DAM/DAW

PROGRAMACIÓN

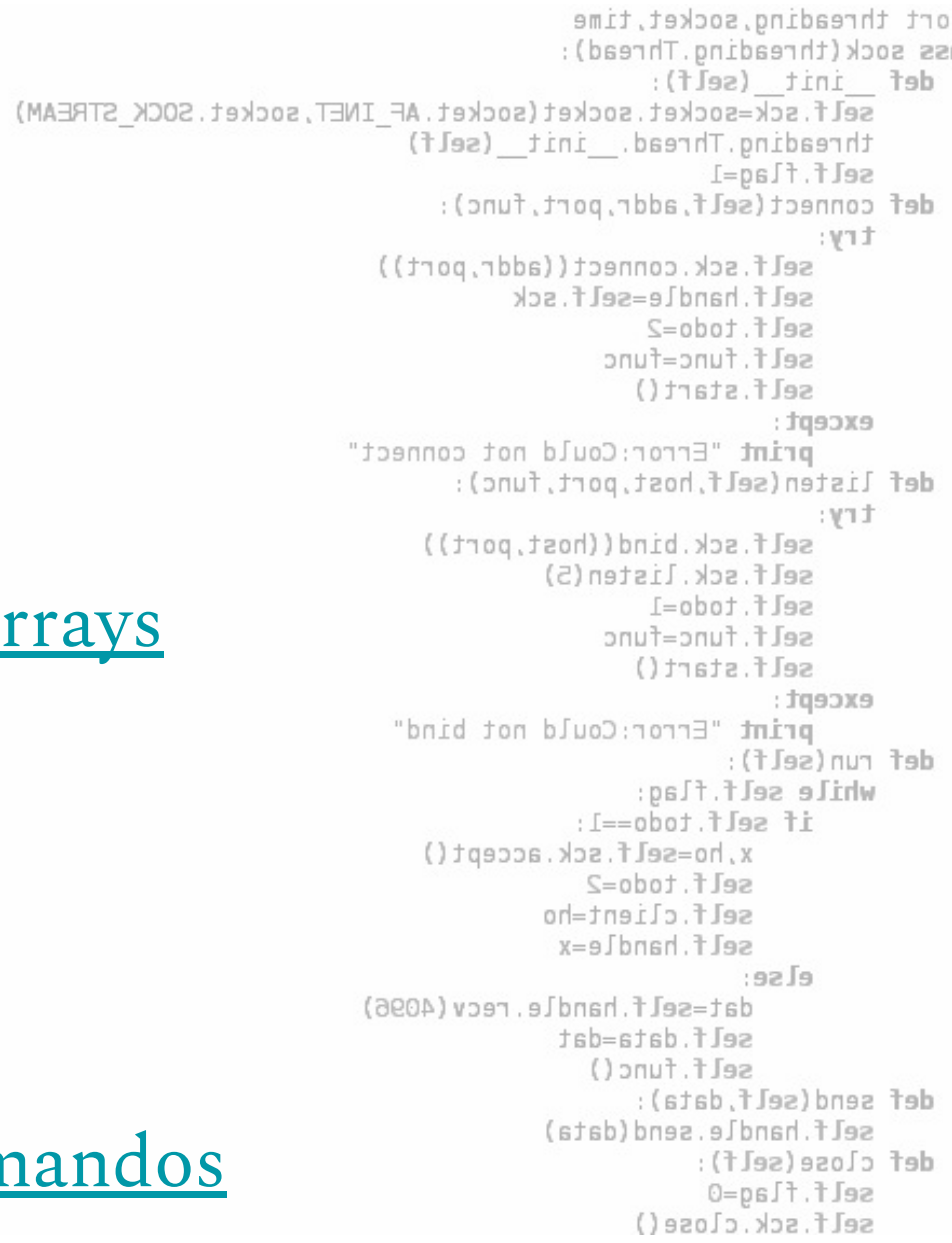
05.1

Arrays en Java

Rev: 3.1

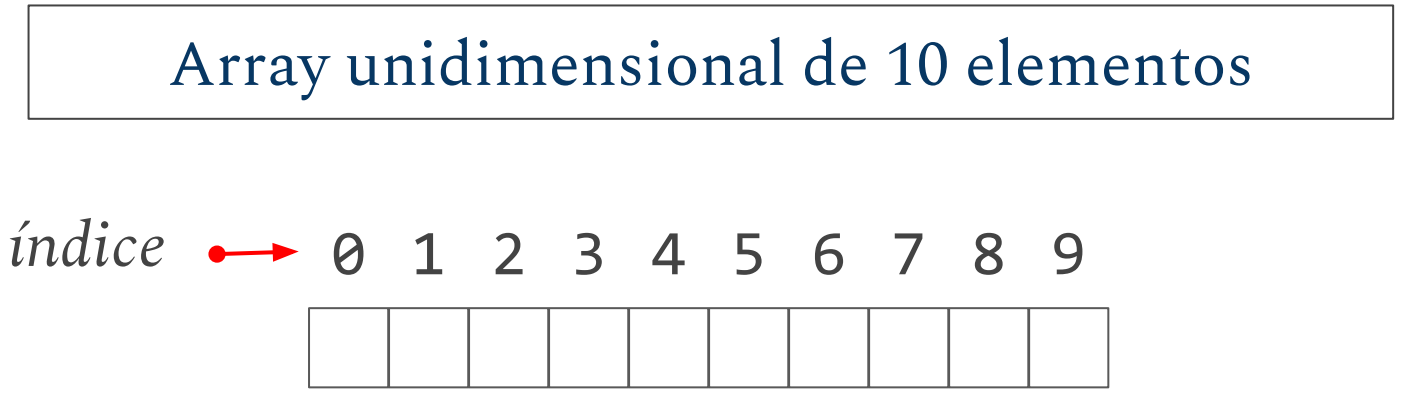
fran@iessanclemente.net

Indice



Introducción (I)

- En computación, un *array*, vector o arreglo, es una estructura de datos consistente en una colección de *elementos* (valores o variables), cada uno de ellos identificado por un *índice* o *clave*
- El *array* comprende una zona de almacenamiento contiguo para alojar una serie de elementos del mismo