Gestión de la memoria en Java

Introducción

Por lo que se refiere al manejo de datos en Java, podemos hablar de dos áreas de memoria: el *stack*[1] y el *heap*. Las variables locales se ubican en el *stack* automáticamente cuando se inicia la ejecución del método al que pertenecen y son eliminadas cuando la ejecución del método termina. Por su parte, todos los objetos (cualquier entidad de cualquier clase descendiente de la clase *Object*, es decir, cualquier entidad de cualquier clase) se ubican en el *heap* mediante una operación *new*, no existiendo ninguna operación explícita para eliminarlos.

Es importante comprender bien las implicaciones derivadas de este modelo.

El Heap y el Stack

Las variables locales y el stack

¿Qué significa que las variables locales se ubican en el *stack* y los objetos en el *heap*? ¿Cuál es la diferencia entre una variable local y un objeto? A todas estas, ¿qué es una variable local? Empezando por esta última cuestión, la respuesta es fácil (y se supone que conocida por aquellos a quienes va dirigido este documento): una variable local es cualquier variable declarada en un método[2]. Es local al método en el que está declarada en el sentido de que solo existe "dentro del mismo".

```
public void myMethod() {
    int x, y;
    float f;
    String s1, s2;
    …
}
```

En realidad, el ámbito (espacio en que una variable es accesible) puede ser menor que un método y restringirse a un bloque dentro de un método, si es ahí donde está declarada, y no en el bloque

[1] En realidad "los *stacks*", dado que, en una aplicación concurrente (que ejecuta varios "hilos" en paralelo), habría un *stack* por cada "hilo".
[2] A los efectos, los parámetros formales de un método (que aparecen en su lista de parámetros) también son variables locales del método, a las cuales se les asignan los valores de los parámetros reales en el momento de la llamada, con las implicaciones que se discutirán más adelante para la asignación.

principal del método. En el siguiente ejemplo, el ámbito de la variable *i* está restringido al bloque del bucle *for*.

```
for (int i = 0; i < n; i++) {
    …
}
```

Volviendo al primer ejemplo, se declaran cinco variables locales (*x*, *y*, *f*, *s1* y *s2*), que, como se ha dicho, se ubican en el *stack* (Ilustración 1), pero esta ubicación no tiene el mismo significado para todas.

Ilustración 1

La declaración de una variable de un tipo primitivo[3], como lo son *x*, *y* o *f* da lugar a la reserva de un trozo de memoria en el *stack* para representar un valor de ese tipo, que, en principio, tiene un valor indefinido, hasta que se inicializan (esto quiere decir que tienen un valor al azar, no, que no tengan un valor).

```
x = 0;
y = 0;
f = 0.0;
```

Ilustración 2

Por su parte, las variables cuyo tipo es una clase, como *s1* y *s2*, son solo referencias a objetos de esa clase que van a estar ubicados en el *heap*; es decir, lo que almacenan esas variables es una dirección de memoria con la que se puede acceder al objeto propiamente dicho. Como la declaración de la referencia por sí sola no da lugar a la existencia de un objeto, las referencias se inicializan automáticamente con el valor *null*, que significa que no "apuntan" a ningún objeto. Tras las declaraciones del primer ejemplo, tal como muestra la Ilustración 1, tenemos en memoria, dos valores enteros, un valor real, y dos referencias a objetos de la clase *String*, pero ningún objeto de la clase *String*.

[3] *byte*, *short*, *int*, *long*, *float*, *double*, *boolean* y *char*

Asignación de valores a las referencias

La forma básica de que una referencia pase de tener un valor *null* a referenciar a un objeto concreto de la clase correspondiente es ejecutando un constructor mediante la operación *new*[4] y asignando el resultado a la referencia:

```
s1 = new String("Hola mundo");
```

La operación *new* hace tres cosas:

1. Reserva una porción de memoria en el *heap*, suficiente para representar un objeto de la clase correspondiente.
2. Llama al constructor de la clase para inicializar esa memoria con los valores adecuados.
3. Devuelve la dirección que permite acceder a la memoria que representa el objeto

El efecto en nuestro ejemplo se muestra en la Ilustración 3, donde se ve que se ha creado un objeto de la clase *String* en el *heap*, se ha inicializado con el valor "Hola mundo" y se ha vinculado a la referencia *s1* al asignarle su dirección.

