TEMA 6: Estrucuturas de Almacenamiento.

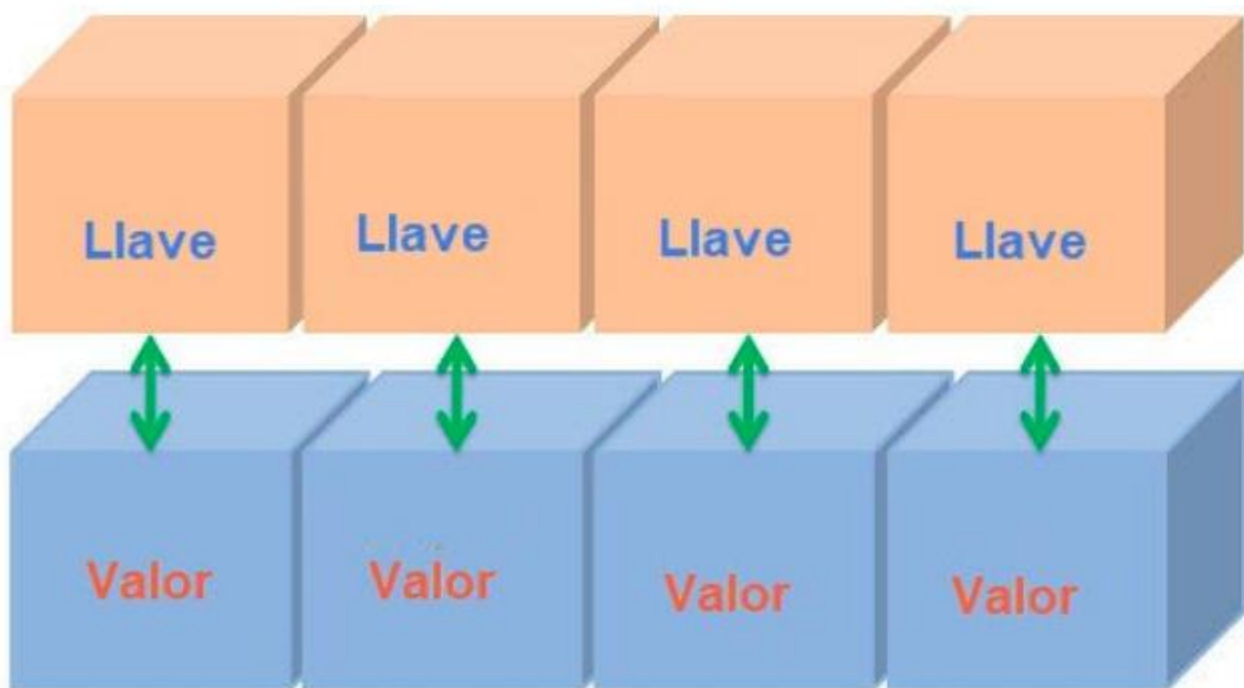
6.A. Introducción a las estructuras de almacenamiento.

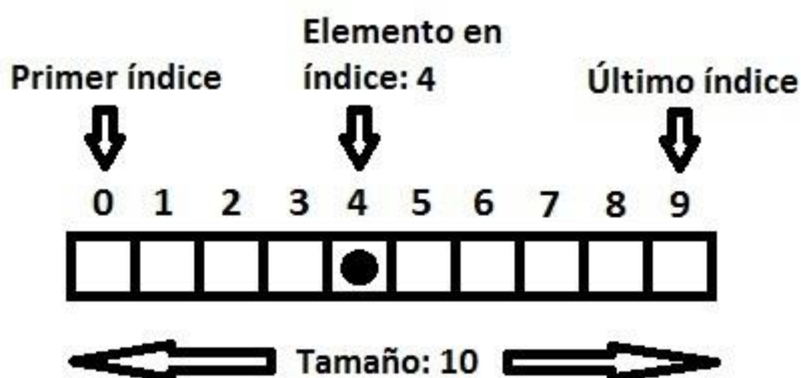
6.B. Cadenas de caracteres.

6.C. Expresiones regulares.

6.D. Arrays Unidimensionales.

6.E. Arrays multidimensionales.





6.A. Introducción a las estructuras de almacenamiento.

1. Introducción.

Cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables. Manejar los datos de un único pedido en una aplicación puede ser relativamente sencillo, pues un pedido está compuesto por una serie de datos y eso simplemente se traduce en varias variables. Pero, ¿qué ocurre cuando en una aplicación tenemos que gestionar varios pedidos a la vez?

Lo mismo ocurre en otros casos. 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**, es decir, datos que están compuestos a su vez de varios datos más simples. Por ejemplo, un pedido está compuesto por varios datos, los datos podrían ser el cliente que hace el pedido, la dirección de entrega, la fecha requerida de entrega y los artículos del pedido.

Los datos compuestos son un tipo de estructura de datos, y en realidad ya los has manejado. **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. Pero, a veces, los datos tienen estructuras aún más complejas, y son necesarias soluciones adicionales.

Aquí podrás aprender esas soluciones adicionales. Esas soluciones consisten básicamente en la capacidad de poder manejar varios datos del mismo o diferente tipo de forma dinámica y flexible.

En otras ocasiones nos encontramos otro tipo de dificultades. Por ejemplo, ¿cómo almacenarías en memoria un listado de números del que tienes que extraer el valor máximo? Seguro que te resultaría fácil. Pero, ¿y si el listado de números no tiene un tamaño fijo, sino que puede variar en tamaño de forma dinámica? Entonces la cosa se complica.

Un listado de números que aumenta o decrece en tamaño es una de las cosas que aprenderás a utilizar aquí, utilizando estructuras de datos. Pasaremos por alto las clases y los objetos, pues ya los has visto con anterioridad, pero debes saber que las clases en sí mismas son la evolución de un tipo de estructuras de datos conocidas como datos compuestos (también llamadas registros). Las clases, además de aportar la ventaja de agrupar datos relacionados entre sí en una misma estructura (característica aportada por los datos compuestos), permiten agregar métodos que manejen dichos datos, ofreciendo una herramienta de programación sin igual. Pero todo esto ya lo sabías.

Las estructuras de almacenamiento, en general, se pueden clasificar de varias formas. Por ejemplo, atendiendo a si pueden almacenar datos de diferente tipo, o si solo pueden almacenar datos de un solo tipo, se pueden distinguir:

- Estructuras con capacidad de **almacenar varios datos del mismo tipo:** varios números, varios caracteres, etc. Ejemplos de estas estructuras son los arrays, las cadenas de caracteres, las listas y los conjuntos.

- Estructuras con capacidad de **almacenar varios datos de distinto tipo:** números, fechas, cadenas de caracteres, etc., todo junto dentro de una misma estructura. Ejemplos de este tipo de estructuras son las clases.

Otra forma de clasificar las estructuras de almacenamiento va en función de si pueden o no cambiar de tamaño de forma dinámica:

- **Estructuras cuyo tamaño se establece en el momento de la creación o definición** y su tamaño no puede variar después. Ejemplos de estas estructuras son los **arrays** y las matrices (arrays multidimensionales).

- **Estructuras cuyo tamaño es variable (conocidas como estructuras dinámicas)**. Su tamaño crece o decrece según las necesidades de forma dinámica. Es el caso de las listas, árboles, conjuntos y, como veremos también, el caso de algunos tipos de cadenas de caracteres. Serán tratadas en otra unidad de contenidos posterior.

Por último, atendiendo a la forma en la que los datos se ordenan dentro de la estructura, podemos diferenciar varios tipos de estructuras:

- **Estructuras que no se ordenan de por sí**, y debe ser el programador el encargado de ordenar los datos si fuera necesario. Un ejemplo de estas estructuras son los arrays.

- **Estructuras ordenadas**. Se trata de estructuras que, al incorporar un dato nuevo a todos los datos existentes, este se almacena en una posición concreta que irá en función del orden. El orden establecido en la estructura puede variar dependiendo de las necesidades del programa: alfabético, orden numérico de mayor a menor, momento de inserción, etc. Serán tratadas en otra unidad de contenidos posterior.

Todavía no conoces mucho de las estructuras, y probablemente todo te suena raro y extraño. No te preocupes, poco a poco irás descubriéndolas. Verás que son sencillas de utilizar y muy cómodas.

6.B. Cadenas de caracteres.

