Grado en Ingeniería Informática
Introducción a la Programación
2021-2022

Tema 1. Fundamentos de Programación

Uso de subprogramas

Los lenguajes de programación permiten descomponer un programa complejo en distintos **subprogramas**:

- **Subprogramas** = **Funciones** y **procedimientos** en lenguajes de programación estructurada
- **Subprogramas** = **Métodos** en lenguajes de programación orientada a objetos

Razones para crear un **subprograma** son:

- Reducir la complejidad del programa (“divide y vencerás”).
- Eliminar código duplicado.
- Mejorar la legibilidad del código.
- Promover la reutilización de código

Uso de subprogramas

Ejemplo: Supongamos que necesitamos obtener la suma de enteros de 1 a 10, de 20 a 30, de 35 a 45 respectivamente.

Podríamos escribir el fragmento de código siguiente:

```
int suma = 0;
for (int i = 1; i <= 10; i++)
        suma += i;
System.out.println("La suma de 1 a 10 es  " + suma);

suma = 0;
for (int i = 20; i <= 30; i++)
        suma += i;
System.out.println("La suma de 20 a 30 es " + suma);

suma = 0;
for (int i = 35; i <= 45; i++)
        suma += i;
System.out.println("La suma de 35 a 45 es " + suma);
```

Uso de subprogramas

- Podemos observar en el ejemplo anterior que para obtener las sumas anteriores el código es muy parecido excepto que el comienzo y el final de los números enteros es diferente.
- Mediante la definición de un **método** y la invocación del mismo podemos evitar escribir tres veces el mismo fragmento de código. Es decir, evita la **duplicidad del código** con la correspondiente reducción de tamaño del programa (reducción del número de líneas de código) y haciéndolo más **legible**.
- Facilita el **mantenimiento**, ya que cualquier modificación necesaria dentro del fragmento de código repetido tendría que realizarse en todos y cada uno, mientras que, teniendo todo el código en un método, únicamente se tendría que hacer la modificación en dicho método.

Tema 1. Fundamentos de Programación

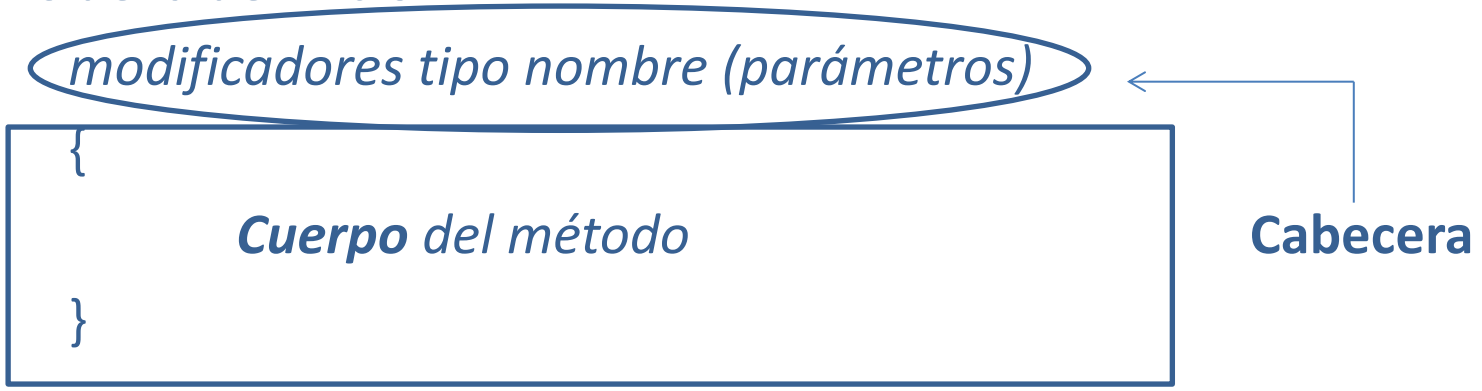
Métodos

- Definición e invocación
- Estado de la memoria
- Paso de parámetros
- Sobrecarga de métodos
- Recursión básica
- Abstracción de métodos y refinamiento por pasos

Métodos

Definición e invocación

Sintaxis de la definición



La estructura de un **método** se divide en **Cabecera** y **Cuerpo**.

- **Cabecera** (determina su interfaz)

  modificadores tipo nombre (parámetros)

```
public static int sumaEnteros(int i1, int i2) {
      int suma = 0;
      for (int i = i1; i <= i2; i++)
    suma += i;
      return suma;
}
```

Métodos

En la cabecera de un método se distingue:

- **Modificadores**

El modificador **public** indica que se puede **acceder** al método desde el exterior de la clase.

El modificador **static** indica que se trata de un **método de clase** (es decir un método común para todos los objetos de la clase).

- **Tipo devuelto** (cualquier tipo primitivo, no primitivo o `void`)

Indica de que tipo es la salida del método, es decir, el **tipo** del resultado que se obtiene tras llamar al método desde el exterior.

NOTA:

`void` se emplea cuando el método **no** devuelve ningún valor.

- **Nombre del método**

Identificador válido en Java.

Métodos

CONVENCIÓN:

En Java, los nombres de métodos (identificador) comienzan con minúscula.

- Cuando el método no devuelve ningún valor

(métodos `void`):

El nombre del método suele estar formado por un **verbo**.

Ejemplo: mostrarTriangulo

- Cuando el método devuelve un valor

(métodos con tipo):

El nombre del método suele ser una descripción del valor devuelto por el método.

Ejemplo: esPrimo

Métodos

- **Parámetros formales**

Entradas que necesita el método para realizar la tarea de la que es responsable. Se indica el tipo y el identificador del parámetro. Los parámetros se separan por comas.

MÉTODOS SIN PARÁMETROS:

Cuando un método no tiene entradas, hay que poner **()**

Signatura de un método

El nombre de un método, los tipos de sus parámetros y el orden de los mismos definen la **signatura** de un método.

Los modificadores y el tipo del valor devuelto por un método no forman parte de la signatura del método.

Métodos

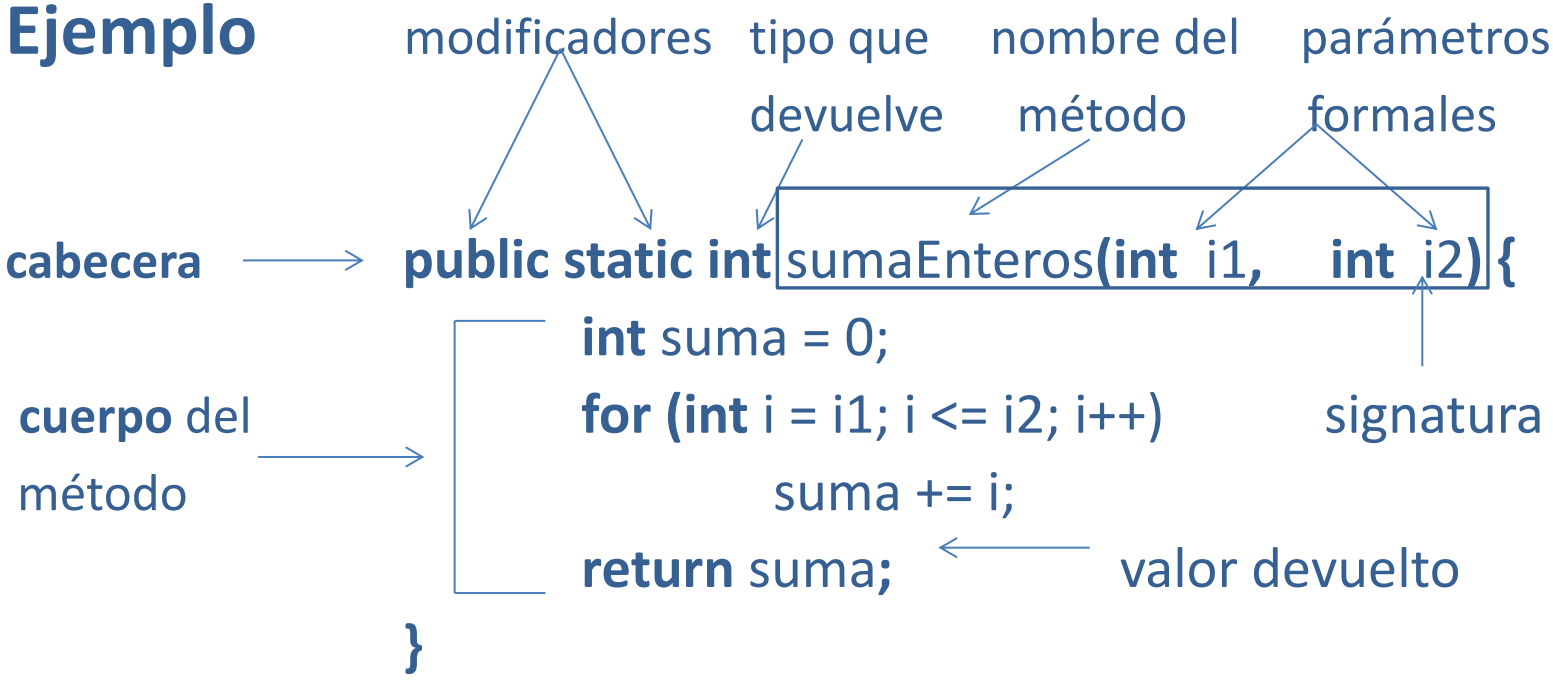
- **Cuerpo** (define la **implementación** del método)

```
{
        // Declaraciones de variables
        ...
        // Sentencias ejecutables
        ...
        // Devolución de un valor (opcional)
        ...
}
```