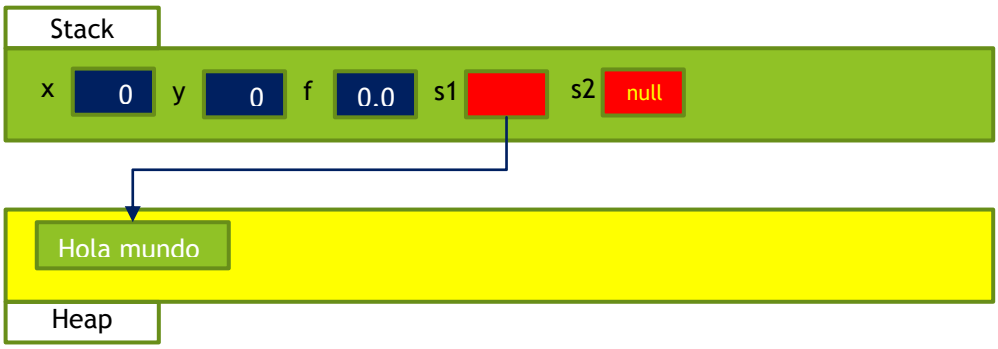

Ilustración 3

La creación de un objeto nuevo puede estar "escondida" dentro de una operación que cree un objeto nuevo a partir de uno ya existente. Por ejemplo, la operación *substring* de la clase *String* crea un nuevo objeto que inicializa copiando un trozo de un objeto existente (Ilustración 4).

```
s2 = s1.substring(0, 4);
```

[4] Para la clase *String* y las clases *wrapper* de los tipos primitivos, esta operación puede ejecutarse de forma implícita al asignar un valor literal: `String s = "Hola mundo"; Integer i = 3;`

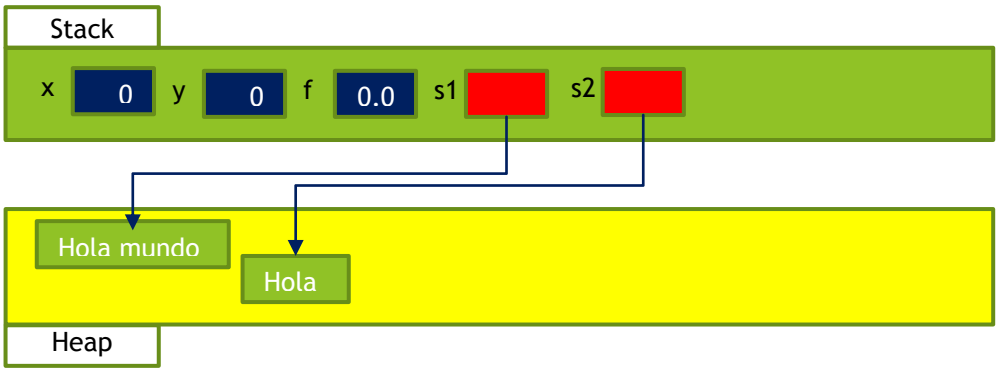

Ilustración 4

A una referencia, también, puede asignársele el valor de otra, con lo cual las dos "apuntarán" al mismo objeto (Ilustración 5).

```
s2 = s1;
```

