

Funciones

Objetivos

- Asimilar el concepto de función, las ventajas de su uso y la implicación en la mejora del mantenimiento de aplicaciones.
- Entender y usar el concepto de parámetro de entrada, así como el mecanismo para generalizar el comportamiento de las funciones.
- Escribir programas que hagan un uso adecuado de las funciones y del valor devuelto por estas.
- Resolver problemas mediante el uso de funciones recursivas.

Contenidos

- 4.1. Conceptos básicos
- 4.2. Ámbito de las variables
- 4.3. Paso de información a una función
- 4.4. Valor devuelto por una función
- 4.5. Sobrecarga de funciones
- 4.6. Recursividad

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(); //sustitución por 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
}

```

Argot técnico



En la Unidad 7, donde veremos las clases, se explicará que lo que ahora estamos llamando *función* también se conoce con el nombre de *método*.

También veremos en profundidad el concepto de método estático (`static`).

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()`...

Argot técnico



En la Apartado 4.4 se verá con detalle el tipo devuelto por una función. Por ahora, nos basta saber que una función devolverá un dato con algún resultado. Dicho dato, como todas las variables, deberá tener un tipo.

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 Figura 4.1 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.



Figura 4.1. Representación de un programa sin y con funciones.

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 (véase la Figura 4.2) 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.

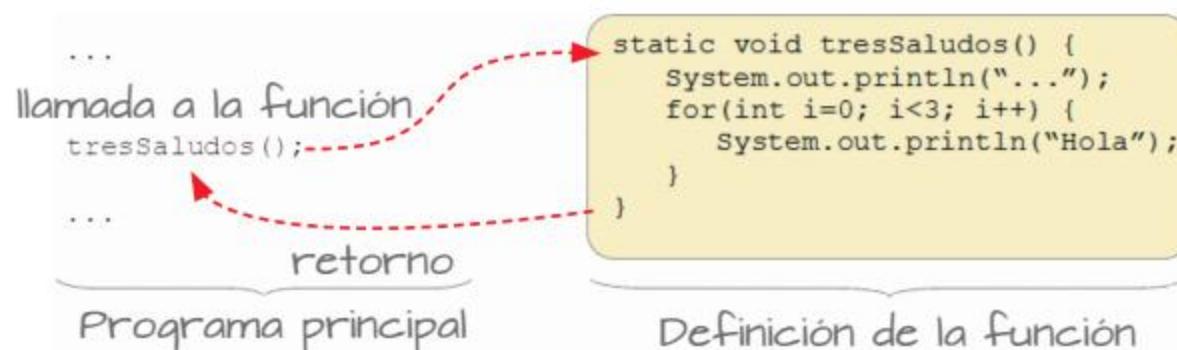


Figura 4.2. Visualización del flujo de ejecución en la llamada a una función.

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 Figura 4.3, se definen dos funciones `func1()` y `func2()` y se representa gráficamente el ámbito de las distintas variables declaradas.

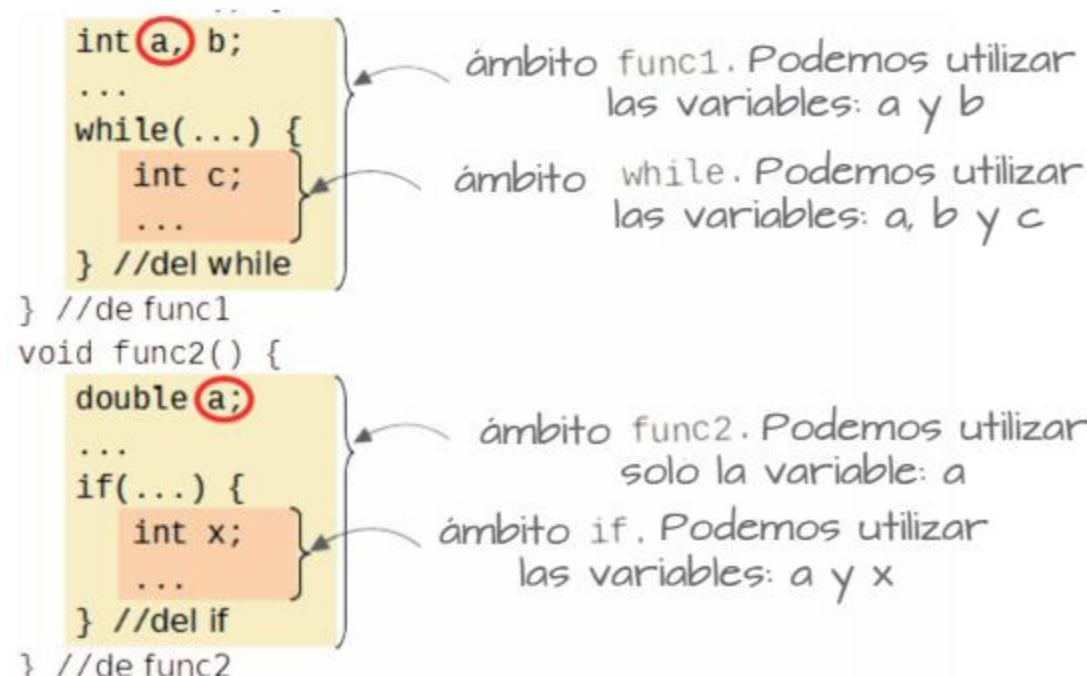


Figura 4.3. Ámbitos de distintas variables. Aunque las variables `a` de `func1()` y de `func2()` (marcadas en rojo) comparten identificador, son variables distintas.

■ 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 Figura 4.4.

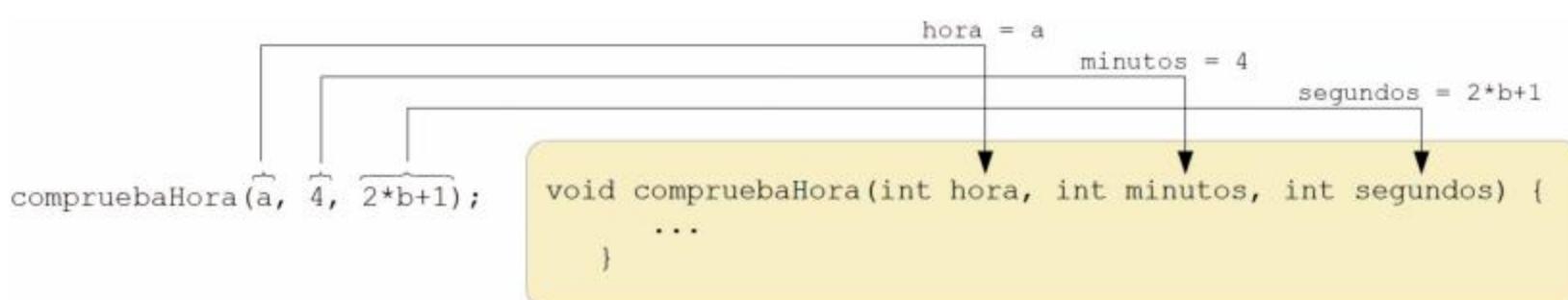


Figura 4.4. Paso de parámetros a una función.

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 Figura 4.5 se aprecia cómo se realiza la copia de los valores de las variables utilizadas en la llamada a las variables empleadas como parámetros.

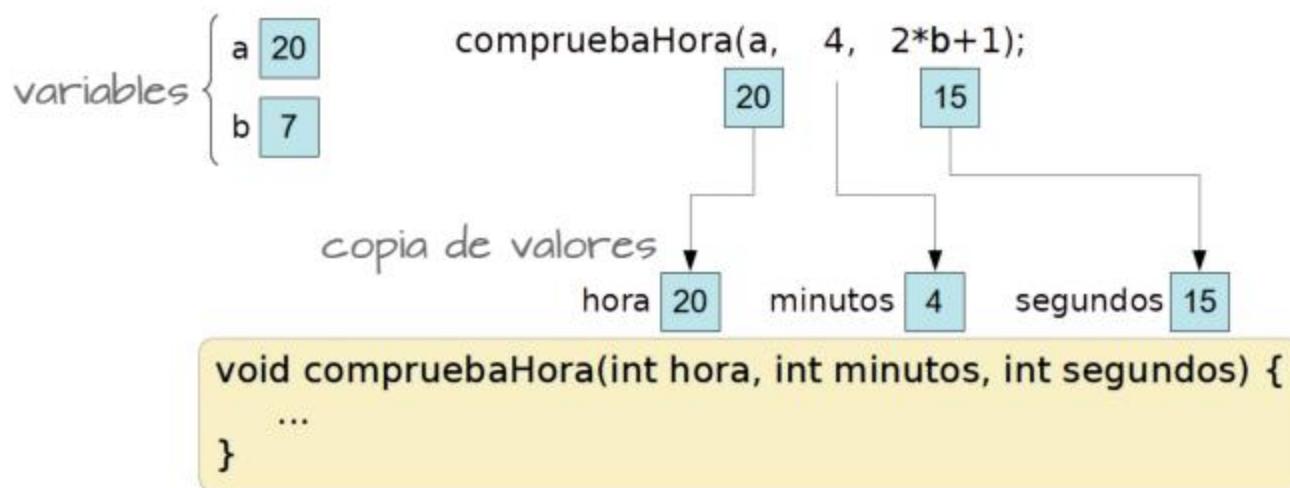


Figura 4.5. Mecanismo de copia de valores a los parámetros.

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
static void compruebaHora(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.

Actividad resuelta 4.1

Diseñar la función `eco()` a la que se le pasa como parámetro un número n , y muestra por pantalla n veces el mensaje «Eco...».

Solución

