UT6 - ESTRUCTURAS DE ALMACENAMIENTO DE INFORMACIÓN

Cadenas de caracteres y arrays

Introducción

Introducción (i)

- Cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables.
- Para poder realizar ciertas aplicaciones se necesita poder manejar datos que van más allá de meros datos simples (números y letras).
- A veces, los datos que tiene que manejar la aplicación son datos compuestos de varios datos más simples
- Las clases son un ejemplo de estructuras de datos que permiten almacenar datos compuestos, y el objeto en sí, la instancia de una clase, sería el dato compuesto.

Introducción (y II)

- ¿Cómo almacenarías en memoria un listado de números del que tienes que extraer el valor máximo?
- Un array es una posible opción, una estructura de datos que permite almacenar un lista datos del mismo tipo
- En general, una estructura de datos nos permite almacenar un conjunto de datos asociados entre sí y que tiene lógica que sea tratado de forma agrupada
- Las estructuras de almacenamiento pueden clasificarse de varias formas

Tipos de estructuras de almacenamiento de información (I)

- Según si pueden almacenar datos de diferente tipo o no:
  - Almacenan varios datos del MISMO tipo: arrays, cadenas de caracteres, listas y conjuntos.
  - Almacenan varios datos de DIFERENTE tipo: clases
- Si pueden o no cambiar de tamaño (cantidad máxima de datos) de forma dinámica:
  - Estructuras estáticas. En el momento de declararlas y definirlas se define el tamaño y posteriormente no se puede modificar: arrays y arrays multidimensionales
  - Estructuras dinámicas. No es necesario declarar el tamaño que van a tener y de forma dinámica modifican su tamaño según se añadan o quiten elementos: listas, árboles, conjuntos y clase StringBuilder

Tipos de estructuras de almacenamiento información (y II)

- Según el orden de los datos dentro de la estructura
  - Estructuras que no se ordenan de por sí y el programador debe implementar un orden si lo desea: arrays.
  - Estructuras ordenadas. Cuando se añaden datos este se almacena en un orden predefinido determinado por el resto de datos que ya hay: orden alfabéticos, de mayor a menor, etc.

Cadenas de caracteres

- Se representan en Java con la clase String
- Sirven para almacenar un texto largo de una longitud mayor a una única letra
- Internamente almacena el texto como una lista de caracteres (char)
- Es un tipo de dato por referencia
- Para declararlas:

```
String cadena = "Ejemplo de cadena";
String cadena2 = new String ("Ejemplo de cadena");
```

OPERACIONES CON CADENAS DE CARACTERES

Concatenación de cadenas de caracteres (I)

- Consiste en unir 2 o más cadenas de caracteres obteniendo una nueva cadena de caracteres como resultado.
- Se puede hacer mediante el uso del operador +

```
String texto = "¡Bien"+"venido!";
System.out.println(texto);
```

- O mediante el método concat de la clase String

```
String cadena = "¡Bien".concat("venido!");
System.out.printf(cadena);
```

Concatenación de cadenas de caracteres (y II)

- La concatenación de cadenas entre String y otros tipos de objetos se consigue mediante el método toString()
- Es un método disponible en todas las clases de Java. Su objetivo es permitir la conversión de una instancia de clase en cadena de texto, de forma que se pueda convertir a texto el contenido de la instancia.
- Convertir a String no siempre es posible, hay clases fácilmente convertibles a texto
- Las clases que definamos pueden modificar ese método toString() y devolver un texto más acorde a lo que representa la clase. Lo veremos en la próxima UT

Obtener la Longitud de una cadena de caracteres

- Método **int length().** Retorna un número entero que contiene la longitud de una cadena, resultado de contar el número de caracteres por la que está compuesta. Recuerda que un espacio es también un carácter.

```
String cadena = "Mi cadena de caracteres";
System.out.println(cadena.length());
```