1. Cadenas de caracteres.

    1.1. Operaciones avanzadas con cadenas de caracteres (I).

    1.2. Operaciones avanzadas con cadenas de caracteres (II).

    1.3. Operaciones avanzadas con cadenas de caracteres (III).

    1.4. Operaciones avanzadas con cadenas de caracteres (IV).

    1.5. Operaciones avanzadas con cadenas de caracteres (V).

1. Cadenas de caracteres.

Probablemente, una de las cosas que más utilizarás cuando estés programando en cualquier lenguaje de programación son las cadenas de caracteres. Las cadenas de caracteres son estructuras de almacenamiento que permiten almacenar una secuencia de caracteres de casi cualquier longitud. Y la pregunta ahora es, ¿qué es un carácter?

En Java y en todo lenguaje de programación, y por ende, en todo sistema informático, los caracteres se codifican como secuencias de bits que representan a los símbolos usados en la comunicación humana. Estos símbolos pueden ser letras, números, símbolos matemáticos e incluso ideogramas y pictogramas.

Para saber más

Si quieres puedes profundizar en la codificación de caracteres leyendo el siguiente artículo de la Wikipedia.

Codificación de caracteres.

La forma más habitual de ver escrita una cadena de caracteres es como un literal de cadena. Consiste simplemente en una secuencia de caracteres entre comillas dobles, por ejemplo:

```
"Ejemplo de cadena de caracteres".
```

En Java, los literales de cadena son en realidad instancias de la clase `String`, lo cual quiere decir que, por el mero hecho de escribir un literal, se creará una instancia de dicha clase. Esto da mucha flexibilidad, puesto que permite crear cadenas de muchas formas diferentes, pero obviamente consume mucha memoria. La forma más habitual es crear una cadena partiendo de un literal:

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

En este caso, el literal de cadena situado a la derecha del igual es en realidad una instancia de la clase `String`. Al realizar esta asignación hacemos que la variable `cad` se convierta en una referencia al objeto ya creado. Otra forma de crear una cadena es usando el operador `new` y un constructor, como por ejemplo:

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

Cuando se crean las cadenas de esta forma, se realiza una copia en memoria de la cadena pasada por parámetro. La nueva instancia de la clase `String` hará referencia por tanto a la copia de la cadena, y no al original.

Reflexiona

Fijate en la siguiente línea de código, ¿cuántas instancias de la clase `String` ves?

```
String cad="Ejemplo de cadena 1"; cad="Ejemplo de cadena 2";
cad=new String("Ejemplo de cadena 3");
```

Solución

Pues en realidad hay 4 instancias. La primera instancia es la que se crea con el literal de cadena `"Ejemplo de cadena 1"`. El segundo literal, `"Ejemplo de cadena 2"`, da lugar a otra instancia diferente a la anterior. El tercer literal, `"Ejemplo de cadena 3"`, es también nuevamente otra instancia de `String` diferente. Y por último, al crear una nueva instancia de la clase `String` a través del operador `new`, se crea un nuevo objeto `String` copiando para ello el contenido de la cadena que se le pasa por parámetro, con lo que aquí tenemos la cuarta instancia del objeto `String` en solo una línea.

1.1. Operaciones avanzadas con cadenas de caracteres (I).

¿Qué operaciones puedes hacer con una cadena? Muchas más de las que te imaginas. Empezaremos con la operación más sencilla: la concatenación. La concatenación es la unión de dos cadenas, para formar una sola. En Java es muy sencillo, pues sólo tienes que utilizar el operador de concatenación (signo de suma):

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

En la operación anterior se está creando una nueva cadena, resultado de unir dos cadenas: una cadena con el texto "¡`Bien`", y otra cadena con el texto "`venido`!". La segunda línea de código muestra por la salida estándar el resultado de la concatenación. El resultado de su ejecución será que aparecerá el texto `"¡Bienvenido!"` por la pantalla.

Otra forma de usar la concatenación, que ilustra que cada literal de cadena es a su vez una instancia de la clase `String`, es usando el método `concat` del objeto `String`:

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

Fíjate bien en la expresión anterior, pues genera el mismo resultado que la primera opción y en ambas participan tres instancias de la clase `String`. Una instancia que contiene el texto "¡Bien", otra instancia que contiene el texto "venido!", y otra que contiene el texto "¡Bienvenido!". La tercera cadena se crea nueva al realizar la operación de concatenación, sin que las otras dos hayan desaparecido. Pero no te preocupes por las otras dos cadenas, pues se borrarán de memoria cuando el recolector de basura detecte que ya no se usan.

Fíjate además, que se puede invocar directamente un método de la clase `String`, posponiendo el método al literal de cadena. Esto es una señal de que al escribir un literal de cadena, se crea una instancia del objeto inmutable `String`.

Pero no solo podemos concatenar una cadena a otra cadena. Gracias al método `toString()` podemos concatenar cadenas con literales numéricos e instancias de otros objetos sin problemas.

El método `toString()` es un método disponible en todas las clases de Java. Su objetivo es simple, 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. Lo de convertir, no siempre es posible, hay clases fácilmente convertibles a texto, como es la clase `Integer`, por ejemplo, y otras que no se pueden convertir, y que el resultado de invocar el método `toString()` es información relativa a la instancia.

La gran ventaja de la concatenación es que el método `toString()` se invocará automáticamente, sin que tengamos que especificarlo, por ejemplo:

```
Integer i1=new Integer (1223); // La instancia i1 de la clase
Integer contiene el número 1223.
System.out.println("Número: " + i1); // Se mostrará por pantalla
el texto "Número: 1223"
```

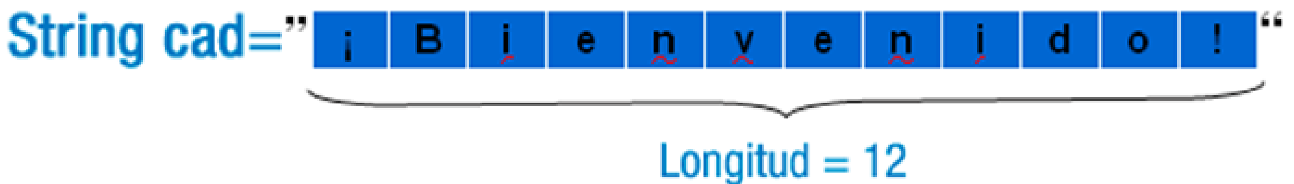
En el ejemplo anterior, se ha invocado automáticamente `i1.toString()`, para convertir el número a cadena. Esto se realizará para cualquier instancia de clase concatenada, pero cuidado, como se ha dicho antes, no todas las clases se pueden convertir a cadenas.

1.2. Operaciones avanzadas con cadenas de caracteres (II).

Vamos a continuar revisando las operaciones que se pueden realizar con cadenas. Como verás las operaciones a realizar se complican un poco a partir

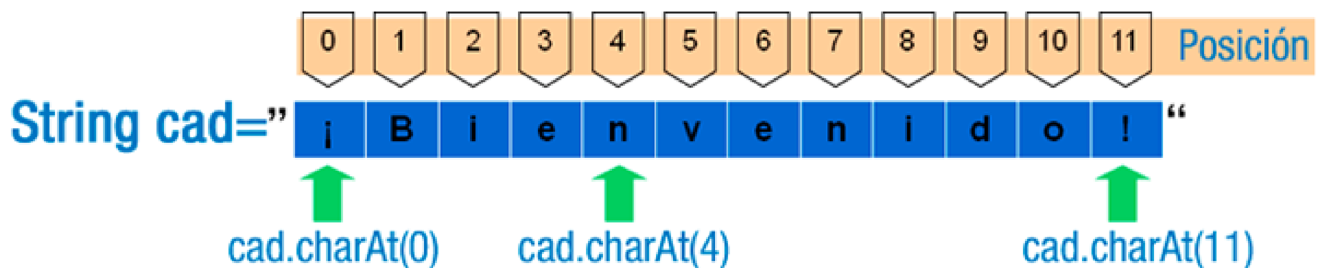
de ahora. En todos los ejemplos la variable `cad` contiene la cadena `"¡Bienvenido!"`, como se muestra en las imágenes.

- `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.

