UD6. Estructuras de Datos I
Strings y Arrays

Módulo: Programación

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

Germán Gascón Grau
g.gascongrau@edu.gva.es

Contenidos

- Tipos de datos
- Strings
- Arrays

Tipos 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 vs Tipo 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

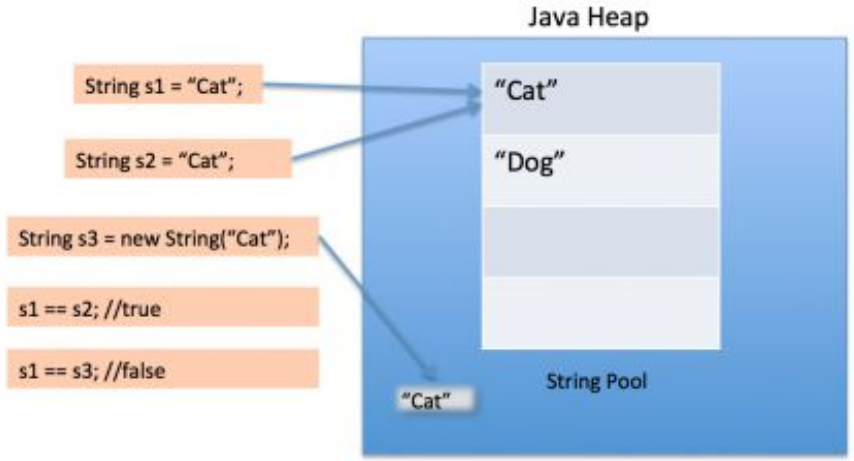
- Para Java una cadena de caracteres no forma parte de los tipos primitivos sino que es un objeto de la clase **String** (`java.lang.String`)
- Los objetos de la clase String se pueden crear:
  - Implícitamente: se utiliza un literal cadena entre comillas dobles. Por ejemplo, al escribir `System.out.println("Hola")`, 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 s1 = "Hola";
String s2 = new String("Hola");
```

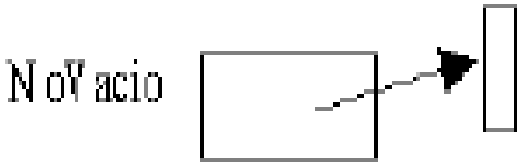
- Aunque parece lo mismo, la forma en la que se crea el objeto no es igual.
- La segunda forma siempre crea un nuevo objeto en el heap, una zona de memoria especial para las variables dinámicas, mientras que la primera forma puede crear o no un nuevo objeto (si no lo crea lo reutiliza del String Pool,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



Declarar e inicializar un String

- Para crear un String sin caracteres se puede hacer:

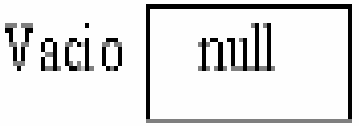
```
String str = "";
String str = new String();
```



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

- Pero si hacemos:

```
String str;
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

- Métodos **estáticos** vs **dinámicos**
  - Los métodos **estáticos** se invocan utilizando el nombre de la clase (String) seguidos de un punto:
    - String.método(argumentos)
    - Ejemplo:

```
String numStr = String.valueOf(15);
String precioStr = String.format("Precio: %.2f", 1.2523);
```

  - Los métodos **dinámicos** se invocan utilizando la variable de tipo String.
    - variableString.método(argumentos)
    - Ejemplo:

```
String s = "Hola";
int longitud = s.length();
char c = s.charAt(0); // c valdrá 'H'
```

  - Recuerda que el primer carácter de un String ocupa la posición la 0

Métodos estáticos de la clase String

- `String` **`String.valueOf`**`(...)` (sobrecargado)
- Método sobrecargado (múltiples versiones). Este método 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.

```
String numStr = String.valueOf(1234);
String floatStr = String.valueOf(1.23f);
String fechaStr = String.valueOf(new Date());
```

