Estructuras de Almacenamiento

Juan Carlos Pérez Rodríguez

Sumario

Introducción

Las estructuras de almacenamiento, tanto si se manejan de manera clásica como si están contenidas en un objeto, pueden ser de dos tipos:

- **Estáticas**: son las que ocupan un espacio determinado en la memoria del ordenador. Este espacio es invariable y lo especifica el programador durante la escritura del código fuente.

- **Dinámicas**: son aquellas cuyo espacio ocupado en la memoria puede modificarse durante la ejecución del programa.

Además, se pueden mencionar como una clase de estructura de almacenamiento diferente las estructuras externas, entendiendo como tales aquellas que no se almacenan en la memoria principal (RAM) del ordenador, sino en alguna memoria secundaria (típicamente, un disco duro). Las estructuras externas, que suelen organizarse en archivos o ficheros, son en realidad estructuras dinámicas almacenadas en memoria secundaria. De los ficheros se hablará más adelante.

Estructuras Estáticas

La estructura más genérica y, por mucho, la que más se utiliza en casi cualquier lenguaje, es el array. Hagamos un leve recordatorio de los array unidimensionales:

La declaración de un array en Java se puede hacer de dos modos:

```
TipoBase nombreArray[];
TipoBase[] nombreArray;
```

Por ejemplo:

```
int serie[];
int[] serie;
```

La variable serie será un array que contendrá números enteros. Todos los números recibirán el mismo nombre, es decir, serie. Observa que aún no hemos especificado cuántos elementos contendrá el array.

Java trata a los arrays unidimensionales como objetos. Por lo tanto, para crear el array se usará esta expresión:

```
serie = new int[5];
```

Como objeto que es si asignamos a una nueva variable mediante el símbolo: “=” lo que tendremos es una referencia al objeto. No un array nuevo

Una forma de recorrer un array es ya conocida: mediante un bucle for que pase por todos los índices:

```
        int array[] = { 7, 3, 9, 2, 8 };

        for (int i = 0; i < array.length; i++) {
            System.out.println(array[i]);
        }
```

Bucle for each

Vamos a ver otra alternativa, mediante un bucle for-each. Es importante hacer notar que el bucle fori que hemos visto antes permite leer y escribir en el array. Sin embargo eso no siempre es posible en el bucle for-each

```
int array[] = { 7, 3, 9, 2, 8 };

for (int elemento : array) {
    System.out.println(elemento);
}
```

Observar que declaramos una variable que debe ser del tipo de objeto que está almacenado en el array (en este caso usamos la variable elemento que es de tipo entero) y va tomando cada elemento desde el que está en la posición 0 hasta el último elemento del array.

Como sabemos cuando estamos con tipos primitivos y estamos creando una variable de tipo entero que es un primitivo, ocurrirá que se copiará lo que está almacenado en cada posición del array en el elemento. Pero como es una COPIA entonces no podemos modificar los valores que están almacenando en el array

Por ejemplo, si hiciéramos:

```
int array[] = { 7, 3, 9, 2, 8 };

for (int elemento : array) {
    elemento = 20;
}
```

No estaríamos modificando la información que está almacenada en el array ya que en la variable elemento lo que se va guardando es una copia del valor almacenado en cada posición del array

Práctica 1: Probar el código anterior y luego recorrer el array para mostrar en pantalla la información que tiene almacenada. Toma captura de pantalla de la salida que muestra. ¿ se ha modificado el array ?

Así pues el bucle for-each está siempre bien si recorremos un array para leer los datos almacenados, pero si la idea es modificarlos pudiera no ser apropiado como se ha visto antes.

Arrays multidimensionales

Una matriz, tabla o array bidimiensional, como un array unidimensional, es una colección de elementos individuales, todos del mismo tipo, agrupados bajo el mismo identificador. La diferencia con el array unidimensional es que, en el momento de declararlo y de acceder a cada elemento individual, debemos utilizar dos índices en lugar de uno. Por ejemplo:

```
int[][] almacen = new int[3][4];
```

Tenemos aquí una matriz que no consta de 4 elementos enteros, sino de 12, es decir, 3x4. Podemos representar gráficamente la matriz como una tabla:

Filas	Columnas 0	1	2	3
0				
1				
2				

Cada casilla de la tabla o matriz es identificable mediante una pareja de índices. Normalmente, el primero de los índices se refiere a la fila, y el segundo, a la columna. Por ejemplo, si hacemos estas asignaciones:

```
almacen[0][0] = 5;
almacen[1][0] = 1;
almacen[2][2] = 13;
```

…el estado en el que quedará la matriz será el siguiente:

	0	1	2	3
0	5			
1	1			
2			13	

Una forma rápida (pero no siempre posible) para crear e inicializar un array multidimensional es establecerlo mediante llaves:

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

Debemos tener en cuenta que los array multidimensionales son array de array. Esto significa que: almacen[i] es también un array

almacen[0] //es un array que tiene todos los elementos de la fila0 de nuestro array multidimensional => {**5**,**1**,**3**,**2**}

Veámoslo:

```
System.out.println(almacen[0].length);
```

Nos va a imprimir 4 en pantalla, diciéndonos con eso que es un array de 4 elementos

Ese array, ya lo sabemos recorrer:

```
        for (int i = 0; i < 4; i++) {
            System.out.println(almacen[0][i]);
        }
```

Ahora veamos como recorremos el array multidimensional completo.

Para recorrerlo procede hacer uso de bucles anidados.

