UD7.- Estructuras de Datos I

Módulo: Programación
1.º DAM

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

CONTENIDOS

- Tipo de datos
- Strings
- Arrays

Tipo de datos en Java

- Tipo de datos primitivos
  - Numéricos
    - Enteros: byte, int, short, long
    - Reales: float, double
  - Carácter: char
  - Lógicos: boolean

- Tipo de datos referencia
  - String, Array, Class (objetos), Interface

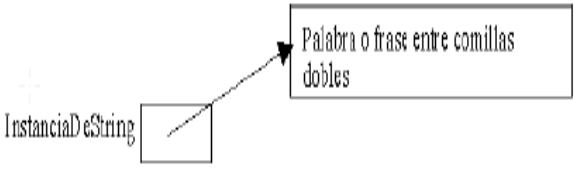
Tipos primitivos *versus* tipos referencia

- **Variables de tipos primitivos**
  - Contienen directamente los datos.
  - Cada variable tiene su propia copia de los datos, de forma que las operaciones en una variable de un tipo primitivo no pueden afectar a otra variable.

- **Variables de tipos referencia**
  - Contienen una referencia o puntero al valor del objeto, no el propio valor.
  - Dos variables de tipos referencia pueden referirse al mismo objeto, de forma que las operaciones en una variable de tipo referencia pueden afectar al objeto referenciado por un otra variable de tipo referencia.

Tipo de datos *String*

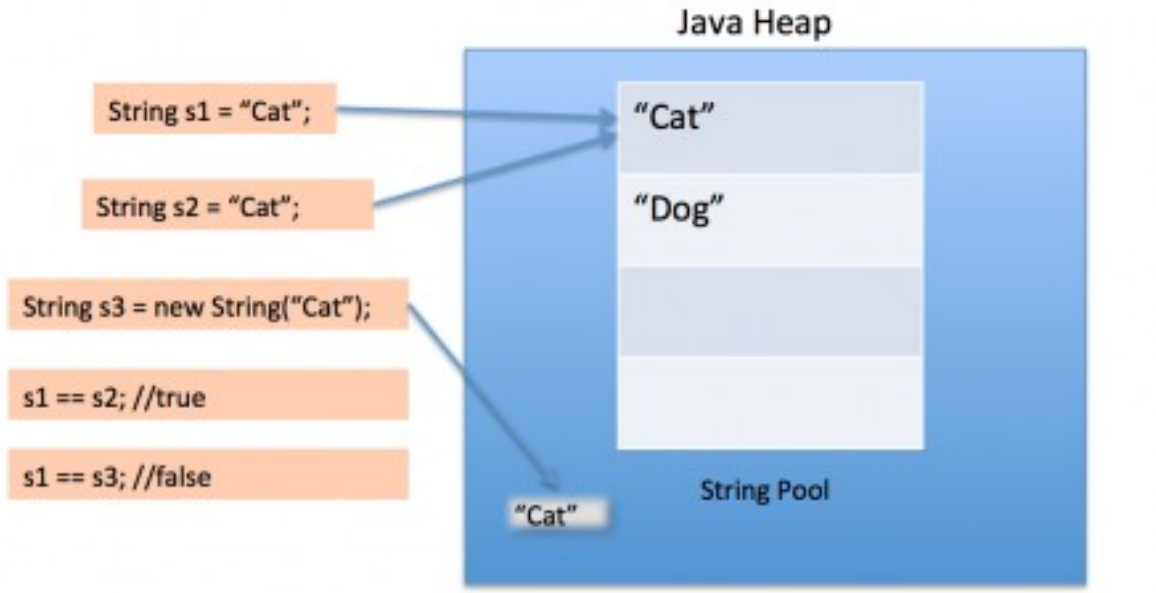
- Para Java una cadena de caracteres no forma parte de los tipos primitivos sino que es un objeto de la clase *String* (java.lang.String)
- Los objetos de la clase *String* se pueden crear:
  - Implícitamente: se utiliza un literal cadena entre comillas dobles. Por ejemplo, al escribir `System.out.println("Hola")`, Java crea un objeto de la clase *String* automáticamente.
  - Explícitamente:
    - `String str="Hola";` // modo tradicional
    - `String str=new String("Hola");` // o también



