Programación en Java:

Tablas

1º DAM

Tablas

Una tabla es una estructura en el que guardamos varios valores de manera simultánea y se lo asociamos a una sola variable.

Importante:

A diferencia de lo que vimos con **Python**, en el que las tablas o arrays son **estructuras dinámicas tanto en tamaño como en el tipo** de información que almacen, en **Java** estas **estructuras son estáticas,** es decir, deben crearse con un **tamaño fijo** y para almacenar un **tipo concreto.**

```
tabla = []
```

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

Python **VS** **Java**

Tablas: Construcción

Las tablas son estructuras estáticas:

- Debo conocer y controlar el tamaño de mis arrays en el momento de la definición de la misma.
- NO puedo almacenar datos de distintos tipos en una misma tabla o array.

Para trabajar con tablas debemos realizar dos etapas:

1. **Declaración** de una variable que contiene una tabla en Java:

```
tipoDeDatos[] nombreTabla
```

```
tipoDeDatos nombreTabla []
```

Ejemplo:

```
int [] edad;
```

```
int edad [];
```

Tablas: Construcción

2. **Definición.** En esta fase es en la que se hace la reserva de memoria según el tipo y el tamaño definido.

   El operador new construye una tabla donde todos los elementos se inicializan a 0, para tipo numérico o a false para tipo booleano.

```
nombreTabla = new int[entero_tamaño];
```

Ejemplo:

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

Es posible declarar y definir la variable table en una misma línea:

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

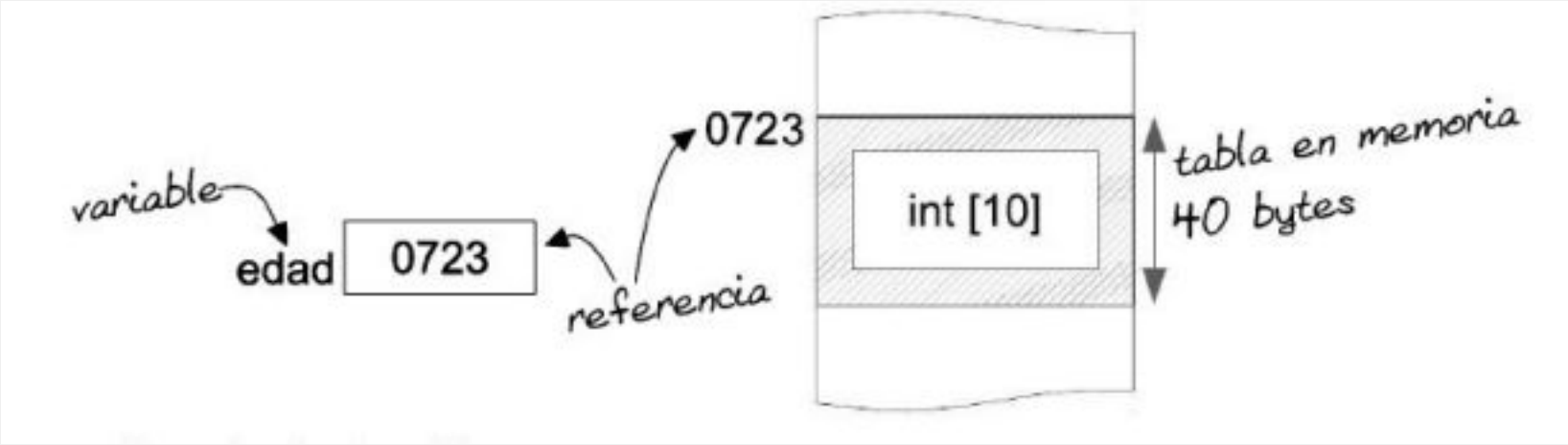
Tablas: Construcción

2. **Definición** `edad = new int[5];`

Veamos en detalle cómo funciona el operador new:

1. Reserva memoria para almacenar 5 elementos de tipo entero.
2. Recorre todos los elementos de la tabla inicializándolos a su valor por defecto.

