# 6. Estructuras de almacenamiento de información.

| | |
|---|---|
| Autor | Ⓧ Xerach Casanova |
| Clase | Programación |
| Fecha | @Feb 27, 2021 5:42 PM |



Cuando un programa maneja  datos compuestos (datos compuestos a su vez de datos simples)

Los datos compuestos son un tipo de estructura de datos. Las clases son un ejemplo de estructuras de datos que permiten almacenar datos compuestos. El objeto o instancia de una clase sería un dato compuesto. A veces los datos tienen estructuras aún más complejas y necesitan otro tipo de soluciones.

## 1. Introducción a las estructuras de almacenamiento

A la hora de utilizar estructuras de datos del mismo tipo que varían en tamaño de forma dinámica se utilizan estructuras de datos. Las clases son una evolución de un tipo de estructuras conocidas como datos compuestos, también llamadas registros).

Las estructuras de almacenamiento se pueden clasificar atendiendo a:

- **Estructuras con capacidad de almacenar datos del mismo tipo:** varios números, varios caracteres, etc... Arrays, cadenas de caracteres, listas y conjuntos.
- **Estructuras con capacidad de almacenar varios datos de distinto tipo:** fechas, cadenas de caracteres, etc... Todo dentro de una misma estructura (las clases son un ejemplo de ello).

Atendiendo a la forma que los datos se ordenan en la estructura podemos diferenciar:

- **Estructuras que no se ordenan de por sí** y debe ser el programador el encargado de hacerlo si es necesario.
- **Estructuras ordenadas.** Se trata de estructuras que incorporan un dato nuevo a los ya existentes, este se almacena en una posición concreta e irá en función del orden.

# 2. Cadenas de caracteres

Son estructuras de almacenamiento que permiten almacenar secuencias de caracteres de casi cualquier longitud.  Un carácter se codifica como secuencias de bits que representan los símbolos usados en la comunicación humana: letras, números, símbolos matemáticos, ideogramas y pictogramas.

La forma más habitual de ver escrita una cadena de caracteres es como un literal de cadena, que consiste simplemente en una secuencia de caracteres entre comillas dobles.

Los literales de cadena en Java son instancias de la clase String. Cada vez que escribimos un literal, se crea una instancia de la clase String. Esto da flexibilidad pero a la vez consume memoria.

**String cad = "Ejemplo de cadena";**

En este caso, el literal de cadena a la derecha es una instancia de la clase String y la variable cad se convierte en una referencia a ese objeto ya creado. Otra forma de hacerlo sería:

**String cad= new String ("Ejemplo de cadena");**

En este caso, se realiza una copia en memoria de la cadena pasada por parámetro, la nueva instancia de la clase String hará referencia a la copia de la cadena y no a la original.

# 2.1. Operaciones avanzadas con cadenas de caracteres

## 2.1.1. Operaciones avanzadas con cadenas de caracteres (I)

La operación avanzada más sencilla es la concatenación con el signo suma

```
String cad = "Bien"+"venido";
System.out.println(cad);
```

En la operación anterior se crea una nueva cadena a partir de dos cadenas.

Otra forma de usar la concatenación sería usando el método concat del objeto String

```
String cad = "Bien".concat("vendido");
System.out.printf(cad);
```

En ambas expresiones participan tres instancias de la clase String, una contiene el texto "Bien", otra el texto "venido" y otra que contiene "Bienvenido", la cual se crea al realizar la operación de concatenación, sin que las otras dos hayan desaparecido. El recolector de basura se encargará de borrar las otras dos cadenas cuando detecte que ya no se usan.

Además, en el ejemplo se puede observar que se puede invocar un método de la clase String poniendo el método al literal de cadena, esto demuestra que escribiendo el literal se está creando un objeto inmutable String.

Con el método tuString() podemos concatenar cadenas con literales numéricos e instancias de otros objetos.