Obtener el carácter en una posición

- Método **char charAt(int pos).** Retorna el carácter ubicado en la posición pasada por parámetro. El carácter obtenido de dicha posición será almacenado en un tipo de dato char. Las posiciones se empiezan a contar desde el 0 (y no desde el 1), y van desde 0 hasta longitud - 1.

```
String texto = "¡Programación es mi módulo preferido!";
char letra = texto.charAt(5);
System.out.println(letra );
```

OBTENER PARTE DE UNA CADENA (SUBCADENA) (I)

- Método **String substring(int beginIndex).**
- Estos método permite extraer una subcadena de otra de mayor tamaño. Una cadena compuesta por todos los caracteres existentes entre la posición beginIndex el final.

```
String texto = "Solamente quiero el final del texto";
String subcad = texto.substring(10);
System.out.println(subcad);
```

OBTENER PARTE DE UNA CADENA (SUBCADENA) (Y II)

- Método **String substring(int beginIndex, int endIndex).** Este método permite extraer una subcadena de otra de mayor tamaño. Una cadena compuesta por todos los caracteres existentes entre la posición beginIndex y la posición endIndex - 1.

```
String texto = "Tres tristes tigres";
String subcad = texto.substring(5, 12);
System.out.println(subcad);
```

Convertir un número a texto

- El método toString() que tiene todo objeto en Java nos permite hacer ese trabajo con cualquier objeto
- Alternativamente, si se trata de un tipo básico como números (int, float, short, double, etc.) también se puede concatenar con una cadena vacía

```
String numeroComoTexto = "" + 5;
System.out.println(numeroComoTexto);
```

Convertir un texto a número (I)

- A veces tenemos un dato numérico almacenado como un texto en una variable de tipo String
- No se pueden hacer las operaciones habituales de suma, resta, multiplicación, etc. ni tratar el dato como un número hasta que se convierta
- La clase Integer tiene el método parseInt que nos permite convertir texto a números enteros

```
String miNumeroDeTexto = "3456789";
try {
      int miNumero = Integer.parseInt(miNumeroDeTexto);
 } catch(NumberFormatException ex) {
      System.out.println("El número no es un entero");
 }
```

Convertir un texto a número (y II)

- Si necesitamos obtener un entero con más o menos precisión podemos usar **Short.parseShort**, **Long.parseLong**, etc.
- Para números con decimales podemos usar los métodos **Double.parseDouble** y **Float.parseFloat**

```
String miNumeroConDecimalesDeTexto = "13456.45678";
try {
        float miFloat = Float.parseFloat(miNumeroConDecimalesDeTexto );
        double miDouble= Double.parseDouble(miNumeroConDecimalesDeTexto );
         System.out.println(miFloat);
         System.out.println(miDouble);
  } catch(NumberFormatException ex) {
        System.out.println("El número no es un entero");
  }
```

Formatear valores numéricos en una cadena de texto

- El método **String.format** nos permite usar cadenas de texto formateadas como ya vimos en la UT2: https://docs.oracle.com/javase/tutorial/java/data/numberformat.html
- Para obtener un texto con un valor numérico con 2 decimales, un número entero y un salto de línea al final.

```
String miTextoFormateado = String.format(
"PI es más o menos %.2f si redondeamos a %d decimales%n", Math.PI, 2);

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

OPERACIONES AVANZADAS CON CADENAS DE CARACTERES

Comparar dos cadenas de texto (I)

- Método **int compareTo(String anotherString).**
- Permite comparar dos cadenas entre sí lexicográficamente. Retornará 0 si son iguales, un número menor que cero si la cadena es anterior en orden alfabético a la que se pasa por argumento (anotherString), y un número mayor que cero si la cadena es posterior en orden alfabético.

```
String cadena1 = "abc";
String cadena2 = "bbc";
String cadena3 = "Bcd";

System.out.println(cadena1.compareTo(cadena2));
System.out.println(cadena1.compareTo(cadena3));
```

Comparar dos cadenas de texto (II)

- Método **boolean equals(Object anObject)**.
- 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.

```
String cadena1 = "abc";
String cadena2 = "bbc";

System.out.println(cadena1.equals(cadena2));
```

Comparar dos cadenas de texto (III)

- Método **int compareToIgnoreCase(String str).**
- 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.