```

import java.util.Scanner;
/* Las soluciones irán acompañadas de una función main que sirva de prueba.
 * El prototipo de la función es: void eco(int n). */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduzca un número: ");
        int n = sc.nextInt();
        System.out.println("--Antes de llamar a la función--");
  
```

```

    eco(n); //invocamos la función
    System.out.println("--Después de llamar a la función--");
}
//La función lo único que hace es mostrar un mensaje repetido mediante un bucle
static void eco(int a) { //el parámetro no tiene por qué llamarse como en la
                        //llamada
    for (int i = 0; i < a; i++) {
        System.out.println("Eco... ");
    }
}
}

```

Actividad resuelta 4.2

Escribir una función a la que se le pasen dos enteros y muestre todos los números comprendidos entre ellos.

Solución

```

import java.util.Scanner;
//Tenemos que saber si los números están en orden creciente (3, 7) o decreciente (7, 3).
public class Main {
    //la función ordena los valores pasados y los utiliza como valores de un bucle for
    static void mostrar(int a, int b) {
        int mayor = a > b ? a : b; //asignamos a mayor el mayor entre a y b
        int menor = a < b ? a : b; //y en menor el más pequeño entre a y b
        for (int i = menor; i <= mayor; i++) { //siempre iremos del menor al mayor
            System.out.println(i);
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduzca primer número: ");
        int a = sc.nextInt();
        System.out.print("Introduzca segundo número: ");
        int b = sc.nextInt();
        mostrar(a, b);
    }
}

```

Actividad resuelta 4.3

Realizar una función que calcule y muestre el área o el volumen de un cilindro, según se especifique. Para distinguir un caso de otro se le pasará como opción un número: 1 (para el área) o 2 (para el volumen). Además, hay que pasarle a la función el radio de la base y la altura.

$$\text{área} = 2\pi \cdot \text{radio} \cdot (\text{altura} + \text{radio})$$

$$\text{volumen} = \pi \cdot \text{radio}^2 \cdot \text{altura}$$

Solución

```

import java.util.Scanner;
/* Recordemos que el área de un cilindro es 2*PI*radio*(altura+radio) y
 * la fórmula para el volumen es PI*(radio al cuadrado)*altura. */

```

```

public class Main {
    static void areaVolumenCilindro(double radio, double altura, int opcion) {
        double volumen, area;
        switch (opcion) {
            case 1 -> {
                volumen = Math.PI * Math.pow(radio, 2) * altura; //aplicamos la fórmula
                System.out.println("El volumen es de: " + volumen);
            }
            case 2 -> {
                area = 2 * Math.PI * radio * (altura + radio);
                System.out.println("El área es de: " + area);
            }
            default -> System.out.println("Indicador del cálculo erróneo");
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduzca radio: ");
        double radio = sc.nextDouble();
        System.out.print("Introduzca altura: ");
        double alt = sc.nextDouble();
        System.out.print("Qué desea calcular (1 (área)/ 2 (volumen): ");
        int tipoCalculo = sc.nextInt();
        System.out.println();
        areaVolumenCilindro(radio, alt, tipoCalculo);
    }
}

```

■ 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,

```
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:

```
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` en cada función.

Argot técnico



Aunque es posible que una función tenga varios puntos de salida (`return`), no es recomendable que disponga de más de uno. El hecho de usar un único punto de salida aumenta la legibilidad, facilita el mantenimiento y la depuración del código.

Actividad resuelta 4.4

Diseñar una función que recibe como parámetros dos números enteros y devuelve el máximo de ambos.

Solución

```
import java.util.Scanner;
//En caso de que ambos números sean iguales, el algoritmo también es válido.
public class Main {
    //compara los parámetros, a y b, y devuelve el mayor de ambos
    static int maximo(int a, int b) {
        int max;
        if (a > b) { // si a es mayor que b
            max = a;
        } else { // si son iguales o b mayor que a
            max = b;
        }
        return (max); //devuelve el valor de la variable max
    }
    /* main para probar la función */
    public static void main(String[] args) {
```