Veamos su recorrido mediante dos índices:

```
      for (int i = 0; i < 3; i++) {
          for (int j = 0; j < 4; j++) {
              System.out.println(almacen[i][j]);
          }
      }
```

Veamos su recorrido mediante un índice y un for-each:

```
for (int[] filaAlmacen : almacen) {
    for (int i = 0; i < filaAlmacen.length; i++) {
        System.out.println(filaAlmacen[i]);
    }
}
```

Debemos observar que por las circunstancias de como trabaja Java, en la forma de recorrer que acabamos de utilizar SÍ podríamos modificar el array ya que los elementos: filaAlmacen son objetos y como tales lo que obtenemos es una referencia al objeto original. Así cuando modificamos filaAlmacen estarémos modificando parte del array. Finalmente como el bucle interior utiliza índices las modificaciones en: filaAlmacen[i] también modificarán el array original

Veamos ahora mediante dos for-each:

```
for (int[] filaAlmacen : almacen) {
    for (int elemento : filaAlmacen) {
        System.out.println(elemento);
    }
}
```

Práctica 2: crear un programa que lea por teclado números enteros y los guarde en una matriz de 5 filas por 4 columnas. Se deberá buscar el número mayor y el número menor mostrándolos así como las posiciones que ocupen. Finalmente se mostrará el array completo (poner un '\n' en los print al final de cada fila del array) recorriéndolo mediante bucles for-each **Nota:** crear una clase: Matriz métodos: setValue() getValue() sirven para establecer/obtener en una posición (x,y) de la matriz el valor. Los métodos: max(), min() devuelven la posición del máximo y el mínimo respectivamente

Práctica 3: Se introducirá por teclado las dimensiones de una matriz (la cantidad de filas y la de columnas) esa matriz se rellenará con números aleatorios enteros desde 1 a 99 inclusives. Calcular el valor medio de cada fila de la matriz y mostrarlo en pantalla especificando a que fila corresponde cada media. **Nota**. Mejor en la clase Matriz de antes tener un método: rellenar() que rellene de los aleatorios. El valor medio mediante método: media()

Debemos entender que al escribir:

```
new int[n][m];
```

estamos creando un array de n posiciones y asignando en cada una de esas posiciones un array de enteros de tamaño m

Eso es lo que hemos visto antes, donde cada vez que escribíamos almacen[i] estábamos haciendo referencia a un array que representaba la fila completa

Pues bien… nada nos impide hacer que el tamaño de cada fila sea diferente. Así podríamos tener un array multidimensional así:

	Columnas			
Filas	0	1	2	3
0				
1				
2				

En el gráfico anterior se ha modelado que:
fila0 tenga tamaño 3
fila1 tenga tamaño 2
fila2 tenga tamaño 4

Una forma de crear ese array multidimensional sería con el código:

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

array[0] = new int[3];
array[1] = new int[2];
array[2] = new int[4];
```

Práctica 4: Utilizando bucles crear un array de dos dimensiones donde la primera fila haya un elemento, en la segunda fila 2 elementos, en la tercera fila 3 elementos … Llegar así hasta 20.

Práctica 5: Observa la siguiente salida de programa:

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1

Siempre el primer y el último elemento de cada fila es 1. Fijémonos ahora en la fila:
1 4 6 4 1
el 4 se obtiene sumando 1 3 que tiene encima.
el 6 se obtiene sumando 3 3 que tiene encima.
Si nos fijamos todos los elementos intermedios se generan así:
array[i][j] = array[i-1][j-1] + array[i-1][j]
Utiliza el array que creaste en la práctica anterior para rellenarlo y posteriormente mostrarlo en pantalla de la forma que acabamos de describir

Práctica 6: Emular la suma de dos matrices. Se creará una clase: MatrizCuadrada con un constructor que reciba el tamaño en filas de la matriz, digamos: **n**. Entonces el constructor establece como atributo un array de n*n Se deberá crear un método llamado: MatrizCuadrada suma(MatrizCuadrada) que sirve para hacer la suma de dos matrices devolviendo la matriz suma (la suma de dos matrices es una nueva matriz donde cada compotenente es la suma de las componentes .

Creación algorítmica de Pilas, Colas y Listas

Más adelante en el tema veremos que hay estructuras dinámicas de librería que tienen contemplado las listas, colas y pilas. Pero vamos a dar una introducción con estructuras estáticas para que entendamos como funcionan internamente

Tanto para pilas como para colas crearemos una clase java con el nombre correspondiente e internamente incluirán un array.

Pilas

Una pila (stack) es un objeto similar a una pila de platos, donde se puede agregar y sacar datos sólo por el extremo superior.

La Pila se utiliza con las siguientes funciones:

- push(dato): agrega el elemento dato a la pila
- pop() : quita el elemento que está en la cima de la pila. También se usa el nombre: *pull.*

Las pilas son estructuras LIFO (*last in, first out*), el último que entra es el primero en salir.

Colas

Pensemos en nuestro día a día. Si nosotros estamos gestionando la cola delante de un local al que quieren acceder múltiples personas, podríamos tener una máquina donde el usuario introduzca (el número sin la letra) del dni en el momento que llegue a la cola, se queda sentado esperando y cuando aparezca su número en pantalla accede al local. Podemos emular esa situación con una Cola de números enteros. Esta estructura es una estructura FIFO (*first in, first out*), el primero en entrar es el primero en salir.

La Cola se utiliza, en forma análoga a una Pila, con las siguientes funciones:

- add(dato): agrega el elemento dato al final de la cola.
- remove(): devuelve el siguiente elemento de la cola . En el ejemplo de antes, devolvería un número entero

Práctica 7: Crear una clase ColaEntero con un constructor: ColaEntero(int size) que recibe el tamaño del array de enteros, con los métodos antes descritos
Se recomienda usar un atributo apuntador a quién es la posición siguiente del array a atender en la cola y otro apuntador a quién es la posición última de la cola que está ocupada

Práctica 8: Crear una clase ColaString con un constructor: ColaString(int size) que recibe el tamaño del array de String, con los métodos antes descritos
Se recomienda usar un atributo apuntador a quién es la posición siguiente del array a atender en la cola y otro apuntador a quién es la posición última de la cola que está ocupada

Al realizar las anteriores dos actividades seguramente observemos que son prácticamente iguales, pero únicamente cambia el tipo de dato que utilizamos.

Para no estar repitiendo código innecesariamente en situaciones de este tipo, en las que lo único que difiere es el tipo de datos que estamos tratando, existen los genéricos:

Genéricos

En muchas ocasiones la única diferencia entre una clase y otra es el tipo de datos que recibe. En ese caso son convenientes los tipos genéricos

Cuando tenemos esa situación sería interesante poder hacer referencia a cualesquier tipo de datos cuando creamos la clase y que sea luego cuando se va a hacer uso de la clase que se establezca el tipo de datos a usar. Lo que se hace es poner una etiqueta identificadora (típicamente: T) allí donde tiene que establecerse el tipo de dato.

Vamos a inventar una clase llamada Incubadora que puede cuidar animales:

```
class Incubadora<T> {
    T paciente;
    int tiempo;
    static final int precioHora=5;
    Incubadora(T paciente, int tiempo){
        this.paciente = paciente;
        this.tiempo = tiempo;
    }
    int coste(){
        return tiempo*precioHora;
    }
    String datosPaciente(){
        return "" + paciente + " tiempo: " + tiempo + " coste: " + coste();
    }
}

public class NewClass {
    public static void main(String[] args) {
        Incubadora inc = new Incubadora<Perro>(new Perro(),10);
        Incubadora inc1 = new Incubadora<Gato>(new Gato(),50);
        System.out.println(inc.datosPaciente());
        System.out.println(inc1.datosPaciente());
    }
}

class Perro{
    String apodo;
}

class Gato{
    String nombre;
    int edad;
}
```

En el anterior código se observa que se crea un objeto de tipo Incubadora<Gato> y otro de tipo Incubadora<Perro>

En el siguiente ejemplo se ve como se pueden usar dos tipos de datos genéricos a la vez:

```
class ParDatos<T,E>{
    T clave;
    E valor;
    ParDatos(T clave, E valor){
        this.clave = clave;
        this.valor = valor;
    }
    void mostrarPar(){
        System.out.println("clave: " + clave + " valor: "+valor);
    }
}

public class NewClass {
    public static void main(String[] args) {


        ParDatos pd1 = new ParDatos<Integer,String>(10,"diez");
        ParDatos pd2 = new ParDatos<Double,String>(2.3,"dos coma tres");
    }
}
```

El ejemplo anterior crea una clase ParDatos que almacena un objeto que hace de clave y otro objeto que hace de valor. Se han creado dos identificadores cualquiera que en este caso hemos llamado T y E separados por coma

En general el formato para una clase que usa genéricos es:

```
class NombreClase<T1, T2, ..., Tn> {
      /* ... */
}
```

Práctica 10: Crear una clase Camión con atributos: String matricula , double tara y T carga Esta carga identifica el tipo de carga que es capaz de transportar el camión (no es lo mismo un camión que transporte petroleo que otro que transporte ganado). Crear un Camion que transporte en su carga a un perro: Camion<Perro> (observar el ejemplo de clase perro de arriba) y Otro que lleve un Gato

Ayuda para entender y trabajar Cola de genéricos (faltan los métodos add() y remove()):

```
class Cola<T>{
	//puntero a la siguiente posición del array libre para que
	//se incorpore alguien a la cola
	private int nextPosFree;
	//puntero a la siguiente posición del array a atender
	private int nextPosQueue;

	private T array[];
	public Cola(int size) {
		//creamos el array
		array = (T [])new Object[size];
	}
}
```

- **Práctica 11**: Realizar con genérico una clase: Cola<T> y así que sea innecesario tener una ColaEntero y otra ColaString

- **Práctica 12**: Realizar con genérico una clase: Pila<T>

Introducción algorítmica a las listas

Otro tipo de objeto de almacenamiento que se usa mucho son las listas. Éstas son dinámicas (permiten crecer en tamaño) y se basan en ir creando objetos y “enlazando” unos a otros

Veamos ejemplo:

```
class Nodo<T>{
	T dato;
	Nodo next;
}

public class Main {
	public static void main(String[] args) {
		Nodo<Integer> nodo1 = new Nodo<Integer>();
		nodo1.dato = 8;
		Nodo<Integer> nodo2 = new Nodo<Integer>();
		nodo2.dato = 5;
		Nodo<Integer> nodo3 = new Nodo<Integer>();
		nodo3.dato = 9;

		//los enlazamos mediante next:
		nodo2.next = nodo3;
		nodo1.next = nodo2;

		//Ahora desde el primero se accede a todos los datos:
		System.out.println(nodo1.dato);
		System.out.println(nodo1.next.dato);
		System.out.println(nodo1.next.next.dato);
	}
}
```

La idea es apoyarse en la clase Nodo, para crear una clase Lista que permita mostrar el array, agregar datos, etc

String

Podríamos considerar un String una especie de array unidimensional cuyos elementos son caracteres

Disponemos de hecho, de un constructor específico para llevar un array de char a String:

```
int array[][] = new int[3][];
char cadena[] = new char[50];
cadena[0] = 'H';
cadena[1] = 'o';
cadena[2] = 'l';
cadena[3] = 'a';

String texto = new String(cadena);
System.out.println(texto);
System.out.println(texto.length());
char cadena1[] = texto.toCharArray();
System.out.println(cadena1);
```

Con lo anterior se ha creado una string que dice “Hola” y que tiene tamaño 50

A su vez se ha vuelto a convertir de la String obtenida a un nuevo char array mediante el método: toCharArray()

Java nos ha facilitado bastante mediante String el trabajo de textos. Nos permite pasar a mayúscula o minúscula: toLowerCase() toUpperCase() localizar un texto dentro de otro texto: matches() y muchos más métodos

¿Sabes cuál es el principal problema de los String? Su alto consumo de memoria. Cuando realizamos un programa que realiza muchísimas operaciones con cadenas, es necesario optimizar el uso de memoria.

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 mutable, y permite una mayor optimización de la memoria

Práctica 13: Tarea de autoaprendizaje: Investiga un poco sobre StringBuilder toma una cadena de texto y haz uso de los métodos: delete(), append(), insert(), replace() debes mostrar casos en los que es útil el uso de esos métodos y como usarlos.

Expresiones Regulares

Prueba el siguiente código:

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

Matcher m=p.matcher("00001010");

if (m.find()) System.out.println("Si, contiene el patrón");

else System.out.println("No, no contiene el patrón");
```

Se ha escrito una expresión regular: “[01]+” y se ha probado en una String a ver si se ajusta a la expresión regular establecida.