String Pool

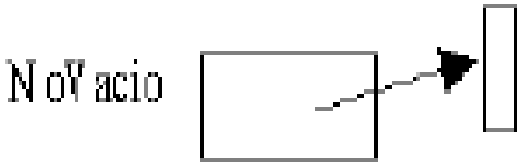
```
String str="Hola"; // modo tradicional

String str=new String("Hola"); // o también
```

- Aunque parece lo mismo, la forma en la que se crea el objeto no es igual.
- La segunda forma siempre crea un nuevo objeto en el heap, una zona de memoria especial para las variables dinámicas, mientras que la primera forma puede crear o no un nuevo objeto (si no lo crea lo reutiliza del String Pool, una memoria caché diseñada para reciclar Strings).



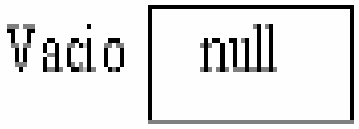
Declarar e inicializar un *String*

- Para crear un *String* sin caracteres se puede hacer:
  - `String str=""; //o`
  - `String str=new String();`



- Es un cadena sin caracteres pero es un objeto de la clase *String*.

- Pero si hacemos:
  - `String str; //o`
  - `String str=null;`



- Se está declarando un objeto de la clase *String*, pero no se ha creado el objeto (su valor es null).

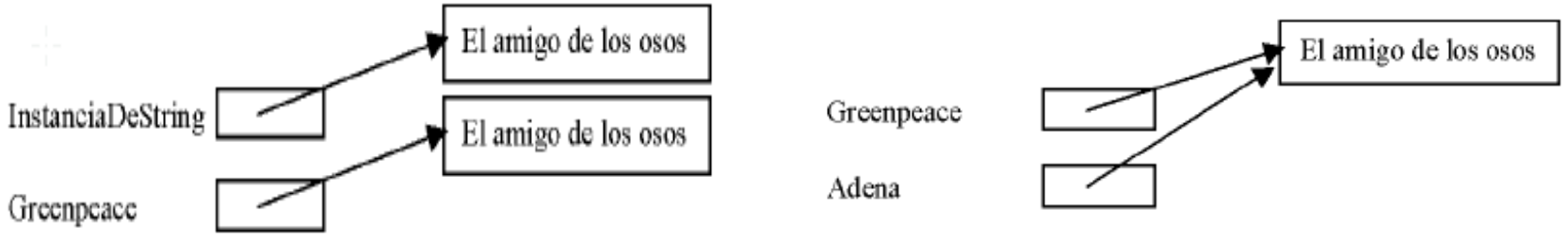
Algunos métodos de la clase *String* (I)

- Para utilizarlos, RECUERDA:
  - Si el método es dinámico
    - variableString.método(argumentos)
  - Si el método es estático
    - String.método(argumentos)
  - La posición del primer carácter de un string es 0, no 1.

- Ante cualquier duda, CONSULTA el API de Java, disponible a :
  - http://docs.oracle.com/javase/7/docs/api/

Comparar objetos *String* (I)

- Es importante tener en cuenta que si tenemos dos instancias de la clase *String* que apuntan a contenidos idénticos, esto no significa que sean iguales aplicando la operador comparación (==)



- Dos strings serán iguales (==) si apuntan a la misma estructura de datos (dirección de memoria).
- Por lo tanto, los objetos *String* no pueden compararse directamente con los operadores de comparación (==).

Comparar objetos *String* (II)

- Disponemos del siguientes métodos:
  - **cadena1.equals(cadena2)**
    - *true* si la cadena1 es igual a la cadena2.
    - Las dos cadenas son variables de tipo *String*.
  - **cadena1.equalsIgnoreCase(cadena2)**
    - Igual que el anterior, pero no se tienen en cuenta mayúsculas y minúsculas.

