UNIDAD 4: PROGRAMACIÓN ESTRUCTURADA Y MODULAR

Profesor: José Ramón Simó Martínez

Contenido

4.1 Introducción

Hasta ahora hemos escrito todo el código de nuestros programas dentro de un bloque único conocido como bloque principal o main:

```
public static void main(String[] args)
{
    // Nuestro código…
}
```

Sin embargo, estas son alguna de las consecuencias cuando aumenta la complejidad de nuestro programa:

- Código redundante: bloques de código repetidos.
- Algoritmos extensos: solo hay un algoritmo que puede ser difícil de entender por su extensión.
- Conflicto entre variables: puede haber un gran número de variables que no tienen relación entre ellas y podemos confundir la finalidad de unas con otras.

En esta unidad aprenderemos, en primer lugar, a crear y utilizar nuestras propias funciones. En segundo lugar, estudiaremos las funciones recursivas, las cuales nos permitirán escribir versiones más sencillas de algoritmos complejos. Finalmente, veremos las ventajas de la programación modular y el diseño descendente.

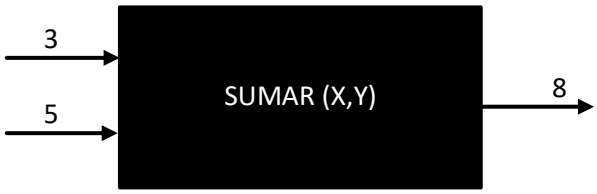
4.2 Concepto de función

En programación, una función es un bloque de código destinado a realizar una función específica dentro de nuestro programa. Una forma de ver una función es como si fuera una caja negra. Esta caja realiza una única tarea a partir de unos datos de entrada y, como resultado de procesar esos datos, puede devolver o no datos de salida.

El concepto de caja negra viene porque no nos importa como la función haga su tarea. Sólo nos importa que hace y qué necesita para hacerlo.

Por ejemplo, necesitamos una función que sume dos números x e y. Esta función se puede representar matemáticamente como: sumar(x, y) = x + y. Sin embargo, al usuario sólo debe preocuparle los datos que se piden de entrada, x e y, y no cómo hace la operación.

En formato de caja negra, la función sumar se representaría así:



Las ventajas del uso de funciones es nuestros programas serían las siguientes:

- Evitar repeticiones de código.
- Incrementar la legibilidad de nuestro programa.
- Dividir un problema complejo en otros más simples.
- Reducir la probabilidad de cometer errores.
- Facilidad en la modificación de nuestro programa.

En Java a las funciones se las llama métodos debido a que están asociada un objeto. Por tanto, el término "método" está asociado al paradigma de programación orientada a objetos que estudiaremos en las siguientes unidades.

4.3 Definición de una función

Al definir una función estamos construyendo un bloque de código que posteriormente podremos utilizar. La definición de una función tiene la siguiente sintaxis:

```
modificador tipoDatoDevuelto nombreDeFuncion (lista de parámetros de entrada)
{
    // Cuerpo de la función

    return datoDevuelto;
}
```

Veamos cada componente de la función:

- **modificador**: define ciertas características de la función. Por ahora escribiremos delante de cada función el modificador public static.
- **tipoDatoDevuelto**: el dato de salida de la función. Puede ser de tipo primitivo (int, float, etc) u objeto.
- **nombreDeFunción**: es el nombre identificador que utilizaremos para que la función se ejecute.

- **lista de parámetros de entrada**: es el conjunto de datos de entrada que tendrá la función. Los parámetros se definen con tipo de variable e identificador y separados por comas. Sin embargo, puede haber funciones donde no tengan parámetros de entrada.

- **return** datoDevuelto: la palabra reservada return indica que la variable o valor que se pone a continuación (datoDevuelto) es la salida de la función.

Veamos el siguiente ejemplo donde definimos una función que simplemente muestra un mensaje por pantalla:

```
public static void saludar()
{
    System.out.println("¿Quién es Niklaus Wirth?");
}
```

Podemos apreciar dos detalles característicos de esta función:

- La lista de parámetros está vacía.

