

Sumario

UT05: Utilización de Objetos.....	2
1 Características de los objetos.....	2
1.1 Ciclo de vida de los objetos.....	2
1.2 Declaración.....	2
1.3 Instanciación.....	3
1.4 Manipulación.....	4
1.5 Destrucción de objetos y liberación de memoria.....	5
2 Utilización de métodos.....	6
2.1 Parámetros y valores devueltos.....	7
2.2 Constructores.....	8
2.3 El operador this.....	9
2.4 Métodos estáticos.....	12
3 Introducción a las estructuras de almacenamiento.....	12
4 Cadenas de caracteres.....	13
4.1 Operaciones avanzadas con cadenas de caracteres: concatenación y conversión (toString).....	14
4.2 Operaciones avanzadas con cadenas de caracteres: substring.....	16
4.3 Operaciones avanzadas con cadenas de caracteres: valueOf.....	17
4.4 El método format.....	17
4.5 Operaciones avanzadas con cadenas de caracteres: comparación.....	20
4.6 Más operaciones avanzadas con cadenas de caracteres.....	21
4.7 Expresiones regulares. Definición.....	22
4.8 Expresiones regulares. Clases "Pattern" y "Matcher".....	23
4.9 Expresiones regulares.....	25
5 Arrays.....	26
5.1 Uso de arrays unidimensionales.....	26
5.2 Inicialización.....	28
5.3 Arrays multidimensionales.....	29
5.4 Uso de arrays multidimensionales.....	30
5.5 Inicialización de arrays multidimensionales.....	31

UT05: Utilización de Objetos

1 Características de los objetos

Todo programa en Java parte de una única clase, que como hemos comentado se trata de la clase principal. Esta clase ejecutará el contenido de su método **main()**, el cual será el que utilice las demás clases del programa, cree objetos y lance mensajes a otros objetos.

1.1 Ciclo de vida de los objetos

Las **instancias** u **objetos** tienen un tiempo de vida determinado. Cuando un objeto no se va a utilizar más en el programa, es destruido por el recolector de basura para liberar recursos que pueden ser reutilizados por otros objetos. A la vista de lo anterior, podemos concluir que los objetos tienen un **ciclo de vida**, en el cual podemos distinguir las siguientes fases:

- **Creación**, donde se hace la reserva de memoria e inicialización de atributos.
- **Manipulación**, que se lleva a cabo cuando se hace uso de los atributos y métodos del objeto.
- **Destrucción**, eliminación del objeto y liberación de recursos.

1.2 Declaración

Para la creación de un objeto hay que seguir los pasos:

- **Declaración**: Definir el tipo de objeto.
- **Instanciación**: Creación del objeto utilizando el operador new.

Veamos primero cómo declarar un objeto. Para la definición del tipo de objeto debemos emplear la siguiente instrucción:

```
<tipo> nombre_objeto;
```

- **tipo** es la clase a partir de la cual se va a crear el objeto
- **nombre_objeto** es el nombre de la variable referencia con la cual nos referiremos al objeto.

Los tipos referenciados o referencias se utilizan para guardar la dirección de los datos en la memoria del ordenador.

Nada más crear una referencia, ésta se encuentra vacía. Cuando una referencia a un objeto no contiene ninguna instancia se dice que es una referencia nula, es decir, que contiene el valor null. Esto quiere decir que la referencia está creada pero que el objeto no está instanciado todavía, por eso la referencia apunta a un objeto inexistente llamado "nulo".

Para entender mejor la declaración de objetos, veamos un ejemplo.

```
String mensaje;
```

Los nombres de la clase empiezan con mayúscula, como String, y los nombres de los objetos con minúscula, como mensaje, así sabemos qué tipo de elemento utilizando.

Pues bien, String es realmente la clase a partir de la cual creamos nuestro objeto llamado mensaje. Antes decíamos que mensaje era una variable del tipo de dato String. Ahora realmente vemos que mensaje es un objeto de la clase String. Pero mensaje aún no contiene el objeto porque no ha sido instanciado. Cuando creamos un objeto estamos haciendo uso de una variable que almacena la dirección de ese objeto en memoria. Esa variable es una referencia o un tipo de datos referenciado, porque no contiene el dato sino la posición del dato en la memoria del ordenador.

```
String saludo = new String ("Bienvenido a Java");  
String s; //s vale null  
s = saludo; //asignación de referencias
```

En las instrucciones anteriores, las variables s y saludo apuntan al mismo objeto de la clase String. Esto implica que cualquier modificación en el objeto saludo modifica también el objeto al que hace referencia la variable s, ya que realmente son el mismo.

1.3 Instanciación

Una vez creada la referencia al objeto, debemos crear la instancia u objeto que se va a guardar en esa referencia. Para ello utilizamos la orden new con la siguiente sintaxis:

```
nombre_objeto = new <Constructor_de_la_Clase>([<par1>, <par2>, ..., <parN>]);
```

Donde:

- nombre_objeto es el nombre de la variable referencia con la cual nos referiremos al objeto,
- new es el operador para crear el objeto
- Constructor_de_la_Clase es un método especial de la clase, que se llama igual que ella, y se encarga de inicializar el objeto, es decir, de dar unos valores iniciales a sus atributos,
- par1-parN, son parámetros que puede o no necesitar el constructor para dar los valores iniciales a los atributos del objeto.

