2.3 Funciones

1. Introducción

Una subrutina o subprograma, como idea general, se presenta como un subalgoritmo que forma parte del algoritmo principal, el cual permite resolver una tarea específica.

Podemos distinguir tres términos que poseen diferencias:

1. **Función**: conjunto de instrucciones que devuelven un resultado.
2. **Procedimiento**: conjunto de instrucciones que se ejecutan sin devolver ningún resultado.
3. **Método**: función o procedimiento que pertenece a un objeto.

Dichas funciones se pueden utilizar desde muchos sitios diferentes, por lo que de manera general, no se suelen poner mensajes en consola en las funciones ya que puede ser que dichos mensajes no interesen en todos los sitios donde se utilice dicha función. A no ser que la función se haya creado específicamente para dar mensajes informativos en consola.

2. Construcción

Una función se construye de la siguiente manera:

```
modificador_acceso tipo_resultado nombre_función (tipo_parámetro
nombre_parámetro, ... ) {
   instrucciones
   return expresión;
}
```

- **modificador_acceso**: es la visibilidad que posee la función (Lo veremos más adelante. De momento, lo utilizaremos como public).
- **tipo_resultado**: es el tipo del resultado que devuelve la función.
- **nombre_función**: es el nombre que identifica a la función. Utiliza la notación lowerCamelCase. Ejemplo: imprimirResultadoDecimal.
- **tipo_parámetro nombre_parámetro, ...**: puede ocurrir que la función necesite ciertos valores para efectuar la misión para la que ha sido creada. Por ejemplo, la función suma necesitaría los valores que tiene que sumar. En este caso, se deben indicar cada uno de dichos valores con sus tipos correspondientes. A estos valores se les conoce como parámetros de la función. Si la función no necesita parámetros, entonces solamente se ponen los paréntesis: `nombre_funcion( )`.

A todo esto se le conoce como la **firma** (signature) de la función:

```
modificador_acceso tipo_resultado nombre_función (tipo_parámetro
nombre_parámetro, ... )
```

- **instrucciones**: instrucciones que conforman el algoritmo de la función, para que realice la misión para la que ha sido creada.
- **return expresión** : con el return se termina la ejecución de la función y si va acompañado de una expresión, la función devuelve como resultado el valor de dicha expresión. Dicho valor tiene que ser del *tipo_resultado* indicado en la firma de la función.

Ejemplo de función con dos parámetros:

```
public static int add(int sum1,int sum2) {
    return sum1+sum2;
}
```

Puede ser también que haya más de un return. En ese caso, el flujo de ejecución abandona la función en cuanto ejecute el primer return. Ejemplo:

```
public static boolean isPair(int n){
    if(n%2==0){
        return true;
    }
    else{
        return false;
    }
}
```

En el caso de que estemos definiendo un **procedimiento**, no tendremos `return expresión` ya que no devuelve ningún resultado y el tipo_resultado es *void*. Como por ejemplo `System.out.println`, que escribe en pantalla lo que recibe por parámetro pero no devuelve nada.

3. Llamada a la función

Una función permite que reutilicemos un algoritmo ya que se puede utilizar cuando nos haga falta. Para ello, solamente tendremos que llamar a la función por su nombre y pasarle los parámetros en el mismo orden que se han definido y pertenecientes al mismo tipo de dato o compatible. En la llamada, dichos parámetros se llaman **argumentos**, es decir, los argumentos son los valores iniciales de los parámetros.

```
package tema2_3_Funciones;

public class Functions1 {

    public static void main(String[] args) {

        boolean pair;
        int result;

        /*
         * Se llama a la función isPair con un valor de 5 en el argumento, es
decir,
         * el valor inicial del parámetro n es 5:
         */
        pair = isPair(5);
        System.out.println(pair);//Mostrará false
```

```
        /*
         * Se llama a la función isPair con un valor de 4 en el argumento, es
decir,
         * ahora el valor inicial del parámetro n es 4:
         */
        pair = isPair(4);
        System.out.println(pair);//Mostrará true

        /*
         * Se llama a la función add con los valores 5 y 2 en los argumentos,
es decir,
         * los valores iniciales de los parámetros sum1 y sum2 son 5 y 2
respectivamente:
         */
        result = add(5, 2);
        System.out.println(result);//Mostrará 7

    }

    public static boolean isPair(int n) {

        if (n % 2 == 0) {
            return true;
        } else {
            return false;
        }

    }

    public static int add(int sum1, int sum2) {

        return sum1 + sum2;
    }

}
```