```

Scanner sc = new Scanner(System.in);
System.out.print("Introduzca un número: ");
int a = sc.nextInt();
System.out.print("Introduzca otro número: ");
int b = sc.nextInt();
System.out.println("El número mayor es: " + maximo(a, b));
}
}

```

Actividad resuelta 4.5

Crear una función que, mediante un booleano, indique si el carácter que se pasa como parámetro de entrada corresponde con una vocal.

Solución

```

/* La función tendrá en cuenta las vocales minúsculas y mayúsculas. No
 * consideraremos las vocales acentuadas (á, é, ...) o con diéresis. */
public class Main {
    //programa principal para probar la función
    public static void main(String[] args) {
        System.out.println("La i es una vocal " + esVocal('i'));
        System.out.println("La E es una vocal " + esVocal('E'));
        System.out.println("La f es una vocal " + esVocal('f'));
    }

    //comparamos el parámetro de entrada c, con cada posible valor de una vocal.
    //Por simplicidad, obviamos las vocales acentuadas y con diéresis.
    static boolean esVocal(char c) {
        boolean resultado; //true si es una vocal y false en caso contrario
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' ||
            c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U') {
            resultado = true;
        } else {
            resultado = false;
        }
        return resultado;
    }
}

```

Actividad resuelta 4.6

Diseñar una función con el siguiente prototipo:

```
boolean esPrimo(int n)
```

que devolverá `true` si `n` es primo y `false` en caso contrario.

Solución

```

/* La función esPrimo() indica con un booleano si el número pasado como parámetro
 * es primo. Un número n es primo si no es divisible por ningún número entre 2 y n-1
 * Recordemos que un número primo es solo divisible por él mismo y por 1. */

```

```

public class Main {
    static boolean esPrimo(int num) {
        boolean primo = true; // suponemos que el número es primo
        int i = 2; //primer número por el que veremos si es divisible
        if (num < 2) { // el primer primo es 2
            primo = false;
        }
        while (i < num && primo == true) { // se detiene si encuentra un divisor de num
            if (num % i == 0) { // si num es divisible por i
                primo = false; // entonces no es un número primo
            }
            i++;
        }

        // este algoritmo puede mejorar sabiendo que si un número no es divisible por
        // ningún entero comprendido entre 2 y su raíz cuadrada, entonces ya no será
        // divisible por ningún otro número y será primo. Quedaría:
        // while (i <= (int) Math.sqrt(num) && primo == true) {
        //     ...
        // }
        //lo cual ahorra muchas vueltas para números primos grandes
        return (primo);
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 15; i++) {
            if (esPrimo(i)) {
                System.out.println(i + " es primo");
            } else {
                System.out.println(i + " es compuesto");
            }
        }
    }
}

```

Actividad resuelta 4.7

Escribir una función a la que se le pase un número entero y devuelva el número de divisores primos que tiene.

Solución

```

/* Para calcular los divisores de un número, solo tendremos en cuenta los números
 * primos comprendidos entre 1 y el número que nos interese.
 * Un ejemplo: los divisores de 24 son: 1, 2 y 3.
 * Aunque 4 y 6 también dividen a 24, no se consideran al no ser primos. */
public class Main {
    // la función esPrimo() está implementada en la Actividad Resuelta anterior.
    static boolean esPrimo(int num) {
        ... //en el ejercicio anterior
    }

    static int numDivisoresPrimos(int num) {
        int cont = 0; // contador de divisores

```

```

        for (int i = 2; i <= num; i++) {
            if (esPrimo(i) && num % i == 0) { // si i es primo y divide a num
                cont++; // incrementamos el número de divisores
            }
        }
        return (cont);
    }

    //programa principal para probar la función
    public static void main(String[] args) {
        System.out.println("Divisores de 24: " + numDivisoresPrimos(24));
    }
}

```

Actividad resuelta 4.8

Diseñar la función `calculadora()`, a la que se le pasan dos números reales (operando) y qué operación se desea realizar con ellos. Las operaciones disponibles son: sumar, restar, multiplicar o dividir. Estas se especifican mediante un número: 1 para la suma, 2 para la resta, 3 para la multiplicación y 4 para la división. La función devolverá el resultado de la operación mediante un número real.

Solución

