Sobre el Garbage Collector y el uso de memoria

(21/11/2023)

Después de hacer algunas pruebas “confusas” en clase, vamos a ver si podemos aclarar un poco el asunto.

Como me indicasteis en clase, el problema era mi interpretación del código en lo referente a la reserva del array con que cada objeto ocuparía una porción de memoria. Usé intencionadamente un valor hexadecimal para indicar fácilmente una potencia de 2 y posteriormente me empeciné en tomarlo por decimal. Tras diversas pruebas, la cantidad que dejé finalmente en el código era 0x100_000, es decir 2^20 = 1Mbyte, y mi error fue hacer las pruebas pensando en 100_000 bytes, es decir 100 Kbytes (y aquí he de añadir que serían en todo caso “kilobytes comerciales”, es decir 100x1000 bytes, no kilobytes “de verdad” –llamados en ocasiones kibibytes–, que se corresponderían con 100x1024 bytes)

Para esta revisión del asunto he modificado ligeramente el código, y en lo que se refiere a esta reserva de memoria, he usado la expresión 1<<20 que no se presta a confusión. Aunque se trate de una expresión a evaluar en vez de ser un literal, el compilador entenderá que es un valor que puede calcular y codificar directamente como una constante.

Otro cambio que puede ser ligeramente clarificador es el añadido de una clase interna para la generación de los objetos que iremos abandonando posteriormente.

```
package edu.upv.ehu.gbg.tap.GarbageCollector;

public class GarbageCollectorTest {

    //Generaremos objetos de una clase que define un campo de 1 MByte (2^20)
    private static class GastadorDeMemoria {byte[] s = new byte[1<<20];}

    //Generaremos indefinidamente objetos de los que se conservarán úncamente un número limitado.
    public static void main(String[] args) throws InterruptedException {
        GastadorDeMemoria[] accesibles = new GastadorDeMemoria[Integer.parseInt(args[0])];
        while(true)
            for (int i = 0; i < accesibles.length; i++) {
                accesibles [i] = new GastadorDeMemoria();
                Thread.sleep(100);
            }
    }
}
```

Para realizar varias pruebas, la dimensión del array de objetos a mantener se definirá en función de un parámetro de entrada (recibido a través del parámetro “args” del **`main`** como un entero en texto que se convierte a **`int`** mediante **`Integer.parseInt(.)`** al dimensionar el array “accesibles”), y dispondremos en el proyecto de varias configuraciones de ejecución (de perfilado en este caso) como puede verse en la Figura 1

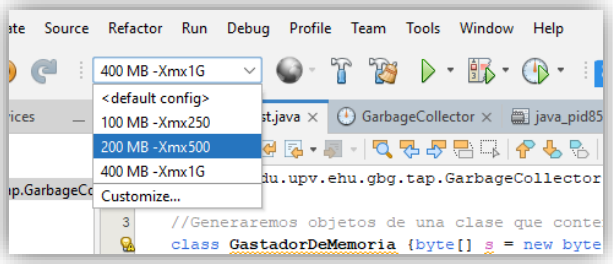


Figura 1.-Configuraciones para la ejecución/debug/perfilado

La versión de JDK usada es la 21, con el Garbage Collector por defecto (G1GC)

Figura 2.-Ejecución con array para 200 objetos y límite de memoria en 500 MBytes

En la Figura 2 se aprecia el funcionamiento del GC con una ejecución que mantendrá 200 objetos accesibles (200 MB) y un límite de uso de memoria de 500 MB. Se pueden apreciar varias fases en su comportamiento, tanto conforme la memoria va siendo ocupada y no hay objetos perdidos, como cuando se pasa a perder acceso a objetos y por tanto a necesitar que el GC actúe recuperando memoria como espacio libre.

Inicialmente la reserva por defecto es algo menor a 200 MB y el espacio ocupado va creciendo hasta acercarse a ese valor, momento en que la reserva comienza a crecer. El GC entra frecuentemente para seguir reservando más memoria conforme ve que se generan más objetos, decidiendo no perder el tiempo recogiendo los que ya son no accesibles. Sólo cuando se acerca a unos 100 MB del límite establecido comienza a "limpiar". En esta fase deja crecer de nuevo la ocupación hasta casi consumir la memoria, momento en que realiza una limpieza que restaura el margen de 100 MB y continúa manteniendo ese nivel con "limpiezas" constantes. Esto lo hace un par de veces, tras lo cual cambia de nuevo de estrategia y deja crecer la ocupación hasta cerca del límite, para restaurar el margen de 100 MB por un instante y repetir.