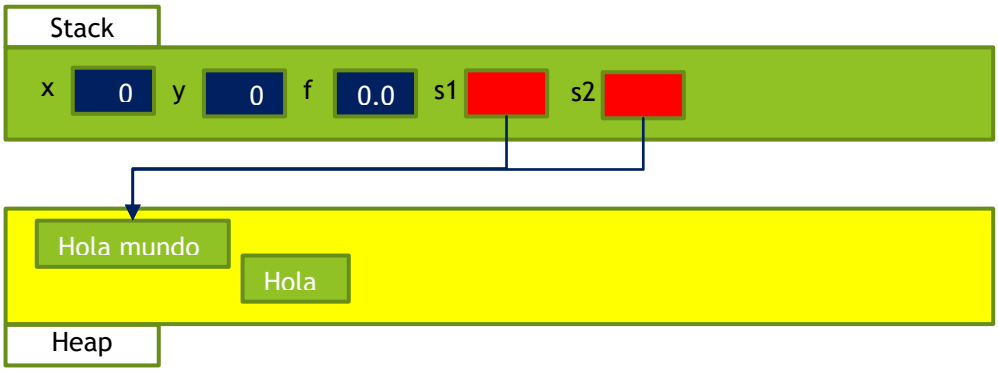

Ilustración 5

Atención a los *arrays*

Como acabamos de mostrar, una variable que no sea de un tipo primitivo es solo una referencia a un objeto, pero no el objeto en sí. Los objetos se crean usando la operación *new*, pero el uso de esta operación en el caso de los *arrays*, puede resultar confuso para algunos programadores noveles, que, ante una sentencia como la siguiente:

```
String[] v = new String[5];
```

pueden pensar que ya disponen de 5 objetos *String* que pueden utilizar, ya que han ejecutado un *new*. La realidad es que ese *new* lo que ha creado es un *array* con cinco referencias a objetos *String*, cada

una de ellas con el valor *null*. Este *array* es, a su vez, referenciado por la variable v, como muestra la Ilustración 6.

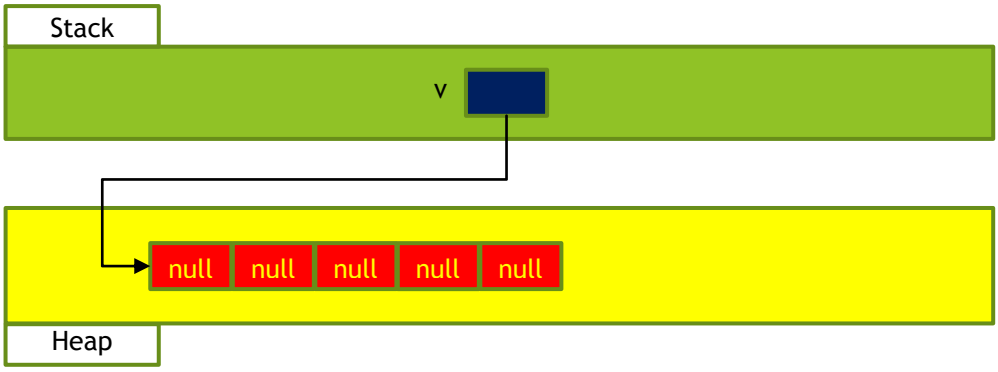

Ilustración 6

A cada elemento del *array* se le tendrá que asignar, bien la referencia de un objeto existente, bien una referencia a uno nuevo que se cree, como en el siguiente ejemplo:

```
for (int i = 0; i < v.length; i++) {
    v[i] = "Objeto " + i;
}
```

