Tema 4. Métodos

1. INTRODUCCIÓN

Definición: Un método es un conjunto de instrucciones definidas dentro de una clase, que realizan una determinada tarea y a las que podemos invocar mediante un nombre.

Algunos métodos que hemos utilizado hasta ahora:

- `Math.pow()`
- `Math.sqrt()`
- `Character.isDigit()`
- `System.out.println()`

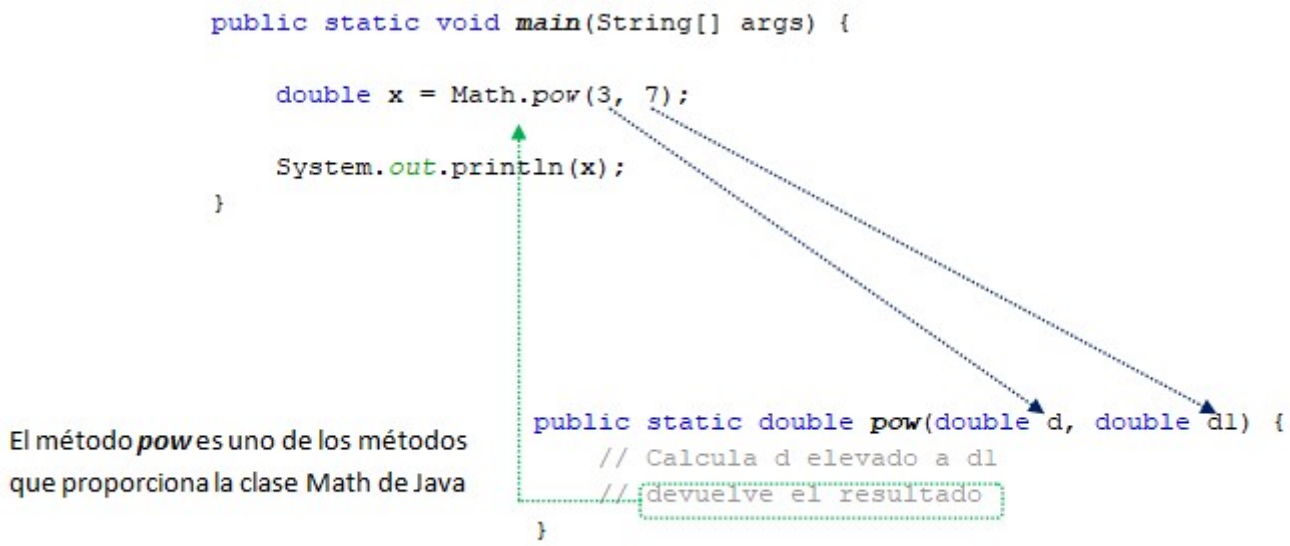
```
public static void main(String[] args) {
    double x = Math.pow(3,7);
    System.out.println(x);
}
```

Llamada al método Math.pow

Llamada al método println

En la imagen vemos la llamada al método *Math.pow* para que realice la operación de elevar 3 a 7 y la llamada al método *println* para mostrar el valor de x.

Cuando se llama a un método, la ejecución del programa pasa al método y cuando éste acaba, la ejecución continúa a partir del punto donde se produjo la llamada.



En la imagen se muestra el flujo de ejecución del programa cuando se produce la llamada al método Math.pow:

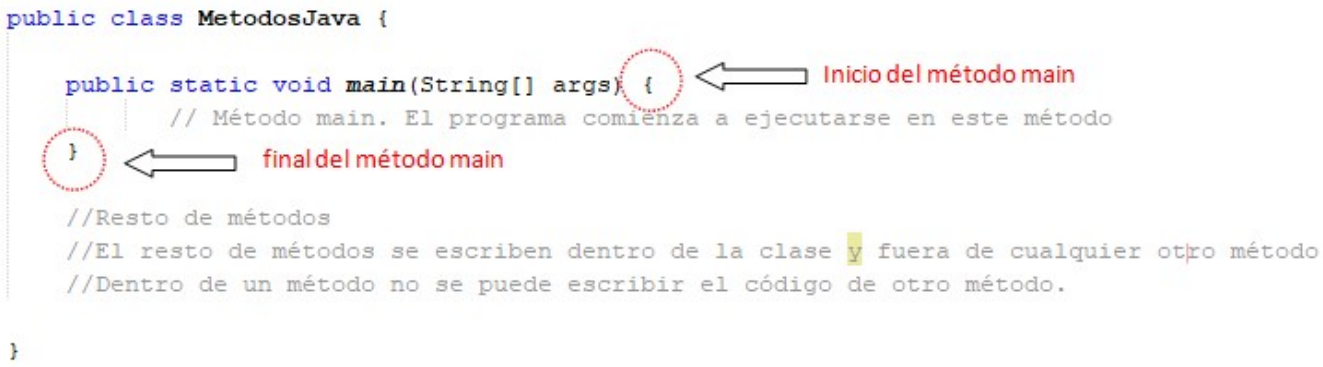
- En la llamada al método *pow* le enviamos los valores con los que tiene que realizar la operación.
- La ejecución del programa continúa dentro del método. Su tarea en este caso es realizar la operación de elevar 3 a 7.
- Cuando termina devuelve el resultado al punto donde se invocó el método. La ejecución del programa continúa en el método *main* desde el punto donde se produjo la llamada.

Utilizando métodos:

- Podemos construir programas modulares.
- Se consigue la **reutilización de código**. En lugar de repetir el mismo código cuando se necesite, por ejemplo para validar una fecha, se hace una llamada al método que lo realiza.

Cuando trabajamos con métodos debemos tener en cuenta lo siguiente:

- Un método siempre pertenece a una clase. **No podemos escribir métodos fuera de una clase**.
- No podemos escribir métodos dentro de otros métodos.
- Todo programa java tiene un método llamado **main**. La ejecución del programa empieza en este método. El método *main* es el punto de entrada al programa y también el punto de salida.
- Un método tiene un único punto de inicio, representado por la llave de inicio {
- La ejecución de un método termina cuando se llega a la llave final } o cuando se ejecuta una instrucción *return*.
- La instrucción *return* puede aparecer en cualquier lugar dentro del método, no tiene que estar necesariamente al final.
- Cuando un método finaliza, la ejecución del programa continúa a partir del punto donde se produjo la llamada al método.
- Desde dentro de un método se puede a su vez invocar a otro método.



2. ESTRUCTURA GENERAL DE UN MÉTODO JAVA

La estructura general de un método Java es la siguiente:

```
[especificadores] tipoDevuelto nombreMetodo([lista parámetros]) [throws listaExcepciones]{
    instrucciones
   [return [expresión];]
}
```

Los elementos que aparecen entre corchetes son opcionales.

especificadores (opcional): determinan el tipo de acceso al método. Se verán en detalle más adelante durante el curso.

tipoDevuelto: indica el tipo del valor que devuelve el método. En Java es imprescindible que en la declaración de un método, se indique el tipo de dato que devuelve. El dato se devuelve mediante la instrucción return. Si el método no devuelve ningún valor el tipo devuelto será *void*.

nombreMetodo: es el nombre que le hemos dado al método. Para crearlo hay que seguir las mismas normas que para crear identificadores.