```

public class Main {
    //programa para probar
    public static void main(String[] args) {
        for (int operacion = 1; operacion <= 4; operacion++) { //todas las operaciones
            double resultado = calculadora(3.0, 4.0, operacion); //operamos con 3.0
                                                               //y 4.0
            System.out.println(resultado);
        }
    }
    //Realiza la operación indicada:
    // 1- suma
    // 2- resta
    // 3- multiplicación
    // 4- división
    static double calculadora(double a, double b, int operacion) {
        double result; // resultado de la operación
        result = switch (operacion) {
            case 1 -> //suma
                a + b; //si solo existe una instrucción no hace falta escribir yield
            case 2 -> //resta
                a - b;
            case 3 -> //multiplicación
                a * b;
            case 4 -> //división
                (double)a / b;
                //falta comprobar que no es una división por 0
                //el cast fuerza que la división sea real
            default -> {
                System.out.println("Operación no válida");
                yield 0; //si la operación no tiene sentido devolveremos 0
            }
        };
    }
}

```

```
        }
    }

    return (result);
}
}
```

■ 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 su 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 sobrecargada
static int suma(int a, int b) {
    int suma;
    suma = a + b;
    return(suma);
}

//función sobrecargada
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.

Actividad resuelta 4.9

Repetir la Actividad resuelta 4.4 con una versión que calcule el máximo de tres números.

Solución

```
/* Vamos a sobrecargar la función para que tenga tres parámetros: maximo(a,b,c).
 * Para implementar la función podemos escribir el algoritmo desde cero o
 * basarnos en la función maximo() de la Act. resuelta 4.4. En este caso vamos
 * reutilizar el código existente. */
public class Main {
    // función maximo para tres números
    static int maximo(int a, int b, int c) {
        int aux = maximo(a, b); //la variable auxiliar contiene el mayor entre a y b
        return (maximo(aux, c)); //devuelve el mayor entre aux y c
    }

    // función máximo para dos números, necesaria para la definición anterior
    static int maximo(int a, int b) {
        ... //en la actividad resuelta 4.4
    }

    // main para probar la función
    public static void main(String[] args) {
        int max = maximo(2, 9, 7);
        System.out.println("El mayor es: " + max);
    }
}
```

■ 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 `funcionRecursiva()` se invoca a `funcionRecursiva()`, donde a su vez, se volverá a llamar a `funcionRecursiva()`, 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:

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

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:

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

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:

```

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
```

Actividad resuelta 4.10

Diseñar una función que calcule a^n , donde a es real y n es entero no negativo. Realizar una versión iterativa y otra recursiva.

Solución a)

```

import java.util.*;
// El exponente podrá ser 0, pero la base no. Ya que 0 elevado a 0 no está definido.
public class Main {
    static double aElevadoN(double a, int n) {
        double res = 1; // el resultado se inicializa a 1, ya que multiplicamos
        for (int i = 1; i <= n; i++) {
            res = res * a; //multiplicamos
        }
        return (res);
    }

    //programa principal para probar la función
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US); //para permitir puntos (.) en los decimales
        System.out.print("Introduzca base (real): ");
        double base = sc.nextDouble();
        System.out.print("Introduzca exponente (entero no negativo): ");
        int exp = sc.nextInt();
        double res = aElevadoN(base, exp);
        System.out.println(base + " elevado a " + exp + " = " + res);
    }
}

```

Solución b)

```

import java.util.*;
/* La funciones recursivas suelen tener la misma estructura:
 * - caso base: que permite salir de la recursividad
 * - llamada recursiva.
 * En nuestro caso: el caso base es aElevadoN(x, 0) = 1
 * y la llamada recursiva: aElevadoN(a, n) = aElevadoN(a, n-1) * a      */
public class Main {
    //programa principal para probar la función aElevadoN(), de forma recursiva.
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US);
        System.out.print("Introduzca base (real): ");
        double base = sc.nextDouble();
        System.out.print("Introduzca el exponente: ");
        int exp = sc.nextInt();
        System.out.println("El resultado es: " + aElevadoN(base, exp));
    }

    static double aElevadoN(double a, int n) {
        double res;
        if (n == 0) { // caso base
            res = 1; //a elevado a 0 es 1
        } else {
            res = a * aElevadoN(a, n - 1); //llamada recursiva
        }
        return (res);
    }
}

```

Actividad resuelta 4.11

Escribir una función que calcule de forma recursiva el máximo común divisor de dos números. Para ello sabemos:

$$mcd(a, b) = \begin{cases} mcd(a - b, b) & \text{si } a \geq b \\ mcd(a, b - a) & \text{si } b > a \\ a & \text{si } b = 0 \\ b & \text{si } a = 0 \end{cases}$$

Solución

