UD04 – FUNCIONES

Contenido

1 Nociones de diseño modular de programas

Igual que con la inmensa mayoría de tareas con cierto grado de complejidad en el mundo real, la creación de un programa requiere un paso previo donde hay que reflexionar sobre qué es exactamente lo que desea hacer y cómo se va a lograr el objetivo. Es muy poco recomendable afrontar esta tarea sentándose directamente delante del ordenador, abriendo el entorno de trabajo y comenzando a escribir líneas de código. Cuando nos enfrentamos a un problema nuevo es imprescindible una fase de análisis del problema.

Desgraciadamente, la capacidad de los humanos para captar problemas complejos es limitada, ya que, en general, sólo somos capaces de mantener una visión simultánea de unos pocos elementos. Por lo tanto, cuando el proceso a realizar es largo o se basa en la manipulación de muchos elementos diferentes, es muy fácil, no sólo equivocarse, sino también no saber por dónde empezar.

Por lo tanto, resultará útil disponer de alguna estrategia que permita hacer frente a la resolución de problemas con diferentes grados de complejidad. Una de las más populares en todos los campos es considerar que un problema complejo en realidad no es más que la agregación de un conjunto de problemas más simples, cada uno de ellos más fáciles de resolver. Por tanto, si somos capaces de entender y resolver estos problemas simples, también seremos capaces de resolver el problema complejo.

En consecuencia, el primer paso para poder llevar a cabo una tarea compleja será encontrar como descomponerla en otros más simples, que entonces se irán resolviendo uno a uno.

Hay dos estrategias básicas para resolver la descomposición de un problema: el diseño descendente y el ascendente. Este apartado se centra en la primera de ellas.

El **diseño descendente** (*top-down*, en inglés) es la técnica que se basa en partir de un problema general y dividirlo en problemas más simples, denominados subproblemas. De entre todos estos, los considerados todavía demasiado complejos se vuelven a dividir en nuevos subproblemas. Se denomina descendente porque partiendo del problema grande se pasa a problemas más pequeños a los que dará solución individualmente.

El número de niveles a los que hay que llegar dependerá de la complejidad del problema general. Para problemas no demasiado complejos, bastará con uno o dos niveles, pero para resolver problemas muy complejos puede ser necesario un gran número de sucesivas descomposiciones. También vale la pena destacar que, aunque es recomendable que la complejidad de los subproblemas de un mismo nivel sea aproximadamente equivalente, puede ser que algunos problemas queden resueltos completamente en menos niveles que en otros.

Los objetivos finales de aplicar esta estrategia son:

- Establecer una relación sencilla entre problemas planteados y el conjunto de tareas a realizar para resolverlos.
- Establecer más fácilmente los pasos para resolver un problema.
- Hacer más fácil de entender estos pasos.
- Limitar los efectos de la interdependencia que un conjunto de pasos tiene sobre otro conjunto.

La descomposición mediante diseño descendente permite hacer uso de una característica muy útil cuando se usa para diseñar algoritmos. Se trata de la posibilidad de buscar subproblemas idénticos, o al menos muy similares, y reaprovechar su solución en más de un lugar dentro del problema general. Una vez se han resuelto una vez, no tendría sentido volver a resolver de nuevo repetidas veces. Sobre esta circunstancia, por el momento se estudiará sólo el caso de subproblemas exactamente iguales.

En la descomposición de un problema, es especialmente acertado intentar hacerlo de manera que se fuerce la aparición de subproblemas repetidos, y así su resolución se puede reaprovechar en varios lugares.

Un aspecto que se debe tener en cuenta al aplicar diseño descendente es que se trata de una estrategia basada en unas directrices generales para atacar problemas complejos, pero no es ningún esquema determinista que le garantice que siempre obtendrá la mejor solución. Esto significa que, partiendo de un mismo problema, diferentes personas pueden llegar a conclusiones diferentes sobre cómo llevar a cabo la descomposición. De entre todas las soluciones diferentes posibles, algunas pueden considerarse mejores que otras. De hecho, nada impide, a partir de una solución

concreta, aplicar refinamientos que la mejoren. Por lo tanto, es interesante poder evaluar si la descomposición que ha hecho va por buen camino o no.