Es fácil observar que si quisiéramos saber si el usuario ha introducido las cosas en el formato que nosotros queremos debiéramos poder hacer algún tipo de contraste para verificarlo. Ese es un posible uso de las expresiones regulares. La expresión anterior obliga a que se introduzca una secuencia de 0 y 1 y que al menos haya uno

Veamos algunas construcciones adicionales que pueden ayudarnos a especificar expresiones regulares más complejas:

- "`[^abc]`". El símbolo `"^"`, cuando se pone justo detrás del corchete de apertura, significa "negación". La expresión regular admitirá cualquier símbolo diferente a los puestos entre corchetes. En este caso, cualquier símbolo diferente de "a", "b" o "c".
- "`^[01]+$`". Cuando el símbolo `"^"` aparece al comienzo de la expresión regular, permite indicar comienzo de línea o de entrada. El símbolo "`$`" permite indicar fin de línea o fin de entrada. Usándolos podemos verificar que una línea completa (de principio a fin) encaje con la expresión regular, es muy útil cuando se trabaja en multilnea y con el método `find()`.
- "`.`". El punto simboliza cualquier carácter.
- "`\\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 un espacio en blanco.
- "`\\w`". Cualquier carácter que podrías encontrar en una palabra. Equivale a "`[a-zA-Z_0-9]`".
- X? X, una o ninguna vez
- X* X, cero o ninguna vez
- X+ X, una o mas veces
- X{n} X, exactamente n veces

- X(n,} X, por lo menos n veces
- X{n,m} X, por lo menos n veces pero no mas de m veces
- XY X seguido de Y
- X|Y X o Y

Para hacer uso de esas expresiones regulares y buscarlas en una String ya hemos visto matcher() veamos ese método y otro con más detalle:

- `m.find()`. Devolverá `true` 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 `m.start()` y `m.end()`, para saber la posición inicial y final donde se ha encontrado. Una segunda invocación del método `find()` irá a la segunda coincidencia (si existe), y así sucesivamente. Podemos reiniciar el método `find()`, para que vuelva a comenzar por la primera coincidencia, invocando el método `m.reset()`
- `m.matches()`. Devolverá `true` si toda la cadena (de principio a fin) encaja con el patrón o `false` en caso contrario.

Los paréntesis, de los cuales no hemos hablado hasta ahora, tienen un significado especial, permiten indicar repeticiones para un conjunto de símbolos, por ejemplo: "`(#[01]){2,3}`". En el ejemplo anterior, 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`".

Pero los paréntesis tienen una función adicional, y es la de permitir definir grupos. Un grupo comienza cuando se abre un paréntesis y termina cuando se cierra el paréntesis. Los grupos permiten acceder de forma cómoda a las diferentes partes de una cadena cuando esta coincide con una expresión regular. Lo mejor es verlo con un ejemplo:

```
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));
}
```

Algunos ejemplos de expresiones regulares:

- “((p+)|(q+))en” Una secuencia de como mínimo de tamaño 1 de p seguida de en , o una secuencia de al menos 1 de q seguida de en

- “^[0-4a-d].{3,5}Z*” El comienzo de línea debe tener un dígito de 0 a 4 o una letra de a a d seguido de 3,4 o 5 caracteres y una secuencia de Z de cualquier tamaño (puede no aparecer ninguna) Así hace coincidencia con: "2ñá%Ç=ZZZZZ" "c¡¡€"

Nota: Para una forma fácil para probar expresiones regulares ir a la web: https://regex101.com/

Práctica 14: obtener una expresión regular que verifique si el identificador introducido por el usuario se adapta al formato establecido: Nombre-Edad-siglasdelciclo. Ej
Juan-23-daw

donde Nombre tiene primera letra mayúscula y resto minúscula.
Edad: un número de 20 a 99 años
siglasdelciclo: tres letras minúsculas

Práctica 15: Hacer un programa que el usuario introduzca un texto por teclado y mediante expresiones regulares se determine si es un número válido. Observa que esto significa que debe empezar por una cifra o por los símbolos +- Que después aparezcan cifras y sólo cifras salvo la coma: “,” la cuál nos serviría para delimitar los decimales.
Hacer dos versiones, en la primera debe haber una coincidencia completa en el String que el usuario nos pase, en la segunda basta con que lo primero que muestre la cadena sea un número válido aunque después aparezca más texto.

Práctica 16: Obtener una expresión regular que de coincidencia (mediante find()) de las cadenas (y no de las otras) a), b), c), d), h), i)
a)abcc
b)aBbcaR
c)abbbcPrr
d)abbbbcCC
e)aabbbbcCC
f)aBbrc
g)aabcc
h)abBbc
i)abbBBc

Vamos a ver el caso de una expresión regular compleja. Direcciones IPv4

Las direcciones IPv4 vienen expresadas como 4 Bytes binarios. Cada Byte va de 0 a 255 así que si representamos 4 Bytes separados por puntos queda por ejemplo: 192.255.1.230

El rango general es: 0-255.0-255.0-255.0-255

Vamos a intentar obtener su expresión regular:

Esto no es válido porque son caracteres. No números:

~~[0-255]\.[0-255]\.[0-255]\.[0-255]~~

pero debe ser algo similar. Como primera aproximación vale.