tipo. Desde el punto de vista lógico, se puede ver como un conjunto de elementos ordenados en fila (filas y columnas si tuviera dos dimensiones).



Introducción (II)

- El array podrá definir cualquier número de dimensiones:
  - unidimensionales (*tuplas, vectores,...*):
    - Ej: lista de los nombres de los alumnos de un grupo:

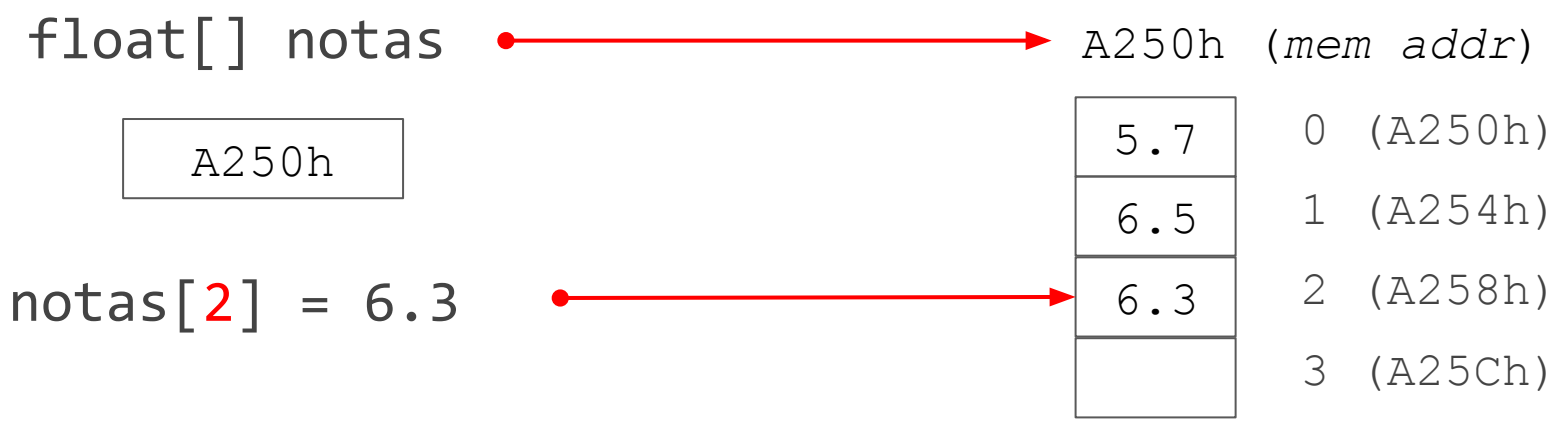
      `alumnos[num_alumnos]`
  - bidimensionales (*matrices*):
    - Ej: lista de notas de los alumnos de un grupo:

      `notas[num_alumnos][num_evaluaciones]`
  - tres o más dimensiones:
    - Ej: datos en bruto de una imagen digital:

      `raw_data[ancho_pixels][alto_pixels][componentes_RGBA]`
- El producto de las dimensiones del array determinará la capacidad (en elementos) del mismo

Introducción (III)

- Cada elemento del *array* es referenciado por la posición que ocupa dentro del vector o índice.
- Este tipo de indexación se denomina directa, pues nos permite acceder al elemento deseado a partir de su índice, sin necesidad de recorrer los elementos precedentes (p.e., *lista enlazada*). El índice, que es 0 para el primer elemento (indexado base-0), se utiliza como *desplazamiento* desde la dirección de memoria de comienzo del *array*




```
notas[2] = valor en [A250h + 2*size(float)] = [A258h] = 6.3
```

Introducción (y IV)

- Los *arrays* en Java nos permiten almacenar tanto valores de tipos primitivos (*int*, *char*,...) como referencias a objetos
- Una particularidad de como implementa Java los *arrays* es que los trata como objetos. La principal ventaja de ello es que el recolector de basura se encargará de liberar automáticamente la memoria ocupada por el *array*
- Los *arrays* en Java son estructuras estáticas, es decir, no podemos modificar su tamaño. Al crear un array, la JVM se reservará en el *heap* el espacio de memoria necesario para almacenar el número de elementos deseados. Ese tamaño ya no podrá ser modificado
- Los *arrays* son un elemento estructural fundamental de la computación desde sus inicios. John von Neumann escribió el primer programa de ordenación de arrays (*merge-sort*) para el EDVAC en 1945. Se emplean desde el almacenamiento de registros de bases de datos, hasta la implementación de otras estructuras como colas, pilas, tablas *hash*,...

Arrays unidimensionales (I)

❖ Declaración y creación

- La declaración y creación de un *array* en Java es como la de cualquier otro objeto:
  - En la declaración del *array* indicamos el tipo de los datos (primitivo o referenciado) que se almacenarán en el mismo. Deberemos añadir unos corchetes vacíos [ ] al final del nombre de la variable (o del tipo) para indicarle a Java que estamos declarando un *array*

```
tipo[] nombre;
```

  - Ejemplos:
    - `int[] ref_producto;`
    - `float max_temp_diaria[];`
    - `String[] nombres;`
    - `Clientes clientes[];`

Arrays unidimensionales (II)

- Una vez declarado, podremos crear el array
  - Utilizaremos el operador *new*, indicando la capacidad (de elementos del tipo especificado) del nuevo *array*.

```
nombre = new tipo [capacidad];
```

  - La JVM se encargará de reservar en el *heap* el espacio de memoria necesario dependiendo del tipo de dato de los elementos:
    - tipo primitivo:
      - espacio (bytes) = capacidad x tamaño_tipo (bytes)
    - tipo referenciado (*depende de la implementación de la JVM*):
      - JVM 32b: espacio (bytes) = capacidad x 4 (bytes)
      - JVM 64b: espacio (bytes) = capacidad x 8 (bytes)
  - Ejemplo:
    - `max_temp_diaria = new float[365];  // size = 365*4(bytes)`

Arrays unidimensionales (III)

- Al igual que con el resto de objetos en Java, podremos combinar declaración y creación en una única sentencia
  - Ejemplos:
    - `int[] ref_producto = new int[10000];`
    - `float max_temp_diaria = new float[365];`
    - `String[] nombres = new String[12];`
    - `Clientes clientes[] = new Clientes[125];`
- Es importante que seamos conscientes de que cuando creamos un *array* de tipos referenciados (en el ejemplo, *nombres* y *clientes*), simplemente estamos reservando espacio para almacenar las referencias a los objetos del tipo indicado. No se crean los objetos en sí. Por ejemplo,