**El método toString() es un método disponible en todas las clases de Java, permite la conversión de una instancia de clase en cadena de texto. Convertir no siempre es posible, hay clases fácilmente convertibles en texto, como Integer y otras que el resultado de invocar toString() es información relativa a la instancia.**

La ventaja del método toString() es que se invoca automáticamente sin especificarlo:

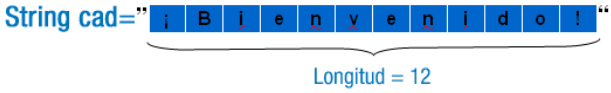
```
Integer i1 = new Integer(1223);
System.out.println("Número: " + i1);
```

En el ejemplo anterior se ha invocado automáticamente i1.tostring();

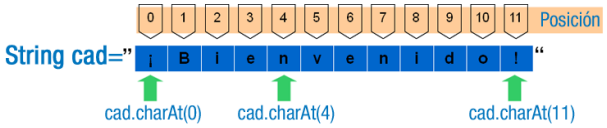
## 2.1.1. Operaciones avanzadas con caracteres de cadena (II)

Partiendo de que en todos los siguientes ejemplos, la cadena cad contiene la cadena ¡Bienvenidos!

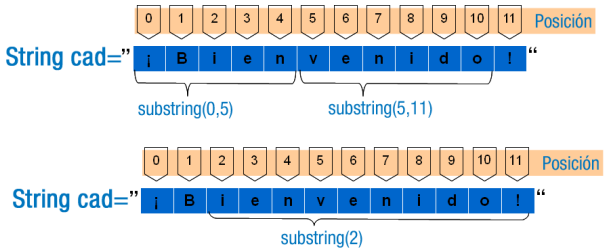
- **int Length().** Retorna un número entero que contiene la longitud de la cadena, incluyendo los espacios.




- **char charAt(int pos).** Retorna el carácter ubicado de la posición pasada por parámetro. El carácter obtenido se almacena en un tipo de dato char. Las posiciones empiezan a contar desde el 0.




- **String substring(int beginIndex, int endIndex).** Permite extraer una subcadena de otra de mayor tamaño entre la posición que se indica en el parámetro beginIndex y la de endIndex -1. cad.substring(0,5) devuelve el valor ¡Bien




- **String substring(int beginIndex).** Si solo se le proporciona un parámetro al método substring, se extraerá una cadena que comience a partir de la posición beginIndex

```
String subcad = cad.substring(2);
System.out.println(subcad);
//Se devuelve "ienvenido!"
```

Otra operación habitual es convertir números a cadenas y cadenas a números. Esto ayuda a evitar errores, haciendo que el usuario siempre inserte cadenas, aunque el dato que vaya a insertar sea un número. Seguidamente se convierte el número en cadena.

Los números almacenados en memoria se hace en números binarios, no se debe confundir tipos de datos numéricos (int, short, long, float y double) con secuencias de caracteres.

La conversión de numéricos es fácil gracias al método toString:

```
String cad2 = "Número cinco: " + 5;
System.out.println(cad2);
```

Esto es posible sin errores gracias a que Java convierte el número 5 a su clase envoltorio (wrapper class) correspondiente: Integer, Float, Double... y después ejecuta el método toString de dicha clase.

## 2.1.2. Operaciones avanzadas con cadenas de caracteres (III)

Para hacer operaciones numéricas con cadenas de caracteres que contienen números, primero se deben convertir esas cadenas a tipos numéricos. El método que ofrece java para ello es valueOf, existente en todas las clases envoltorio descendientes de Number: Integer, Long, Short, Float y Double.

```
String c="1234.5678"; //Cadena que contiene un número flotante

double n; //declaramos una variable double
try {

  n = Double.valueOf(c).doubleValue();
  /* Con el método valueOf convertimos el contenido de
  la variable de cadena c en el tipo primitivo double. */

} catch (NumberFormatException e) {
   //código a ejecturas si no se puede convertir.
}
```

Con el formateado de cadenas conseguimos darle el formato que queramos a tipos numéricos. En Java podemos formatear cadenas a través del método estático format, disponible en el objeto String.

```
float precio = 3.3f;

String salida = string.format("El precio es: %.2f €", precio));

System.out.println(salida);
```

El formato de salida, también denominado "cadena de formato" es el primer argumento del método format, la variable precio es la variable que se proyectará en la salida siguiendo un formato concreto.  %.2f es un especificador de formato.