Durante la instanciación del objeto, se reserva memoria suficiente para el objeto. De esta tarea se encarga Java y en ella juega un papel muy importante el recolector de basura, que se encarga de eliminar de la memoria los objetos no utilizados para que ésta pueda volver a ser utilizada. Para instanciar un objeto String, haríamos lo siguiente:

```
mensaje = new String;
```

Así estaríamos instanciando el objeto mensaje. Para ello utilizaríamos el operador new y el constructor de la clase String a la que pertenece el objeto según la declaración que

hemos hecho en el apartado anterior. A continuación utilizamos el constructor, que se llama igual que la clase, String.

En el ejemplo anterior el objeto se crearía con la cadena vacía (""), si queremos que tenga un contenido debemos utilizar parámetros en el constructor, así:

```
mensaje = new String ("El primer programa");
```

Java permite utilizar la clase String como si de un tipo de dato primitivo se tratara, por eso no hace falta utilizar el operador new para instanciar un objeto de la clase String.

La declaración e instanciación de un objeto pueden realizarse en la misma instrucción:

```
String mensaje = new String ("El primer programa");
```

1.4 Manipulación

Una vez creado e instanciado el objeto ¿cómo accedemos a su contenido? Para acceder a los atributos y métodos del objeto utilizaremos el nombre del objeto seguido del operador punto (.) y el nombre del atributo o método que queremos utilizar. Cuando utilizamos el operador punto se dice que estamos enviando un mensaje al objeto. La forma general de enviar un mensaje a un objeto es:

```
nombre_objeto.mensaje
```

Por ejemplo, para acceder a las variables instancia o atributos se usa esta sintaxis:

```
nombre_objeto.atributo
```

Y para acceder a los métodos o funciones miembro del objeto se utiliza la sintaxis es:

```
nombre_objeto.método( [par1, par2, ..., parN] )
```

En la sentencia anterior par1, par2, etc. son los parámetros que utiliza el método. Aparecen entre corchetes para indicar son opcionales.

Veamos un ejemplo para el que necesitaremos la Biblioteca de Clases Java o API (Application Programming Interface). Uno de los paquetes de librerías o bibliotecas es java.awt. Este paquete contiene clases destinadas a la creación de objetos gráficos e imágenes. Vemos por ejemplo cómo crear un rectángulo.

En primer lugar instanciamos el objeto utilizando el método constructor, que se llama igual que el objeto, e indicando los parámetros correspondientes a la posición y a las dimensiones del rectángulo:

```
Rectangle rect = new Rectangle(50, 50, 150, 150);
```

Una vez instanciado el objeto rectángulo si queremos cambiar el valor de los atributos utilizamos el operador punto. Por ejemplo, para cambiar la dimensión del rectángulo:

```
rect.height=100;  
rect.width=100;
```

O bien podemos utilizar un método para hacer lo anterior:

```
rect.setSize(200, 200);
```

Este sería el código de ejemplo:

```
/*
 * Muestra como se manipulan objetos en Java
 */
import java.awt.Rectangle;
/**
 *
 * @author autor
 */
public class Manipular {
    public static void main(String[] args) {
        // Instanciamos el objeto rect indicando posicion y dimensiones
        Rectangle rect = new Rectangle( 50, 50, 150, 150 );
        //Consultamos las coordenadas x e y del rectangulo
        System.out.println( "----- Coordenadas esquina superior izqda. -----");
        System.out.println("\tx = " + rect.x + "\n\ty = " + rect.y);
        // Consultamos las dimensiones (altura y anchura) del rectangulo
        System.out.println( "\n----- Dimensiones -----");
        System.out.println("\tAlto = " + rect.height );
        System.out.println( "\tAncho = " + rect.width);
        //Cambiar coordenadas del rectangulo
        rect.height=100;
        rect.width=100;
        rect.setSize(200, 200);
        System.out.println( "\n-- Nuevos valores de los atributos --");
        System.out.println("\tx = " + rect.x + "\n\ty = " + rect.y);
        System.out.println("\tAlto = " + rect.height );
        System.out.println( "\tAncho = " + rect.width);
    }
}
```

1.5 Destrucción de objetos y liberación de memoria

Cuando un objeto ya no se usa, hay que liberar el espacio de memoria y otros recursos que poseía para poder ser reutilizados ("destrucción del objeto"). Esta tarea corre a cargo del recolector de basura (garbage collector), o sistema de destrucción automática de objetos que ya no son utilizados. Se libera la zona de memoria que fue reservada previamente por el operador `new`. Esto permite que los programadores se despreocupen de ello. Se ejecuta en segundo plano y de manera muy eficiente, sin afectar a la velocidad del programa en ejecución. El recolector de basura busca periódicamente objetos que ya no son referenciados y los marca para su eliminación, que realiza más tarde.

Justo antes de eliminar un objeto, el recolector de basura ejecuta su método `finalize()`. Si queremos forzar que se ejecute el proceso de finalización de todos los objetos del programa podemos utilizar el método `runFinalization()` de la clase `System`. La clase `System` forma parte de la Biblioteca de Clases de Java, contiene clases para la entrada y salida de información, acceso a variables de entorno del programa y otros métodos de diversa utilidad. Para forzar el proceso de finalización ejecutaríamos:

```
System.runFinalization();
```

2 Utilización de métodos

Los métodos, junto con los atributos, forman parte de la estructura interna de un objeto. Los métodos contienen la declaración de variables locales y las operaciones que se pueden realizar para el objeto, y que son ejecutadas cuando el método es invocado. Se definen en el cuerpo de la clase y posteriormente son instanciados para convertirse en métodos instancia de un objeto.

