

RECURSIVIDAD

Análisis de funcionamiento de la pila

Para entender bien recursividad, es conveniente, entender cómo funciona la pila de llamadas.

Sintéticamente, un programa java tiene asignada memoria RAM para su ejecución. Esta memoria RAM se organiza en una serie de subzonas, las dos principales son la pila y el montículo. En el montículo se almacenan los objetos. En la pila se almacenan las variables locales de las llamadas a los métodos. Cada vez que se invoca un método, en la pila se reserva espacio para las variables locales del método y también para almacenar el valor de retorno y otros detalles. Cuando acaba la ejecución del método esta memoria se libera.

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

```
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);
    }
}

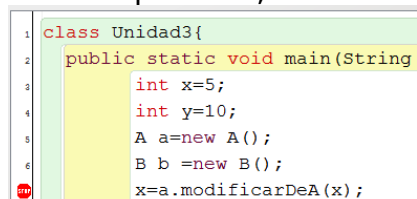
class A{
    int modificarDeA(int i){
        int j=3;
        return j+i;
    }
}

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;
    }
}
```

Como novedad en el uso del depurador, 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 en el tracer con detalle el cuadro *Secuencia de llamado*. Es muy didáctico. En este cuadro se aprecia la cadena de llamadas entre métodos que aún no terminaron su ejecución.

Y con el profesor, marcas el punto de interrupción y analizas la ejecución del depurador



```
1 class Unidad3{
2     public static void main(String
3         int x=5;
4         int y=10;
5         A a=new A();
6         B b =new B();
7         x=a.modificarDeA(x);
```

Debe quedarte claro que cuando un método llega a su última instrucción se libera en la pila el espacio de sus variables locales

RECURSIVIDAD

Hay dos mecanismos 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.

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.

Ejecuta este ejemplo en consola y también en Netbeans ya que se ve mejor el error que se produce que en consola (por cuestiones de visualización).

```
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 hay una nueva llamada a imprimir y así sucesivamentehasta que peta la pila.

Cada llamada a un método recursivo implica una reserva en la pila para esa llamada. Aunque el código java siempre es el mismo y está almacenado con el objeto, para cada llamada a ese método se crea una entrada en la pila. 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){//hay una condición de parada
            System.out.println(x);
            imprimir(x-1);//observa que x evoluciona
        }
    }
}
```

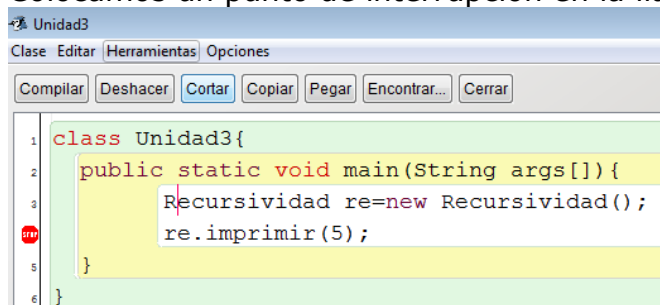
la salida imprime 5,4,3,2,1

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

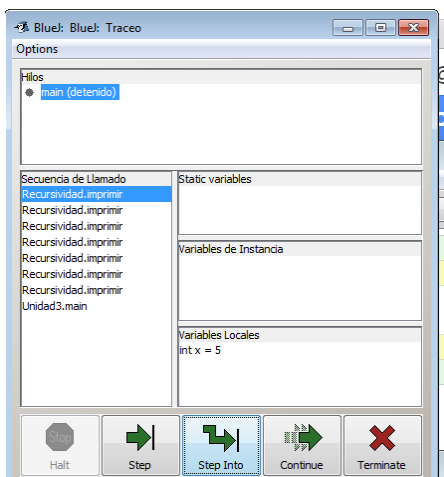
Ahora ejecutaremos de nuevo los tres ejemplos, desde BlueJ, utilizando el tracer. Colocamos un punto de interrupción en la llamada al método



EJEMPLO1:

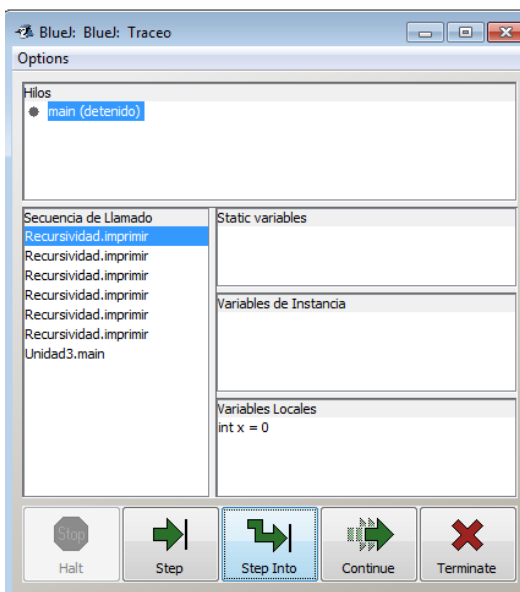
El primer método es invocado por la MVJ, es main, luego main llamó a imprimir, este imprimir llamó a otro imprimir y así sucesivamente. En azul está el último método llamado y en el cuadro variables locales están las variables locales correspondientes a esa llamada. En este caso, la variable local x 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.



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".

ESTRUCTURA DE UN MÉTODO RECURSIVO

Formalizando lo visto anteriormente, un método recursivo debe contener:

- Uno o más casos base: casos para los que existe una solución directa.
- Una o más llamadas recursivas: casos en los que de alguna forma se incluye una llamada recursiva.