Todo esto se aprecia bien en la gráfica de ocupación de memoria. En la de instantes de ejecución del GC vemos una primera zona de alta frecuencia que se corresponde al tiempo en que va reservando más memoria hasta llegar al límite; una segunda mientras mantiene el nivel de 400 MB durante un tiempo; y otra tercera semejante, para terminar estabilizado con una estrategia donde ha calibrado el tiempo que puede estar dejando crecer la ocupación para entrar mucho más infrecuentemente a limpiar hasta los 400 MB.

La imagen es un poco engañosa al parecer que en las zonas de alta frecuencia el GC consume prácticamente todo el tiempo. Esto no es más que un efecto gráfico: en la Figura 3 se ha ampliado la escala sobre la primera zona y pueden verse los instantes en que entra el GC y la relación de tiempos en que está activo e inactivo. En todo caso en la primera gráfica se aprecia que tanto nuestro hilo como el del GC no suponen ninguna carga para la CPU (hemos programado un "descanso" de 0.1 segundos).

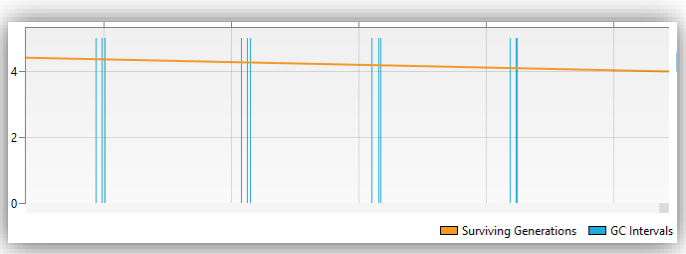


Figura 3.-Detalle de la zona en que el GC se ejecuta frecuentemente

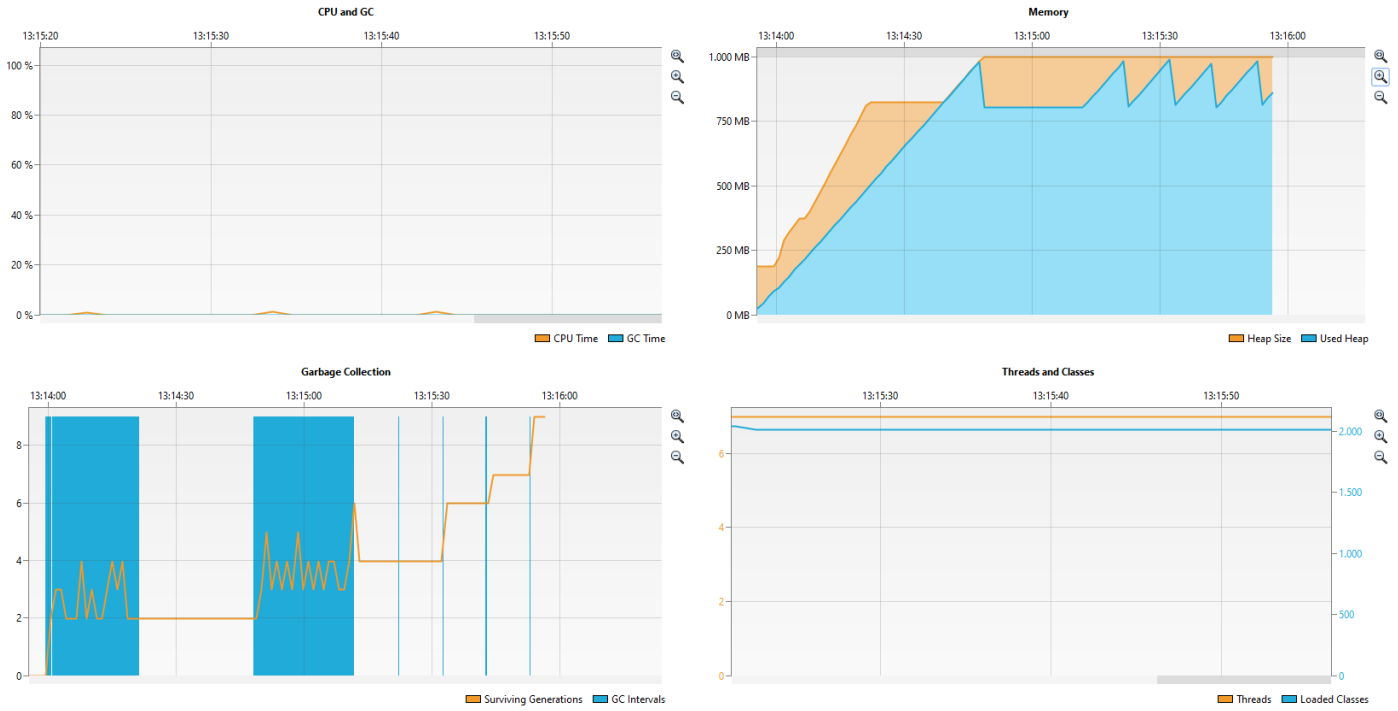


Figura 4.- Ejecución con array para 400 objetos y límite de memoria en 1 GByte

En la Figura 4 se aprecia el funcionamiento del GC con una ejecución que mantendrá 400 objetos accesibles (400 MB) y un límite de uso de memoria de 1 GB. En este caso hay una mayor holgura que hace que el comportamiento resulte ligeramente distinto: vemos que permite ocupar memoria directamente hasta casi el límite, y después ejecuta un asola fase de mantenimiento de un margen amplio (unos 200 MB) para finalmente entrar en la fase estable en la que ha determinado un tiempo de espera ajustado para entrar a hacer limpiezas de unos 200 MB. Con esto tenemos sólo dos zonas en que la frecuencia de entrada es elevada. Es curioso observar cómo una vez sobrepasados los 800MB por primera vez, la ampliación de memoria va en paralelo con su uso, trabajando con un margen que parece nulo.