- El tipo de dato se llama `void`.

- No utilizamos la palabra reservada `return`.

En primer lugar, al definir esta función sin parámetros estamos diciendo que no recibe datos de entrada.

En segundo lugar, el tipo de dato `void` es no lo hemos visto hasta ahora. Se utiliza en la gran mayoría de veces en la definición de las funciones. Sirve para indicar que la función no devuelve ningún resultado después de realizar terminar el bloque de instrucciones que contiene. En programación, a este tipo de funciones que no devuelven ningún dato de salida se les conoce como **subprogramas**.

Por último, al definir esta función como subprograma, no utilizaremos por tanto la palabra reservada `return`.

Otro ejemplo más completo sería definir una función que recibe dos números enteros, hace el algoritmo que calcula el cuál es mayor y por último devuelve el número mayor.

```
public static int mayor(int a, int b)
{
    if(a >= b)
        mayor = a;
    else
        mayor = b;

    return mayor;
}
```

Iremos viendo con más detalle los conceptos que hemos visto hasta ahora.

En el siguiente apartado aprenderemos a acceder al código fuente que se encuentra en el bloque de la función que hayamos creado.

4.4 Llamada a una función

Cuando definimos una función podemos entender que estamos etiquetando un bloque de código al cuál podremos acceder en cualquier momento para que se ejecute.

Para llamar a una función podemos hacerlo de dos formas:

- Si la función no devuelve ningún dato se ejecuta como una instrucción sin tener que almacenar el resultado de salida en ninguna variable, por ejemplo:

```
saludar();
```

- Si la función devuelve un dato, por ejemplo:

```
int resultado = mayor (3,4);
```

Observa que podemos hacer lo siguiente:

```
mayor(3, 4);
```

Esto lo que haría es llamar a la función, pero no guardaría en ningún sitio el resultado de vuelta. Hay veces que nos interesa recoger lo que devuelve la función y otras no, dependerá de la lógica del programa.

Por otra parte, también podemos mostrar directamente el valor devuelto por la función:

```
System.out.println(mayor(3, 4));
```

¿Y desde dónde se puede llamar a las funciones? Desde cualquier otra función de nuestro programa. Por ahora, haremos las llamadas desde la función que más conocemos: el main.

¿Cómo se comporta nuestro programa cuando se hace una llamada a una función? Cuando estamos ejecutando una serie de instrucciones en la función F1 y hacemos una llamada a la función F2, nuestro programa pasa a ejecutar el código que está en el cuerpo de F2. Al finalizar el bloque de código que contiene F2, el programa devuelve el control a F1 en el mismo punto donde se hizo la llamada a F2.

El siguiente esquema ilustra los descrito anteriormente e indicando el orden de ejecución de cada instrucción:

```
F1(){
  Instrucción1;
  Instrucción2;
  F2(); // Punto de llamada y regreso.
  Instrucción5;
}

F2(){
  Instrucción3;
  Instrucción4;
}
```

Un ejemplo completo donde podemos ejecutar la función saludar y mayor que hemos definido sería:

```
public class Ejemplo51
{
    public static void saludar()
    {
        System.out.println("¿Quién es Niklaus Wirth?");
    }

    public static int mayor(int a, int b)
    {
        int mayor;

        if (a > b)
            mayor = a;
        else
            mayor = b;

        return mayor;
    }

    public static void main(String[] args)
    {
        saludar();

        int resultado = mayor(3,4);

        System.out.println("El mayor de 3 y 4 es: " + resultado);
    }
}
```

4.5 Paso de parámetros

Ya hemos visto que a las funciones le podemos pasar una serie de parámetros que serán los datos de entrada a la función. Sin embargo, tenemos que tener presente que existen técnicamente dos formas de pasar parámetros:

- Paso por valor
- Paso por referencia

Es importante entender bien en qué se diferencia y por tanto vamos a verlo en detalle.

4.5.1 Por valor

La función hace una copia de los parámetros que le pasamos, es decir, no modifica los valores de las variables desde donde se llama a la función.