```

import java.util.Scanner;
/* Para calcular el máximo común divisor usaremos, según el caso, una de las dos
 * llamadas recursivas:
 * - mcd(a, b) = mcd(a-b, b) si a >= b o
 * - mcd(a, b) = mcd(a, b-a) si b > a
 *
 * Ambas llamadas recursivas pueden unificarse en una sola, teniendo en cuenta el valor
 * máximo y mínimo de a y b. Si min = mínimo(a,b) y max = máximo(a, b), entonces:
 * - mcd(min, max) = mcd (min, max-min);
 * Y tenemos dos casos bases:
 * - mcd(a, b) = a si b es 0
 * - mcd(a, b) = b si a es 0. */
public class Main {
    //programa principal que pide dos números y calcula su mcd
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a, b, resultado;
        System.out.print("Introduzca primer número: ");
        a = sc.nextInt();
        System.out.print("Introduzca segundo número: ");
        b = sc.nextInt();
        resultado = mcd(a, b);
        System.out.println("El mcd es " + resultado);
    }

    //función recursiva para calcular el mcd.
    static int mcd(int a, int b) {
        int resultado;
        if (a == 0) { // primer caso base
            resultado = b;
        } else if (b == 0) {
            resultado = a; //segundo caso base
        } else {
            int min = a <= b ? a : b; //valor mínimo entre a y b
            int max = a <= b ? b : a; //valor máximo entre a y b
            resultado = mcd(min, max-min); //llamada recursiva
        }
        return (resultado);
    }
}

```

Actividad resuelta 4.12

Diseñar una función recursiva que calcule el enésimo término de la serie de Fibonacci. En esta serie el enésimo valor se calcula sumando los dos valores anteriores de la serie. Es decir:

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

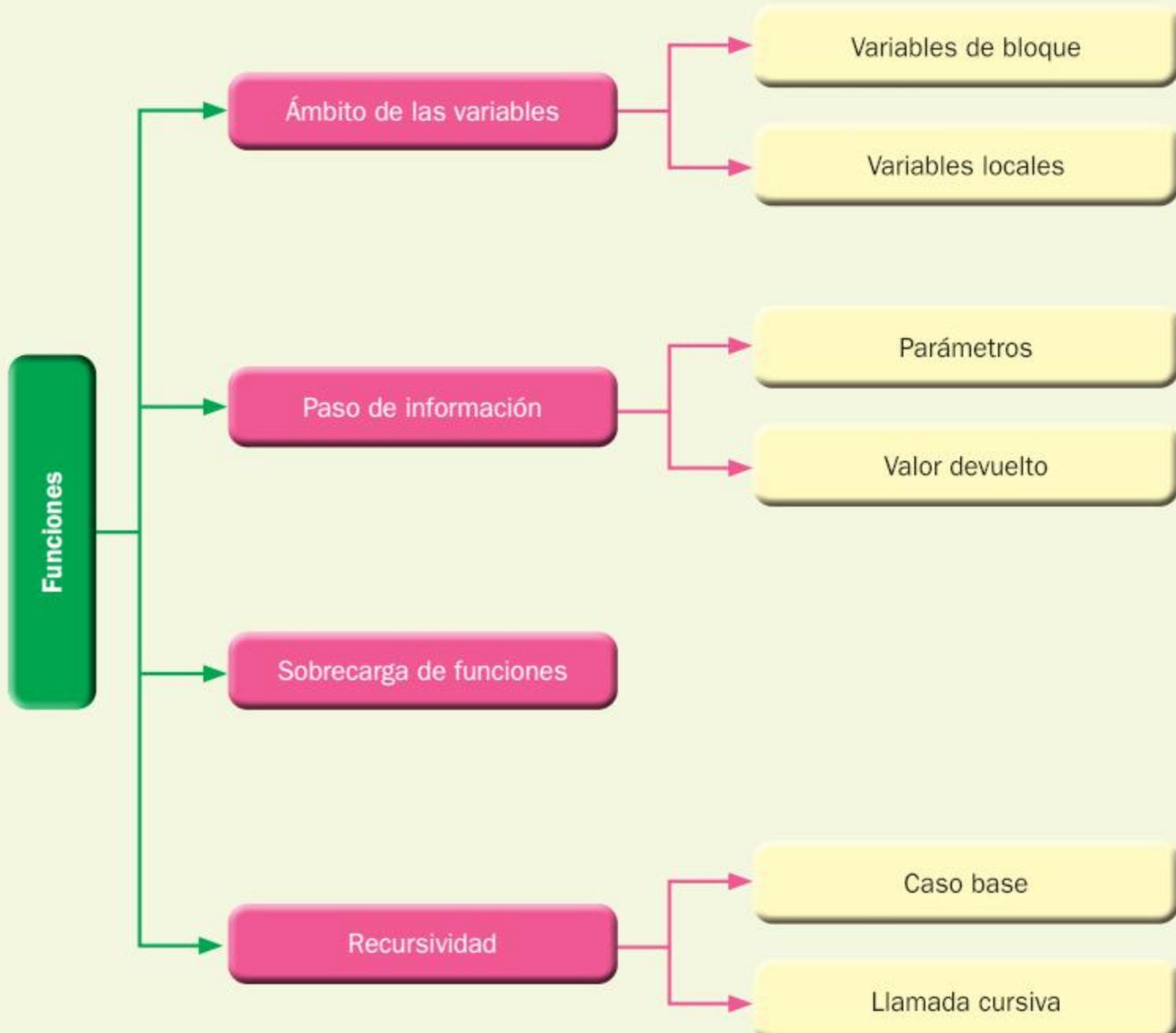
$$\text{fibonacci}(0) = 1$$

$$\text{fibonacci}(1) = 1$$

Solución

