LA PILA Y LA RECURSIVIDAD

Partes de la memoria de un programa Java

Para ejecutar un programa java la MVJ solicita memoria libre al sistema operativo. Luego, dentro de esa memoria asignada al programa, se organizan una serie de zonas tal y como explica en
https://www.cs.swarthmore.edu/~newhall/unixhelp/Java_mem.pdf
Hay muchas explicaciones al respecto pero la url de arriba lo explica de forma sencilla y apropiada para nosotros. Lo que me gustaría matizar es que la Pila realmente almacena "llamadas a métodos" no es sólo "un almacén de variables locales"

Análisis de funcionamiento de la pila

Para entender el funcionamiento interno de la recursividad, es necesario conocer cómo funciona la pila de llamadas, aunque realmente una vez visto y creído el funcionamiento interno de la recursividad debemos olvidarnos de la existencia de la pila y procurar pensar a alto nivel.
Sintéticamente, un programa java tiene asignada memoria RAM para su ejecución. Esta memoria RAM se organiza en una serie de subzonas. Ahora necesitamos hacer referencia a las subzonas pila (stack) y el montículo(heap), pero hay más subzonas. En el montículo se almacenan los objetos. En la pila se almacena información del estado de un método que fue llamado.

Cada vez que se invoca un método, en la pila se reserva espacio para guardar el estado de ejecución de ese método, lo que implica guardar respecto al método:

- parámetros y variables locales con su valor actual
- dirección de la instrucción en ejecución
- dirección de retorno (la dirección del método que llamó a este)

Cuando acaba la ejecución del método esta memoria se libera.
Una explicación gráfica de la relación entre pila e invocación a métodos podemos consultarla en
https://www.geeksforgeeks.org/run-time-stack-mechanism-java/

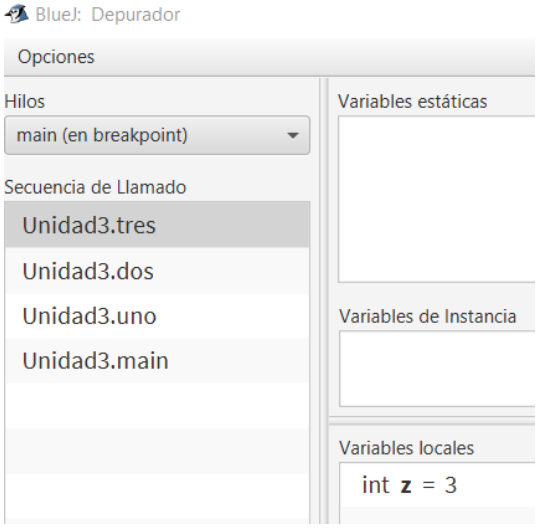
Para entender un poco mejor qué es la pila de llamadas, vamos a utilizar el tracer de BlueJ con el siguiente código, código sin sentido lógico que simplemente usamos para observar que un método llama a otro y que cada llamada a método tiene asociada un contexto de variables locales

```
class Unidad3 {

    static void uno() {
        int x=1;
        System.out.println("hola uno");
        dos();
        System.out.println("adios uno");
    }

    static void dos() {
        int y=2;
        System.out.println("hola dos");
        tres();
        System.out.println("adios dos");
    }
```

```
    static void tres() {
        int z=3;
        System.out.println("hola tres");
        System.out.println("adios tres");
    }

    public static void main(String args[]) {
        System.out.println("hola main");
        uno();
        System.out.println("adios main");
    }
}
```

Como novedad en el uso del tracer, ahora en lugar de utilizar *Step* **utilizaremos *Step Into***. *Step into* también "ejecuta paso a paso" las instrucciones del método que se llama, a diferencia de *Step* que no entra paso a paso en el método y lo ejecuta de un tirón. Nos interesa ahora usar *step into*.

Observa detenidamente en el tracer el cuadro *Secuencia de llamado o Call Sequence* Es muy didáctico. En este cuadro se aprecia la cadena de llamadas entre métodos que aún no terminaron su ejecución. Por ejemplo, en el gráfico de abajo como tres() está en la cima de la pila es el método que está en ejecución. El método dos() está "no activo" o "a medio ejecutar" esperando a que tres() acabe y se elimine de la pila con lo que dos() pasa a la cima de la pila y pasa a estar de nuevo activo



Observamos en la ejecución paso a paso que en la pila de llamadas puede haber apiladas una o muchas llamadas a métodos. La llamada que está en la cima de la pila decimos que está activa. Las otras llamadas, las que no están en la cima, están a medio ejecutar, cuando vuelvan a estar en la cabeza de la pila continuarán su ejecución.