Ahora vamos paso a paso de forma correcta:

Primer octeto: 0 - 255

- caso sin restricciones en la segunda y tercera cifra del octeto

[01]?\d\d

- caso la segunda cifra restringida y la tercera libre:

2[0-4]\d

- caso la tercera cifra restringida:

25[0-5]

Así para el primer octeto queda usando expresiones OR:

[01]?\d\d | 2[0-4]\d | 25[0-5]

Hay que poner el backslash para que no interprete el punto entre octetos: \.

Tener en cuenta que se puede "multiplicar" la expresióntantas veces como se quiera con las llaves.
Así: (expresion){3}
queda:
expresionexpresionexpresion

Práctica 17: Obtener expresión regular para una dirección ip válida: 0-255.0-255.0-255.0-255

Práctica 18: Expresión regular para direcciones de clase C (lo mismo que lo anterior pero con restricción del primer octeto a: 192-223

Estructuras Dinámicas

La biblioteca de Java tiene gran variedad de estructuras dinámicas que se pueden usar. Lleva tiempo trabajarlas y usarlas todas. Nosotros veremos algunas nada más. Queda como actividad de profundización indagar y conocer más respecto e estas estructuras.

Listas: ArrayList, LinkedList,...

Cuando el tamaño de nuestro array puede variar bastante, o necesitamos ser precisos con el espacio que ocupe nuestro array, una mejor opción es ArrayList que permite crecer dinámicamente nuestra estructura de datos. Las LinkedList son otra opción pero se alejan más del concepto de array . Cada una de estas estructuras son más apropiadas según que situación.

Veamos características de las listas:

- Las listas pueden almacenar duplicados, si no queremos duplicados, hay que verificar manualmente que el elemento no esté en la lista antes de su inserción.
- Acceso posicional. Podemos acceder a un elemento indicando su posición en la lista.
- Búsqueda. Es posible buscar elementos en la lista y obtener su posición.
- Extracción de sublistas. Es posible obtener una lista que contenga solo una parte de los elementos de forma muy sencilla.

En Java, para las listas se dispone de una interfaz llamada `java.util.List`, y dos implementaciones (`java.util.LinkedList` y `java.util.ArrayList`)

¿Y en qué se diferencia un `LinkedList` de un `ArrayList`? Los `LinkedList` utilizan listas doblemente enlazadas, que son listas enlazadas, pero que permiten ir hacia atrás en la lista de elementos. Los elementos de la lista se encapsulan en los llamados nodos. Los nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento. Tener un doble enlace significa que en cada nodo se almacena la información de cuál es el siguiente nodo y además, de cuál es el nodo anterior. Si un nodo no tiene nodo siguiente o nodo anterior, se almacena null o nulo para ambos casos.

No es el caso de los `ArrayList`. Estos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente a nosotros, no nos enteramos cuando se produce, pero eso redunda en una diferencia de rendimiento notable dependiendo del uso. Los `ArrayList` son más rápidos en cuanto a acceso a los elementos, acceder a un elemento según su posición es más rápido en un array que en una lista doblemente enlazada (hay que recorrer la lista). En cambio, eliminar un elemento implica muchas más operaciones en un array que en una lista enlazada de cualquier tipo.

¿Y esto que quiere decir? **Que si se van a realizar muchas operaciones de eliminación de elementos sobre la lista, conviene usar una lista enlazada (`LinkedList`), pero si no se van a realizar muchas eliminaciones, sino que solamente se van a insertar y consultar elementos por posición, conviene usar una lista basada en arrays redimensionados (`ArrayList`).**

`LinkedList` tiene otras ventajas que nos puede llevar a su uso. Implementa las interfaces `java.util.Queue` y `java.util.Deque`. Dichas interfaces permiten hacer uso de las listas como si fueran una cola de prioridad o una pila, respectivamente.

Las colas, también conocidas como colas de prioridad, son una lista pero que aportan métodos para trabajar de forma diferente. ¿Tú sabes lo que es hacer cola para que te atiendan en una ventanilla? Pues igual. Se trata de que el que primero llega es el primero en ser atendido (FIFO). Simplemente se aportan tres métodos nuevos: meter en el final de la lista (`add` y `offer`), sacar y eliminar el elemento más antiguo (`poll`), y examinar el elemento al principio de la lista sin eliminarlo (`peek`).

Ejemplos de uso de ArrayList

```
        Random rnd = new Random();
        ArrayList<Integer> al = new ArrayList<>(10);
        for(int i=0;i<8;i++)
            al.add(rnd.nextInt(100));

        for (int num : al) {
            System.out.println(num);
        }
```

En el anterior código se ha creado un ArrayList de Integer de tamaño inicial 10. Se le han introducido 8 números aleatorios y finalmanete se muestran

Observar que utilizamos genéricos (símbolo diamond: <>) para establecer el tipo de objeto. ArrayList no acepta datos primitivos, únicamente trabaja con objetos y por eso utilizamos un wrapper para introducir enteros en el ejemplo anterior. Recordar que establecemos el tipo de datos que vamos a utilizar mediante diamond: <>

Al haber puesto 10 en la creación del arraylist internamente ya ha creado un array de esa dimensión. Si hubiéramos creado el ArrayList de esta otra forma: new ArrayList<>() empezaría por tener tamaño cero y se vería obligado a redimensiones innecesarias al ir insertando los objetos

Debemos entender que el uso del wrapper para enteros hace que internamente haya hecho una conversión del literal entero que le pasamos mediante random en un objeto Integer. Lo que pusimos antes, es equivalente a:

```
            al.add(new Integer(rnd.nextInt(100)));
```

Práctica 19: Crear un programa para la “frase del día” El usuario introduce al principio todas las frases que quiera (escribirá la palabra “fin” para finalizar la entrada de frases) Todas las frases quedarán almacenadas en un ArrayList<String> posteriormente se le muestra una frase elegida al azar de entre todas las introducidas. Diciendo: “la frase del día es: ”
se debe garantizar que todas puedan ser elegidas (un aleatorio desde 0 hasta arraylist.length())