En el siguiente ejemplo podemos ver que a la función incrementar le pasamos un parámetro de tipo entero, pero los cambios realizados a ese parámetro dentro de la función incrementar no afectan a su valor en la función main. El ejemplo, además, ilustra el comportamiento de la llamada a una función:

```
public static void incrementar(int a)
{
    System.out.println("Estoy dentro función incrementar...");
    System.out.println("Valor de variable antes de a = a + 1: " + a);

    a = a + 1;

    System.out.println("Valor de variable después de a = a + 1: " + a);
    System.out.println("Termina función incrementar...");
}

public static void main(String[] args)
{
    int a = 5;

    System.out.println("Empieza función main...");
    System.out.println("Valor de variable a: " + a);
    System.out.println("Llamo a función incrementar desde main...");

    incrementar(a);

    System.out.println("Vuelvo a función main...");
    System.out.println("Valor de variable a: " + a);
    System.out.println("Termina función main...");
}
```

Salida:

```
Empieza función main...
Valor de variable a: 5
Llamo a función incrementar desde main...
Estoy dentro función incrementar...
Valor de variable antes de a = a + 1: 5
Valor de variable después de a = a + 1: 6
Termina función incrementar...
Vuelvo a función main...
Valor de variable a: 5
Termina función main...
```

Para obtener los cambios realizados en la función incrementar deberemos definirla para que devuelva un entero y recogeremos ese valor devuelto en la función main:

```
public static int incrementar(int a) // devuelve un entero
{
    System.out.println("Estoy dentro función incrementar...");
    System.out.println("Valor de variable antes de a = a + 1: " + a);

    a = a + 1;

    System.out.println("Valor de variable después de a = a + 1: " + a);
    System.out.println("Termina función incrementar...");

    return a; // devuelvo el resultado
}

public static void main(String[] args)
{
    int a = 5;

    System.out.println("Empieza función main...");
    System.out.println("Valor de variable a: " + a);
    System.out.println("Llamo a función incrementar desde main...");

    a = incrementar(a); // guardo el valor devuelto por la función incrementar

    System.out.println("Vuelvo a función main...");
    System.out.println("Valor de variable a: " + a);
    System.out.println("Termina función main...");
}
```

Salida:

```
Empieza función main...
Valor de variable a: 5
Llamo a función incrementar desde main...
Estoy dentro función incrementar...
Valor de variable antes de a = a + 1: 5
Valor de variable después de a = a + 1: 6
Termina función incrementar...
Vuelvo a función main...
Valor de variable a: 6
Termina función main...
```

4.5.2 Por referencia

Estudiaremos esta técnica en las unidad de estructuras estáticas (arrays).

4.6 Retorno de un valor

Como hemos visto hasta ahora, siempre que queramos devolver un dato desde una función utilizaremos la palabra reservada `return`.

Normalmente situaremos el return al final del bloque de la función, como hemos hecho en los ejemplos anteriores. Sin embargo, hay soluciones en las se puede utilizar el `return` en diversas partes del código para simplificar o por otros motivos. Por ejemplo:

```
public static int mayor (int a, int b)
{
    if (a > b)
        return a;

    return b;
}
```

Hay que tener en cuenta que cuando la función termina cuando ejecuta una sentencia `return`. En el anterior ejemplo si a es mayor que b, se ejecutará `return a` y terminará el bloque de la función mayor. Por tanto, no se ejecutará `return b`.

4.7 Ámbito de variables

Hay dos tipos de ámbitos de variables:

- **Global:** se declara fuera de cualquier función. El tiempo de vida de la variable afecta a todo el bloque de la clase, es decir, la variable puede ser usada en todas las funciones y bloques. La variable deja de existir al terminar el programa.
- **Local:** se define dentro de una función. El tiempo de vida de la variable afecta al bloque de la función. La variable deja de existir al terminar la función.

4.7.1 Ámbito global

Observemos el siguiente ejemplo donde hacemos uso de una variable global:

```
public class Test
{
    static int x = 5; // variable global static

    public static void incrementar()
    {
        x++;
    }

    public static void main(String[] args)
    {
        System.out.println(x);

        incrementar();

        System.out.println(x);
    }
}
```

Salida:

```
5
6
```

Podemos observar, por tanto, que la variable x siendo global se puede utilizar y modificar en cualquier función que esté dentro de la clase.