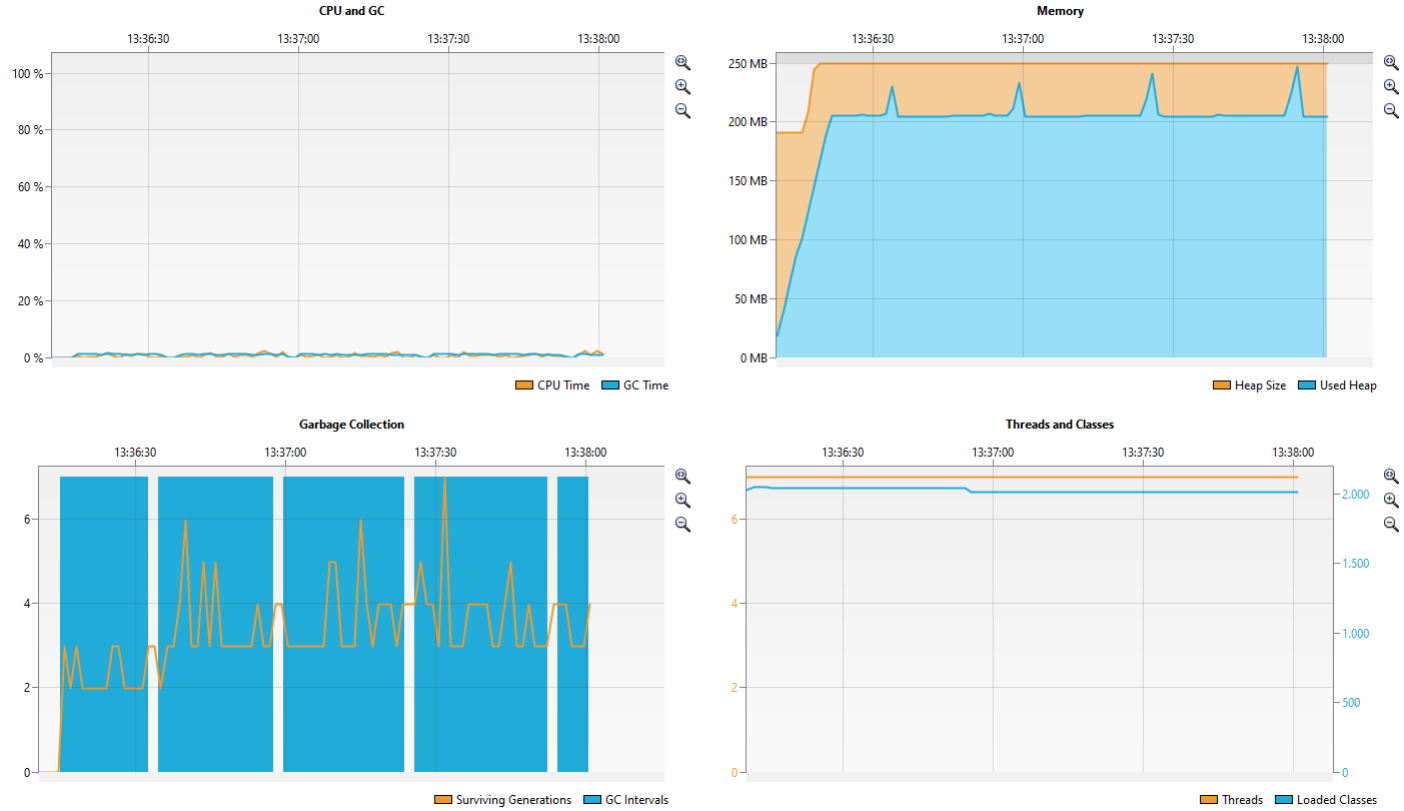