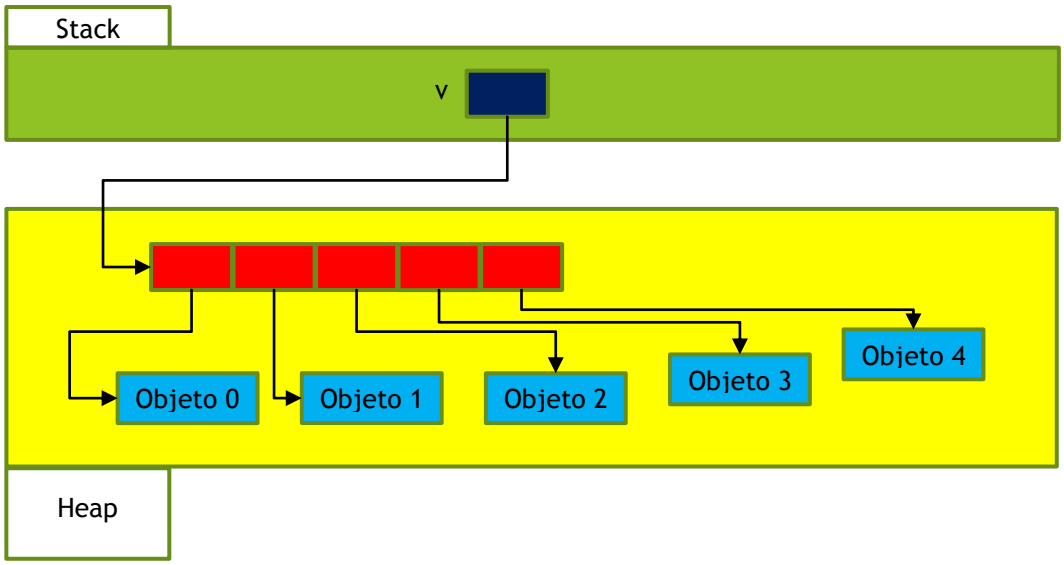

Ilustración 7

Naturalmente, si los elementos del *array* son de un tipo primitivo, sí que están creados al ejecutar la operación *new* para el *array*, ya que, en ese caso, no son referencias.

Consecuencias de la gestión de memoria basada en el *heap*

Alias, mutable e inmutable

El que dos referencias apunten al mismo objeto, es decir sean alias diferentes del mismo objeto, tiene consecuencias diferentes según los objetos de la clase en cuestión sean mutables o inmutables. Los objetos de la clase *String* son inmutables. Un objeto es inmutable cuando no existen operaciones que permitan cambiar su valor una vez creado. Si a una referencia *String* le asignamos un nuevo valor, aunque ese valor se construya a partir del que tenga en ese momento, se creará un nuevo objeto para representarlo, como muestra la Ilustración 4. Consideremos, ahora, la siguiente clase:

```
public class Cuadrado {
    private double lado;

    public Cuadrado(double lado) {
        this.lado = lado;
    }

    public double getLado() {
        return this.lado;
    }

    public void setLado(double lado) {
        this.lado = lado;
    }

    public double área() {
        return this.lado * this.lado;
    }

    public double perímetro() {
        return 4 * this.lado;
    }
}
```

Esta clase es mutable; tiene una operación (*setLado*) que permite modificar atributos del objeto después de crearlo, sin crear uno nuevo. Supongamos que creamos un objeto de la clase *Cuadrado* y copiamos luego su referencia:

```
Cuadrado c1 = new Cuadrado (3.0);
Cuadrado c2 = c1;
```

La situación quedaría como muestra la Ilustración 8.

