

PROGRAMACIÓN

Desarrollo de Aplicaciones Multiplataforma

Manual del módulo

UD 4 – FUNCIONES



Ander Gonzalez Ortiz



Esta obra está bajo una Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional.

3.1. INTRODUCCIÓN

Conforme aumenta la extensión y la complejidad de un programa, es habitual tener que implementar, en distintas partes, la misma funcionalidad, cosa que implica copiar una y otra vez, donde sea necesario, el mismo fragmento de código. Esto genera dos problemas:

- **Duplicidad del código:** aumenta el tamaño del código y lo hace menos legible.
- **Dificultad en el mantenimiento:** cualquier modificación necesaria dentro del fragmento de código repetido tendría que realizarse en todos y cada uno de los lugares donde se encuentra.

Podríamos pensar en utilizar un bucle, pero si el código se repite en lugares separados del programa, esto no es posible.

4.1. CONCEPTOS BÁSICOS

La solución para cuando necesitamos la misma funcionalidad en distintos lugares de nuestro código no es más que etiquetar con un nombre un fragmento de código y sustituir en el programa dicho fragmento, en todos los lugares donde aparezca, por el nombre que le hemos asignado. Esta idea puede verse en el siguiente ejemplo:

```
public static void main(String[] args) {  
    ... // código  
    System.out.println("Voy a saludar tres veces: "); //fragmento repetido  
    for (int i = 0; i < 3; i++) {  
        System.out.println("Hola.");  
    }  
    ... // más código  
    System.out.println("Voy a saludar tres veces:"); //fragmento repetido  
    for (int i = 0; i < 3; i++) {  
        System.out.println("Hola.");  
    }  
    ... // otro código  
    System.out.println("voy a saludar tres veces: "); //fragmento repetido  
    for (int i = 0 ; i < 3; i++) {  
        System.out.println("Hola. ");  
    }  
    ... // resto del código  
}
```

Cada fragmento de código repetido (en rojo) a lo largo del programa, con la misma funcionalidad en nuestro ejemplo, mostrar una serie de mensajes por consola, puede sustituirse por el nombre que le hemos asignado, en nuestro caso `tresSaludos()`, quedando:

```
public static void main(String[] args) {  
    ... //código  
    tresSaludos(); //sustitución por una función  
    ... //más código  
    tresSaludos(); //sustituciónpor una función
```

```

... //otro código
tresSaludos(); //sustitución por una función
... //resto del código
}

```

La función `tresSaludos()` tendrá que definirse, de forma que se especifique el conjunto de instrucciones que la forma.

```

public static void main(String[] args) {
    ...
}
static void tresSaludos() {
    System.out.println("Voy a saludar tres veces: ");
    for (int i = 0; i < 3; i++) {
        System.out.println("Hola.");
    }
}

```

La definición de una función puede hacerse antes o después del `main()`.

Esto representa el concepto de función: un conjunto de instrucciones agrupadas bajo un nombre y con un objetivo común, que se ejecuta al ser invocada.

En general, la sintaxis para definir una función es:

```

static tipo nombreFunción() {
    cuerpo de la función
}

```

Por ahora, definiremos las funciones `static` y utilizaremos como tipo la función `void`, que indica que la función no devuelve nada. Habitualmente, los nombres de funciones siguen el estilo Camel: los nombres comienzan en minúscula, distinguiendo en los nombres compuestos cada palabra mediante la mayúscula inicial, lo que recuerda las jorobas de un camello; algunos ejemplos son: `suma()`, `tresSaludos()`, `calculaRaizCuadrada()`, `muestraTodosDatosCliente()`...

En la definición de una función, `cuerpo de la función` se sustituye por un bloque de instrucciones (limitado por llaves) que implementa la función.

Definimos algunos conceptos necesarios para seguir trabajando con funciones:

- **Llamada a la función:** es el nombre de la función, seguido de () -paréntesis-. Se convierte en una nueva instrucción que podemos utilizar para invocarla.
- **Prototipo de la función:** es la declaración de la función, donde se especifica su nombre, el tipo que devuelve y, entre paréntesis, los parámetros de entrada que utiliza. En nuestro ejemplo, el prototipo de la función `tresSaludos()` es:

```
static void tresSaludos()
```

- **Cuerpo de la función:** es el bloque de código que ejecuta la función cada vez que se invoca y que aparece entre llaves después del prototipo.

- **Definición de una función:** está formada por el prototipo más el cuerpo de la función.

De forma esquemática, la siguiente figura representa un programa en el que no se utilizan funciones (a la izquierda) y el mismo programa en el que se ha empleado una función. Se puede apreciar que, en el segundo caso, el cuerpo de la función solo está escrito una vez.