- En el cuerpo del método se **implementa el algoritmo** necesario para realizar la tarea de la que el método es responsable.
- Para devolver valores se utiliza la sentencia **`return`.** Los métodos que no devuelven nada no tendrán esta sentencia.

Métodos

Ejemplo



Nota: sería incorrecto en la cabecera del método una declaración de parámetros del tipo (es decir, cada parámetro debe llevar su tipo):

public static int sumaEnteros**(int** i1, i2**)** **¡¡ERROR!!**

Métodos

Ejemplo. Definición de un método para obtener el valor máximo de dos enteros pasados como parámetros.

```
public static int max(int num1, int num2) {
        int result;
        if (num1 > num2)
                result = num1;
        else
                result = num2;
        return result;
}
```

Métodos

Invocación de un método - Llamada a un método

Para usar un método, es necesario **llamarlo** o **invocarlo**. Hay dos formas de llamar a un método, dependiendo de si devuelve o no un valor.

- Si el método devuelve un valor, la llamada al método normalmente se trata como un valor. Por ejemplo:

```
int maximo = max(3, 4);
```

parámetros actuales o reales

nombre del método

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

- Si el método no devuelve ningún valor (`void`), la llamada se hará dando únicamente el identificador del método. Por ejemplo:

```
mostrarMenu();
```

Métodos

Plantilla de un programa con uso de métodos

```
package org.ip.sesionNN;
public class IdentificadorClase {
        // Definición de métodos
        public static tipo identificadorMetodo(parámetros) {
                // Declaraciones de variables
                ...
                // Sentencias ejecutables
                ...
                // Devolución de un valor (opcional)
        }
        public static void main(String[] args) {
                .......
                // Llamada o invocación al método
        }
}
```

Métodos

El programa para obtener la suma entre dos números enteros, utilizando un **método**

```
1  package org.ip.tema01;
2
3  public class Suma {
4
5      public static int sumaEnteros(int i1, int i2) {
6            int suma = 0;
7            for (int i = i1; i <= i2; i++)
8          suma += i;
9            return suma;
10      }
11
12     public static void main(String[] args) {
13           System.out.println(" La suma de 1 a 10 es  " + sumaEnteros(1, 10));
14           System.out.println (" La suma de 20 a 30 es  " + sumaEnteros(20, 30));
15           System.out.println (" La suma de 35 a 45 es  " + sumaEnteros(35, 45));
16     }
17 }
```

Salida

```
La suma de 1 a 10 es  55
La suma de 20 a 30 es  275
La suma de 35 a 45 es  440
```

Métodos

El programa **sin utilizar métodos** para obtener el máximo de dos números enteros

```
package org.ip.sesion04;
public class MayorValor {
        public static void main(String[] args) {
                int num1 = 9, num2 = 2, result;
                if (num1 > num2)
                        result = num1;
                else
                        result = num2;
                System.out.println("El maximo entre  " + num1 + "  y " +
                        num2 +  " es " + result);
        }
}
```

Métodos

El programa **utilizando un método** quedaría:

```
package org.ip.sesion04;
public class MayorValor {
        public static int max(int num1, int num2) {
                int  result;
                if (num1 > num2)
                        result = num1;
                else
                        result = num2;

                return  result;
        }
        public static void main(String[] args) {
                int  i = 9,  j = 2;
                int  k = max(i,  j);
                System.out.println("El maximo entre  " + i +
                 "  y "+  j +  " es " + k );
        }
}
```

