

Hoja resumen del tema 3: Recursividad

Nota: esta hoja es tan solo una hoja-resumen con parte de los contenidos de la asignatura. No viene explicada la teoría, tan solo es un esquema para que puedas apoyarte a la hora de hacer los ejercicios.

El concepto de recursividad es bastante complejo y gran parte de los problemas que suponen los algoritmos recursivos es el hecho, ya no solo entender el concepto, si no que además entender cómo funcionan las “llamadas recursivas”.

En primer lugar, debemos definir recursividad y para que se entienda claro vamos a hacerlo de la forma más directa y simple posible:

¿Qué pasaría si te dijera: “Recursividad es recursividad”?

Lo que ocurre es que no te he dicho nada, ¿o sí?

“Un plato es un plato, una taza es una taza” ¿Acaso tiene sentido decir eso?

No podemos definir una palabra con la misma palabra.

Pues, curiosamente, eso significa recursividad (*más o menos*).

“Recursividad es recursividad” está correcto porque exactamente ese concepto de definirse a si mismo es lo que significa recursividad.

Al principio es algo complicado, pero en la práctica es más fácil de ver:

Supongamos un método “R” que recibe por parámetros dos enteros ‘a’ y ‘b’ y lo que queremos hacer multiplica ‘a’ x ‘b’ de forma **recursiva**.

Bien pues la solución sería la siguiente:

```
public static int R(int a , int b){  
    if(b == 1){  
        return a;  
    } else {  
        return a+R(a,b-1);  
    }  
}
```

Analicemos el método por encima:

Método R: recibe un entero ‘a’, y un entero ‘b’ por parámetro y devuelve un entero.

Si ‘b’ es 1 entonces:

devolvemos el valor de ‘a’.

Si no (es decir, ‘b’ es distinto de 1) entonces:

devolvemos la ‘a’ + lo que devuelva la llamada al método R al que le pasamos ‘a’ y ‘b’-1 por parámetros.

Fin método.

¿Qué es lo que está ocurriendo aquí?

Estamos llamando al método “R” **DENTRO** del método “R”. Estamos llamando al mismo método estando en dicho método.

¿Qué implica esto? ¿Qué es lo que ocurre entonces?

Estamos aplicando el concepto de recursividad. Cada vez que se llame al método R se denomina “llamada recursiva” y, por tanto, el método “R”, es un método recursivo que implementa un algoritmo recursivo.

Por otro lado, si analizamos el programa, en el enunciado querían que multiplicásemos dos números, pero en el método no hay ninguna multiplicación. De hecho, lo que hay es una suma.

Lo que ocurre es que, al llamarse a si mismo, si queremos multiplicar, no podemos hacerlo directamente, debemos un buscar un proceso alternativo que permita hacer una operación consecutiva y que de lo mismo que la multiplicación.

Para ello, vamos a la definición de multiplicación, y podemos comprobar que la multiplicación es lo mismo que sumar sucesivamente el mismo número 'a', un número de 'b' veces.

Es decir: $a \times b = a + a + \dots(b) \dots a + a$.

Pongamos a prueba el programa:

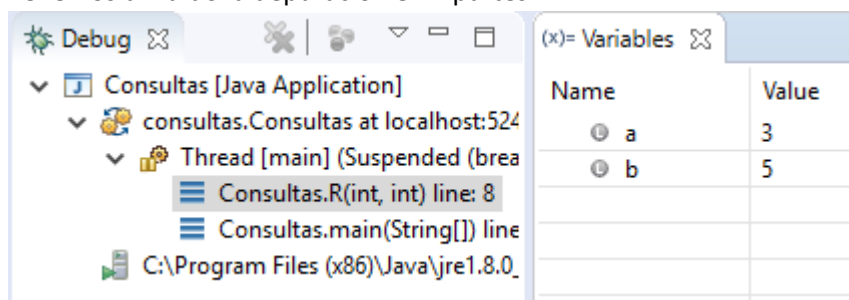
Supongamos que nos pasan $a = 5$ y $b = 3$. El resultado debería ser $5 \times 3 = 15$.

Para ello vamos a **depurar** el programa.

Declaramos el main que llame al método R y desarrollamos el método R.

```
3 public static void main(String[] arg){
4     System.out.println(R(3,5));
5 }
6
7 public static int R(int a , int b){
8     if(b == 1){
9         return a;
10    } else {
11        int j = a+R(a,b-1);
12        return j;
13    }
14 }
```

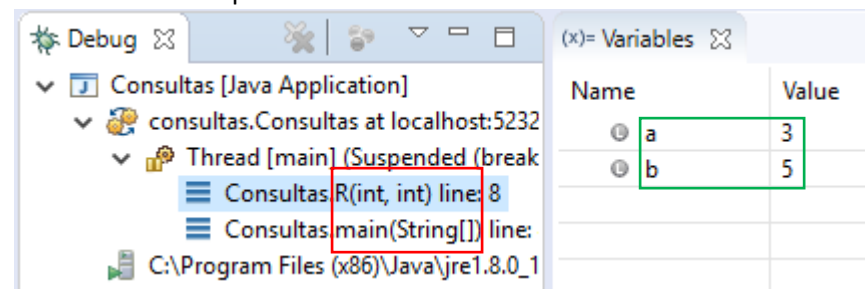
Tenemos dividido la depuración en 2 partes:



debug: Nos va a indicar el estado de la depuración. Lo importante que hay que hacer es fijarse en los métodos que se invocan.

