El tercer constructor

Como ya vimos, el tercer constructor recibe como parámetros el tamaño que debe tener el catálogo y un arreglo con los primeros discos a ingresar. Vimos ya la construcción de los arreglos como en los otros dos constructores. Hasta ahora, el código que llevamos (sin la documentación de Javadoc) es el siguiente:

```
122   public Catalogo ( int numDscs, Disco[ ] nuevos ) {
123     int numNvos = nuevos == null ? 0 : nuevos.length;
124     numDscs    = numDscs < numNvos ? numNvos : numDscs;
125     numDscs    = Disco.checaEntero(1, numDscs, MAX_DISCOS);
126     /* El numero de discos a copiar es el minimo entre el tamanho final
127      * del arreglo y el numero de discos nuevos */
128     numNvos = numNvos > numDscs ? numDscs : numNvos;
129     catalogo   = new Disco[numDscs];
130     numDiscos  = 0;
131     fechas     = new GregorianCalendar[catalogo.length][];
132     historico  = new GregorianCalendar[catalogo.length][2][];
133     numHist    = new int[catalogo.length];
```

Empezamos ahora a partir de la copia del arreglo `nuevos` al arreglo `catalogo` y la construcción de columnas para los discos conforme se van copiando.

Una vez construidos los arreglos básicos, procedemos a copiar cada disco de `nuevos` al catálogo usando un `for`. Debemos recorrer el arreglo `nuevos` y copiar cada disco al catálogo.

```
134     /* Copias ahora los discos nuevos al catalogo en los primeros
135      * lugares del catalogo */
136     for (int i = 0; i < numNvos; i++) {
```

Sabemos que en el catálogo hay, al menos, los lugares necesarios para copiar el contenido de `nuevos`. Sin embargo, uno o más de los espacios en `nuevos` podrían ser una referencia nula (`null`), por lo que cuidamos no agregar estas referencias al catalogo.

Conforme copiamos un disco incrementamos el número de discos en el catálogo (`numDiscos`) y, de acuerdo al número de transmisiones permitidas en ese disco procedemos a construir las columnas en `fechas` y en `historico`.

```
137       catalogo[i]    = nuevos[i];  // se copia el disco
138       int numPrest   = catalogo[i].getPermitidas();
139         /* numero posible de fechas para el disco i */
140       fechas[i]      = new GregorianCalendar[numPrest];
141         /* numero posible de historicos para el disco i */
142       historico[i][0] = new GregorianCalendar[2 * numPrest];
143       historico[i][1] = new GregorianCalendar[2 * numPrest];
144       numDiscos++;
145       numHist        = new int[catalogo.length]; // son ceros
146     }
```

Si consideramos los enunciados que van de la línea con el número 138 a la línea 143 estamos usando el disco en la posición i para construir las columnas en cada uno de estos arreglos. Si por alguna razón tuviésemos una referencia nula en esa posición la ejecución abortaría. Debemos asegurar que esto no suceda, preguntando si la referencia es nula y si lo es, regresando al encabezado del `for`.

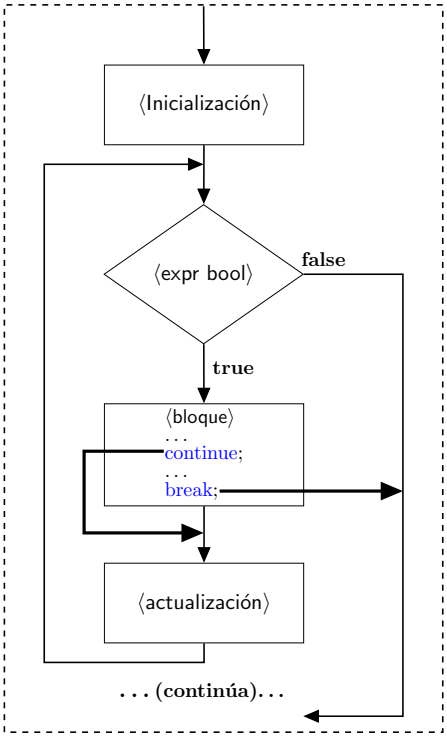
Mencionamos, cuando vimos el flujo de ejecución del `for` que teníamos dos formas de eludir lo que resta en el ⟨bloque⟩ del `for`:

continue : Este enunciado hace que el flujo de control continúe en la parte que corresponde a la actualización, ejecutándola y regresando a la evaluación de la expresión booleana.

break : Este enunciado hace que la ejecución continúe en el enunciado que sigue al `for`. Saca a la ejecución de la iteración más cercana en la que esté.

Modificamos el diagrama que hicimos para las iteraciones para indicar qué sucede cuando se usa el enunciado `continue` o `break`.