4. Ámbito de vida de los parámetros

A nivel de visibilidad y de ámbito de vida, los parámetros funcionan como las variables locales (*Ver tema 1.3 Variables y constantes 4. Ámbito de vida de las variables*), por lo tanto el ámbito de vida de los parámetros es el bloque donde han sido definidos, es decir, la propia función. Cada vez que se llame a la función, los parámetros nacen, se ejecuta la función y una vez que la función ha terminado de ejecutarse, los parámetros mueren.

```
package tema2_3_Funciones;

public class Functions2 {

    public static void main(String[] args) {

        boolean pair;
        int result;

        pair = isPair(5);//Nace el parámetro n con el valor 5
        //Aquí n ya no existe porque la función isPair ya ha terminado de
ejecutarse
        System.out.println(pair);
        pair = isPair(4);//Vuelve a nacer n pero esta vez con un valor de 4
        //Aquí n ya no existe porque la función isPair ya ha terminado de
ejecutarse
        System.out.println(pair);
```

```
        result = add(5, 2);//Nacen los parámetros sum1 y sum2 con los valores
5 y 2 respectivamente
        //Aquí sum1 y sum2 ya no existen porque la función add ha terminado de
ejecutarse
        System.out.println(result);

    }

    public static boolean isPair(int n) {//Comienzo del ámbito de vida del
parámetro n

        if (n % 2 == 0) {
            return true;
        } else {
            return false;
        }

    }//Fin del ámbito de vida del parámetro n

    public static int add(int sum1, int sum2) {//Comienzo del ámbito de vida
de los parámetros sum1 y sum2

        return sum1 + sum2;
    }//Fin del ámbito de vida de los parámetros sum1 y sum2

}
```

5. Ejemplo de función: el factorial de un número

El factorial de un entero positivo *n*, también indicado como **n!**, se define como el producto de todos los números enteros positivos desde 1 hasta *n*. Por ejemplo:

5 ! = 1 x 2 x 3 x 4 x 5 = 120

La operación de factorial aparece en muchas áreas de las matemáticas, particularmente en combinatoria y análisis matemático. De manera fundamental, el factorial de *n* representa el número de formas distintas de ordenar *n* objetos distintos (elementos sin repetición). Este hecho ha sido conocido desde hace varios siglos, en el siglo XII, por los hindúes.

Veamos cómo se programaría dicha función factorial y las llamadas con distintos tipos de argumentos:

```
package tema2_3_Funciones;

import java.util.Scanner;

public class Factorial1 {

    @SuppressWarnings("resource")
    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);
        int n, variable;

        System.out.println("Llamada a la función con argumentos literales: ");
        System.out.printf("El factorial de 5 es %d\n", factorial(5));

        System.out.println("Llamada a la función usando una variable como
argumento: ");
        variable = 5;
```

```
        System.out.printf("El factorial de %d es %d\n", variable,
factorial(variable));

        System.out.println("Llamada a la función usando una expresión como
argumento: ");
        variable = 3;
        System.out.printf("El factorial de %d es %d\n", variable + 2,
factorial(variable + 2));

        System.out.println("Llamada a la función con argumentos introducidos
por el usuario: ");
        do {
            System.out.println("Introduzca un número entero positivo: ");
            n = keyboard.nextInt();
        } while (n <= 0);

        System.out.printf("El factorial de %d es %d", n, factorial(n));

    }

    public static int factorial(int n) {

        int result = 1;

        for (int i = 2; i <= n; i++) {
            result *= i;
        }

        return result;
    }

}
```

El concepto del factorial se aplica a los números enteros positivos, pero como los *int* admiten números negativos, se podría llamar a la función con un número negativo aunque no tenga sentido:

```
package tema2_3_Funciones;

public class Factorial2 {

    public static void main(String[] args) {

        int result;

        result = factorial(-5);//No tiene mucho sentido porque el factorial se
aplica a números positivos
        System.out.printf("El factorial de %d es %d\n", -5, result);

    }

    public static int factorial(int n) {

        int result = 1;

        for (int i = 2; i <= n; i++) {
            result *= i;
        }

        return result;
    }

}
```