Comparar objetos *String* (III)

- **cadena1.compareTo(cadena2)**
  - Compara las dos cadenas considerando el orden alfabético (de la tabla ASCII)
  - si la primera cadena es mayor que la segunda: devuelve 1
  - si son iguales: devuelve 0
  - si la segunda es mayor que la primera: devuelve -1

- **cadena.compareToIgnoreCase(cadena2)**
  - Igual que la anterior, pero ignorando mayúsculas

Algunos métodos de la clase *String*

- **length**
  - Devuelve la longitud de una cadena
    - `String texto=”prueba2”;`
    - `System.out.println(texto.length());`//escribe 7

- Para concatenar cadenas se puede hacer de dos formas: con el método **concat** o con el operador **+**.
  - `String s1=”Buen”, s2=”día”, s3, s4;`
  - `s3=s1+s2;`
  - `s4=s1.concat(s2);`

Algunos métodos de la clase *String*

- **charAt**
  - Devuelve un carácter de la cadena.
  - El carácter que se tiene que devolver se indica por su posición.
  - Si la posición es negativa o sobrepasa la longitud de la cadena, se produce un error de ejecución.
    - `String s1=”prueba”;`
    - `char c1=s1.charAt(2);` //c1 valdrá u

Algunos métodos de la clase *String*

- **substring**
  - Devuelve un segmento de una cadena.
  - El segmento va desde una posición inicial hasta una posición final (la posición final no se incluye).
  - Si las posiciones indicadas no son válidas, se genera una excepción.
    - `String s1=”Buenas tardes”;`
    - `String s2=s1.substring(7,12);` //s2=tarde

Algunos métodos de la clase *String*

- **indexOf**
  - Devuelve la primera posición en la cual aparece un texto concreto en la cadena.
  - En caso de que no se encuentre la cadena buscada, devuelve -1.
  - El texto a buscar puede ser *char* o String .
    - `String s1="Quería decirte que quiero que te vayas";`
    - `System.out.println(s1.indexOf("que"));` // escribe 15
  - Se puede buscar desde una posición determinada. Siguiendo el ejemplo anterior:
    - `System.out.println(s1.indexOf("que",16));`
      `// ahora escribe 26`

Algunos métodos de la clase *String*

- **lastIndexOf**
  - Devuelve la última posición en la cual aparece un texto concreto en la cadena.
  - Es casi idéntica a indexOf, pero empieza a buscar desde el final.
    - `String s1="Quería decirte que quiero que te vayas";`
    - `System.out.println(s1.lastIndexOf("que")); // 26`
  - También permite empezar a buscar desde una determinada posición.

Algunos métodos de la clase *String*

- **endsWith**
  - Devuelve *true* si la cadena acaba con un texto en concreto.
    - `String s1="Quería decirte que quiero que te vayas";`
    - `System.out.println(s1.endsWith("vayas"));` // true
- **startsWith**
  - Devuelve true si la cadena empieza con un texto en concreto.

Algunos métodos de la clase *String*

- **replace**
  - Reemplaza todas las apariciones de un carácter por otro en el texto que se indica y lo almacena como resultado.
  - No se modifica el texto original, por tanto debe asignarse el resultado de replace a un *String* para guardar el texto modificado:
    - `String s1=”cosa”;`
    - `System.out.println(s1.replace('o','a'));` // escribe casa
    - `System.out.println(s1);` // continúa con cosa

Algunos métodos de la clase *String*

- **replaceAll**
  - Modifica en un texto cada entrada de una cadena por otra y devuelve el resultado.
  - El primer parámetro es el texto que se busca (que puede ser una expresión regular) y el segundo parámetro es el texto con el cual se reemplaza el texto buscado. La cadena original no se modifica:
    - `String s1=”Cazar armadillos”;`
    - `System.out.println(s1.replaceAll(“ar”,”er”));` // escribe Cazer ermadillos
    - `System.out.println(s1);` // escribe Cazar armadillos

Algunos métodos de la clase *String*

- **toUpperCase**
  - Devuelve la versión en mayúsculas de la cadena.
- **toLowerCase**
  - Devuelve la versión en minúsculas de la cadena.
- **toCharArray**
  - Obtiene un array de caracteres a partir de una cadena.