```
String cadena1 = "abc";
String cadena2 = "bbc";
String cadena3 = "Bcd";

System.out.println(cadena1.compareToIgnoreCase(cadena2));
System.out.println(cadena1.compareToIgnoreCase(cadena3));
```

Comparar dos cadenas de texto (y IV)

- Método **boolean equalsIgnoreCase(String anotherString).**
- El método equalsIgnoresCase es igual que el método equals pero sin tener en cuenta las minúsculas.

```
String cadena1 = "abc";
String cadena2 = "ABC";

System.out.println(cadena1.equalsIgnoreCase(cadena2));
```

Quitar caracteres en blanco

- Método **string trim().**
- Genera una copia de la cadena eliminando los espacios en blanco, tabuladores y saltos de línea anteriores y posteriores de la cadena.

```
String cadena = "   \tabc  \n";
 System.out.println(cadena.trim());
```

Convertir a minúsculas

- Método **string toLowerCase().**
- Genera una copia de la cadena con todos los caracteres a minúscula.

```
String cadena = "¡ESTE ES UN TEXTO EN MAYÚSCULAS casi ENTERO!";
System.out.println(cadena.toLowerCase());
```

Convertir a mayúsculas

- Método **string toUpperCase().**
- Genera una copia de la cadena con todos los caracteres a mayúsculas.

```
String cadena = "¡este es un texto en minúsculas CASI entero!";
System.out.println(cadena.toUpperCase());
```

Buscar un texto dentro de una cadena (I)

- Método **int indexOf(String str).**
- Si la cadena o carácter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1.

```
String cadena = "Texto con palabras, algunas palabras repetidas";
 System.out.println(cadena.indexOf("palabras"));
```

Buscar un texto dentro de una cadena (y II)

- Método **int indexOf(String str, int fromIndex).**
- Si la cadena o carácter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1. fromIndex indica la posición a partir de la cual buscar, lo cual es útil para buscar varias apariciones de una cadena dentro de otra.

```
String cadena = "Texto con palabras, algunas palabras repetidas";
 System.out.println(cadena.indexOf("palabras", 10));
```

Comprobar si contiene dentro otra subcadena (I)

- Método **boolean contains(CharSequence s).**
- Retornará true si la cadena pasada por argumento está contenida dentro de la cadena en cualquier posición. En caso contrario retornará false. Tiene en cuenta mayúsculas y minúsculas

```
String cadena = "texto con algunas cosas";
 System.out.println(cadena.contains("to"));
```

Comprobar si contiene dentro otra subcadena (II)

- Método **boolean startsWith(String prefix).**
- Retornará true si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará false. Tiene en cuenta mayúsculas y minúsculas

```
String cadena = "texto con algunas cosas";
System.out.println(cadena.startsWith("to"));
```

Comprobar si contiene dentro otra subcadena (y III)

- Método **boolean endsWith(String prefix).**
- Retornará true si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará false.Tiene en cuenta mayúsculas y minúsculas

```
String cadena = "texto con algunas cosas";
System.out.println(cadena.endsWith("sas"));
```

Reemplazar un texto

- Método **String replace(CharSequence target, CharSequence replacement).**
- Generará una copia de la cadena en la que se reemplazarán todas las apariciones de target por replacement. El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "zzz" por "xx" en la cadena "zzzzz" generará "xxzz" y no "zzxx".

```
String cadena = "zzzzz";
System.out.println(cadena.replace("zzz", "xx"));
```

CADENAS DE TEXTO MUTABLES (CAMBIANTES)

Cadenas de texto mutables

- En Java, String es un objeto inmutable, lo cual significa, entre otras cosas, que cada vez que creamos un String, o un literal de String, se crea un nuevo objeto que no es modificable.
- Java proporciona la clase StringBuilder, la cual es un mutable, y permite una mayor optimización de la memoria.
- Básicamente la clase StringBuilder se diferencia respecto a String en que permite modificar la cadena que contiene, mientras que la clase String no.
- Todas las operaciones que hemos visto con Strings implicar crear una nueva instancia de la clase String, con el coste que ello conlleva de memoria y tiempo de ejecución

Clase StringBuilder (I)

- Se instancia de forma muy similar a como crearíamos una nuevo objeto String pero con la clase StringBuilder

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

- Método **StringBuilder delete(int start, int end)**. Elimina los caracteres que están en la posición start hasta la posición end - 1.