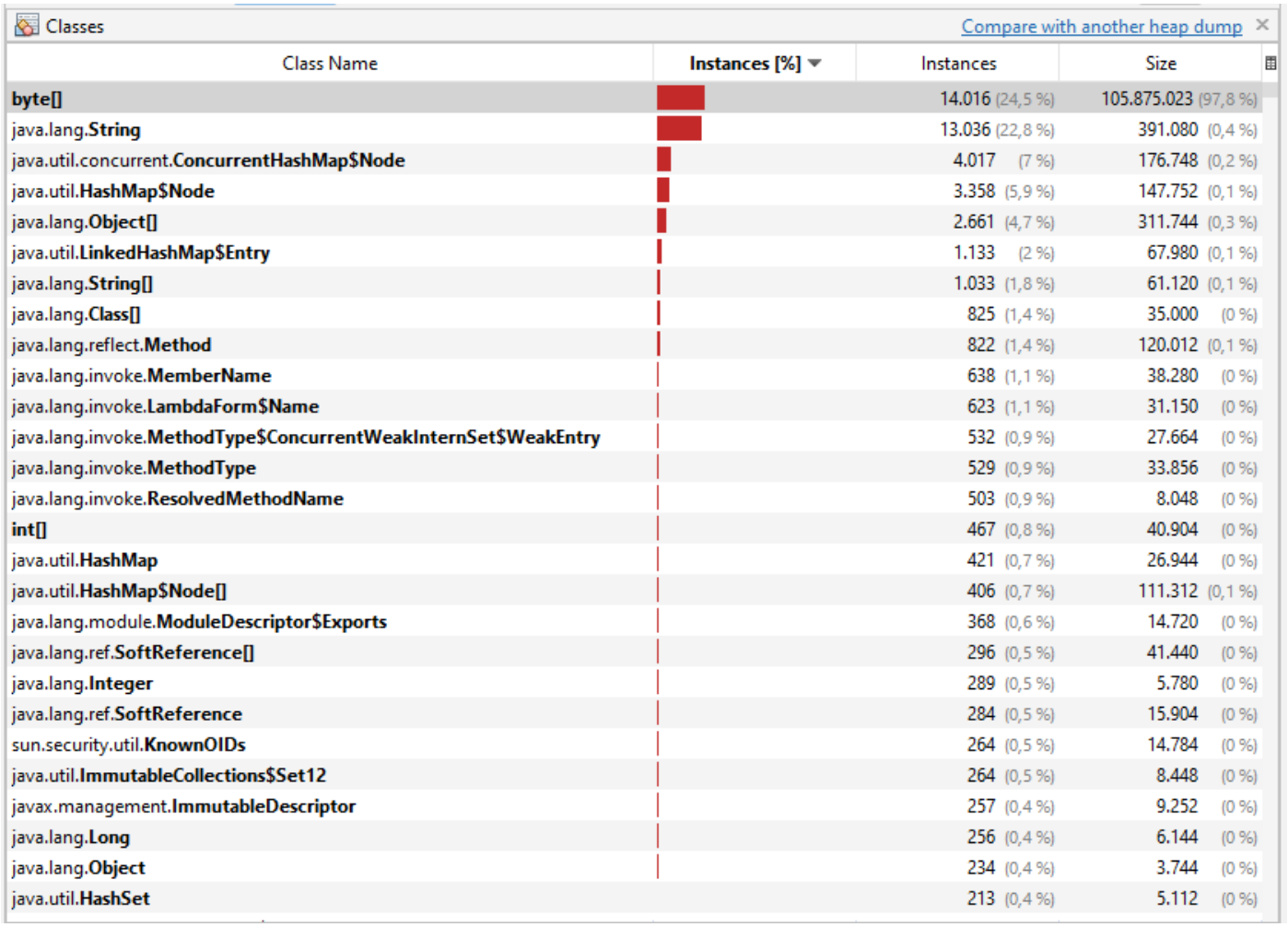