Algunos métodos de la clase *String*

- **String.valueOf** (método estático)
  - Este método no es exclusivo de la clase *String*, también está presente en otras clases (por ejemplo Integer) y siempre convierte valores de una clase a otra.
  - En el caso de los objetos *String*, permite convertir valores que no son de cadena a forma de cadena.
    - Ejemplos:
      - `String numero = String.valueOf(1234);`
      - `String fecha = String.valueOf(new GregorianCalendar());`
  - En los ejemplos se observa que es un método estático.

Clase *StringBuilder*

- La clase *String* representa una cadena de caracteres no modificable (Inmutable)
  - Una operación como convertir a mayúsculas no modificará el objeto original sino que devolverá un nuevo objeto con la cadena que resultó de la operación.

- La clase *StringBuilder* representa una cadena de caracteres modificable tanto en contenido como en longitud.

- NOTA: La clase *StringBuffer* tiene la misma funcionalidad (constructores y métodos) pero *StringBuilder* tiene mayor rendimiento.

Algunos métodos para *StringBuilder*

- Métodos **length** y capacity.
  - Devuelven, respectivamente, la cantidad real de caracteres que contiene el objeto y la cantidad de caracteres que el objeto puede contener.

- Método **append**
  - Añadimos caracteres al final del objeto.

- Métodos **delete**, **replace** e **insert**
  - Modifican el objeto actual

Clase *StringBuilder*

- Hemos visto que podemos concatenar cadenas de caracteres haciendo uso del operador **+** y también mediante el método **concat** de la clase String.
- Ahora bien, los String son “inmutables” y por tanto solo se pueden crear y leer pero no se pueden modificar.
- Examinamos el siguiente código:

```
public String getMensaje(String[] palabras){
   String mensaje="";
   for(int i=0; i < palabras.length; i++){
      mensaje+=" "+palabras[i];
   }
   return mensaje;
}
```

Clase *StringBuilder*

- Cada vez que se añade una nueva palabra, se reserva una nueva porción de memoria y se rechaza la vieja porción de memoria que es más pequeña (una palabra menos) para que sea liberada por el recolector de basura (garbage collector). Si el bucle se ejecuta 1000 veces, habrá 1000 porciones de memoria que el recolector de basura tiene que identificar y liberar.
- Para evitar este trabajo extra al recolector de basura, se puede emplear la clase StringBuffer que nos permite crear objetos dinámicos, que pueden modificarse.

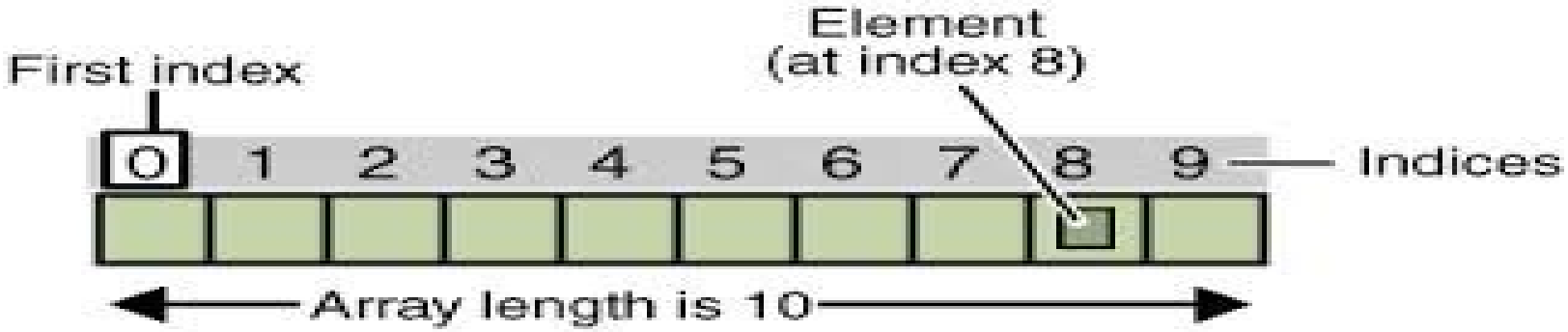
```
public String getMensaje(String[] palabras){
   StringBuilder mensaje=new StringBuilder();
   for(int i=0; i < palabras.length; i++){
      mensaje.append(" ");
      mensaje.append(palabras[i]);
   }
   return mensaje.toString();
}
```

¿Qué es *un array*? (I)

- Un array es una **serie de elementos**.
  - Todos los elementos del array tienen el mismo tipo de datos.
  - Se accede a cada elemento mediante un índice entero.
  - La primera posición del array es la 0.
- La longitud de la array se determina al crearlo.
  - El número de elementos no puede ser modificado en tiempo de ejecución. Los arrays son estáticos. No podremos ni eliminar posiciones ni insertar posiciones.
- Al crear un array, si no se especifican valores, todas las posiciones son inicializadas al valor por defecto del tipo de datos de que se trate.