Las variables de las tablas realmente almacenan una referencia a la posición de memoria donde se guarda la posición 0 es por ello que en Java, si hago, obtendré algo similar a `[I@6bf2d08e`



Tablas: Construcción

Construcción por asignación

Es posible declarar y definir una tabla, asignándole directamente otra tabla.

```
int [] tabla = {1,2,3};
```

Esto sería equivalente a:

```
int[] tabla = new int[3];
tabla[0]= 1;
tabla[1]= 2;
tabla[2]= 3;
```

Importante:

La construcción por asignación, sólo es posible cuando se realiza la definición y la declaración en la misma línea.

Tablas: Construcción

Importante:

La construcción por asignación, sólo es posible cuando se realiza la definición y la declaración en la misma línea.

```
int datos[] = {2, -3, 0, 7}; //tabla de tamaño 4
```

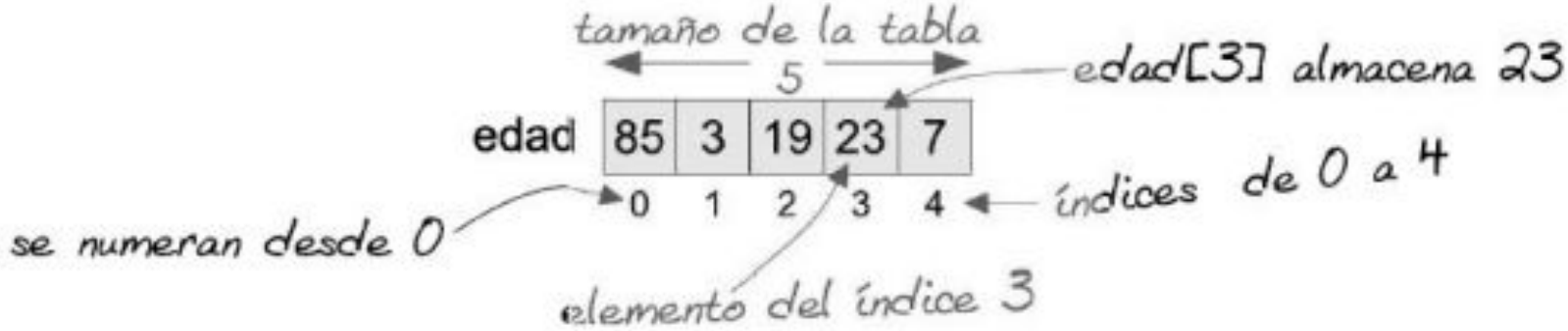
No pudiéndose hacer en dos etapas diferenciadas:

```
int datos[];
datos = {2, -3, 0, 7} //no permitido, solo en la declaración
```

Tablas: Acceso

Como ya conoces el acceso a un valor concreto, se realiza mediante su **índice**.

La forma de utilizar un elemento concreto de una tabla es a través del nombre de la variable junto con la posición en la que se encuentra, comenzando por 0, y delimitado por []




```
System.out.println(edad[2]);
edad[3] = 8;
```

Tablas: Acceso: Índices fuera de rango

Como se ha comentado, las tablas tienen un tamaño fijo, es por ello que NO puedo acceder a una posición más allá de su tamaño. Es decir:

Si el tamaño es X, sus índices irán desde el 0 hasta X-1

¿Qué ocurre si intento a acceder con un índice que está fuera de rango? Obtendrás un error o excepción en tiempo de ejecución del programa.