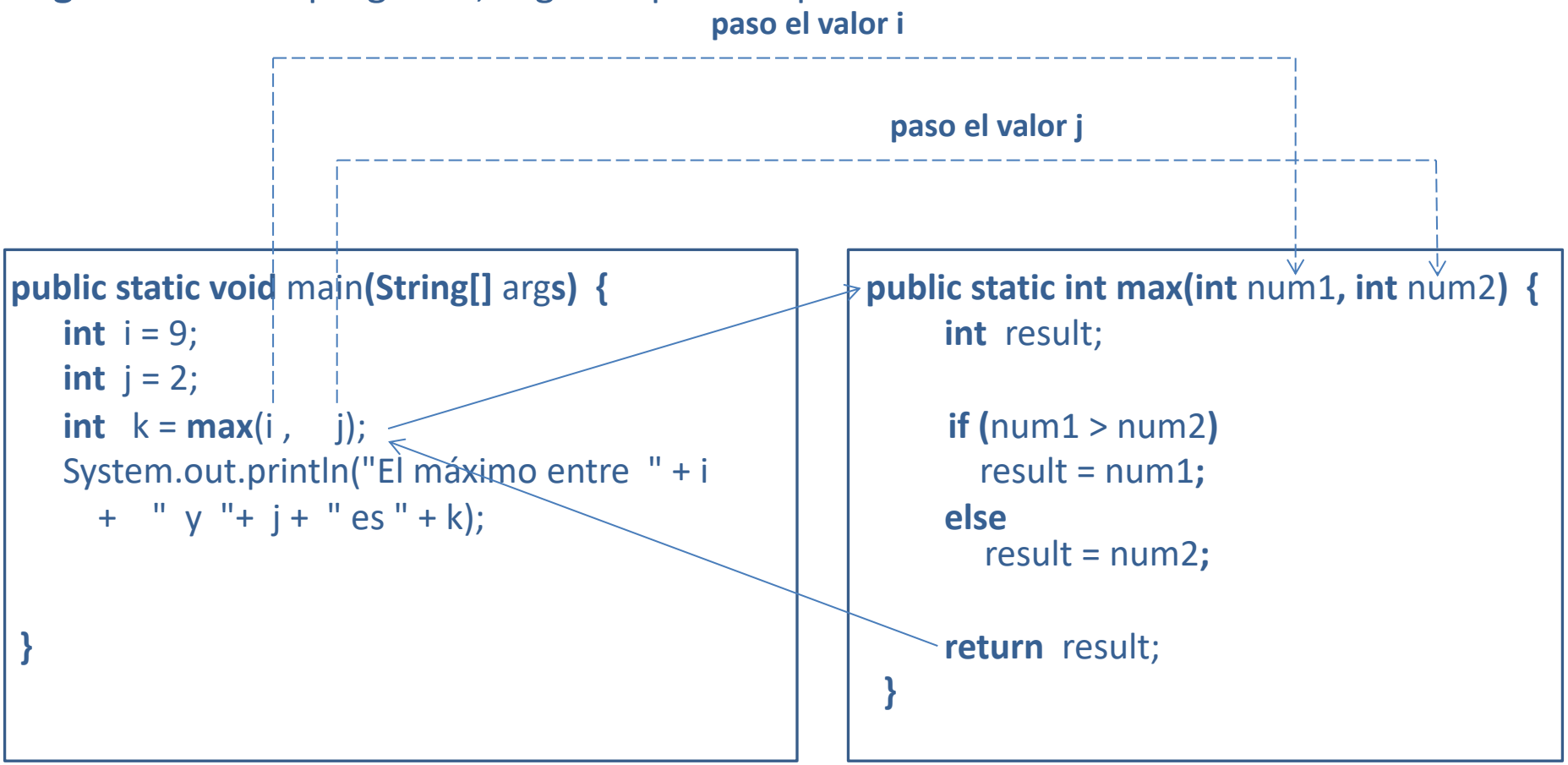
parámetros formales ……

parámetros actuales o reales ……

Correspondencia: nº, posición y tipo

Métodos

Seguimiento del programa, según el paso de parámetros



Métodos

Ejemplo. Suma de enteros de 1 a 10, de 20 a 30 de 35 a 45

```
package org.ip.sesion04;
public class Suma {
        public static int sumaEnteros(int i1, int i2) {
            int suma = 0;
            for (int i = i1; i <= i2; i++)
                    suma += i;
            return suma;
         }

        public static void main(String[] args) {
            System.out.println(" La suma de 1 a 10 es  " + sumaEnteros(1, 10));
            System.out.println (" La suma de 20 a 30 es  " + sumaEnteros(20, 30));
            System.out.println (" La suma de 35 a 45 es  " + sumaEnteros(35, 45));
        }
}
```

Métodos

Ejemplo. Algoritmo de Euclides **sin uso de métodos**

```
package org.ip.sesion03;

public class Euclides {
    /**
     * Calcula el máximo común divisor de dos enteros positivos utilizando el
     * algoritmo de Euclides
     *
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        int dato1 = 25;
        int dato2 = 10;
        int aux;
        while (dato1 % dato2 != 0) {
            aux = dato1;
            dato1 = dato2;
            dato2 = aux % dato2;
        }
        System.out.println("El MCD de los valores introducidos es " + dato2);
    }
}
```

Métodos

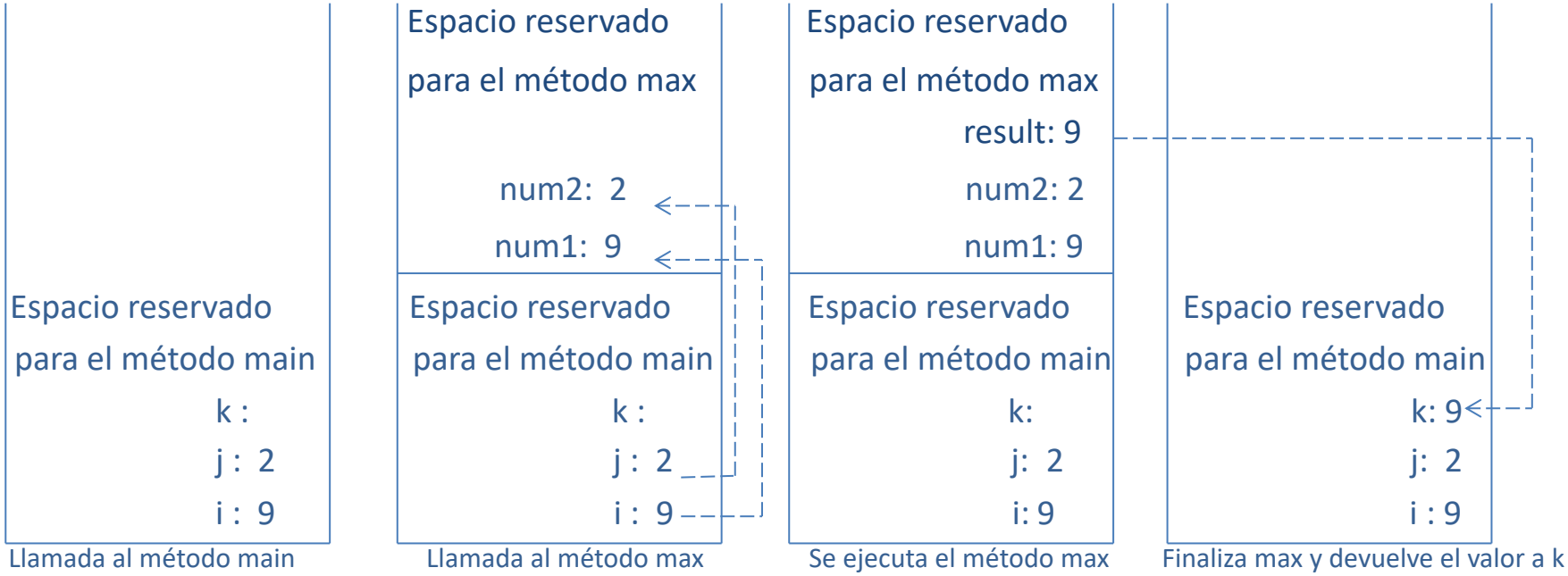
Ejemplo. Algoritmo de Euclides **con uso de métodos**

```
package org.ip.sesion04;

public class Euclides {
    /**
     * Calcula el máximo común divisor de dos enteros positivos utilizando el
     * algoritmo de Euclides
     *
     * @param args
     */
    public static int mcdEuclides(int dato1, int dato2) {
        int aux;
        while (dato1 % dato2 != 0) {
            aux = dato1;
            dato1 = dato2;
            dato2 = aux % dato2;
        }
        return dato2;
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int dato1 = 25;
        int dato2 = 10;
        System.out.println("El MCD de " + dato1 + " y " + dato2 + " es " + mcdEuclides(dato1, dato2));
    }
}
```

Métodos

Estado de la memoria

Cuando se hace una **llamada a un método**, el sistema almacena los parámetros y las variables en una zona de memoria conocida como **pila** o **stack** (último elemento en entrar, primero en salir). En la figura se muestra el funcionamiento para el programa que calcula el máximo de dos enteros



Métodos

❑ Paso de parámetros

Cuando hacemos una llamada a un método o función necesitamos proporcionarle parámetros. Se relaciona el **parámetro real** o actual (en la llamada) con el **parámetro formal** (en el método o función). El ***parámetro real*** (en la llamada) pasa el valor al ***parámetro formal*** (parámetro del método) y si éste se modifica en el método, el ***parámetro real*** no se ve afectado. Estamos haciendo un paso de parámetros **por valor**.

En el ejemplo siguiente el valor de **x** (1) se pasa al *parámetro formal* **n** al hacer la llamada al método `incremento`. En ese método, **n** se incrementa en 1 pero esa modificación **no afecta** para nada a **x**, se ha hecho una **copia** y son posiciones de memoria distintas.