Lista de parámetros (opcional): después del nombre del método y siempre entre paréntesis puede aparecer una lista de parámetros, también llamados **argumentos**, separados por comas. Estos parámetros son los datos de entrada que recibe el método para operar con ellos. Un método puede recibir cero o más argumentos. Se debe especificar para cada argumento su tipo. **Los paréntesis a continuación del nombre del método son obligatorios** aunque estén vacíos.

throws listaExcepciones (opcional): indica las excepciones que puede generar y manipular el método. Las excepciones las veremos más adelante durante el curso.

return: Se utiliza para devolver un valor a quien ha llamado al método.

La palabra clave *return* va seguida de una expresión que será evaluada para saber el valor de retorno. Esta expresión puede ser una variable de tipo primitivo, una constante, un array, un String, un objeto, etc.

El tipo del valor devuelto debe coincidir con el *tipoDevuelto* que se ha indicado en la declaración del método.

Si el método no devuelve nada (tipoDevuelto = void) la instrucción return es opcional. Si se escribe no irá seguida de ningún valor o expresión ya que el método no devuelve nada. En este caso *return* indicará únicamente el final del método.

La instrucción return puede aparecer en cualquier lugar dentro del método.

La ejecución de un método termina cuando se llega a su llave final } o cuando se ejecuta la instrucción return.

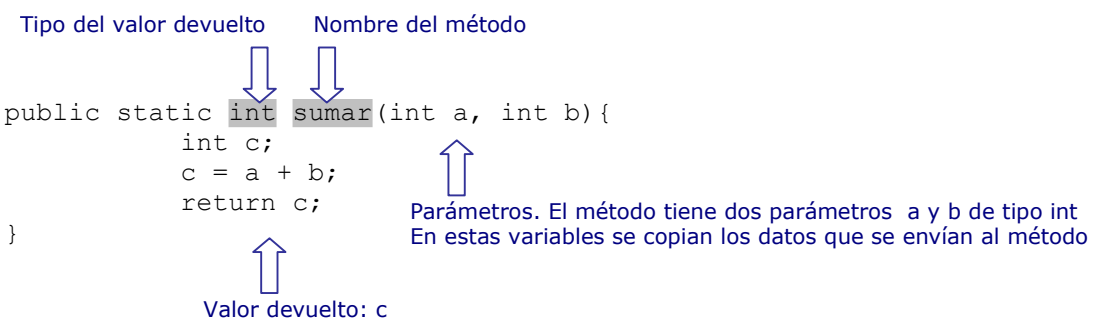
3. IMPLEMENTACIÓN DE MÉTODOS

Pasos para implementar un método:

1. Describir lo que el método debe hacer.
2. Determinar las entradas del método, es decir, lo que el método recibe.
3. Determinar los tipos de datos de las entradas.
4. Determinar lo que debe devolver el método y el tipo del valor devuelto.
5. Escribir las instrucciones que forman el cuerpo del método.
6. Prueba del método: diseñar distintos casos de prueba.

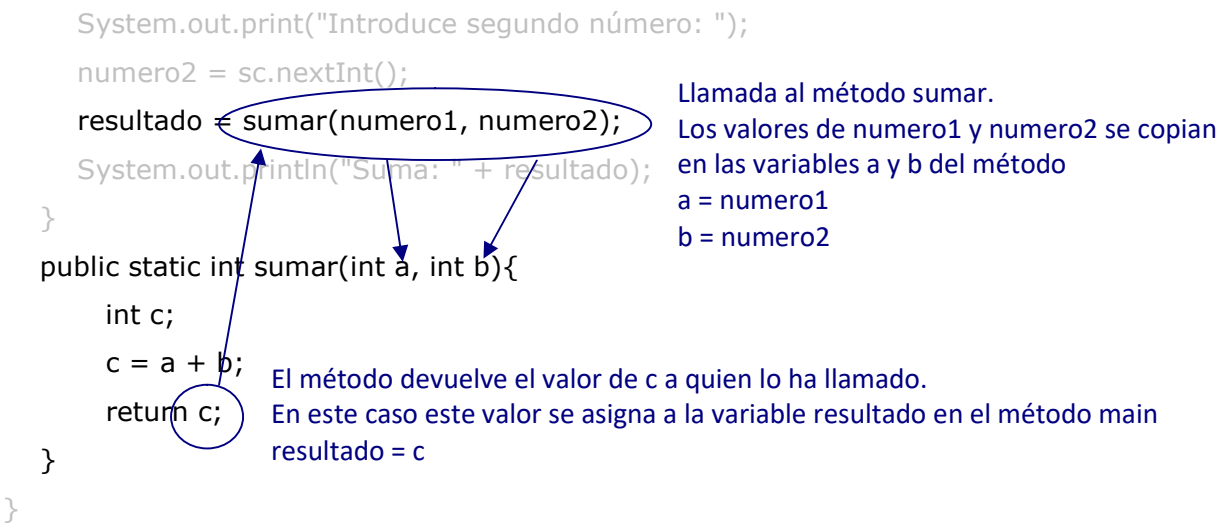
Ejemplo: método llamado *sumar* que recibe dos números enteros y calcula y devuelve su suma.

```
import java.util.Scanner;

public class Metodos1 {
    public static void main(String[] args) {  //inicio del método main
        Scanner sc = new Scanner(System.in);
        int numero1, numero2, resultado;
        System.out.print("Introduce primer número: ");
        numero1 = sc.nextInt();
        System.out.print("Introduce segundo número: ");
        numero2 = sc.nextInt();
        resultado = sumar(numero1, numero2);  //llamada al método sumar
        System.out.println("Suma: " + resultado);
    } //fin del método main

    //método sumar
    public static int sumar(int a, int b){
           int c;
           c = a + b;
           return c;
    }
}
```



El flujo de ejecución del programa es el siguiente:

1. En el método *main* cuando se produce la llamada al método sumar, los valores de las variables numero1 y numero2 se copian en las variables a y b respectivamente. Las variables a y b son los parámetros del método.
2. El flujo de ejecución del programa pasa al método sumar.
3. El método suma los dos números y guarda el resultado en c.
4. El método devuelve mediante la instrucción *return* la suma calculada.
5. Finaliza la ejecución del método.
6. El flujo de ejecución del programa continúa en el método main a partir de dónde se produjo la llamada al método sumar.
7. El valor devuelto por el método es lo que se asigna a la variable *resultado* en main.



Si un método devuelve un valor, la llamada al método puede estar incluida en una expresión que utilice el valor devuelto.

En el ejemplo anterior, el método sumar devuelve un valor entero que hemos guardado en la variable *resultado*. Después solo vamos a utilizar esta variable para mostrarla por pantalla. En este caso podemos hacer la llamada al método dentro de la instrucción System.out.println para que muestre el valor devuelto por el método. El método main quedaría así:

```
public static void main(String[] args) {  //inicio del método main
        Scanner sc = new Scanner(System.in);
        int numero1, numero2, resultado;
        System.out.print("Introduce primer número: ");
        numero1 = sc.nextInt();
        System.out.print("Introduce segundo número: ");
        numero2 = sc.nextInt();

        //Llamada al método dentro de System.out.println
        System.out.println("Suma: " + sumar(numero1, numero2));

} //fin del método main
```

Ejemplo: Programa que lee por teclado un año y calcula y muestra si es bisiesto.