Algunos de los criterios en que te puedes basar para hacer esta evaluación son los siguientes:

- Si un problema que parece *a priori* bastante complejo se descompone en muy pocos niveles, tal vez valga la pena echar un segundo vistazo. Inversamente, si un problema no demasiado complejo tiene demasiados niveles, tal vez se haya ido demasiado lejos en la descomposición.
- Ver si el número de pasos incluidos en cada uno de los subproblemas no es excesivamente grande y es fácil de seguir y entender. De lo contrario, puede que todavía haría falta aplicar nuevos niveles de descomposición.
- Repasar los nombres asignados a los subproblemas sean autoexplicativos y expresen claramente la tarea que están resolviendo.
- Si absolutamente ninguno de los subproblemas es reutilizado en ninguna parte, especialmente en descomposiciones en muchos niveles, es muy posible que no se haya elegido correctamente la manera de descomponer algunos subproblemas.
- Vinculado al punto anterior, la aparición de subproblemas muy similares o idénticos, pero tratados por separado en diferentes lugares, también suele ser indicio de que no se está aplicando la capacidad de reutilizar subproblemas correctamente.

Una vez se dispone de un marco de referencia general sobre cómo aplicar diseño descendente, es el momento de aplicar la misma técnica para la creación de un programa.

Veamos un pequeño ejemplo: imaginemos que tenemos que desarrollar un programa que calcule el área o el volumen de diferentes figuras geométricas. En un análisis rápido, nuestro programa deberá ofrecer al usuario las distintas figuras geométricas que el programa es capaz de calcular, por ejemplo: cuadrado, rectángulo, círculo, cubo, esfera, cono, etc. Para las figuras planas se podría calcular la longitud del perímetro y el área, para las figuras en tres dimensiones área y volumen. La resolución de nuestro programa sería la suma de la resolución de cada figura geométrica que vayamos a incluir en el programa, es decir la agregación de la resolución de distintos problemas más pequeños. Si observamos las ecuaciones de algunas de las figuras: área cuadrado = $lado^2$, área círculo = $\pi * r^2$, volumen cubo = $lado^3$, volumen esfera = $\pi*r^3*4/3$, vemos que desarrollar una función que devuelva el resultado de elevar un número a otro será de gran ayuda en nuestros cálculos. Esta función sería utilizada para resolver varios de los objetos geométricos objeto de nuestro programa. También podemos concluir que el cálculo del área de un cubo no es más que la suma del área de cada una de las seis caras del cubo, es decir de cada uno de los seis cuadrados. Por

lo tanto, si sabemos cómo resolver el área del cuadrado ya sabemos cómo resolver el área del cubo.

2 Subprogramas

Es evidente que, si esta metodología nos lleva a tratar con **subproblemas**, entonces también tengamos la necesidad de poder crear y trabajar con **subprogramas** para resolverlos. A estos subprogramas se les suele llamar módulos o subrutinas, de ahí viene el nombre de **programación modular**. En general se suele hablar de dos tipos de módulos: los **procedimientos** y las **funciones**; un procedimiento es un subprograma que no devuelve ningún resultado y hablamos de función cuando el subprograma devuelve un resultado. La diferencia entre ambos es muy pequeña, de hecho, en OOP hablamos genéricamente de **métodos** sin distinguir si son funciones o procedimientos, es decir si devuelven o no resultado.

En general dentro de los lenguajes de programación, se llama una **función** a un conjunto de instrucciones con un objetivo común que se declaran de manera explícitamente diferenciada dentro del código fuente mediante una etiqueta o identificador.

Como hemos hablado, en la descomposición de problemas debemos buscar subproblemas repetidos, esto se trasladará en que nuestro programa tendrá un subprograma que se utilizará varias veces en la ejecución de nuestro programa. Dicho de otro método nuestro programa tendrá un método al que se llamará varias veces. Aparece aquí el concepto de **reutilización de código**. En el programa del punto anterior la función para elevar un número a otro sería un ejemplo de reutilización. También podemos poner el ejemplo del método *println* de la clase *System.out*, en los programas que hemos hecho hasta ahora hemos llamado a este método innumerables veces.