```
import java.util.Scanner;
/* En la serie de Fibonacci tendremos:
 * - caso general: fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
 * - existen dos casos base: fibonacci(0) = 1
 *                         fibonacci(1) = 1      */
public class Main {
    //programa principal para probar la función fibo()
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Vamos a calcular fibonacci(n)");
        System.out.print("Introduzca n (se recomienda n<40): ");
        int num = sc.nextInt();
        int resultado = fibo(num); // si n es muy grande esto puede tardar bastante
        System.out.println("\nfibonacci(" + num + ") = " + resultado);
    }

    //función recursiva
    static int fibo(int num) {
        int res;
        if (num == 0 || num == 1) { // casos base
            res = 1;
        } else {
            res = fibo(num - 1) + fibo(num - 2); // caso general recursivo
        }
        return (res);
    }
}
```



Actividades de comprobación

- 4.1. Los parámetros en la llamada a una función en Java pueden ser opcionales si:**
- a) Todos los parámetros son del mismo tipo.
 - b) Todos los parámetros son de distinto tipo.
 - c) Nunca pueden ser opcionales.
 - d) Siempre que el tipo devuelto no sea `void`.
- 4.2. Una variable local (declarada dentro de una función) puede usarse:**
- a) En cualquier lugar del código.
 - b) Solo dentro de `main()`.
 - c) Solo en la función donde se ha declarado.
 - d) Ninguna de las opciones anteriores es correcta.
- 4.3. El tipo devuelto de todas las funciones definidas en nuestro programa tiene que ser siempre:**
- a) `int`.
 - b) `double`.
 - c) `void`.
 - d) Ninguna de las opciones anteriores es correcta.
- 4.4. ¿Qué instrucción permite a una función devolver un valor?**
- a) `value`.
 - b) `return`.
 - c) `static`.
 - d) `function`.
- 4.5. La forma de distinguir entre dos o más funciones sobrecargadas es:**
- a) Mediante su nombre.
 - b) Mediante el tipo devuelto.
 - c) Mediante el nombre de sus parámetros.
 - d) Mediante su lista de parámetros: número o tipos.
- 4.6. ¿Cuál es la definición de una función recursiva?**
- a) Es aquella que se invoca desde dentro de su propio bloque de instrucciones.
 - b) Es aquella cuyo nombre permite la sobrecarga y además realiza alguna comprobación mediante `if`.
 - c) Es aquella cuyo bloque de instrucciones utiliza alguna sentencia `if` (lo que llamamos caso base).
 - d) Es aquella que genera un bucle infinito.
- 4.7. El paso de parámetros a una función en Java es siempre:**
- a) Un paso de parámetros por copia.
 - b) Un paso de parámetros por desplazamiento.
 - c) Un paso de parámetros recursivo.
 - d) Un paso de parámetros funcional.

- 4.8.** En el caso de que una función devuelva un valor, ¿cuál es la recomendación con respecto a la instrucción `return`?
- Utilizar tantos como hagan falta.
 - Emplear tantos como hagan falta, pero siempre que se encuentren en bloques de instrucciones distintas.
 - Usar solo uno.
 - Utilizar solo uno, que será siempre la primera instrucción de la función.
- 4.9.** ¿Cuáles de las siguientes operaciones se pueden implementar fácilmente mediante funciones recursivas?
- $a^n = a \times a^{n-1}$.
 - $\text{esPar}(n) = \text{esImpar}(n - 1)$ y $\text{esImpar}(n) = \text{esPar}(n - 1)$.
 - $\text{suma}(a, b) = \text{suma}(a + 1, b - 1)$.
 - Todas las respuestas anteriores son correctas.
- 4.10.** En los identificadores de las funciones, al igual que en los de las variables, se recomienda utilizar la siguiente nomenclatura:
- `suma_notas_alumnos ()`.
 - `sumanotasalumnos ()`.
 - `SumaNotasAlumnos ()`.
 - `sumaNotasAlumnos ()`.

Actividades de aplicación

- 4.11.** Diseña una función que calcule y muestre la superficie y el volumen de una esfera.

$$\text{Superficie} = 4\pi \cdot \text{radio}^2$$

$$\text{Volumen} = \frac{4\pi}{3} \cdot \text{radio}^3$$

- 4.12.** Implementa la función