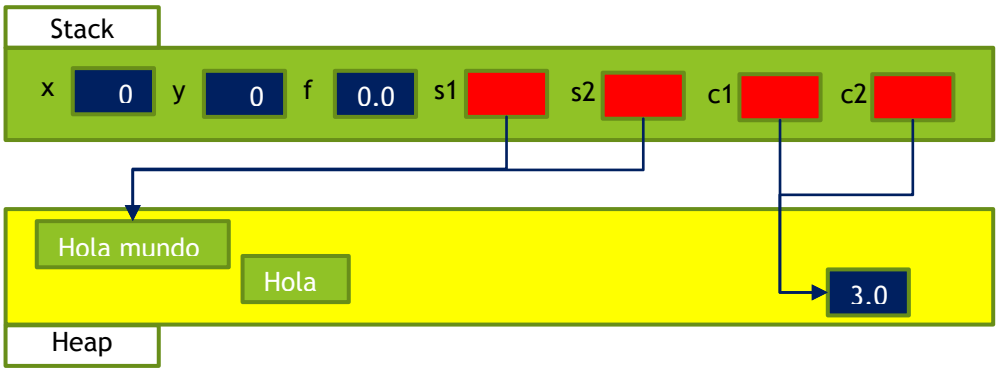

Ilustración 8

Si ahora, con cualquiera de las dos referencias, usamos una de las operaciones modificadoras:

```
c2.setLado(8.0); ≡ c1.setLado (8.0);
```

debemos ser conscientes de que el objeto que estamos modificando es el mismo al que se accede usando la otra referencia y, por tanto, queda modificado, accedamos por donde accedamos.

Comparación

Igualdad

En Java, los operadores relacionales (<, <=, ==, !=, >=, >) actúan en el *stack*; esto significa que obtendremos el valor *true* si, en la situación reflejada en la Ilustración 9, hacemos la siguiente comparación:

```
if (x == y)…
```

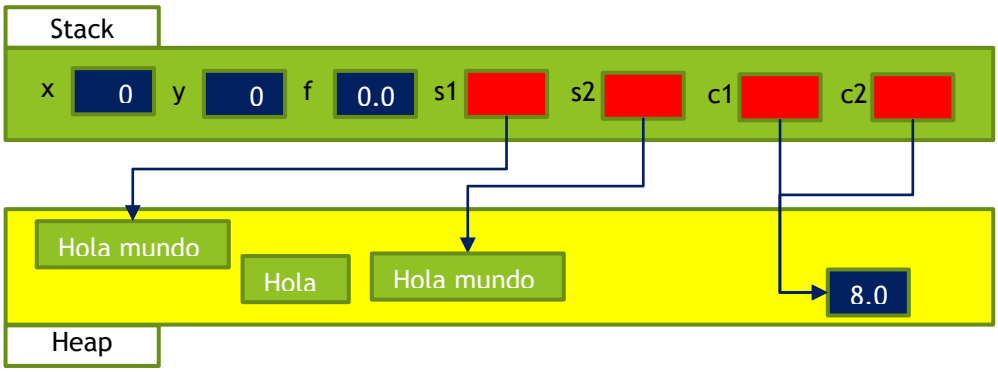

Ilustración 9

Sin embargo, si hacemos la comparación:

```
if (s1 == s2)…
```

obtendremos el valor *false*, ya que, aunque *s1* y s2 referencian objetos que tienen valores iguales, son objetos distintos, con distinta dirección de memoria, y son estas direcciones, que están contenidas en *s1* y *s2*, las que se comparan, no los objetos. Si, en la misma situación reflejada en la Ilustración 9, comparamos *c1* con *c2*, obtendremos el valor *true*, no porque los objetos referenciados por *c1* y *c2* sean iguales, sino porque ambas referencian al mismo objeto y, por tanto, ellas tienen valores iguales (la dirección de dicho objeto).

Si quisiésemos ver si los objetos de la clase *String* referenciados por *s1* y *s2* son iguales, usaríamos un método booleano llamado *equals*:

```
if (s1.equals(s2))…
```

El método *equals* es un método definido en la clase *Object*, que es la superclase última de todas las clases pero, normalmente, cada nueva clase con la que se deseen hacer comparaciones de igualdad debe redefinirlo para adaptarlo a su idiosincrasia. Eso es lo que han hecho los diseñadores de la clase *String*, y lo que deberíamos hacer en la clase *Cuadrado* para poder comparar los objetos de esa clase.

```
@Override
public boolean equals(Object c) {
    if (!(c instanceof Cuadrado)) {
        return false;
    } else {
        return (this.lado == ((Cuadrado) c).lado);
    }
}
```

La expresión *@Override* que precede al método indica que es una redefinición del correspondiente de su superclase (*Object*). Esa es la razón de que el parámetro sea de la clase *Object* y haya luego que, explícitamente, indicar que es un cuadrado cuando se usa. Se podría haber hecho un método *equals* con un parámetro de la clase *Cuadrado*, pero no sustituiría al de la clase base, con lo que, en determinadas situaciones, podría haber problemas, según el compilador decidiese, en función de contexto, usar uno u otro. También podríamos definir un método con cualquier otro nombre, pero nos separaríamos del uso estándar.