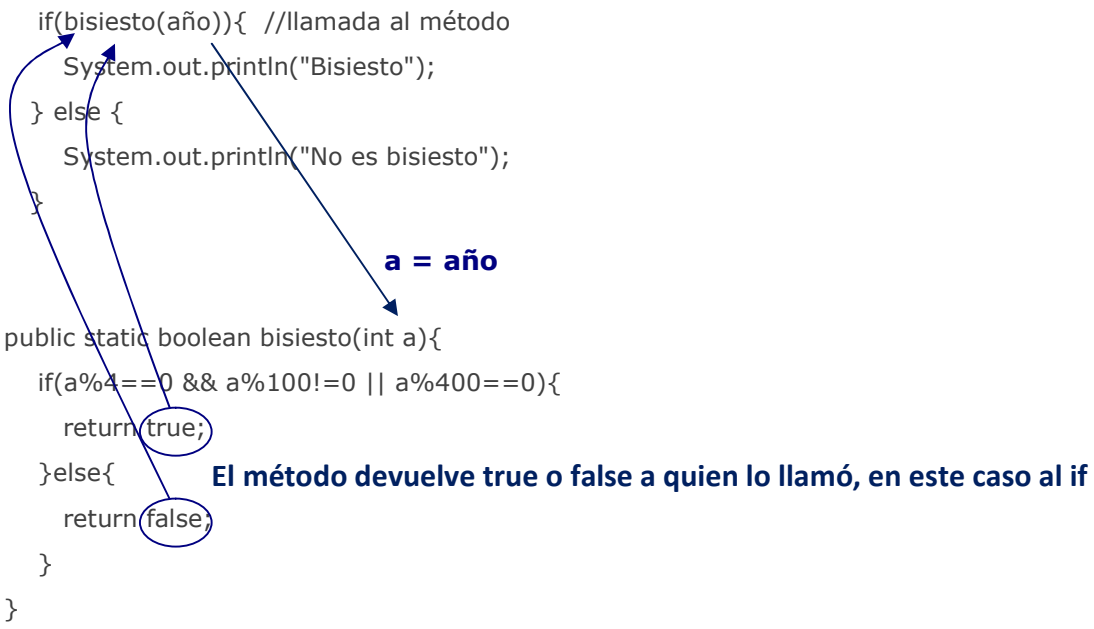
Para realizar el cálculo se utiliza un **método llamado *esBisiesto***. El método esBisiesto tiene un parámetro de tipo entero llamado *a*. Será el encargado de recibir el valor del año a comprobar si es bisiesto o no. El método devuelve un valor de tipo boolean. Devuelve true si el año recibido es bisiesto y false si no lo es.

Este método además es un ejemplo de método con varios *return*.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int año;
        System.out.print("Introduce año: ");
        año = sc.nextInt();

        if(esBisiesto(año)){  //llamada al método
           System.out.println("Bisiesto");
        }else{
           System.out.println("No es bisiesto");
        }
    }
    // método que calcula si un año es o no bisiesto
    public static boolean esBisiesto(int a){
        if(a%4==0 && a%100!=0 || a%400==0){
           return true;
        }else{
           return false;
        }
    }
}
```

Flujo de ejecución:

1. En el método main se llama al método esBisiesto dentro de la instrucción if.
2. El valor de la variable año se copia en la variable a.
3. El flujo de ejecución pasa al método esBisiesto.
4. El método comprueba si el año recibido es o no bisiesto.
5. El método devuelve true si el año es bisiesto o false si no lo es.
6. Finaliza la ejecución del método.
7. El flujo de ejecución continúa en el método *main* en la instrucción *if* donde se produjo la llamada al método.
8. El valor true o false devuelto por el método es lo que determina si la condición es cierta o no.



Ejemplo: Programa que lee una cadena de caracteres por teclado y la muestra por pantalla rodeada por un borde.

Por ejemplo, si se introduce por teclado *Hola*

El programa mostrará por pantalla:

```
########
# Hola #
########
```

Para resolverlo escribiremos un **método llamado *cajaTexto*** que reciba el String introducido por teclado y lo muestre por pantalla rodeado por el borde.

Este es un ejemplo de método que no devuelve nada, solo se limita a mostrar el String por pantalla.

Cuando un método no devuelve nada hay que indicar **void** como tipo devuelto.

Como el método no devuelve nada no es necesario escribir la sentencia return. El método acaba cuando se alcanza su llave final.

```
import java.util.Scanner;
public class MetodoVoid {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String cadena;
        System.out.print("Introduce cadena de texto: ");
        cadena = sc.nextLine();
        cajaTexto(cadena); //llamada al método
    }
    // método que muestra un String rodeado por un borde
    public static void cajaTexto(String str){
            int n = str.length(); //obtenemos la longitud del String
            for (int i = 1; i <= n + 4; i++){ //borde de arriba
                 System.out.print("#");
            }
            System.out.println();
            System.out.println("# " + str + " #"); //cadena con un borde en cada lado
            for (int i = 1; i <= n + 4; i++){ //borde de abajo
                 System.out.print("#");
            }
            System.out.println();
    }
}
```

Ejemplo: Método que recibe un número entero positivo y devuelve la suma de sus cifras. En el método main se introduce un número entero positivo y se envía su valor al método. El método calcula y devuelve la suma de las cifras del número que ha recibido.

```
import java.util.Scanner;
public class EjemploSumaCifras {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n;
        do {
            System.out.print("Introduzca numero > 0: ");
            n = sc.nextInt();
            if (n <= 0) {
                System.out.println("Valor no válido");
            }
        } while (n <= 0);
        System.out.println("Los dígitos de " + n + " suman -> " + sumaDigitos(n));
        //El método se llama desde la instrucción System.out.println
        //Se mostrará por pantalla el valor que devuelva el método
    }
```