```
strb.delete(6,8);
System.out.println(strb);
```

Clase StringBuilder (II)

- Método **StringBuilder append(CharSequence s).** Añade el texto **s** al final de la cadena

```
strb.append ("!");
System.out.println(strb);
```

- Método **StringBuilder insert(int dstOffset, CharSequence s).** Inserta el texto **s** a partir de la posición **dstOffset.**

```
strb.insert (0,"i");
System.out.println(strb);
```

Clase StringBuilder (y III)

- Método **StringBuilder replace(int start, int end, String str)**. Reemplaza los caracteres entre la posición **start** y **end - 1** por el texto str.

```
strb.replace (3,5,"la");
System.out.println(strb);
```

Expresiones regulares

Definición y alcance de las expresiones regulares

- Las expresiones regulares permitir comprobar si una cadena sigue o no un patrón preestablecido.
- La definición del patrón se hace mediante una expresión regular
- Las expresiones regulares solamente permiten definir lenguajes regulares (según la lingüística computacional)
  - Generados por una gramática regular
  - Reconocido por un autómata finito

Sintaxis de expresiones regulares en Java (I)

- Una expresión regular se define a su vez como un texto que sigue ciertas reglas y usa una sintaxis para definir el patrón que quiere representar.
- Para indicar que una cadena contiene un conjunto de símbolos fijo ponemos dichos símbolos en el patrón tal cual, excepto para algunos símbolos especiales
  - Por ejemplo: **"aaa"** admitirá cadenas que contengan tres aes exactamente

Sintaxis de expresiones regulares en Java (II)

- Entre corchetes podemos indicar opcionalidad. Solo uno de los símbolos que hay entre los corchetes podrá aparecer en el lugar donde están los corchetes.
  - Por ejemplo, la expresión regular "**aaa[xy]**" admitirá como válidas las cadenas "aaax" y la cadena "aaay".
- Usando el guión y los corchetes podemos indicar que el patrón admite cualquier carácter entre la letra inicial y la final.
  - "**[a-z]**" representa cualquier letra en minúsculas de la a la z
  - "**[A-Z]**" representa cualquier letra en mayúsculas de la A la Z
  - "**[a-zA-Z]**" representa cualquiera letras de los 2 casos anteriores
  - "**[0-9]**". Indicar cualquier dígito numérico entre 0 y 9, pero solo uno

Sintaxis expresiones regulares en Java (III)- Cardinalidad

- Para indicar que un símbolo puede aparecer o no usamos el interrogante ?
  - "**a?**". La letra "a" podrá aparecer una vez o simplemente no aparecer.
- Para indicar que un símbolo puede no aparecer, aparecer 1 vez o muchas veces usamos el asterisco *
  - "**a***". Cadenas válidas para esta expresión regular serían "aa", "aaa", "aaaaaaaa" o la cadena vacía ""
- Para indicar que un símbolo debe aparecer al menos una vez, pudiendo repetirse cuantas veces se quiera, usaremos el símbolo de suma +
  - "**a+**". Cadenas válidas para esta expresión regular serían "aa", "aaa", "aaaaaaaa" pero NO la cadena vacía ""

Sintaxis expresiones regulares en Java (IV)- Cardinalidad

- Para indicar el número mínimo y máximo de veces que un símbolo puede repetirse usaremos las llaves {}
  - "**a{1,4}**" El primer número 1 indica decir que la letra "a" debe aparecer al menos una vez. El segundo número 4 indica que como máximo puede repetirse cuatro veces. Solamente serían válidas las cadenas "a", "aa", "aaa", "aaaa"
- Para indicar solamente el número mínimo de veces se obvia poner el máximo pero se deja la coma
  - "**a{2,}**". Indica que la letra a debe aparecer al menos 2 veces. Sería válidas las cadenas "aa", "aaa, "aaaa", etc. pero NO lo sería "a"
- Para indicar que un símbolo debe aparecer solamente un número concreto de veces se deja un número sin la coma
  - "**a{5}**". Indica que la letra a debe aparecer exactamente 5 veces. La única cadena válida sería "aaaaa"

Sintaxis expresiones regulares en Java (V)

- Se puede combinar el uso de corchetes visto anteriormente para indicar conjunto de símbolos junto con los de cardinalidad para hacer expresiones más complejas
  - "**[a-z]{1,4}[0-9]+**". Indica una cadena que tiene de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.
  - "**[0-9]{8}[A-Z]**". Indica una cadena con 8 números del 0 al 9 seguidos de una letra mayúscula
  - "**[0-9]{4}[A-Z]{3}**". Indica una cadena de 4 números del 0 al 9 seguidos de 3 letras mayúsculas
  - "**[a-z0-9A-Z]***". Indica una cadena que contenga cualquier cantidad de letras mayúsculas, minúsculas y números incluida la cadena vacía

Uso de expresiones regulares en código Java

Uso de expresiones regulares en código Java (I)

- Para usar las expresiones regulares en Java usaremos dos clases diferentes **Pattern** y **Matcher** definidas en el package **java.util.regex.**