Aquí vemos también dos tipos de subprogramas o funciones; por un lado, las **funciones del sistema**, aquellas que nos vienen dadas por el lenguaje, en nuestro caso Java. Por el otro, **funciones definidas por el usuario**, serán aquellas que desarrollaremos nosotros para resolver un problema concreto y que utilizaremos posteriormente en nuestros programas.

3 Definición, declaración y llamada a funciones

Veamos a continuación la sintaxis de la declaración más básica de un método en Java. Vemos que su formato es muy similar a como se declara el método *main*.

```
void idenficadorMetodo(){

     instrucciones del método…
     return;
}
```

Los identificadores de los métodos siguen las mismas convenciones de código que las variables (lowerCamelCase) y podemos darle el valor que queramos evitando las palabras reservadas. Es aconsejable usar algún texto que sea auto-explicativo.

La declaración del función se puede llevar a cabo en cualquier lugar del archivo de código fuente, siempre que sea entre las claves que identifican el inicio y fin de archivo *(public class MiClase { … })* y fuera del bloque de instrucciones método *main*, o cualquier otro método. Normalmente, se suele hacer inmediatamente después del método principal (main).

La palabra *void* indica que el método no devuelve ningún valor.

Para invocar el método, basta con referirnos a su identificador desde otra parte del programa añadiendo paréntesis al final del identificador. Nos referimos a esto normalmente como llamada al método.

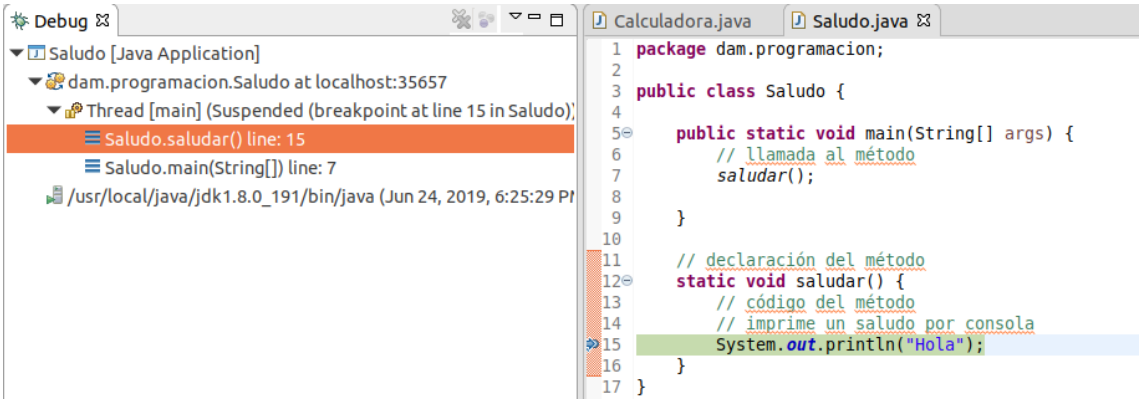
```
idenficadorMetodo();
```

Veamos un ejemplo; en la clase “Saludo” creamos el método “saludar()” al que podemos llamar siempre que queramos ejecutar las acciones que contiene, en este caso mostrar un saludo por consola