Variables: Nos indicará las variables que se han declarado, inicializado y su valor.

Comenzamos la depuración:



Se han creado las variables $a = 3$, $b = 5$. Y si nos fijamos se ha llamado al método R (sombreado azul)

Avanzamos. Como b es distinto de 1, llama al método R y ocurre esto:

Debug Console:

- Consultas [Java Application]
- consultas.Consultas at localhost:524
 - Thread [main] (Suspended (breakpoint at line 11))
 - Consultas.R(int, int) line: 11
 - Consultas.R(int, int) line: 11
 - Consultas.main(String[]) line: 11

Variables:

Name	Value
a	3
b	4

¡Hay dos métodos R siendo invocados! Además, el valor de 'b' ha cambiado.

Sigamos:

Debug Console:

- Consultas [Java Application]
- consultas.Consultas at localhost:524
 - Thread [main] (Suspended (breakpoint at line 11))
 - Consultas.R(int, int) line: 11
 - Consultas.R(int, int) line: 11
 - Consultas.R(int, int) line: 11
 - Consultas.main(String[]) line: 11

Variables:

Name	Value
a	3
b	3

Como podemos observar, se ha invocado a otro método R a parte de los que ya estaban, ¿por qué?

Vamos directamente a cuando b = 1.

Debug Console:

- Consultas [Java Application]
- consultas.Consultas at localhost:5232
 - Thread [main] (Suspended (breakpoint at line 8))
 - Consultas.R(int, int) line: 8
 - Consultas.R(int, int) line: 11
 - Consultas.R(int, int) line: 11
 - Consultas.R(int, int) line: 11
 - Consultas.R(int, int) line: 11
 - Consultas.R(int, int) line: 11
 - Consultas.main(String[]) line: 11

Variables:

Name	Value
a	3
b	1

Se ha llamado 5 veces el método R.

Ahora, b = 1. Avanzamos una instrucción más y...

Debug Console:

- Consultas [Java Application]
- consultas.Consultas at localhost:524
 - Thread [main] (Suspended (breakpoint at line 12))
 - Consultas.R(int, int) line: 12
 - Consultas.R(int, int) line: 11
 - Consultas.R(int, int) line: 11
 - Consultas.R(int, int) line: 11
 - Consultas.R(int, int) line: 11
 - Consultas.main(String[]) line: 11

Variables:

Name	Value
a	3
b	2
j	6

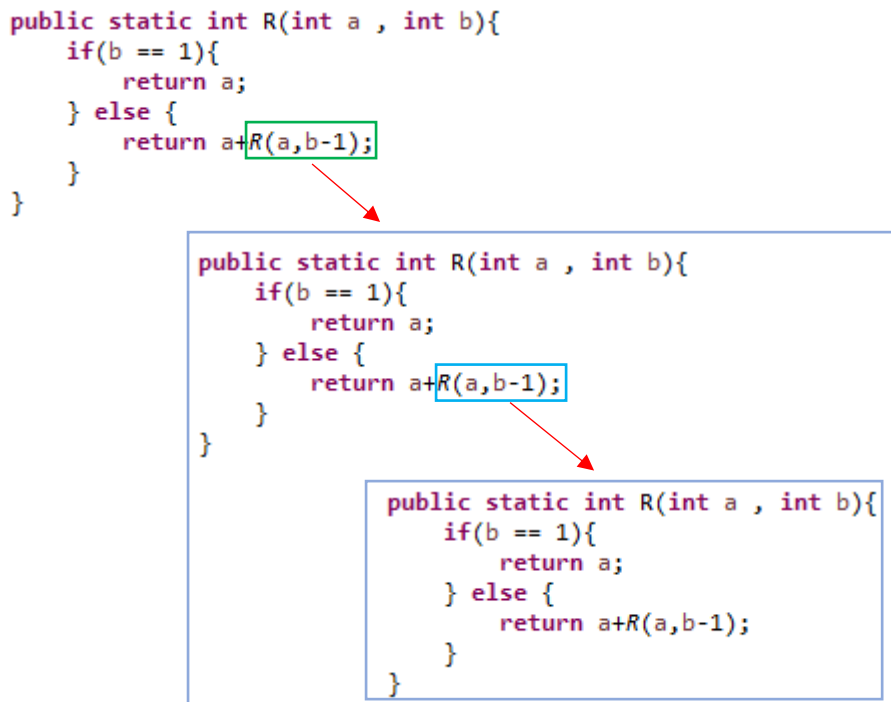
Aparece la variable j con valor 6. La 'b' vuelve a valer 2 y se ha eliminado un método de la lista.

