Recursividad en Java

La recursividad es una técnica potente de programación que puede utilizarse en lugar de la iteración para resolver determinados tipos de problemas. Una función recursiva es aquella que se llama a sí misma.

Por ejemplo, para escribir un método que calcule el factorial de un número entero no negativo, podemos hacerlo a partir de la definición de factorial:

```
Si n = 0 entonces
0! = 1
si n>0 entonces
n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1
```

Esto dará lugar a una solución iterativa en Java mediante un bucle for:

```
// Método Java no recursivo para calcular el factorial de un número
public double factorial(int n){
    double fact=1;
    int i;
    if (n==0){
        fact=1;
    }else{
        for(i=1;i<=n;i++){
            fact=fact*i;
        }
    }
    return fact;
}
```

Pero existe otra definición de factorial en función de sí misma:

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

Esta definición da lugar a una solución recursiva del factorial en Java:

```
// Método Java recursivo para calcular el factorial de un número
public double factorial(int n){
    if (n==0){
        return 1;
    }
    else
    {
        return n*(factorial(n-1));
    }
}
```

Un método es recursivo cuando entre sus instrucciones se encuentra una llamada a sí mismo.

La solución iterativa es fácil de entender. Utiliza una variable para acumular los productos y obtener la solución. En la solución recursiva se realizan llamadas al propio método con valores de n cada vez más pequeños para resolver el problema.

Cada vez que se produce una nueva llamada al método se crean en memoria de nuevo las variables y comienza la ejecución del nuevo método.

Para entender el funcionamiento de la recursividad, podemos pensar que cada llamada supone hacerlo a un método diferente, copia del original, que se ejecuta y devuelve el resultado a quien lo llamó.

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 se llama sí mismo

Caso base: Siempre ha de existir uno o más casos en los que los valores de los parámetros de entrada permitan al método devolver un resultado directo. Estos casos también se conocen como **solución trivial** del problema.

En el ejemplo del factorial el caso base es la condición:

```
if (n==0)
     return 1;
```

si n=0 el resultado directo es 1 No se produce llamada recursiva

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 del factorial en cada llamada recursiva se utiliza n-1

```
return n * ( factorial(n-1) );
```

por lo que en cada llamada el valor de n se acerca más a 0 que es el caso base.

La recursividad es especialmente apropiada cuando el problema a resolver (por ejemplo, calculo del factorial de un número) o la estructura de datos a procesar (por ejemplo los árboles) tienen una clara definición recursiva.
No se debe utilizar la recursión cuando la iteración ofrece una solución obvia. Cuando el problema se pueda definir mejor de una forma recursiva que iterativa lo resolveremos utilizando recursividad.
Para medir la eficacia de un algoritmo recursivo se tienen en cuenta tres factores:

- Tiempo de ejecución
- Uso de memoria
- Legibilidad y facilidad de comprensión

Las soluciones recursivas suelen ser más lentas que las iterativas por el tiempo empleado en la gestión de las sucesivas llamadas a los métodos. Además, consumen más memoria ya que se deben guardar los contextos de ejecución de cada método que se llama.

A pesar de estos inconvenientes, en ciertos problemas, la recursividad conduce a soluciones que son mucho más fáciles de leer y comprender que su correspondiente solución iterativa. En estos casos una mayor claridad del algoritmo puede compensar el coste en tiempo y en ocupación de memoria.

De todas maneras, numerosos problemas son difíciles de resolver con soluciones iterativas, y sólo la solución recursiva conduce a la resolución del problema (por ejemplo, Torres de Hanoi o recorrido de Árboles).

Ejemplos de Recursividad.

1. Calcular 2 elevado a n

Programa Java recursivo para calcular 2 elevado a n siendo n un número entero mayor o igual que 0.

La solución recursiva se basa en lo siguiente:

Caso Base:

si n = 0 entonces 2^0 = 1

Si n > 0 Podemos calcular 2^n como 2 * 2^n-1

Por ejemplo: 2^5 = 2 * 2^4

El programa que calcula 2 elevado a n de forma recursiva puede quedar así:

```
//programa Java para calcular 2 elevado a n de forma recursiva
import java.util.Scanner;

public class Elevar2aNRecursivo {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        int num;

        do{
           System.out.print("Introduce un numero entero >=0 ");
           num = sc.nextInt();
        }while(num < 0);

        System.out.println("2 ^ " + num + " = " + potencia(num));
    }

    //método recursivo para elevar 2 a n
    public static double potencia(int n){
           if(n == 0){    //caso base
              return 1;
           } else {
              return 2 * potencia(n-1);
```

```
            }
    }
}
```

La solución iterativa a este problema sería esta:

```
//programa Java para calcular 2 elevado a n de forma iterativa
import java.util.Scanner;

public class Potencia2N {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        int num;

        do{
           System.out.print("Introduce un numero entero >=0 ");
           num = sc.nextInt();
        }while(num < 0);

        System.out.println("2 ^ " + num + " = " + potencia(num));
    }

    //método iterativo para elevar 2 a n
    public static double potencia(int n){
           double resultado = 1;
           for(int i = 1; i <= n; i++){
               resultado = resultado * 2;
           }
           return resultado;
    }
}
```