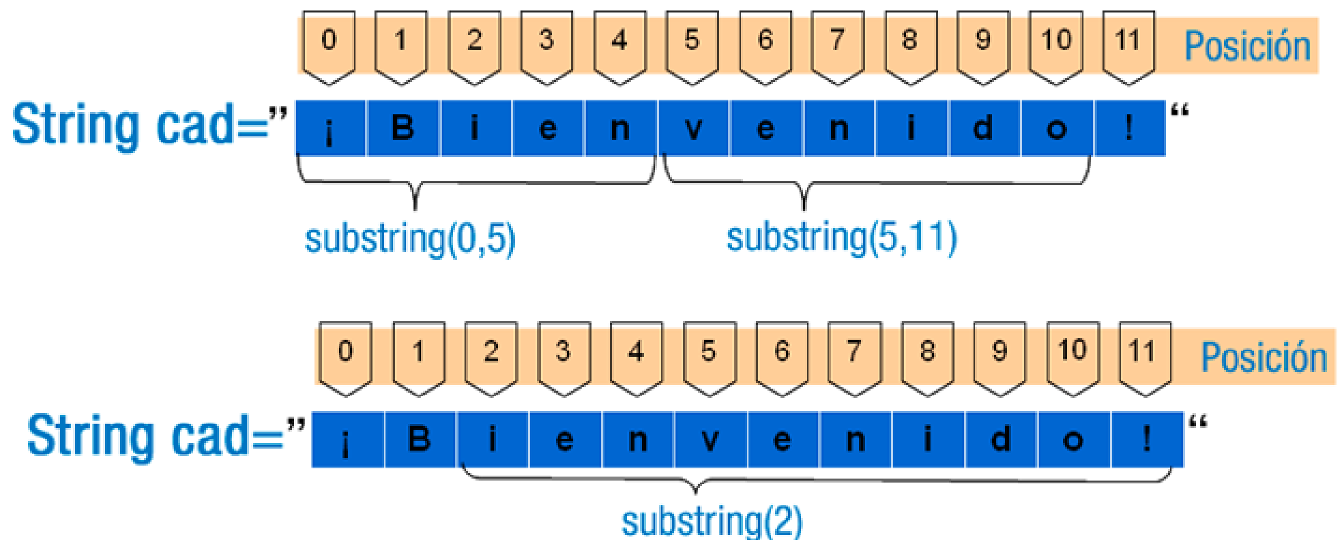


- `char charAt(int pos)`. Retorna el carácter ubicado en la posición pasada por parámetro. El carácter obtenido de dicha posición será almacenado en un tipo de dato `char`. Las posiciones se empiezan a contar desde el 0 (y no desde el 1), y van desde 0 hasta longitud - 1.Por ejemplo, el código siguiente mostraría por pantalla el carácter "v":

```
char t = cad.charAt(5);
System.out.println(t);
```



- `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`. Por ejemplo, si pusiéramos `cad.substring(0,5)` en nuestro programa, sobre la variable `cad` anterior, dicho método devolvería la subcadena `"¡Bien"` tal y como se muestra en la imagen.



- `String substring (int beginIndex)`. Cuando al método `substring` solo le proporcionamos un parámetro, extraerá una cadena que comenzará en el carácter con posición `beginIndex` e irá hasta el final de la cadena. En el siguiente ejemplo se mostraría por pantalla la cadena "ienvenido!":

```
String subcad = cad.substring(2);
System.out.println(subcad);
```

Otra operación muy habitual es la conversión de número a cadena y de cadena a número. Imagínate que un usuario introduce su edad. Al recoger la edad desde la interfaz de usuario, capturarás generalmente una cadena, pero, ¿cómo compruebas que la edad es mayor que 0? Para poder realizar esa comprobación tienes que pasar la cadena a número. Empezaremos por ver como se convierte un número a cadena.

Los números generalmente se almacenan en memoria como números binarios, es decir, secuencias de unos y ceros con los que se puede operar (sumar, restar, etc.). No debes confundir los tipos de datos que contienen números (`int`, `short`, `long`, `float` y `double`) con las secuencias de caracteres que representan un número. No es lo mismo 123 que "`123`", el primero es un número y el segundo es una cadena formada por tres caracteres: '`1`', '`2`' y '`3`'.

Convertir un número a cadena es fácil desde que existe, para todas las clases Java, el método `toString()`. Gracias a ese método podemos hacer cosas como las siguientes:

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

El resultado del código anterior es que se mostrará por pantalla "`Número cinco: 5`", y no dará ningún error. Esto es posible gracias a que Java convierte el número 5 a su "clase envoltorio" (wrapper class) correspondiente (`Integer`, `Float`, `Double`, etc.), y después ejecuta automáticamente el método `toString()` de dicha clase.

Reflexiona

¿Cuál crees que será el resultado de poner `System.out.println("A"+5f)`? Pruébalo y recuerda: no olvides indicar el tipo de literal (f para lo literales de números flotantes, y d para los literales de números dobles), así obtendrás el resultado esperado y no algo diferente.

1.3. Operaciones avanzadas con cadenas de caracteres (III).

¿Cómo comprobarías que la cadena "3" es mayor que 0? No puedes comparar directamente una cadena con un número, así que necesitarás aprender cómo convertir cadenas que contienen números a tipos de datos numéricos (`int`, `short`, `long`, `float` o `double`). Esta es una operación habitual en todos los lenguajes de programación, y Java, para este propósito, ofrece los métodos `valueOf`, existentes en todas las clases envoltorio descendientes de la clase `Number`: `Integer`, `Long`, `Short`, `Float` y `Double`.

Veamos un ejemplo de su uso para un número de doble precisión, para el resto de las clases es similar:

```
String c="1234.5678";
double n;
try {
     n=Double.valueOf(c).doubleValue();
} catch (NumberFormatException e)
{  /* Código a ejecutar si no se puede convertir */  }
```

Fíjate en el código anterior, en él puedes comprobar cómo la cadena c contiene en su interior un número, pero escrito con dígitos numéricos (caracteres). El código escrito está destinado a transformar la cadena en número, usando el método `valueOf`. Este método lanzará la excepción `NumberFormatException` si no consigue convertir el texto a número. De momento no te preocupes demasiado acerca de este nuevo concepto, aunque es probable que ya te haya sucedido; más adelante dedicaremos todo un tema al tratamiento de excepciones en Java. En el siguiente código, tienes un ejemplo más completo, donde aparecen también ejemplos para las otros tipos numéricos:

```
import javax.swing.JOptionPane;
/**
 * Ejemplo en el que se muestra la conversión de varias cadenas de
texto, que
 * contienen números, a números.
 * @author Salvador Romero Villegas
 */

public class EjemplosConversionStringANumero {

    boolean operacionCancelada;
```

```
    /**
     * Constructor de la clase.
     */
    public EjemplosConversionStringANumero() {
        setOperacionCancelada(false);
    }

    /**
     * Método que permite comprobar si la última operación tipo
Pedir ha sido
     * cancelada.
     * @return true si la última operación realizada ha sido
cancelada, false
     * en otro caso.
     */
    public boolean isOperacionCancelada() {
        return operacionCancelada;
    }

    /**
     * Método que permite cambiar el estado de la variable privada
     * operacionCancelada. Este método es privado y solo debe
usarse desde
     * un método propio de esta clase.
     * @param operacionCancelada True o false, el nuevo estado
para la variable.
     */
    private void setOperacionCancelada(boolean operacionCancelada)
{
        this.operacionCancelada = operacionCancelada;
    }

    /**
     * Clase que pide al usuario que introduzca un número. El
número esperado
     * es un número de doble precisión, en cualquiera de sus
formatos. Admitirá
     * números como: 2E10 (2*10^10); 2,45; etc.
     * @param titulo
     * @param mensaje
     * @return
     */
    public double PedirNumeroDouble(String titulo, String mensaje)
{
        double d = 0;
        setOperacionCancelada(false);
        boolean NumeroValido = false;
        do {

            String s = (String) JOptionPane.showInputDialog(null,
mensaje,
```

```
                    titulo, JOptionPane.PLAIN_MESSAGE, null, null,
"");

            if (s != null) {
                try {
                    d = Double.valueOf(s).doubleValue();
                    NumeroValido = true;
                } catch (NumberFormatException e) {
                    JOptionPane.showMessageDialog(null, "El número
introducido no es válido.", "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
            else { NumeroValido=true; // Cancelado
                   setOperacionCancelada(true);
            }
        } while (!NumeroValido);
        return d;
    }

    /**
     * Clase que pide al usuario que introduzca un número. El
número esperado
     * es un número de precisión sencilla, en cualquiera de sus
formatos.
     * @param titulo
     * @param mensaje
     * @return
     */
    public float PedirNumeroFloat (String titulo, String mensaje)
{
        float d = 0;
        setOperacionCancelada(false);
        boolean NumeroValido = false;
        do {

            String s = (String) JOptionPane.showInputDialog(null,
mensaje,
                    titulo, JOptionPane.PLAIN_MESSAGE, null, null,
"");

            if (s != null) {
                try {
                    d = Float.valueOf(s).floatValue();
                    NumeroValido = true;
                } catch (NumberFormatException e) {
                    JOptionPane.showMessageDialog(null, "El número
introducido no es válido.", "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
            else { NumeroValido=true; // Cancelado
                   setOperacionCancelada(true);
            }
```

```
        } while (!NumeroValido);
        return d;
    }

    /**
     * Clase que pide al usuario que introduzca un número. El
número esperado
     * es un número entero.
     * @param titulo
     * @param mensaje
     * @return
     */
    public int PedirNumeroInteger (String titulo, String mensaje)
{
        int d = 0;
        setOperacionCancelada(false);
        boolean NumeroValido = false;
        do {

            String s = (String) JOptionPane.showInputDialog(null,
mensaje,
                    titulo, JOptionPane.PLAIN_MESSAGE, null, null,
"");

            if (s != null) {
                try {
                    d = Integer.valueOf(s).intValue();
                    NumeroValido = true;
                } catch (NumberFormatException e) {
                    JOptionPane.showMessageDialog(null, "El número
introducido no es válido.", "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
            else { NumeroValido=true; // Cancelado
                   setOperacionCancelada(true);
            }
        } while (!NumeroValido);
        return d;
    }

    /**
     * Clase que pide al usuario que introduzca un número. El
número esperado
     * es un número entero.
     * @param titulo
     * @param mensaje
     * @return
     */
    public long PedirNumeroLong (String titulo, String mensaje) {
        long d = 0;
        setOperacionCancelada(false);
        boolean NumeroValido = false;
```

```
        do {

            String s = (String) JOptionPane.showInputDialog(null,
mensaje,
                    titulo, JOptionPane.PLAIN_MESSAGE, null, null,
"");

            if (s != null) {
                try {
                    d = Long.valueOf(s).longValue();
                    NumeroValido = true;
                } catch (NumberFormatException e) {
                    JOptionPane.showMessageDialog(null, "El número
introducido no es válido.", "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
            else { NumeroValido=true; // Cancelado
                   setOperacionCancelada(true);
            }
        } while (!NumeroValido);
        return d;
    }

     /**
     * Clase que pide al usuario que introduzca un número. El
número esperado
     * es un número entero corto.
     * @param titulo
     * @param mensaje
     * @return
     */
    public short PedirNumeroShort (String titulo, String mensaje)
{
        short d = 0;
        setOperacionCancelada(false);
        boolean NumeroValido = false;
        do {

            String s = (String) JOptionPane.showInputDialog(null,
mensaje,
                    titulo, JOptionPane.PLAIN_MESSAGE, null, null,
"");

            if (s != null) {
                try {
                    d = Short.valueOf(s).shortValue();
                    NumeroValido = true;
                } catch (NumberFormatException e) {
                    JOptionPane.showMessageDialog(null, "El número
introducido no es válido.", "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
```

```
            else { NumeroValido=true; // Cancelado
                   setOperacionCancelada(true);
            }
        } while (!NumeroValido);
        return d;
    }
}
```