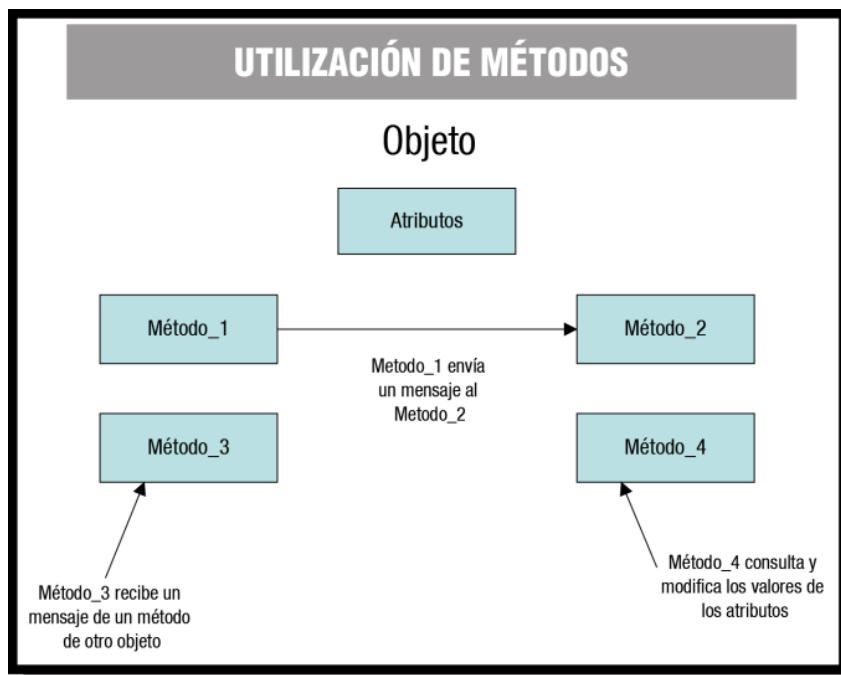
Para utilizar los métodos adecuadamente es conveniente conocer la estructura básica de que disponen.

Cabecera del método

```
public tipo_dato_devuelto nombre_metodo (par1, par2, ..., parN) {  
    // Declaracion de variables locales  
    // Instrucciones del metodo  
}
```

Cuerpo del método

De la misma manera que las clases, los métodos están compuestos por una cabecera y un cuerpo. La cabecera también tiene modificadores, en este caso "public" para indicar que el método es público, es decir, le pueden enviar mensajes los métodos del objeto y también los métodos de otro objeto externo.



Dentro de un método nos encontramos el cuerpo del método, que contiene el código de la acción a realizar. Las acciones que un método puede realizar son:

- Inicializar los atributos del objeto
- Consultar los valores de los atributos
- Modificar los valores de los atributos
- Llamar a otros métodos, del mismo del objeto o de objetos externos

2.1 Parámetros y valores devueltos

Los métodos se pueden utilizar tanto para consultar información sobre el objeto como para modificar su estado. La información consultada del objeto se devuelve a través de lo que se conoce como valor de retorno, y la modificación del estado del objeto, o sea, de sus atributos, se hace mediante la lista de parámetros. En general, la lista de parámetros de un método se puede declarar de dos formas diferentes:

- **Por valor.** El valor de los parámetros no se devuelve al finalizar el método, es decir, cualquier modificación que se haga en los parámetros no tendrá efecto una vez se salga del método. Esto es así porque cuando se llama al método desde cualquier parte del programa, dicho método recibe una copia de los argumentos, por tanto cualquier modificación que haga será sobre la copia, no sobre las variables originales.
- **Por referencia.** La modificación en los valores de los parámetros sí tienen efecto tras la finalización del método. Cuando pasamos una variable a un método por referencia lo que estamos haciendo es pasar la dirección del dato en memoria, por

tanto cualquier cambio en el dato seguirá modificado una vez que salgamos del método.

En el lenguaje Java, todas las variables se pasan por valor, excepto los objetos que se pasan por referencia. En Java, la declaración de un método tiene dos restricciones:

- Un método siempre tiene que devolver un valor (no hay valor por defecto). Este valor de retorno es el valor que devuelve el método cuando termina de ejecutarse, al método o programa que lo llamó. Puede ser un tipo primitivo, un tipo referenciado o bien el tipo void, que indica que el método no devuelve ningún valor.
- Un método tiene un número fijo de argumentos. Los argumentos son variables a través de las cuales se pasa información al método desde el lugar del que se llama, para que éste pueda utilizar dichos valores durante su ejecución. Los argumentos reciben el nombre de parámetros cuando aparecen en la declaración del método.

El valor de retorno es la información que devuelve un método tras su ejecución. Según hemos visto en el apartado anterior, la cabecera de un método se declara como sigue:

```
public tipo_de_dato_devuelto nombre_metodo (lista_de_parametros);
```

Como vemos, el tipo de dato devuelto aparece después del modificador public y se corresponde con el valor de retorno.

La lista de parámetros aparece al final de la cabecera del método, justo después del nombre, encerrados entre signos de paréntesis y separados por comas. Se debe indicar el tipo de dato de cada parámetro así:

```
(tipo_parámetro1 nombre_parámetro1, ..., tipo_parámetroN nombre_parámetroN)
```

Cuando se llame al método, se deberá utilizar el nombre del método, seguido de los argumentos que deben coincidir con la lista de parámetros. La lista de argumentos en la llamada a un método debe coincidir en número, tipo y orden con los parámetros del método, ya que de lo contrario se produciría un error de sintaxis.