Práctica 20: Modificar el programa anterior para que lo que muestre sea todas las frases que empiecen con la letra a mayúscula: “A”

Práctica 21: Crear un programa para la gestión de clientes. Al usuario se le mostrará un menú donde pueda elegir entre agregar nuevo cliente a la cola, atender cliente, ver estado de la cola.
Crear una clase Cliente con: nombre, apellido, edad y un método: boolean menorDeEdad() que determina si el clientes es o no menor de edad. Cuando el usuario intenta agregar un nuevo cliente al final de la cola, si fuera menor de edad se usa el método y no se incorpora a la cola mostrando un mensaje al usuario diciendo los datos del cliente que se iba a insertar en cola y que no puede agregar a menores de edad.
Cuando el usuario elige atender cliente se toma el próximo elemento que corresponda en la cola quitándolo de la cola y se muestra en pantalla
Cuando el usuario escoge ver estado de la cola se le muestra el tamaño actual de la cola (método size())

Iterator

Veamos un problema que puede darse con los tipos de datos dinámicos:

```
ArrayList<String> al = new ArrayList<>();
String s0 = "yea";
String s1 = "yea1";
String s2= "yea2";
String s3= "yea3";
String s4="yea4";

al.add(s0);
al.add(s1);
al.add(s2);
al.add(s3);
al.add(s4);

//Se recorre el arraylist y si se encuentra lo que quiere se borra
for(String elemento: al){
    if(elemento.equals(s2))
        al.remove(s2);
}
for(String elemento: al){
    System.out.println(elemento);
}
```

Práctica 22: Ejecuta el anterior código en el IDE y has captura de pantalla del error mostrado ¿ qué tipo de error es ?

Si mientras recorremos una lista eliminamos un elemento el bucle puede no saber como continuar con el siguiente elemento. Es por esto que son útiles los iteradores:

En muchos lenguajes orientados a objeto existen unos objetos especiales llamados iteradores. Son objetos pensados para recorrer una colección de datos (del tipo que sean) de forma sencilla y, sobre todo, independiente de la estructura y del tipo de objetos que almacene.

Puedes usar un iterador para recorrer un ArrayList en lugar de un bucle for convencional. Basta con crear dicho iterador con el método iterator() del ArrayList. Una vez creado, el iterador nos proporcionará al menos dos métodos:

hasNext(): devuelve true si aún quedan objetos por recorrer en la estructura.
next(): devuelve el siguiente objeto.

Esta es la forma de recorrer un ArrayList mediante un iterador:

```
ArrayList<Integer> miArray = new ArrayList<Integer>();
...
// Aquí rellenamos el ArrayList con objetos Integer
...
Iterator it = miArray.iterator();
while(it.hasNext()) {
        System.out.println(it.next());
}
```

Observar los metodos:
hasNext() para detectar cuando se finaliza la lista.
next() se desplaza y retorna el siguiente elemento.

Veamos un ejemplo con bucle for en el que se va eliminando elementos una vez mostrados

```
ArrayList<Integer> miArray = new ArrayList<Integer>();
for(int i=0;i<10;i++){
    Random rnd = new Random();
    miArray.add(rnd.nextInt(20));
}

for(Iterator it=miArray.iterator();it.hasNext(); ){
    int numero = (int)it.next();
    System.out.println(numero);
    it.remove();
    System.out.println("tamaño array: " + miArray.size());
}
```

Observar que cuando se ejecuta **it.remove()** se elimina de la lista el último objeto que se ha obtenido con el iterador

Los iteradores pueden usarse con muchas clases de Java (genéricamente, con clases "contenedoras" de objetos). En concreto, con todas las que implementen el interfaz Iterable. Eso incluye clases conocidas para nosotros (como ArrayList o LinkedList).

Práctica 23: Usando el código que dio error en la práctica anterior modifícalo para ahora usar un iterator y que ahora funcione debidamente

Pares clave/valor: mapas

Hemos visto dos estructuras dinámicas muy usadas con ArrayList y LinkedList pero hay muchas estructuras más en la biblioteca de Java. Otras muy útiles son los pares clave/valor. Imaginemos que quisiéramos guardar la puntuación que le están asignando los jueces de una prueba deportiva a los competidores. Necesitamos almacenar el nombre/id de cada competidor y la puntuación que ha obtenido Mediante los mapas podemos almacenar esa información, incluso ordenarlos, etcétera

Un ejemplo clásico de mapas está en los diccionarios. Por un lado tendríamos la palabra y por otro lado guardamos su significados

En Java existe la interfaz java.util.Map que define los métodos que deben tener los mapas, y existen tres implementaciones principales de dicha interfaz: HashMap, TreeMap y LinkedHashMap.

Todas estas estructuras no permiten duplicados. Así que la clave debe ser única.

Veamos un Ejemplo de HashMap

```
HashMap<String,Integer> t=new HashMap<String,Integer>();
```

El HashMap t estará compuesto de un par string/integer que nos podría servir para guardar el nombre de los competidores y la puntuación que le han dado los jueces

Ahora ¿ cómo lo recorremos ?

```
HashMap<String,Double> t=new HashMap<String,Double>();
t.put("Ana", 9.2);
t.put("Luis", 8.5);
t.put("Marta", 6.0);
t.put("Marco", 5.5);
t.put("Lidia",8.0);
Iterator it = t.entrySet().iterator();

while (it.hasNext()) {
    Map.Entry<String, Double> entry = (Map.Entry<String, Double>) it.next();
    System.out.println("Clave="+entry.getKey()+", Valor="+entry.getValue());
    // it.remove();
}
```

Observar que hemos usado **entrySet()** y **Map.Entry** .Mediante Map.Entry obtenemos el par clave/valor. Observar también que se ha usado un iterator. En un comentario aparece la opción para eliminar una entrada del HashMap si quisiéramos (de forma segura ya que usamos el iterator)

Sin embargo tenemos más opciones si lo que queremos es recorrer únicamente el grupo de claves o el grupo de valores:

```
//iterando sobre claves
for (String key : map.keySet()) {
    System.out.println("Key = " + key);
}

//iterando sobre valores
for (Double value : map.values()) {
    System.out.println("Value = " + value);
}
```

Todo lo anterior lo hemos visto con HashMap pero sería perfectamente aplicable con TreeMap y LinkedHashMap

Pero ¿ por qué usar un HashMap u otra de las estructuras?