Figura 1 Flujo en la ejecución de una iteración en Java



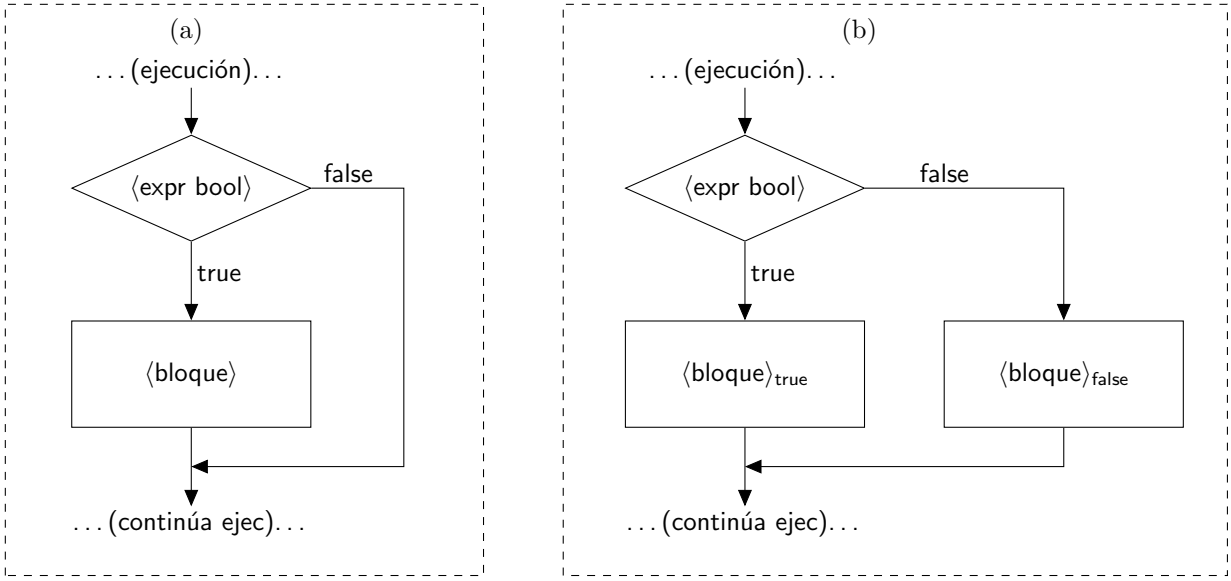
Debemos usar el primer enunciado, pues simplemente deseamos brincar ese elemento. Sin embargo, si lo escribimos incondicionalmente nunca va a seguir adelante. Tenemos que preguntar si la referencia en `nuevos[i]` es nula y en este caso utilizar el enunciado `continue`. No podemos usar, como lo hemos hecho hasta ahora, la condicional aritmética pues esta sólo elige uno de dos valores pero no es eso lo que queremos hacer.

Enunciado condicional

Conocemos la condicional aritmética que evalúa una expresión booleana y dependiendo de si es verdadera o falsa procede a entregar un valor. Como acabamos de explicar, cuando necesitamos que se ejecuten uno o más enunciados, dependiendo de si una expresión booleana es verdadera o falsa, la condicional aritmética no es suficiente. Para ello necesitamos un **enunciado condicional**.

El flujo de ejecución en un enunciado condicional es como se ve en la 2.

Figura 2 Flujo de la ejecución en un enunciado condicional



Tenemos, como podemos ver en la figura 2 dos tipos de enunciados condicionales, uno sólo con el bloque que corresponde a que la expresión booleana sea verdadera y el segundo da bloques a ejecutar tanto cuando la expresión booleana es verdadera como para cuando la expresión booleana es falsa. En el primer caso, si la expresión booleana es verdadera se ejecuta el bloque, que puede ser un solo enunciado o un bloque entre llaves. Una vez terminado de ejecutar el bloque, y al menos que haya una salida abrupta con un `return`, `break` o `continue`, sigue con la ejecución del enunciado que sigue a la condicional.

En el segundo caso si la expresión booleana es verdadera ejecuta el bloque correspondiente, como en el primer caso. Si la expresión booleana es falsa, ejecuta el bloque correspondiente en el diagrama. En ambos casos, a menos que su bloque contenga alguna de las salidas abruptas mencionadas, la ejecución continúa en el enunciado que sigue al enunciado condicional.

En cualquiera de los dos enunciados la ejecución de los bloques depende de la evaluación de la expresión booleana. En el primer caso puede o no ejecutarse el bloque correspondiente. En el segundo caso se ejecuta uno y sólo uno de los bloques, dependiendo nuevamente de si la expresión booleana se evalúa a falso o verdadero.