```
static double distancia(double x1, double y1, double x2, double y2)
```

que calcula y devuelve la distancia euclídea que separa los puntos (x_1, y_1) y (x_2, y_2) . La fórmula para calcular esta distancia es:

$$\text{distancia} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- 4.13.** Crea la función `muestraPares(int n)` que muestre por consola los primeros `n` números pares.
- 4.14.** Escribe una función a la que se pase como parámetros de entrada una cantidad de días, horas y minutos. La función calculará y devolverá el número de segundos que existen en los datos de entrada.

- 4.15. Diseña una función a la que se le pasan las horas y minutos de dos instantes de tiempo, con el siguiente prototipo:

```
static int diferenciaMin(int hora1, int minuto1, int hora2, int minuto2)
```

La función devolverá la cantidad de minutos que existen de diferencia entre los dos instantes utilizados.

- 4.16. Implementa la función `divisoresPrimos()` que muestra, por consola, todos los divisores primos del número que se le pasa como parámetro.

- 4.17. Escribe una función que decida si dos números enteros positivos son amigos. Dos números a y b son amigos si la suma de los divisores propios (distintos de él mismo) de a es igual a b . Y viceversa.

Para probar se pueden usar los números 220 y 284, que son amigos.

- 4.18. Crea una función que muestre por consola una serie de números aleatorios enteros. Los parámetros de la función serán: la cantidad de números aleatorios que se mostrarán y los valores mínimos y máximos que estos pueden tomar.

- 4.19. Sobrecarga la función realizada en la Actividad de aplicación 4.18 para que el único parámetro sea la cantidad de números aleatorios que se muestra por consola. Los números aleatorios serán reales y estarán comprendidos entre 0 y 1.

Actividades de ampliación

- 4.20. Busca en internet información sobre el paradigma de programación funcional.
- 4.21. Completa tu conocimiento sobre las funciones recursivas. Investiga sobre la recursividad directa e indirecta, y sobre los mecanismos para convertir un algoritmo recursivo en uno iterativo.
- 4.22. Realizad un pequeño trabajo de investigación en grupo, donde busquéis mecanismos para que una función devuelva más de un valor.
- 4.23. Es interesante comprender cómo una función trabaja a bajo nivel y realiza el cambio de contexto que permite llevar a cabo el salto desde el lugar donde se invoca hasta la definición de la función y, posteriormente, el retorno al lugar donde se invoca la función. Realizad un debate en clase y aportad ideas de cómo implementaríais las funciones si fueseis un compilador.
- 4.24. Algunos lenguajes de programación disponen de una herramienta denominada *funciones inline*. Realiza una búsqueda del concepto de función inline y cómo lo implementa la máquina virtual de Java.
- 4.25. Existen funciones a las que se les pasa como parámetros otras funciones. Lee sobre esto y motiva para qué crees que puede ser útil.



Tablas

Objetivos

- Conocer las tablas, que permiten almacenar múltiples valores en una variable.
- Crear tablas de distinto tipo y longitudes.
- Utilizar las operaciones básicas que se emplean con las tablas.
- Diseñar programas que hagan uso de tablas, donde se almacenan los datos necesarios.
- Modificar la longitud de una tabla en tiempo de ejecución sin pérdida de los datos que contiene.
- Usar la API de Java relacionada con las tablas y aplicar su uso a la resolución de problemas.

Contenidos

- 5.1. Variables escalares versus tablas
- 5.2. Índices
- 5.3. Construcción de tablas
- 5.4. Referencias
- 5.5. Uso de tablas
- 5.6. Tablas como parámetros de funciones
- 5.7. Operaciones con tablas: la clase Arrays
- 5.8. Tablas n -dimensionales