# ¿Cómo ocurren pérdidas de memoria en una aplicación Java

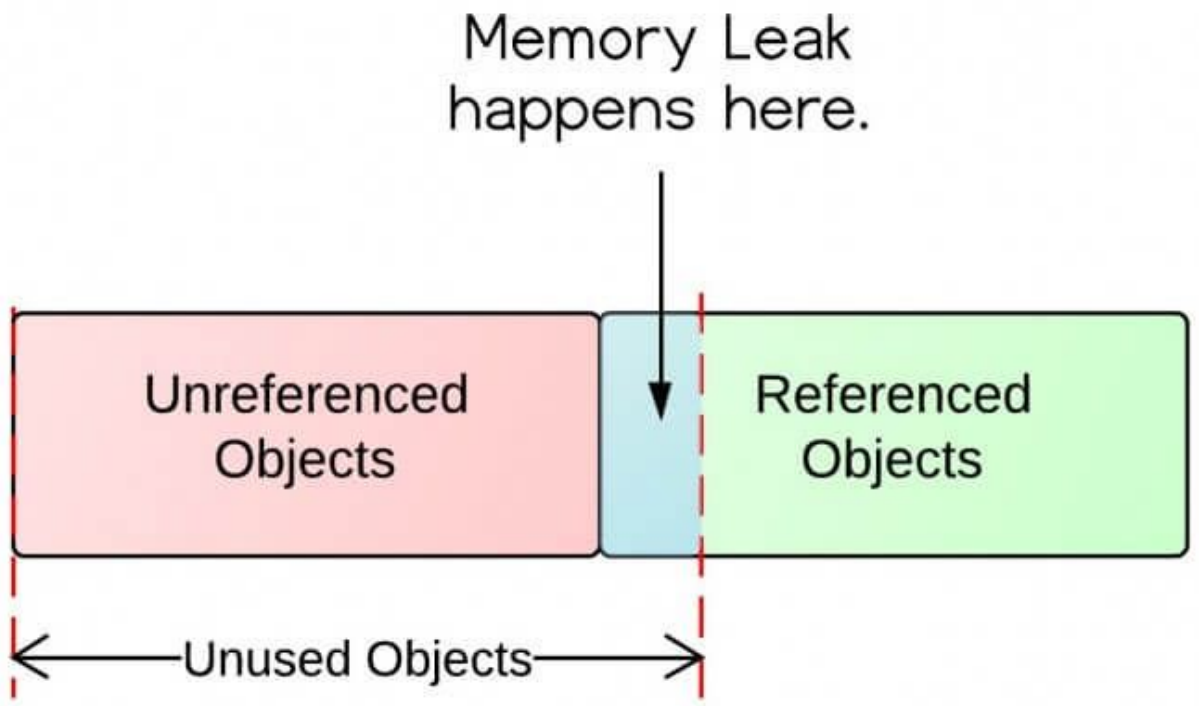
## Introducción a pérdidas de memoria en las aplicaciones Java

Una de las ventajas principales de Java es la JVM, que es un *out of the box* en la gestión de memoria .En esencia, podemos crear objetos y la Java recolector de basura se encargará de asignar y liberar memoria para nosotros.

**Sin embargo, las pérdidas de memoria pueden ocurrir en aplicaciones Java.**
En este artículo, vamos a describir las pérdidas de memoria más comunes, entender sus causas, y mirar algunas técnicas para detectar / evitarlos. También vamos a utilizar el perfilador de Java YourKit lo largo del artículo, para analizar el estado de la memoria en tiempo de ejecución.