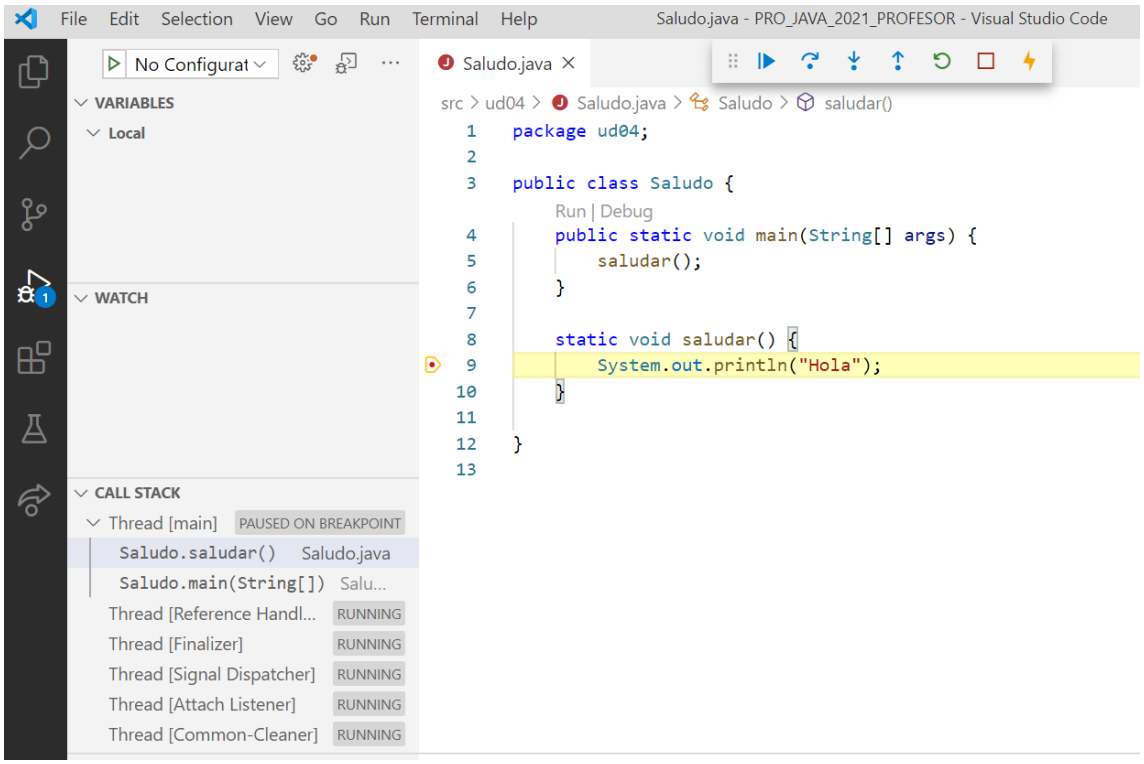
```
public class Saludo {

	public static void main(String[] args) {
		// llamada al método
		saludar();

	}

	// declaración del método
	static void saludar() {
		// código del método
		// imprime un saludo por consola
		System.out.println("Hola");
	}
}
```

Observemos que ocurre con la pila del programa al llamar al método *saludar()* desde el método *main*; se interrumpe la ejecución del método *main* en la línea 7 y se transfiere la ejecución del programa al método *saludar()*.

En Eclipse:



En VS Code:


File Edit Selection View Go Run Terminal Help
Saludo.java - PRO_JAVA_2021_PROFESOR - Visual Studio Code
No Configurat
VARIABLES
Local
WATCH
CALL STACK
Thread [main] PAUSED ON BREAKPOINT
Saludo.saludar() Saludo.java
Saludo.main(String[]) Salu...
Thread [Reference Handl... RUNNING
Thread [Finalizer] RUNNING
Thread [Signal Dispatcher] RUNNING
Thread [Attach Listener] RUNNING
Thread [Common-Cleaner] RUNNING
Saludo.java
src > ud04 > Saludo.java > Saludo > saludar()
1 package ud04;
2
3 public class Saludo {
Run | Debug
4 public static void main(String[] args) {
5 saludar();
6 }
7
8 static void saludar() {
9 System.out.println("Hola");
10 }
11
12 }
13

3.1 Ámbito de variables

Hemos visto que una variable puede usarse desde que se declara hasta que termina el bloque (}) en el que se ha declarado. En los programas que hemos creado hasta ahora hemos declarado las variables en el método *main* y dentro de los bloques de las estructuras de control. En la función que acabamos de crear no podemos acceder a las variables declaradas en el método *main*. De igual manera las variables que declaremos dentro de nuestros métodos sólo serán visibles dentro de estos.

Si necesitamos que una variable sea compartida por todos los métodos dentro de nuestra clase podemos declararla a nivel de clase (entre las llaves de principio y fin de clase y fuera de cualquiera de los métodos de la clase.

Modificamos el ejemplo anterior, incluiremos una variable *nombre* accesible a todos los métodos de la clase:

```
import java.util.Scanner;

public class Saludo {

	// variable global que puede ser accedida
	// desde cualquier método de la clase
	static String nombre;

	public static void main(String[] args) {
		// inicializar objeto scanner para la entrada de datos
		Scanner sc = new Scanner(System.in);

		// damos valor a la variable nombre desde la consola
		System.out.println("Introduzca su nombre");
		nombre = sc.nextLine();

		// llamada al método
		saludar();

		// cerrar objeto Scanner
		sc.close();

	}

	// declaración del método
	static void saludar() {
		// código del método
		// imprime un saludo por consola
		System.out.println("Hola " + nombre);
	}
}
```

Una **variable global** es una variable que puede ser accedida desde cualquier instrucción dentro de un mismo archivo de código fuente. Su ámbito es todo el archivo. A las variables declaradas dentro de un método se les denomina **variables locales**.