2.2 Constructores

Un constructor es un método especial con el mismo nombre de la clase y que no devuelve ningún valor tras su ejecución.

Estructura interna de un método constructor

```
public NombreClase (par1, par2, ..., parN)
{
    // Inicializacion de atributos
    // Resto de instrucciones del constructor
} // Fin del metodo
```

parámetros opcionales

el modificador normalmente siempre es public

puede llamar a otros métodos de la clase aunque no es recomendable

Cuando creamos un objeto debemos instanciarlo utilizando el constructor de la clase. Veamos la clase `Date` proporcionada por la Biblioteca de Clases de Java. Si queremos instanciar un objeto a partir de la clase `Date` tan sólo tendremos que utilizar el constructor seguido de una pareja de abrir / cerrar paréntesis:

```
Date fecha = new Date();
```

Con la anterior instrucción estamos creando un objeto `fecha` de tipo `Date`, que contendrá la fecha y hora actual del sistema. La estructura de los constructores es similar a la de cualquier método, salvo que no tiene tipo de dato devuelto porque no devuelve ningún valor. Está formada por una cabecera y un cuerpo, que contiene la inicialización de atributos y resto de instrucciones del constructor.

El método constructor tiene las siguientes particularidades:

- El constructor se invoca automáticamente solo una vez en la creación de un objeto.
- Los constructores no empiezan con minúscula, como el resto de los métodos, ya que se llaman igual que la clase y las clases empiezan con letra mayúscula.
- Puede haber varios constructores para una clase.
- Como cualquier método, el constructor puede tener parámetros para definir qué valores dar a los atributos del objeto.
- El constructor por defecto es aquel que no tiene argumentos o parámetros. Cuando creamos un objeto llamando al nombre de la clase sin argumentos, estamos

utilizando el constructor por defecto.

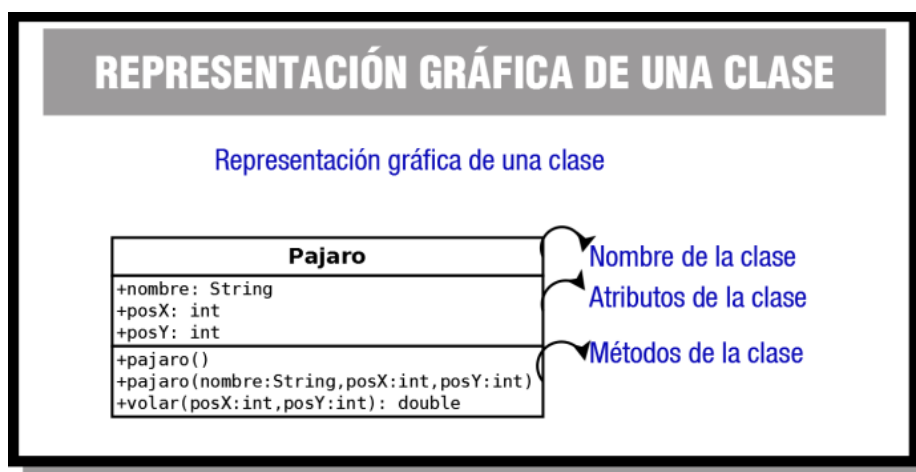
- Es necesario que toda clase tenga al menos un constructor. Si no definimos constructores para una clase, y solo en ese caso, el compilador crea un constructor por defecto vacío, que inicializa los atributos a sus valores por defecto, según del tipo que sean: 0 para los tipos numéricos, false para los boolean y null para los tipo carácter y las referencias. Dicho constructor llama al constructor sin argumentos de la superclase (clase de la cual hereda); si la superclase no tiene constructor sin argumentos se produce un error de compilación.
- Cuando definimos constructores personalizados, el constructor por defecto deja de existir, y si no definimos nosotros un constructor sin argumentos cuando intentemos utilizar el constructor por defecto nos dará un error de compilación.

2.3 El operador this

Los constructores y métodos de un objeto suelen utilizar el operador this. Este operador sirve para referirse a los atributos de un objeto cuando estamos dentro de él. Sobre todo se utiliza cuando existe ambigüedad entre el nombre de un parámetro y el nombre de un atributo, entonces en lugar del nombre del atributo solamente escribiremos this.nombre_atributo, y así no habrá duda de a qué elemento nos estamos refiriendo.

Vamos a ilustrar mediante un ejemplo la utilización de objetos y métodos, así como el uso de parámetros y el operador this. Aunque la creación de clases la veremos en las siguientes unidades, en este ejercicio creamos una pequeña clase para que podamos instanciar el objeto con el que vamos a trabajar.

Las clases se suelen representar como un rectángulo, y dentro de él se sitúan los atributos y los métodos de dicha clase.



En la imagen, la clase Pajaro está compuesta por tres atributos, uno de ellos el nombre y otros dos que indican la posición del ave, posX y posY. Tiene dos métodos constructores y

un método volar(). Como sabemos, los métodos constructores reciben el mismo nombre de la clase, y puede haber varios para una misma clase, dentro de ella se diferencian unos de otros por los parámetros que utilizan.