```
Pattern pattern=Pattern.compile("[01]+");
Matcher matcher=pattern.matcher("00001010");
if (matcher.matches()) {
	System.out.println("Si, contiene el patrón");
}
else {
	System.out.println("No, no contiene el patrón");
}
```

Uso de expresiones regulares en código Java (II)

- El método **static Pattern compile(String regex)** de la clase **Pattern** permite crear un patrón
- Dicho método compila la expresión regular pasada por parámetro y genera una instancia de **Pattern**
- El objeto **Pattern** podrá ser usado múltiples veces para verificar si una cadena coincide o no con el patrón, dicha comprobación se hace invocando el método **Matcher matcher(CharSequence input)**
- El método combina el patrón con la cadena de entrada y genera una instancia de la clase **Matcher**
- La clase **Matcher** contiene el resultado de la comprobación y ofrece varios métodos para analizar la forma en la que la cadena ha encajado con un patrón

Uso de expresiones regulares en código Java (III) - Matcher

- El método **boolean matches()** de la clase Matcher devolverá true si toda la cadena (de principio a fin) encaja con el patrón o false en caso contrario.

```
Pattern pattern=Pattern.compile("[01]+");

Matcher matcher=pattern.matcher("00001010");
System.out.println(matcher.matches());

Matcher matcher2=pattern.matcher("000010102");
System.out.println(matcher2.matches());
```

Uso de expresiones regulares en código Java (IV) - Matcher

- El método **boolean lookingAt()** de la clase Matcher devolverá true si el patrón se ha encontrado al principio de la cadena. A diferencia del método **matches**(), la cadena podrá contener al final caracteres adicionales a los indicados por el patrón, sin que ello suponga un problema.

```
Pattern pattern=Pattern.compile("[01]+");
Matcher matcher=pattern.matcher("000010102");
System.out.println(matcher.lookingAt());
```

Uso de expresiones regulares en código Java (V) - Matcher

- El método **boolean find()** de la clase Matcher devolverá si el patrón existe en algún lugar la cadena (no necesariamente toda la cadena debe coincidir con el patrón) y false en caso contrario, pudiendo tener más de una coincidencia.
- Para obtener la posición exacta donde se ha producido la coincidencia con el patrón podemos usar los métodos **int start()** e **int end()**, para saber la posición inicial y final donde se ha encontrado.
- Una segunda invocación del método **boolean find()** irá a la segunda coincidencia si existe, y así sucesivamente. Para reiniciar el método **boolean find()** y que vuelva a comenzar por la primera coincidencia se invoca el método **Matcher reset()**

Uso de expresiones regulares en código Java (VI) - Matcher

```
Pattern pattern=Pattern.compile("[01]+");
Matcher matcher=pattern.matcher("ABC0100CDF1110K");