Seguro que ya te vas familiarizando con este embrollo y encontrarás interesante todas estas operaciones. Ahora te planteamos otro reto: imagina que tienes que mostrar el precio de un producto por pantalla. Generalmente, si un producto vale, por ejemplo 3,3 euros, el precio se debe mostrar como `"3,30 €"`, es decir, se le añade un cero extra al final para mostrar las centésimas. Con lo que sabemos hasta ahora, usando la concatenación en Java, podemos conseguir que una cadena se concatene a un número flotante, pero el resultado no será el esperado. Prueba el siguiente código:

```
float precio=3.3f;
System.out.println("El precio es: "+precio+"€");
```

Si has probado el código anterior, habrás comprobado que el resultado no muestra `"3,30 €"` sino que muestra `"3,3 €"`. ¿Cómo lo solucionamos? Podríamos dedicar bastantes líneas de código hasta conseguir algo que realmente funcione, pero no es necesario, dado que Java y otros lenguajes de programación (como C), disponen de lo que se denomina formateado de cadenas. En Java podemos "formatear" cadenas a través del método estático `format` disponible en el objeto `String`. Este método permite crear una cadena proyectando los argumentos en un formato específico de salida. Lo mejor es verlo con un ejemplo, veamos cuál sería la solución al problema planteado antes:

```
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`, situada como segundo argumento, es la variable que se proyectará en la salida siguiendo un formato concreto. Seguro que te preguntarás, ¿qué es `"%.2f"`? Pues es un especificador de formato, e indica cómo se deben formatear o proyectar los argumentos que hay después de la cadena de formato en el método `format`.

Debes conocer

Es necesario que conozcas bien el método format y los especificadores de formato. Por ese motivo, te pedimos que estudies el siguiente anexo:

Anexo I.- Formateado de cadenas en Java.

Sintaxis de las cadenas de formato y uso del método `format`.

En Java, el método estático `format` de la clase `String` permite formatear los datos que se muestran al usuario o la usuaria de la aplicación. El método `format` tiene los siguientes argumentos:

- Cadena de formato. Cadena que especifica cómo será el formato de salida, en ella se mezclará texto normal con especificadores de formato, que indicarán cómo se debe formatear los datos.

- Lista de argumentos. Variables que contienen los datos cuyos datos se formatearán. Tiene que haber tantos argumentos como especificadores de formato haya en la cadena de formato.

Los especificadores de formato comienzan siempre por "%", es lo que se denomina un carácter de escape (carácter que sirve para indicar que lo que hay a continuación no es texto normal, sino algo especial). El especificador de formato debe llevar como mínimo el símbolo "%" y un carácter que indica la conversión a realizar, por ejemplo "%d".

La conversión se indica con un simple carácter, y señala al método `format` cómo debe ser formateado el argumento. Dependiendo del tipo de dato podemos usar unas conversiones u otras. Veamos las conversiones más utilizadas:

Listado de conversiones más utilizada y ejemplos.				
Tipo de conversión	**Especificación de formato**	**Tipos de datos aplicables**	**Ejemplo**	**Resultado del ejemplo**

Valor lógico o booleano.	"%b" o "%B"	`Boolean` (cuando se usan otros tipos de datos siempre lo formateará escribiendo `true`).	`boolean b=true;`
			`String d= String.format("Resultado: %b", b);`
			`System.out.println (d);`
				Resultado: `true`

Cadena de caracteres.	"%s" o "%S"	Cualquiera, se convertirá el objeto a cadena si es posible (invocando el	`String cad="hola mundo";`
			`String d= String.format("Resultado: %s", cad);`
			`System.out.println (d);`
				Resultado: `hola mundo`

Listado de conversiones más utilizada y ejemplos.				
Tipo de conversión	**Especificación de formato**	**Tipos de datos aplicables**	**Ejemplo**	**Resultado del ejemplo**
		método `toString`).		

Entero decimal	`"%d"`	Un tipo de dato entero.	`int i=10;`
			`String d= String.format("Resultado: %d", i);`
			`System.out.println (d);`
				Resultado: `10`

Número en notación científica	`"%e"` o `"%E"`	Flotantes simples o dobles.	`double i=10.5;`
			`String d= String.format("Resultado: %E", i);`
			`System.out.println (d);`
				Resultado: `1.050000E+01`

Número decimal	`"%f"`	Flotantes simples o dobles.	`float i=10.5f;`
			`String d= String.format("Resultado: %f", i);`
			`System.out.println (d);`
				Resultado: `10,500000`

Número en notación científica o decimal (lo más corto)	`"%g"` o `"%G"`	Flotantes simples o dobles. El número se mostrará como decimal o en notación científica dependiendo de lo que sea más corto.	`double i=10.5;`
			`String d= String.format("Resultado: %g", i);`
			`System.out.println (d);`
				Resultado: `10.5000`

Ahora que ya hemos visto alguna de las conversiones existentes (las más importantes), veamos algunos modificadores que se le pueden aplicar a las conversiones, para ajustar como queremos que sea la salida. Los modificadores se sitúan entre el carácter de escape ("%") y la letra que indica el tipo de conversión (d, f, g, etc.).

Podemos especificar, por ejemplo, el número de caracteres que tendrá como mínimo la salida de una conversión. Si el dato mostrado no llega a ese ancho en caracteres, se rellenará con espacios (salvo que se especifique lo contrario):

```
%[Ancho]Conversión
```

El hecho de que esté entre corchetes significa que es opcional. Si queremos por ejemplo que la salida genere al menos 5 caracteres (poniendo espacios delante) podríamos ponerlo así:

```
String.format ("%5d",10);
```

Se mostrará el "10" pero también se añadirán 3 espacios delante para rellenar. Este tipo de modificador se puede usar con cualquier conversión.

Cuando se trata de conversiones de tipo numéricas con decimales, solo para tipos de datos que admitan decimales, podemos indicar también la precisión, que será el número de decimales mínimos que se mostrarán:

