Un **Catálogo** de discos

Como te dijimos al principio de este curso, el objetivo final es el de emular una empresa de streaming que maneja un catálogo de discos. Este catálogo posee una colección de discos, organizados en una lista.

El catálogo será usado de la siguiente manera:

- Podrá iniciar su funcionamiento con un catálogo inicial de discos o bien ir comprando los discos uno por uno o en una colección conforme va funcionando.
- Una vez registrado un catálogo, éste dará sus servicios a través de Internet mediante un menú.
- Las opciones que dará el catálogo son:
  - Agregar un disco al catálogo
  - Iniciar el catálogo con una colección de discos
  - Mostrar el catálogo de discos
  - Iniciar una transmisión de un disco seleccionado y registrarla
  - Mostrar los discos que están en transmisión
  - Para una transmisión activa de un disco dado:
    - Terminar la transmisión y liberarla
    - Registrar el fin de una transmisión en el histórico de transmisiones para ese disco
  - Mostrar el histórico de transmisiones para cada disco
  - Mostrar un menú con las opciones disponibles

De esta descripción puedes hacer tarjetas de responsabilidades. Analizarás de lo general a lo particular, por lo que empezarás con el catálogo de discos (aunque ya tienes la clase `Disco`).

Clase: **Catalogo**

Datos:

Nombre:	**Descripción**
catalogo	Colección de discos. Cada disco es un objeto.
Para el catálogo:	
numDiscos	Número de discos en el catálogo.
fechas	Colección de fechas de transmisión para cada disco registrado. Cada registro tiene la fecha de inicio.
Para llevar registro:	
historico	Colección para cada disco de parejas de fechas que guarda el inicio y final de una transmisión
numHist	Colección que guarda el número de registros históricos para cada disco.

También de la lista dada puedes obtener las responsabilidades que debe cumplir la clase Catalogo.

Clase: Catalogo			
Responsabilidades:			
Nombre	**Salida**	**Entradas**	**Descripción**
Constructores:			
Catalogo	catálogo	(nada)	Construye un catálogo con el máximo número de posiciones y todos los arreglos asociados
Catalogo	catálogo	numDscs	Construye un catálogo con numDscs posiciones y todos los arreglos asociados
Catalogo	catálogo	numDscs, arreglo inicial de discos	Construye un catálogo con numDscs posiciones con nuevos como contenido inicial y todos los arreglos asociados
addCatalogo	boolean	Disco	Agrega un disco al catalogo si es que hay lugar. Dice si lo agregó o no
daTransmision	boolean	posición del disco	Registra la transmisión del disco elegido a una cierta hora. Avisa si pudo o no.
terminaTrans	boolean	posición del disco y consola	Pregunta, de este disco, cual es la transmisión a eliminar. Transcribe el inicio y final al histórico. Avisa si pudo o no.
mstraCatalogo	Cadena con lista de discos	encabezado	Muestra la lista de discos dados de alta
mstraActivos	Cadena con la lista de discos en transmisión	encabezado	Da una lista de los discos en transmision junto con la lista de hora de inicio
mstraActivas	Cadena con la lista de transmisiones activas	posición del disco	Da una lista de las transmisiones activas del disco en esa posición del catálogo
mstraHist	Cadena con el histórico del disco	posición del disco	Muestra la lista de transmisiones iniciadas y finalizadas para ese disco.
mstraHistrs	Cadena con el histórico de todos los discos	(nada)	Muestra la lista de transmisiones iniciadas y finalizadas para todos los discos
conectaCatlgo	(nada)	(nada)	Es el encargado de la interacción entre el usuario y el catálogo

Respecto a los datos que requiere tener el Catálogo, tienes ya la clase para los discos, excepto que para registrar la transmisión del disco el método que tienes genera su propia fecha y hora, que usa para regresar una cadena *interpretando* ese calendario. Recuerdas el encabezado:

```
public String daTransmision ( );
```

En este proyecto, en cambio, es el catálogo el que define la hora, cuando en el menú se elija esa opción. Lo único que tienes que hacer es agregar en la clase `Disco` un método que dé una transmisión, pero que reciba como entrada la fecha en que da la transmisión. Queda así en la tarjeta de responsabilidades que ya tienes de la clase `Disco`:

Clase: Disco			(agregar)
Responsabilidades o métodos			
Nombre	**Salida**	**Entradas**	**Descripción**
daTransmision	Mensaje	calendario	Si es posible, otorga una transmisión del disco al que se le solicita la transmisión

Java distingue entre los dos métodos `daTransmision` dependiendo si lo llamas con el argumento de la fecha o sin él. Como las firmas son distintas, no tiene ningún problema para distinguirlos. Harás ahora la implementación de este nuevo método.