Y si seguimos:

Three screenshots of a Java IDE showing the state of variables `a`, `b`, and `j` during recursive calls to method `R`. The first screenshot shows `a=3`, `b=3`, `j=9`. The second shows `a=3`, `b=4`, `j=12`. The third shows `a=3`, `b=5`, `j=15`. In each, `b` and `j` are highlighted with a blue box, and the current call to `R(int, int)` is highlighted with a red box.

Cuando 'b' vuelve a valer 5 se termina el programa y se devuelve el valor de 'j', que es 15.

Cada vez que llega a la parte de "`return a + R(a, b-1)`" se llama al método R. Esto significa:



Entonces para que la primera vez (señalado con verde) se devuelva `a + R(a, b-1)` primero debe resolverse la llamada (señalado con azul) y ésta a su vez, igual y así sucesivamente hasta.... ¿cuándo?

El problema en la recursividad es definir cuándo un método para de llamarse a sí mismo, puesto que, de no definir dicha condición de parada o de no hacer la llamada correctamente, puede caer en lo que se denomina “recursividad infinita”, en la que se acabarían llamando infinitos métodos sin que estos devuelvan nada.

Es por ello que aparece el concepto de “**caso base**”.

Si cogemos cualquier problema y lo queremos hacer mediante un algoritmo recursivo, primero debemos definir hasta cuando termina; es decir, cual es el **caso base**. Para ello, debemos pensar en cómo se resuelve el problema.

Vamos a resolver el siguiente problema mediante recursividad: el factorial de un número.

1º) Definimos el problema: sabemos por definición que, el factorial de un número, es la multiplicación sucesiva de todos los términos que hay entre 1 y dicho número incluido.

Ponemos un par de ejemplos:

$n = 3 \rightarrow 3! = 3 * 2 * 1$

$n = 5 \rightarrow 5! = 5 * 4 * 3 * 2 * 1$

El procedimiento a seguir sería algo tal que: $n! = n * (n-1) * (n-2) \dots 2 * 1$

2º) ¿Cuál sería el caso base? ¿Cómo se llega a él?

Para saberlo debemos plantearnos: ¿Cuándo termina el procedimiento? Cuando la n sea = 1 y esto ocurre porque la n va disminuyendo de 1 en 1; es decir, en cada iteración, se le resta 1 a la n .

Quizás alguno piense, “pero, si 1 por lo que sea es 1, ¿no sería el caso base 2?” pero entonces estamos obviando algo importante: ¿Cuál es el factorial de 1?

Se podría poner como excepción, pero podemos utilizarlo perfectamente de caso base.

3º) Desarrollamos el método.

Ya tenemos el caso base, y sabemos cómo llegar a él. Ahora vamos a plantear el método:

```
public static resultado método(){
    if(CasoBase()){
        return resultado();
    } else {
        return LlamadaRecursiva();
    }
}

public static int factorial(int n){
    if(n == 1){ //Caso base
        return 1;
    } else {
        return n * factorial(n-1); // llamada recursiva
    }
}
```

Aquí es donde se va a realizar la operación de $n * (n-1) * (n-2) \dots$

4º) Revisamos. Añadimos excepciones.

```
public static int factorial(int n){
    if(n < 0){ //excepcion
        return 0;
    }
    if(n == 1 || n == 0){ //Caso base + excepcion del factorial de 0
        return 1;
    } else {
        return n * factorial(n-1); // llamada recursiva
    }
}
```

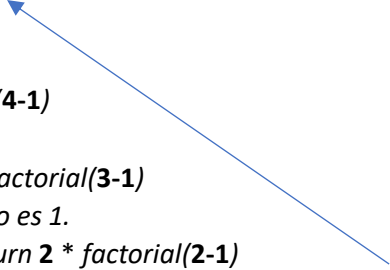
Ya hemos terminado.

Vamos a analizar lo que ocurre en el programa depurando a mano:

Supongamos que $n = 5$. Es decir, queremos hacer el $\text{factorial}(5) \Rightarrow 5! = 120$.

Lo que va a ocurrir es:

```
return 5 * factorial(5-1)
    4 no es 1.
    return 4 * factorial(4-1)
        3 no es 1.
        return 3 * factorial(3-1)
            2 no es 1.
            return 2 * factorial(2-1)
                1 si es 1 -> return 1. -> Una vez hemos llegado al caso base, resolvemos.
```



Una forma más cómoda de verlo:

$5! = 5 * 4!$

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1!$

$1! = 1 \rightarrow 2! = 2 * 1 = 2 \rightarrow 3! = 3 * 2 = 6 \rightarrow 4! = 4 * 6 = 24 \rightarrow 5! = 5 * 24 = 120$

Realmente para hallar el factorial de 5 hemos tenido que hallar primero el factorial de 4, de 3, de 2 y de 1.

Los algoritmos recursivos son bastante potentes y cómodos para resolver problemas, pero consumen mucho y tardan más que utilizando bucles.

Por regla general, casi todo lo que se puede hacer con recursividad, se puede hacer con bucles, aunque estos serían muy complejos de implementar.

Si no te ha quedado como funciona la recursividad, te recomiendo que realices varios ejercicios y sobre todo fijándote en las llamadas.