```
%[Ancho][.Precisión]Conversión
```

Como puedes ver, tanto el ancho como la precisión van entre corchetes, los corchetes no hay que ponerlos, solo indican que son modificaciones opcionales. Si queremos, por ejemplo, que la salida genere 3 decimales como mínimo, podremos ponerlo así:

```
String.format ("%.3f",4.2f);
```

Como el número indicado como parámetro solo tiene un decimal, el resultado se completará con ceros por la derecha, generando una cadena como la siguiente: `"4,200"`.

Una cadena de formato puede contener varios especificadores de formato y varios argumentos. Veamos un ejemplo de una cadena con varios especificadores de formato:

```
String np="Lavadora";
int u=10;
float ppu = 302.4f;
float p=u*ppu;
String output=String.format("Producto: %s; Unidades: %d; Precio
por unidad: %.2f €; Total: %.2f €", np, u, ppu, p);
System.out.println(output);
```

Cuando el orden de los argumentos es un poco complicado, porque se reutilizan varias veces en la cadena de formato los mismos argumentos, se puede recurrir a los índices de argumento. Se trata de especificar la posición del argumento a utilizar, indicando la posición del argumento (el primer argumento sería el 1 y no el 0) seguido por el símbolo del dólar ("$"). El índice se ubicaría al comienzo del especificador de formato, después del porcentaje, por ejemplo:

```
int i=10;
int j=20;
String d=String.format("%1$d multiplicado por %2$d (%1$dx%2$d) es
%3$d",i,j,i*j);
```

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

El ejemplo anterior mostraría por pantalla la cadena `"10 multiplicado por 20 (10x20) es 200"`. Los índices de argumento se pueden usar con todas las conversiones, y es compatible con otros modificadores de formato (incluida la precisión).

Para saber más

Si quieres profundizar en los especificadores de formato puedes acceder a la siguiente página (en inglés), donde encontrarás información adicional acerca de la sintaxis de los especificadores de formato en Java:

[Sintaxis de los especificadores de formato.]()

1.4. Operaciones avanzadas con cadenas de caracteres (IV).

¿Cómo puedo comprobar si dos cadenas son iguales? ¿Qué más operaciones ofrece Java sobre las cadenas? Java ofrece un montón de operaciones más sobre. En la siguiente tabla puedes ver las operaciones más importantes. En todos los ejemplos expuestos, las variables `cad1`, `cad2` y `cad3` son cadenas ya existentes, y la variable `num` es un número entero mayor o igual a cero.

Métodos importantes de la clase String.	
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)` `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.

Métodos importantes de la clase String.	
Método.	**Descripción**
`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)` `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`".

1.5. Operaciones avanzadas con cadenas de caracteres (V).

¿Sabes cuál es el principal problema de las cadenas de caracteres? 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 un mutable, y permite una mayor optimización de la memoria. También existe la clase `StringBuffer`, pero consume mayores recursos al estar pensada para aplicaciones multi-hilo, por lo que en nuestro caso nos centraremos en la primera.

Pero, ¿en que se diferencian `StringBuilder` de la clase `String`? Pues básicamente en que la clase StringBuilder permite modificar la cadena que contiene, mientras que la clase String no. Como ya se dijo antes, al realizar operaciones complejas se crea una nueva instancia de la clase `String`.

Veamos un pequeño ejemplo de uso de esta clase. En el ejemplo que vas a ver, se parte de una cadena con errores, que modificaremos para ir haciéndola legible. Lo primero que tenemos que hacer es crear la instancia de esta clase. Se puede inicializar de muchas formas, por ejemplo, partiendo de un literal de cadena:

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

Y ahora, usando los métodos `append` (insertar al final), `insert` (insertar una cadena o carácter en una posición específica), `delete` (eliminar los caracteres que hay entre dos posiciones) y replace (reemplazar los caracteres que hay entre dos posiciones por otros diferentes), rectificaremos la cadena anterior y la haremos correcta:

1. `strb.delete(6,8);` Eliminamos las 'uu' que sobran en la cadena. La primera 'u' que sobra está en la posición 6 (no olvides contar el espacio), y la última 'u' a eliminar está en la posición 7. Para eliminar dichos caracteres de forma correcta hay que pasar como primer argumento la posición 6 (posición inicial) y como segundo argumento la posición 8 (posición contigua al último carácter a eliminar), dado que la posición final no indica el último carácter a eliminar, sino el carácter justo posterior al último que hay que eliminar (igual que ocurría con el método `substring`).
2. `strb.append ("!");` Añadimos al final de la cadena el símbolo de cierre de exclamación.
3. `strb.insert (0,"¡");` Insertamos en la posición 0, el símbolo de apertura de exclamación.
4. `strb.replace (3,5,"la");` Reemplazamos los caracteres 'al' situados entre la posición inicial 3 y la posición final 4, por la cadena 'la'. En este método ocurre igual que en los métodos `delete` y `substring`, en vez de indicar como posición final la posición 4, se debe indicar justo la posición contigua, es decir 5.

`StringBuilder` contiene muchos métodos de la clase `String` (`charAt, indexOf, lenght, substring, replace, setCharAt`, etc.), pero no todos, pues son clases diferentes con funcionalidades realmente diferentes.

Debes conocer

En la siguiente página puedes encontrar más información (en inglés) sobre como utilizar la clase `StringBuilder`.

Uso de la clase StringBuilder.

6.C. Expresiones regulares.

1. Expresiones regulares.

En la Wikipedia podemos que se menciona que en los lenguajes formales (en general lo son todos los lenguajes de programación y los de marcas), **una expresión regular**, también conocida como **regex**, **regexp** o **expresión racional**, es una secuencia de caracteres que forma un patrón de búsqueda, principalmente utilizada para la búsqueda de patrones de cadenas de caracteres u operaciones de sustituciones. Concretamente en la Wikipedia se muestra el siguiente ejemplo:

el grupo formado por las cadenas Handel, Händel y Haendel se describe con el patrón "H(a|ä|ae)ndel".

Esto proporcionará, tal y como se indica, una herramienta versátil, flexible y muy potente para permitir buscar o reconocer ciertos bloques de texto dentro de una cadena de texto a analizar.

1.1. Expresiones regulares (I).

¿Tienen algo en común todos los números de DNI y de NIE? ¿Podrías hacer un programa que verificara si un DNI o un NIE es correcto? Seguro que sí. Si te fijas, los números de DNI y los de NIE tienen una estructura fija: X1234567Z (en el caso del NIE) y 1234567Z (en el caso del DNI). Ambos siguen un patrón que podría describirse como: una letra inicial opcional (solo presente en los NIE), seguida de una secuencia numérica y finalizando con otra letra. ¿Fácil no?

Pues esta es la función de las expresiones regulares: **permitir comprobar si una cadena sigue o no un patrón preestablecido**. Las expresiones regulares son un mecanismo para describir esos patrones, y se construyen de una forma relativamente sencilla. Existen muchas librerías diferentes para trabajar con expresiones regulares, y casi todas siguen, más o menos, una sintaxis similar, con ligeras variaciones. Dicha sintaxis nos permite indicar el patrón de forma cómoda, como si de una cadena de texto se tratase, en la que determinados símbolos tienen un significado especial. Por ejemplo "[01]+" es una expresión regular que permite comprobar si una cadena conforma un número binario. Veamos cuáles son las reglas generales para construir una expresión regular:

- Podemos indicar que una cadena contiene un conjunto de símbolos fijo, simplemente poniendo dichos símbolos en el patrón, excepto para algunos símbolos especiales que necesitarán un carácter de escape como veremos más adelante. Por ejemplo, el patrón "aaa" admitirá cadenas que contengan tres aes.

- "[xyz]". 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". **Los corchetes representan una posición de la cadena que puede tomar uno de varios valores posibles.**

- "[a-z]" "[A-Z]" "[a-zA-Z]". Usando el guión y los corchetes podemos indicar que el patrón admite cualquier carácter entre la letra inicial y la final. Es importante que sepas que se diferencia entre letras mayúsculas y minúsculas, no son iguales de cara a las expresiones regulares.

- "[0-9]". Y nuevamente, usando un guión, podemos indicar que se permite la presencia de un dígito numérico entre 0 y 9, cualquiera de ellos, pero solo uno.

Con las reglas anteriores podemos indicar el conjunto de símbolos que admite el patrón y el orden que deben tener. Si una cadena no contiene los símbolos especificados en el patrón, en el mismo orden, entonces la cadena no encajará con el patrón. Veamos ahora como indicar repeticiones:

- "a?". Usaremos el interrogante para indicar que un símbolo puede aparecer una vez o ninguna. De esta forma la letra "a" podrá aparecer una vez o simplemente no aparecer.
- "a*". Usaremos el asterisco para indicar que un símbolo puede aparecer una vez o muchas veces, pero también ninguna. Cadenas válidas para esta expresión regular serían "aa", "aaa" o "aaaaaaaa".
- "a+". Usaremos el símbolo de suma para indicar que otro símbolo debe aparecer al menos una vez, pudiendo repetirse cuantas veces quiera.
- "a{1,4}" . Usando las llaves, podemos indicar el número mínimo y máximo de veces que un símbolo podrá repetirse. El primer número del ejemplo es el número 1, y quiere decir que la letra "a" debe aparecer al menos una vez. El segundo número, 4, indica que como máximo puede repetirse cuatro veces.
- "a{2,}". También es posible indicar solo el número mínimo de veces que un carácter debe aparecer (sin determinar el máximo), haciendo uso de las llaves, indicando el primer número y poniendo la coma (no la olvides).
- "a{5}". A diferencia de la forma anterior, si solo escribimos un número entre llaves, sin poner la coma detrás, significará que el símbolo debe aparecer un número exacto de veces. En este caso, la "a" debe aparecer exactamente 5 veces.
- "[a-z]{1,4}[0-9]+". Los indicadores de repetición se pueden usar también con corchetes, dado que los corchetes representan, básicamente, un símbolo. En el ejemplo anterior se permitiría de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.

Ejercicio resuelto

Con lo visto hasta ahora ya es posible construir una expresión regular capaz de verificar si una cadena contiene un DNI o un NIE, ¿serías capaz de averiguar cuál es dicha expresión regular?

Solución

La expresión regular que permite verificar si una cadena contiene un DNI o un NIE es la siguiente: "[XYxy]?[0-9]{1,9}[A-Za-z]"; aunque no es la única solución.

Nota: En realidad esta solución no es del todo precisa, ya que hay ciertas letras que no se crean con el algoritmo de generacíón de letra del DNI (concretamente las letras I, O, U y Ñ).

1.2. Expresiones regulares (II).

¿Y cómo uso las expresiones regulares en un programa? Pues de una forma sencilla. Para su uso, 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", lo cual significa verificar que es correcta y dejarla lista para su utilización. La clase `Matcher` sirve para comprobar si una cadena cualquiera sigue o no un patrón. Veamoslo con un ejemplo:

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

En el ejemplo, el método estático `compile` de la clase `Pattern` permite crear un patrón, dicho método compila la expresión regular pasada por parámetro y genera una instancia de `Pattern` (p en el ejemplo). El patrón p podrá ser usado múltiples veces para verificar si una cadena coincide o no con el patrón, dicha comprobación se hace invocando el método `matcher`, el cual combina el patrón con la cadena de entrada y genera una instancia de la clase `Matcher` (m en el ejemplo). 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:

- `m.matches()`. Devolverá `true` si toda la cadena (de principio a fin) encaja con el patrón o `false` en caso contrario.

- `m.lookingAt()`. 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.

- `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()`.

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 modo multilínea 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]".

1.3. Expresiones regulares (III).

¿Te resultan difíciles las expresiones regulares? Al principio siempre lo son, pero no te preocupes. Hasta ahora has visto como las expresiones regulares permiten verificar datos de entrada, permitiendo comprobar si un dato indicado sigue el formato esperado: que un DNI tenga el formato esperado, que un email sea un email y no otra cosa, etc. Pero ahora vamos a dar una vuelta de tuerca adicional.

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 (seguro que te resultará familiar):