```
    //método que calcula la suma de las cifras del número que recibe como parámetro
    public static int sumaDigitos(int numero){
        int suma = 0;
        while(numero != 0){
            suma += numero % 10; //sumamos la cifra de la derecha del número
            numero /= 10; //le quitamos al número la cifra de la derecha
        }
        return suma;
    }
}
```

Para sumar las cifras del número se obtiene su última cifra (la de la derecha) mediante la operación `numero%10` se suma esta cifra y se le quita al número mediante la operación `numero=numero/10`. El proceso se repite mientras el resultado de dividir el número entre 10 sea != 0. Cuando esta operación de como resultado cero significa que al número ya no le quedan más cifras y el proceso finaliza.

Ejemplo: Vamos a ver ahora un ejemplo de programa más complejo en el que intervienen varios métodos y en el que vemos como desde unos métodos se puede llamar a otros.

Programa que calcule el importe total que tiene que pagar un cliente por la compra de artículos en una tienda. Se trata de una tienda de piezas de repuesto y el importe total de cada venta se obtiene a partir del precio de las pieza, el tipo de cliente (A,B,C ó D) y el tipo de pieza (0, 1 ó 2).

Para calcular el importe total a pagar por el cliente, primero se debe introducir el tipo de cliente y a continuación el precio de la pieza y el tipo de pieza para todas las piezas compradas. Cuando no haya más piezas que introducir se introducirá un cero en el precio.

Para cada pieza introducida se mostrará el importe a pagar por ella.

Cuando finalice el proceso de introducir piezas se mostrará el importe total a pagar por el cliente.

Al importe a pagar por cada pieza se le aplica un descuento según su tipo:

- Tipo 1: 5% Tipo 2: 8% Tipo 0: No tiene descuento

Al importe total a pagar por el cliente se le aplica un descuento según el tipo de cliente:

- Tipo A: 2% Tipo B: 4% Tipo C: 6% Tipo D: No tiene descuento

Para resolverlo utilizaremos varios métodos:

- Método para comprobar si el tipo de cliente introducido es correcto.
- Método para comprobar si el tipo de pieza es correcto.
- Método para calcular el importe a pagar por cada pieza según el tipo de pieza.
- Método para calcular el importe final a pagar según el tipo de cliente.
- Método que devuelva el porcentaje de descuento a aplicar según el tipo de pieza.
- Método que devuelva el porcentaje de descuento a aplicar según el tipo de cliente.

```
import java.util.Scanner;
public class EjemploTiendaRepuestos {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double precio, importe, totalImporte = 0;
        char tipoCliente;
        int tipoPieza;
        boolean correcto;
        do { //Se lee el tipo de cliente
            correcto = true;
            System.out.print("Introduce tipo de cliente (A, B, C, D): ");
            tipoCliente = sc.nextLine().charAt(0);
            //se llama al método que comprueba si el tipo de cliente es válido
            if (!(correctoTipoCliente(tipoCliente))) {
                System.out.println("Valor no válido");
                correcto = false;
            }
        } while (!correcto);
```

```
        do { //se lee el precio y el tipo de todas las piezas
            do {
                System.out.print("Precio de la pieza: (0 para terminar): ");
                precio = sc.nextDouble();
                if (precio < 0) {
                    System.out.println("Valor no válido");
                }
            } while (precio < 0);
            if (precio != 0) { //Precio cero significa fin de lectura de las piezas
                do { //se lee el tipo de pieza
                    correcto = true;
                    System.out.print("Tipo de pieza (0, 1, 2): ");
                    tipoPieza = sc.nextInt();
                    //se llama al método que comprueba si el tipo de pieza es válido
                    if (!(correctoTipoPieza(tipoPieza))) {
                        System.out.println("Valor no válido");
                        correcto = false;
                    }
                } while (!correcto);
                //se llama al método que calcula el importe a pagar por la pieza
                importe = calcularImportePieza(precio, tipoPieza);
                //se muestra el importe a pagar por la pieza
                System.out.printf("Importe de la pieza: %.2f€ %n", importe);
                //se suma el importe de la pieza al total de la compra
                totalImporte = totalImporte + importe;
            }
        } while (precio != 0);
        //se muestra el importe total a pagar.
        //Para calcularlo se llama al método calcularImporteTotal
        System.out.printf("Importe total: %.2f€ %n", calcularImporteFinal(totalImporte,tipoCliente));
    }
    //método para comprobar si el tipo de cliente es válido
    public static boolean correctoTipoCliente(char cliente) {
        switch (cliente) {
            case 'A':
            case 'B':
            case 'C':
            case 'D':
            case 'a':
            case 'b':
            case 'c':
            case 'd': return true;
        }
        return false;
    }
    //método para comprobar si el tipo de pieza es válido
    public static boolean correctoTipoPieza(int pieza) {
        switch (pieza) {
            case 0:
            case 1:
            case 2: return true;
        }
        return false;
    }
    //método para calcular el precio a pagar por una pieza
    public static double calcularImportePieza(double precio, int tp) {
        //calculamos el descuento de la pieza según el tipo de pieza
        //se llama al método descuentoTipoPieza
        //Le enviamos el tipo de pieza y nos devuelve el porcentaje a aplicar
        double descuento = precio * descuentoTipoPieza(tp);
        return precio - descuento; //precio final de la pieza
    }
    //método para calcular importe final a pagar aplicando el descuento
    //según el tipo de cliente
    public static double calcularImporteFinal(double importe, int tipo){
        //calculamos el descuento de la compra según el tipo de cliente
        //se llama al método descuentoTipoCliente
        //Le enviamos el tipo de cliente y nos devuelve el porcentaje a aplicar
        double descuento = importe * descuentoTipoCliente(tipo);
        return importe - descuento; //importe final de toda la compra
    }
```

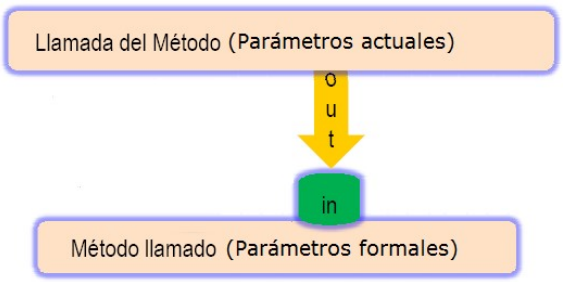
```
    //método que devuelve el % a aplicar según el tipo de pieza
    public static double descuentoTipoPieza(int tipo) {
        if (tipo == 1) {
            return 0.05;
        } else if (tipo == 2) {
            return 0.08;
        } else {
            return 0;
        }
    }
    //método que devuelve el % a aplicar según el tipo de cliente
    public static double descuentoTipoCliente(int tipo) {
        switch (tipo) {
            case 'A':
            case 'a': return 0.02;
            case 'B':
            case 'b': return 0.04;
            case 'C':
            case 'c': return 0.06;
        }
        return 0;
    }
}
```

4. PARÁMETROS ACTUALES Y FORMALES

Como hemos visto en los ejemplos anteriores, los valores que recibe un método para que trabaje con ellos se envían y reciben mediante lo que se conoce como parámetros.

Los parámetros los podemos clasificar en dos grupos según se utilicen para enviar valores al método o para recibir estos valores:

- **Parámetros actuales**: Son los parámetros que aparecen en la **llamada a un método**. Contienen los valores que se le envían al método. Un parámetro actual puede ser una variable, un objeto, un literal, etc.
- **Parámetros formales**: Son los parámetros que aparecen **en la cabecera del método**. Reciben los valores que se envían en la llamada al método. Estos parámetros se utilizan como variables dentro del método.



En este ejemplo se indican cuáles son los parámetros actuales y formales:

```
public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int numero1, numero2, resultado;
        System.out.print("Introduce primer número: ");
        numero1 = sc.nextInt();
        System.out.print("Introduce segundo número: ");
        numero2 = sc.nextInt();
        resultado = sumar( numero1, numero2 );
        System.out.println("Suma: " + resultado);
    }

    //método sumar
    public static int sumar(int a, int b){
            int c;
            c = a + b;
            return c;
    }
```

numero1 y numero2 son los parámetros actuales

a y b son los parámetros formales

Los parámetros actuales y los formales **deben coincidir en número, orden y tipo**.

Si el tipo de un parámetro actual no coincide con su correspondiente parámetro formal, el sistema podrá realizar la asignación de valores siempre que se trate de tipos compatibles (como ocurre cuando por ejemplo se asigna un valor float a una variable de tipo double). Si no es posible la asignación, el compilador dará los mensajes de error correspondientes.

En el siguiente ejemplo podemos ver que los tipos de los parámetros actuales y formales no coinciden pero no se produce error. Los parámetros actuales son de tipo int y los formales de tipo double. En ese caso se puede realizar la asignación de un valor int a un valor double.