System.out.println(matcher.find());
System.out.printf("Inicio:%d-Fin:%d%n", matcher.start(), matcher.end());

System.out.println(matcher.find());
System.out.printf("Inicio:%d-Fin:%d%n", matcher.start(), matcher.end());

matcher.reset();
System.out.println(matcher.find());
System.out.printf("Inicio:%d-Fin:%d%n", matcher.start(), matcher.end());
```

EXPRESIONES REGULARES MÁS COMPLEJAS

EXPRESIONES REGULARES MÁS COMPLEJAS (I)

- Para indicar cualquier carácter excepto los que se indican se usa el operador negación **^**
  - "**[^abc]**" Cualquier símbolo diferente a los de entre corchetes se admite. En este caso, cualquier símbolo diferente de "a", "b" o "c".
- Para indicar que el patrón debe estar exactamente al inicio del texto se usa el mismo operador **^** al inicio del patrón.
  - "^[XYZ]". El texto debe comenzar por el carácter X, Y o Z
- Para indicar que el patrón debe estar exactamente al final del texto se usa el operador **$**
  - "[0-9]{2}$". El texto debe terminar por 2 números del 0 al 9

EXPRESIONES REGULARES MÁS COMPLEJAS (II)

- Usando los operador **^** y **$** conjuntamente podemos verificar que una línea completa (de principio a fin) encaje con la expresión regular, es muy útil cuando se trabaja en modo multilínea y con el método **boolean find()** de **Matcher.**
  - **"^[01]+$"** El texto debe empezar por un 0 o un 1 que se puede repetir más de 1 vez y luego terminar. Es decir, que no puede empezar ni terminar por ningún otro símbolo
- El símbolo **.** representa cualquier carácter excepto saltos de línea.
  - **"[a-zA-Z][.]*".** El texto debe ser una letra en mayúsculas seguido de 0 más caracteres cualesquiera.

EXPRESIONES REGULARES MÁS COMPLEJAS (III)

- Para representar un dígito numérico podemos emplear **\\d**. Equivale a "[0-9]"
  - **"[\\d]{9}"** Un número con 9 cifras exactamente.
- Para representar cualquier cosa EXCEPTO un dígito numérico se puede emplear **\\D**. Equivale a "[^0-9]"
  - **"[\\D]+"** Un texto sin números de al menos 1 letra de longitud
- Para representar un espacio en blanco incluyendo tabulaciones, saltos de línea y otras formas de espacio podemos usar **\\s**
  - **"[A-Z]{2,}[\\s]+[\\d]+"** Un texto con al menos 2 letras seguidos de 1 o más espacios, tabuladores o saltos de línea y de 1 o más números

EXPRESIONES REGULARES MÁS COMPLEJAS (IV)

- Para representar cualquier cosa excepto un espacio en blanco se usa **\\S**
- Para representar cualquier carácter que podrías encontrar en una palabra se usa **\\w**. Equivale a "[a-zA-Z_0-9]"
- Para usar los caracteres especiales como simbolos, hay que poner \\ delante
  - **"\\."** Representa el símbolo del .
- ¿Como sería una expresión regular para reconocer un identificador válido en Java?
  - **"[a-zA-Z_$][\\w$]*"** Debe empezar por una letra mayúscula o minúscula el símbolo _ o $ y seguidos de 0 o más caracteres de una palabra más el símbolo $

Grupos de símbolos (I)

- Es posible usar los paréntesis para agrupar repeticiones de un grupo de símbolos
  - **"#[01]){2,3}"** La expresión "#[01]" admitiría cadenas como "#0" o "#1", pero al ponerlo entre paréntesis e indicar los contadores de repetición, lo que estamos diciendo es que la misma secuencia se tiene que repetir entre dos y tres veces, con lo que las cadenas que admitiría serían del estilo a: "#0#1" o "#0#1#0".
- Los grupos también sirve para identificar partes separadas de los textos (grupos) y así extraer partes concretas de un texto más complejo

Grupos de símbolos (II)

- Un ejemplo de expresión regular con grupos que nos permita validar una dirección de email (no para el 100% de los casos) y extraer el nombre de usuario y dominio

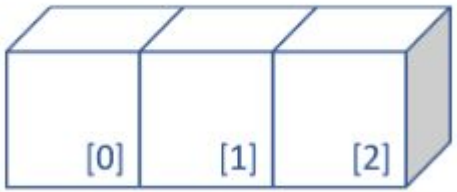
```
Pattern pattern =
Pattern.compile("([a-zA-Z_][\\w]*(\\.[a-zA-Z_][\\w]*)*)@([a-zA-Z_][\\w]*(\\.[a-zA-Z_][
\\w]*)*\\.[a-zA-Z]{2,5})");
Matcher matcher=pattern.matcher("test.usuario@murciaeduca.es");