Qué es *un array*? (II)

- Un array en Java es un clase especial (definida en java.util.Arrays).
- Un array puede contener tipos primitivos o tipos complejos.
- Si intentamos acceder a una posición fuera de la array se producirá un error y se lanzará una excepción (java.lang.IndexOutOfBoundsException)



Declarar y crear un *array* unidimensional (I)

- Es diferente…
  - Declarar una variable array
    - Se crea una variable capaz de apuntar (contener la dirección de memoria) a un objeto array.
    - El objeto todavía no está creado y la variable apunta o contiene null.
  - Crear o instanciar un array
    - Creamos físicamente el objeto. Se le asignan posiciones de memoria.
    - Se utiliza la palabra reservada *new* y se invoca al constructor.
  - Podemos declarar e instanciar en la misma instrucción.

Declarar y crear un *array* unidimensional (II)

- Declarar:
  - tipo_datos[] nomArray;
  - Ejemplo: int[] numeros;
- Crear un array y asignarlo:
  - nomArray = new tipo_datos[num_elementos]
  - Ejemplo: numeros=new int[10];
- Declarar y crear un array
  - tipo_datos[] nomArray = new tipos_datos[num_elementos]
  - Ejemplo: int[] numeros = new int[10];
- Declarar, crear e inicializar:
  - tipo_datos[] nomArray = {v1, v2, v3, ...};
  - Ejemplo: int[] numeros={53, 15, -22, 60, 6, 8 ,14, -75,12, 64};

Acceso a los elementos de un *array* unidimensional

- Si tenemos: int[ ] numeros = {2, -4, 15, -25};
- Si un array unidimensional **a** tiene n elementos:
  - Al primer elemento se accede con a[0]
  - Y al último elemento se accede con a[n-1]

- Acceso para lectura (leer el valor):
  - Ejemplo: System.out.println(numeros[3]);

- Acceso para escritura (modificar un valor):
  - numeros[2] = 99;

Recorrido de los elementos de un *array* unidimensional (I)

- Es una operación muy frecuente recorrer los elementos de un array.
- Se puede utilizar un bucle con contador…

```
int[] datos = {1, 2, 3, 4};
for(int i=0; i < datos.length; i++){
  System.out.println(datos[i] + " ");
}
```

- O iterar sobre los elementos…

```
for(int dato : datos){
  System.out.println(dato + " ");
}
```

Recorrido de los elementos de un *array* unidimensional (II)

- Ejemplo:

```
int maximo = Integer.MIN_VALUE;
for (int i = 0; i < vector.length; i++) {
   if (vector[i] > maximo)
   maximo = vector[i];
}
```

```
int maximo = Integer.MIN_VALUE;
for (int n: vector) {
   if (n > maximo)
   maximo = n;
}
```

Tamaño de los arrays. Arrays vs. Listas

- Los *arrays* son de medida fija, mientras que las listas son de medida variable.
- Si no sabemos el tamaño de un *array* al crearlo, tenemos dos posibilidades:
  - 1.- Crear un *array* muy grande, de forma que quepan los datos. Mala gestión del espacio (se desperdicia).
  - 2.- Crear un *array* de tamaño reducido, pero tener en cuenta que si llegan más datos se tendrá que ampliar (es decir, crear un *array* más grande y copiar los datos). Supone una penalización del tiempo de ejecución total.
- Las listas (clase List) son de medida variable. Las listas son una forma cómoda de aplicar la segunda opción. Lo veremos más adelante…

Arrays y métodos

- Podemos pasar un *array* como parámetro a un método, teniendo en cuenta que **los cambios que realizamos sobre el *array* en el método invocado se mantendrán** al volver el flujo de la ejecución al método invocador.