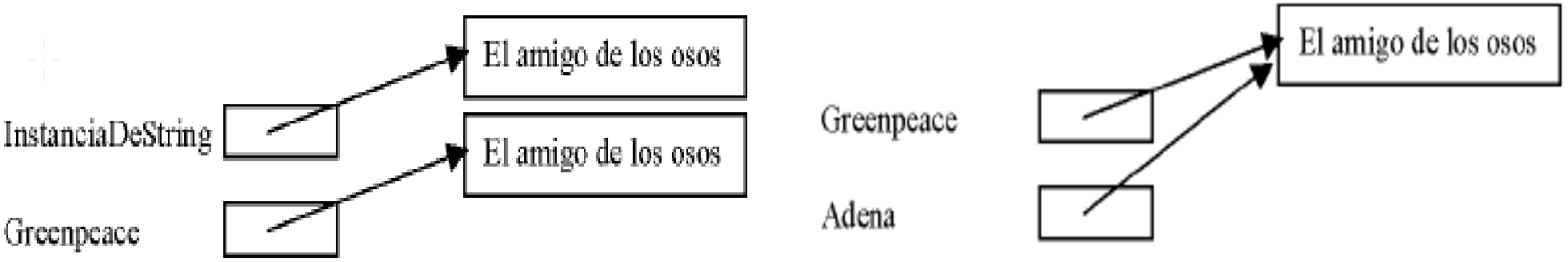
Métodos estáticos de la clase String

- `String` **`String.format`**`(String format, Object… args)`
- Formatea una cadena de texto con el formato especificado.
- La sintaxis de format es la misma que la utilizada por el método printf.

```
String texto = String.format("Precio: %.2f", 1.432); // Precio: 1.43
```

Comparar objetos String

- 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 (==), deben compararse con el método **equals**()

Comparar objetos String

- `boolean` **`equals`**`(Object obj)`
  - true si la cadena1 es igual a la cadena2.
  - Las dos cadenas son variables de tipo String.

```
String s1 = “Juan”;
boolean comparacion = s1.equals(“juan”); // false
```

  - Recuerda que las mayúsculas y minúsculas tienen códigos ASCII distintos.
- `boolean` **`equalsIgnoreCase`**`(String str)`
  - Igual que el anterior, pero no se tienen en cuenta mayúsculas y minúsculas.

```
String s1 = “Juan”;
boolean comparacion = s1.equalsIgnoreCase(“juan”); // true
```

Comparar objetos String

- `int` **`compareTo`**`(String str)`
  - Compara las dos cadenas considerando el orden alfabético (de la tabla ASCII):
  - si la primera cadena es mayor que la segunda devuelve un valor **> 0**
  - si son iguales devuelve **0**
  - si la segunda es mayor que la primera devuelve un valor **< 0**

```
int distancia = "A".compareTo("B"); // distancia vale -1
distancia = "C".compareTo("A"); // distancia vale 2
distancia = "A".compareTo("a"); // distancia vale -32
```

  - Recuerda que las mayúsculas y las minúsculas tienen códigos ASCII distintos
- `int` **`compareToIgnoreCase`**`(String str)`
  - Igual que la anterior, pero ignorando mayúsculas y minúsculas.

Algunos métodos de la clase String

- `int` **`length`**`()`
  - Devuelve la longitud de una cadena

```
String texto = "Hola";
System.out.println(texto.length()); // escribe 4
```

- `String` **`concat`**`(String str)`
  - Para concatenar cadenas se puede hacer de dos formas con el método **concat** o con el operador **+**.

```
String s1 = "Buenos", s2 = "días", s3, s4;
s3 = s1 + " " + s2; // s3 vale Buenos días
s4 = s1.concat(" ").concat(s2); // s4 vale Buenos días
```

  - La concatenación de String en Java es una operación considerada ineficiente debido a la inmutabilidad de estos. En breve veremos como concatenar Strings de forma eficiente con la clase StringBuilder.

Algunos métodos de la clase String

- `char` **`charAt`**`(int index)`
  - Devuelve el carácter de la cadena que ocupa el índice indicado.
  - El primer carácter de la cadena tiene el índice 0.
  - 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`**`(int beginIndex, int endIndex)`
  - Devuelve un segmento de una cadena que va desde un índice inicial hasta un índice final (el índice 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

- `int` **`indexOf`**`(String str)`
  - 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
```

- `int` **`indexOf`**`(String str, int fromIndex)`
  - Se puede buscar desde un índice determinado. Siguiendo el ejemplo anterior:

```
System.out.println(s1.indexOf("que",16));  // escribe 26
```

Algunos métodos de la clase String

- `int` **`lastIndexOf`**`(String str)`
  - 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"));
```

- `int` **`lastIndexOf`**`(String str, int fromIndex)`
  - También permite empezar a buscar desde un determinado índice. Hay que tener en cuenta que la búsqueda la realiza **empezando por el final de la cadena**.

Algunos métodos de la clase String

- `boolean` **`endsWith`**`(String suffix)`

  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
```

- `boolean` **`startsWith`**`(String prefix)`

  Devuelve true si la cadena empieza con un texto en concreto.