Métodos

Salida

```
Antes de la llamada al metodo, x es 1
n dentro del metodo es 2
Despues de la llamada al metodo, x es 1
```

```java
package org.ip.sesion04;

public class Incremento {

    public static void incremento(int n) {
        n++;
        System.out.println("n dentro del metodo es " + n);
    }

    public static void main(String[] args) {
        int x = 1;
        System.out.println("Antes de la llamada al metodo, x es " + x);
        incremento(x);
        System.out.println("Despues de la llamada al metodo, x es " + x);
    }
}
```

```java
package org.ip.sesion04;

public class TestPasoPorValor {

    /** Intercambia dos variables */
    public static void swap(int n1, int n2) {
        System.out.println("\t*** Dentro del metodo swap ***");
        System.out.println("\t\tAntes del intercambio n1 es " + n1 + " n2 es " + n2);
        // Intercambio n1 con n2
        int temp = n1;
        n1 = n2;
        n2 = temp;
        System.out.println("\t\tDespues del intercambio n1 es " + n1 + " n2 es " + n2);
    }

    public static void main(String[] args) {
        // Declarar e inicializar variables
        int num1 = 1;
        int num2 = 2;

        System.out.println("Antes de la llamada al metodo swap, num1 es " + num1
                + " y num2 es " + num2);

        // Llamada al metodo swap
        swap(num1, num2);

        System.out.println("Despues de la llamada al metodo swap, num1 es " + num1
                + " y num2 es " + num2);
    }
}
```

Salida

```
Antes de la llamada al metodo swap, num1 es 1 y num2 es 2
        *** Dentro del metodo swap ***
                Antes del intercambio n1 es 1 n2 es 2
                Despues del intercambio n1 es 2 n2 es 1
Despues de la llamada al metodo swap, num1 es 1 y num2 es 2
```

Métodos

❑ Valor devuelto por un método - `return`

En lugar de pasar información a un método, es posible el paso de información en sentido contrario, es decir, desde el método o función hacia el código donde se realiza la llamada al método o función. Con esto conseguimos que un **método se convierta en un valor cualquiera** y se puede utilizar desde el lugar donde se invoca.

Es posible utilizar cualquier tipo para especificar que la llamada al método se sustituirá por un valor del tipo indicado. Para ello se dispone de la instrucción `return`, que finaliza la ejecución del método y devuelve el valor indicado a la llamada. Debe existir concordancia entre el tipo del método y el tipo del valor devuelto con la instrucción `return`.

La última instrucción del método debe ser **`return`** (fuerza su fin) y no se ejecutarán instrucciones posteriores. Puede haber más de un **`return`**

Métodos

❑ Valor devuelto por un método

```
1  package org.ip.tema01;
2
3  public class ValorDevuelto {
4
5      // cada llamada se sustituye por un int
6      public static int sumaEnteros(int entero1, int entero2) {
7          int resultado;
8          resultado = entero1 + entero2;
9
10         return resultado;  // sustituye la llamada por el valor especificado
11     }
12
13     public static void main(String[] args) {
14         int suma = sumaEnteros(2, 3);  // se sustituye por 5 que se asigna a suma
15         int valor = sumaEnteros(1, 7) * 5; // se sustituye por 8 que se multiplica por 5
16
17         System.out.println("Suma  = " + suma);
18         System.out.println("Valor = " + valor);
19     }
20 }
```

Métodos

Sobrecarga de métodos

Lenguajes como Java permiten que existan distintos métodos con el mismo nombre siempre, y cuando su signatura no sea idéntica. Esto se conoce como **sobrecarga de métodos**.

Si dos métodos tienen el mismo nombre, pero distinta lista de parámetros (tanto número, como tipo), el compilador de Java determina cual debe utilizar basándose en la **signatura**.

Los métodos sobrecargados pueden devolver distintos tipos, aunque estos no sirven para distinguir un método sobrecargado de otro.

En el ejemplo siguiente tres métodos tienen el mismo nombre, **max**, pero difieren en la lista de parámetros.

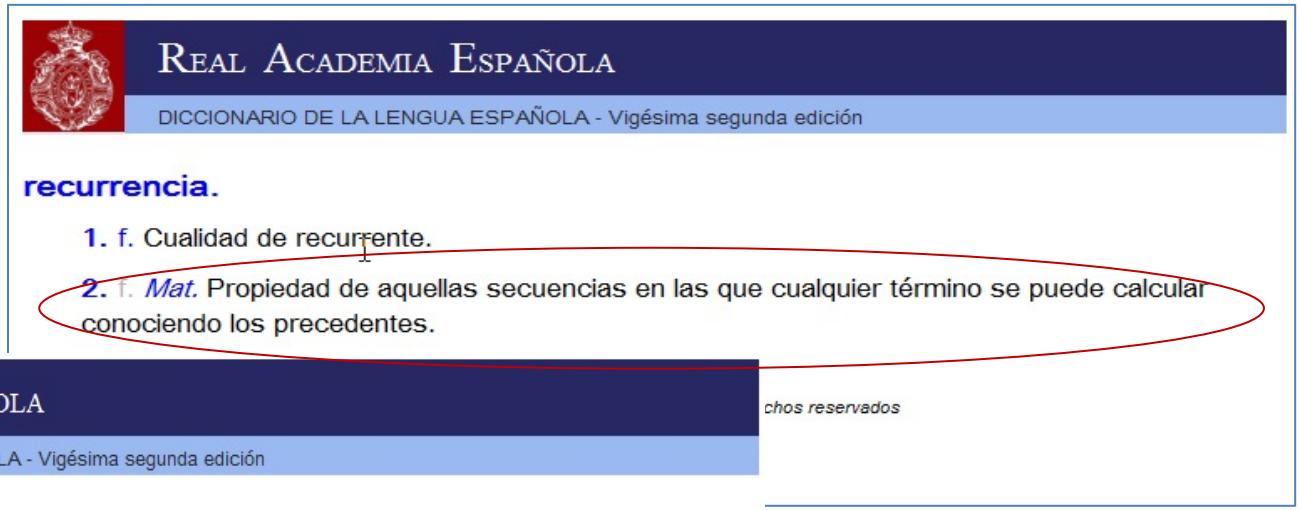
```java
package org.ip.sesion04;

public class TestSobrecargaMetodos {
    /** Devuelve el maximo entre dos valores enteros */
    public static int max(int num1, int num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
    /** Devuelve el maximo entre dos valores reales */
    public static double max(double num1, double num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
    /** Devuelve el maximo entre tres valores reales */
    public static double max(double num1, double num2, double num3) {
        return max(max(num1, num2), num3);
    }

    public static void main(String[] args) {
        System.out.println("El maximo entre 3 y 4 es " + max(3, 4));
        System.out.println("El maximo entre 3.0 y 5.4 es " + max(3.0, 5.4));
        System.out.println("El maximo entre 10.7, 3.8 y 5.1 es " + max(10.7, 3.8, 5.1));
    }
}
```

Salida

```
El maximo entre 3 y 4 es 4
El maximo entre 3.0 y 5.4 es 5.4
El maximo entre 10.7, 3.8 y 5.1 es 10.7
```

❑ Recursión básica

Si consultamos en la Real Academia Española (www.rae.es) lo que se entiende por **recurrencia** y **recurrente** encontramos:



REAL ACADEMIA ESPAÑOLA

DICCIONARIO DE LA LENGUA ESPAÑOLA - Vigésima segunda edición

recurrencia.

1. f. Cualidad de recurrente.
2. f. *Mat.* Propiedad de aquellas secuencias en las que cualquier término se puede calcular conociendo los precedentes.

REAL ACADEMIA ESPAÑOLA

DICCIONARIO DE LA LENGUA ESPAÑOLA - Vigésima segunda edición

chos reservados

recurrente.

(Del ant. part. act. de *recurrir*; *recurrens, -entis*).

1. adj. Que recurre.
2. adj. Que vuelve a ocurrir o a aparecer, especialmente después de un intervalo.
3. adj. *Anat.* Dicho de un vaso o de un nervio: Que en algún lugar de su trayecto vuelve hacia el origen.
4. adj. *Mat.* Dicho de un proceso: Que se repite.
5. com. Persona que entabla o tiene entablado un recurso.

Hablamos de **recurrencia**, cuando definimos algo en función de sí mismo (una propiedad, un tipo de objeto, una operación, etc.). La recurrencia aparece de forma natural en algunas definiciones o problemas matemáticos. Por ejemplo:

✓ Los **números naturales** se definen como:

- 0 es un nº natural
- sucesor(x) es un nº natural si x lo es.

✓La **potencia con exponentes enteros** se puede definir como:

- $a^0 = 1$
- $a^n = a \cdot a^{(n-1)}$

✓El **factorial** de un entero positivo se define como:

- $0! = 1$
- $n! = n \cdot (n-1)!$

Diremos que una **definición recurrente** es aquella en la que se define algo en términos de *versiones más pequeñas de sí mismo*. En programación supone la posibilidad de que un **método se llame a sí mismo** (diremos que el método es recursivo) y también supone el poder definir tipos de datos recursivos. Prácticamente todos los lenguajes de programación tienen esa posibilidad y para otros (los funcionales) es la herramienta por excelencia.