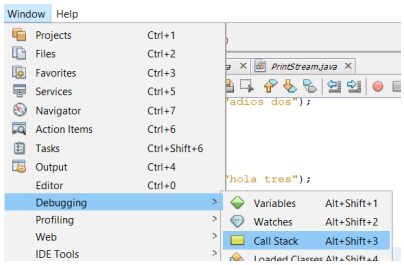
Debe quedarte claro que la ejecución de un método está ligada a una entrada en la pila para ese método y cuando llega a su última instrucción se libera en la pila el espacio de sus variables locales para esa llamada

Los métodos pueden ser de distintas clases, siempre hay una pila por programa(Realmente hay una pila por hilo. Hilo es un concepto que veremos más adelante) por ejemplo podemos probar a ejecución paso a paso con bluej

```
class Unidad3{
  public static void main(String args[]){
        int x=5;
```

```
        int y=10;
        A a=new A();
        B b =new B();
        x=a.modificarDeA(x);
        y=b.modificarDeB(y);
    }
}

class A{
    int modificarDeA(int i){
        int j=3;
        return j+i;
    }
}

class B{
    int modificarDeB(int k){
        int p=2;
        int z=0;
        z= suma1(p);
        return k+z;
    }
    int suma1(int w){
        return w + 1;
    }
}
```

Después de verlo con bluej también puedes verlo con netbeans. Al depurar debes de tener abierta la ventana call stack



RECURSIVIDAD

Hay dos mecanismo para ejecutar un bloque de instrucciones repetidamente:

- escribiendo un bucle
- escribiendo un método recursivo

La recursividad en programación consiste en que un método se invoque a sí mismo, de forma que contenga al menos una instrucción que consista en llamarse a sí mismo. Los métodos que se llaman a sí mismo se llaman recursivos.

nota: los siguientes ejemplos utilizan un método recursivo imprimir() cuyo resultado se conseguiría más fácilmente con un bucle pero lo hacemos recursivo simplemente para ejemplificar el funcionamiento de la recursividad con algo sencillo

Ejemplo1: el método imprimir() se llama así mismo, es por tanto un método recursivo. ¿Se llama a sí mismo infinitas veces? NO, llegado un punto, se para porque se produce desbordamiento de pila (stack overflow). Si no hubiera desbordamiento de pila el efecto sería el mismo que el de un bucle infinito.

```
class Unidad3{
  public static void main(String args[]){
        Recursividad re=new Recursividad();
        re.imprimir(5);
    }
}
```

```
class Recursividad {
   void imprimir(int x) {
      imprimir(x);
      System.out.println(x);
   }
}
```

Cuando ejecutas el programa anterior nunca se llega a ejecutar ningún println() ya que justo antes del println() hay una nueva llamada a imprimir y así sucesivamente …..hasta que desborda la pila.
Cada llamada a un método recursivo implica una reserva en la pila para esa llamada. Si se llamó 100 veces al método imprimir, en la pila se habrán creado reservas de memoria para ese método 100 veces, cada vez probablemente con unos valores de variables diferentes. Un método recursivo mal diseñado, puede invocarse a sí mismo indefinidamente (tendiendo a infinito), y cómo por cada invocación hay una reserva de memoria en la pila, si se producen múltiples invocaciones y no se termina la ejecución de ninguna, estamos pidiendo espacio en la pila y no liberando ninguno. Como el programa tiene una cantidad de RAM limitada y por tanto también una cantidad limitada para su pila, tarde o temprano se agota la memoria de la pila y se produce el famoso desbordamiento de pila

Un método recursivo bien diseñado tiene que parar de forma similar a como lo hace un bucle, de forma que el código debe especificar una condición que detecta cuándo parar y por otro lado el código del método debe escribirse de tal forma que cada llamada produzca un valor que en algún momento cumpla la condición de parada.

Ejemplo2: Añadimos al método recursivo un mecanismo de parada.

