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. **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.

1.- Introducción a las estructuras de almacenamiento.

¿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. 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 trabajaremos más en profundidad.

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

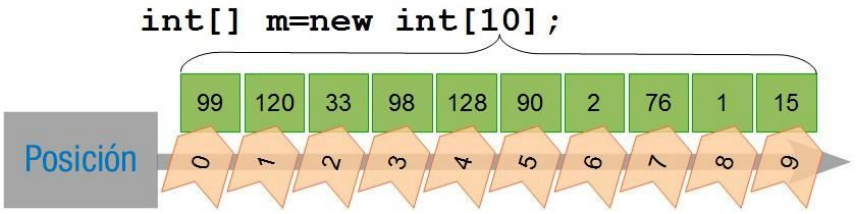
- **Estructuras cuyo tamaño se establece en el momento de la creación o definición** y su tamaño no puede variar después. Ejemplos de estas estructuras son los **arrays** y las matrices (arrays multidimensionales).
- **Estructuras cuyo tamaño es variable (conocidas como estructuras dinámicas)**. Su tamaño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

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

2.- Creación de arrays.



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.

2.1.- Uso de arrays unidimensionales.

Ya sabes declarar y crear de arrays, pero, ¿cómo y cuándo se usan? Pues existen varios ámbitos: modificación de una posición del array y acceso a una posición del array.

```
nombrearray[posicion]
```

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);
```

2.2.- Inicialización.

Rellenar un array, para su primera utilización, es una tarea muy habitual que puede ser rápidamente simplificada.

```
int[] vector = new int[totalNumeros];
for (int i=0; i< vector.length; i++){
        vector[i]=i;
}
```

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 sólo 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[] vector = new StringBuilder[10];
for (int i=0; i<vector.length; i++) {
 System.out.println("Valor"+i+"="+vector[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[] vector = new StringBuilder[10];
for (int i=0; i<vector.length; i++){
   vector[i] = new StringBuilder("cadena"+i);  //en cada posición se crea un objeto
}
```

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**. Fíjate 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));
```

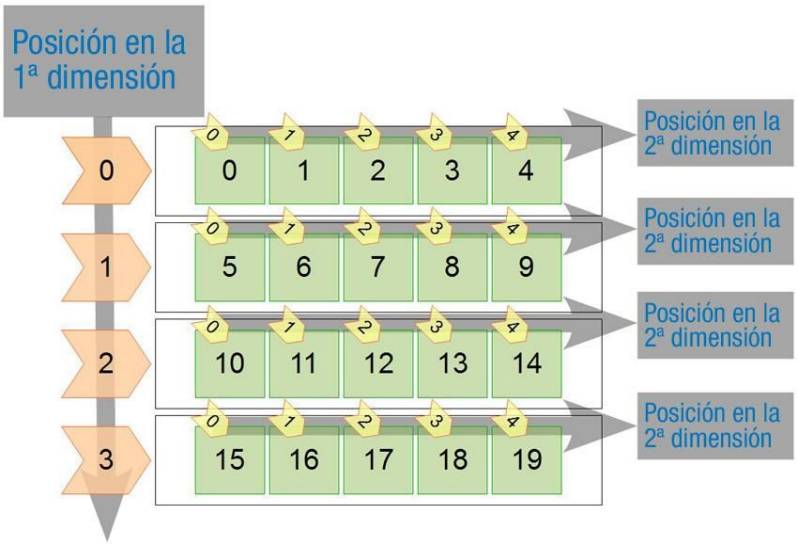
3.- Arrays multidimensionales.

¿Qué estructura de datos utilizarías para almacenar los píxeles de una imagen digital? Normalmente las imágenes son cuadradas 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[][] matriz = 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.

3.1.- Uso de arrays multidimensionales.

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

```
int[][] matriz= 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:

```
matriz[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:

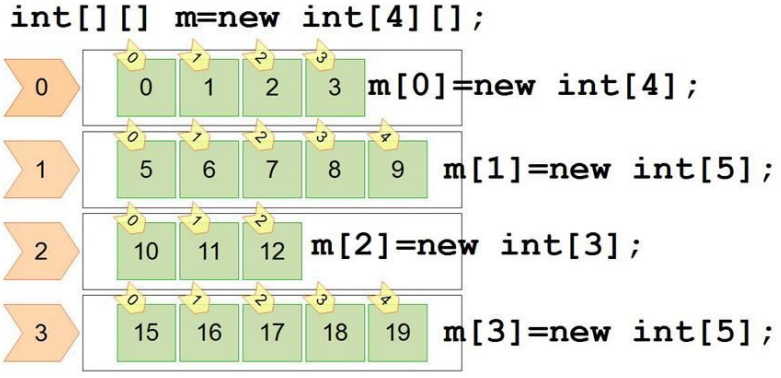
```
int suma = matriz[0][0] + matriz[0][1] + matriz[0][2] + matriz[0][3] + matriz[0][4];
```

Recorrido de un array de dos dimensiones. Por ejemplo:

```
int suma = 0;
for (int fila= 0; fila < matriz.length; fila++) {
    for (int columna = 0; columna < matriz[fila].length; columna++) {
         suma += matriz[fila][columna];
    }
 }
```

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 (**matriz.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** (**matriz[fila].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 cómo 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) {...}
```

3.2.- 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.

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}};
```

El array anterior sería un array de 4 por 3. Como puedes observar 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}};
```

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.