> ¡Atención!
>
> El uso de variables globales tipo `static` debe evitarse todo lo posible. Su uso debe estar restringido para programadores expertos en casos totalmente controlados. Estudiaremos mejor el modificador `static` en las unidades de programación orientada a objetos.

4.7.2 Ámbito local

En el siguiente ejemplo podemos observar que la variable `x` se declara de forma local dentro del bloque de la función `incrementar` y de la función `main`:

```java
public class Test
{
    public static void incrementar()
    {
        int x = 5; // variable local a la función incrementar
        System.out.println("Valor de x en incrementar: " + (x + 1));

    } // Se destruye variable x de incrementar

    public static void main(String[] args)
    {
        int x = 3; // variable local a la función main
        incrementar();
        System.out.println("Valor de x en main: " + x);

    } // Se destruye variable x de main
}
```

Salida:

```
Valor de x en incrementar: 6

Valor de x en main: 3
```

Asimismo, podemos declarar el mismo nombre de variable en distintas funciones ya que Java trata estas variables como locales y por tanto no tienen relación entre sí.

4.7.3 Conflicto entre variables

Podríamos pensar que si declaramos una variable global ya no podremos declarar una variable local con el mismo nombre identificador. Sin embargo, esto se puede hacer y en Java la variable local tiene prioridad sobre la global. Esto es, si en una función declaramos una variable local con el mismo nombre que la global, entonces la variable global ya no tiene validez dentro de esa función.

Veamos el siguiente ejemplo para entenderlo mejor:

```java
public class Test
{
    static int x = 5; // variable global

    public static void test1()
    {
        int x = 3;
        System.out.println("Valor de x en test1: " + x); // local
    }

    public static void test2()
    {
        System.out.println("Valor de x en test2: " + x); // es la global
    }

    public static void main(String[] args)
    {
        int x = 7;

        test1();
        test2();

        System.out.println("Valor de x en main: " + x); // local
    }
}
```

Salida:

```
Valor de x en test1: 3
Valor de x en test2: 5
Valor de x en main: 7
```

En `test1()` al declarar `int x = 3` la variable x ya no hará referencia a la global (x = 5) sino a la local (x = 3). Lo mismo para la función `main`. En cambio, en `test2()` hace uso de la variable global al no declararse una variable con el mismo nombre que esta.

4.8 La función main

Ya sabemos que la función main (o método main en Java) es la función principal o punto de entrada de nuestro programa. Es decir, todo programa empezará en la función main. En este apartado vamos a analizar otros aspectos importantes de esta función, como el paso de parámetros por consola.

La función main siempre se declara con este encabezado:

```java
public static void main (String[] args)
```

> **Nota**
>
> Ahora ya sabes que la función main no devuelve nada. En otros lenguajes, como C o C++, la función main sí que devuelve un entero.

4.8.1 Parámetros de entrada a la función main

> **Nota**
>
> En este apartado hacemos una breve aproximación al concepto de array. Por tanto, no hace falta entender completamente el concepto de array, simplemente seguir los pasos que se indican en este apartado para obtener parámetros de entrada a la función main.

Analizando el encabezado de la función main, también podemos entender ahora que tiene un parámetro de entrada:

```
String[] args
```

También sabemos que este parámetro es un array de tipo String, es decir, que en cada elemento del array almacenará una cadena de texto de tipo String.

Sin embargo, ¿para qué sirve este parámetro? Vamos a verlo.

Es muy frecuente que un programa llamado desde la línea de comandos (la consola de Windows o Linux) tenga ciertas opciones que le indicamos como argumentos.

Por ejemplo, bajo Linux podemos ver la lista detallada de ficheros que terminan en .java haciendo:

```
ls -l *.java
```

En este caso, la orden sería `ls` y las dos opciones (o parámetros) que le indicamos son `-l` y `*.java`

La orden equivalente en la consola de comandos de Windows sería:

```
dir *.java
```

Pues bien, estas opciones que se les pasa al programa en línea de comandos se pueden leer desde Java. Hay que tener en cuenta que `dir` o `ls` sería el programa propiamente.

