UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Recogiendo la basura.

2013
Transversal Programación Orientada a Objetos
Proyecto Curricular de Ingeniería de Sistemas

Introducción

No basta con inicializar una variable, hay que no olvidar limpiar la memoria muchas veces. Los objetos podrían no ser eliminados por el recolector de basura.

Generalmente se tiende a pensar que cuando dejamos de ejecutar un programa, pero no siempre las bibliotecas dejan que un objeto se libere una vez que se termina con este. Inclusive hay cosas de las que el recolector de basura no se hace cargo, como aquellos objetos a los que se les asigna memoria sin un **new**. Para estos casos "especiales" de creación existe la función **finalize()** la cual puede ser usada en cada clase, realizando un trabajo conjunto pues el recolector de basura invoca a este y mientras **finalize** se encarga de limpiar un espacio relacionado a algo importante el recolector limpiará el resto a la vez.

Se debe tener claro que la destrucción de un objeto (C++) dista de la recolección de basura que se realiza en Java, esta última tiene solo que ver con la memoria. También se debe tener en cuenta que si el espacio de almacenamiento no se satura y el programa se finaliza el recolector no es necesario y la sobrecarga que genera en cuanto gasto de recursos no se da.

1. Finalize() y su utilidad.

Este método está explícitamente relacionado con la memoria su des asignación.

Finalize() se requiere explícitamente para casos especiales, donde un objeto reservar memoria de almacenamiento sin la creación tradicional de un objeto (sin emplear el **new**()). Esos casos especiales son aquellos que se dan por medio de métodos nativos, los cuales son código no-Java, que se puede hacer necesario para acceder al sistema operativo de manera específica o para interactuar con dispositivos hardware. Un ejemplo es invocar la familia de funciones de **malloc()** de C para asignar espacio de almacenamiento, en C existe una función para des asignar dicho espacio denominada **free(),** sin embargo debería ser necesario invocar esa función en un método nativo desde el **finalize().**

Esto nos ayuda a concluir que el uso de esta función no es muy común.

2. Funciones de limpieza

Como se observa **finalize()** está diseñado para limpiezas de memoria no comunes que de hecho casi no se usarán, así que es necesario crear funciones de limpieza. Sin embargo **finalize()** brinda una funcionalidad aparte denominada verificación de la condición de muerte de un objeto.

Una vez que un objeto se desee eliminar (sea innecesario) este debe estar en un estado que permita liberar si memoria de manera segura, el caso de un fichero abierto, debe cerrarse antes de ser eliminado por el corrector de basura, sino se cierra puede acarrear un fallo , luego que no se elimina completamente en objeto, lo cual no será sencillo de supervisar.

Finalize() ayuda a encontrar el fallo y descubrir el problema.

Ejemplo:

```
class  Comic{
      boolean comprobado = false;
      Comic (boolean comprobar) {
            comprobado = comprobar;
      }
      void correcto () {
            comprobado = false;
      }
      public void finalize() {
            if (comprobado)
            System.out.println("Error: comprobado");
      }
}
public class CondicionMuerte {

public static void main (String[] args) {
      Comic aventura = new Comic(true) ;
// Eliminación correcta:
      aventura. correcto () ;
// Cargarse la referencia, olvidando la limpieza:
      new Comic (true) ;
// Forzar la recolección de basura y finalización:
      System.gc ();
}
}
```

El anterior main muestra un error de parte del programador quien no comprueba algunos libros y sin finalize () este error demoraría en encontrarse.

3. Recolector de basura

La existencia del recolector se debe a la necesidad de recuperar la memoria que un programa deja de utilizar.

La asignación de objetos en el montículo tal como lo maneja Java es costosa en cuanto tiempo, el recolector de basura ayuda que la liberación de espacio genere un impacto positivo en la asignación del mismo.

En Java, el espacio se asigna de manera contigua es decir digamos que se parcela y cada vez se añade un puntero al siguiente sucesivamente (en el libro de Eckel se define como una cinta transportadora), en vista de que los recursos no son infinitos, podría agotarse la memoria o darse un error de paginación excesiva. Aquí aparece el recolector, este constantemente moverá el puntero del montículo, pues reorganiza los elementos.

¿Cómo funciona?

El recolector de basura (Garbage Collector) emplea diversas técnicas entre ellas:

Técnica	Descripción	Rendimiento
Contar referencias.	Se tiene un contador de referencias que incrementa cuando se adjunta una referencia a un objeto. Si una referencia se pone en null se decrementa el contador.	Es lento y es una carga constante aunque pequeña que se produce durante la vida del programa. No se implementa en ninguna máquina virtual de Java
Adaptativo.	Los objetos vivos se pueden recorrer, realizando una traza y encontrando los objetos vivos. Cuando los encuentra puede hacer diversas cosas entre ellas: **parar y copiar**: Copia el objeto de un montículo a otro (requiere cambiar todas las referencias que apuntan a este objeto), dejando detrás toda la basura.	Más rápido. Al requerir dos montículos y manejar memoria del traslado entre estos, consumen demasiada memoria.
	Otro esquema es **marcar y barrer** donde cada vez que se encuentra un objeto vivo lo marca con un indicador, luego de acabar de marcar todo lo que puede, barre los objetos muertos.	

Bibliografía

- Eckel, Bruce. Thinking in Java. Prentice Hall. 1998. Estados Unidos.