Abres en DrJava la clase `Disco`. Puedes implementar este nuevo método inmediatamente después del que tenías originalmente. Copias el método junto con su documentación de Javadoc. En esta documentación agregas que recibe como parámetro un GregorianCalendar. Asimismo cambias el encabezado del método para que reciba el calendario y eliminas dentro del nuevo método la declaración local y construcción del mismo, usando directamente el que se le pasa como parámetro. Copias el método que ya tienes para `daTransmision` en la clase `Disco`, poniéndole como parámetro a `GregorianCalendar miCal` y usándolo directamente en lugar de crearlo.

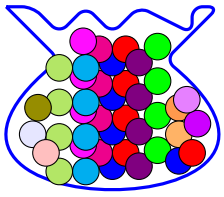
```
/**
 * Otorga una transmision, interpretando la fecha y hora en que
 * la esta dando. Si no la puede dar, responde negativamente.
 * Actualiza el numero de transmisiones activas.
 * @param miCal La fecha y hora en la que se esta pidiendo
 *              la transmision.
 * @return     Un mensaje diciendo si pudo o no otorgar la
 *              transmision y con la fecha y hora si pudo.
 */
public String daTransmision(GregorianCalendar miCal) {
  boolean siHay = activas < permitidas;
  activas += siHay ? 1 : 0;
  return siHay
    ? ("Transmision␣dada␣a␣las" + daHora(miCal)
        + "del␣" + daFecha(miCal))
    : "␣No␣hay␣transmisiones␣disponible";
}
```

Compilas nuevamente este método y ves que todo está bien. Cierras el archivo de la clase `Disco` y ya resuelta esta discrepancia con el catálogo de discos, empiezas a diseñar al catálogo, que es una colección de discos.

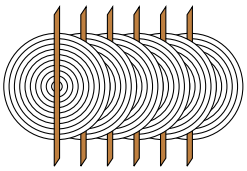
Catálogo de discos

Como ya escuchaste antes, en este módulo estarás trabajando con *colecciones* de objetos. El catálogo de discos tiene una colección de discos. Para cada disco de esa colección, el catálogo mantiene una colección de fechas en las que fueron concedidas transmisiones en curso y una colección más del histórico de transmisiones, con el inicio y fin de cada transmisión. El tamaño de la primera es el número de transmisiones permitidas en el disco correspondiente, pues no se podrá dar transmisión alguna si éstas ya están completas. El histórico tendrá el tamaño máximo de transmisiones permitidas a un disco, pues puede haber más de las permitidas mientras no sean simultáneas. Estas colecciones contienen únicamente las referencias a los objetos dados: discos y fechas, que fueron construidos una sola vez en la memoria (el heap) de la computadora que está ejecutando el programa.

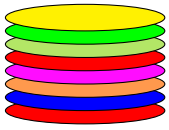
La clase que corresponde a cada disco ya la tienes, pero no sabes todavía cómo definir "colecciones", ya sea de objetos o datos primitivos. A las "colecciones" se les llama, en programación *estructuras de datos*, porque se refieren a la manera de asociar (estructurar) entre sí a datos. Un primer ejemplo de una estructura de datos es el concepto de *clase*, en el que agrupamos datos de distintos tipos. Otros ejemplos de colecciones son:



Bolsa de canicas



Estante de discos



Pila de platos

La estructura de datos más comúnmente usada en programación es la que corresponde a los *arreglos*.

Arreglos

En Java los arreglos corresponden a objetos que agrupan datos del mismo tipo. Como su uso es tan común, en lugar de métodos de consulta y actualización, o constructores específicos, tienes operadores para ello, haciendo más conciso, fácil y directo su uso.