Figura 5.- Ejecución con array para 100 objetos y límite de memoria en 250 MBytes

En la Figura 5 observamos el comportamiento cambiando las cantidades en dirección contraria: la ejecución mantendrá sólo 100 objetos accesibles (100 MB) y el límite de uso de memoria será de 250 MB. Mantenemos de este modo el margen de los casos anteriores (un 150% de memoria libre frente a la estrictamente necesaria), y vemos cómo el margen que el GC se da también se mantiene en la misma proporción, pero, al ser menor en números absolutos, parece ser que no llega a establecer una estimación del tiempo de espera adecuado (dejándolo evolucionar mucho más tiempo no cambia la estrategia)


Classes — Compare with another heap dump

Class Name	Instances [%]	Instances	Size
byte[]		14.016 (24,5 %)	105.875.023 (97,8 %)
java.lang.**String**		13.036 (22,8 %)	391.080 (0,4 %)
java.util.concurrent.**ConcurrentHashMap$Node**		4.017 (7 %)	176.748 (0,2 %)
java.util.**HashMap$Node**		3.358 (5,9 %)	147.752 (0,1 %)
java.lang.**Object[]**		2.661 (4,7 %)	311.744 (0,3 %)
java.util.**LinkedHashMap$Entry**		1.133 (2 %)	67.980 (0,1 %)
java.lang.**String[]**		1.033 (1,8 %)	61.120 (0,1 %)
java.lang.**Class[]**		825 (1,4 %)	35.000 (0 %)
java.lang.reflect.**Method**		822 (1,4 %)	120.012 (0,1 %)
java.lang.invoke.**MemberName**		638 (1,1 %)	38.280 (0 %)
java.lang.invoke.**LambdaForm$Name**		623 (1,1 %)	31.150 (0 %)
java.lang.invoke.**MethodType$ConcurrentWeakInternSet$WeakEntry**		532 (0,9 %)	27.664 (0 %)
java.lang.invoke.**MethodType**		529 (0,9 %)	33.856 (0 %)
java.lang.invoke.**ResolvedMethodName**		503 (0,9 %)	8.048 (0 %)
int[]		467 (0,8 %)	40.904 (0 %)
java.util.**HashMap**		421 (0,7 %)	26.944 (0 %)
java.util.**HashMap$Node[]**		406 (0,7 %)	111.312 (0,1 %)
java.lang.module.**ModuleDescriptor$Exports**		368 (0,6 %)	14.720 (0 %)
java.lang.ref.**SoftReference[]**		296 (0,5 %)	41.440 (0 %)
java.lang.**Integer**		289 (0,5 %)	5.780 (0 %)
java.lang.ref.**SoftReference**		284 (0,5 %)	15.904 (0 %)
sun.security.util.**KnownOIDs**		264 (0,5 %)	14.784 (0 %)
java.util.**ImmutableCollections$Set12**		264 (0,5 %)	8.448 (0 %)
javax.management.**ImmutableDescriptor**		257 (0,4 %)	9.252 (0 %)
java.lang.**Long**		256 (0,4 %)	6.144 (0 %)
java.lang.**Object**		234 (0,4 %)	3.744 (0 %)
java.util.**HashSet**		213 (0,4 %)	5.112 (0 %)

Figura 6.- Vista parcial de las clases cargadas. Se muestran las que más instancias han generado con el número de las mismas y la memoria ocupada. En primer lugar aparece byte[], algunas instancias de las cuales se deben a nuestro programa. Nuestras clases GarbageCollectorTest y GarbageCollectorTest.GastadorDeMemoria son más infrecuentes y no llegan a verse.