Nótese que, antes de comparar los lados del objeto referenciado por *this* y el referenciado por el parámetro *c*, se comprueba si dicho parámetro está realmente referenciando a un cuadrado; ello se debe a que el parámetro *c* ha sido declarado de tipo *Object*, no de tipo *Cuadrado*.

Otras comparaciones

Las mismas consideraciones hechas para los operadores relacionales de igualdad (== y !=) son válidas para el resto de operadores. Si queremos saber si un objeto de una clase determinada es mayor o menor que otro (suponiendo que la cuestión tenga sentido para esa clase), dicha clase deberá disponer de sus propios métodos para ese propósito. Por ejemplo, la clase *String* lo hace con un único método, llamado *compareTo*, que devuelve un valor negativo, cero o positivo, según el objeto con cuya referencia se llama al método sea menor, igual o mayor que el referenciado por el parámetro que se le pasa.

```
int comp = s1.compareTo("Adiós mundo");  // devuelve un valor positivo
                                         // para el caso de la Ilustración 9
```

Para hacer algo parecido para la clase *Cuadrado* primero debemos definir cómo consideramos que un cuadrado es mayor o menor que otro. Supondremos que, dados dos cuadrados, es mayor el que tenga un lado mayor, y vamos a limitarnos a devolver los valores -1, 0 o +1 para indicar que el pasado como parámetro es el menor, que son iguales, o que es el mayor, respectivamente.

```
public int compareTo(Cuadrado c) {
    if (c == null || this.lado > c.lado) {
        return -1;  // this referencia al mayor
    } else if (this.lado < c.lado) {
        return 1;   // this referencia al menor
    } else {
        return 0;   // son iguales
    }
}
```

Con esto cumpliríamos nuestro propósito de tener un método general de comparación. Si, además, modificamos la cabecera de la clase de la siguiente forma:

```
public class Cuadrado implements Comparable<Cuadrado> {
```

estaríamos declarando que nuestra clase es comparable usando un método concreto (*compareTo*) con un comportamiento conocido. *Comparable<T>*, donde *T* es una clase, es lo que en Java se conoce como una interfaz, que, a grandes rasgos, viene a ser una especie de superclase con la que no se pueden crear objetos y que lo único que tiene es una lista de métodos (solo la lista, sin implementaciones), de tal manera que cualquier clase que declare que implementa (*implements*) esa interfaz lo que hace es comprometerse a implementar los métodos listados por ella; en este caso, el método *compareTo*.

Asignación y copia

La gestión de memoria basada en el *heap* también causa una asimetría en el comportamiento de la asignación entre los tipos primitivos y las clases. El comportamiento "tradicional" de la asignación es copiar la entidad asignada, que es lo que ocurre en Java con las entidades almacenadas en el *stack*, pero, como los objetos no se almacenan en el *stack*, asignar a una variable, cuyo tipo es una clase, la referencia de un objeto existente solo copia la referencia, pero no el objeto en sí, como ya hemos visto en un par de ejemplos anteriores y muestra la Ilustración 10.

```
s2 = s1;
```