Caso base: Siempre ha de existir uno o más casos para los que el método devuelve un valor directo, es decir, sin hacer uso de recursión. Es necesario ya que el caso base es donde el método recursivo "se detiene".

En el ejemplo anterior el caso base está encubierto (ya que el else es "no hacer nada")

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

Para entender mejor cuál es el caso base reescribimos el método

```
class Recursividad {  
    void imprimir(int x) {  
        if(x==0){//caso base  
            System.out.print("");  
        }else{//llamadas recursivas  
            System.out.println(x);  
            imprimir(x-1);//no hay infinitas llamadas por que x evoluciona  
        }  
    }  
}
```

Llamada recursiva: Si los valores de los parámetros de entrada no cumplen la condición del caso base se llama recursivamente al método. En las llamadas recursivas el valor del parámetro en la llamada se ha de modificar de forma que se aproxime cada vez más hasta alcanzar al valor del caso base. En el ejemplo el caso recursivo sería el bloque

```
{//llamadas recursivas  
    System.out.println(x);  
    imprimir(x-1);//no hay infinitas llamadas por que x evoluciona  
}  
que incluye la llamada recursiva
```

Ejemplo con varios casos base: Multiplicación con recursividad

Ya tengo una instrucción java que me multiplica números enteros así que no vemos este ejemplo por utilidad si no por didáctica.

Para simplificar suponemos que se multiplican números enteros ≥ 0

Si razonamos que multiplicar es sumar sucesivamente puedo escribir un método multiplicar recursivamente

multiplicar(a,b)=>sumar a , b veces

la definición recursiva podría ser:

caso base $b=1$: devuelve a (es decir $a*1=a$)

caso recursivo $b>1$: devuelve $a + \text{multiplicar}(a,b-1)$

En java

```
class Unidad3 {  
    public static void main(String[] args) {  
        Multiplicacion m = new Multiplicacion();  
        System.out.println(m.multiplicar(2,10));  
        System.out.println(m.multiplicar(1,3));  
        System.out.println(m.multiplicar(0,9));  
        //System.out.println(m.multiplicar(2,0));  
    }  
}
```

```
class Multiplicacion{  
    int multiplicar(int a, int b){  
        if(b==1)  
            return a;  
        else  
            return a + multiplicar(a,b-1);  
    }  
}
```

la última multiplicación está comentada por que provoca stack overflow ya que cuando b vale 0 no hay que caso base que pare la recursión. Debes de entender porqué ocurre esto.

La multiplicación se comporta de forma diferente cuando b es 1 o 0, necesito dos casos base:

```
class Multiplicacion{  
    int multiplicar(int a, int b){  
        if(b==0)  
            return 0;  
        if(b==1)  
            return a;  
        else  
            return a + multiplicar(a,b-1);  
    }  
}
```

Los casos de recursividad con varios casos recursivos irán saliendo a lo largo del curso.

EJERCICIO 1. Calcular el factorial de un número con bucle.

Recuerda la definición de factorial para un número n :

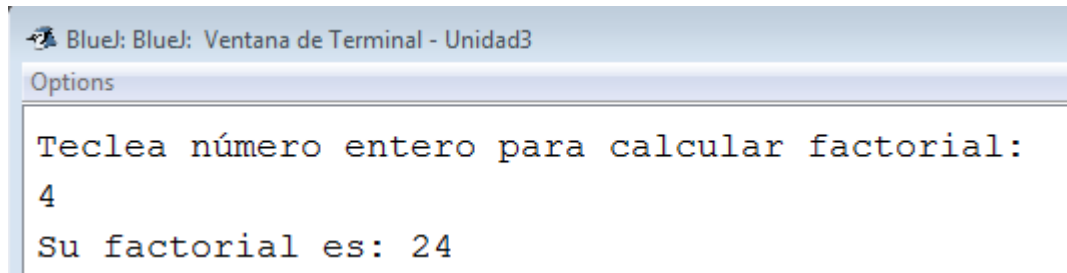
Si $n = 0$ entonces

$0! = 1$

si $n > 0$ entonces

$n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$

Tu programa tendrá el siguiente aspecto



```
Blue: Blue: Ventana de Terminal - Unidad3
Options
Teclea número entero para calcular factorial:
4
Su factorial es: 24
```

EJERCICIO 2. Calcular el factorial de un número con recursividad

Podemos definir el factorial de un número $n \geq 0$ de forma recursiva

$0! = 1$

$n! = n \cdot (n - 1)!$, si $n > 0$ (El factorial de n es n multiplicado el factorial de $n-1$)

Los ejemplos anteriores fueron bastantes "forzados" en el sentido que se usaron para ejemplificar diversos aspectos de la recursividad pero no tenían realmente utilidad. Factorial por el contrario ya es un ejemplo en el que la solución recursiva, puede ser más fácil y natural que con bucle.

Recursividad versus iteración con bucles

La recursividad es más ineficiente en términos de tiempo y uso de memoria, ya que para cada llamada al método hay que efectuar una serie de operaciones en la pila, no obstante, hay problemas que por su naturaleza recursiva son mucho más fáciles de resolver con recursividad que con intrincados bucles. Por lo tanto, si la eficiencia no es un problema y el problema a resolver tiene una naturaleza recursiva, mejor recursividad ya que se produce un programa más fácil de entender y de depurar. Con todo, este tipo de situaciones, son las menos y lo que abunda es la utilización de bucles más que llamadas recursivas.