- Como ya escuchaste, son la estructura de datos más usada en programación y casi todos los lenguajes de programación presentan alguna variación sobre el tema.
- Un arreglo es un objeto, pero por ser tan comunes hay operadores específicos, lo que les da una notación particular concisa y fácil para manejarlos.
- Los arreglos agrupan datos del mismo tipo y los organizan entre sí, lo que los hace homogéneos. Por ejemplo, una zapatera en la que colocamos pares de zapatos; un librero; un buzón para colocar allí las llaves de los cuartos de un hotel; un estacionamiento, todos ellos son arreglos homogéneos porque contienen datos del mismo tipo.
- Tienen un orden por lo que están organizados como listas, asignándoles posiciones en la lista, lo que los hace estructuras lineales. Piensa en un edificio de estacionamiento con varios niveles, cada nivel varias filas y cada fila con varios lugares. Puedes hacer una lista de los

lugares dando primero el nivel, luego la fila y por último el lugar dentro de esa fila. En cada lugar se puede estacionar un vehículo.

- Puedes obtener un dato específico dando su ubicación dentro del arreglo, lo que los hace estructuras de acceso directo. Por ejemplo, si quieres saber si un lugar está ocupado o no, puedes ver el mapa de foquitos del estacionamiento y simplemente verificar el foquito del nivel, fila, columna.
- Una vez construido el arreglo no puedes disminuir o aumentar su capacidad, lo que los hace estructuras estáticas. Pueden tener espacio disponible y puedes eliminar a alguno de los elementos que ocupa un espacio, pero el número de espacios queda fijo en la construcción del arreglo.
- Cada elemento de un arreglo puede ser, a su vez, un arreglo (y así sucesivamente) por lo que se dice que pueden tener una o más dimensiones. En el ejemplo del edificio de estacionamiento tienes tres dimensiones: el nivel, la fila y la posición en la fila.

Los arreglos agrupan a los datos (objetos o de tipo primitivo) acomodando a cada dato en un lugar específico.

Dimensiones en arreglos

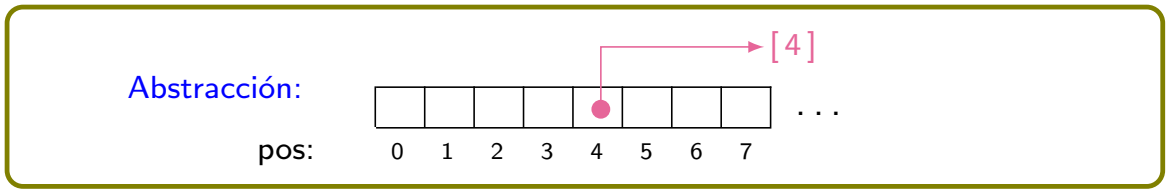
A continuación verás la sintaxis de Java para construir arreglos, cuál es la forma en que piensas en un arreglo y, finalmente, cómo los acomoda la Máquina Virtual de Java en la memoria durante la ejecución.

El número de dimensiones queda definido por el número de corchetes en la declaración. Si el arreglo es de una sola dimensión (un único renglón) queda declarado como sigue:

```
⟨tipo⟩ [ ] ⟨identif⟩ // Una dimension
```

- ⟨tipo⟩ es el tipo que van a tener **todos los elementos** del arreglo, que puede ser primitivo (int, boolean, ...), una clase (Disco, GregorianCalendar, ...) o una interfaz.
- El identificador ⟨identif⟩ se nombra con las reglas de Java.
- La presencia de una sola pareja de corchetes indica que es de una sola dimensión.

Podemos *pensar* en los arreglos de una dimensión como se muestra en la figura.



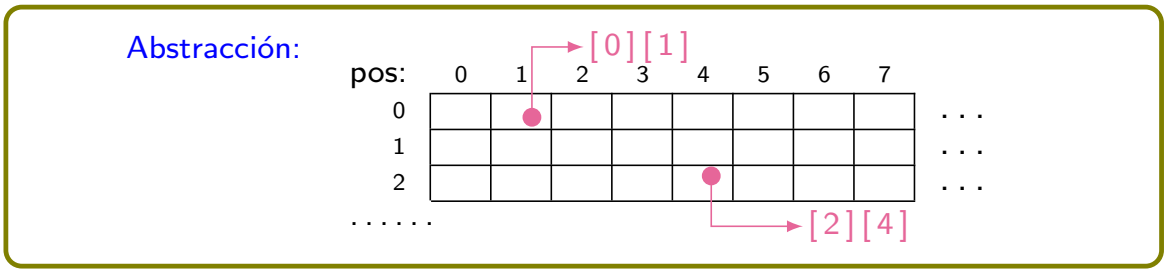
El número en rojo indica la ubicación de un elemento en el único renglón, por lo que necesita únicamente una pareja corchetes y una expresión entera. En este caso se trata del quinto elemento del arreglo (los índices de los arreglos siempre empiezan en 0 (cero).