## 2.1.3. Opciones avanzadas con cadenas de caracteres (IV)

Java ofrece muchas más operaciones sobre cadenas de caracteres. En la siguiente tabla, cad1, cad2 y cad3 son cadenas ya existentes, la variable num es un número entero mayor o igual a cero.

cad

| Método. | Descripción |
|---|---|
| `cad1.compareTo(cad2)` | Permite comparar dos cadenas entre sí lexicográficamente. Retornará 0 si son iguales, un número menor que cero si la cadena (`cad1`) es anterior en orden alfabético a la que se pasa por argumento (`cad2`), y un número mayor que cero si la cadena es posterior en orden alfabético. |
| `cad1.equals(cad2)` | Cuando se comparan si dos cadenas son iguales, no se debe usar el operador de comparación "==", sino el método `equals`. Retornará `true` si son iguales, y `false` si no lo son. |
| `cad1.compareToIgnoreCase(cad2)`<br>`cad1.equalsIgnoreCase(cad2)` | El método `compareToIgnoreCase` funciona igual que el método compareTo, pero ignora las mayúsculas y las minúsculas a la hora de hacer la comparación. Las mayúsculas van antes en orden alfabético que las minúsculas, por lo que hay que tenerlo en cuenta. El método `equalsIgnoresCase` es igual que el método `equals` pero sin tener en cuenta las minúsculas. |
| `cad1.trim()` | Genera una copia de la cadena eliminando los espacios en blanco anteriores y posteriores de la cadena. |
| `cad1.toLowerCase()` | Genera una copia de la cadena con todos los caracteres a minúscula. |
| `cad1.toUpperCase()` | Genera una copia de la cadena con todos los caracteres a mayúsculas. |
| `cad1.indexOf(cad2)`<br>`cad1.indexOf(cad2,num)` | Si la cadena o carácter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1. Opcionalmente se le puede indicar la posición a partir de la cual buscar, lo cual es útil para buscar varias apariciones de una cadena dentro de otra. |
| `cad1.contains(cad2)` | Retornará `true` si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará `false`. |
| `cad1.startsWith(cad2)` | Retornará `true` si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará `false`. |
| `cad1.endsWith(cad2)` | Retornará `true` si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará `false`. |
| `cad1.replace(cad2,cad3)` | Generará una copia de la cadena `cad1`, en la que se reemplazarán todas las apariciones de `cad2` por `cad3`. El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "`zzz`" por "`xx`" en la cadena "`zzzzz`" generará "`xxzz`" y no "`zzxx`". |

### 2.1.4. Operaciones avanzadas con cadenas de caracteres (V)

El principal problema de las cadenas de caracteres es su alto consumo de memoria. Cuando un programa realiza muchas operaciones con cadenas es conveniente optimizar el uso de memoria.

En Java String es un objeto inmutable, esto significa que cuando creamos un String o un literal de String, se crea un objeto no modificable. Java proporciona la clase StringBuilder, mutable, que permite una mayor optimización de memoria, así como la clase StringBuffer, pensada para apliaciones multihilo.

La clase StringBuilder permite modificar la cadena que contiene, mientras que String no.

Por ejemplo:

```
StringBuilder strb = new StringBuilder("Hoal Muuundo");
```

Con los métodos append (insertar al final), insert (insertar una cadena o carácter en una posición específica, delete y replace podemos rectificar la cadena anterior:

1. **strb.delete(6, 8);** eliminamos las 'uu' que sobran en la cadena, la primera u está en la posición 6 empezando desde 0 y la última está en la posición 7 (hay que pasar como parámetro la posición contigua al carácter a eliminar como en el método substring.
2. **strb.append("!");** añadimos al final de la cadena el cierre de exclamación.
3. **strb.insert(0, "¡");** añadimos al principio el símbolo de apertura de exclamación.
4. **strb.replace(3,5, "la");** Reemplazamos los caracteres 'al' situados en la posición inicial 3 y la posición final 4, por 'la', al igual que en el método delete y substring se pasa como parámetro la posición contigua al carácter final (5)

## 2.2. Expresiones regulares

### 2.2.1. Expresiones regulares (I)

La función de las expresiones regulares es permitir comprobar si una cadena sigue o no un patrón preestablecido. Son un mecanismo que describe esos patrones y se construyen de una forma relativamente sencilla.

Existen librerías distintas para trabajar con expresiones regulares y casi todas siguen una sintaxis más o menos similar con ligeras variaciones.