```
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, podemos obtener por separado el texto contenido en cada uno de los grupos. En el ejemplo anterior, en el patrón hay tres grupos: uno para la letra inicial (grupo 1), otro para el número del DNI o NIE (grupo 2), y otro para la letra final o letra NIF (grupo 3). Al ponerlo en grupos, usando el método `group()`, podemos extraer la información de cada grupo y usarla a nuestra conveniencia.

Ten en cuenta que el primer grupo es el 1, y no el 0. Si pones `m.group(0)` obtendrás una cadena con toda la ocurrencia o coincidencia del patrón en la cadena, es decir, obtendrás la secuencia entera de símbolos que coincide con el patrón.

En el ejemplo anterior se usa el método `find`, éste buscará 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 devolverá `true` si ha encontrado una ocurrencia. Si no encuentra en una iteración ninguna ocurrencia es porque no existen más, y retornará `false`, saliendo del bucle. Esta construcción `while` es muy típica para este tipo de métodos y para las iteraciones, que veremos más adelante.

Lo último importante de las expresiones regulares que debes conocer son las secuencias de escape. Cuando en una expresión regular necesitamos especificar que lo que tiene que haber en la cadena es un paréntesis, una llave, o un corchete, tenemos que usar una secuencia de escape, dado que esos símbolos tienen un significado especial en los patrones. Para ello, simplemente antepondremos "\\" al símbolo. Por ejemplo, "\</code>" significará que debe haber un paréntesis en la cadena y se omitirá el significado especial del paréntesis. Lo mismo ocurre con "<code>\\[</code>", "<code>\\]</code>",
"<code>\

", etc. Lo mismo para el significado especial del punto, éste, tiene un significado especial (¿Lo recuerdas del apartado anterior?) salvo que se ponga "\\.", que pasará a significar "un punto" en vez de "cualquier carácter". **La excepción son las comillas, que se pondrían con una sola barra: "\"".**

Para saber más

Con lo estudiado hasta ahora, ya puedes utilizar las expresiones regulares y sacarles partido casi que al máximo. Pero las expresiones regulares son un campo muy amplio, por lo que es muy aconsejable que amplíes tus conocimientos con el siguiente enlace:

[Tutorial de expresiones regulares en Java (en inglés).]

6.D. Arrays Unidimensionales.

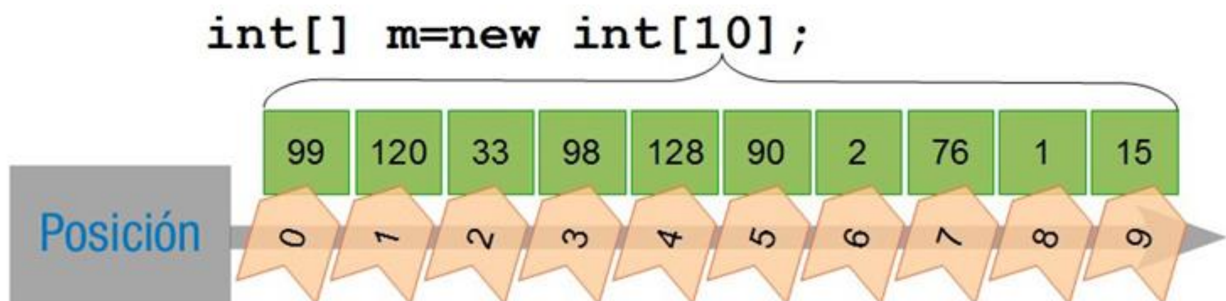
1. Arrays unidimensionales.

    1.1. Uso de arrays unidimensionales.

    1.2. Inicialización.

1. Arrays unidimensionales.

Todos los lenguajes de programación permiten el uso de arrays, veamos como son en Java.



Los arrays permiten almacenar una colección de objetos o datos del mismo tipo. Son muy útiles y su utilización es muy simple:

- **Declaración del array**. La declaración de un array consiste en decir "esto es un array" y sigue la siguiente estructura: "tipo[] nombre;". El tipo será un tipo de variable o una clase ya existente, de la cual se quieran almacenar varias unidades.
- **Creación del array**. La creación de un array consiste en decir el tamaño que tendrá el array, es decir, el número de elementos que contendrá, y se pone de la siguiente forma: "nombre=new tipo[dimension]", donde dimensión es un número entero positivo que indicará el tamaño del array. Una vez creado el array este no podrá cambiar de tamaño.

Veamos un ejemplo de su uso:

```
int[] n; // Declaración del array.
n = new int[10]; //Creación del array reservando para el un espacio en memoria.
int[] m=new int[10]; // Declaración y creación en un mismo lugar.
```

Una vez hecho esto, ya podemos almacenar valores en cada una de las posiciones del array, usando corchetes e indicando en su interior la posición en la que queremos leer o escribir, teniendo en cuenta que la primera posición es la cero y la última el tamaño del array menos uno. En el ejemplo anterior, la primera posición sería la 0 y la última sería la 9.

1.1. Uso de arrays unidimensionales.

Ya sabes declarar y crear de arrays, pero, ¿cómo y cuándo se usan? Pues existen tres ámbitos principalmente donde se pueden usar, y los tres son muy importantes: modificación de una posición del array, acceso a una posición del array y paso de parámetros.