4 Parámetros

El uso de variables globales debe restringirse en la medida de lo posible, ya que al poder ser modificadas desde cualquier parte del programa va a ser más difícil predecir su comportamiento. Una manera de que una función trabaje con unos determinados valores es pasarle estos valores como parámetros. La sintaxis es la siguiente

```
void idenficadorMetodo([tipo1 idParam1][, tipo2 idParam2]…){

      instrucciones del método…
      return;
}
```

De manera que dentro del método podemos acceder a los valores pasados por parámetro utilizando sus identificadores tal y como hacemos con las variables locales o globales.

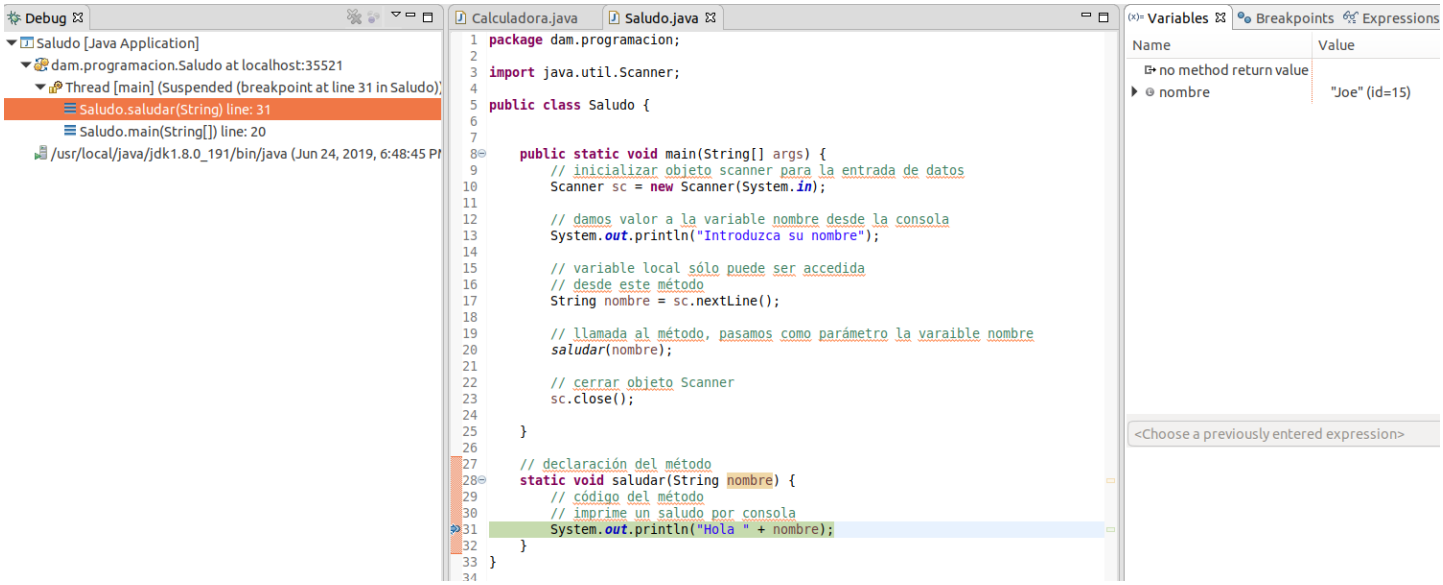
Rehacemos el programa anterior eliminado la variable global y pasando el nombre como parámetro:

```
import java.util.Scanner;

public class Saludo {


	public static void main(String[] args) {
		// inicializar objeto scanner para la entrada de datos
		Scanner sc = new Scanner(System.in);

		// damos valor a la variable nombre desde la consola
		System.out.println("Introduzca su nombre");

		// variable local sólo puede ser accedida
		// desde este método
		String nombre = sc.nextLine();

		// llamada al método, pasamos como parámetro la variable nombre
		saludar(nombre);

		// cerrar objeto Scanner
		sc.close();

	}

	// declaración del método
	static void saludar(String nombre) {
		// código del método
		// imprime un saludo por consola
		System.out.println("Hola " + nombre);
	}
}
```

De esta manera eliminamos el uso de una variable global y tenemos un mayor control sobre los valores de la variable nombre.