Esta sintaxis permite indicar el patrón de forma cómoda, como si se tratase de una cadena de texto, en la que determinados símbolos tienen un significado especial. Las reglas generales para construir expresiones regulares son:

- Se puede indicar que una cadena contiene un conjunto de símbolos fijo, poniendo esos símbolos en el patrón, excepto para algunos símbolos especiales que necesitarán el código de escape. Por ejemplo el patrón "aaa" admitirá cadenas que contengan tres aes.
- **"[xyz]".** Entre corchetes indicamos opcionalidad, solo uno de los símbolos podrá aparecer en el lugar donde están los corchetes. Por ejemplo "aaa[xy]" admitirá como válidas las cadenas "aaax" y "aaay". Los corchetes representan una posición de la cadena que puede tomar uno o varios valores.
- **"[a-z]" "[A-Z] " "[a-zA-Z]"** indicamos que el patrón admite cualquier carácter entre la letra inicial y final.
- **"[0-9]"** igual que con caracteres pero con numéricos.

Con estas reglas podemos indicar el conjunto de símbolos que admite el patrón y su orden. Si una cadena no contiene los símbolos especificados en el patrón, en el mismo orden, la cadena no encajará en el patrón.

- "a?" El interrogante indica que un símbolo puede aparecer una vez o ninguna.
- "a*" El asterisco indica que un símbolo puede aparecer una, muchas o ninguna vez.
- "a+" El símbolo suma indica que otro símbolo debe aparecer al menos una vez.
- "a{1,4}" Usando llaves se indica el número mínimo y máximo de veces que el símbolo puede repetirse.
- a{2,} El símbolo aparece un mínimo de veces sin determinar el máximo
- a{5} Sin la coma el símbolo debe aparecer exactamente las veces que se indica.
- "[a-z]{1,4}[0-9]+" Los indicadores de repetición también se pueden utilizar entre corchetes. En este ejemplo se permiten de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.

### 2.2.1. Expresiones regulares (II)

Para usar las expresiones regulares, Java ofrece las clases Pattern y Matcher contenidas en el paquete java.util.regex.*.

La clase Pattern se utiliza para procesar la expresión regular y compilarla. Verifica que es correcta y la deja lista para su utilización. Matcher sirve para comprobar si la cadena sigue o no un patrón.

```
Pattern p = Pattern.compile("[01]");
Matcher m = p.matcher("00001010");
if (m.matches()) System.out.println("Sí, contiene el patrón");
else System.out.println("No, no contiene el patrón");
```

El método estático compile de la clase Pattern crea un patrón, el cual compila la expresión regular pasada por parámetro y genera una instancia de Pattern (p). El patrón p se utilizará siempre que quieras para verificar si una cadena coincide o no con el patrón.

Esta comprobación se hace con el método marcher, que combina el patrón con la cadena de entrada y genera una instancia de la clase Matcher (m). La clase Matcher contiene el resultado de dicha comprobación y ofrece varios métodos para analizar la forma en la que la cadena ha encajado con un patrón:

- **m.matches()** devuelve true si toda la cadena (de principio a fin) encaja con el patrón o false en caso contrario.
- **m.lookingAt()** devuelve true si el patrón se encontró al principio de la cadena.
- **m.find()** devuelve true si el patrón existe en algún lugar de la cadena y false en caso contrario puede haber más de una coincidencia. Para obtener la posición exacta donde se produjo la coincidencia usamos m.start() y m.end, una segunda invocación del método find() irá a la segunda coincidencia y así sucesivamente. podemos volver a comenzar invocando al método m.reset();

Algunas construcciones adicionales que pueden ayudarnos a especificar expresiones más complejas:

- **"[^abc]"** Cuando ^ se pone justo detrás del corchete de apertura significa negación. En este caso se admite cualquier símbolo distinto a, b o c.
- **"^[01]+$"** Cuando el símbolo ^ se pone al comienzo, permite indicar comiendo de línea o de entrada, y $ indica fin de línea o fin de entrada. Útil trabajando con modo multilínea y con el método find().
- **\\d** Un dígito numérico (equivale a "[0-9]").
- **\\D** Cualquier cosa excepto un dígito numérico (equivale a "[^0-9]".
- **\\s** Un espacio en blanco (incluye tabulaciones, saltos de línea y otras formas de espacio).
- **\\S** Cualquier cosa excepto espacio en blanco.
- **\\w** Cualquier carácter que podrías encontrar en una palabra, equivale a "[a-zA-Z_0-9]".