La modificación de una posición del array se realiza con una simple asignación. Simplemente se especifica entre corchetes la posición a modificar después del nombre del array. Veámoslo con un simple ejemplo:

```
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.
```

El acceso a un valor ya existente dentro de una posición del array se consigue de forma similar, simplemente poniendo el nombre del array y la posición a la cual se quiere acceder entre corchetes:

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

Para nuestra comodidad, los arrays, como objetos que son en Java, disponen de una propiedad pública muy útil. La propiedad length nos permite saber el tamaño de cualquier array, lo cual es especialmente útil en métodos que tienen como argumento un array.

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

El tercer uso principal de los arrays, como se dijo antes, es en el paso de parámetros. Para pasar como argumento un array a una función o método, esta debe tener en su definición un parámetro declarado como array. Esto es simplemente que uno de los parámetros de la función sea un array. Veamos un ejemplo:

```
int sumaarray (int[] j) {
        int suma=0;
        for (int i=0; i<j.length;i++)
            suma=suma+j[i];
        return suma;
}
```

En el método anterior se pasa como argumento un array numérico, sobre el cual se calcula la suma de todos los números que contiene. Es un uso típico de los arrays, fijate que especificar que un argumento es un array es igual que declarar un array, sin la creación del mismo. Para pasar como argumento un array a una función, simplemente se pone el nombre del array:

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

En Java las variables se pasan por copia a los métodos, esto quiere decir que cuando se pasa una variable a un método, y se realiza una modificación de su valor en dicho método, el valor de la variable en el método desde el que se ha realizado la invocación no se modifica. Pero cuidado, eso no pasa con los arrays. Cuando dicha modificación se realiza en un array, es decir, se cambia el valor de uno de los elementos del array, si que cambia su valor de forma definitiva. Veamos un ejemplo que ilustra ambas cosas:

```
public static void main(String[] args) {
     int j=0; int[] i=new int[1];
     i[0]=0;
     modificaArray(j,i);
     System.out.println(j+"-"+i[0]);  /* Mostrará por pantalla "0-1", puesto que el contenido del array es
          modificado en la función, y aunque la variable j también se modifica, se modifica una copia de la misma,
          dejando el original intacto */
}

int modificaArray(int j, int[] i) {
     j++;
     i[0]++;  /* Modificación de los valores de la variable, solo afectará al array, no a j */
}
```

1.2. Inicialización.

Rellenar un array, para su primera utilización, es una tarea muy habitual que puede ser rápidamente simplificada. Vamos a explorar dos sistemas que nos van a permitir inicializar un array de forma cómoda y rápida.

En primer lugar, una forma habitual de crear un array y rellenarlo es simplemente a través de un método que lleve a cabo la creación y el rellenado del array. Esto es sencillo desde que podemos hacer que un método retorne un array simplemente indicando en la declaración que el valor retornado es tipo[], donde tipo de dato primitivo (`int, short, float,…`) o una clase existente (`String` por ejemplo). Veamos un ejemplo:

```
static int[] ArrayConNumerosConsecutivos (int totalNumeros) {
        int[] r=new int[totalNumeros];
        for (int i=0;i<totalNumeros;i++) r[i]=i;
        return r;
}
```

En el ejemplo anterior se crea un array con una serie de números consecutivos, empezando por el cero, ¿sencillo no? Este uso suele ahorrar bastantes líneas de código en tareas repetitivas. Otra forma de inicializar los arrays, cuando el número de elementos es fijo y sabido a priori, es indicando entre llaves el listado de valores que tiene el array. En el siguiente ejemplo

puedes ver la inicialización de un array de tres números, y la inicialización de un array con tres cadenas de texto:

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

Pero cuidado, la inicialización solo se puede usar en ciertos casos. La inicialización anterior funciona cuando se trata de un tipo de dato primitivo (`int, short, float, double`, etc.) o un String, y algunos pocos casos más, pero no funcionará para cualquier objeto.

Cuando se trata de un array de objetos, la inicialización del mismo es un poco más liosa, dado que el valor inicial de los elementos del array de objetos será `null`, o lo que es lo mismo, crear un array de objetos no significa que se han creado las instancias de los objetos. Hay que crear, para cada posición del array, el objeto del tipo correspondiente con el operador new. Veamos un ejemplo con la clase `StringBuilder`. En el siguiente ejemplo solo aparecerá `null` por pantalla:

```
StringBuilder[] j=new StringBuilderStringBuilder[10];
for (int i=0; i<j.length;i++) System.out.println("Valor" +i +
"="+j[i]); // Imprimirá null para los 10 valores.
```

Para solucionar este problema podemos optar por lo siguiente, crear para cada posición del array una instancia del objeto:

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

Reflexiona

Para acceder a una propiedad o a un método cuando los elementos del array son objetos, puedes usar la notación de punto detrás de los corchetes, por ejemplo: `diassemana[0].length`. Fijate bien en el array `diassemana` anterior y piensa en lo que se mostraría por pantalla con el siguiente código:

```
System.out.println(diassemana[0].substring(0,2));
```

6.E. Arrays multidimensionales.

1. Arrays multidimensionales.
    1.1. Uso de arrays multidimensionales.
    1.2. Inicialización de arrays multidimensionales.

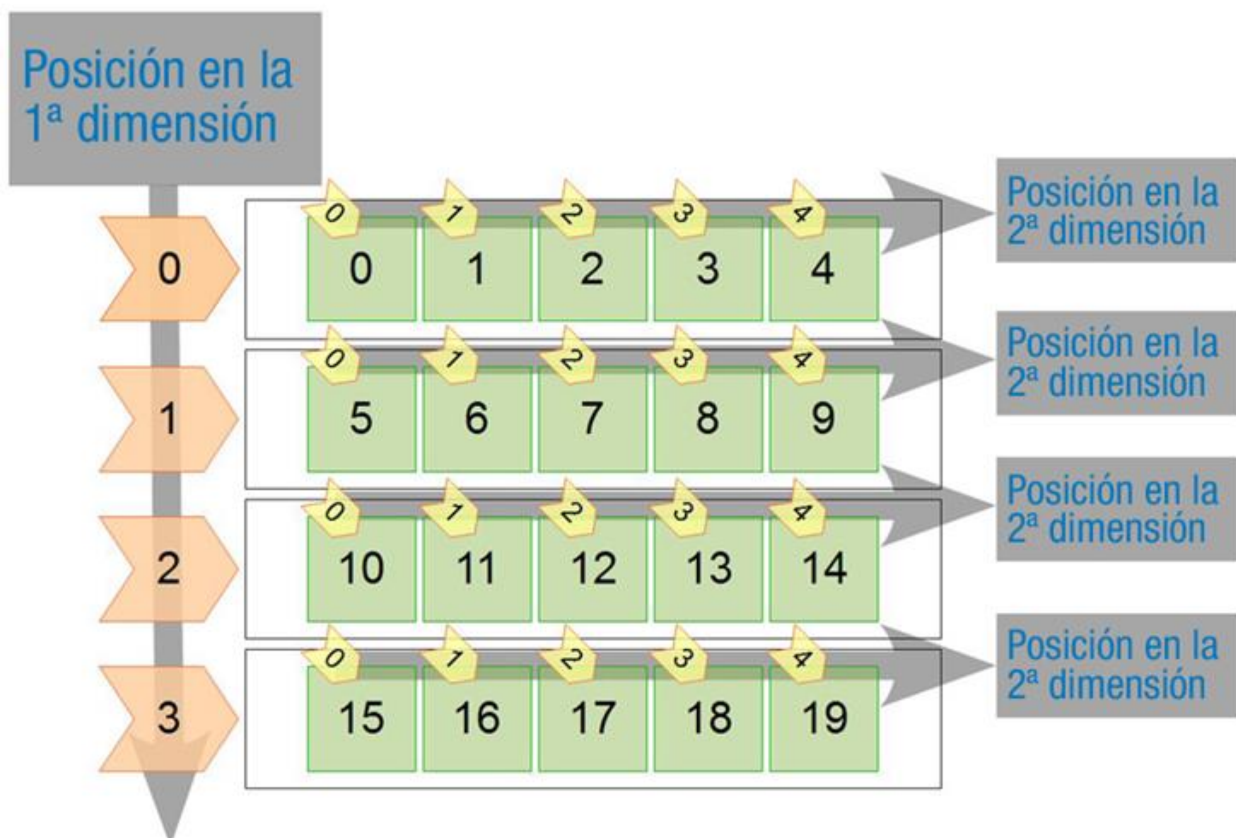
1. Arrays multidimensionales.