Arreglos de dos dimensiones

Si el arreglo es de dos dimensiones (uno o más renglones, una o más columnas), queda declarado con la siguiente sintaxis:

⟨tipo⟩ [ ][ ] ⟨identif⟩ *// Dos dimensiones: arreglo de arreglos*

- ⟨tipo⟩ es como en el caso anterior, el tipo que va a tener cada uno de los elementos de la estructura.
- Usas **dos** pares de corchetes para indicar **dos** dimensiones.
- Piensas en una tabla o matriz, pero donde todas las columnas son del mismo tipo y se distinguen entre sí por la posición en la que están.



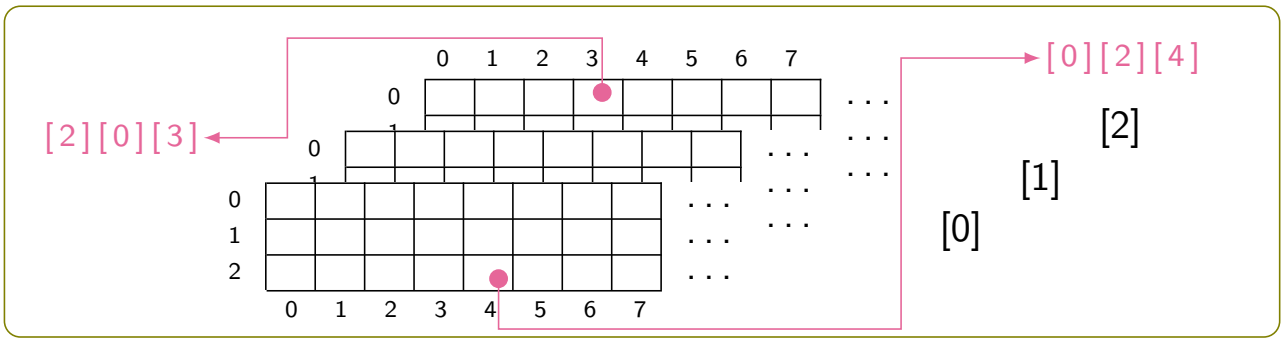
En este arreglo, arriba, tienes marcado, con dos parejas de corchetes, al segundo elemento (índice 1 en el segundo corchete) del primer renglón (índice 0 dentro de la primera pareja de corchetes). Para llegar a un elemento particular de un arreglo, vas desde afuera hacia adentro en el orden que das las expresiones enteras dentro de los corchetes (no tienen que ser constantes).

Arreglos de tres dimensiones

Si el arreglo es de **tres** dimensiones, debe haber **tres** parejas de corchetes:

⟨tipo⟩ [ ][ ][ ] ⟨identif⟩ *// 3 dimensiones: arreglo de arreglos de arreglos*

Concebimos un arreglo de tres dimensiones como un cubo hecho de "caras" o un paquete de hojas cuadriculadas, una hoja detrás de la otra, y cada hoja es una tabla de renglones y columnas. Cada dimensión se numera desde cero.



- Para el elemento [2][0][3] eliges primero la tercera página, en ella el primer renglón y en él la cuarta columna.
- El elemento [0][2][4] lo encuentras en la primera página, tercer renglón y quinta columna.

Sintaxis en Java para arreglos

La declaración de un identificador como arreglo debe llenar la siguiente sintaxis:

<acceso> <modifs> <tipo><núm de dimnsions> <identif>;

Significado de esta especificación:

- <acceso> y <modifs> es sólo en declaración de campos o atributos, no en declaraciones locales y ya viste que pueden ser static, final para los modificadores.
- <tipo> corresponde al que van a tener todos y cada uno de los elementos del arreglo.
- <núm de dimnsions> lo defines por el número de parejas *sucesivas* de corchetes ([ ]), una pareja por cada dimensión.
- <identif> sigue las reglas de Java para identificadores.
- Puedes declarar más de un identificador del mismo tipo, separándolos entre sí por comas:

```
int[] enteros, pares, impares;
```

  Los tres identificadores corresponden a arreglos de enteros de una dimensión.