En este ejemplo, se controla como parar las llamadas recursivas a través de una condición, y para que esa condición se llegue a cumplir, x va cambiando de valor en cada llamada. Cuando x llegue a valer 1 se invoca imprimir(1-1), es decir a imprimir(0). Con imprimir(0) las llamadas recursivas paran ya que no se entra en el if

```
class Unidad3{
  public static void main(String args[]){
      Recursividad re=new Recursividad();
      re.imprimir(5);
   }
}
class Recursividad {
   void imprimir(int x) {
      if(x>0){//cuando x<=0 no hay llamada recursiva: se para.
         System.out.println(x);
         imprimir(x-1);//observa que x evoluciona
      }
   }
}
```

la salida imprime 5,4,3,2,1

Ejemplo3: observa cómo al cambiar de orden la invocación del método ahora imprimo al revés (1,2,3,4,5)

```
class Recursividad {
   void imprimir(int x) {
      if(x>0){
          imprimir(x-1);
         System.out.println(x);
      }
   }
}
```

```
class Unidad3{
  public static void main(String args[]){
       Recursividad re=new Recursividad();
       re.imprimir(5);
   }
}
```

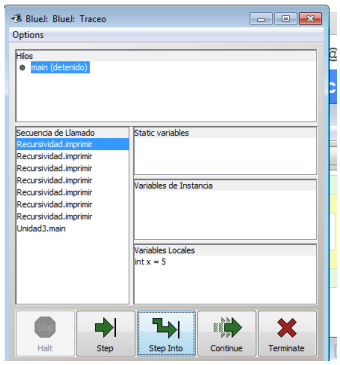
EJECUTAR LOS TRES EJEMPLOS ANTERIORES CON TRACER

Ahora ejecutaremos de nuevo los tres ejemplos, desde BlueJ, utilizando el tracer. Colocamos un punto de interrupción en la llamada al método. Ponemos primero la clase Unidad3 por que en bluej si comenzamos con la clase Recursividad nos cambia el nombre del fichero .java

```
class Unidad3{
   public static void main(String args[]){
        Recursividad re=new Recursividad();
        re.imprimir(5);
   }
}
class Recursividad {
    void imprimir(int x) {
        imprimir(x);
        System.out.println(x);
    }
}
```

EJEMPLO1:

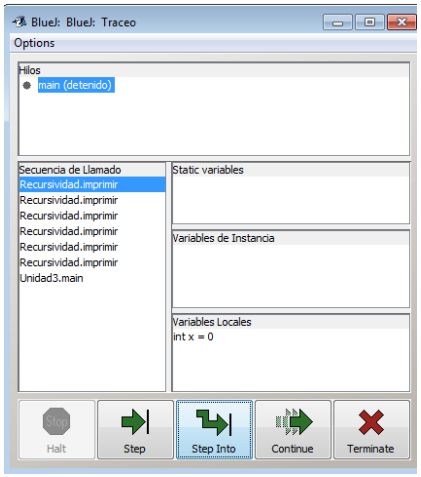
El primer método invocado por la MVJ, es main, luego main llamó a imprimir, este imprimir llamó a otro imprimir y así sucesivamente. En azul está el último método llamado y en el cuadro variables locales están las variables locales correspondientes a esa llamada. En este caso, la variable local x vale 5 en todas las llamadas. Esto último se entiende mejor probando ejemplo2 y ejemplo3 donde la variable x toma valores diferentes en cada llamada.



en cualquier caso para constatar que cada llamada tiene sus propios valores para parámetros y variables locales podemos añadir una variable y que tome valores aleatorios y vemos que en cada llamada lógicamente y cambia de valor.

```
class Unidad3{
  public static void main(String args[]){
       Recursividad re=new Recursividad();
       re.imprimir(5);
  }
}
class Recursividad {
    void imprimir(int x) {
       double y=Math.random();
       imprimir(x);
       System.out.println(x);
    }
}
```

EJEMPLO2:
Ejecutamos ahora el ejemplo 2, la foto de abajo está tomada en el momento que hay la mayor cantidad de "llamadas sin terminar" en la pila. Observa que hay 5 llamadas al método imprimir, cada llamada tiene sus propios datos en la pila, puedes comprobarlo ya que haciendo clic en cada uno de ellos ves que el valor de x es diferente



y al seguir progresando con el tracer desde que el primer método termina, todos terminan en cadena

EJEMPLO3:
Ejecutamos ahora el ejemplo 3 con el tracer ies el más didáctico para ver la evolución de la pila y para entender la recursividad!
Haz pruebas similares, con step into y debes de entender la evolución de los valores de las ventanas del tracer y "atar cabos conceptuales".

Observa que un método puede invocarse muchas veces y puede ocurrir que del mismo método existan en un momento dado muchas llamadas en la pila. Hay un pequeño paralelismo con el hecho de que una clase puede tener muchas instancias(objetos), podemos pensar, aunque no sea del todo correcto, que una llamada a un método es una especie de instancia del método.