if(matcher.matches()) {
	for(int i = 0; i< matcher.groupCount(); i++) {
		System.out.printf("Grupo %d=%s%n", i, matcher.group(i));
	}
} else {
	System.out.println("La expresión no es una dirección de email válida");
}
```

ARRAYS

Uso de arrays unidimensionales(I)

- Los arrays permiten almacenar una colección de objetos o datos del mismo tipo usando una única declaración de variable
- Para declarar un array se usa la sintaxis **<tipo_de_dato>[] <identificador>**
  - **int[] edades** declara un array de enteros con el identificador edades
  - **String[] nombreAlumnos** declara un array de objetos String con identificador nombreAlumnos



Uso de arrays unidimensionales(II)

- Para un declarar un array y asignarlo a la variable declarada se usa la sintaxis **<identificador>=new <tipo_de_dato> [<dimension>]"**, donde dimensión es un número entero positivo que indicará el tamaño del array.
  - **edades = new int [10]** crea un array de int con tamaño 10 elementos
  - **nombreAlumnos = new String[50]** crea un array de String con tamaño 50 elementos
- Una vez creado el array este no podrá cambiar de tamaño.

Uso de arrays unidimensionales(III)

- Para modificar el elemento que hay en una determinada posición del array se hace referencia a la posición poniendo entre corchetes un valor entero (que puede variar entre 0 y el tamaño del array -1) y el operador de asignación

```
int[] numeros=new int[3]; // Array de 3 números (posiciones del 0 al 2).
numeros[0]=99; // Primera posición del array.
numeros[1]=120; // Segunda posición del array.
numeros[2]=33; // Tercera y última posición del array.
```

Uso de arrays unidimensionales(IV)

- Para acceder el elemento que hay en una determinada posición del array se hace referencia a la posición poniendo entre corchetes un valor entero (que puede variar entre 0 y el tamaño del array -1)

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

- Para saber el tamaño de un array, se puede consultar la propiedad **int length**

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

Uso de arrays unidimensionales(V)

- **Cuando se crea un array con new inicialmente cada elemento del mismo tiene asignado el valor por defecto del tipo de datos**
- Por ejemplo, un array **int[]** tendrá todos sus elementos con valor **0**, un array de **String** todos a **null**, etc.
- Se puede inicializar un array cuando se declara haciendo una asignación a una lista de elementos del tipo separados por comas y entre llaves

```
int[] array = {10, 20, 30};
String[] dias = {"Lunes","Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"};
```

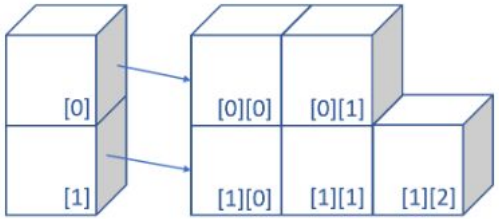
Uso de arrays unidimensionales(VI)

- La inicialización anterior funciona cuando se trata de un tipo de dato primitivo (int, short, float, double, etc.), un String, y algunos pocos casos más, pero no funcionará para cualquier objeto
- Cuando se trata de un array de objetos, la inicialización se debe hacer con un bucle que recorra todos los elementos del array y les asigne un nuevo objeto