Te sugerimos algunos ejemplos:

- Arreglo de dos dimensiones cuyos elementos serán enteros:

```
int[][] juegosGanados;       // Tabla de equipos
```

- Arreglo de una dimensión cuyos elementos serán objetos de la clase Disco:

```
private Disco[] catalogo;    // catalogo de discos
```

- Arreglo de cadenas (objetos) de una dimensión, pero como es static (de clase) y final (constante) tienes que inicializarla en el momento de declarar: (aunque lo único que haremos ahora es dejarlo indicado):

```
public static final String[] NDIAS = // Dias de la semana
  /* Aca debe ir la inicializacion del arreglo */
```

- Tomas a la clase GregorianCalendar, y declaras un arreglo de dos dimensiones, con el primer renglón para el inicio de la transmisión y el segundo para el final de la transmisión. El tamaño de cada dimensión se da en la construcción (o definición).

```
private GregorianCalendar[][] historico; // fechas de inicio
                                         // y final
```

Construcción de arreglos con contenido en el momento de declaración

Lo que has hecho hasta ahora es ver nada más la declaración de los identificadores como arreglos de un cierto tipo de elementos. Como los arreglos son objetos, cuando se declara un identificador para un arreglo, como el arreglo no ha sido construido, el valor de la variables es null. No es sino hasta que se construyen los arreglos, ya sea en la misma declaración o posteriormente, que el espacio para los mismos es asignado en el heap y la dirección donde quedó ese objeto construido es asignada a la variable.

Puedes construir un arreglo directamente en la declaración. Si se trata de un arreglo constante (final) de clase (static) es obligatorio hacerlo en la declaración y determinar su contenido.

Tienes dos formas de construir un arreglo en la declaración, a continuación de un operador de asignación (=).

- **Con contenido:** Obligatorio para arreglos static y final, consiste en listar **todos** los componentes de cada dimensión entre llaves y separando los componentes por comas:

```
public static final String[] NDIAS = { "", // dia 0
                  "Domingo",              // dia 1
                  "Lunes",                // dia 2
                  "Martes",               // dia 3
                  "Miercoles",            // dia 4
                  "Jueves",               // dia 5
                  "Viernes",              // dia 6
                  "Sabado"};              // dia 7
```

  Como en GregorianCalendar el número del día empieza en 1, el primer elemento (en la posición 0) tiene una cadena vacía (distinto a una referencia null) y los nombres de los días están colocados en la posición dada por su número de día. Como cada nombre es una cadena que tiene asociada una posición en el arreglo, no tienes que preocuparte por su tamaño, como cuando lo hiciste cuando todos los nombres estaban en una misma cadena.

  Para un arreglo de dos dimensiones, encierras entre llaves todo el contenido del arreglo y después das el contenido de cada renglón, a su vez entre llaves:

```
 private int[][] juegosGanados = {{0,0,0,0,0}, // renglon 0
                   {0,3,0,4,3},               // renglon 1
                   {0,2,2,2,1},               // renglon 2
                   {0,3,3,3,1}                // renglon 3
                  };
```

  En esta definición (construcción y llenado) se trata de un arreglo de dos dimensiones con 4 renglones y 5 columnas. El primer renglón (0) tiene 5 posiciones con ceros y la primera columna de los demás renglones tienen cero como valor inicial.

  Cada renglón puede tener un número distinto de columnas, como sería la siguiente declaración:

```
 private int[][] juegosGanados = {{0}, // renglon 0
                   {0, 1},             // renglon 1
                   {0, 2, 2},          // renglon 2
                   {0, 3, 3, 3}        // renglon 3
                 };
```

  donde el primer renglón tiene una columna, el segundo dos columnas, el tercero tres columnas y el cuarto cuatro columnas.

- **Tamaño de dimensión:** Al declarar un arreglo de una o más dimensiones puedes indicar nada más el tamaño de cada dimensión con

  new ⟨ tipo⟩ [<expr entera>]...[<expr entera>]

  una [<expr entera>] por cada pareja de corchetes de la declaración.

Por ejemplo, para el arreglo NDIAS, si no lo tuvieses que llenar al declararlo (si no fuera constante de clase) podías haber hecho la siguiente declaración:

```
public String[] NDIAS = new String[8];  // del 0 al 7
```

Esta declaración hubiese construido un arreglo de cadenas con 8 posiciones, cada una de ellas con el valor null inicial.