### 1. ¿Qué es una pérdida de memoria en Java?

La definición estándar de una pérdida de memoria es un escenario que se produce cuando **los objetos ya no están siendo utilizados por la aplicación, pero el recolector de basura no es capaz de eliminarlos de la memoria de trabajo** - porque todavía están siendo referenciados. Como resultado, la aplicación consume cada vez más recursos - que finalmente conduce a una *OutOfMemoryError* fatal.
Para una mejor comprensión del concepto, esto es una representación visual simple:




Como podemos ver, tenemos dos tipos de objetos - que se hace referencia y no referenciados; el recolector de basura puede eliminar objetos que se encuentran sin referencias. No se recogen los objetos referenciados, aunque en realidad son no más utilizados por la aplicación.
La detección de fugas de memoria puede ser difícil. Una serie de herramientas de realizar el análisis estático para determinar posibles fugas, pero estas técnicas no son perfectos ya que el aspecto más importante es el comportamiento en tiempo de ejecución real del sistema en funcionamiento.
Por lo tanto, vamos a echar un vistazo centrado en algunas de las prácticas estándar de la prevención de pérdidas de memoria, **mediante el análisis de algunos escenarios** comunes.

## 2. Las fugas heap de Java

En esta sección inicial, nos vamos a centrar en el clásico escenario de pérdida de memoria - donde los objetos Java se crean continuamente sin ser liberado.
Una técnica ventajosa para entender estas situaciones es hacer que la reproducción de una pérdida de memoria más fácil mediante el de **establecimiento un tamaño menor para el** heap.Por eso, al iniciar nuestra aplicación, podemos ajustar la JVM para satisfacer las necesidades de nuestra memoria:

```
-Xms <tamaño>
-Xmx <tamaño>
```

Estos parámetros especifican el tamaño inicial heap de Java, así como el tamaño máximo del heap.

### 2.1. Campo estático aferrarse al objeto de referencia

El primer escenario que podría provocar una pérdida de memoria de Java hace referencia a un objeto pesado con un campo estático.

Vamos a echar un vistazo a un ejemplo rápido:

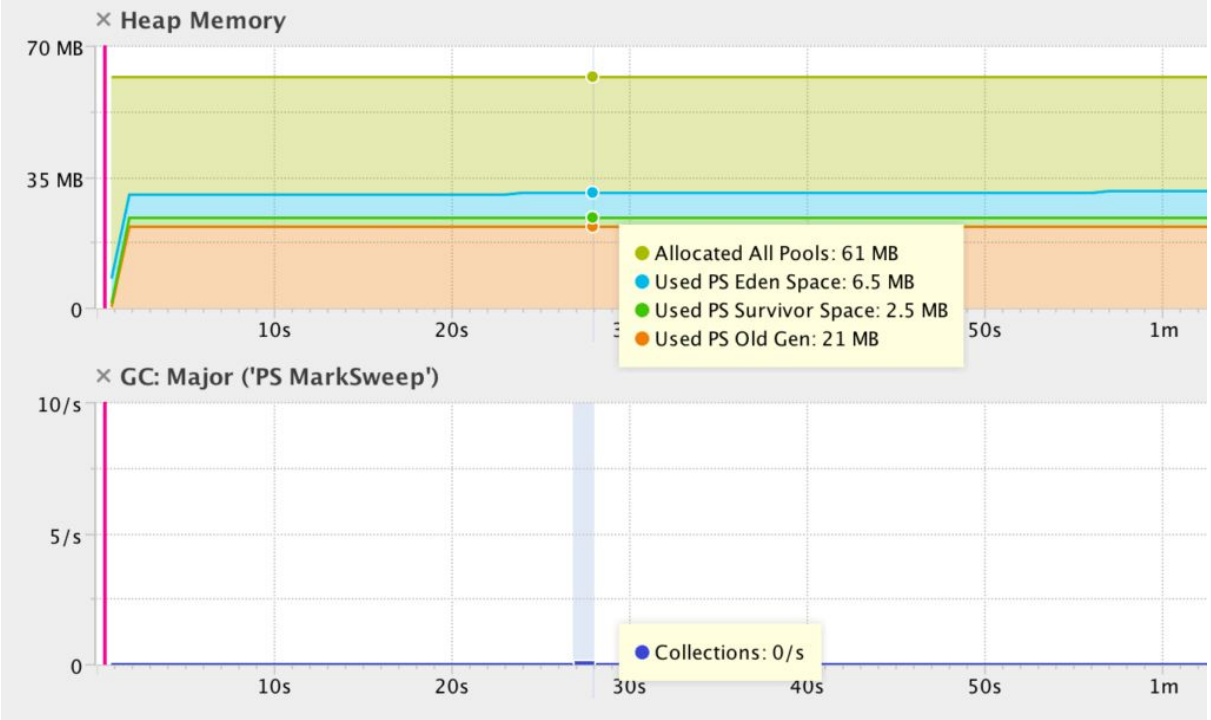
```
privada aleatoria al azar = new Random ();
Lista pública static final ArrayList <Double> = new ArrayList
<Double> (1000000);

@test
givenStaticField_whenLotsOfOperations_thenMemoryLeak   public
void () lanza InterruptedException
    {for(int i = 0; i <1.000.000; i ++)
        {list.add(random.nextDouble ());
    }

    System.gc ();
    Thread.sleep (10000); // para que haga su trabajo
GC}
```