Definiremos un **algoritmo recurrente** o basado en relaciones de recurrencias como aquel que *se expresa en términos de instancias más pequeñas de sí mismo y un caso base*. Por tanto, un algoritmo recurrente tendrá:

- ✓ **Un caso base**. Caso para el que la solución puede establecerse de forma no recurrente. Dará la *condición de terminación*.
- ✓ **Un caso general**. También llamado caso recurrente, para el que la solución se expresa en términos de una versión más pequeña de sí mismo.

Importante:

Siempre que planteemos una solución recurrente a un problema deberemos asegurarnos de tener el **caso base** para el cual la evaluación no es recurrente y, además, cada llamada se realiza con un valor más pequeño para conseguir llegar al caso base o a la condición de terminación.

Ejemplo. Define un método recursivo que permita calcular el **factorial** de un entero no negativo.

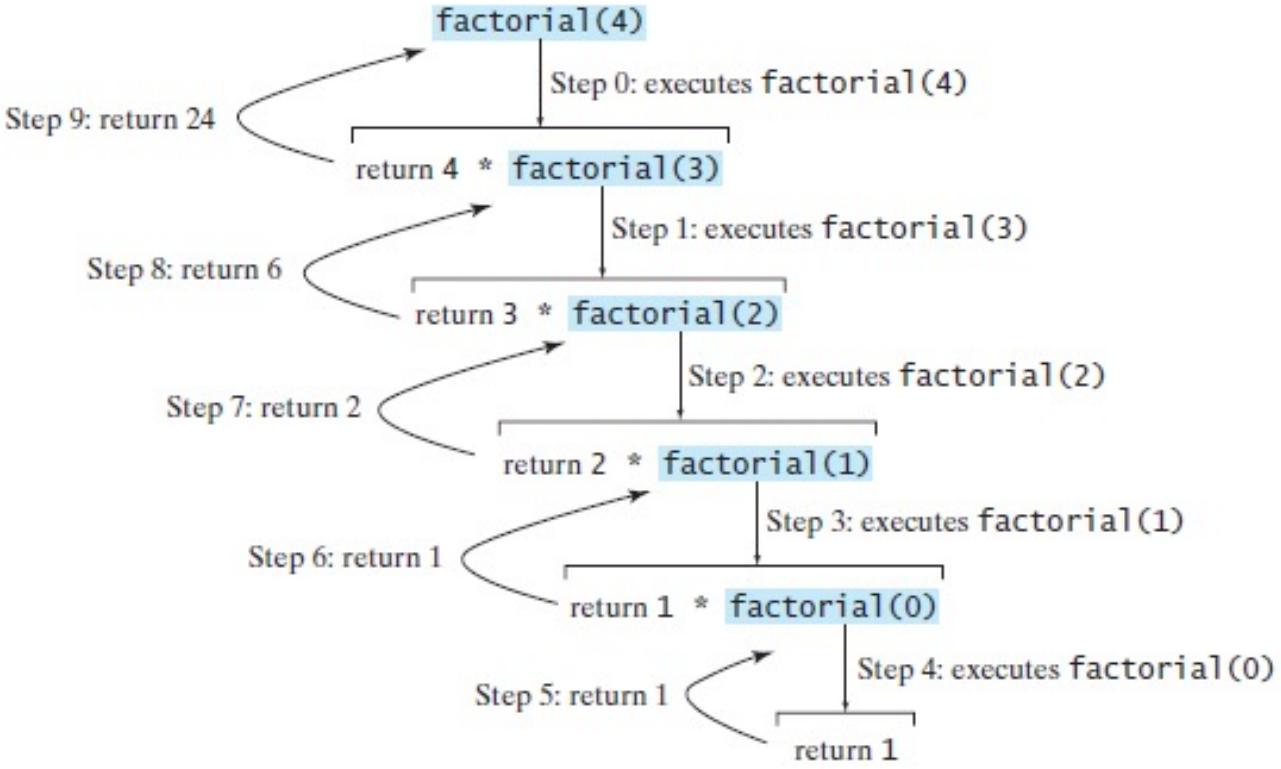
```
public static long  factorialRecursivo(int n) {
        if (n == 0)                          <-- Caso base
            return 1;
        else
            return n * factorialRecursivo(n - 1) ;   <-- Llamada recursiva
}
```

```java
package org.ip.tema01;

public class Factorial {

    // Metodo iterativo para calcular el factorial
    public static long factorialIterativo(int n){
        if (n == 0) return 1;
        long fact = 1;
        int i = 1;
        while (i <= n) {
            fact = fact * i;
            i++;
        }
        return fact;
    }

    // Metodo recursivo para calcular el factorial
    public static int factorialRecursivo(int n){
        if (n == 0)
            return 1;
        else
            return n * factorialRecursivo(n - 1);
    }

    public static void main(String[] args) {
        System.out.println("El factorial recursivo de 7 es " + factorialRecursivo(7));
        System.out.println("El factorial iterativo de 7 es " + factorialIterativo(7));
    }
}
```

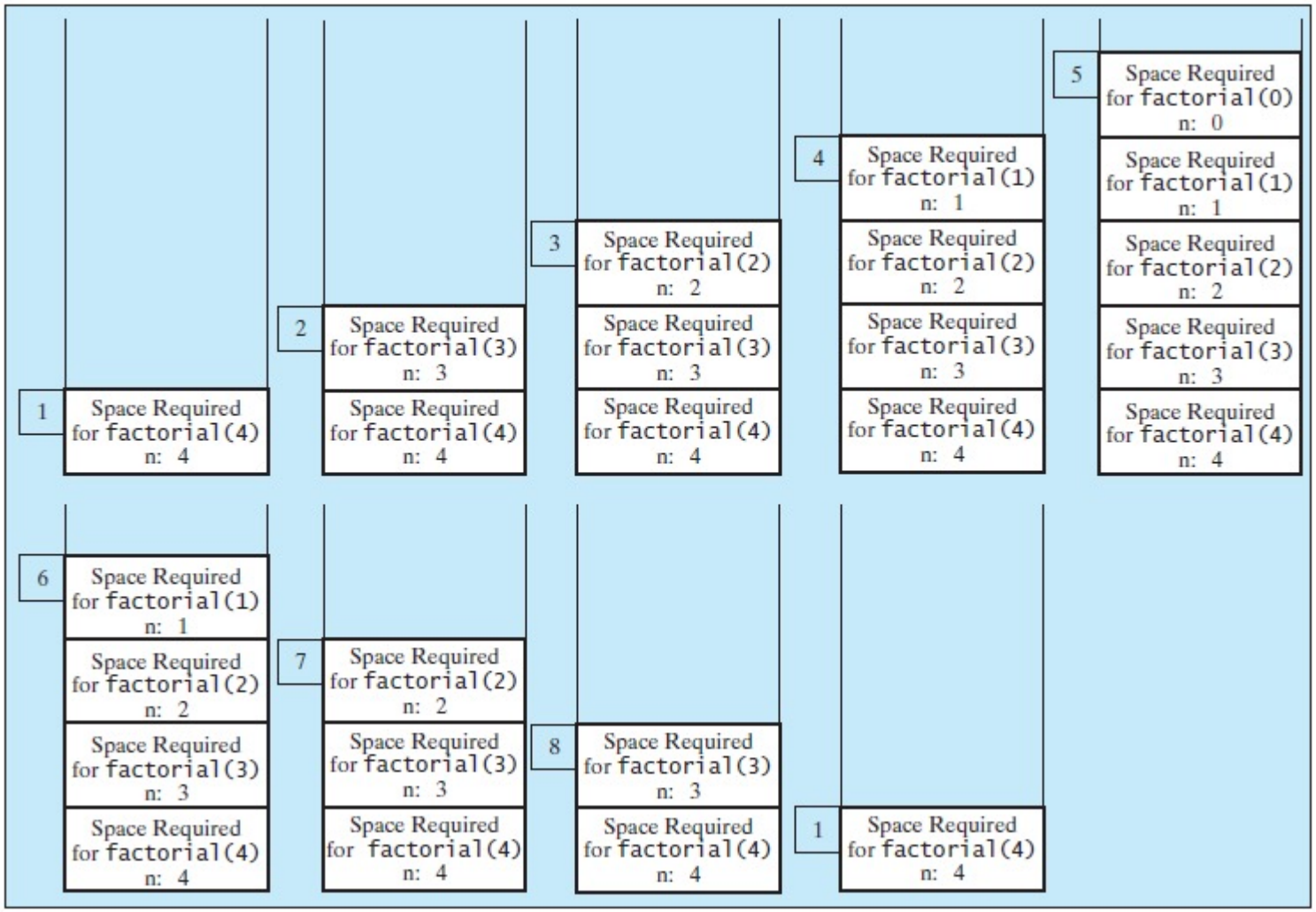
Método erróneo

```java
// Metodo recursivo erroneo para calcular el factorial
public static int factorialRecursivoErroneo(int n){
    return n * factorialRecursivoErroneo(n - 1);
}
```