  `String[] nombres = new String[12];`

  reserva espacio para almacenar 12 referencias a objetos de tipo String, pero no crea dichos objetos String

Arrays unidimensionales (IV)

❖ Inicialización

- Una vez creado el *array*, cada uno de sus elementos se inicializará a un valor que dependerá del tipo de datos del *array*:
  - 0, para tipos primitivos numéricos
  - false, para el tipo *boolean*
  - null, para tipos referenciados

❖ Lectura y Escritura

- El acceso a cualquiera de los elementos del *array*, tanto para leer como para modificar su valor, lo realizaremos empleando el identificador del *array* y, entre corchetes, el índice del elemento en cuestión. Por ejemplo:
  - `max_temp_diaria[12] = 23.12;`
  - `System.out.println(nombres[2]);`
  - `clientes[0] = new Cliente("John Doe", "jd007@gmail.com")`

Arrays unidimensionales (V)

- Ejemplo:

```
//: ArrayDemo.java
/**
* Cuadrados de los 100 primeros números en 10 columnas
*/
class ArrayDemo {
  public static void main(String[] args) {
    int[] sq = new int[100];       // creación array
    for(int i=0; i<100; i++)      // bucle inicialización
      sq[i] = i*i;
    for(int i=0; i<100; i++)      // bucle impresión
      System.out.printf("%04d%c", sq[i], ((i+1)%10==0)?'\n':'\t');
  }
}
```

Arrays unidimensionales (VI)

❖ Inicialización en la creación

- Java nos permite inicializar nuestro array en el momento de la declaración (y creación). Para ello, deberemos suministrar los valores de cada uno de sus elementos:

```
tipo[] nombre = { val1, val2, …, valN };
```

- Fíjate que no indicamos el tamaño del *array* ni usamos el operador *new*. Java ya se encarga de reservar el espacio necesario para alojar los valores indicados e inicializar cada una de las posiciones del *array*
  - Ejemplos:
    - `String[] dias_lab = { "Lunes", "Martes", "Jueves" };`
    - `System.out.println(dias_lab[1]);  // → Martes`
    - `dias_lab[2] = "Viernes";`
    - `dias_lab[4] = "Sábado";  // → Error`

Arrays unidimensionales (y VI)

❖ Tamaño (propiedad *length*)

- En Java, al estar implementados los *arrays* como objetos, tendremos acceso a una variable de instancia denominada *length* que nos indicará el número de elementos que puede contener el array (capacidad)

```
int[] list = new int[10];
System.out.println(list.length); // → 10
```

- Al acceder a un elemento de un *array* deberemos tener especial cuidado de que el índice se encuentre dentro del rango permitido, que viene determinado por el intervalo: [0, length), es decir, desde 0 hasta (length-1)
- En caso de que el valor del índice no se encuentre en dicho intervalo, se producirá una excepción *ArrayIndexOutOfBoundsException* en tiempo de ejecución. Esta excepción provocaría la finalización abrupta del programa (salvo que sea capturada por el propio programa)

El bucle *for-each* (I)

- Al trabajar con *arrays* es frecuente encontrarnos en situaciones donde debemos examinar cada uno de los elementos del *array*, desde el principio hasta el final. Por ejemplo: buscar un valor, copiar el *array*, imprimirlo,...
- Este tipo de operaciones son tan comunes que Java, a partir del JDK5, definió una segunda variante de la sentencia *for*, de tipo "*for-each*", más cómoda para estos casos
- Un bucle *for-each* itera de forma secuencial a través de una colección de objetos como, por ejemplo, un *array*. En cada una de estas iteraciones irá extrayendo, uno a uno, los elementos de la colección, desde el primero hasta el último
- Por tanto, a diferencia de con un *for* tradicional, no podremos especificar los valores de índice inicial, final, o un paso determinado. Sí que podremos usar las sentencias *break* y *continue* para alterar su ejecución

El bucle *for-each* (y II)

