

Destrucción de objetos y *garbage Collection*

Todos los programas necesitan memoria para correr. En la memoria, almacenamos valores, etiquetados por lo que conocemos como “variables”.

En Java, almacenamos objetos en la memoria. Para ello, a diferencia de otros lenguajes, como C, no debemos reservar previamente la memoria para guardar objetos ni liberarla una vez que terminamos de usarlos. Estas acciones de reservar y liberar memoria se conocen como “**manejo de memoria**” o “**memory management**”. Para todos los lenguajes, principalmente los orientados a objetos, almacenar valores o la creación de objetos no suele ser un problema, aunque sí lo es la liberación de memoria o destrucción de objetos. No liberar memoria a tiempo puede hacer que un programa no responda o, peor aún, que simplemente el sistema operativo decida terminarlo.

Manual vs. automático

El *memory management* puede ser manual o automático. En muchos lenguajes, como Pascal, C, C++ o Delphi se usa el manejo de memoria manual. La gran mayoría de los lenguajes populares utiliza, hoy en día, manejo de memoria automático, como por ejemplo Java, JavaScript, PHP, Go, etc.

El manejo de memoria manual requiere de reservar memoria cada vez que se necesita crear un objeto y liberarla cuando ya no lo necesitamos más. Para reservar memoria, se debe saber cuánto va a ocupar el objeto o valor que vayamos a almacenar. En C, se utiliza el *infame malloc* para reservar memoria y luego la función *free* para liberarla. En Java, reservamos memoria con el *new* y, afortunadamente, no tenemos que hacer nada más, pues la memoria se liberará automáticamente.

En el manejo de memoria automático, todo ocurre detrás de escena. Se crean los objetos con *new* o alguna palabra clave similar, y luego hay un proceso que recorre la memoria y va detectando y recolectando objetos en desuso y destruyéndolos, liberando así la memoria. A este proceso se lo conoce como “**garbage collection**” o “**recolección de basura**” que comúnmente se los llama *GC* por sus siglas en inglés. También se suele llamar “*garbage collector*” al encargado de correr este proceso, identificado también por la sigla *GC*.

Es evidente que el manejo de memoria manual es mucho más complejo, dado que el programador no solo debe estar al tanto de la sintaxis, del negocio, de los modelos, sino también ser un experto administrador de la memoria. Cuando se reserva memoria y no se libera adecuadamente, se genera lo que se conoce como “*memory leak*”. Esta es una situación que puede llevarnos a tener programas lentos, situaciones críticas de nuestro programa o que simplemente se cierre por falta de memoria. Un manejo automático de memoria no nos protege contra un mal programador; pueden generar *memory leaks*, pero es mucho menos probable.

Habiendo manejo de memoria automático, ¿por qué existen todavía lenguajes con manejo de memoria manual? El *garbage collection* supone cierto trabajo adicional que podría afectar la *performance*. Dada su naturaleza, lenguajes destinados a controlar sistemas (sistemas operativos, cálculo computacional o sistemas continuos de producción, por ejemplo) requieren de un control total del *hardware* disponible y necesitan aprovecharlo al máximo, con lo cual no pueden perder tiempo de procesamiento con un proceso adicional de GC.

Garbage collection en Java

En Java, el GC corre automáticamente, aunque puede ser configurado, si es necesario, con algunos parámetros extra al momento de correr el programa. De todas formas, para correr el *garbage collector* no debemos hacer nada especial en nuestro código. El GC limpiará los objetos de la memoria bajo ciertas condiciones. Simplificando mucho, podemos mencionar algunas aquí:

- La referencia que antes apuntaba a un objeto ahora se hizo *null*.

```
Car c = new Car();  
c.setBrand("Ford");  
...  
...  
c = null;
```

- La referencia que antes apuntaba a un objeto ahora apunta a otro, haciendo al primer objeto elegible para el GC.

```
Car c = new Car();
```

```
Car d = new Car();
```

```
c = d;
```

Aquí c y d apuntan al mismo auto, el primero. El segundo queda desreferenciado y, por tanto, es candidato a GC.