Hemos creado nuestra *ArrayList* como un campo estático - que nunca serán recogidos por el recolector de basura JVM durante el tiempo de vida del proceso de JVM, incluso después de los cálculos que se utilizó para la llevan a cabo. También hemos invocado *Thread.sleep (10000)* para permitir que el GC para realizar una colección completa y tratar de recuperar todo lo que se puede recuperar.

Vamos a ejecutar la prueba y analizar la JVM con nuestro generador de perfiles:




Observe cómo, desde el principio, toda la memoria es, por supuesto, gratis.

Luego, en sólo 2 segundos, la ejecuta el proceso de iteración y acabados - todo lo carga en la lista (por supuesto esto dependerá de la máquina que está ejecutando la prueba de encendido).

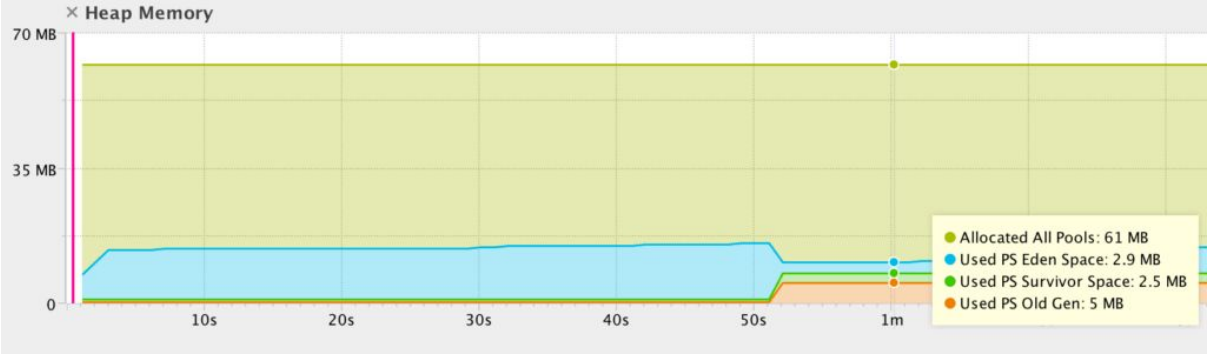
Después de eso, un ciclo completo de Garbage collector se activa, y la prueba continúa ejecutando, para permitir que este tiempo de ciclo para ejecutar y terminar. Como se puede ver, la lista no se recupera y el consumo de memoria no baja.

Veamos ahora el mismo ejemplo exacta, sólo que esta vez, la *ArrayList* no está referenciado por una variable estática. En cambio, es una variable local que se crea, usa y luego se desecha:

```
@Test
public                                                    void
givenNormalField_whenLotsOfOperations_thenGCWorksFine()
throws InterruptedException {
    addElementsToTheList();
    System.gc();
    Thread.sleep(10000); // to allow GC do its job
}

private void addElementsToTheList(){
    ArrayList<Double> list = new ArrayList<Double>(1000000);
    for (int i = 0; i < 1000000; i++) {
        list.add(random.nextDouble());
    }
}
```

Una vez que el método termina su trabajo, observaremos la colección importante de GC, alrededor de 50º segundo en la imagen de abajo:




Observe cómo el GC es ahora capaz de recuperar parte de la memoria utilizada por la JVM.