- Su formato general es:

```
for (tipo var: collection) {
    bloque de acciones;
}
```

donde,

  - *tipo*, indica el tipo de los datos almacenados en la colección
  - *var*, almacena el elemento de la colección extraído en cada iteración
  - *colection*, colección de elementos que recorrerá el bucle

```
int nums[] = { 2, 4, 7, 1, 5 }, sum = 0;
double media;
for(int n: nums)     // bucle for-each
    sum += n;        // n ← cada elemento de la colección nums
media = (double)sum/nums.length;
System.out.printf("La suma es: %d y media es: %.2f%n", sum, media);
```

Asignando *referencias* de Arrays (I)

- Como ya sabemos, los *arrays* son objetos y, sus variables, contienen únicamente referencias a dichos objetos en memoria
- Por tanto, debemos tener las mismas precauciones que con cualquier otro objeto a la hora de asignar una variable que referencia un *array* a otra, o cada vez que es usada como argumento de un método. No estamos creando una copia del *array*. Simplemente, copiamos su referencia
- Esto supone que, si dos variables están "apuntando" al mismo *array*, cualquier modificación en el mismo a través de una de dichas variables, afectará a la otra (pues es el mismo array)
- De igual modo, cualquier modificación hecha en el interior de un método a un *array* recibido como argumento, persistirá al retornar de dicho método

Asignando *referencias* de Arrays (y II)

- Ejemplo:

```
class ArrayRef {
  public static void main(String[] args) {
    int[] ar1 = {1, 2, 3, 4, 5};
    int[] ar2 = ar1;
    initArray(ar2, -1);
    ar2[2] = 99;
    for(int n: ar1)      // bucle impresión
      System.out.print(n + " ");
  }
  private static void initArray(int[] a, int n) {
    for(int i=0; i<a.length; i++) a[i] = n;  // bucle inicialización
  }
}
/* output:
-1 -1 99 -1 -1
*///:~
```

Arrays multidimensionales (I)

- Los arrays multidimensionales nos permitirán crear estructuras de almacenamiento con más de una dimensión. Un ejemplo típico sería el de una tabla, caracterizada por sus dos dimensiones: filas y columnas
- Cada uno de los elementos del *array* multidimensional será referenciado por un valor único para cada una de sus dimensiones. Volviendo al ejemplo anterior de la tabla, cada una de sus celdas está identificada por los valores correspondientes a su fila y columna.

	COLUMNA - 0	COLUMNA - 1	COLUMNA - 2
FILA - 0	3.5	128.12	-34.2
FILA - 1	4.0	110.19	-12.23

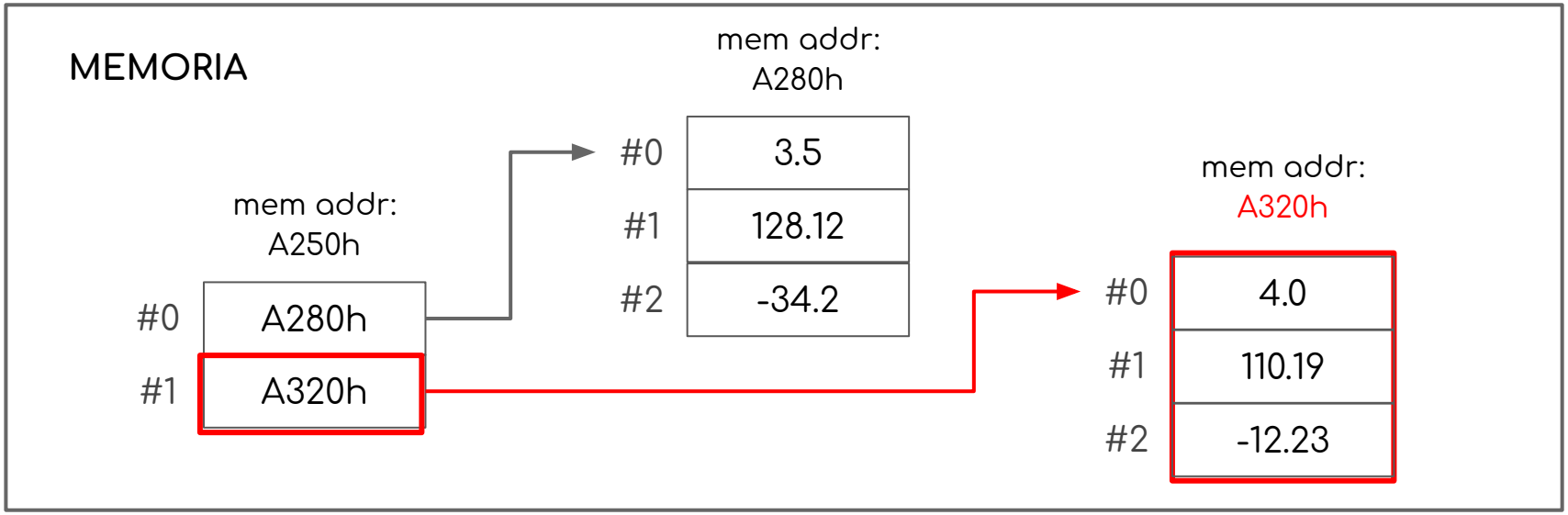
Valor correspondiente a la celda en la posición:

Fila: 1, Columna: 2

Arrays multidimensionales (II)

- Para Java, un array multidimensional no es más que un de *array* de *arrays*
  Así, la tabla del ejemplo anterior básicamente sería una *array* de dos elementos (uno por fila), donde cada uno de dichos elementos es, a su vez, un *array* de 3 valores numéricos (uno por cada columna)

	COLUMNA - 0	COLUMNA - 1	COLUMNA - 2
FILA - 0	3.5	128.12	-34.2
FILA - 1	4.0	110.19	-12.23



Arrays multidimensionales (III)

❖ Declaración y creación

- Para declarar *arrays* de dos o más dimensiones aplicaremos lo aprendido para arrays de una dimensión. Simplemente añadiremos una nueva pareja de corchetes ([ ]) por cada nueva dimensión y el número de elementos

```
tipo[][]...[] nombre = new tipo [cap1][cap2]...[capN];
```

- Volviendo al ejemplo de nuestra tabla de dos filas y tres columnas, podríamos crear un *array* para almacenar sus valores así:

```
double[][] tabla = new double [2][3];
```

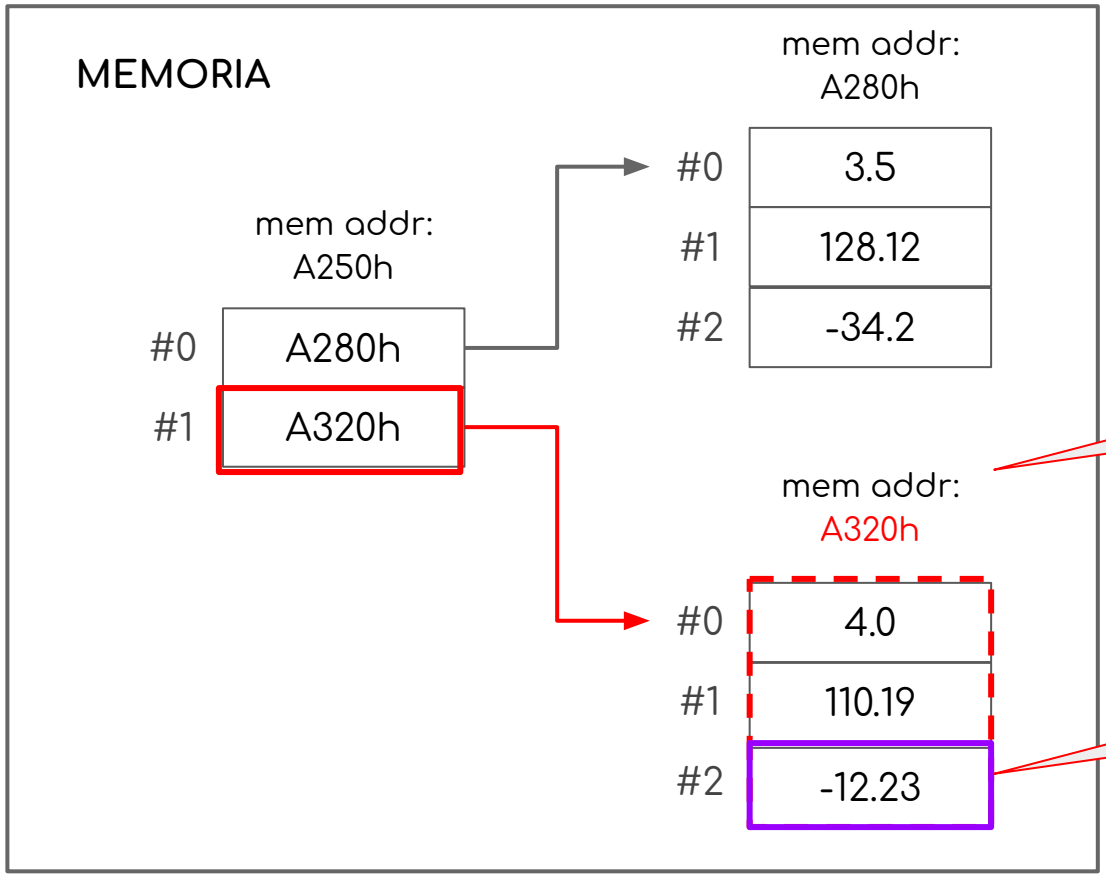
donde la primera dimensión representa la fila y, la segunda, la columna