Estos errores no se detectan en tiempo de compilación.

```
Exception in thread "main" 3
java.lang.ArrayIndexOutOfBoundsException: Index 9 out of bounds for length 5
```

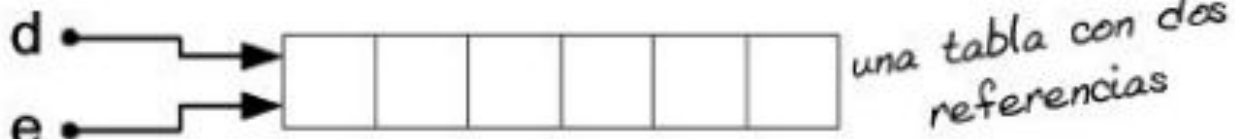
Tablas: Referencias

Anteriormente, hemos visto que las variables de tablas realmente guardan la dirección de memoria dónde comienza la tabla.

Es por ello, que dos variables podrían apuntar a la misma dirección de memoria, es decir, es posible acceder a una misma tabla mediante más de o variable, o lo que es lo mismo, que una tabla esté referenciada por más de una variable.

Por ejemplo:

```
int d[], e[]; //declaración
d = new int[6]; //definición d
e = d;
```



Tablas: Recorridos

Podríamos recorrer una tabla a través de su índice, teniendo en cuenta que el valor máximo es el tamaño -14

```
String[] coches = {"Volvo", "BMW", "Ford"};
for (int i = 0; i < coches.length; i++)
{
   System.out.println(coches[i]);
}
```

Tablas: Recorridos

También podríamos hacer un segundo recorrido de la siguiente forma:

```
String[] coches = {"Volvo", "BMW", "Ford"};
for (String coche : coches)
{
   System.out.println(coche);
}
```

Tablas: Clase Arrays

Java tiene definido una clase para facilitar el trabajo con tablas.

Esta clase contiene métodos estáticos para manipular y realizar operaciones con tablas.

```
import java.util.Arrays;
```

Vamos a entender qué funcionalidad tiene esta clase

Tablas: Clase Arrays

- **Mostrar el contenido de una tabla: toString()**

```
int[] edades = {8, 41, 24, 7};
System.out.println(edades);
```

Salida: `[I@6bf2d08e`

Como se ha explicado, esto devolvería la referencia a la posición de memoria dónde está almacenado nuestra tabla.
Es por ello que podemos apoyarnos en la clase Arrays para convertir su contenido en una cadena y que se pueda imprimir con el print.

```
System.out.println(Arrays.toString(edades));
```

Salida: `[8, 41, 24, 7]`

Tablas: Clase Arrays

- **Inicializar valores de un tabla: fill()**

Como se ha visto antes, para almacenar valores en la tabla tendríamos que ir posición a posición o hacerlo junto con la declaración

```
int[] edades = new int[3];
edades[0]= 8;
edades[1] = 5;
```

El método fill puede simplificar esta tarea.

Tablas: Clase Arrays

- **Inicializar valores de un tabla: fill()**

El método fill puede simplificar esta tarea. Tiene dos maneras de ser llamado:

1. Inicializando todo el contenido de la tabla

```
static void fill(tipo t[], tipo valor):
```

Ejemplo:

```
Arrays.fill(edades, 22);
```

Ahora todos los valores de la tabla edades tendrán el valor 22

Tablas: Clase Arrays

- **Inicializar valores de un tabla: fill()**

2. Inicializando un rango de elementos de la tabla

```
static void fill(tipo t[], int desde, int hasta, tipo valor):
```

Ejemplo:

```
Arrays.fill(edades, 1,2, 22);
```

Ahora la posición 1 y 2 de la tabla edades tendrán el valor 22 pero la 0 valdrá su valor por defecto (0)

Tablas: Clase Arrays

- **Comparar dos tablas: equals**

  ¿Qué ocurre si comparo dos tablas con el ==?

  Cada tabla apunta a una zona de memoria diferente, por lo que al preguntar con == aunque tengan el mismo contenido, nos dirá que no son las mismas.