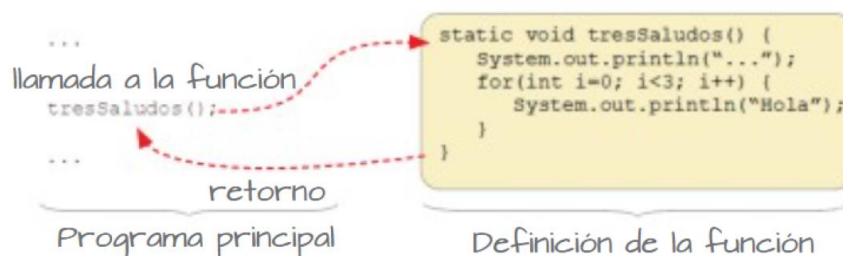


Con esto evitamos:

- La duplicidad del código: ya que el código se escribe una única vez, en la definición de la función.
- La dificultad en el mantenimiento: ahora, las modificaciones, en el caso de que sean necesarias, solo se realizan en un lugar: en la definición de la función.

El comportamiento de una llamada a una función consiste en:

1. Las instrucciones del programa principal se ejecutan hasta que encuentra la llamada a la función, en nuestro caso, `tresSaludos();`
2. La ejecución salta a la definición de la función.
3. Se ejecuta el cuerpo de la función.
4. Cuando la ejecución del cuerpo termina, retornamos al punto del programa desde donde se invocó la función.
5. El programa continúa su ejecución.



Tanto en la llamada a la función como en el retorno es posible incluir información útil. Estos flujos de información se verán a lo largo de la unidad.

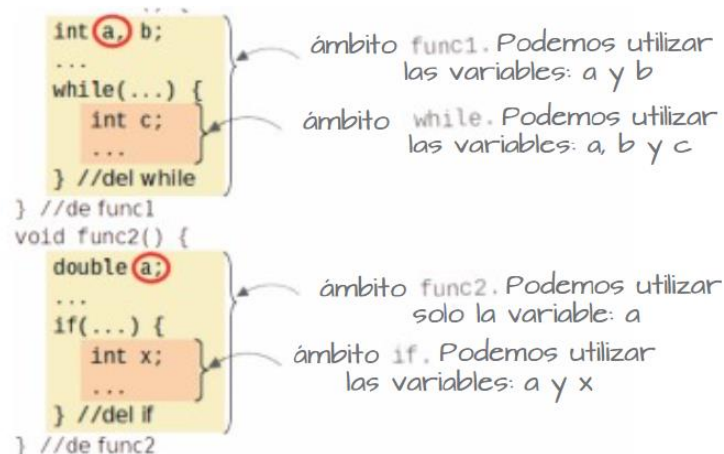
Los métodos que hemos utilizado de algunas clases de la API (como `nextInt()` de `Scanner`) en realidad son funciones. Una función y un método son conceptualmente idénticos, la única

diferencia está en el nombre, que varía dependiendo del paradigma de programación usado. En la programación estructurada se llaman **funciones**, y en la programación orientada a objetos, se denominan **métodos**.

4.2. ÁMBITO DE LAS VARIABLES

En el cuerpo de una función podemos declarar variables, que se conocen como **variables locales**. El ámbito de estas, es decir, donde pueden utilizarse, es la propia función donde se declaran, no pudiéndose utilizar fuera de ella. Nada impide que dentro del cuerpo de una función se utilicen sentencias (**if**, **if-else**, etc.) con sus respectivos bloques de instrucciones, donde a su vez, se pueden volver a declarar nuevas variables que se conocen como **variables de bloques**, siempre y cuando su nombre no coincida con una variable declarada antes, fuera del bloque, ya que esto producirá un error.

En el código de la siguiente figura, se definen dos funciones **func1()** y **func2()** y se representa gráficamente el ámbito de las distintas variables declaradas.



4.3. PASO DE INFORMACIÓN A UNA FUNCIÓN

En ocasiones, una función necesita conocer información externa para poder llevar a cabo su tarea. Veamos un ejemplo: la función **tresSaludos()** es conveniente cuando queremos saludar exactamente tres veces. Si deseamos saludar un número distinto de veces, estaríamos obligados a implementar las funciones: **unSaludo()**, **dosSaludos()**, **cuatroSaludos()**, etc. Es mucho más práctico implementar la función **variosSaludos()** a la que se le pasa el número de veces que deseamos saludar. De esta manera, si ejecutamos **variosSaludos(7)**, saludará siete veces y si ejecutamos **variosSaludos(2)**, lo hará en dos ocasiones.

```
static void variosSaludos(int veces) {  
    for (int i = 0; i < veces; i++) {  
        System.out.println("Hola.");  
    }  
}
```

La variable veces es un parámetro de entrada de la función `variosSaludos()`. Un parámetro de entrada de una función no es más que una variable local a la que se le asigna valores en cada llamada.