- Así, para acceder a cualquiera de sus elementos (para lectura o modificación), deberemos proporcionar el índice correspondiente por cada una de las dimensiones (empezando en cero)

Arrays multidimensionales (IV)

	COLUMNA - 0	COLUMNA - 1	COLUMNA - 2
FILA - 0	3.5	128.12	-34.2
FILA - 1	4.0	110.19	-12.23

¿Podríamos intercambiar filas y columnas?




```
double[][] tabla = new double[2][3];
```

tabla[1] sería el *array* de valores correspondiente a la fila de índice 1: [4.0, 110.19, -12.23]

tabla[1][2] sería el valor correspondiente a la celda de la fila 1 y columna 2

Arrays multidimensionales (V)

❖ Inicialización

- Tal como hacíamos con los *arrays* de una dimensión, podemos inicializar el *array* en el momento de su creación. Usaremos llaves ({ }) para agrupar los miembros de cada una de las diferentes dimensiones que, a su vez, irán separados por comas.

	COLUMNA - 0	COLUMNA - 1	COLUMNA - 2
FILA - 0	3.5	128.12	-34.2
FILA - 1	4.0	110.19	-12.23

```
double[][] tabla = {
    { 3.5, 128.12, -34.2 },
    { 4.0, 110.19, -12.23 }
};
```

Arrays multidimensionales (VI)

❖ Arrays irregulares

- Cuando asignamos espacio de almacenamiento para un *array* multidimensional, sólo estamos obligados a especificar inicialmente el valor de la primera de sus dimensiones. El tamaño de las siguientes dimensiones puede ser asignado *a posteriori*. Por ejemplo:

	COLUMNA - 0	COLUMNA - 1	COLUMNA - 2
FILA - 0	3.5	128.12	-34.2
FILA - 1	4.0	110.19	-12.23

```
double[][] tabla = new double [2][];
tabla[0] = new double[3];      // array de 3 elementos inicializados a 0
tabla[1] = { 4.0, 110.19, -12.23 };   // array de 3 elementos inicializados
                                      // a los valores suministrados
```

Arrays multidimensionales (VI)

- El hecho de que, para Java, los *arrays* multidimensionales se tratan como *arrays* de *arrays* (se almacenan sus referencias), la longitud de esos *arrays* puede ser distinta (*arrays* irregulares)
- Imagina que se quiere crear una tabla para almacenar los tiempos de paso de los puntos de control de un rally, sabiendo que cada una de sus etapas tiene diferente longitud y, por tanto, más o menos puntos de control:

```
double[][] tiempos = new double[4][];   // el rally tiene 4 etapas
tiempos[0] = new double[1];   // etapa 1 (un punto de control)
tiempos[1] = new double[3];   // etapa 2 (tres puntos de control)
tiempos[2] = new double[4];   // etapa 3 (cuatro puntos de control)
tiempos[3] = new double[4];   // etapa 4 (cuatro puntos de control)
```

Esta estructura sólo nos permitiría almacenar los tiempos de uno de los pilotos participantes.
Suponiendo que hay 20 pilotos,
¿cómo la podríamos modificar para almacenar los tiempos de control de todos ellos?
¿cómo accederíamos al tiempo en la meta en la 3ª etapa del 7º piloto de la lista?