Para comparar el contenido de dos tablas debemos usar el método equals

```
static boolean equals(tipo a[], tipo b[]):
```

Ejemplo:

```
int[] edades = {7,9,10};
int[] edades2 = {7,9,10};
System.out.println(edades == edades2);
System.out.println(Arrays.compare(edades, edades2));
```

Tablas: Clase Arrays

- **Búsqueda en tablas ORDENADAS: binarySearch**

Si la tabla está ordenada crecientemente, sabemos que, dada una posición en concreto, ejemplo edades[1]

- ❏ Los elementos que se encuentran en posiciones menores que 1 serán menores o iguales que el que se encuentra en edades[1]
- ❏ Los que se encuentra después de edades[1] serán igual o mayor que él.

El método `static int binarySearch(tipo t[], tipo aBuscar)`

nos devolverá:

- ❏ Valor menor que 0, si no existe un elemento con ese valor. El número que devuelve, sería en la posición en el que habría que insertar el elemento para que la lista siguiera estando ordenada
- ❏ La posición en la que ha encontrado

Tablas: Clase Arrays

- **Búsqueda en tablas ORDENADAS: binarySearch**

  Ejemplo:

```
int pos = Arrays.binarySearch(precios, 19.95);
if (pos >= 0) {
   System.out.println("Encontrado en el índice: " + pos);
} else {
   System.out.println("Lo sentimos, no se ha encontrado.");
}
```

Tablas: Clase Arrays

- **Ordenación: sort**

EL proceso de ordenación es muy costoso, es por ello que se recomienda usarlo, sólo cuando sea necesario.

```
static void sort(tipo t[])
```

Ejemplo:

```
int[] edades = {17, 58, 7,9,10, 28, 34};
Arrays.sort(edades);
```

Tablas: Clase Arrays

- **Copia: copyOf**

  Para realizar una copia exacta tendremos que:

  ❏ Crear nueva tabla del mismo tamaño
  ❏ Ir guardando elemento a elemento

  El método copyOf nos simplifica esta tarea.

  De nuevo, este método puede llamarse de dos formas:

1. Indicando cuántos elementos queremos copiar

```
static tipo[] copyOf(tipo origen[], int longitud)
```

```
int t[] = {1, 2, 1, 6, 23};
int a[], b[];
a = Arrays.copyOf(t, 3); //a = [1, 2, 1]
b = Arrays.copyOf(t, 10); //b = [1, 2, 1, 6, 23, 0, 0, 0, 0, 0]
```

Tablas: Clase Arrays

- **Copia: copyOf**

2. Indicando desde dónde y hasta dónde queremos copiar

```
static tipo[] copyOfRange(tipo origen[], int desde, int hasta):
```

```
int t[] = {7, 5, 3, 1, 0};
int a[] = Arrays.copyOf(t, 1, 4); //a = [5, 3, 1]
```

Tablas: Clase Arrays

- **Inserción en tabla NO ORDENADA**

  En este caso, lo podemos insertar al final. Tan sólo tenemos que controlar que hay posiciones libres.

```
/* inserccción al final: en el primer elemento disponible */
if (numElem == t.length) {
    // tabla llena, no insertamos
    ...
} else {
   t[numElem] = nuevo;
   numElem ++; //ahora tenemos un elemento más
}
```

Tablas: Clase Arrays

- **Inserción en tabla ORDENADA**

  Ahora, para insertarlo, no valdrá hacerlo al final. Tendremos que:

  1. Buscar en qué posición debemos insertar.
  2. Desplazar desde esta posición hasta el final para hacerle hueco
  3. Realizar la inserción

Tablas: Clase Arrays

- **Inserción en tabla ORDENADA** Ejemplo

```
int[] edades = {7,9,10, 28, 34};
int eltoBuscar = 27;
int posicion = Arrays.binarySearch(edades, eltoBuscar);
System.out.println(posicion);
if(posicion < 0)
   posicion = -posicion;
for(int i = posicion; i < edades.length; i++)
{
   edades[i] = edades[i-1];
}
edades[posicion-1] = eltoBuscar;
System.out.println(Arrays.toString(edades));
```


¿Dudas?