Salida por consola:

```
El factorial de -5 es 1
```

Es decir, cuando programamos una función, no podemos dar por hecho que el programador que la vaya a utilizar lo haga de manera adecuada con la lógica que representa su funcionalidad. Así que siempre que programemos una función, debemos asegurarnos que va a funcionar correctamente para todos los valores posibles del parámetro. En nuestro caso, no podemos dar un resultado coherente para los números negativos puesto que no tiene sentido matemáticamente el factorial de un número negativo, así que lo más conveniente es lanzar un error cuando llamen a la función con un número negativo.

```java
package tema2_3_Funciones;

public class Factorial3 {

    public static void main(String[] args) {

        int result;

        result = factorial(-5);//Al que utilice la función con números
negativos le salta un error
        System.out.printf("El factorial de %d es %d\n", -5, result);

    }

    public static int factorial(int n) {

        int result = 1;

        if (n < 0) {
            throw new IllegalArgumentException("El factorial se aplica a
números positivos");
        }

        for (int i = 2; i <= n; i++) {
            result *= i;
        }

        return result;
    }

}
```

Salida por consola:

```
Exception in thread "main" java.lang.IllegalArgumentException: El factorial se
aplica a números positivos
    at tema2_3_Funciones.Factorial3.factorial(Factorial3.java:19)
    at tema2_3_Funciones.Factorial3.main(Factorial3.java:9)
```

6. Ejemplo de procedimiento

No tendremos `return expresión` ya que no devuelve ningún resultado y el tipo_resultado es *void*:

```java
package tema2_3_Funciones;

import java.util.Scanner;
```

```java
import static tema1_12_EscrituraEnPantalla.colores.Colors.*;

public class Procedure {

    @SuppressWarnings("resource")
    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);
        String string;

        System.out.print("Introduce una cadena: ");
        string = keyboard.nextLine();
        paintGreen(string);

    }

    public static void paintGreen(String string) {

        System.out.printf("La cadena que has introducido en verde: %s", GREEN
+ string + RESET);

    }
}
```

7. Resultado de las funciones

En las llamadas a funciones, no hay que obligatoriamente utilizar el valor devuelto:

```java
package tema2_3_Funciones;

import java.util.Scanner;
import static tema1_12_EscrituraEnPantalla.colores.Colors.*;

public class Result2 {

    @SuppressWarnings("resource")
    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);
        String string, stringGreen;

        System.out.print("Introduce una cadena: ");
        string = keyboard.nextLine();
        /*
         * En la llamada a la función turnGreen, no estamos utilizando el
valor devuelto:
         */
        turnGreen(string);
        /*
         * En la siguiente llamada sí lo vamos a utilizar:
         */
        stringGreen = turnGreen(string);
        System.out.printf("La cadena %s convertida a verde: %s", string,
stringGreen);

    }

    public static String turnGreen(String string) {

        String result = String.format("%s", GREEN + string + RESET);

        System.out.println(result);
```

```
        return result;

    }
}
```

8. Recursividad

La recursividad es una técnica de escritura de funciones pensada para problemas complejos. La idea parte de que una función pueda invocarse a sí misma.

Esta técnica es peligrosa ya que se pueden generar fácilmente llamadas infinitas (la función se llama a sí misma, tras la llamada se vuelve a llamar a sí misma, y así sucesivamente sin freno ni control). Por lo tanto, es muy importante tener en cuenta cuándo la función debe dejar de llamarse.

Hay que ser muy cauteloso cuando se utiliza la recursividad, pero permite soluciones muy originales y abre la posibilidad de solucionar problemas muy complejos.

