOS", "BYE");
    dic.put("MESA","TABLE");
    dic.put("SILLA","CHAIR");
    dic.put("CABEZA","HEAD");
    dic.put("CARA","FACE");
    String saludo  = (String) dic.get("HOLA");
    String despedida  = (String) dic.get("ADIOS");
    String brazo = (String) dic.get("BRAZO");
    System.out.println("HOLA  : " + saludo);//muestra HELLO por pantalla
    System.out.println("ADIOS : " + despedida);//muestra BYE por pantalla
    System.out.println("BRAZO : " + brazo); //muestra null por pantalla
    System.out.println("dic contiene " + dic.size() + " pares.");
    if( dic.containsKey("HOLA") ){
    System.out.println("dic contiene HOLA como clave");
    }else{
    System.out.println("dic NO contiene HOLA como clave");
    }
    if( dic.contains("HELLO") ){
    System.out.println("dic contiene HELLO como valor");
    }else{
    System.out.println("dic NO contiene HELLO como valor");
    }
    System.out.println("Mostrando todos los datos de la tabla hash...");
    Enumeration k = dic.keys();
    while( k.hasMoreElements() ) System.out.println( k.nextElement() );
    System.out.println("Mostrando todos los elementos de la tabla hash...");
```

```
        Enumeration e = dic.elements();
        while( e.hasMoreElements() ) System.out.println( e.nextElement() );
        System.out.println( "Eliminando el dato " + dic.remove("HOLA"));
        }
}
```


RESUMEN DEL CAPÍTULO

En este tema se estudia el concepto de vector o array. Existen numerosos problemas que se resuelven con estructuras de este tipo, por lo tanto es de obligado cumplimiento estudiarlo. También se estudia en profundidad las cadenas de caracteres que ya se vieron en capítulos anteriores. Estudiar arrays y no ver algoritmos de ordenación es un pecado, por lo tanto, se estudiarán dos de los más útiles como son el método de la burbuja, el cual es sencillo de comprender y el método de inserción directa. Existen multitud de algoritmos mucho más eficientes y rápidos que el de la burbuja. Se aconseja al alumno que busque alguno por Internet e intente comprenderlo.

EJERCICIOS RESUELTOS

1. Realiza un programa que genere una matriz 5 x 8 y muestre los elementos en forma de matriz.

```
public class matriz {
    public static void main(String[] args) {
    int[][] matriz = new int[5][8];
    for (int i=0;i<5;i++){
          for (int j=0;j<8;j++){
                matriz[i][j]=i+j;
          }
    }
    for (int i=0;i<5;i++){
          for (int j=0;j<8;j++){
                System.out.print(matriz[i][j]);
```

```
                System.out.print(" ");
            }
                System.out.println("");
        }
        }
}
```

2. Realiza un programa que cree un vector de 50 posiciones cargado con valores aleatorios. Los valores aleatorios deberán de estar entre el 1 y el 100. Una vez cargado el vector deberá de ordenarlo mediante el método de la burbuja y mostrarlo ordenado por pantalla.

Truco
Observa que para generar un número aleatorio se utiliza el método random de la clase Math el cual genera un número aleatorio (double) entre 0.0 y 1.0. Si este se múltiplica por LIMITE que es el rango de números a generar y se le suma 1, los números generados estarán siempre entre 1 y LIMITE.

```
public class burbuja {
    private static int[]  lista;
    final static int POS=50; //número de posiciones del array
    final static int LIMITE=100; //Números entre 1..Límite
    public static int getaleatorio(){
          return (int) (Math.random()*LIMITE+1);
    }
    public static void ordena(int array[]){
          int aux;
          for (int i=array.length;i>0;i--){
                for (int j=0;j<i-1;j++){
                      if (array[j]>array[j+1]){
                            aux = array[j+1];
                            array[j+1]=array[j];
                            array[j]=aux;
                      }
                }
          }
    }
    public static void muestra(){
          for (int i=0;i<POS;i++){
                System.out.print(lista[i]+" ");
          }
    }
    public static void main(String[] args) {
```

```
            lista = new int[POS];
            for (int i=0;i<POS;i++){
                  lista[i]=getaleatorio();
            }
            muestra();//se muestra el vector desordenado
            System.out.println("");
            ordena(lista); //ordenación por burbuja
            System.out.println("");
            muestra();//se muestra el vector ordenado
            System.out.println("");
    }
}
```

3. Realiza un programa que cree un vector de 50 posiciones cargado con valores aleatorios. Los valores aleatorios deberán de estar entre el 1 y el 100. Una vez cargado el vector deberá de ordenarlo mediante el método del cocktail sort y mostrarlo ordenado por pantalla.