La primera declaración de juegosGanados, haciendo lugar y llenando con ceros quedaría:

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

Después de new ⟨tipo⟩, excepto por el primero, puedes ir construyendo dimensión por dimensión, de izquierda a derecha, dejando el resto de las parejas de corchetes vacías. Por ejemplo, para la segunda declaración que hiciste, con número variable de columnas, podrías haber hecho la siguiente definición:

```
private int[][] juegosGanados;  // no tiene que ser en la declaracion
juegosGanados = new int[4][];   // Se construye renglon por renglon
juegosGanados[0] = new int[1];
juegosGanados[1] = new int[2];
juegosGanados[2] = new int[3];
juegosGanados[3] = new int[4];
```

Como Java ve a los arreglos de dos dimensiones como un arreglo de arreglos, cada uno de los arreglos de la segunda dimensión puede ser construido de manera individual y no forzosamente del mismo tamaño todos.

Como los arreglos son objetos, los elementos del arreglo se inicializan si son números en cero y si son objetos en null.

En cualquier caso en que los arreglos estén declarados los puedes construir en cualquier momento. También puedes reasignar el identificador de un arreglo a otro arreglo del mismo tipo (y mismo número de dimensiones).

Si deseas un arreglo de objetos de la clase Disco, puedes declararlo así:

```
private Disco[] catalogo = new Disco[MAX_DISCOS]; // del 0 a MAX_DISCOS - 1
```

- Tienes la declaración anterior en la que construiste un arreglo de una dimensión, cuyos elementos son **referencias** a objetos de la clase Disco.
- Quieres que tenga MAX_DISCOS elementos, del 0 a MAX_DISCOS - 1.
- MAX_DISCOS debería estar declarada como constante, de la clase en la que está esta declaración.
- Todos los elementos tienen, en este momento, null (una referencia nula) como valor
- El tamaño puede cambiar si vuelves a usar new con el mismo identificador (se pierde el contenido que tengas en ese momento en el viejo arreglo).
- Ya declarado el arreglo como atributo, podías haber definido su tamaño en otro lugar de la clase como, por ejemplo, en los constructores o cualquier otro método:

```
private Disco[] catalogo;
public Catalogo (int maxDiscos) {
  ...
  catalogo = new Disco[maxDiscos];
```

Construcción de arreglos con contenido, posterior a la declaración

Puedes usar la inicialización con llaves si listas el contenido inmediatamente después de la construcción con new.

```
private int[][] juegoGanados;                // declaracion
...
juegosGanados = new int[][] { {0,0,0,0},   // renglon 0
                  {0,3,0,4,3},              // renglon 1
                  {0,2,2,2,1},              // renglon 2
                  {0,3,3,3,1}               // renglon 3
                };
```

En el caso de un arreglo de objetos puedes poner entre las llaves invocaciones a los constructores de los objetos, como se hace a continuación con el arreglo de objetos de la clase Disco:

```
catalogo = new Disco[] {
              new Disco( (short)2, "Ahora los ves, ahora no", 1999, 5 ),
              new Disco( (short)3, "Billions", 2015, 4 ),
              new Disco( (short)3, "Outlander", 2016, 3 ),
              new Disco( (short)1, "Frank Sinatra", 1992, 2)
            };
```

El problema con llenar el contenido del arreglo al construirlo es que el número de elementos queda fijado en el número de objetos entre las llaves. Si dejamos así la construcción, el catalogo únicamente va a contar con 4 posiciones y no se podrán agregar más discos al catálogo.

Se puede subsanar este problema listando el valor por omisión (cero para números, null para objetos, false para booleanos) tantas veces como queramos que tenga el arreglo como tamaño:

```
catalogo = new Disco[] {
              new Disco( (short)2, "Ahora los ves, ahora no", 1999, 5 ),
              new Disco( (short)3, "Billions", 2015, 4 ),
              new Disco( (short)3, "Outlander", 2016, 3 ),
              new Disco( (short)1, "Frank Sinatra", 1992, 2),
              null, null, null, null, null, null, null, null, null, null, null
            };
```

Esto le daría al arreglo 15 posiciones. Por ejemplo, si quisiéramos 100 discos en el catálogo, tendrías que agregar 96 valores null a la lista entre las llaves (¡qué flojera!).

Construcción de arreglos

Como con cualquier objeto, en cualquier momento, (incluso en la declaración), puedes construir un arreglo donde cada uno de los elementos tenga el valor por omisión para su tipo. Para ello, utilizas el nombre del arreglo de lado izquierdo de una asignación y haces una asignación al identificador del arreglo con new.