Veamos como ejemplo la versión recursiva del factorial:

```java
package tema2_3_Funciones;

import java.util.Scanner;

public class RecursiveFactorial1 {

    @SuppressWarnings("resource")
    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);
        int n;

        do {
            System.out.println("Introduzca un número entero positivo: ");
            n = keyboard.nextInt();
        } while (n <= 0);

        System.out.printf("El factorial de %d es %d", n, factorial(n));

    }

    public static int factorial(int n) {

        int result;

        if (n == 1) { //Caso base: devuelve un 1
            result = 1;
        } else { //Caso recursivo
            result = n * factorial(n - 1);
        }

        return result;
    }

}
```

Para entenderlo mejor, vamos a imprimir los resultados intermedios añadiendo la siguiente línea de código en la función recursiva *factorial*: `System.out.printf("Factorial de %d Resultado: %d%n",n,result);`

```java
package tema2_3_Funciones;

import java.util.Scanner;

public class RecursiveFactorial2 {

    @SuppressWarnings("resource")
    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);
        int n;

        do {
            System.out.println("Introduzca un número entero positivo: ");
            n = keyboard.nextInt();
        } while (n <= 0);

        System.out.printf("El factorial de %d es %d", n, factorial(n));

    }

    public static int factorial(int n) {

        int result;

        if (n == 1) { //Caso base: devuelve un 1
            result = 1;
        } else { //Caso recursivo
            result = n * factorial(n - 1);
        }

        System.out.printf("Factorial de %d  Resultado: %d%n", n, result);

        return result;
    }

}
```

¿recursividad o iteración? Hay otra versión del factorial resuelto mediante un bucle for (solución iterativa) en lugar de utilizar la recursividad. La cuestión es ¿cuál es mejor? Ambas implican sentencias repetitivas hasta llegar a una determinada condición, por lo que ambas pueden generar programas que no finalizan si la condición nunca se cumple. En el caso de la iteración es una condición la que permite determinar el final, la recursividad lo que hace es ir simplificando el problema hasta generar una llamada a la función que devuelva un valor y no se vuelva a llamar. Para un ordenador es más costosa la recursividad ya que implica realizar muchas llamadas a funciones, es decir, es más rápida la solución iterativa. Entonces, ¿por qué elegir recursividad? La recursividad se utiliza sólo si:

- No encontramos la solución iterativa a un problema.
- El código es mucho más claro en su versión recursiva.

9. La pila

Una pila(***stack***) es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, el modo de acceso a sus elementos es de tipo LIFO (del inglés *Last In, First Out*, «último en entrar, primero en salir») . Esta estructura se aplica en multitud de supuestos en el área de informática debido a su simplicidad y capacidad de dar respuesta a numerosos procesos.

Para el manejo de los datos cuenta con dos operaciones básicas: apilar (***push***), que coloca un objeto en la pila, y su operación inversa, desapilar (***pop***), que retira el último elemento apilado.

En cada momento sólo se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado (denominado TOS, *Top of Stack*). La operación desapilar permite la obtención de este elemento, que es retirado de la pila permitiendo el acceso al anterior (apilado con anterioridad), que pasa a ser el último, el nuevo TOS.

Para las llamadas entre funciones, se utiliza una estructura de tipo pila: supongamos que se está procesando una función y en su interior llama a otra función. La función se abandona para procesar la función de la llamada, pero antes se almacena en una pila la dirección que apunta a la función. Ahora supongamos que esa nueva función llama a su vez a otra función. Igualmente, se almacena su dirección, se abandona y se atiende la petición. Así en tantos casos como existan peticiones. La ventaja de la pila es que no requiere definir ninguna estructura de control ni conocer las veces que el programa estará saltando entre funciones para después retomarlas, con la única limitación de la capacidad de almacenamiento de la pila. Conforme se van cerrando las funciones, se van rescatando las funciones precedentes mediante sus direcciones almacenadas en la pila y se va concluyendo su proceso, esto hasta llegar a la primera.

En el caso de una función recursiva, esto es posible implementarlo con sencillez mediante una pila. La función se llama a sí misma tantas veces como sea necesario hasta que el resultado de la función cumpla la condición de retorno; entonces, todas las funciones abiertas van completando su proceso en cascada. No se necesita saber cuandas veces se anidará y, por tanto, tampoco cuando se cumplirá la condición, con la única limitación de la capacidad de la pila. De sobrepasarse ese límite, normalmente porque se entra en un bucle sin final, se produce el «error de desbordamiento de la pila» (***stack overflow***).