Propiedad	HashMap	TreeMap	LinkedHashMap
Orden	no ordenado	ordenado	orden de inserción
coste tomar o insertar un elemento	O(1)	O(log(n))	O(1)
Null	permitido	sólo valores, no claves	permitido

Viendo la anterior tabla se ven claramente las diferencias de TreeMap respecto a los otros dos. Pero las diferencias entre HashMap y LinkedHashMap es más difícil

LinkedHashMap es un subtipo de HashMap teniendo las mismas circunstancias de HashMap pero adicionalmente la lista enlazada preserva el orden de inserción en la lista

Precisamente por añadir una lista enlazada al HashMap ocurre que LinkedHashMap ocupa más memoria.

Práctica 24: Crear un diccionario de español/inglés. Se deberán registrar al menos 10 palabras en español y su traducción en inglés. Se deberá realizar con un HashMap y con un TreeMap. Recorrer en ambos casos la estructura completa y mostrar en pantalla Hacer una captura de pantalla de cada una de las ejecuciones ¿ alguno se muestra ordenado ?

Práctica 25: Tratar de introducir en la aplicación de la práctica anterior una palabra repetida como clave con un valor distinto Mostrar en pantalla la estructura completa ¿ ha permitido la inserción de la clave repetida ? (tomar una captura de pantalla) En el caso de que no lo permitiera ¿ que valor es el que permanece ? ¿ el primero introducido o el segundo ?

Conjuntos

Los conjuntos permiten operaciones entre conjuntos obteniendo la intersección etc. podemos encontrar: HashSet, LinkedSHashSet, TreeSet. Queda como práctica de investigación profundizar

Práctica 26: Práctica de autoaprendizaje: Buscar información sobre las 3 estructuras que acabamos de nombrar. Mostrar algún ejemplo respecto a los métodos: addAll(), removeAll(), retainAll()

Después de haber visualizado varias de las estructuras que están disponible en la biblioteca de Java debiéramos tener claro que:

- los array son muy rápidos pero son estructuras estáticas con pocos métodos asociados si se compara con las otras estructuras

- Usaremos mapas cuando precisemos estructuras dinámicas y pares clave/valor siendo TreeMap ordenado y ninguno de ellos permitiendo duplicados

- Usaremos listas cuando precisemos estructuras dinámicas que no son del tipo clave/valor. Siendo ArrayList la estructura más parecida a un Array y LinkedList la más útil si vamos a hacer múltiples inserciones/borrados aparte que permite ser tratada como una cola

Ordenación de los elementos: Comparator

La clase **Collections** y la clase **Arrays** facilitan el método **sort()**, que permiten ordenar respectivamente listas y arrays. Los siguientes ejemplos ordenarían los números de forma ascendente (de menor a mayor):

```
        int []arr= {8,1,11,5,21,10,4,15};
        Arrays.sort(arr);
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);

        }
```

```
        ArrayList<Integer> arl=new ArrayList<Integer>(10);

        arl.add(8); arl.add(1);arl.add(11);
        arl.add(5); arl.add(21);arl.add(10);
        arl.add(4); arl.add(15);
        //antes de ordenar:
        for (Integer num : arl) {
            System.out.print(num +" ");
        }
        System.out.println("");
        Collections.sort(arl);
        //ya ordenado:
        for (Integer num : arl) {
            System.out.print(num +" ");
        }
```

Lo anterior es solución cuando son enteros, texto,.. pero ¿ qué haremos si queremos ordenar objetos. Por ejemplo la clase Persona?

```
class Persona{
    private String nombre;
    private String apellido;
    private int edad;
    Persona(String nombre,String apellido,int edad){
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    @Override
    public String toString(){
        return nombre + " " + apellido + " edad: "+edad;
    }
}
```

Mirando la clase Persona que tenemos puesta ¿ cuál es el criterio de ordenación ? Podemos elegir ordenar por edad, ordenar por apellido, por nombre. Tenemos que decirle a Java como debe ordenar los objetos de la clase persona para que nos los pueda ordenar

Para lo anterior tenemos dos alternativas:

- Primera opción: Modificar nuestra Clase para decirle a Java como se ordena
  Para ello usaremos el interfaz Comparable

```
class Persona implements Comparable<Persona>{
    private String nombre;
    private String apellido;
    private int edad;
    Persona(String nombre,String apellido,int edad){
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    @Override
    public String toString(){
        return nombre + " " + apellido + " edad: "+edad;
    }

    @Override
    public int compareTo(Persona p) {
        return Integer.compare(this.edad, p.edad);
    }
}
```

En el código que se muestra se le dice a Java que para ordenar los objetos Persona lo haga mediante el atributo entero edad. Respecto a lo que hace: Integer.compare(x,y) es devolver un 0 si x e y son iguales. Menor que cero si x < y ; y mayor que 0 si x > y
Por lo que el método compare anterior es equivalente a:

```
    @Override
    public int compareTo(Persona p) {
        int resultado;
        if( this.edad > p.edad)
            resultado = 1;
        else if( this.edad == p.edad)
            resultado = 0;
        else
            resultado = -1;
```

```
        return resultado;
    }
```

O todavía más sencillo:

```
    @Override
    public int compareTo(Persona p) {
        return this.edad - p.edad;
    }
```