Si el arreglo es de una dimensión dices cuántos elementos tiene el arreglo con la siguiente sintaxis:

⟨identif⟩ = new ⟨tipo⟩ [⟨ expr entera⟩] ;

A continuación del identificador escribes el operador de asignación (=) y el operador new. El ⟨tipo⟩ con el que estás construyendo tiene que ser el mismo con el que declaraste el identificador. A continuación vienen tantas parejas de corchetes (dimensiones) como tengas en la declaración del arreglo y dentro de cada pareja debe ir una ⟨ expr entera⟩ que entregue un entero.

Para la expresión que decide el tamaño en esa dimensión puedes invocar funciones, hacer aritmética o cualquier otra cosa que entregue un entero. Esta expresión dice el tamaño del arreglo, con las posiciones (o índices) empezando en 0. El índice mayor que tiene el arreglo es el que corresponde a (⟨ expr entera⟩ - 1).

Cuando construyes un arreglo indicando únicamente el tamaño de cada dimensión, como se trata de objetos, se inicializa a cada uno de los elementos con el valor por omisión según su tipo:

- Si ⟨tipo⟩ es numérico el arreglo se llena de ceros.
- Si el tipo del arreglo es booleano se llena de valores false.
- Si el tipo es el nombre de una clase o interfaz se llena de referencias null.

Acomodo de arreglos en memoria durante la ejecución

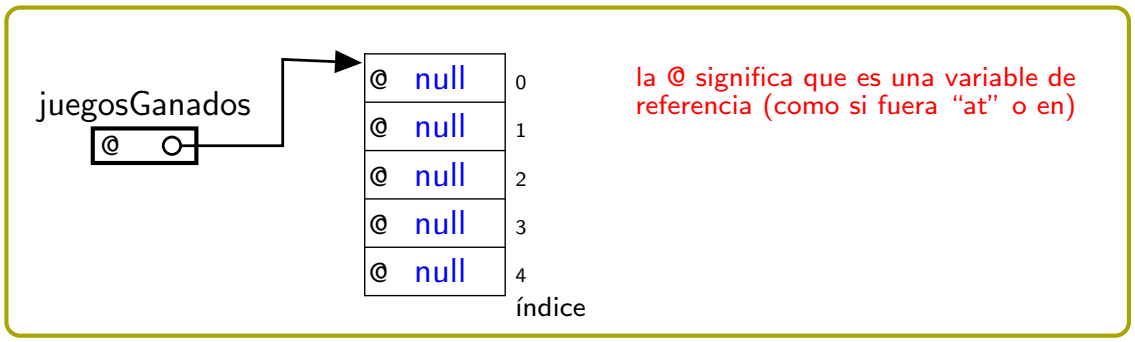
Si bien las vistas abstractas que viste de cómo pensar en arreglos responden precisamente a eso, cómo *piensas* en los arreglos, es importante conocer cuál es el acomodo en memoria de los arreglos. Este conocimiento ayuda a una mejor comprensión de errores que pudiesen surgir en la ejecución y para conseguir una mayor eficiencia en el manejo de los mismos.

Los arreglos, como a cualquier objeto, Java los construye en el *heap*, que es el espacio de la memoria de la computadora reservada para ellos. Como ya escuchaste, Java maneja a los arreglos de dos dimensiones como un arreglo donde cada elemento del arreglo es, a su vez, un arreglo y, finalmente cada elemento de este segundo arreglo es del tipo especificado para el arreglo. Los arreglos de tres dimensiones son para Java un arreglo, donde cada elemento de ese arreglo es otro arreglo, donde cada elemento de los arreglos en el segundo nivel son, a su vez, arreglos; finalmente cada elemento de los arreglos en el tercer nivel es del tipo especificado para el arreglo.

Por este manejo de Java para los arreglos, como un arreglo de dos dimensiones ([ ][ ]) es un arreglo de arreglos, puedes primero definir cuantos "elementos" va a tener el primer arreglo. La siguiente declaración y definición de un arreglo de dos dimensiones

```
int[][] juegosGanados = new int[5][]; // 5 renglones, cada uno sin tamanho definido
```

hace que, en ejecución, Java construya una estructura como la que sigue:



En la siguiente lectura vienen más ejemplos de la utilización de arreglos.