```
String s1 = "Quería decirte que quiero que te vayas";
System.out.println(s1.startsWith("Que")); // true
```

Algunos métodos de la clase String

- `String` **`replace`**`(char oldChar, char newChar)`
  - Reemplaza todas las apariciones de un carácter por otro en el String que invoca al método y lo devuelve como resultado.
  - No se modifica el texto original del String que invoca al método, ya que la clase String es **inmutable**, 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);   // escribe cosa
```

Algunos métodos de la clase String

- `String` **`replace`**`(CharSequence target, CharSequence replacement)`
  - Reemplaza todas las apariciones de una subcadena por otra subcadena en el String que invoca al método y lo devuelve como resultado.
  - No se modifica el texto original del String que invoca al método, ya que la clase String es **inmutable**, por tanto debe asignarse el resultado de replace a un String para guardar el texto modificado:

```
String s1 = "cosa";
System.out.println(s1.replace("sa","do")); // escribe codo
System.out.println(s1);   // escribe cosa
```

Algunos métodos de la clase String

- `String` **`replaceAll`**`(String regex, String rep)`
  - Reemplaza texto de un cadena según las reglas indicadas por una **expresión regular** (regex) y devuelve el texto reemplazado como resultado.
  - El primer parámetro es el texto que se busca en forma de expresión regular.
  - 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
```

Expresiones regulares. Patrones

- `[]` Los corchetes se utilizan para definir conjuntos de valores.
  - `[abc]` Hace matching con los caracteres indicados.

```
String s = "Hola mundo";
System.out.println(s.replaceAll("[au]", "o")); // Holo mondo
```

  - `[^abc]` Hace matching con todos los caracteres excepto los indicados.
  - `[0-9]` Hace matching con el rango indicado. Válido también para caracteres. Basado en su valor ASCII.

```
String s = "abcdefg";
System.out.println(s.replaceAll("['b-f']", "*")); // a*****g
```

Expresiones regulares. Metacaracteres

- | Hace matching con alguno (OR) de los patrones indicados.

```
String s = "Tengo un perro y un gato";
System.out.println(s.replaceAll("perro|gato", "animal")); // Tengo un animal y un animal
```

- . Actúa como comodín de un sólo carácter

```
String s = "Esta vela no pesa";
System.out.println(s.replaceAll("e.a", "illa")); // Esta villa no pilla
```

Expresiones regulares. Metacaracteres

- ^ Hace matching con cadenas que empiecen con los caracteres indicados.

```
String s = "Hola mundo. Hola";
System.out.println(s.replaceAll("^Hola",
"Te saludo")); // Te saludo mundo. Hola
```

- $ Hace matching con cadenas que finalicen con los caracteres indicados.

```
String s = "Hola mundo. Hola";
System.out.println(s.replaceAll("Hola$",
"Te saludo")); // Hola mundo. Te saludo
```

Expresiones regulares. Metacaracteres

- `\d` Hace matching con dígitos.

  Como el carácter '\' es un carácter especial hay que escaparlo.

```
String s = "Tengo 18 años";

System.out.println(s.replaceAll("\\d",
"*")); // Tengo ** años
```

- `\s` Hace matching con espacios en blanco.

```
String s = "Carro cero";

System.out.println(s.replaceAll("\\s","")); //
Carrocero
```

Expresiones regulares. Cuantificadores

- **n+** Hace matching con cualquier cadena que contenga almenos un n.

```
String s = "Texto    mal         separado";
System.out.println(s.replaceAll("\\s+", " ")); // Texto mal separado
```

- **n*** Hace matching con cualquier cadena que contenga cero o más ocurrencias de n.

```
String s = "Texto    mal         separado";
System.out.println(s.replaceAll("\\s*", " ")); //" T e x t o  m a l
s e p a r a d o "
```

- **n?** Hace matching con cualquier cadena que contenga cero o una ocurrencia n.

```
String s = "Texto    mal         separado";
System.out.println(s.replaceAll("\\s?", " ")); //" T e x t o  m a l
        s e p a r a d o "
```

- Para más información sobre **expresiones regulares** ver:

  https://docs.oracle.com/javase/tutorial/essential/regex/

Algunos métodos de la clase String

- `String` **`toUpperCase`**`()`

  Devuelve la versión en mayúsculas de la cadena.

```
String s = "hola".toUpperCase(); // "HOLA"
```

- `String` **`toLowerCase`**`()`

  Devuelve la versión en minúsculas de la cadena.

```
String s = "Hola".toLowerCase(); // "hola"
```

- `char []` **`toCharArray`**`()`

  Obtiene un array de caracteres a partir de una cadena.