Ahora ya se puede ejecutar la ordenación:

```
        ArrayList<Persona> personas = new ArrayList<>(3);
        personas.add(new Persona("Marta","León",25));
        personas.add(new Persona("Julián","Luz",20));
        personas.add(new Persona("Pilar","Ramos",29));

        Collections.sort(personas);
        for(Persona p: personas)
            System.out.println(p);
```

- Segunda opción: Crear una nueva Clase que haga uso de la que queremos ordenar y que le digamos a Java como se ordena esa nueva Clase

```
class ComparadorPersonas implements Comparator<Persona>{

    @Override
    public int compare(Persona p1, Persona p2) {
        return Integer.compare(p1.getEdad(),p2.getEdad());
    }
}
```

Una vez tenemos la nueva clase le decimos que la utilice para ordenar

```
        ArrayList<Persona> personas = new ArrayList<>(3);
        personas.add(new Persona("Marta","León",25));
        personas.add(new Persona("Julián","Luz",20));
        personas.add(new Persona("Pilar","Ramos",29));

        Collections.sort(personas, new ComparadorPersonas());
```

```
        for(Persona p: personas)
            System.out.println(p);
```

Práctica 27: Práctica de autoaprendizaje: Buscar información sobre las 3 estructuras que acabamos de nombrar. Mostrar algún ejemplo respecto a los métodos: addAll(), removeAll(), retainAll()

Práctica 28: Crear la clase Telegrama con atributos: String texto, String remitente, String receptor, double precioPalabra Deberá tener como mínimo un constructor. La longitud del atributo texto determina el coste del telegrama al multiplicarlo por precioPalabra. Deberá haber un método: double coste() y un método toString() que ponga una cabecera que diga quién es el remitente, a quién va dirigido y luego el texto del telegrama. Crear un programa que use la clase telegrama donde el usuario introduzca los datos de cada telegrama y estos queden insertados ordenados por coste en una lista. Se debe crear una nueva clase: ComparadorTelegrama que implemente Comparator para mantener la lista ordenada

Práctica 29: El usuario debe ir introduciendo el nombre de cada partido político y el número de votos que ha tenido. Después se le mostrará un menú donde puede elegir introducir el nombre de un partido y que le muestre los votos que ha tenido. Así como introducir un límite inferior de votos y uno superior para que se le muestren los partidos y los votos que han tenido que estén dentro de los límites. Utilizar la estructura dinámica más apropiada para este caso. Justificar su elección entre comentarios

Argumentos Variables

Pongamos por ejemplo que queremos crear un constructor para una clase llamada Matriz

Las Matrices, pueden ser de n filas * m columnas (NxM) y quisiéramos que esa clase Matriz nos valga para cualquier dimensión de matriz. Sin embargo eso debiera estar en la elección del usuario y que nos pase los datos pertinentes para esa matriz en el propio constructor:

```
class Matriz{
    int columnas;
    int filas;
    int datos[][];
    public Matriz(int fil, int col, int ...valores){
        columnas = col;
        filas = fil;
        datos = new int[filas][columnas];
        int i=0,j=0;

        for (int val : valores) {
            datos[i][j] = val;
            j++;
            if( j >= columnas){
                i++;
                j=0;
            }
        }
    }
}
```

Observamos que el constructor tiene declarados 3 parámetros: **fil, col** que son enteros y **valores** que es un array de enteros Veamos posibles llamadas a ese constructor:

```
        Matriz m = new Matriz(2,3,1,0,0,0,1,0);

        Matriz a = new Matriz(3,2,1,2,-1,0,-3,-1);
        Matriz b = new Matriz(2,3,2,0,1,-5,2,3);
```

m es la matriz:
1 0 0
0 1 0

a es la matriz:
 1   2
-1 0
-3 -1

b es la matriz:
 2 0 1
-5 2 3

Lo bueno es que podemos hacer matrices de la dimensión que queramos y únicamente necesitamos un constructor.

Lo único que se tiene que tener en cuenta para los varargs (argumentos variables) es que **debe ser el último parámetro que se declare**, en otro caso no sería válido. Ej.

```
//Declaración válida:
int suma_a(String cadena, int... numero){...}

//Declaración NO válida:
int suma_a(int... numero, String cadena){...}
```

Anexo: ArrayList.toArray() Arrays.asList()

El siguiente código muestra ordenación mediante sort() tanto de collections como de arrays y como transformar de un array a una lista y una lista a un array

```
class comparadorPersonas implements Comparator<Persona> {

    @Override
    public int compare(Persona p1, Persona p2) {
        return p1.edad - p2.edad;
    }
}

public class ToArrayToList {
    public static void main(String[] args) {
        ArrayList<Persona> al = new ArrayList<Persona>();
        Persona array[] = new Persona[2];
        al.add(new Hombre("franc", "nazer", 34, 180, 75));
        al.add(new Mujer("ana", "alcazar", 24, 172, 52));

        array[0] = new Mujer("eva", "linares", 25, 165, 55);
        array[1] = new Hombre("marco", "armas", 22, 175, 80);

        /*Conversión de lista a array */
        Persona array1[] = al.toArray(new Persona[al.size()]);
        for (Persona persona : array1) {
            System.out.println((Persona) persona);
        }
        System.out.println("");
        /*Conversión de array a arraylista*/
        ArrayList<Persona> al1 = new ArrayList<Persona>(Arrays.asList(array));
        for (Persona persona : al1) {
            System.out.println(persona);
        }
        System.out.println("\n\n");

        /*Ordenando mediante Arrays.sort*/
        Arrays.sort(array1,new comparadorPersonas());
        for (Persona persona : array1) {
            System.out.println(persona);
        }
        System.out.println("");
        /*Ordenando mediante Collections.sort*/
        Collections.sort(al1,new comparadorPersonas());
        for (Persona persona : al1) {
            System.out.println(persona);
        }
    }
```