Seguimiento de la llamada



Estado de la **pila**



Ejemplo. Define un método recursivo que permita obtener el número de **Fibonacci** para un índice dado.

La **serie de Fibonacci** comienza con 0 y 1, y cada número siguiente es la suma de los dos que le preceden. Es decir:

Serie	0	1	1	2	3	5	8	13	21	34	55	89	...
índice	0	1	2	3	4	5	6	7	8	9	10	11	

Se puede definir recursivamente:

fibonacci(0) = 0;
fibonacci(1) = 1; } **Caso base**

fibonacci(indice) = fibonacci(indice – 2) + fibonacci(indice – 1); **Caso general**

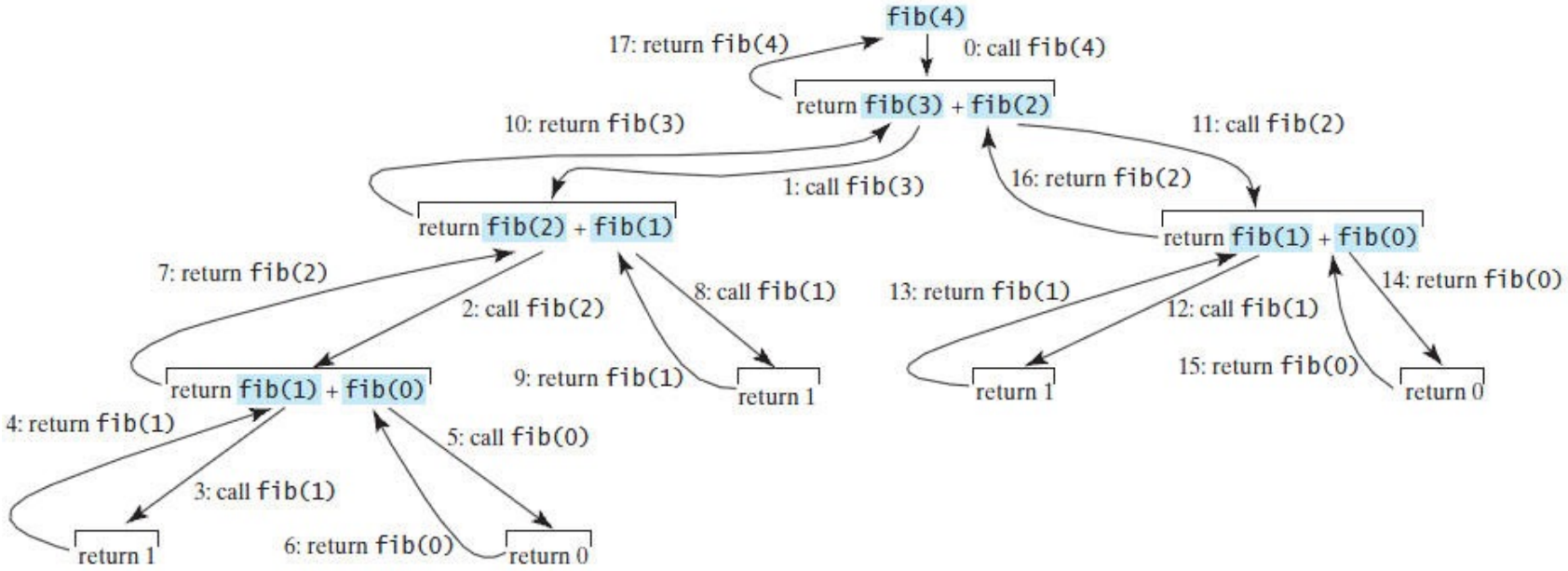
```
public static long  fibonacci(long n) {
        if (n == 0)         //caso base
            return 0;
        else if (n == 1)  // caso base
             return 1;
        else
             return fibonacci(n – 1) + fibonacci(n – 2); }
```

```java
package org.ip.tema01;

import java.util.Scanner;

public class CalcularFibonacci {

    // Método recursivo para calcular el numero de fibonacci
    public static long fibonacci(int n) {
        if (n == 0)          // Caso base
            return 0;
        else if (n == 1)     // Caso base
            return 1;
        else
            // Reduccion y llamada recursiva
            return fibonacci(n - 1) + fibonacci(n - 2);
    }

    public static void main(String[] args) {
        @SuppressWarnings("resource")
        Scanner entrada = new Scanner(System.in);
        System.out.print("Introduce el indice para obtener el numero de Fibonacci: ");
        int indice = entrada.nextInt();
        System.out.println("El numero de Fibonacci para el indice " + indice
                + " es " + fibonacci(indice));
        int terminos; // nº de términos a mostrar
        System.out.print("Indica el numero de términos que desea mostrar de la serie de Fibonacci: ");
        terminos = entrada.nextInt();
        for (int i = 0; (i < terminos); i++) {
            System.out.print(fibonacci(i) + "\t");
        }
    }
}
```

Salida

```
Introduce el indice para obtener el numero de Fibonacci: 7
El numero de Fibonacci para el indice 7 es 13
Indica el numero de términos que desea mostrar de la serie de Fibonacci: 10
0       1       1       2       3       5       8       13      21      34
```

Seguimiento de una llamada al método recursivo fibonacci mostrando el orden de las llamadas.



Observamos que hay muchas llamadas recursivas repetidas. Por ejemplo, fib(2) se llama 2 veces, fib(1) 3 veces esto va a suponer más consumo de tiempo y memoria. Este es un claro ejemplo de una implementación del problema sencilla de entender por la propia definición, pero poco eficiente. Resultaría más eficiente la solución **iterativa**.

Ejemplo de recursividad. Cálculo de la potencia de dos enteros, base^{exponente}

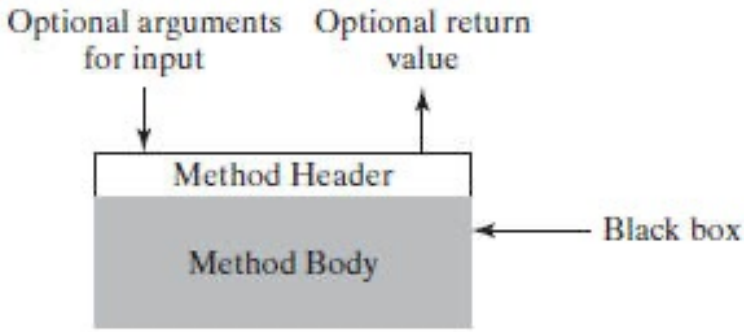
```java
package org.ip.tema01;

public class Potencia {

    public static double potencia(int base, int exponente) {
        if (exponente == 0) {          // caso base
            return 1.0;                // base^0 = 1
        } else if (exponente == 1) {   // caso base
            return base;               // base^1 = base
        } else if (exponente < 0) {    // Exponente negativo
            return potencia(base, exponente + 1) / base;
        } else {                       // Exponente positivo
            return base * potencia(base, exponente - 1);
        }
    }

    public static void main(String[] args) {
        int base = 10;
        int exponente1 = 4;
        int exponente2 = -3;

        System.out.println(base + "^" + exponente1 + " = " + potencia(base, exponente1));
        System.out.println(base + "^" + exponente2 + " = " + potencia(base, exponente2));
    }
}
```

Salida

```
10^4 = 10000.0
10^-3 = 0.001
```

Abstracción de métodos y refinamiento por pasos

La clave para el desarrollo de software es aplicar el concepto de *abstracción*. Aprenderemos a lo largo del curso distintos niveles de abstracción. La **abstracción de métodos** se alcanza separando el uso del método de su implementación. El usuario o cliente utilizará el método sin necesidad de tener ningún conocimiento de cómo ha sido implementado. Los detalles de implementación se encapsulan en el método y son ocultos para el usuario que invoca el método. Esto se conoce como **ocultación de la información** o **encapsulamiento**. Si decidiese cambiar la implementación, el usuario del programa no se debe ver afectado porque no se cambiará la signatura del método.