Si nos fijamos en la pila del programa vemos que se ha pasado el valor de la variable "*nombre*" del método *main* al parámetro "*nombre*" del método "*saludar()*". En este

caso el nombre de variable y parámetro coinciden (*nombre*), pero esto no tiene por qué ser así, lo que debe coincidir es el tipo de ambos. Lo que está ocurriendo es que al transferir la ejecución del método *main* al método saludar se hace una copia del valor de la variable nombre y se asigna dicho valor al parámetro nombre del método saludar. Cuando se accede al identificador nombre en el método saludar se obtiene el valor copiado y se puede hacer uso del él.



5 Valor de retorno

Podemos construir nuestras funciones de manera que devuelvan un valor asociado a los parámetros de entrada, de manera análoga a la definición de una función en matemáticas.
A unos valores de entrada les corresponde un valor de salida.

Si pensamos en cualquier función matemática resulta directa la comparación:
f(x) = 3x + 1. Trasladamos esta función a código Java:

```
double tresEquisMasUno(double x){

      return 3.0*x+1.0;
}
```

Podemos asignar el valor devuelto por la función a una variable para después utilizarlo:

```
Double resultado = tresEquisMasUno(5.0);
```

También podemos incluir la llamada a un método o una función dentro de una expresión.

```
System.out.println(tresEquisMasUno(5.0));
```

Modificamos el programa de ejemplo con el que vamos trabajando a lo largo de la unidad para que el método devuelve un mensaje en lugar de imprimirlo y movemos el código que lo imprime el método principal:

```
import java.util.Scanner;

public class Saludo {


	public static void main(String[] args) {
		// inicializar objeto scanner para la entrada de datos
		Scanner sc = new Scanner(System.in);

		// damos valor a la variable nombre desde la consola
		System.out.println("Introduzca su nombre");

		// variable local sólo puede ser accedida
		// desde este método
		String nombre = sc.nextLine();

		// llamada al método, pasamos como parámetro la variable nombre
		// salida del mensaje por consola
		System.out.println(saludar(nombre));

		// cerrar objeto Scanner
		sc.close();

	}

	// declaración del método
	static String saludar(String nombre) {
		// código del método
		// devuelve el mensaje del saludo
		return "Hola " + nombre;
	}
}
```

6 Sobrecarga de métodos

En Java podemos definir el mismo método con diferentes parámetros, de manera que para la misma acción podemos tener varias maneras de hacer.

Modificamos el programa que vamos utilizando de ejemplo para crear distintos mensajes de saludo:

```
import java.util.Scanner;

public class Saludo {

	public static void main(String[] args) {
		// inicializar objeto scanner para la entrada de datos
		Scanner sc = new Scanner(System.in);

		// damos valor a la variable nombre desde la consola
		System.out.println("Introduzca su nombre");

		// variable local sólo puede ser accedida
		// desde este método
		String nombre = sc.nextLine();

		// llamada al método, pasamos como parámetro la variable nombre
		// salida del mensaje por consola
		System.out.println(saludar(nombre));
		System.out.println("Presione cualquier tecla");
		sc.nextLine();
		System.out.println(saludar("Adiós", nombre));

		// cerrar objeto Scanner
		sc.close();

	}

	// declaración del método
	static String saludar(String nombre) {
		// código del método
		// devuelve el mensaje del saludo
		return "Hola " + nombre;
	}

	// sobrecarga del método para modificar saludo
	static String saludar(String saludo, String nombre) {
		// código del método
		// devuelve el mensaje del saludo
		return saludo + " " + nombre;
	}
}
```

Aquí aparece un concepto nuevo que es el de ***signatura de un método***, que vendrá determinada por *el orden y los tipos de los parámetros recibidos además del nombre de la clase y nombre el método*. No forman parte de la signatura del método el valor devuelto, las excepciones o los nombres de los parámetros.