```
public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int numero1, numero2;
        System.out.print("Introduce primer número: ");
        numero1 = sc.nextInt();
        System.out.print("Introduce segundo número: ");
        numero2 = sc.nextInt();
        System.out.println("Suma: " +  sumar(numero1, numero2) ;
    }

    //método sumar
    public static double sumar(double a, double b) {
        double c;
        c = a + b;
        return c;
    }
```

numero1 y numero2 parámetros actuales de tipo int

a y b parámetros formales de tipo double

En la llamada al método no se produce error porque se pueden hacer las asignaciones
a = numero1
b = numero2

5. ÁMBITO DE UNA VARIABLE

El ámbito o alcance de una variable es la zona del programa donde la variable es accesible.

El ámbito lo determina el lugar donde se declara la variable.

En Java las variables se pueden clasificar según su ámbito en:

- Variables miembro de una clase o atributos de una clase
- Variables locales
- Variables de bloque

Variables miembro o atributos de una clase

Son las declaradas dentro de una clase y fuera de cualquier método.

Aunque suelen declararse al principio de la clase, se pueden declarar en cualquier lugar siempre que sea fuera de un método.

Son accesibles en cualquier método de la clase.

Pueden ser inicializadas. Si no se les asigna un valor inicial, el compilador les asigna uno por defecto:

- 0 para las numéricas
- '\u0000' para las de tipo char
- null para String y resto de referencias a objetos.

Variables locales

Son las declaradas dentro de un método.

Su ámbito comienza en el punto donde se declara la variable.

Se crean en memoria cuando se declaran y se destruyen cuando acaba la ejecución del método.

Están disponibles desde su declaración hasta el final del método donde se declaran.

No tienen un valor inicial por defecto. El programador es el encargado de asignarles valores iniciales válidos.

No son visibles desde otros métodos.

Distintos métodos de la clase pueden contener variables con el mismo nombre. Se trata de variables distintas.

El nombre de una variable local debe ser único dentro de su ámbito.

Si se declara una variable local con el mismo nombre que una variable miembro de la clase, la variable local oculta a la miembro. La variable miembro queda inaccesible en el ámbito de la variable local con el mismo nombre.

Los parámetros formales son variables locales al método.

Variables de bloque

Son las declaradas dentro de un bloque de instrucciones delimitado por llaves { }.

Su ámbito comienza en el punto donde se declara la variable.

Están disponibles desde su declaración hasta el final del bloque donde se declaran.

No son visibles desde otros bloques.

Distintos bloques pueden contener variables con el mismo nombre. Se trata de variables distintas.

Si un bloque de instrucciones contiene dentro otro bloque de instrucciones, en el bloque interior no se puede declarar una variable con el mismo nombre que otra del bloque exterior.

Se crean en memoria cuando se declaran y se destruyen cuando acaba la ejecución del bloque.

No tienen un valor inicial por defecto. El programador es el encargado de asignarles valores iniciales válidos.

Ejemplo de ámbito de variables:

```
public class UnaClase {
    private int numero1;          ⟹ numero1 es una variable miembro o atributo de la clase
    public void calcular() {
        int a = 1;                ⟹ a es una variable local del método calcular()
        if (a!=0){
            System.out.println(a + ", " + numero1);
            int b = 2;            ⟹ b es una variable de bloque. Se ha declarado dentro del bloque if
            System.out.println(a + ", " + b + ", " + numero1);
            if(b!=0){
                int c = 3;        ⟹ c es una variable de bloque. Se ha declarado dentro del bloque if
                System.out.println(a + ", " + b + ", " + c);
            } // Fin del ámbito de c. Final del bloque donde se ha declarado
            System.out.println(a + ", " + b + ", " + c); // esta línea provoca un
                                                         // error de compilación.
                                                         // c esta fuera de su ámbito
                                                         // y por tanto, no declarada
        } //Fin del ámbito de b. Final del bloque donde se ha declarado
    } // Fin del ámbito de a. Final del método calcular
}//Fin del ámbito de numero1. Final de la clase
```

Un ejemplo típico de declaración de variables dentro de un bloque es hacerlo en las instrucciones for:

```
for(int i = 1; i<=20; i+=2){
        System.out.println(i);
}
```

Ámbito de la variable i

La variable i se ha declarado dentro del for y solo es accesible dentro de él.

Si a continuación de la llave final del for escribimos:

```
System.out.println(i);
```

se producirá un error de compilación: la variable i no existe fuera del for y el compilador nos dirá que no está declarada.

Se pueden declarar variables con el mismo nombre en ámbitos distintos. En el siguiente ejemplo se declaran dos variables x en cada bloque for:

```
public static void main(String[] args) {
   int suma = 0;
   for (int x = 1; x <= 10; x++) {
      suma = suma + x;
   }
   for (int x = 1; x <= 10; x++) {
      suma = suma + x * x;
   }
   System.out.println(suma);
}
```

Ámbito de x

Ámbito de x

Las variables locales deben tener nombres únicos dentro de su ámbito. De no ser así se produce lo que se llama un **choque o solapamiento** de variables.

Ejemplo de solapamiento de variables que producen un error de compilación:

```
public static int sumaCuadrados(int n) {
   int n = 0; // ERROR ya hay declarada una variable n en este ámbito
    for (int i = 1; i <= n; i++) {
         n = i * i;
        suma = suma + n;
    }
    return suma;
}
```

Ámbito de n (parámetro formal)
El ámbito de los parámetros formales es todo el método

6. PASO DE PARÁMETROS

En programación hay dos formas de paso de parámetros a un método:

Paso de parámetros por valor

Cuando se invoca al método se crea una nueva variable (el parámetro formal) a la que se le asigna el valor del parámetro actual.

El parámetro actual y el formal son dos **variables distintas** aunque tengan el mismo nombre.

Cualquier modificación que se realice sobre el parámetro formal no afectará al valor del parámetro actual.

En definitiva, en el paso de parámetros por valor el método no puede modificar el valor del parámetro actual.

Paso de parámetros por referencia

Cuando se invoca al método se crea una nueva variable (el parámetro formal) a la que se le asigna la dirección de memoria donde se encuentra el parámetro actual.

En este caso el método trabaja con la **variable original** (el parámetro actual) por lo que puede modificar su valor.

En el paso de parámetros por referencia el método puede modificar el valor del parámetro actual.

Lenguajes como C, C++, C#, php, VisualBasic, etc. soportan ambas formas de paso de parámetros. Java solo soporta el paso de parámetros por valor.

7. PASO DE PARÁMETROS EN JAVA

En Java todos los parámetros se pasan por valor

Cuando se llama a un método en Java siempre se copia el valor de los parámetros actuales en los parámetros formales.

Cuando el **parámetro** es de **tipo primitivo** (int, double, char, boolean, float, short, byte) el paso por valor significa que cuando se invoca al método el valor del parámetro actual se copia en el parámetro formal. **El método no puede modificar el parámetro actual.**

Cuando el **parámetro** es un **objeto** (por ejemplo, un *array* o cualquier otro objeto), ese parámetro contiene la dirección de memoria donde se encuentra el objeto por lo que el paso por valor aquí significa que cuando se invoca al método el parámetro formal recibe la dirección de memoria donde se encuentra el objeto. En este caso el método **sí puede modificar los objetos**.

Tipo del parámetro	Se puede modificar el valor del parámetro actual dentro del método
Tipos primitivos	NO
Arrays y resto de objetos	SI

Los String son un caso particular. Un String es un objeto que representa una cadena de caracteres no modificable. Para poder modificar un String debemos utilizar las clases StringBuilder o StringBuffer. Ambas son Strings modificables (las veremos en el tema siguiente).

Ejemplo de paso de parámetros de tipo básico por valor donde podemos ver que el método no puede modificar el valor del parámetro actual:

El método *incrementar* tiene un parámetro *n* de tipo int. Aunque n se modifica dentro del método, el parámetro actual *n* de main no se modifica.