- Esto es debido a que **los *arrays* son de tipos referencia** y, por lo tanto, las variables *array* con las que trabajamos tanto desde el método invocador como desde el método invocado, son en realidad punteros hacia una misma zona de memoria o referencia (la que contiene la *array)*.

Arrays y métodos. Ejemplo

- Por lo tanto, cuando se invoca a un método y se le pasa un *array*, el método hace su copia de la referencia, pero comparte la *array*.

```
void caso1(int[] x) {
   x[0] *= 10;
}

void test1() {
   int[] a = {1, 2, 3};
   System.out.println(Arrays.toString(a));
   caso1(a);
   System.out.println(Arrays.toString(a));
}
```

Ejecución

[1, 2,3]
[10, 2, 3]

Copia de *arrays* (I)

- Cuando una variable de tipo *array* se hace igual a otra, se copia la referencia (y se **comparte** el *array*):

```
void copia1() {
    int[] a = {1, 2, 3};

    System.out.println(Arrays.toString(a));
    int[] b = a;
    System.out.println(Arrays.toString(b));
    a[0] *= 10;
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
}
```

Ejecución
[1, 2, 3]
[1, 2, 3]
[10, 2, 3]
[10, 2, 3]

Copia de *arrays* (II)

- Si además de compartir la referencia queremos una copia del *array*, se puede emplear el método **clone()** :
  - Si los elementos del *array* son de un tipo **primitivo**, se copia su valor.

```
void copia2() {
    int[] a = {1, 2, 3};
    System.out.println(Arrays.toString(a));
    int[] b = a.clone();
    System.out.println(Arrays.toString(b));
    a[0] *= 10;
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
}
```

Ejecución
[1, 2, 3]
[1, 2, 3]
[10, 2, 3]
[1, 2, 3]

Copia de *arrays* (III)

- Con el método **clone()**, si los elementos del *array* son objetos, se copia la referencia (se comparte el objeto).

```
void copia2Objetos() {
    Punto[] a = {new Punto(1, 2), new Punto(3, 4)};
    System.out.println(Arrays.toString(a));
    Punto[] b = a.clone();
    System.out.println(Arrays.toString(b));
    a[0].multiplica(-1);
    System.out.println(Arrays.toString(b));
}
```

Ejecución
[(1,2), (3,4)]
[(1,2), (3,4)]
[(-1,-2), (3,4)]

Copia de *arrays* (IV)

- Por último, la copia se puede hacer de forma explícita.

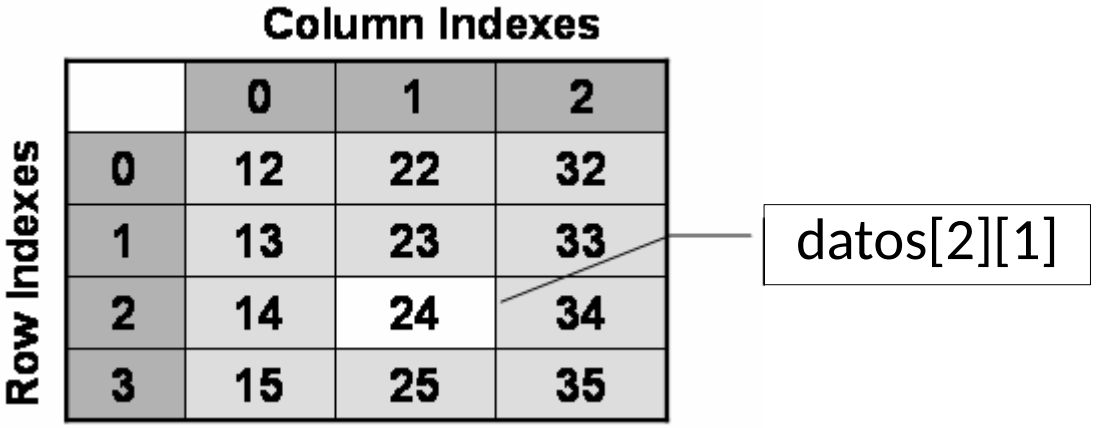
```
tipo[] a = ...;

tipo[] b = new tipo[a.length];

for (int i = 0; i < a.length; i++) {

    b[i] = a[i];
}
```