```
public class cocktail {
    private static int[]  lista;
    final static int POS=50; //número de posiciones del array
    final static int LIMITE=100; //Números entre 1..Límite
    public static int getaleatorio(){
          return (int) (Math.random()*LIMITE+1);
    }
    public static void ordenacocktail(int array[]){
      int i = 0, j = array.length - 1;
      while(i < j) {
         for(int k = i; k < j; k++) { //direccion ->
            if(array[k] > array[k + 1]) {
               int temp = array[k];
               array[k] = array[k + 1];
               array[k + 1] = temp;
            }
         }
         j--;
            for(int k = j; k > i; k--) { //direccion <-
               if(array[k] < array[k - 1]) {
                     int temp = array[k];
                     array[k] = array[k - 1];
                     array[k - 1] = temp;
               }
            }
         i++;
      }
    }
    public static void muestra(){
          for (int i=0;i<POS;i++){
```

```
                System.out.print(lista[i]+" ");
            }
        }
        public static void main(String[] args) {
            lista = new int[POS];
            for (int i=0;i<POS;i++){
                lista[i]=getaleatorio();
            }
            muestra();
            System.out.println("");
            ordenacocktail(lista);
            System.out.println("");
            muestra();
            System.out.println("");
        }
}
```

4. Mejora el método ordenacocktail del ejercicio anterior y utiliza una variable swp como centinela o *flag*, de tal manera que ésta se active cuando hay algún intercambio. En el momento que no haya ningún intercambio el algoritmo debería de parar puesto que el vector ya estaría ordenado.

```
public static void ordenacocktail(int array[]){
    boolean swp = true;
    int i = 0, j = array.length - 1;
    while(i < j && swp) {
        swp = false;
        for(int k = i; k < j; k++) { //direccion ->
            if(array[k] > array[k + 1]) {
                int temp = array[k];
                array[k] = array[k + 1];
                array[k + 1] = temp;
                swp = true;
            }
        }
        j--;
        if(swp) {
            swp = false;
            for(int k = j; k > i; k--) { //direccion <-
                if(array[k] < array[k - 1]) {
                    int temp = array[k];
                    array[k] = array[k - 1];
                    array[k - 1] = temp;
                    swp = true;
                }
            }
        }
        i++;
```

```
        }
}
```

5. Tenemos una cadena notas con los nombres y las notas de 5 de los alumnos de clase. El contenido de la cadena es el siguiente:

"Juan Carlos\n 8.5\n Andrés\n 4.9\n Pedro\n 3.8\n Juan \n 6.3"

El formato es "nombre \n nota \n...."

Realiza un programa que muestre por pantalla por cada alumno lo siguiente:

El alumno X ha sacado la nota Y.

```
StringTokenizer notas;
notas = new StringTokenizer("Juan Carlos\n8.5\nAndrés\n4.9\n Pedro\n3.8\nJuan\
n6.3","\n");
while (notas.hasMoreTokens())
    System.out.println("El alumno "+notas.nextToken()+" ha sacado un "+notas.
nextToken());
```

6. Modifica el ejemplo del apartado 6.4.2 para que muestre los nombres ordenados.

```
public class test {
    private static String[]  lista;
    final static int POS=10; //número de posiciones del array
    public static void ordena(String array[]){
          String aux = new String();
          for (int i=array.length;i>0;i--){
                for (int j=0;j<i-1;j++){
                      if (array[j].compareTo(array[j+1])>0){
                            aux = array[j+1];
                            array[j+1]=array[j];
                            array[j]=aux;
                      }
                }
          }
    }
    public static void muestra(){
          for (int i=0;i<POS;i++)    System.out.print(lista[i]+" ");
    }
    public static void main(String[] args) {
          lista = new String[POS];
          for (int i=0;i<POS;i++){
                String ln = System.console().readLine();
                lista[i]=ln.toString();
          }
```

```
            muestra(); //los muestra desordenados
            System.out.println("");
            ordena(lista); //ordena los nombres
            System.out.println("");
            muestra(); //ahora los muestra ordenados
            System.out.println("");
    }
}
```

7. Realiza un programa que muestre por pantalla los dos primeros argumentos tomados desde la línea de comandos.

Solución:

```
public class muestraArgs {
    public static void main(String[] args) {
            System.out.println("Primer argumento: " + args[0]);
            System.out.println("Segundo argumento: " + args[1]);
    }
}
```

8. Realiza un método *esCapicua* que tome como parámetro un entero y devuelva *true* si el número es capicua y *false* en caso contrario. Utiliza *wrappers* y objetos *String*.

Solución:

```
    public static boolean esCapicua(int dato){
            Integer i = new Integer(dato);
            String reverse = new StringBuffer(i.toString()).reverse().toString();
            return i.toString().equals(reverse.toString());
    }
```

9. Crea una clase *busquedabin* la cual tenga un método que busque un valor en un array ordenado utilizando la búsqueda binaria. Implementa el método de tal manera que muestre las posiciones (min y max) desde las cuales va buscando y los valores de dichas posiciones así como la posición intermedia y el valor de dicha posición.