4.3.1. Valores en la llamada

En la llamada a una función se pueden pasar valores que provienen de literales, expresiones o variables. Por ejemplo, a la función `variosSaludos()` se le pasa un entero (número de veces que deseamos saludar).

```
variosSaludos(2); //llamada con un literal
int n = 3;
variosSaludos(2*n); //llamada con una expresión
```

4.3.2. Parámetros de entrada

Una función puede definirse para recibir tantos datos como necesite. Por ejemplo, una función que realiza la suma puede definirse para que se le pasen dos valores que sumar; y a otra que indica si una fecha es correcta se le pasarán tres valores: el año, el mes y el día de la fecha.

Para una función que calcula y muestra la suma de dos números, la llamada sería:

```
int a = 3;
suma(a, 2); //muestra la suma de a (que vale 3) más 2
```

Cada dato utilizado en la llamada a una función será asignado a un parámetro de entrada, especificado en la definición de la función con la siguiente sintaxis:

```
tipo nombreFuncion(tipo1 parametro1, tipo2 parametro2...) {
    cuerpo de la función
}
```

El primer parámetro de entrada lo hemos llamado `parametro1` y se le puede asignar un valor del tipo `tipo1`, y lo mismo ocurre con el resto de parámetros. El número de parámetros definidos en la función determina el número de valores que hay que utilizar en cada llamada.

Dentro del cuerpo de la función los parámetros son variables locales que han sido inicializadas. ¿De dónde toman los parámetros su valor? De la llamada a la función. De esta forma, en distintas llamadas podemos asignar distintos valores. Solo hay que tener en cuenta que el valor asignado a cada parámetro tiene que corresponder con su tipo. Veamos la función `variosSaludos()`:

```
static void variosSaludos(int veces) {
    int i;
    //disponemos de las variables locales: i y veces
    //el valor de veces se determina en la llamada
    for(i = 0; i < veces; i++) {
        System.out.println("Hola.");
    }
}
```

Si invocamos la función con

```
variosSaludos(7); //7 se asigna al primer parámetro: veces
```

Suponiendo que hubiéramos implementado la función `compruebaHora()`, a la que se le pasa la hora, minutos y segundos de un instante, y muestra en pantalla si la hora es correcta o incorrecta, el prototipo de la función sería:

```
static void compruebaHora(int hora, int minutos, int segundos)
```

Un ejemplo de llamada a la función es:

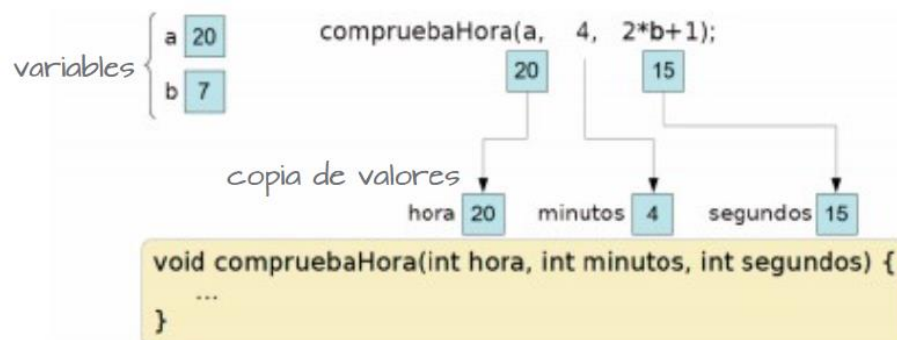
```
compruebaHora(a, 4, 2*b+1);
```

El mecanismo de paso de parámetro se representa en la siguiente figura:



En Java los parámetros toman su valor como una copia del valor de la expresión o variable utilizada en la llamada; este mecanismo de paso de parámetros se denomina **paso de parámetros por valor** o **por copia**.

En la siguiente figura se aprecia cómo se realiza la copia de los valores de las variables utilizadas en la llamada a las variables empleadas como parámetros.



Veámoslo detenidamente: tras ejecutar la llamada a la función, se salta al cuerpo de la función, copiando el valor del primer parámetro (el valor de `a`, que es 20) a la primera variable utilizada como parámetro (`hora`), el segundo valor utilizado en la llamada (4), al segundo parámetro de entrada (`minutos`), etc. El proceso se repite para cada pareja valor-parámetro.

Hay que destacar que cualquier cambio en un parámetro de entrada que se efectúe dentro del cuerpo de la función no repercute en la variable o expresión utilizada en la llamada, ya que lo que se modifica es una copia y no el dato original. Veamos un ejemplo:

```
int a = 1, b = 2, c = 3;
compruebaHora(a, b, c); //llamada
...
//definición de la función
public static void compruebaHroa(int hora, int minutos, int segundos)
    //hora tiene un valor asignado en la llamada (a=1)
    hora = 23;
    ...
}
```

Modificamos `hora`, pero la variable `a` sigue valiendo 1.