**Cómo prevenirlo?**

Ahora que conoce el escenario, hay por supuesto maneras para evitar que se produzcan.
En primer lugar, tenemos que **prestar mucha atención a nuestro uso de la *electricidad***estática;declarar cualquier colección o un objeto pesado como *estáticos* lazossu ciclo de vida para el ciclo de vida de la propia JVM, y hace que todo el gráfico de objetos imposible recoger.
También tenemos que ser conscientes de las colecciones en general - que es una forma común de mantener involuntariamente a las referencias de tiempo de lo que necesitamos.

### 2.2. Llamando *String.intern ()* en Long *String*

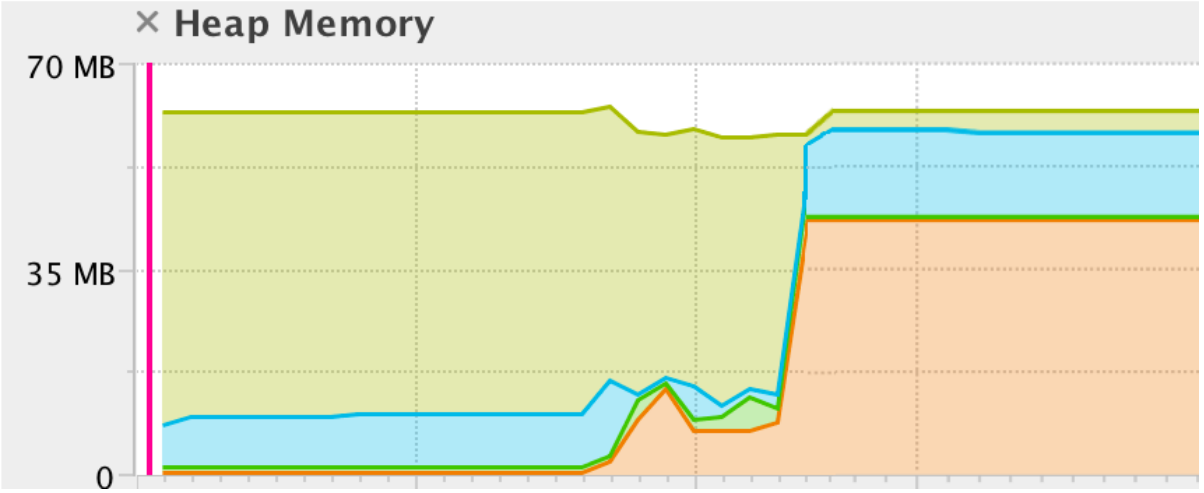
El segundo grupo de escenarios que frecuentemente provoca pérdidas de memoria implica *de cuerda* operaciones- específicamente la *String.intern* API*().*
Vamos a echar un vistazo a un ejemplo rápido:

```
@Test
public void givenLengthString_whenIntern_thenOutOfMemory()
  throws IOException, InterruptedException {
    Thread.sleep(15000);

    String str
      = new Scanner(new File("src/test/resources/large.txt"),
"UTF-8")
      .useDelimiter("\\A").next();
    str.intern();

    System.gc();
    Thread.sleep(15000);
}
```

En este caso, simplemente tratamos de cargar un archivo de texto de gran tamaño en la memoria de correr y luego regresar una forma canónica, utilizando.*pasante*().
La *pasante* API colocar la *str* cadenaen el banco de memoria de JVM - donde no pueda ser recogida - y otra vez, esto hará que la GC sea incapaz de liberar suficiente memoria:



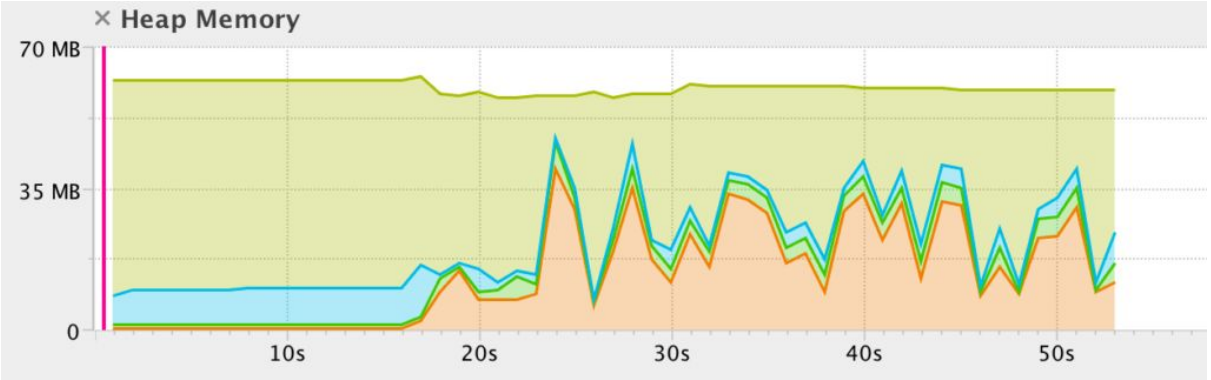

claridad Podemos ver que en el primer segundo 15a JVM es estable, entonces cargamos el archivo y ejecutar una JVM de Garbage collector (20 segundos).
Por último, el *str.intern ()* se invoca, que conduce a la pérdida de memoria - la línea estable que indica el uso de memoria de alta heap, que nunca será liberado.

**Cómo prevenirlo?**

Por favor, recuerde que i*nterned Strings* se almacenan en *PermGen* el espacio- si nuestra aplicación está destinada a realizar una gran cantidad de operaciones de grandes cadenas,que podrían tener que aumentar el tamaño de la generación permanente:

-XX: MaxPermSize = <tamaño>

La segunda solución es el uso de Java 8 - donde el *PermGen* espacio se sustituye por el *metaespacio* - que no conducirá a ninguna *OutOfMemoryError* cuando se utiliza *en prácticas* de cuerdas:




por último, también hay varias opciones de evitar la *.intern ()* APIen cadenas, por supuesto.

### 2.3. Sin cerrar corrientes

olvido para cerrar una secuencia es un escenario muy común, y, desde luego, que la mayoría de los desarrolladores pueden relacionarse. El problema se eliminó parcialmente en Java 7, cuando la capacidad de cerrar automáticamente todos los tipos de flujos se introdujo en el *try-with-resource*s.

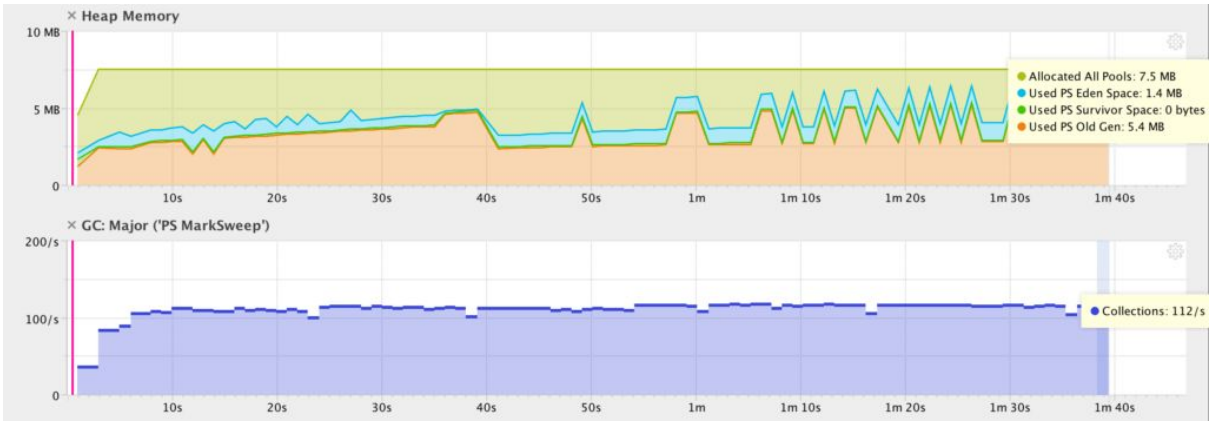
¿Por qué parcialmente? Debido a que **la *try-with-resources* sintaxis es** opcional:

```
@Test(expected = OutOfMemoryError.class)
public void givenURL_whenUnclosedStream_thenOutOfMemory()
  throws IOException, URISyntaxException {
    String str = "";
    URLConnection conn
                                                    =        new
URL("http://norvig.com/big.txt").openConnection();
    BufferedReader br = new BufferedReader(
                new   InputStreamReader(conn.getInputStream(),
StandardCharsets.UTF_8));

    while (br.readLine() != null) {
        str += br.readLine();
    }

    //
}
```