### 2.2.2. Expresiones regulares (III)

Los paréntesis tienen un significado especial, permiten indicar repeticiones en un conjunto de símbolos. Por ejemplo: **"(#[01]){2,3}".** La expresión #[01] admite cadenas como #0 o #1, pero al ponerlo entre paréntesis e indicar los contadores de repetición, decimos que la misma secuencia se debe repetir entre 2 y 3 veces, con lo que las cadenas admitidas serían #0#1 o #0#1#0.

Además los paréntesis llevan una función adicional, permite definir grupos. Los grupos permiten acceder de forma cómoda a las diferentes partes de la cadena cuando coincide con una expresión regular.

```
Pattern p = Pattern.compile("([XY]?)([0-9]{1,9})(A-Za-z])");
Matcher m = p.matcher("X123456789Z Y00110011M 999999T"):
while (m.find()) {
  System.out.println("Letra inicial opcional: " + m.group(1));
  System.out.println("Número: " + m.group(2));
  System.out.println("Letra NIF: " + m.group(3));
}
```

Usando los grupos obtenemos por separado el contenido de ca da uno de los grupos.

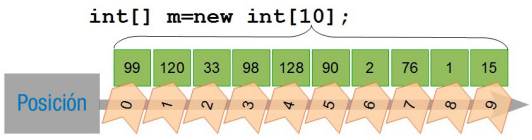
El primer grupo es 1 y no 0, si se pone m.group(0) se obtiene una cadena con toda la concurrencia o coincidencia del patrón en la cadena, es decir, se obtiene la secuencia entera de símbolos que coincide con el patrón.

Si en el ejemplo anterior se usara el método find, este buscaría una a una cada una de las ocurrencias del patrón en la cadena. Cada vez que se invoca, busca la siguiente ocurrencia del patrón y devuelve true. Si no encuentra en una iteración ninguna ocurrencia es porque no existen más y retornará false saliendo del bucle.

**Secuencias de escape.**

Sirven para indicar en una cadena que habrá un paréntesis, una llave, o un corchete. Dado que esos símbolos se usan en los patrones. Para ello se antepone al símbolo \\, excepto para las comillas, en las que se utiliza una sola barra \".

# 3. Creación de arrays




Los arrays permiten almacenar una colección de objetos o datos al mismo tiempo. Su utilización se realiza de la siguiente manera:

- **Declaración.** Sigue la estructura "tipo[] nombre". El tipo será el tipo de dato que almacene o una clase ya existente, de la cual se almacenan varias unidades.
- **Creación del array.** Consiste en indicar el tamaño que tendrá. "nombre = new tipo[dimension]". El array no puede cambiar de tamaño una vez creado.

```
int [] n;
n = new int[10]
```

```
int[] m = new int[10]; //declaración y creación en la misma línea.
```

Una vez declarado y creado podemos almacenar valores en cada una de las posiciones del array, indicando en el interior de los corchetes la posición, siendo cero la primera.

## 3.1. Uso de arrays unidimensionales

Los tres ámbitos donde se utilizan arrays son: modificación de una posición, acceso a una posición y paso de parámetros.

La **modificación de una posición** se realiza con una simple asignación:

```
int[] Numerps = new int[3];

Numeros[0] = 99;
Numeros[1] = 120;
Numeros[2] = 33;
```

El acceso a un valor ya existente se consigue poniendo el nombre del array y la posición entre corchetes

```
int suma = Numeros [0] + Numeros[1] + Numeros[2];
```

Los arrays son como objetos en Java y disponen de la propiedad length, que permite saber el tamaño de cualquier array:

```
System.out.println("Longitud del array: " + Numeros.length);
```

El paso de parámetros  a una función o método se hace de la siguiente manera:

```
int sumaarray (int[] j) {

  int suma = 0;

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

      suma = suma+h[i];

  return suma;
}
```

En este ejemplo pasamos el array j como argumento al método, dentro de él se recorre con un bucle for y se suma el contenido de cada una de las posiciones del array dentro de la variable suma.