```
char c = "Hola".toCharArray(); // c = ['H', 'o', 'l', 'a']
```

El problema de la concatenación de Strings

- Como ya hemos comentado en varias ocaciones, concatenar Strings haciendo uso del operador + o del método concat de la clase String, no es una buena idea.
- Los String son “inmutables” y por tanto solo se pueden crear y leer pero no se pueden modificar.
- Examinemos el siguiente código:

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

El problema de la concatenación de Strings

- 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();
}
```

Clase StringBuilder

- La clase String representa una cadena de caracteres no modificable (Inmutable)

- Por tanto, 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.

Métodos de StringBuilder

- `int` **`length`**`()`

  Devuelve la cantidad real de caracteres que contiene el objeto.

- `int` **`capacity`**`()`

  Devuelve la cantidad de caracteres que el objeto puede contener.

- `StringBuilder` **`append`**`(...)` (sobrecargado)

  Añade caracteres al final del objeto.

- `StringBuilder` **`delete`**`(int start, int end)`

  Elimina los caracteres comprendidos entre start y end.

Métodos de StringBuilder

- `StringBuilder` **`replace`**`(int start, int end, String str)`

  Reemplaza los caracteres comprendidos entre start y end por la cadena indicada.

- `StringBuilder` **`insert`**`(int offset, …)` (sobrecargado)

  Inserta los caracteres indicados a partir del índice indicado.

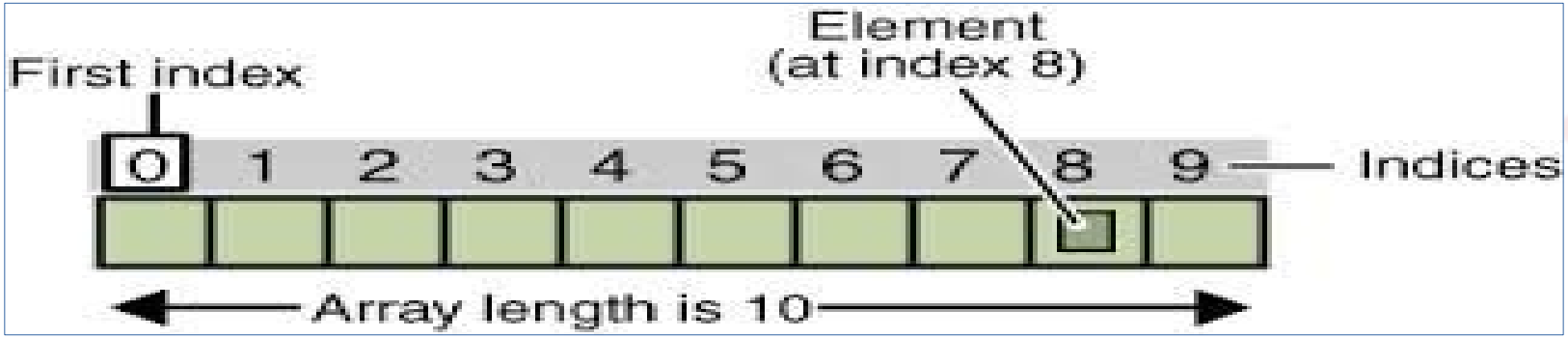
Arrays

¿Qué es un array?

- Un **array** es una **serie de elementos** con las siguientes características:
  - 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 el índice **0**.
  - La **longitud** de la array es **fija**, es decir no puede crecer de tamaño, y debe ser **conocida en el momento de su creación**.
  - 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?

- 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 vs Crear(Instanciar)

- **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**.

```
int[] datos;
```

  - La variable datos apuntará a un array pero el array aún no ha sido creado.
- **Instanciar** o Crear un array
  - Creamos físicamente el objeto. Se le asignan posiciones de memoria.
  - Se utiliza la palabra reservada **new** y se invoca al constructor.

```
int[] datos;
datos = new int[20];
```

  - La variable datos apunta al nuevo array creado de 20 elementos.
- Podemos declarar e instanciar en la misma instrucción:

```
int[] datos = new int[20];
```

Declarar, Instanciar e Inicializar

- 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 tipo_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. 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):

  Ejemplo: `numeros[2] = 99;`

Recorrido de los elementos. Array unidimensional

- 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. Array unidimensional

Veamos un ejemplo para obtener el valor máximo de un array:

```
int[] vector = {53, 15, -22, 60, 6, 8 ,14, -75, 12, 64};
int maximo = Integer.MIN_VALUE;
for (int i = 0; i < vector.length; i++) {
   if (vector[i] > maximo)
   maximo = vector[i];
}
```

