4.6. Recursividad

Una función puede ser invocada desde cualquier lugar: desde el programa principal, desde otra función e incluso desde dentro de su propio cuerpo de instrucciones. En este último caso, cuando una función se invoca a sí misma, diremos que es una **función recursiva.**

```
static int funcionRecursiva() {
    ...
    funcionRecursiva(); //llamada recursiva
    ...
}
```

Este es el esquema general de una función recursiva. Si observamos con atención, se plantea un problema: dentro de `funciónRecursiva()` se invoca a `funciónRecursiva()`, donde a su vez, se volverá a llamar a `funciónRecursiva()`, y así sucesivamente. Esto nos lleva a un ciclo infinito de llamadas a la función. Para evitarlo, hemos de habilitar un mecanismo que detenga, en algún momento, la serie de llamadas recursivas: una sentencia `if` que, utilizando una condición, llamada «caso base», impida que se continúe con una nueva llamada recursiva. Veamos el esquema general:

```
int funcionRecursiva(datos) {
    int resultado;
```

```
    if (caso base) {
       resultado = valorBase;
    } else {
       resultado = funcionRecursiva(nuevosDatos); //llamada recursiva
       ...
    }
    return (resultado);
}
```

Solo cuando la condición del caso base sea `false`, se hará una nueva llamada **recursiva.** Cuando el caso base sea `true` se romperá la cadena de llamadas. La idea principal de la recursividad es solucionar un problema reduciendo su tamaño. Este proceso continúa hasta que tenga un tamaño tan pequeño que su solución sea trivial.

Para conseguir problemas cada vez más pequeños, los datos de entrada deben tender hacia el caso base. Conceptualmente `nuevosDatos` deben ser de menor tamaño que `datos`; así garantizamos que en algún momento los datos utilizados en la función alcanzan el caso base, cortando la serie de llamadas recursivas.

Veamos un ejemplo. Supongamos que deseamos calcular el factorial de un número *n*, que se representan por *n*! Sabemos que:

$$n! = n \times (n-1) \times (n-2) \times \ldots 2 \times 1$$

Por ejemplo:

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Podemos calcular el factorial de cualquier número directamente realizando la multiplicación anterior mediante un bucle, pero existe una solución recursiva. La definición de factorial se puede escribir también del siguiente modo:

$$n! = n \times \underbrace{(n-1) \times (n-2) \ldots \times 2 \times 1}_{(n-1)!} \Rightarrow$$

$$n! = n \times (n-1)!$$

Se considera por definición que el factorial de cero vale uno. Para calcular el factorial de un número, estamos utilizando el factorial de un número más pequeño, con lo cual estamos reduciendo el problema. Hemos de buscar un caso base, es decir, un valor para el que calcular el factorial sea algo trivial y no necesitemos volver a utilizar el método recursivo.

El caso base del factorial es: 0! = 1.

En cada llamada, los datos de entrada van siendo menores y tienden hacia el caso base: para calcular el factorial de *n*, utilizamos el factorial de (*n* – 1) que, a su vez, usará el factorial de (*n* – 2), y así, sucesivamente, hasta llegar a 0, cuyo factorial vale 1. Este será el caso base.

Con toda la información de la que disponemos podemos escribir una función que calcule el factorial de un número de forma recursiva:

```
long factorial(int n) {
   long resultado;
   if (n == 0) { //si n es 0
```

```
        resultado = 1; //caso base
    } else {
        resultado = n * factorial(n - 1); //llamada recursiva
    }
    return(resultado);
}
```

Recuerda

El tipo `long`, que es el tipo primitivo con mayor capacidad para guardar enteros en Java, lo usaremos porque el resultado del factorial suele ser un número muy grande. A modo de ejemplo, el factorial de 10 es 3628800.

Hagamos una traza —ejecución paso a paso de las instrucciones de un programa— de la función `factorial(3)`:

```
long factorial(3) {
    long resultado;
    if (3 == 0) { //falso
        ...
    } else {
        resultado = 3 * factorial(2);
```

La ejecución de `factorial(3)` queda a la espera de que se ejecute `factorial(2)`.

En este instante existen dos funciones `factorial()` en memoria. Veamos qué ocurre en la llamada a `factorial(2)`:

```
long factorial(2) {
    long resultado;
    if (2 == 0) { //falso
        ...
    } else {
        resultado = 2 * factorial(1);
```

Ahora también `factorial(2)` se queda esperando a la ejecución de `factorial(1)`. En este momento existen en memoria las funciones `factorial(3)` y `factorial(2)` esperando a que termine la ejecución de `factorial(1)`. La nueva llamada se ejecuta del siguiente modo:

```
long factorial(1) {
    long resultado;
    if (1 == 0) { //falso
        ...
    } else {
        resultado = 1 * factorial(0);
```

De forma análoga a las anteriores, se detiene la ejecución de la llamada a la función `factorial(1)` para que comience a ejecutarse una nueva instancia de la función recursiva; es el último caso, `factorial(0)`. En este momento de la ejecución, están a la espera de que finalicen las respectivas llamadas recursivas varias instancias, o copias, de la función `factorial()`. Veamos cómo se ejecuta `factorial(0)`:

```
long factorial(0) {
   long resultado;
   if (0 == 0) { //cierto
      resultado = 1;
   } else {
      ...
   }
   return(1);
}
```

La última instancia de la función termina de ejecutarse, devolviendo el valor 1, y permitiendo que la llamada anterior (`factorial(1)`) prosiga su ejecución.

```
long factorial(1) {
      ...
      resultado = 1 * 1; //1
   }
   return(1);
}
```

De nuevo la función que se ejecuta actualmente, `factorial(1)`, termina, devolviendo el valor 1 y permitiendo que la instancia de la función que esperaba su finalización continúe.

Desde el punto en que se quedó esperando, el `factorial(2)` prosigue así:

```
long factorial(2) {
      ...
      resultado = 2 * 1; //2
   }
   return(2);
}
```

Termina devolviendo el control para que siga su ejecución `factorial(3)`:

```
long factorial(3) {
      ...
      resultado = 3 * 2; //6
   }
   return(6);
}
```

Finaliza la primera instancia de la función que se invocó, devolviendo el control al programa principal o donde se llamase. Una posible llamada para la traza anterior sería:

```
long solucion = factorial(3);
System.out.println(solucion); //muestra 6
```