Para pasar un array ya creado como argumento se pasa simplemente poniendo el nombre:

```
int suma = sumaarray (Numeros);
```

## 3.2. Inicialización

La forma más habitual de rellenar un array es a través de un método que lleve a cabo la creación y rellenado del array, seguidamente el método retorna el array indicando en la declaración que el valor retornado es tipo[].

```
static int[] arrayConNumerosConsecutivos (int totalNumeros) {

  int [] r = new int[totalNumeros];

  for (int i=0; i<totalNumeros; i++] r[i] = i;

  return r;
}
```

En este ejemplo se crea dentro de un método que crea un array con una serie de números consecutivos, el tamaño del array se pasa por parámetro al método. El método finalmente devuelve el array inicializado con los valores que hemos introducido en el bucle for.

Otra forma de inicializar es en la propia declaración del array, utilizando llaves.

```
int[] array = {10, 20, 30};

String[] diassemana = {"lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo"};
```

Este tipo de inicialización funciona solo con datos primitivos o Strings y algunos casos más, pero no con cualquier objeto.

En una creación de array de objetos, la inicialización inicial será null. Al crear un array de objetos no significa que se haya creado las instancias de los objetos, las cuales hay que crearlas para cada posición del array.

```
StringBuilder[] j= new StringBuilder[10];

for (int 1=0; i<j.length; i++) j[i] = new StringBuilder("cadena "+i);
```

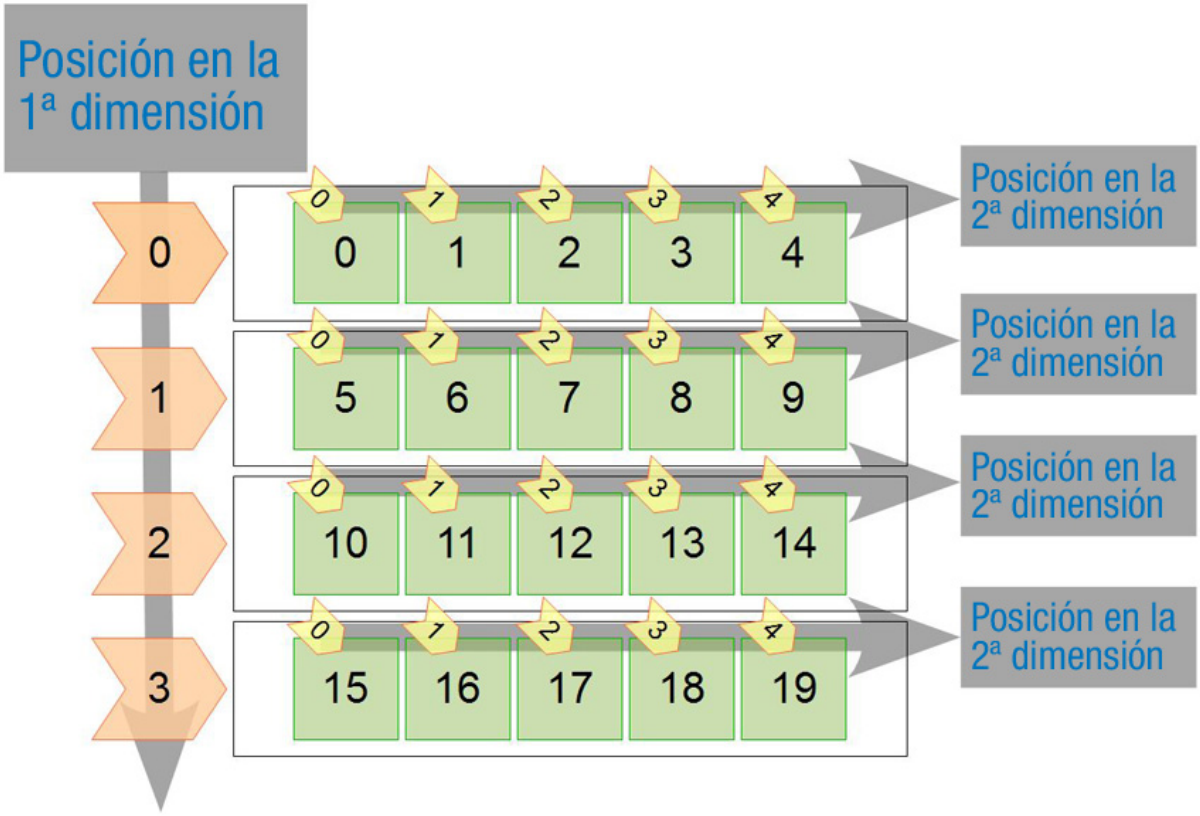
En este ejemplo creamos un array de objetos de la clase StringBuilder, seguidamente utilizamos un bucle for para inicializar cada una de las posiciones del array