```
public class busquedabin {
    private static int[]  lista;
    final static int POS=50; //número de posiciones del array
    final static int LIMITE=100; //Números entre 1..Límite
    public static int getaleatorio(){
            return (int) (Math.random()*LIMITE+1);
    }
```

```
    public static void ordena(int array[]){
        int aux;
        for (int i=array.length;i>0;i--){
            for (int j=0;j<i-1;j++){
                if (array[j]>array[j+1]){
                    aux = array[j+1];
                    array[j+1]=array[j];
                    array[j]=aux;
                }
            }
        }
    }
    public static void muestra(){
        for (int i=0;i<POS;i++){
            System.out.print(lista[i]+" ");
        }
    }
    public static int buscabin(int[] a, int valor, int min, int max){
        if (min == max) {
            System.out.println("SALIDA PORQUE MIN=MAX");
            return -1;
        }
        int mitad = (min + max)/2;
        System.out.println("min"+min+" a[min]"+a[min]+" max"+max+" a[max]"+a[max]+"
mitad"+mitad+" "+a[mitad]);
        if (valor == a[mitad]) return mitad;
        if (valor == a[min]) return min;
        if (valor == a[max]) return max;
        if (valor > a[mitad])
            return buscabin(a,valor,mitad+1,max);
        else
            return buscabin(a,valor,min,mitad-1);
    }
    public static void main(String[] args) {
        lista = new int[POS];
        for (int i=0;i<POS;i++){
            lista[i]=getaleatorio();
        }
        muestra();
        System.out.println("");
        ordena(lista);
        System.out.println("");
        muestra();
        System.out.println("");
        System.out.println(buscabin(lista,50,0,POS-1));
    }
}
```

El ejemplo anterior, como se puede observar genera y ordena la lista antes de hacer la búsqueda binaria.

- Realiza las siguientes modificaciones al ejercicio:
  - Modifica el ejercicio anterior para que no se puedan insertar valores repetidos en el array.
  - Modifica el ejercicio para que el algoritmo en vez de recursivo sea iterativo.

EJERCICIOS PROPUESTOS

- **1.** El ejercicio resuelto número 2, como podrás suponer, no es óptimo del todo. Imagínate que los números están ya ordenados (cosa muy improbable utilizando números aleatorios). Modifica el ejercicio anterior para que el procedimiento ordene y no dé más pasadas de las necesarias.

- **2.** Realiza un programa que cree dos vectores de 100 elementos. El primero almacenará una serie de datos numéricos desordenados. Dichos datos serán datos generados aleatoriamente. El segundo array contendrá los mismos datos pero ordenados por el método *insertion sort*.

- **3.** Modifica el código del ejercicio resuelto 5 para que las notas se almacenen en un vector de datos *double*.

- **4.** (Ejercicio de dificultad alta) Realiza un programa con el que puedas jugar a los barquitos con tu compañero. El programa simulará el juego clásico.

- **5.** (Ejercicio de dificultad alta) Programa que realice una sopa de letras. La sopa de letras tendrá un tamaño de matriz 15 x 15. El programa pedirá 10 palabras, las cuales las irá escondiendo de forma aleatoria por la matriz (obviamente las palabras siempre tendrán 15 letras o menos). Una vez escondidas las palabras rellenará las demás casillas de la matriz con letras de forma aleatoria. Solo se utilizarán mayúsculas. Si el usuario introduce palabras en minúsculas se transformarán a mayúsculas.

- **6.** Tenemos el siguiente método que indica cuándo un número es capicúa o no:

```
public static boolean esCapicua(int dato){
       Integer i = new Integer(dato);
       String reverse = new StringBuffer(i.toString()).reverse().toString();
       return i.toString()==reverse.toString();
}
```

Parece que no funciona porque todos los números que se introducen para comparar el método responde que no son capicúa. Descubre por qué no funciona y razona tu respuesta.

- **7.** Realiza un método que tome como parámetros de entrada dos arrays de enteros y devuelva como salida un único array con los elementos de los anteriores arrays ordenados de forma ascendente.

- **8.** Realiza un programa que cree 1000 números aleatorios y muestre los 10 mayores.

- **9.** Realiza un programa que cree un vector de 100 posiciones con números aleatorios entre 10 y 80. Una vez creado el vector el programa deberá mostrar el mayor, el menor, el valor que más se repite y la media.

- **10.** Realiza un programa que cree un vector de 100 posiciones con números aleatorios entre 1 y 100. Una vez creado el vector el programa deberá ordenar el vector y mostrar los números entre 1 y 100 que no han sido almacenados.