Sintaxis en Java para el enunciado condicional

La sintaxis del enunciado condicional para ambos enunciados se encuentra en la siguiente página:

```
if (⟨expr bool⟩  // condicional simple
   ⟨bloque⟩true

if (⟨expr bool⟩  // condicional con else
   ⟨bloque⟩true
else
   ⟨bloque⟩false
```

Como un ⟨bloque⟩ puede ser un enunciado o una lista de enunciados entre llaves, podemos tener enunciados condicionales anidados:

```
if ( a > b )
   if ( objeto == null )
      ...
   else
      ...
```

En este caso tenemos dos `if` y un único `else`, por lo que las reglas de asociación indican que se asocian de adentro hacia afuera, como los paréntesis en una expresión aritmética. Si deseamos que el único `else` se asocie al primer `if` tenemos que encerrar entre llaves al segundo `if`, de tal forma que quede claro que el bloque correspondiente a `true` solo tiene al `if`; de esta manera obligamos a que el `else` se asocie al primer `if`.

```
if ( a > b ) {
   if ( objeto == null )
      ...
}
else
   ...
```

De regreso en el constructor

En el listado de la página 1, antes de ejecutar la línea con el número 137 debemos asegurarnos que lo que estamos copiando no es una referencia nula. Podemos hacerlo de dos maneras:

- Preguntar si la referencia no es nula, y si no lo es, ejecutar hasta el final del método, encerrando todos los enunciados entre llaves:

```
137   for (int i = 0; i < numNvos; i++) {
138     if (nuevos[i] != null) {// empieza el bloque
139       catalogo[numDiscos]   = nuevos[i];
140       int numPrest          = catalogo[numDiscos].getPermitidas();
141         /* numero posible de fechas para el disco i */
142       fechas[numDiscos]     = new GregorianCalendar[numPrest];
143         /* numero posible de historicos para el disco i */
144       historico[numDiscos][0] = new GregorianCalendar[2 * numPrest];
145       historico[numDiscos][1] = new GregorianCalendar[2 * numPrest];
146       numDiscos++;
147       numHist               = new int[catalogo.length]; // son ceros
148     } // fin del bloque del if
149   }
```

  En general es mala idea tener un enunciado condicional en el que una de las ramas es mucho mayor que la otra.

- Preguntar si la referencia es nula y en este caso sacar la ejecución del bloque usando el enunciado `continue` que consigue que la ejecución siga dentro de la iteración. Proponemos la siguiente organización:

```
136     for (int i = 0; i < numNvos; i++) {
137       if (nuevos[i] == null)
138         continue;           // Va a la actualizacion del for
139       catalogo[numDiscos]   = nuevos[i];
140       int numPrest          = catalogo[numDiscos].getPermitidas();
141         /* numero posible de fechas para el disco i */
142       fechas[numDiscos]     = new GregorianCalendar[numPrest];
143         /* numero posible de historicos para el disco i */
144       historico[numDiscos][0] = new GregorianCalendar[2 * numPrest];
145       historico[numDiscos][1] = new GregorianCalendar[2 * numPrest];
146       numDiscos++;
147       numHist               = new int[catalogo.length]; // son ceros
148     }
```

No deseamos salir de la iteración porque es posible que una referencia nula sea seguida por una referencia no nula, y dejaríamos de copiar el resto de los discos. Con esto terminamos el tercer constructor. Su código completo queda de la siguiente manera:

```
122   public Catalogo ( int numDscs, Disco[ ] nuevos ) {
123     int numNvos = nuevos == null ? 0 : nuevos.length;
124     numDscs    = numDscs < numNvos ? numNvos : numDscs;
125     numDscs    = Disco.checaEntero(1, numDscs, MAX_DISCOS);
126     /* El numero de discos a copiar es el minimo entre el tamanho final
127      * del arreglo y el numero de discos nuevos */
128     numNvos = numNvos > numDscs ? numDscs : numNvos;
129     catalogo   = new Disco[numDscs];
130     numDiscos  = 0;
131     fechas     = new GregorianCalendar[catalogo.length][];
132     historico  = new GregorianCalendar[catalogo.length][2][];
133     numHist    = new int[catalogo.length];
134     /* Copias ahora los discos nuevos al catalogo en los primeros
135      * lugares del catalogo */
136     for (int i = 0; i < numNvos; i++) {
137       if (nuevos[i] == null)
138         continue;           // un unico enunciado para true
139       catalogo[numDiscos]   = nuevos[i];
140       int numPrest          = catalogo[numDiscos].getPermitidas();
141         /* numero posible de fechas para el disco i */
142       fechas[numDiscos]     = new GregorianCalendar[numPrest];
143         /* numero posible de historicos para el disco i */
144       historico[numDiscos][0] = new GregorianCalendar[2 * numPrest];
145       historico[numDiscos][1] = new GregorianCalendar[2 * numPrest];
146       numDiscos++;
147     } // for i = 0
148   } // Termina constructor con dos parametros
```

Con esto terminas la codificación de los constructores, por lo que pasas ahora a la codificación de los métodos. Nos vemos en el siguiente video.