```
public static void main(String[] args) {
        int n = 1;
        System.out.println("Valor de n en main antes de llamar al método:" + n);
        incrementar(n);
        System.out.println("Valor de n en main después de llamar al método:" + n);
}

public static void incrementar(int n){
           n++;
           System.out.println("Valor de n en el método después incrementar:" + n);
}
```

n en main y n en el método incrementar son variables distintas

La salida de este programa es:

```
Valor de n en main antes de llamar al método: 1
Valor de n en el método después incrementar: 2
Valor de n en main después de llamar al método: 1
```

En resumen, si el parámetro actual es de tipo primitivo, su valor no puede ser modificado dentro del método. Si el parámetro actual es un objeto su valor sí puede ser modificado dentro del método.

8. SOBRECARGA. MÉTODOS SOBRECARGADOS

Un método está sobrecargado cuando aparece definido varias veces en una clase con el mismo nombre pero con distinto número de parámetros o con el mismo número de parámetros pero de tipos distintos.

La sobrecarga de métodos **permite que un mismo método realice acciones diferentes** dependiendo de los parámetros que reciba.

Ejemplo: Vamos a sobrecargar un método llamado *sumar* para que calcule y devuelva la suma de los parámetros que le enviamos:

```
public static int sumar(int a, int b) {
	return a + b;
}
public static int sumar(int a, int b, int c) {
	return a + b + c;
}
public static int sumar(int a, int b, int c, int d) {
	return a + b + c + d;
}
public static double sumar(double a, int b) {
	return a + b;
}
public static double sumar(int a, double b) {
	return a + b;
}
```

Ejemplo de uso:

```
public static void main(String[] args) {
	System.out.println(sumar(10, 5, 1, -9));
	System.out.println(sumar(3.0, 5));
	System.out.println(sumar(3, 5.3));
	System.out.println(sumar(3, 5));
	System.out.println(sumar(4, 5, 7));
}
```

El resultado que se muestra por pantalla es:

```
7
8.0
8.3
8
16
```

Ejemplo: Sobrecarga de un método llamado *mostrar*

```
public static void mostrar(int a){
	System.out.println("Valor: " + a);
}
public static void mostrar(char a){
	System.out.println("Valor: " + a);
}
public static void mostrar(double a){
	System.out.printf("Valor: %.2f%n", a);
}
public static void mostrar(String mensaje, int a){
	System.out.println(mensaje + ": " + a);
}
public static void mostrar(String mensaje, double a){
	System.out.printf("%s: %.2f%n", mensaje, a);
}
public static void mostrar(String mensaje, char a){
	System.out.printf("%s: %c%n", mensaje, a);
}
```

Ejemplo de uso:

```
public static void main(String[] args) {
	 mostrar('k');
	 mostrar("Letra inicial", 'p');
	 mostrar(10);
	 mostrar("Nota: ", 10);
	 mostrar(8.7585);
	 mostrar("Nota: ", 8.7585);
}
```

Resultado:

```
Valor: k
Letra inicial: p
Valor: 10
Nota: 10
Valor: 8,76
Nota: 8,76
```

En la sobrecarga de métodos **el tipo de dato de un parámetro actual se puede promocionar de forma implícita al tipo de dato de un parámetro formal** si fuese necesario. Esta operación se realiza si no se encuentra un método cuyos parámetros coincidan con los tipos de los datos utilizados en la llamada a dicho método. Ejemplo:

```
public static void main(String[] args) {
	System.out.println(sumar(3, 5.3));
}
public static int sumar(int a, int b) {
	return a + b;
}
public static double sumar(double a, double b) {
	return a + b;
}
```

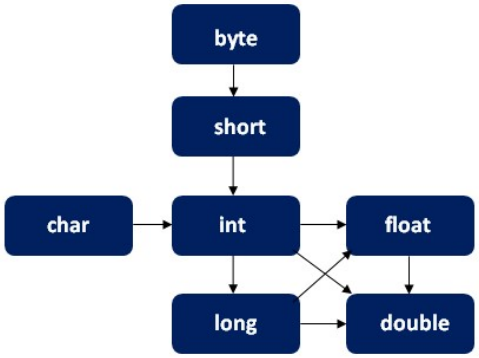
En el ejemplo se está invocando al método *sumar* con un parámetro de tipo int y otro de tipo double:

sumar(3, 5.3)

Sin embargo, no hay ningún método cuyos parámetros coincidan con los tipos de datos utilizados en la llamada. En este caso, el primer parámetro actual de tipo int (3) se convierte a double y se ejecuta el método:

```
public static double sumar(double a, double b) {
	return a + b;
}
```

En la siguiente imagen puedes ver como se realiza la promoción implícita de tipos:



Un byte se puede promocionar a short, int, long, float, double.

Un char a int, long, float, double. etc.

9. MÉTODOS CON UN NÚMERO VARIABLE DE PARÁMETROS

Java permite crear métodos que tengan un número variable de parámetros. A estos métodos se les llama métodos **varargs**. De esta forma podemos llamar al método utilizando un número de parámetros diferente según nos interese en cada momento.
Es una alternativa a la sobrecarga.
El número variable de argumentos se indica mediante tres puntos (. . .) entre el tipo y el nombre de parámetro.

Ejemplo: Creamos un método sumar que puede recibir un número variable de valores de tipo entero.