En todo momento estamos dando por hecho que la cantidad de memoria necesaria viene dada prácticamente por el número de objetos accesibles desde el array. Obviamente es razonable pensar que hará falta algo más para el resto de elementos, pero posiblemente no demasiado. Podemos pedir al perfilador que nos dé un sumario de las clases que ha cargado (ya vemos en las gráficas anteriores, en el cuadrante de abajo a la derecha, que son del orden de 2000) y así como objetos que ha instanciado, con su ocupación de memoria. En la Figura 6 vemos cómo el array de bytes y las Strings son las clases de las que más objetos se han instanciado, y les siguen otras clases que ya son ajenas a nuestro código. No llegan a verse las clases que hemos definido nosotros mismos, pero podemos filtrar el listado como se hace en la Figura 7 para conseguir que se muestren. Vemos cómo hay 100 instancias de **`GarbageCollectorTest.GastadorDeMemoria`** en el momento en que hemos monitorizado la memoria, así como un array para referenciar a 100 de ellas, y de la clase **`GarbageCollectorTest`** no hay ninguna instancia ya que sólo hemos ejecutado el main(.)

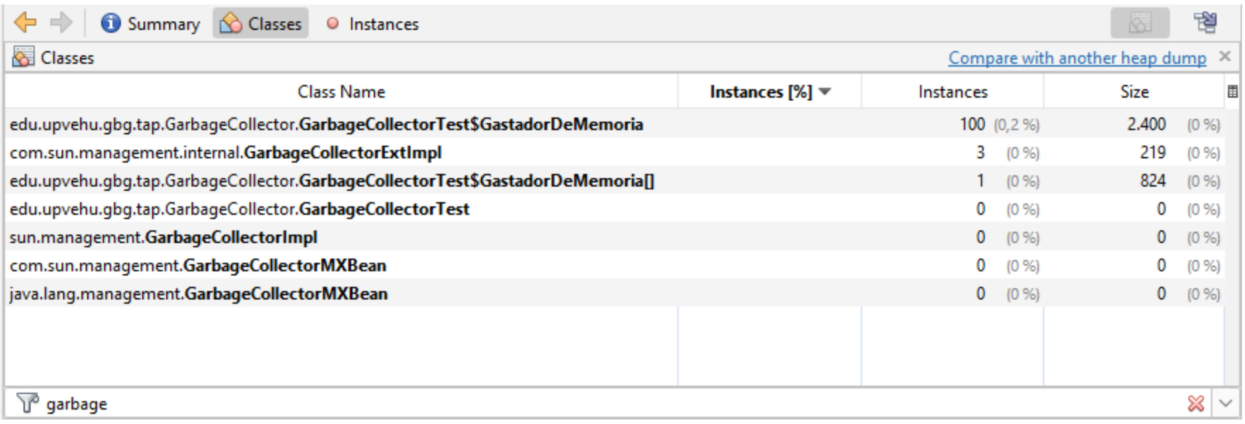


Figura 7.- Filtrando la lista de clases con el término "garbage" podemos ver nuestras clases GarbageCollectorTest, GarbageCollectorTest.GastadorDeMemoria, así como el array de objetos de ésta última.

Si volvemos a fijarnos en los arrays de bytes, vemos cómo ocupan del orden de 100 MB, es decir, son los correspondientes a los 100 objetos que mantenemos, y esto supone un 98% de la memoria usada… un 2% es suficiente para el resto de objetos y clases que componen la aplicación, incluido lo necesario para realizar el perfilado que llevamos a cabo.