Vamos a ver cómo la memoria de la aplicación se ve cuando se carga un archivo grande desde una URL:




Como podemos ver, el uso del heap está aumentando gradualmente con el tiempo - que es el impacto directo de la pérdida de memoria causada por no cerrar el flujo .

Vamos a profundizar un poco más en este escenario porque no es tan clara como el resto. Técnicamente, una corriente sin cerrar dará lugar a dos tipos de fugas - una pérdida de recursos de bajo nivel y pérdida de memoria.

La pérdida de recursos de bajo nivel es simplemente la fuga de un recurso a nivel de sistema operativo - como descriptores de ficheros, conexiones abiertas, etc. Estos recursos también pueden tener fugas, al igual que lo hace la memoria.

Por supuesto, la JVM utiliza la memoria para realizar un seguimiento de estos recursos subyacentes, así, por lo que **este también se traduce en una pérdida de**memoria.

### Cómo prevenirlo?

Siempre tenemos que recordar que cerrar manualmente las corrientes, o para hacer un uso de la característica de auto-cierre introducido en Java 8:

```
tratamos
       (BufferedReaderbr    =    new    BufferedReader    (nuevo    InputStreamReader
(conn.getInputStream (), StandardCharsets.UTF_8))) {
    // aplicación
ulterior}catch (IOException e)
    {e.printStackTrace();
}
```

En este caso, el *BufferedReader* se cerrará automáticamente al final de la *try,* instrucción sin la necesidad de cerrarla de forma explícita *finally.* bloque

### 2.4. Conexiones Unclosed

Este escenario es bastante similar a la anterior, con la diferencia principal de tratar con conexiones no cerradas (por ejemplo, a una base de datos, a un servidor FTP, etc.). Una vez más, la aplicación inadecuada puede hacer mucho daño, lo que lleva a problemas de memoria.

Veamos un ejemplo rápido:

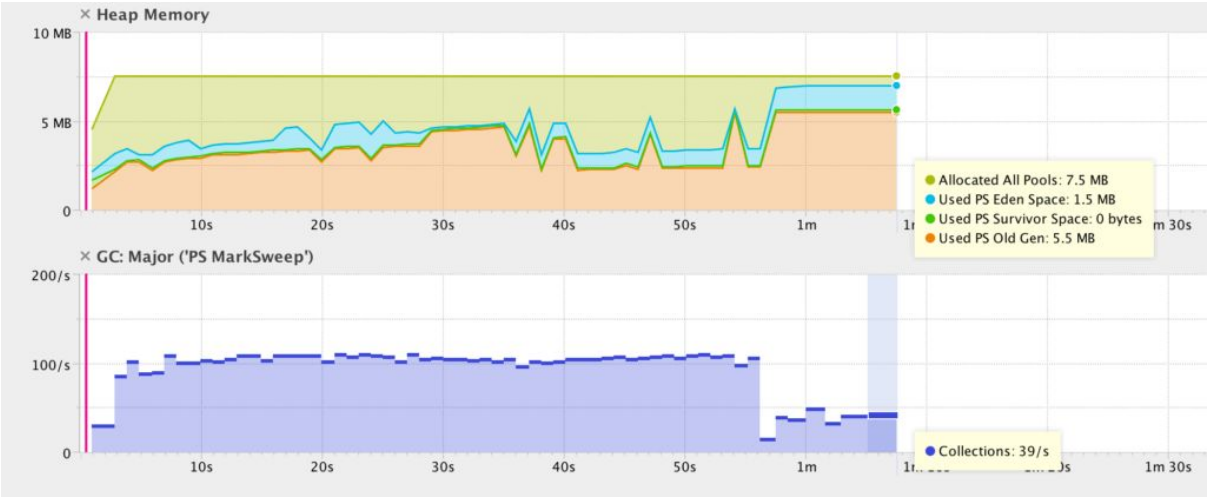
```
@Test(expected = OutOfMemoryError.class)
public void givenConnection_whenUnclosed_thenOutOfMemory()
  throws IOException, URISyntaxException {

    URL url = new URL("ftp://speedtest.tele2.net");
    URLConnection urlc = url.openConnection();
    InputStream is = urlc.getInputStream();
```

```
        String str = "";

        //
    }
```