# 4. Arrays multidimensionales

Los arrays multidimensionales en Java se crean de la siguiente manera:

```
int[][] a2d=new int[4][5];
```

Se crea un array de dos dimensiones, que contendrá 4 arrays de 5 números cada uno:




Se pueden hacer arrays de las dimensiones que queramos y de cualquier tipo. Todos deben ser del mismo tipo y la declaración comienza especificando el tipo de clase de los elementos, después ponemos tantos corchetes como dimensiones tenga el array y después el nombre del array.

## 4.1. Uso de arrays multidimensionales

```
  int[][] a2d = new int[4][5];P
```

Para acceder a cada uno de los elementos del array anterior se indica la posición en las dos dimensiones, teniendo en cuenta que los índices de ambas empiezan en 0.

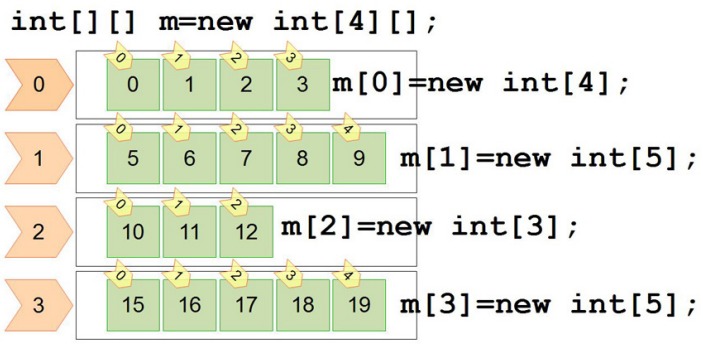
```
ad2[0][0] = 3;
```

También se pueden pasar como parámetros a métodos:

```
static int sumaarray2d(int[][] a2d){

    int suma = =;

    for (int i1=0; i1 < ad2.length; i1++)
        for (in i2 = 0; i2 < a2d[i1].length; i2++)
            suma += a2d[i1][i2];

return suma;

}
```

Aplicando length directamente sobre el array nos permite saber el tamaño de la primera dimensión (a2d.length). Para saber el tamaño de una dimensión superior tenemos que poner el o los índices entre corchetes seguidos de length (a2d[i1].length).

Gracias a esto podemos tener arrays multidimensionales irregulares:




Esto significa que los arrays de la segunda dimensión pueden ser de distinto tamaño entre sí. Para hacer esto se hace de la siguiente manera:

- Declaramos el array sin especificar la segunda dimensión: irregular = new int[3][];
- Después creamos cada uno de los arrays unidimensionales del tamaño que queramos y lo asignamos a la posición correspondiente del array anterior:

  irregular[0] = new int[7];
  irregular[1] = new int[15];
  irregular[2] = new int[9];

## 4.2. Inicialización de arrays multidimensionales

Para que una función retorne un array multidimensional se hace igual que en arrays multidimensionales, simplemente hay que poner el tipo de dato seguido del número de corchetes correspondiente.

```
int[][] inicializarArray (int n, int m)  {

     int[][] ret=new int[n][m];

     for (int i=0;i<n;i++)

          for (int j=0;j<m;j++)

               ret[i][j]=n*m;

     return ret;

}
```

También se puede inicializar usando llaves, poniendo después de la declaración del array un símbolo de igual, encerrando entre llaves los valores del array separados por comas, pero poniendo llaves nuevas en cada nueva dimensión:

```
int[][] a2d={{0,1,2},{3,4,5},{6,7,8},{9,10,11}};

int[][][] a3d={{{0,1},{2,3}},{{0,1},{2,3}}};
```

El primer array es de 4×3 y el segundo array es de 2×2×2. Con esta notación también se pueden inicializar arrays irregulares:

```
int[][] i2d={{0,1},{0,1,2},{0,1,2,3},{0,1,2,3,4},{0,1,2,3,4,5}};

int[][][] i3d={ { {0,1},{0,2} } , {{0,1,3}} , {{0,3,4},{0,1,5} } };
```

Por último se ha de saber que los arrays también reciben el nombre de arreglos, vectores o matrices.

# Mapa conceptual