Algunos métodos de los *arrays*

- Propiedad pública **length:** devuelve el número de elementos del *array*.
- **Arrays.sort():** ordena un *array*.
- **Arrays.binarySearch():** busca un valor en *un array*. El *array* debe estar ordenado.
- **Arrays.equals():** compara arrays.
- **Arrays.fill():** asigna valores a un array.
- **Arrays.toString():** convierte un *array* en un *String*.
- **Arrays.copyOf(), clone()**: copian un *array*.

Arrays multidimensionales o matrices

- Podemos pensar en una matriz de 2 dimensiones como si fuera una cuadrícula.



- Declarar una matriz de 4 filas y 3 columnas de números enteros:
  - `int[][] datos = new int[4][3];`
- Asignar un valor a un elemento específico de la matriz:
  - `datos[2][1] = 24;`

Declarar y crear un *array* multidimensional

- Declarar:
  - tipo_datos[][] nomArray;
  - Ejemplo: int[][] numeros;
- Crear un array y asignarlo:
  - nomArray = new tipo_datos[num_filas][num_columnas]
  - Ejemplo: numeros=new int[10][5];
- Declarar y crear un array
  - tipo_datos[][] nomArray = new tipos_datos[num_elementos]
  - Ejemplo: int[][] numeros = new int[10][5];
- Declarar, crear e inicializar:
  - tipo_datos[][] nomArray = {v1, v2, v3, ...};
  - Ejemplo: int[][] numeros={{1, 2, 3}, {4, 5, 6}};

Arrays bidimensionales o matrices

- En realidad, un *array* multidimensional es *un array* de *arrays*.
- Por ejemplo:
  - Uno *array* 4x3 es *un array* de 4 elementos, en el cual cada uno de ellos es *un array* de 3 elementos:
    - datos[0] es *un array* de 3 elementos
    - …
    - datos[3] es *un array* de 3 elementos
- Podemos tener *arrays* bidimensionales no cuadrados (cada fila puede tener un número diferente de columnas)
  - int[ ][ ] numeros = new int[4][ ];
  - numeros[0]=new int [7];
  - …
  - numeros[3]=new int[3];

Arrays bidimensionales o matrices

- length
  - Proporciona el número de elementos o longitud del *array*
  - matriz.length //número de filas
  - matriz[0].length //número de columnas de la primera fila
  - matriz[1].length //número de columnas de la segunda fila

Recorrido de los elementos de una matriz

- Se puede emplear un bucle con contador

```
double[][] matriz={{1,2,3,4},{5,6},{7,8,9,10,11,12},{13}};

for(int i=0; i < matriz.length; i++) {
    for(int j=0; j < matriz[i].length; j++) {
        System.out.print(matriz[i][j]+"\t");
    }
    System.out.println("");
}
```

- o iterar sobre sus elementos

```
for(double[] fila : matriz) {
    for(double dato : fila) {
        System.out.print(dato + " ");
    }
    System.out.println("");
}
```

Arrays multidimensionales. Ejemplo

```
public class MatrizUnidadApp {
   public static void main (String[] args) {
      double[][] mUnidad= new double[4][4];
      for(int i=0; i < mUnidad.length; i++) {
         for(int j=0; j < mUnidad[i].length; j++) {
            if(i == j) {
               mUnidad[i][j]=1.0;
            }
            else {
               mUnidad[i][j] = 0.0;
            }
         }
      }
      for(int i=0; i < mUnidad.length; i++) {
         for(int j=0; j < mUnidad[i].length; j++) {
            System.out.print(mUnidad[i][j]+"\t");
         }
         System.out.println("");
      }
   }
}
```

Arrays multidimensionales

- Para crear una matriz multidimensional:
  - tipusDades[ ][ ][ ]... nomVariable = new tipusDades[dimensión1] [dimensión2][dimensión3]...

- total elementos = dimensión1 x dimensión x dimensión3 ...