El *URLConnection* permanece abierta, y el resultado es, como era previsible, una pérdida de memoria:




Observe cómo el recolector de basura no puede hacer nada para liberar sin usar, pero la memoria se hace referencia. La situación es inmediatamente claro después de la primera minutos - el número de operaciones de GC disminuye rápidamente, causando un aumento en el uso de memoria Heap, que conduce a la *OutOfMemoryError.*

**Cómo prevenirlo?**

La respuesta es simple - necesitamos siempre cerca de las conexiones de una manera disciplinada.

### 2.5. Adición de objetos sin *hashCode ()* y *equals ()* en un *HashSet*

Un ejemplo sencillo pero muy común que puede conducir a una pérdida de memoria es utilizar un *HashSet* con los objetos que faltan en su *hashCode ()* o *()iguales.* implementaciones

En concreto, cuando comenzamos a añadir objetos duplicados en un *Set* - esto sólo ha de crecer, en lugar de ignorar los duplicados como debería. Asimismo, no será capaz de eliminar estos objetos, una vez añadido.

Vamos a crear una clase simple sin que ninguno *es igual* o hashCode:

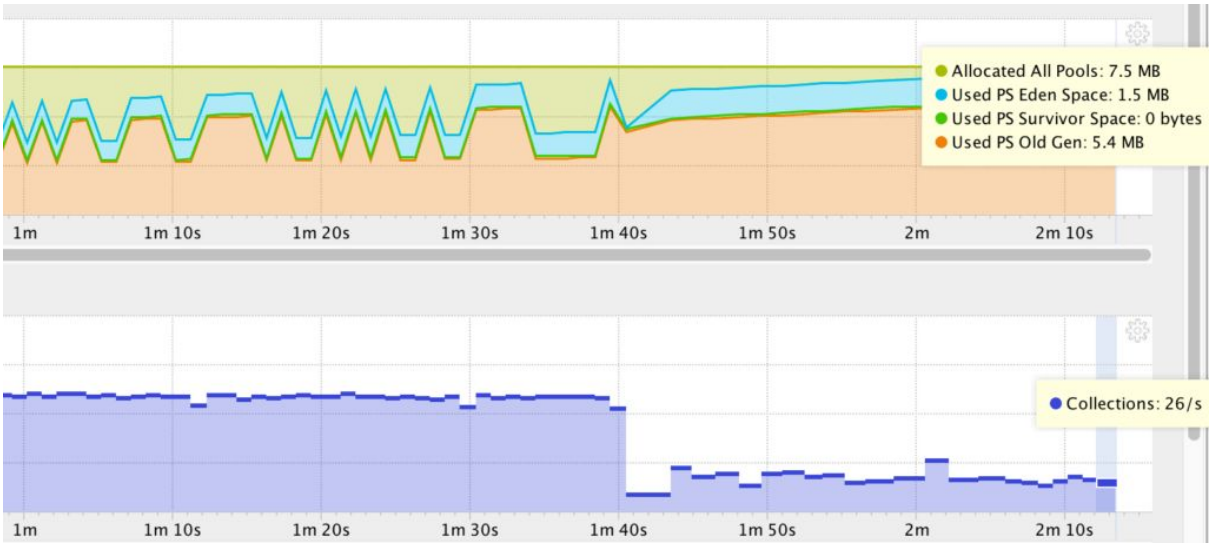
```
    public class Key {
        public String key;

        public Key(String key) {
            Key.key = key;
        }
    }
```