- Utilizando un bucle for con iterador:

```
int[] vector = {53, 15, -22, 60, 6, 8 ,14, -75, 12, 64};
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 tamaño fijo, mientras que las listas son de tamaño variable.
- Si no sabemos el tamaño de un array al crearlo, tenemos dos posibilidades:
  - Crear un array muy grande, de forma que quepan los datos. Mala gestión del espacio (se desperdicia).
  - 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).
- Las listas (clase List) son de medida variable. Las listas son una forma cómoda de aplicar la segunda opción.
- En próximos temas trataremos a fondo las Listas.

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

- 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 el 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

- Cuando una variable de tipo array se asigna a otra, se copia la referencia y por tanto se **comparte** el array, ya que las dos variables apuntan al mismo 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

- 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

- 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

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

```
int[] a =  {53, 15, -22, 60, 6, 8 ,14, -75, 12, 64};
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`**. No se trata en realidad de un método sino de una propiedad de la clase Array y devuelve el número de elementos del array.
- `void` **`Arrays.sort`**`(T[] a)` (sobrecargado)

  Ordena el array pasado como parámetro.
- `int` **`Arrays.binarySearch`**`(T[] a, T key)` (sobrecargado)

  Busca un valor en un array. Si lo encuentra devuelve la posición. El array debe estar ordenado.
- `boolean` **`Arrays.equals`**`(T[] a, T[]b)` (sobrecargado)

  Determina si dos arrays son iguales
- `void` **`Arrays.fill`**`(T[] a, T valor)` (sobrecargado)

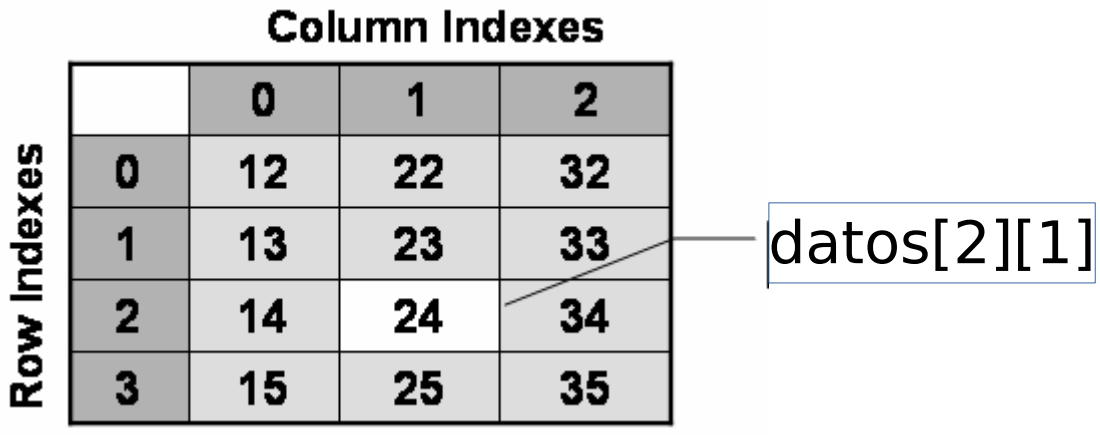
  Rellena el array con el valor indicado.
- `String` **`Arrays.toString`**`(T[]a)` (sobrecargado)

  Convierte un array en un String.
- `T[]` **`Arrays.copyOf`**`(T[] a, int newLength)` (sobrecargado)

  Crea una nueva copia de 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_elem][num_elem];`
  - Ejemplo: `int[][] numeros = new int[10][5];`
- Declarar, crear e inicializar:
  - `tipo_datos[][] nomArray = {a1, a2, a3, …};`

    Donde a1, a2, a3 … son arrays
  - 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:
- Un 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

Arrays bidimensionales o matrices

- 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];
```

- Podemos obtener la longitud de cada fila con consultando su propiedad length
  - `numeros.length` // número de filas
  - `numeros[0].length` // número de columnas de la 1ª fila
  - `numeros[1].length` // número de columnas de la 2ª fila
  - Y así sucesivamente

Recorrido de los elementos de una matriz

- Utilizando un bucle 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("");
}
```

- Utilizando un bucle for con iterador

```
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:
- tipoDatos[ ][ ][ ]... nombreVariable = new tipoDatos[dimensión1][dimensión2] [dimensión3]...
- total elementos = dimensión1 x dimensión x dimensión3 ...