A este mecanismo que permite distintas signaturas del mismo método se le denomina **sobrecarga** de métodos.

7 Recursividad

La **recursividad** es una forma de describir un proceso para resolver un problema de manera que, a lo largo de esta descripción, se usa el proceso mismo que se está describiendo, pero aplicado a un caso más simple.

Un **método recursivo** es aquel que, dentro de su bloque de instrucciones, tiene alguna invocación a él mismo.

El bloque de código de un método recursivo siempre se basa en una estructura de selección múltiple, donde cada rama es de alguno de los dos casos posibles descritos a continuación.

Por un lado, en el **caso base**, que contiene un bloque instrucciones dentro de las cuales no hay ninguna llamada al método mismo. Se ejecuta cuando se considera que, a partir de los parámetros de entrada, el problema ya es suficientemente simple como para ser resuelto directamente.

Por otro lado, está el **caso recursivo**, que contiene un bloque de instrucciones dentro de las cuales hay una llamada al método mismo, dado que se considera que aún no se puede resolver el problema fácilmente. Ahora bien, valores usados como parámetros de esta nueva llamada deben ser diferentes a los originales. Concretamente, serán unos valores que tiendan a acercarse al caso base.

Dentro de la estructura de selección siempre debe haber al menos un caso base y uno recursivo. Normalmente, los algoritmos recursivos más sencillos tienen uno de cada. Es imprescindible que los casos recursivos siempre garanticen que sucesivas llamadas van aproximando los valores de los parámetros de entrada a algún caso base, ya que, de lo contrario, el programa nunca termina y se produce el mismo efecto que un bucle infinito.

En Java, en caso de un bucle infinito en hacer llamadas recursivas se produce un error de *Stack Overflow* ("desbordamiento de pila", en inglés).

Ejemplo: Cálculo de producto por sumas sucesivas, implementación iterativa

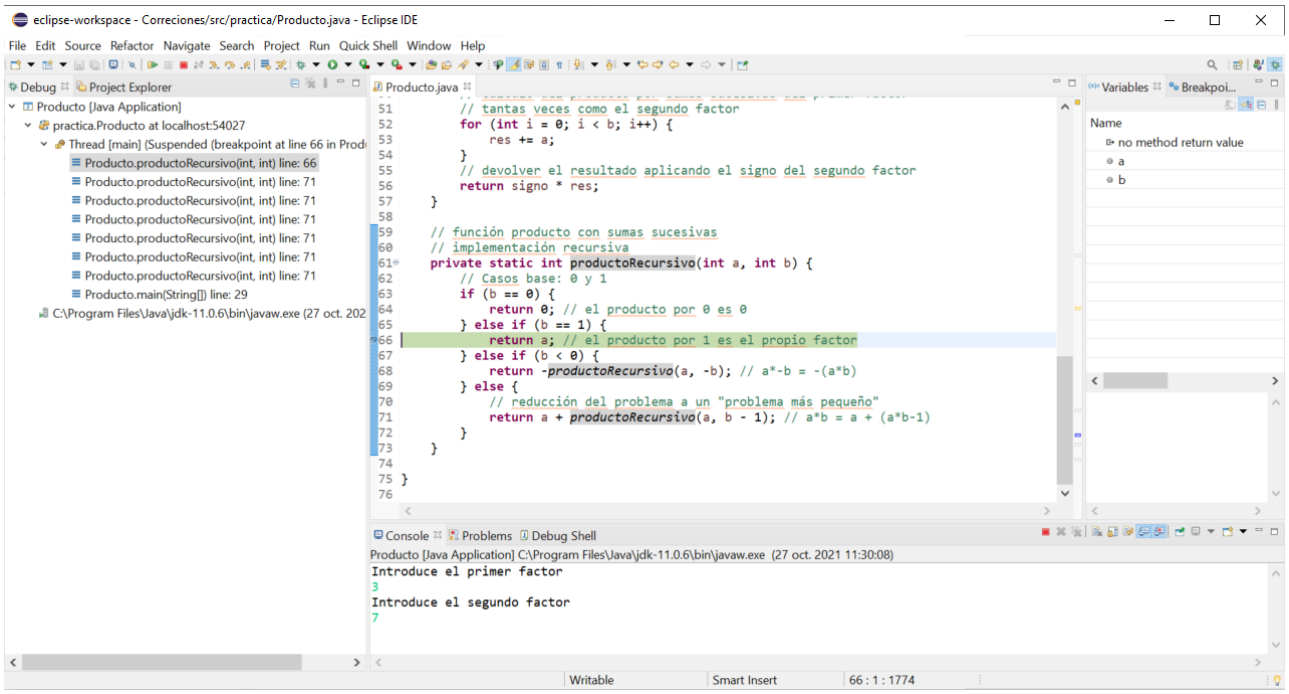
```
public class Producto {
    public static void main(String[] args) {
        // preparar la entrada de datos
        Scanner sc = new Scanner(System.in);

        // asegurar entrada correcta de números
        System.out.println("Introduce el primer factor");
        while (!sc.hasNextInt()) {
            sc.nextLine();
            System.out.println("Introduce el primer factor");
        }
        int i = sc.nextInt();
        sc.nextLine();

        System.out.println("Introduce el segundo factor");
        while (!sc.hasNextInt()) {
            sc.nextLine();
            System.out.println("Introduce el segundo factor");
        }
        int j = sc.nextInt();
        sc.nextLine();

        // llamar a la función producto para obtener el resultado
        int prod = productoIterativo(i,j);
        //mostrar el resultado
        System.out.println(i + " * " + j + " = " + prod);

        // cerrar scanner
        sc.close();
    }

    // función producto con sumas sucesivas
    // implementación iterativa
    private static int productoIterativo(int a, int b) {
        // variable para el resultado, inicializada en 0
        int res = 0;

        // obtenemos el signo para calcular con segundo factor negativo
        int signo = (b>=0)?1:-1;
        // cambiar el signo del segundo factor en caso de que sea negativo
        if(signo==-1){
            b = -b;
        }
        // cálculo del producto por sumas sucesivas del primer factor
        // tantas veces como el segundo factor
        for (int i = 0; i < b; i++) {
            res += a;
        }
        // devolver el resultado aplicando el signo del segundo factor
        return signo * res;
    }
}
```