- Un objeto que se usa una única vez y no queda almacenado en ninguna variable; por lo tanto, no está apuntado por ninguna referencia.

```
//que ruido hace un auto al arrancar?
```

```
new Car().start (); // “brrrooomm!”
```

Aquí, se instanció el auto y se llamó al método start() (que, supongamos, imprime “brrrooomm!” en pantalla), pero esa instancia se usó solo esa vez y para hacer eso. No está “atada” a ninguna referencia, por lo que es directamente elegible para GC.

El GC en Java pasa por tres fases fundamentales.

1. En una primera pasada, el GC recorre el *heap* y “marca” qué objetos son elegibles para realizar GC y cuáles no. En este punto, se determina cuáles son los objetos que están “vivos” y cuáles son considerados basura (“*garbage*”).
2. Luego, el GC libera la memoria, al destruir los objetos basura.
3. Por último, “compacta” todos objetos vivos para que queden todos en un bloque en el *heap* y que así sea más rápido el acceso a ellos (similar al “desfragmentar” de Windows, como para tener una idea).

Invocando al GC

Si bien el GC es un proceso automático en Java, podemos invocar manualmente al GC a demanda. Sin embargo, el proceso de GC durante las fases que vimos consume bastante CPU. Es por eso que, generalmente, no se invoca al proceso de GC manualmente: el *runtime* sabe muy bien cuándo hacer la recolección de acuerdo con una multitud de parámetros y algoritmos específicos. De todas formas, es posible, y siempre que tengamos objetos elegibles podemos lanzar la recolección con el método estático gc() de la clase System:

```
public class Test {  
  
    public static void main (String [] args) {  
  
        Car c = new Car();
```

```

    Car d = new Car();

    c = null;

    System.gc();
}
}

```

Aquí, el primer auto, al quedar des referenciado es elegible para *garbage collection*, entonces System.gc() comenzará, **aunque no necesariamente de forma inmediata**, el proceso de recolección.

El método finalize()

El método finalize() lo tienen disponible todos los objetos y lo podemos usar para cuando queremos realizar una acción en el momento en que el objeto es recolectado. Para ello, debemos escribir el método en nuestra clase:

```

public class Car {

    private String brand;

    private String model;

    // getters & setters

    public void start() {

        System.out.println("brrrooomm!");

    }

    //other methods

    ...

    public void finalize() {

        System.out.println("El auto esta por ser eliminado");

    }

}

```

Entonces, al correr el siguiente código:

```

public class Test {

    public static void main(String[] args) {

        System.out.println("Ejemplo de finalize");

        Car c = new Car();

        Car d = new Car();

        c = null;

        System.gc();

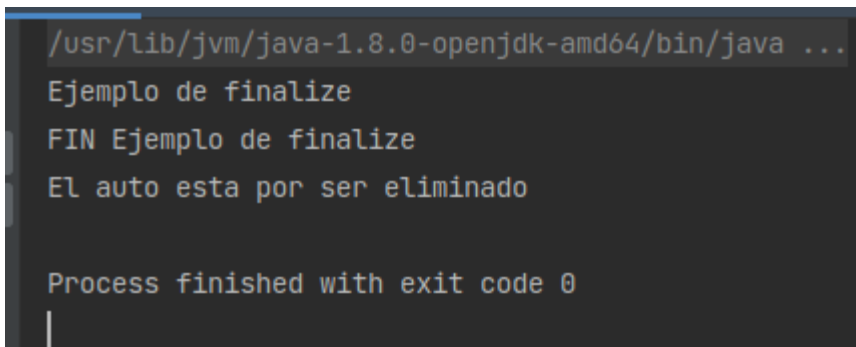
        System.out.println("FIN Ejemplo de finalize");

    }

}

```

Veríamos en la consola:



```

/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
Ejemplo de finalize
FIN Ejemplo de finalize
El auto esta por ser eliminado

Process finished with exit code 0

```

Observando de cerca podemos ver dos cosas. Primero, que se ejecutó el `System.out.println` que agregamos dentro del `finalize()`. Adicionalmente, que el GC no se realizó inmediatamente después de la llamada a `System.gc()`, ya que el mensaje está después, incluso, de la impresión de `System.out.println("FIN Ejemplo de finalize")`;

El uso manual de *finalize* es poco utilizado, pero puede usarse para liberar recursos que algún objeto está usando. De todas formas, hay que ser cuidadoso, ya que, en estos casos, su uso inadecuado puede causar problemas o puede que se termine el programa inesperadamente.