El concepto de **abstracción de métodos** se puede aplicar al proceso de desarrollo de programas. Cuando escribimos un programa largo, se usa la estrategia *divide y vencerás,* también conocida como **refinamiento por pasos**, para descomponer el problema en *sub-problemas*. Los sub-problemas pueden volverse a descomponer en otros más pequeños para hacerlos más manejables.

Supongamos que queremos hacer un programa que muestre el **calendario para un mes y un año dado**. El programa pedirá al usuario que introduzca el año y el mes y se mostrará el calendario como sigue:

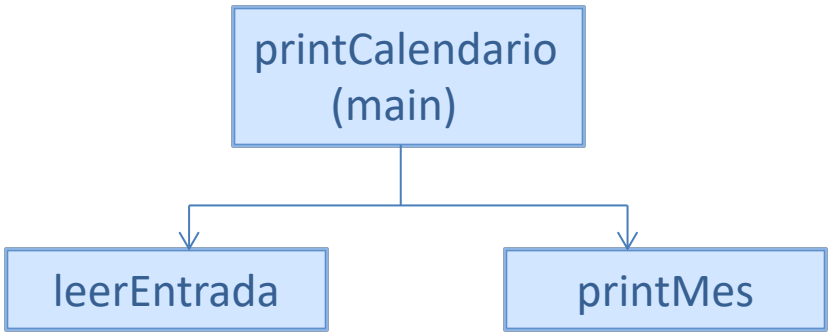
```
Introduzca un anio completo (e.g., 2001): 2021
Introduzca un mes como un numero entre 1 y 12: 10
         Octubre 2021
---------------------------
 Lun Mar Mie Jue Vie Sab Dom
                   1   2   3
   4   5   6   7   8   9  10
  11  12  13  14  15  16  17
  18  19  20  21  22  23  24
  25  26  27  28  29  30  31
```

Diseño descendente

¿Cómo podríamos empezar el programa? ¿Empezaríamos inmediatamente a escribir código? Seguramente que los programadores principiantes comenzarían teniendo en cuenta todos los detalles, sin embargo, esos detalles son importantes, pero solo al final del programa. Para hacer más sencilla la solución del problema utilizamos la **abstracción de métodos** y los detalles se implementan más tarde.

En este ejemplo, el problema se divide en dos **sub-problemas**:

- Obtener la entrada del usuario
- Mostrar el calendario para el mes

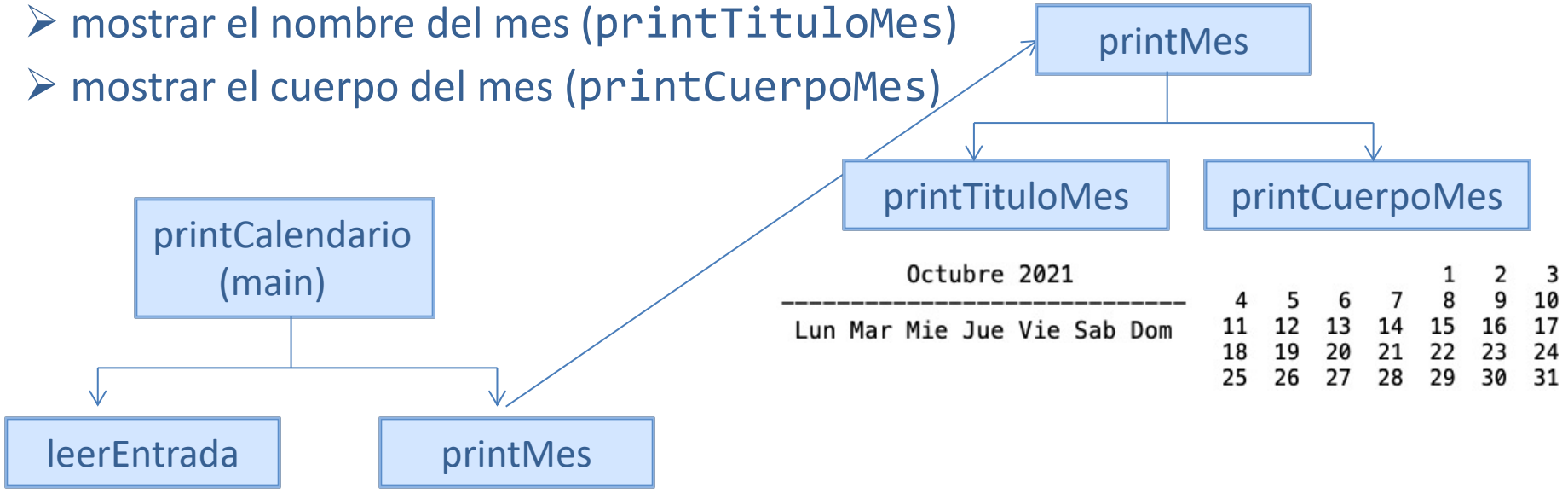



```
Introduzca un anio completo (e.g., 2001): 2021
Introduzca un mes como un numero entre 1 y 12: 10
```

```
                Octubre 2021
 ----------------------------
Lun Mar Mie Jue Vie Sab Dom
                  1   2   3
  4   5   6   7   8   9  10
 11  12  13  14  15  16  17
 18  19  20  21  22  23  24
 25  26  27  28  29  30  31
```

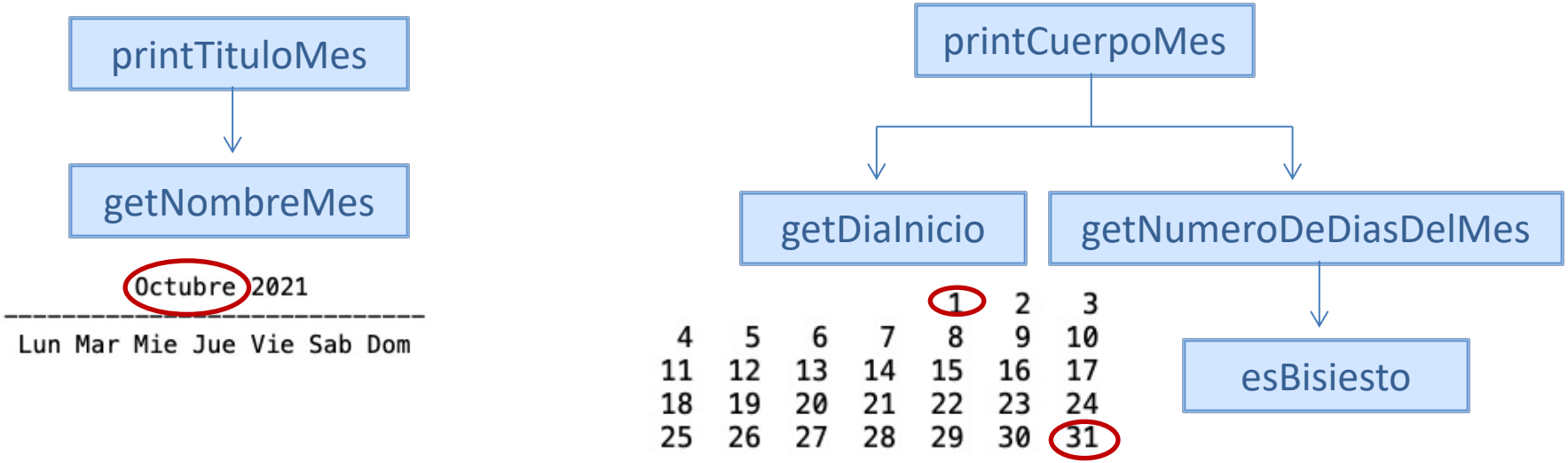
El **sub-problema** de mostrar el calendario (`printMes`) podríamos dividirlo en otros dos **sub-problemas** más sencillos:

- mostrar el nombre del mes (`printTituloMes`)
- mostrar el cuerpo del mes (`printCuerpoMes`)



El titulo del mes consistiría en tres líneas: mes y año, una línea ---------, y el nombre de los siete días de la semana. Necesitaríamos obtener el nombre del mes (por ejemplo, Enero) a partir de un valor numérico del mes (por ejemplo, 1). Esto significa que podríamos hacer `getNombreMes` para mostrar dicho nombre

Es decir,

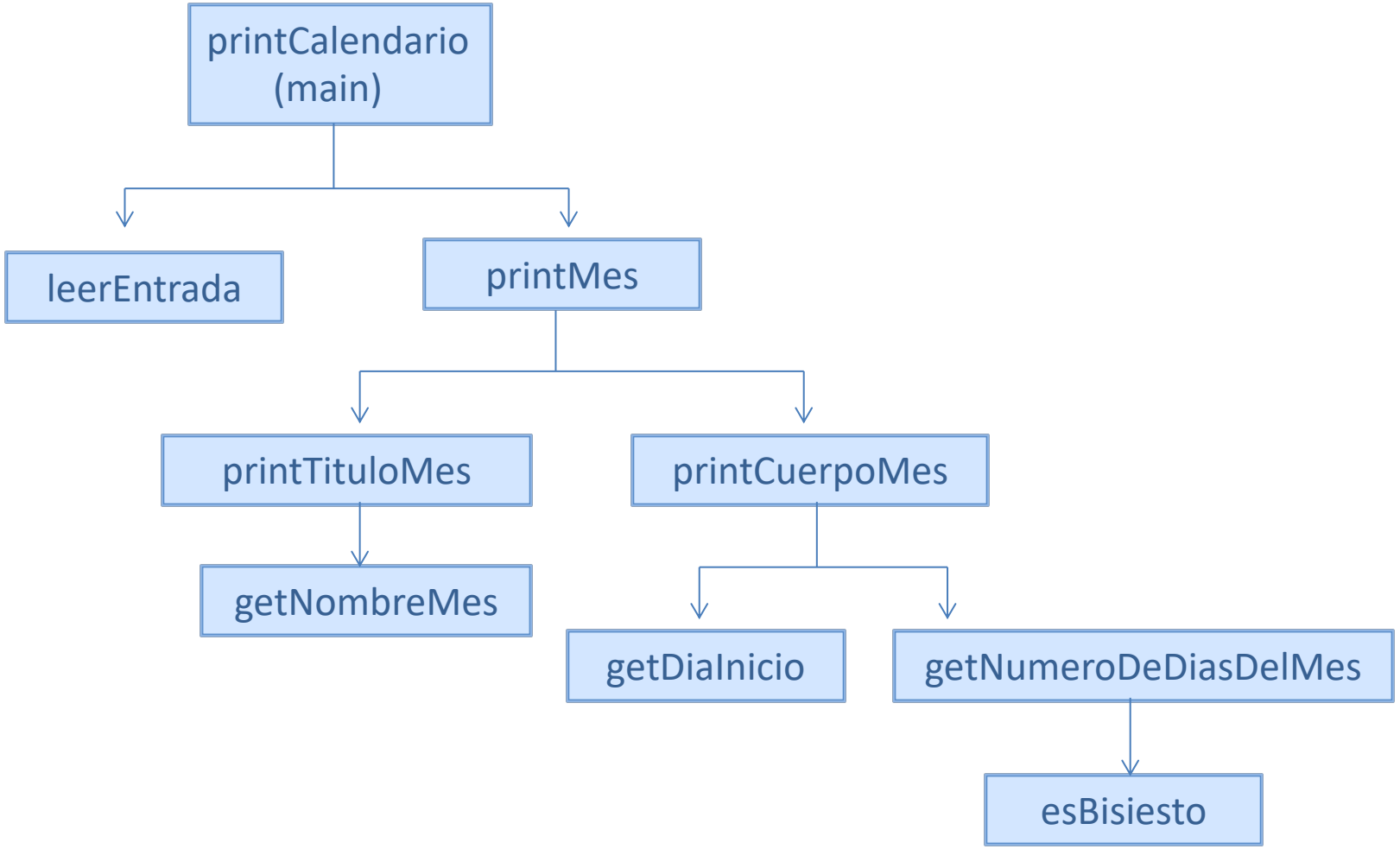


Para mostrar el cuerpo del mes, necesitaríamos conocer:

- El día de la semana en el que empieza el mes (lunes, martes, etc.), `getDiaInicio`.
- El número de días que tiene el mes, `getNumeroDeDiasDelMes`. A su vez, para obtener dicho número, necesitamos saber si el año es bisiesto.

Ejemplo. Diciembre de 2005 tiene 31 días y el 1 de Diciembre fue jueves.

La estructura completa de la división en **sub-programas** quedaría:



Implementación de arriba a abajo (top-down)

Ahora tendríamos que centrar la atención en la implementación. En general, un **sub-problema** corresponde a un **método**, incluso algunos son tan simples que no son necesarios. Tendremos que decidir qué sub-problemas se implementan como métodos y cuales se incluyen en otros métodos. Por ejemplo, el sub-problema `leerEntrada` puede implementarse en el método `main`.

De cada método haríamos una versión simple e incompleta del mismo. Esto permite construir rápidamente un esqueleto del programa. Implementamos el método `main` y usamos los sub-problemas para los métodos y el programa tendría un aspecto:

```
package org.ip.tema01;
import java.util.Scanner;
public class PrintEsqueletoCalendario {
  /** Método main*/
    public static void main(String[] args) {
            Scanner entrada = new Scanner(System.in);
            // El usuario introduce el anio
            System.out.print("Introduzca un anio completo (e.g., 2001): ");
             int anio = entrada.nextInt();
```

```
  // El usuario introduce el mes
  System.out.print("Introduzca un mes como un numero entre 1 y 12: ");
  int mes = entrada.nextInt();
  // Muestra el calendario para el mes y el anio introducidos
  printMes(anio, mes);
}
/** printMes puede parecerse a esto */
public static void printMes(int anio, int mes) {
   // Debe mostrar el titulo del mes y el cuerpo del mes
}
/** printTituloMes puede parecerse a esto */
public static void printTituloMes(int anio, int mes) {
}
/** printCuerpoMes puede parecerse a esto */
public static void printCuerpoMes(int anio, int mes) {
}
```

```
/** getNombreMes puede parecerse a esto */
 public static String getNombreMes(int mes) {
  return "Enero";  // Un valor de ejemplo
 }
 /** getDiaInicio puede parecerse a esto */
 public static int getDiaInicio(int anio, int mes) {
  return 1;  // Un valor de ejemplo
 }
/** getNumeroTotalDeDiasDelMes puede parecerse a esto */
 public static int getNumeroDeDiasDelMes(int anio, int mes) {
  return 31;  // Un valor de ejemplo
 }
 /** esBisiesto puede parecerse a esto*/
 public static boolean esBisiesto(int anio) {
  return true;  // Un valor de ejemplo
 } }
```

¡MUCHAS GRACIAS!

UNIVERSIDAD DE ALMERÍA