También se emplea this() para llamar a un constructor desde otro constructor dentro de la misma clase. Para ello hacemos: this(tipo y número de parámetros del constructor llamado) y lo tenemos que escribir como primera sentencia en el constructor llamante. Ejemplo:

```
public class Prueba
{
    private String a;
    public Prueba(String a)
    {
        his(String a,int b) //Llamada al constructor con 2 parámetros String e int
        this.a=a;
    }
    public Prueba(String z,int w)
    {
        .....Constructor llamado
    }
}
```

Ejercicio resuelto

Dada una clase principal llamada Pajaro, se definen los atributos y métodos que aparecen en la imagen. Los métodos realizan las siguientes acciones:

- pajaro(). Constructor por defecto. En este caso, el constructor por defecto no contiene ninguna instrucción, ya que Java inicializa de forma automática las variables miembro, si no le damos ningún valor.
- pajaro(String nombre, int posX, int posY). Constructor que recibe como argumentos una cadena de texto y dos enteros para inicializar el valor de los atributos.
- volar(int posX, int posY). Método que recibe como argumentos dos enteros: posX y posY, y devuelve un valor de tipo double como resultado, usando la palabra clave return. El valor devuelto es el resultado de aplicar un desplazamiento de acuerdo con la siguiente fórmula:

$$\text{desplazamiento} = \sqrt{\text{posX} * \text{posX} + \text{posY} * \text{posY}}$$

Diseña un programa que utilice la clase Pajaro, cree una instancia de dicha clase y ejecute sus métodos.

Lo primero que debemos hacer es crear la clase Pajaro, con sus métodos y atributos. De acuerdo con los datos que tenemos, el código de la clase sería el siguiente:

```
public class Pajaro {  
  
    String nombre;  
    int posX, posY;  
  
    public Pajaro() {  
  
    }  
  
    public Pajaro(String nombre, int posX, int posY) {  
        this.nombre=nombre;  
        this.posX=posX;  
        this.posY=posY;  
    }  
  
    double volar (int posX, int posY) {  
  
        double desplazamiento = Math.sqrt( posX*posX + posY*posY );  
        this.posX = posX;  
        this.posY = posY;  
  
        return desplazamiento;  
    }  
}
```

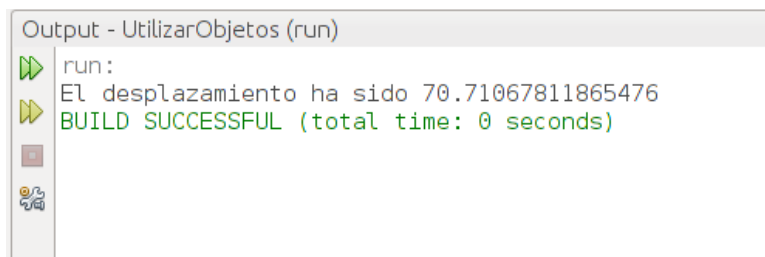
Debemos tener en cuenta que se trata de una clase principal, lo cual quiere decir que debe contener un método main() dentro de ella. En el método main() vamos a situar el código de nuestro programa. El ejercicio dice que tenemos que crear una instancia de la clase y ejecutar sus métodos, entre los que están el constructor y el método volar().

También es conveniente imprimir el resultado de ejecutar el método volar(). Por tanto, lo que haría el programa sería:

- Crear un objeto de la clase e inicializarlo.
- Invocar al método volar.
- Imprimir por pantalla la distancia recorrida.
- Para inicializar el objeto utilizaremos el constructor con parámetros, después ejecutaremos el método volar() del objeto creado y finalmente imprimiremos el valor que nos devuelve el método. El código de la clase main() quedaría como sigue:

```
public static void main(String[] args) {  
  
    Pajaro loro = new Pajaro("Lucy",50,50) ;  
    double d = loro.volar(50,50);  
    System.out.println("El desplazamiento ha sido " + d);  
}
```

Si ejecutamos nuestro programa el resultado sería el siguiente:



2.4 Métodos estáticos

Los métodos estáticos son aquellos métodos definidos para una clase que se pueden usar directamente, sin necesidad de crear un objeto de dicha clase. También se llaman **métodos de clase**.

Para llamar a un método estático utilizaremos:

- El nombre del método, si lo llamamos desde la misma clase en la que se encuentra definido.
- El nombre de la clase, seguido por el operador punto (.) más el nombre del método estático, si lo llamamos desde una clase distinta a la que se encuentra definido:

`nombre_clase.nombre_metodo_estatico`

- El nombre del objeto, seguido por el operador punto (.) más el nombre del método estático. Utilizaremos esta forma cuando tengamos un objeto instanciado de la clase en la que se encuentra definido el método estático, y no podamos utilizar la anterior:

`nombre_objeto.nombre_metodo_estatico`



Los métodos estáticos no afectan al estado de los objetos instanciados de la clase (variables instancia), y suelen utilizarse para realizar operaciones comunes a todos los objetos de la clase. Por ejemplo, si necesitamos contar el número de objetos instanciados de una clase, podríamos utilizar un método estático que fuera incrementando el valor de una variable entera de la clase conforme se van creando los objetos.

En la Biblioteca de Clases de Java existen muchas clases que contienen métodos estáticos. Pensemos en las clases que ya hemos utilizado en unidades anteriores, como hemos comentado la clase String con todas las operaciones que podíamos hacer con ella y con los objetos instanciados a partir de ella. O bien la clase Math. para la conversión de tipos de datos. Todos ellos son métodos estáticos que la API de Java define para esas clases. Lo importante es tener en cuenta que, al tratarse de métodos estáticos, no es necesario crear un objeto para utilizarlos.