```
public static int sumar(int... numeros) {
	int suma = 0;
	for(int i = 0; i < numeros.length; i++){
	    suma+=numeros[i];
	}
	return suma;
}
```

Como vemos en el ejemplo, de forma interna **un vararg es un array**. Veremos los arrays en el tema siguiente.

Este método lo podemos llamar enviándole cada vez un número de parámetros distinto:

```
System.out.println(sumar(3, 5));
System.out.println(sumar(2, 5, 7, 8, 5));
System.out.println(sumar(1, -3, 7, 8, 2, 5, 3, 29, 47));
System.out.println(sumar(100, 45, -2));
```

Los métodos varargs deben seguir las siguientes reglas:

- Un método solo puede tener un parámetro de este tipo.
- Si el método tiene más parámetros, el varargs debe ser el último (el de más a la derecha).
- Un método vararg se puede sobrecargar aunque no es recomendable porque puede producir situaciones de ambigüedad.

10. RECURSIVIDAD

La recursividad es una técnica potente de programación que puede utilizarse en lugar de la iteración para resolver determinados tipos de problemas.

Por ejemplo, para escribir un método que calcule el factorial de un número entero no negativo, podemos hacerlo a partir de la definición de factorial:

```
0! = 1
n! = n * (n – 1) * (n – 2) * ... * 3 * 2 * 1    si n > 0
```

Esto dará lugar a una solución iterativa mediante un bucle for:

```
// Método no recursivo para calcular el factorial
public double factorial(int n){
    double fact = 1;
    for(int i = 1; i<= n; i++){
        fact = fact * i;
    }
    return fact;
}
```

Pero existe otra definición de factorial en función de sí misma:

```
0! = 1
n! = n * (n – 1)! ,     si n > 0
```

⇒ El factorial de n es n multiplicado por el factorial de n-1

Esta definición da lugar a una solución recursiva:

```
// Método recursivo para calcular el factorial
public double factorial(int n){
    if (n==0){
        return 1;
    }else{
        return n*(factorial(n-1));
    }
}
```

⇒ **Un método es recursivo cuando contiene una llamada a sí mismo**

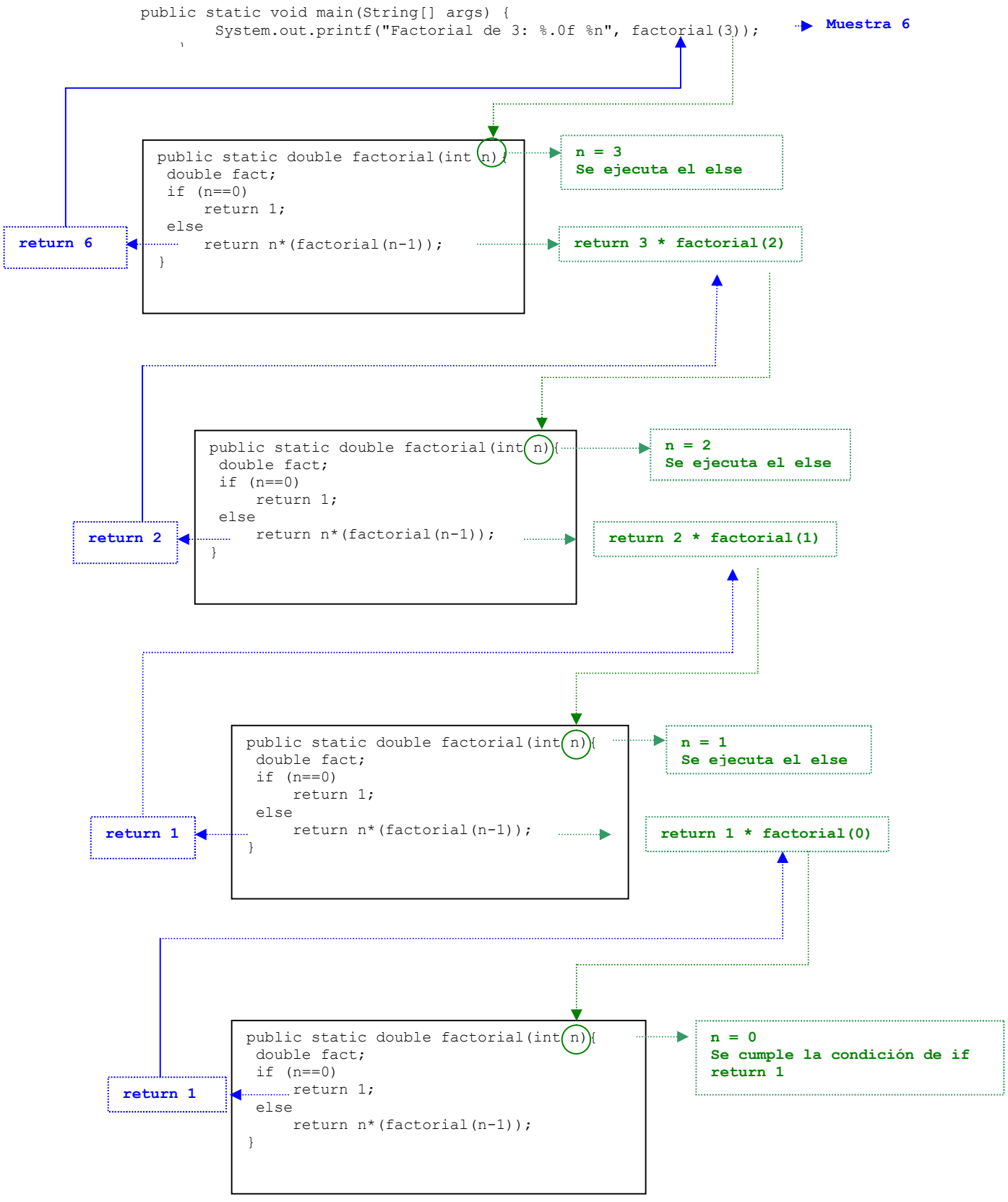
La solución iterativa del factorial es fácil de entender. Utiliza una variable para acumular los productos y obtener la solución.

En la solución recursiva se realizan llamadas al propio método con valores de n cada vez más pequeños para resolver el problema.

Cada vez que se produce una nueva llamada al método se crean en memoria de nuevo las variables y comienza la ejecución del nuevo método.

Para entender el funcionamiento de la recursividad, **podemos pensar que cada llamada recursiva supone llamar a un *método diferente*, copia del original, que se ejecuta y devuelve el resultado a quien lo llamó**.

En la figura siguiente podemos ver como sería la ejecución del programa anterior para calcular el factorial de 3. El color verde indica la secuencia de llamadas recursivas al método. El método se va llamando hasta que recibe 0 como valor de n. Cuando recibe el valor 0 comienza la secuencia de valores devueltos. Cuando finaliza la secuencia de llamadas, la línea azul indica lo que cada método devuelve y a quién se lo devuelve.



Un método recursivo debe contener:

- Uno o más **casos base**: casos para los que existe una solución directa.
- Una o más **llamadas recursivas**: casos en los que se llama sí mismo

Caso base: Siempre ha de existir uno o más casos en los que los valores de los parámetros de entrada permitan al método devolver un resultado directo. Estos casos también se conocen como **solución trivial** del problema.

En el ejemplo del factorial el caso base es la condición:

```
if (n==0){
     return 1;
}
```

si n=0 el resultado directo es 1 No se produce llamada recursiva

Llamada recursiva: Si los valores de los parámetros de entrada no cumplen la condición del caso base se llama recursivamente al método.

En las llamadas recursivas el valor del parámetro en la llamada se ha de modificar de forma que se aproxime cada vez más hasta alcanzar al valor del caso base.

En el ejemplo del factorial en cada llamada recursiva se utiliza n-1

return n * factorial(n-1);

por lo que en cada llamada el valor de n se acerca más a 0 que es el caso base.

11. EJEMPLOS DE RECURSIVIDAD

Ejemplo 1: Calculo de 2^n siendo n un número entero >= 0.

Caso base: si n = 0 entonces $2^0 = 1$

Si n > 0 entonces $2^n = 2 * 2^{n-1}$ Por ejemplo: $2^5 = 2 * 2^4$