Arrays multidimensionales (y VII)

❖ Recorrido

- Para recorrer los elementos de *arrays* multidimensionales, normalmente emplearemos bucles (*for*, *for-each*,...) anidados, uno por cada dimensión.
- Debemos tener presente que, salvo la última de las dimensiones, todas las anteriores referencian *arrays*. Además, como en el caso de los irregulares, pueden presentar longitudes diferentes.

```
for(int i=0; i<tiempos.length; i++) { // tiempos.length = 1ª dimensión
    System.out.print("\netapa " + i + ": ");
    for(int j=0; j<tiempos[i].length; j++) // tiempos[i].length = 2ª dimensión
        System.out.print(tiempos[i][j] + " ");
}
```

```
int i = 0;
for(double[] etapa: tiempos) {  // etapa = array de tiempos de cada etapa
    System.out.print("\netapa " + (++i) + ": ");
    for(double t: etapa)        // t = tiempo
        System.out.print(t + " ");
}
```

La clase Arrays (I)

- El API de Java incluye la clase java.util.Arrays que proporciona una serie de métodos estáticos sobrecargados que facilitan la realización de tareas habituales con *arrays*. Algunos de ellos son:
  - static String toString(*tipo*[ ] a)
    imprime los elementos del array *a* con el formato:
    [*elemento_0*, *elemento_1*, ..., *elemento_N*]
  - static void fill(*tipo*[ ] a, *tipo* valor)
    Inicializa los elementos del array *a* con *valor*
  - static boolean equals(*tipo*[ ] a, *tipo*[ ] b)
    Compara uno a uno los elementos de los arrays *a* y *b*
  - static void sort(*tipo*[ ] a)
    Ordena el contenido del array *a* en orden ascendente

La clase Arrays (y II)

- static void sort(*tipo[ ]* a, int *desde*, int *hasta*)
  Ordena el rango especificado del array *a* en orden ascendente
- static int binarySearch(*tipo[ ]* a, *tipo* valor)
  Devuelve la posición de *valor* en el array *a* (que supone ordenado).
  Devuelve un valor negativo en caso contrario
- static int binarySearch(*tipo[ ]* a, int *desde*, int *hasta*, *tipo* valor)
  Realiza la búsqueda en el rango especificado
- static *tipo[ ]* copyOf(*tipo[ ]* a, int len)
  Devuelve un nuevo *array* de *len* elementos inicializados con los valores del array *a* (ó 0 si *len* es mayor que *a.length*)
- static *tipo[ ]* copyOfRange(*tipo[ ]* a, int *desde*, int *hasta*)
  Devuelve un nuevo *array* inicializado con los valores del rango especificado del array *a*

Argumentos de línea de comandos (I)

- Una aplicación Java puede aceptar un número variable de argumentos desde la línea de comandos.
- Esto permite al desarrollador diseñar aplicaciones parametrizables y configurables cuyo comportamiento pueda ser "ajustado" en el momento de su ejecución. Será el usuario quien especificará la información de configuración necesaria en el momento en que se lance la aplicación
- Estos argumentos se especifican a continuación del nombre de la clase que vamos a ejecutar. Por ejemplo, supongamos que una aplicación Java denominada *Sort* ordena las líneas de un fichero. Para ordenar el contenido de un fichero denominado *amigos.txt*, ejecutaríamos:

```
java Sort amigos.txt
```

- Al lanzar la aplicación, la JVM le pasará los argumentos al método *main* de la aplicación como un *array* de *Strings*

Argumentos de línea de comandos (y II)

❖ Ejemplo: mostrando los argumentos

```
public class Echo {
    public static void main (String[] args) {
        for (String s: args) {
            System.out.println(s);
        }
    }
}
```

Salida:

```
java Echo Drink Hot Java
Drink
Hot
Java
```

- Fíjate como Java emplea el espacio para caracter de separación de los diferentes argumentos