Veamos ahora cómo implementar la misma función de manera recursiva:

```
// función producto con sumas sucesivas
// implementación recursiva
private static int productoRecursivo(int a, int b){
    // Casos base: 0 y 1
    if (b==0){
            return 0; // el producto por 0 es 0
    } else if(b == 1){
        return a; // el producto por 1 es el propio factor
    } else if (b<0){
        return - productoRecursivo(a, -b); // a*-b = -(a*b)
    } else {
        // reducción del problema a un "problema más pequeño"
        // a * b = a + (a * b-1)
        // 3 * 2 = 3 + 3 * 1
        return a + productoRecursivo(a, b-1);
    }
}
```

En Java que un método se llame a sí mismo no tiene nada de particular, desde el punto de vista de la pila de programa vemos que se trata de manera igual a una llamada a cualquier otro método.

Si observamos la pila de llamadas, vemos como la función se llama a sí misma y como va creciendo esta con cada llamada recursiva:



Podemos entender fácilmente que si se entra en un bucle infinito la pila del programa va a crecer tanto que ocupará toda la memoria disponible y entonces se produce el error de desbordamiento de pila que hemos visto.

8 Paso de parámetros por valor y por referencia

¿Qué ocurre si dentro de un método cambiamos los valores de los parámetros de entrada? Veamos el siguiente ejemplo:

```
public static void main(String[] args) {
      int i = 5;
      hacerDoble(i);
      System.out.println(i);
}

void hacerDoble(int i){
      i=i*2;
}
```

La salida de este programa será:

```
5
```

Esto es debido a que en Java el paso de parámetros se hacer por valor, es decir al hacer la llamada a un método y dar valor a sus parámetros lo que está ocurriendo es que se hace una copia de los valores que se pasan y se utilizan estas copias dentro del método, así que dentro de éste podemos hacer lo que necesitemos, modificarlo o usarlo, y no tendrá ninguna repercusión en la variable que se utilizó como parámetro al llamarlo.

En otros lenguajes existe el paso por referencia, que consiste en pasar el puntero a la dirección de memoria que contiene la variable por lo que los cambios dentro de los métodos también tienen efecto fuera de los mismos.

Profundizaremos en estos conceptos a lo largo del curso.