¿Qué estructura de datos utilizarías para almacenar los píxeles de una imagen digital? Normalmente las imágenes son cuadradas o rectangulares, así que una de las estructuras más adecuadas es la matriz. En la matriz cada valor podría ser el color de cada píxel. Pero, ¿qué es una matriz a nivel de programación? Pues es un array con dos dimensiones, o lo que es lo mismo, un array cuyos elementos son arrays de números.

Los arrays multidimensionales están en todos los lenguajes de programación actuales, y obviamente también en Java. La forma de crear un array de dos dimensiones en Java es la siguiente:

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

El código anterior creará un array de dos dimensiones, o lo que es lo mismo, creará un array que contendrá 4 arrays de 5 números cada uno. Veámoslo con un ejemplo gráfico:



Al igual que con los arrays de una sola dimensión, los arrays multidimensionales deben declararse y crearse. Podremos hacer arrays multidimensionales de todas las dimensiones que queramos y de cualquier tipo. En ellos todos los elementos del array serán del mismo tipo, como en el caso de los arrays de una sola dimensión. La declaración comenzará especificando el tipo o la clase de los elementos que forman el array, después pondremos tantos corchetes como dimensiones tenga el array y por último el nombre del array, por ejemplo:

```
int [][][] arrayde3dim;
```

La creación es igualmente usando el operador new, seguido del tipo y los corchetes, en los cuales se especifica el tamaño de cada dimensión:

```
arrayde3dim=new int[2][3][4];
```

Todo esto, como ya has visto en un ejemplo anterior, se puede escribir en una única sentencia.

1.1. Uso de arrays multidimensionales.

¿Y en que se diferencia el uso de un array multidimensional con respecto a uno de una única dimensión? Pues en muy poco la verdad. Continuaremos con el ejemplo del apartado anterior:

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

Para acceder a cada uno de los elementos del array anterior, 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.

Puedes asignar un valor a una posición concreta dentro del array, indicando la posición en cada una de las dimensiones entre corchetes:

```
a2d[0][0]=3;
```

Y como es de imaginar, puedes usar un valor almacenado en una posición del array multidimensional simplemente poniendo el nombre del array y los índices del elemento al que deseas acceder entre corchetes, para cada una de las dimensiones del array. Por ejemplo:

```
int suma=a2d[0][0]+a2d[0][1]+a2d[0][2]+a2d[0][3]+a2d[0][4];
```

Como imaginarás, los arrays multidimensionales pueden también ser pasados como parámetros a los métodos, simplemente escribiendo la declaración del array en los argumentos del método, de forma similar a como se realizaba para arrays de una dimensión. Por ejemplo:

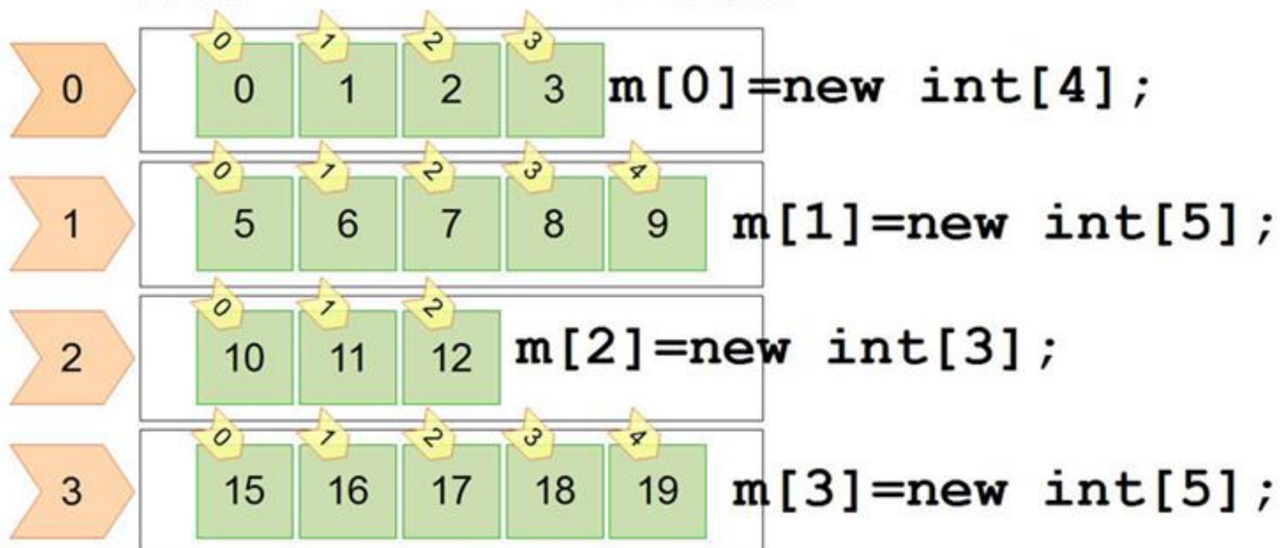
```
static int sumaarray2d(int[][] a2d) {
```

```
        int suma = 0;
        for (int i1 = 0; i1 < a2d.length; i1++)
            for (int i2 = 0; i2 < a2d[i1].length; i2++)   suma +=
a2d[i1][i2];
        return suma;
    }
```

Del código anterior, fíjate especialmente en el uso del atributo `length` (que nos permite obtener el tamaño de un array). Aplicado directamente sobre el array nos permite saber el tamaño de la primera dimensión (`a2d.length`). Como los arrays multidimensionales son arrays que tienen como elementos arrays (excepto el último nivel que ya será del tipo concreto almacenado), para saber el tamaño de una dimensión superior tenemos que poner el o los índices entre corchetes seguidos de `length` (`a2d[i1].length`).

Para saber al tamaño de una segunda dimensión (dentro de una función por ejemplo) hay que hacerlo así y puede resultar un poco engorroso, pero gracias a esto podemos tener arrays multidimensionales irregulares.

Una matriz es un ejemplo de array multidimensional regular, ¿por qué? Pues porque es un array que contiene arrays de números todos del mismo tamaño. Cuando esto no es así, es decir, cuando los arrays de la segunda dimensión son de diferente tamaño entre sí, se puede decir que es un array multidimensional irregular. En Java se puede crear un array irregular de forma relativamente fácil, veamos un ejemplo para dos dimensiones.



- **Declaramos y creamos el array pero sin especificar la segunda dimensión.** Lo que estamos haciendo en realidad es crear simplemente un array que contendrá arrays, sin decir como son de grandes los arrays de la siguiente dimensión: `int[][] 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];`

Recomendación

Cuando uses arrays irregulares, por seguridad, es conveniente que verifiques siempre que el array no sea `null` en segundas dimensiones, y que la longitud sea la esperada antes de acceder a los datos:

```
if (irregular[1]!=null && irregular[1].length>10) {...}
```

1.2. Inicialización de arrays multidimensionales.

¿En qué se diferencia la inicialización de arrays unidimensionales de arrays multidimensionales? En muy poco. La inicialización de los arrays multidimensionales es igual que la de los arrays unidimensionales.

Para que una función retorne un array multidimensional, se hace igual que en arrays unidimensionales. Simplemente hay que poner el tipo de dato seguido del número de corchetes correspondiente, según el número de dimensiones del array. Eso claro, hay que ponerlo en la definición del método:

```
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 un array multidimensional usando las llaves, poniendo después de la declaración del array un símbolo de igual, y encerrado entre llaves los diferentes valores del array separados por comas, con la salvedad de que hay que poner unas llaves nuevas cada vez que haya que poner los datos de una nueva dimensión, lo mejor es verlo con un ejemplo:

```
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 anterior sería un array de 4 por 3 y el siguiente sería un array de 2x2x2. Como puedes observar esta notación a partir de 3 dimensiones ya es muy liosa y normalmente no se usa. Utilizando esta notación también podemos inicializar rápidamente arrays irregulares, simplemente poniendo separados por comas los elementos que tiene cada dimensión:

```
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} } };
```

Es posible que en algunos libros y en Internet se refieran a los arrays usando otra terminología. A los arrays unidimensionales es común llamarlos también **arreglos** o **vectores**, a los arrays de dos dimensiones es común llamarlos directamente **matrices**, y a los arrays de más de dos dimensiones es posible que los veas escritos como **matrices multidimensionales**. Sea como sea, lo más normal en la jerga informática es llamarlos arrays, término que procede del inglés.

Para saber más

Las matrices tienen asociadas un amplio abanico de operaciones (transposición, suma, producto, etc.). En la siguiente página tienes ejemplos de como realizar algunas de estas operaciones, usando por supuesto arrays:

Operaciones básicas con matrices.