4.4. Valor devuelto por una función

Hemos visto que es posible pasar información hacia la función a través de los parámetros de entrada. También es posible que el paso de información sea en sentido contrario, es decir, desde el cuerpo de la función hacia el código donde se hace la llamada. Con esto conseguimos que la llamada a una función se convierta en un valor cualquiera. Este puede ser utilizado desde el lugar donde se invoca. Supongamos que disponemos de una nueva versión de la función `suma()` que, en vez de mostrar el valor de la suma de los números, lo devuelve, pudiéndoselo asignar a una variable:

```
int a = suma(2, 3);
int b = suma(7, 1) * 5;
```

En la primera instrucción, `suma(2,3)` se sustituye por 5, que se asigna a la variable `a`. En la segunda instrucción, `suma(7,1)` se sustituye por el valor 8, que se multiplica por 5, resultando 40, que se asigna a la variable `b`.

Hasta ahora hemos utilizado siempre `void` como tipo devuelto por una función, lo que indica que la función no devuelve nada, o dicho de otra forma: la llamada a la función no se sustituye por ningún valor. Es posible utilizar cualquier tipo para especificar que la llamada a la función se sustituirá por un valor del tipo indicado. ¿Cómo damos ese valor a la llamada de la función? Para ello disponemos de la instrucción `return` que finaliza la ejecución de la función y devuelve el valor indicado. De forma general,

```
public tipo nombreFunción(parámetros) {
    ...
    return (valor);
}
```

donde el tipo de `valor` debe coincidir con `tipo`. La instrucción `return` se utiliza en funciones con un tipo devuelto distinto a `void`.

Veamos cómo se implementa la función que realiza la suma de dos números enteros:


```
public static int suma(int x, int y) { //cada llamada devuelve un int
    int resultado;
    resultado = x + y;
    return(resultado); //sustituye la llamada por el valor de resultado
}
```

Debe existir una concordancia entre el tipo devuelto declarado en la función y el tipo del valor devuelto con `return`. Es importante recordar que la última instrucción de la función debe ser `return`, que fuerza su fin. En caso de existir instrucciones posteriores, no se ejecutarían.

Nada impide utilizar varios `return` en una misma función, pero es una práctica desaconsejable, ya que una función debe tener un único punto de entrada y de salida; el uso de varios `return` rompe esta norma. En las actividades resueltas utilizamos un único `return` encada función.

4.5. SOBRECARGA DE FUNCIONES

Java permite que dos o más funciones compartan el mismo identificador en un mismo programa. Esto es lo que se conoce como **sobrecarga de funciones**. La forma de distinguir entre las distintas funciones sobrecargadas es mediante sus listas de parámetros, que deben ser distintas, ya sean en número o en tipo.

Las funciones sobrecargadas pueden devolver tipos distintos, aunque estos no sirven para distinguir una función sobrecargada de otra.

Supongamos que queremos diseñar una función para calcular la suma de dos enteros, pero también es útil hacer una suma ponderada, donde cada sumando tenga un peso distinto.

Veamos las dos funciones sobrecargadas:

```
//función sobrecargda
public static int suma(int a, int b) {
    int suma;
    suma = a+ b;
    return(suma);

//función sobrecargda
public static double suma(int a, double pesoA, int b, double pesoB ) {
    double suma;
    suma = a * pesoA / (pesoA + pesoB) + b * pesoB / (pesoA + pesoB);
    return(suma);
}
```

A partir de la definición de las funciones, ambas están disponibles, y cumplen con la única restricción de las funciones sobrecargadas: que se puedan distinguir mediante sus parámetros.

Si invocamos a la función `suma()` de la forma: `suma (2,3)`, se ejecutará la primera versión, y devolverá 5. En cambio, si se llama con: `suma (2,0.25, 3,0.75)`, se ejecutará la segunda versión, y devolverá 3,25.

Es muy común encontrar en la API funciones (métodos) sobrecargadas, ya que permiten agrupar distintas funcionalidades, cuyo uso es similar, bajo el mismo identificador. Por ejemplo, la función que más hemos utilizado hasta ahora, `System.out.println`, se encuentra sobrecargada para poder mostrar en pantalla cualquier tipo de dato.

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**.

```
public 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 evitar lo, 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:

```
public int funcionReucrsiva(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:

$$\begin{aligned} n! &= n \times (n-1) \times (n-2) \dots \times 2 \times 1 \Rightarrow \\ n! &= n \times (n-1)! \end{aligned}$$

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:

```
public 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);
}
```

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

```
public long factorial(3) {
    long resultado;
    if (3 == 0) { // falso
        ...
    } else {
        resultado = 1 * 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)`:

```
public 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:

```
public 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)`:

```
public 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.

```
public 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:

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

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

```
public 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
```