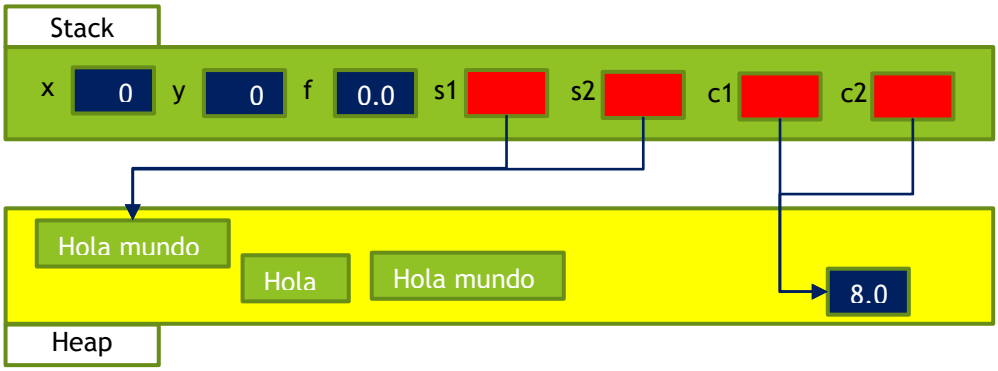

Ilustración 10

Si necesitamos tener un objeto separado, lo cual puede tener más sentido cuando los objetos son mutables, necesitamos dotar a la clase de mecanismos para copiar objetos. Una opción sencilla es tener un constructor que acepte como parámetro un objeto de la clase y lo use para inicializar un nuevo objeto copiando su información (Ilustración 11).

```
s2 = new String(s1);
```

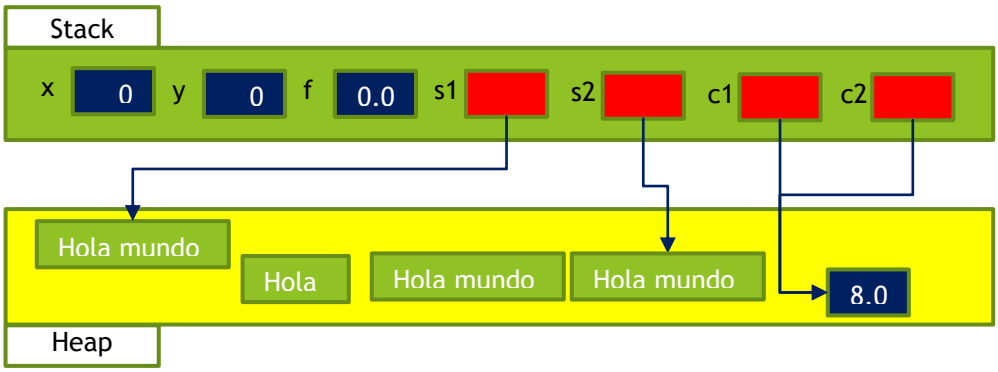

Ilustración 11

También se puede tener un método ordinario que, al llamarlo, cree un objeto nuevo y copie la información del objeto con el que se llama. En este caso, una opción estándar es redefinir el método *clone* de la clase *Object* para que lo haga.

Recuperación de la memoria del *heap*

En las sucesivas ilustraciones de los ejemplos, a medida que hemos asignado nuevos valores a las referencias, han ido quedando objetos en el *heap* que no son referenciados (objetos con los valores "Hola y "Hola mundo" en la Ilustración 11) y, que por tanto son inaccesibles. A diferencia de las variables locales, el tiempo de vida de los objetos en el *heap* no está ligado con el de los métodos donde se crean. Esto no significa que el *heap* vaya a llenarse de objetos sin uso hasta colapsarse; para evitarlo, en Java se usa un *garbage collector* ("recolector de residuos") que es un proceso que se lanza automáticamente cuando es necesario para recuperar la memoria ocupada por objetos que han quedado inaccesibles.

Hay otros lenguajes que dejan la responsabilidad de recuperar la memoria al programador, que debe usar alguna operación complementaria del new para destruir los objetos creados en el *heap* cuando deja de necesitarlos. Esta política requiere que el programador sea muy cuidadoso porque la liberación explícita de memoria puede dar lugar a problemas bien conocidos, cuya explicación escapa al propósito de este documento.