3 Introducción a las estructuras de almacenamiento

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.

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.

4 Cadenas 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.

Fijate en el 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");
```

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.

4.1 Operaciones avanzadas con cadenas de caracteres: concatenación y conversión (toString)

Veamos algunas operaciones posibles con cadenas

La **concatenación** es la unión de dos cadenas para formar una sola. En Java es muy sencillo, pues sólo hay 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 mediante el método "concat" del objeto String :

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

Esta expresión 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 se borrarán de memoria cuando el recolector de basura detecte que ya no se usan.

También 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 (aquellos que no se pueden modificar una vez creados. Es el caso de los String de Java, así como las clases envoltorio Integer, Float, Double, etc.).

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() está disponible en todas las clases de Java. Su objetivo es permitir la conversión de una instancia de clase en cadena de texto, de forma que se pueda convertir a texto el contenido de la instancia. No siempre es posible convertir. Hay clases fácilmente convertibles a texto, como es la clase Integer, por ejemplo, y otras que no se pueden convertir, en las que el resultado de invocar el método toString() es información relativa a la instancia.

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

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

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.

Otra operación muy habitual es la conversión de número a cadena y de cadena a número. Al recoger un valor por consola se recoge generalmente una cadena, pero, ¿cómo podemos comprobar que la edad es mayor que 0? Para poder realizar esa comprobación hay 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 se debe 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:

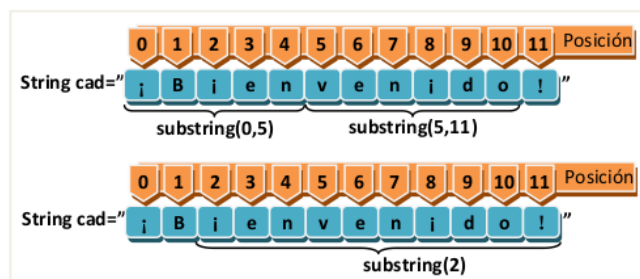
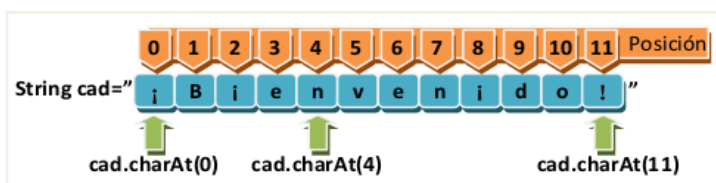
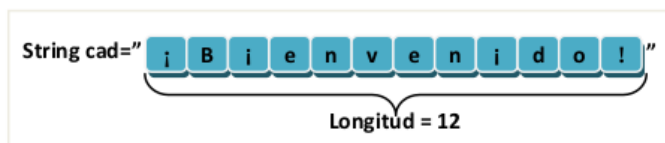
```
String cad2="Número cinco: " + 5;
System.out.println(cad2);
```

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.

Curiosidad: Ejecuta la instrucción `System.out.println("A"+5f)` (teniendo en cuenta que la letra "f" indica que se use el 5 como literal en punto flotante)

4.2 Operaciones avanzadas con cadenas de caracteres: substring

En los siguientes 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. 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 siguiente código 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, 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.

4.3 Operaciones avanzadas con cadenas de caracteres: valueOf.

¿Cómo comprobarías que la cadena "3" es mayor que 0? No se puede comparar directamente una cadena con un número, así que tendremos que aprender a 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);  
} catch (NumberFormatException e)  
{ /* Código a ejecutar si no se puede convertir */ }
```

En el código anterior se puede 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.

4.4 El método format.

Ahora nos planteamos otro reto: supongamos que queremos mostrar el precio de un producto por pantalla, por ejemplo, "3,30 €". 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. Por ejemplo:

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

El resultado no muestra "3,30 €" sino "3,3 €". Para conseguirlo, usaremos el "formateado de cadenas". En Java se puede hacer mediante el método estático format, disponible en el objeto String . Este método permite crear una cadena proyectando los argumentos en un formato específico de salida. Lo vemos en este ejemplo:

```
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. La cadena "%.2f" 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.

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.				
Conversión	Formato	Tipos de datos	Ejemplo	Resultado
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 método toString).	String cad="hola mundo"; String d=String.format("Resultado: %s", cad); System.out.println (d);	Resultado: hola mundo
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 mas corto.	double i=10.5; String d=String.format("Resultado: %g", i); System.out.println (d);	Resultado: 10.5000

Veamos ahora algunos modificadores que se 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` : Tanto el ancho como la precisión van entre corchetes, es decir, son modificadores opcionales. Si queremos, por ejemplo, que la salida genere 3 decimales como mínimo, podremos ponerlo así:

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

Dado que el 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, 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).

4.5 Operaciones avanzadas con cadenas de caracteres: comparación

Java ofrece muchas más operaciones sobre cadenas. En la siguiente tabla aparecen algunas de 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.

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 (<i>cad1</i>) es anterior en orden alfabético a la que se pasa por argumento (<i>cad2</i>), 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 <code>equals</code> . Retornará <i>true</i> si son iguales, y <i>false</i> si no lo son.
cad1.compareToIgnoreCase(cad2) cad1.equalsIgnoreCase(cad2)	El método <code>compareToIgnoreCase</code> funciona igual que el método <code>compareTo</code> , 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 <code>equalsIgnoreCase</code> es igual que el método <code>equals</code> pero sin tener en cuenta las minúsculas.
cad1.trim()	Genera una copia de la cadena eliminando los espacios en blanco anteriores y posteriores de la cadena.
cad1.toLowerCase()	Genera una copia de la cadena con todos los caracteres a minúscula.
cad1.toUpperCase()	Genera una copia de la cadena con todos los caracteres a mayúsculas.
cad1.indexOf(cad2) 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 <i>true</i> si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará <i>false</i> .
cad1.startsWith(cad2)	Retornará <i>true</i> si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará <i>false</i> .
cad1.endsWith(cad2)	Retornará <i>true</i> si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará <i>false</i> .
cad1.replace(cad2,cad3)	Generará una copia de la cadena <i>cad1</i> , en la que se reemplazarán todas las apariciones de <i>cad2</i> por <i>cad3</i> . El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "zzz" por "xx" en la cadena "zzzzz" generará "xxzz" y no "zzxx".

4.6 Más operaciones avanzadas con cadenas de caracteres

El principal problema de las cadenas de caracteres radica en su alto consumo de memoria. Cuando escribimos 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 (objetos que una vez creados se pueden modificar sin problemas), permitiendo así una mayor

optimización de la memoria. También existe la clase `StringBuffer`, pensada para aplicaciones multihilo (que procesan varias líneas de ejecución simultáneamente y que necesitan acceder a datos compartidos, pero esta clase es un poco delicada). En nuestro caso nos centraremos en la primera.

Pero, ¿en qué 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 ejemplo de uso de esta clase. Partimos de una cadena con errores, que modificaremos para ir haciéndola legible. En primer lugar creamos la instancia de esta clase, 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:

```
strb.delete(6,8); //Eliminamos las 'uu' que sobran
```

Hay que pasar como primer argumento la posición 6 (posición inicial) y como segundo argumento la posición 8 (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`)

```
strb.append ("!"); // Añadimos al final el cierre de exclamación  
strb.insert (0,"i"); // Insertamos la apertura de exclamación.  
strb.replace (3,5,"la"); // Reemplazamos 'al' por 'la'
```

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

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

<http://download.oracle.com/javase/tutorial/java/data/buffers.html>

4.7 Expresiones regulares. Definición.

¿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 (Modelo con el que encaja la información que estamos analizando o que simplemente se ha utilizado para construir dicha información). Un patrón, en el caso de la informática, está constituido por una serie de reglas fijas que deben cumplirse o ser seguidas para formar la información. Las

direcciones de correo electrónico, por ejemplo, todas tienen la misma forma, y eso es porque todas siguen el mismo patrón para ser construidas) que podría describirse como:

- una letra inicial opcional (solo presente en los NIE), seguida de una secuencia numérica y finalizando con otra letra.

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 "a".
- "[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.
- "a?". Usaremos el interrogante para indicar que un símbolo puede aparecer una vez o ninguna. 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 "aaaaaaaaa".
- "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.

Con lo visto hasta ahora ya es posible construir una expresión regular capaz de verificar si una cadena contiene un DNI o un NIE. Esta expresión sería:

```
"[XYxy]?[0-9]{1,9}[A-Za-z]";
```

 aunque no es la única solución.

4.8 Expresiones regulares. Clases "Pattern" y "Matcher"

Para el uso de expresiones regulares, Java ofrece las clases Pattern y Matcher contenidas en el paquete `java.util.regex.*`. La clase Pattern se utiliza para procesar la expresión regular y "compilarla", 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. Veámoslo 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 (cadenas que contienen en su interior más de un salto de línea a veces no se procesan bien con las expresiones regulares, dado que estas se limitan a verificar generalmente solo la primera línea. El modo multilínea permite buscar líneas completas que encajen con un patrón dentro de una cadena que contiene varias líneas) y con el método `find()` .
- `"."` El punto simboliza cualquier carácter.
- La barra invertida también se usa como carácter de reemplazo con diferentes significados:

`"\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 en una palabra. Equivale a `"[a-zA-Z0-9]"`.

4.9 Expresiones regulares.

Hasta ahora hemos visto como las expresiones regulares permiten verificar datos de entrada, para 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. Veamos 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));
}
```

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.

Atención porque el primer grupo es el 1, y no el 0. Si se pone `m.group(0)` se obtiene 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.

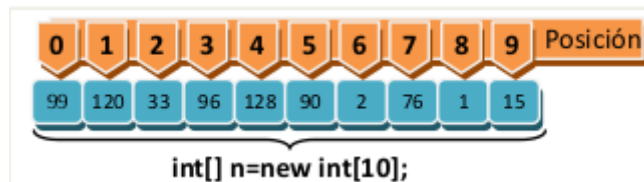
En el ejemplo anterior se usa el método `find`, éste buscará una a una, cada una de las ocurrencias del patrón en la cadena. Cada vez que se invoca, busca la siguiente ocurrencia del patrón y devolverá `true` si ha encontrado una ocurrencia. Si no encuentra en una iteración ninguna ocurrencia es porque 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, `"\"` significará que debe haber un paréntesis en la cadena y se omitirá el significado especial del paréntesis. Lo mismo ocurre con `"\"`, `"\"`, `"\"`, etc. La excepción son las comillas, que se pondrían con una sola barra: `"\"`

5 Arrays

Todos los lenguajes de programación permiten el uso de arrays:



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[dimensión]`, 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 un espacio en memoria.  
int[] m=new int[10]; // Declaración y creación en un mismo lugar.
```

Ya podemos almacenar valores en cada posición, 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.

5.1 Uso de arrays unidimensionales

Existen tres operaciones principales con arrays:

- modificación de una posición del array
- acceso a una posición del array
- paso de parámetros.

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

```
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, poniendo el nombre del array y la posición deseada entre corchetes:

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

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:

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

El tercer uso principal de los arrays es en el paso de parámetros. Por 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.