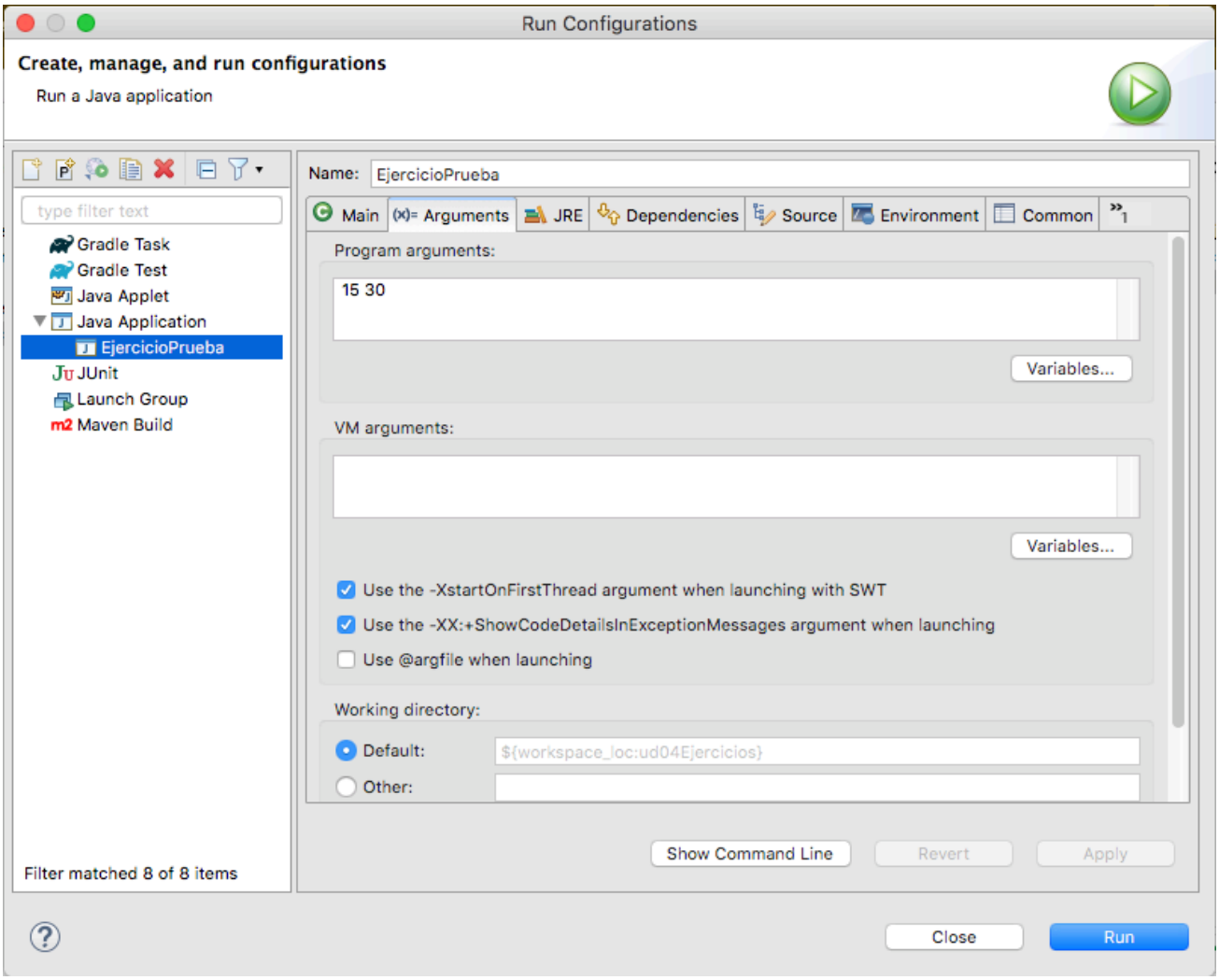
Ahora ya podemos imaginar que la utilidad del parámetro String[] args será almacenar estas opciones que se les pasa al programa ya que son cadenas de texto. Veamos un ejemplo práctico de cómo funciona.

```
public class LineaComandos
{
    public static void main(String[] args)
    {
        System.out.println("Parámetro 1: " + args[0]);
        System.out.println("Parámetro 2: " + args[1]);
    }
}
```