Ahora, vamos a ver el escenario:

```
@Test(expected = OutOfMemoryError.class)
public                                                    void
givenMap_whenNoEqualsNoHashCodeMethods_thenOutOfMemory()
  throws IOException, URISyntaxException {
    Map<Object, Object> map = System.getProperties();
    while (true) {
        map.put(new Key("key"), "value");
    }
}
```

Esta sencilla aplicación dará lugar a la siguiente situación en tiempo de ejecución:




Observe cómo el recolector de basura dejó de ser capaz de recuperar la memoria en torno a 1:40, y notar la pérdida de memoria; el número de colecciones de GC se redujo casi cuatro veces inmediatamente después.

**Cómo prevenirlo?**

En estas situaciones, la solución es simple - es crucial para proporcionar el *hashCode ()* y *equals ()* implementaciones.

Una herramienta importante mencionar aquí que Lombok - esto proporciona una gran cantidad de aplicación por defecto de anotaciones, por ejemplo @EqualsAndHashCode.

## 3. Cómo encontrar a Fuentes que se escapa en la aplicación de

diagnóstico de fugas de memoria es un proceso largo que requiere una gran cantidad de experiencia práctica, habilidades de depuración y el conocimiento detallado de la aplicación. Vamos a ver las técnicas que pueden ayudar, además de perfiles estándar.

### 3.1. Garbage collector verbosa

Una de las maneras más rápidas para identificar una pérdida de memoria es permitir la Garbage collector verbose.

Al añadir el *-verbose: gc* parámetro a la configuración de nuestra aplicación JVM, estamos permitiendo una huella muy detallada de GC. Los informes de resumen se muestran en el

archivo de salida de error por defecto, lo que debería ayudar a entender cómo se está manejando su memoria.

### 3.2. No Perfilado

La segunda técnica es la que hemos estado usando en este artículo - y eso es perfilado. El generador de perfiles más popular es Visual VM - que es un buen lugar para empezar a moverse más allá de la línea de comandos de JDK herramientas y en los perfiles de peso ligero.
En este artículo, se utilizó otra perfilador - YourKit - que tiene algunas características adicionales, más avanzados en comparación con Visual VM.

### 3.3. Comentario Su Código

Finalmente, esto es más de una buena práctica general que una técnica específica para hacer frente a las pérdidas de memoria.
En pocas palabras - revisar su código de fondo, la práctica de revisiones de código regulares y hacer buen uso de las herramientas de análisis estático para ayudarle a entender su código y su sistema.

## Conclusión

En este tutorial, tenía una mirada práctica a cómo ocurren pérdidas de memoria en la JVM. La comprensión de cómo ocurren estos escenarios es el primer paso en el proceso de tratar con ellos.
Luego, con las técnicas y herramientas para ver realmente lo que está sucediendo en tiempo de ejecución, ya que se produce la fuga, es fundamental también. El análisis estático de código y comentarios centrado-cuidadosas sólo se puede hacer mucho, y - al final del día - es el tiempo de ejecución que le mostrará las fugas más complejos que no son inmediatamente identificables en el código.
Por último, **los leaks pueden ser notoriamente difíciles de encontrar y reproducir debido a que muchos de ellos sólo suceda bajo carga intensa, que ocurre generalmente en la producción.** Aquí es donde tiene que ir más allá del análisis a nivel de código y trabajar en dos aspectos principales - la reproducción y la detección temprana.
La forma mejor y más fiable para **reproducir pérdidas de memoria** es simular los patrones de uso de un entorno de producción lo más cerca posible, con la ayuda de una buena serie de pruebas de rendimiento.
Y **la detección temprana** es donde un sólido solución de gestión de rendimiento e incluso una solución de la detección temprana puede hacer una diferencia significativa, ya que es la única manera de tener el conocimiento necesario en el tiempo de ejecución de la aplicación en la producción.