```
StringBuilder[] builders=new StringBuilder[10];
for (int i=0; i<builders.length;i++) {
      builders[i] = new StringBuilder("cadena "+i);
}
```

ARRAYS MULTIDIMENSIONALES(I)

- Los arrays multidimensionales permiten almacenar una coleccion de datos pero ordenándolos en forma de tabla si tienes 2 dimensiones (filas y columnas)
- Para declarar un array multidimensional se usa la sintaxis **<tipo_de_dato>[<i>][<j>] … [<k>] <identificador>**
  - **float[][] anios** declara un array arrays de enteros con el identificador anios
  - **StringBuilder[][] apellidos** declara un array de arrays objetos StringBuilder con identificador apellidos



ARRAYS MULTIDIMENSIONALES(II)

- Para un declarar un array multidimensiona y asignarlo a la variable declarada se usa la sintaxis **<identificador>=new <tipo_de_dato> [<dimension1>][dimension2]"**, donde dimension1 y dimensiones2 son números entero positivos que indicarán el números de filas y columnas respectivamente.
  - **anios= new float [4][5]** crea un array de float detamaño 4 filas y 4 columnas
  - **apellidos = new StringBuilder[10][50]** crea un array de objetos StringBuilder con tamaño 10 filas y 50 columnas

ARRAYS MULTIDIMENSIONALES(III)

- Para acceder a cada uno de los elementos del array multidimensional habrá que indicar su posición en las dos dimensiones, teniendo en cuenta que los índices de cada una de las dimensiones empieza a numerarse en 0 y que la última posición es el tamaño de la dimensión en cuestión menos 1.
- Para asignar un valor hacemos uso de la referencia con los corchetes y el operador de asignación

```
anios[0][0] = 3.5f;
apellidos[1][2] = new StringBuilder();
```

ARRAYS MULTIDIMENSIONALES(IV)

- Para acceder el elemento que hay en una determinada posición del array se hace referencia a la posición poniendo entre corchetes un valor entero (que puede variar entre 0 y el tamaño del array -1) para la fila y otra para la columna

```
float suma = anios[0][1] + anios [2][3];
```

- Para saber el número de filas de un array, se puede consultar la propiedad **int length**

```
System.out.println("Filas  del array: "+ anios.length);
```

ARRAYS MULTIDIMENSIONALES(V)

- El número de columnas de cada fila puede ser diferente, por lo que para saberlo hay que hacer referencia a una fila en concreto con un único corchete y accediendo a la propiedad length de esta

```
System.out.println("NºColumnas de la segunda fila  del array: "+ anios[1].length);
```

- Los arrays en los cuales el tamaño de las columnas en cada fila es diferente se llaman arrays irregulares o jagged arrays

ARRAYS MULTIDIMENSIONALES(VI)

- Para declarar e inicializar un array irregular se indica primero en un primer paso solamente el número de filas

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

- A continuación se inicializa cada fila con el tamaño (nº de columnas) que se desee

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

Uso de arrays multidimensionales(VII)

- Se puede inicializar un array multidimensional cuando se declara haciendo una asignación a una lista de elementos del tipo separados por comas y entre llaves, de manera similar a los arrays unidimensionales

```
int[][] array2d={{0,1,2},{3,4,5},{6,7,8},{9,10,11}};

int[][][] array3d={{{0,1},{2,3}},{{0,1},{2,3}}};

int[][] irregular2d={{0,1},{0,1,2},{0,1,2,3},{0,1,2,3,4},{0,1,2,3,4,5}};

int[][][] irregular3d={ { {0,1},{0,2} } , {{0,1,3}} , {{0,3,4},{0,1,5} } };
```

Uso de arrays multidimensionales(VIII)

- Cuando se trata de un array de objetos,no de tipos primitivos, la inicialización se debe hacer con un bucle que recorra todos los elementos del array (filas y columnas)y les asigne un nuevo objeto

```
StringBuilder[][] builders=new StringBuilder[10][5];
for (int i=0; i<builders.length;i++) {
	for(int j = 0; j < builders[i].length;j++) {
		builders[i][j] = new StringBuilder("cadena "+i+j);
	}
}
```