```
import java.util.Scanner;
public class Potencia2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int num;
        do{
           System.out.print("Introduce un numero entero >=0 ");
           num = sc.nextInt();
        }while(num<0);
        System.out.println("2 ^ " + num + " = " + potencia(num));
    }

    //Método recursivo para calcular 2 elevado a n
    public static double potencia(int n){
           if(n==0){   //caso base
              return 1;
           }else{
              return 2 * potencia(n-1);
           }
    }
}
```

Ejemplo 2: método recursivo que calcula la suma desde 1 hasta un número N leído por teclado.

Caso base: Si N == 1 la suma es 1

Si N > 1 la suma es N + la suma de los anteriores hasta 1

Por ejemplo, si N = 5 suma = 5 + suma desde el 1 hasta el 4

```
import java.util.Scanner;
public class Suma1N {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int num;
        do{
           System.out.print("Introduce un numero entero >0 ");
           num = sc.nextInt();
        }while(num<=0);
        System.out.println("Suma desde 1 hasta " + num + " = " + suma1N(num));
    }
    //Método recursivo para calcular la suma desde 1 hasta n
    public static double suma1N(int n){
           if(n == 1){  //caso base
              return 1;
           }else{
              return n + suma1N(n-1);
           }
    }
}
```

Ejemplo 3: método recursivo que calcula el número de cifras de un número entero. La solución recursiva se basa en:

Caso base: Si n < 10 tiene 1 cifra

Si n >=10 tiene las cifras de un número con una cifra menos + 1

Por ejemplo, si n = 1234 el número de cifras es 1 + las cifras de 123

```
import java.util.Scanner;
public class CuentaCifras {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int num;
        do{
           System.out.print("Introduce un numero entero >0 ");
           num = sc.nextInt();
        }while(num<=0);
        System.out.println("Número de cifras: " + numeroCifras(num));
    }
    //Método recursivo para calcular el número de cifras de un número entero
    public static int numeroCifras(int n){
           if(n < 10){  //caso base
              return 1;
           }else{
              return 1 + numeroCifras(n/10);
           }
    }
}
```

La recursividad es especialmente apropiada cuando el problema a resolver (por ejemplo cálculo del factorial de un número) o la estructura de datos a procesar (por ejemplo los árboles) tienen una clara definición recursiva. Cuando el problema se pueda definir mejor de una forma recursiva que iterativa lo resolveremos utilizando recursividad.

No se debe utilizar la recursión cuando la iteración ofrece una solución obvia.

Para medir la eficacia de un algoritmo recursivo se tienen en cuenta tres factores:

- Tiempo de ejecución
- Uso de memoria
- Legibilidad y facilidad de comprensión

Las soluciones recursivas suelen ser **más lentas** que las iterativas por el tiempo empleado en la gestión de las sucesivas llamadas a los métodos. Además consumen **más memoria** ya que se deben guardar los contextos de ejecución de cada método que se llama.

A pesar de estos inconvenientes, en ciertos problemas, la recursividad conduce a soluciones que son mucho más fáciles de leer y comprender que su correspondiente solución iterativa. En estos casos una mayor claridad del algoritmo puede compensar el coste en tiempo y en ocupación de memoria.

De todas maneras, numerosos problemas son difíciles de resolver con soluciones iterativas, y sólo la solución recursiva conduce a la resolución del problema (por ejemplo, Torres de Hanoi o recorrido de Árboles).

Torres de Hanoi

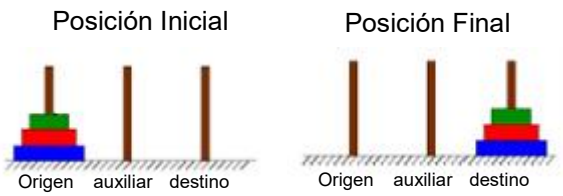
Es un juego oriental que consta de tres columnas llamadas *origen, destino* y *auxiliar* y una serie de discos de distintos tamaños.

Los discos están colocados de mayor a menor tamaño en la columna origen.

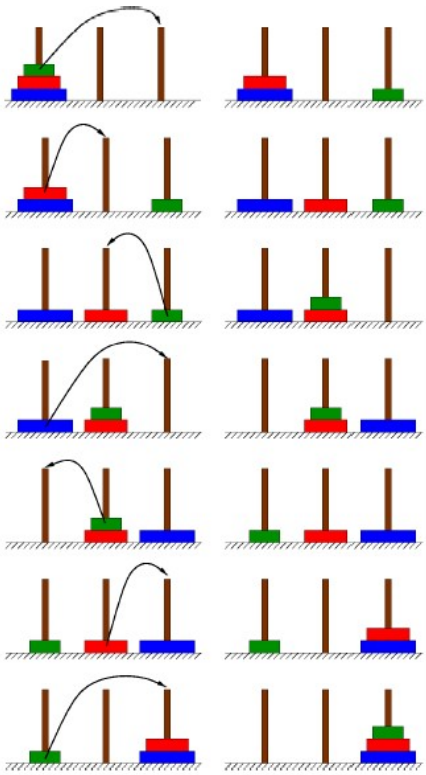
El juego consiste en pasar todos los discos a la columna destino y dejarlos como estaban de mayor a menor (el más grande en la base, el más pequeño arriba).

Las reglas del juego son las siguientes:

- Sólo se puede mover un disco cada vez.
- Para cambiar los discos de lugar se pueden usar las tres columnas.
- Nunca deberá quedar un disco grande sobre un disco pequeño.



Secuencia de movimientos a realizar:



El problema de las torres de Hanoi se puede resolver de forma muy sencilla usando la recursividad. Para ello basta con observar que si sólo hay un disco (caso base), entonces se lleva directamente de la varilla *origen* a la varilla *destino.* Si hay que llevar n>1 (caso general) discos desde *origen* a *destino* entonces:

Se llevan n-1 discos de la varilla *origen* a la *auxiliar*.

Se lleva un solo disco (el que queda) de la varilla *origen* a la *destino*

Se traen los n-1 discos de la varilla *auxiliar* a la *destino*.

Utilizando recursividad se obtiene una solución muy simple al problema de las Torres de Hanoi:

```
import java.util.Scanner;
public class Hanoi {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n;
        System.out.println("Numero de discos: ");
        n = sc.nextInt();
        Hanoi(n,1,2,3);  //1:origen  2:auxiliar 3:destino
    }

   //Método Torres de Hanoi
   public static void Hanoi(int n, int origen, int auxiliar, int destino){
      if(n==1)
         System.out.println("mover disco de " + origen + " a " + destino);
      else{
          Hanoi(n-1, origen, destino, auxiliar);
          System.out.println("mover disco de "+ origen + " a " + destino);
          Hanoi(n-1, auxiliar, origen, destino);
      }
    }
}
```