Si ejecutamos el programa LineaComandos desde consola y queremos pasar dos parámetros, por ejemplo, “uno” y “dos”, sería así:

```
java LineaComandos uno dos
```

Si lo queremos pasar los parámetros desde un IDE, como por ejemplo Eclipse, deberemos hacer click derecho sobre el fichero *.java* que queramos ejecutar → Run as → Run Configurations → Pestaña "(x)=Arguments" → Escribir los parámentros, separados por espacions, en el campo de texto "Program arguments". Por ejemplo, si le quiero pasar los valores 15 y 30 a "EjercicioPrueba.java", quedaría así en el Run Configurations:



Salida:

Parámetro 1: uno

Parámetro 2: dos

4.9 Recursividad

La recursividad es una técnica que permite solucionar un problema a partir de casos más simples del mismo problema. ¿Cómo se aplica esta técnica a las funciones? Llamando una función a sí misma:

```
public static void funcion1()
{
    // Instrucciones...
    funcion1();
    // Instrucciones....
}
```

En el anterior ejemplo podemos ver, en forma de código, que una de las instrucciones que tiene la `funcion1()` es llamarse a sí misma.

Ahora nos podemos preguntar que si una función se llama a sí misma, ¿cuándo va a parar?

En este punto debemos tener en cuenta dos conceptos importantes de la recursión:

- **Caso recursivo:** es una versión más sencilla (o subproblema) que la del problema original. También se conoce como la llamada recursiva.
- **Caso base:** es el caso más simple del problema original. También se conoce como Condición de Parada.

> Nota
>
> Para afrontar una solución recursiva siempre debemos pensar en la versión recursiva del problema y cuál es su caso base. Además, debemos tener en cuenta que en cada llamada recursiva el problema se vuelve más simple, hasta llegar a la versión más simple de todas que es el caso base.

4.9.1 Ejemplo del factorial de un número

En matemáticas, un problema típico que se puede afrontar recursivamente es el factorial de un número. Por ejemplo, el factorial del número 4 sería:

$$4! = 4\ x\ 3\ x\ 2\ x\ 1 = 24$$

Por tanto, podemos definir el factorial de un número es el resultado de multiplicar ese número por los que le siguen hasta llegar a 1.

El problema original sería calcular el factorial de cualquier número, que llamamos n:

$$n! = n\ x\ (n-1)\ x\ (n-2)\ x \ldots x\ 3\ x\ 2\ x\ 1$$

Ahora podemos pensar en una versión más simple de $n!$, que sería $(n-1)!$

$$(n-1)! = (n-1)\ x\ (n-2)\ x \ldots x\ 3\ x\ 2\ x\ 1$$

Finalmente, debemos encontrar el caso más simple. Este caso puede ser cuando n = 1, ya que por definición el factorial de 1 es 1:

$$1! = 1$$

Por tanto, ya tenemos los dos componentes esenciales para afrontar recursivamente el factorial de un número:

- **Caso recursivo:** $n! = n\ x\ (n-1)!$ para n > 1
- **Caso base:** $1!$

> Nota
>
> También podríamos pensar que el caso base es 0, ya que el el factorial de 0! = 1

Ahora ya podemos programar el caso factorial con una función recursiva de la siguiente forma:

```
public class FactorialRecursivo
{
    public static int factorial(int n)
    {

        if (n == 1) // Caso base (en este caso también puede ser 0)
            return 1;
        else
            return n * factorial(n-1); // Caso recursivo
    }

    public static void main(String[] args)
    {
        int n = 4;
        int resultado;

        resultado = factorial(4);
        System.out.println("Factorial de " + n + " = " + resultado);
    }
}
```