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

En Java las variables se pasan por copia (por valor) a los métodos. Si 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 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, sí se cambia su valor de forma definitiva. Veamos un ejemplo:

```
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  
    */  
}
```

5.2 Inicialización.

Rellenar un array, para su primera utilización, es una tarea muy habitual que puede ser rápidamente simplificada. 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 relleno 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. Este uso suele ahorrar bastantes líneas de código en tareas repetitivas. Cuando el número de elementos es fijo y conocido a priori, se puede especificar entre llaves el listado de valores que tiene el array:

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

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 compleja, 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:

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

Para acceder a una propiedad o a un método cuando los elementos del array son objetos, se puede usar la notación de punto detrás de los corchetes, por ejemplo: diasemana[0].length.

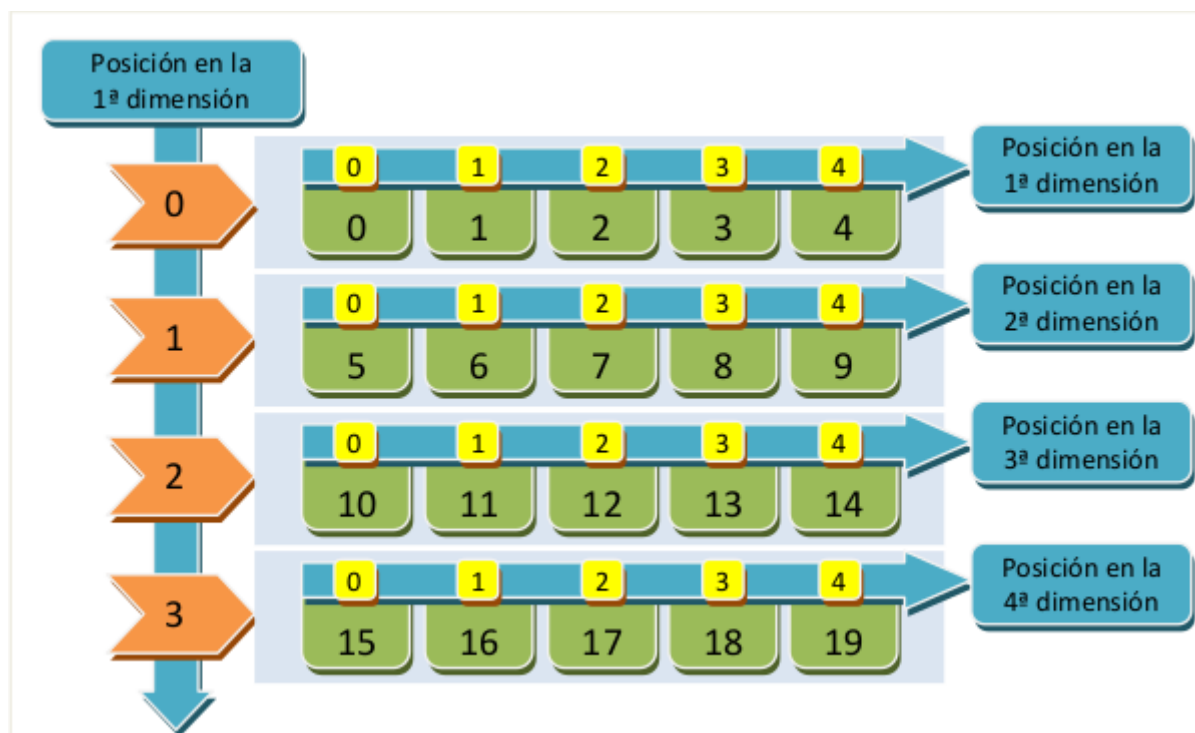
5.3 Arrays multidimensionales

Los arrays multidimensionales nos permiten representar una de las estructuras más usadas en matemáticas: la matriz. Es un array con dos dimensiones, o lo que es lo mismo, un array cuyos elementos son arrays de números.

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:



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 se hace 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];
```

5.4 Uso de arrays multidimensionales

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 empiezan a numerarse en 0 y que la última posición es el tamaño de la dimensión en cuestión menos 1.

Se puede asignar un valor concreto a una posición:

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

Y se puede acceder al valor de una posición poniendo el nombre del array y los índices del elemento 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];
```

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;  
}
```

El atributo "length", aplicado directamente sobre el array, nos permite saber el tamaño de la primera dimensión (a2d.length). Para saber el tamaño de una dimensión superior tenemos que poner los índices entre corchetes seguidos de length (a2d[i1].length). Gracias a esto podemos tener arrays multidimensionales irregulares.

Una matriz es un ejemplo de array multidimensional regular, porque 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];
```

Cuando se usen arrays irregulares, es conveniente verificar 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) {...}
```

5.5 Inicialización de arrays multidimensionales

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. 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:

```
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. A partir de tres dimensiones resulta muy complejo manejar arrays. Utilizando esta notación también podemos inicializar rápidamente arrays irregulares, separando 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**.