Podemos mirar esto con más detalle aún, cosa que hacemos en la Figura 8. A la izquierda vemos un listado de las instancias **`byte[]`** en bloques de 500 objetos con el primer bloque desplegado. El primer array se corresponde con un objeto del perfilador (un campo estático de la clase **`org.netbeans.lib.profiler.server.ProfilerRuntime`**), pero desde el segundo objeto en adelante se encuentran objetos con 1.048.600 posiciones que, sin duda, son nuestros arrays (1 MB = 1.048.576 bytes, a lo que hay que añadir algún campo más del objeto array como p.ej. el campo length). Tanto a la derecha como abajo tenemos representaciones del contenido del array seleccionado (el #2, que es el primero de los nuestros) mostrando todos los valores a cero. En la parte inferior derecha se muestra cómo el objeto seleccionado (this) está referenciado como campo "s" de un objeto de clase **`GarbageCollectorTest.GastadorDeMemoria`** y éste a su vez está referenciado en el array **`GarbageCollectorTest.GastadorDeMemoria[]`** (que es un objeto local dentro del **`main`**) en la posición 3.

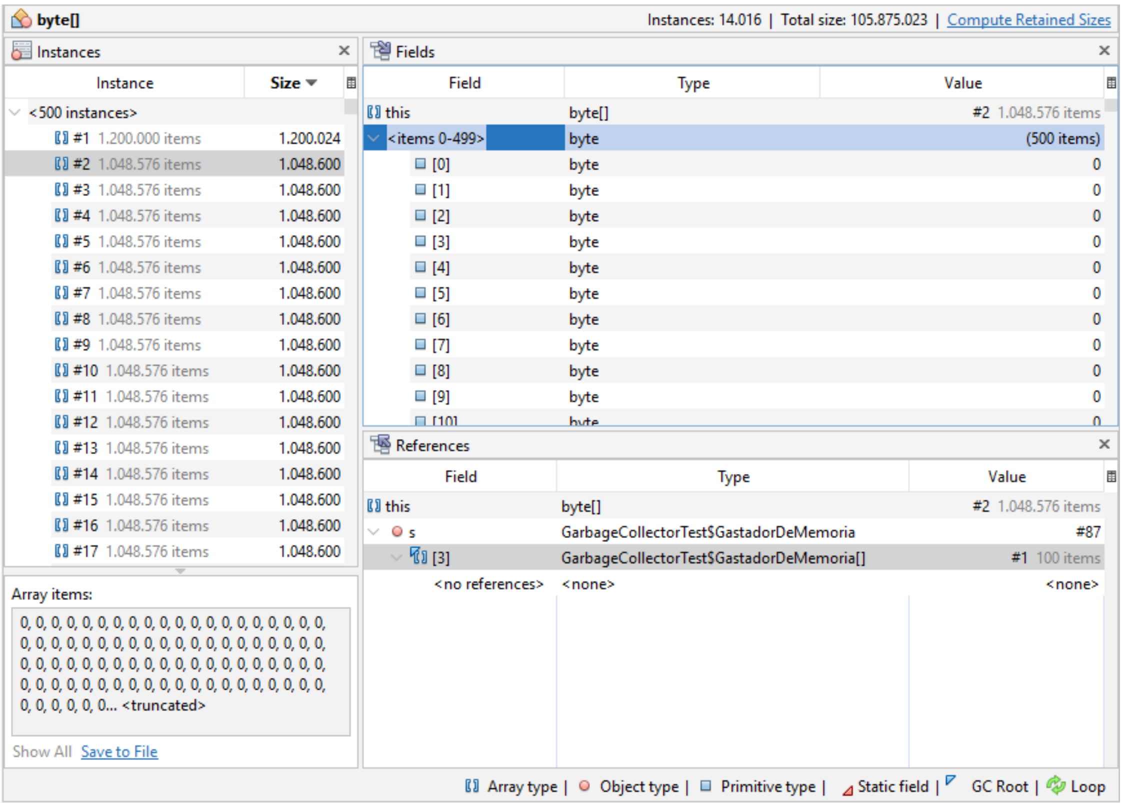


Figura 8.- El objeto #2 es uno de los arrays de bytes generados por nuestro programa, y podemos ver que tiene sus elementos a cero y que está referenciado como campo "s" de un objeto GarbageCollectorTest.GastadorDeMemoria que está a su vez referenciado por el array para este tipo de objetos en la posición con índice 3.