

# TEMA 3: Estructuras de control2.

## Introducción estructuras de almacenamiento. Recursividad

### Sumario

.....	1
Métodos.....	2
Enum.....	7
Arrays unidimensionales.....	9
Excepciones.....	14
Recursividad.....	22

## Métodos

Un método en Java 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 de uso habitual que ya hemos usado:

- **Math.sqrt()**
- **System.out.print();**
- **System.out.println();**

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.

Utilizando métodos:

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

En Java un método siempre pertenece a una clase.

Todo programa java tiene un método llamado main. Este método es el punto de entrada al programa y también el punto de salida.

### ESTRUCTURA GENERAL DE UN MÉTODO JAVA

Primero veamos un ejemplo

```
static int calcularDoble( int num ){  
    int resultado = num * 2;  
    return resultado;  
}
```

El método anterior se llama calcularDoble y calcula el doble de un número.

Recibe un número en la variable num

luego crea una nueva variable llamada resultado y le asigna el doble de lo que valga num  
finalmente devuelve mediante la instrucción return lo que vale la variable resultado

Observar que lo que se pone entre paréntesis son los argumentos que recibe el método. En este caso nos está diciendo que el número que se le pase como argumento tiene que ser de tipo entero.

De momento no vamos a explicar la palabra static. Pero justo después de esa palabra, y justo antes del nombre del método es el tipo de dato que el método va a devolver. En este caso devolverá un dato de tipo int

Podemos usar ese método de la siguiente forma, por ejemplo:

```
static void main(String args[] ){  
    int valor;  
    int dobleDeValor;  
    valor = 5;  
    dobleDeValor = calcularDoble(valor);  
    System.out.println("El doble de la variable valor es: " + dobleDeValor);  
}
```

Como se puede observar se le pasa el contenido de la variable valor ( en este caso 5 ) y devuelve el doble de ese dato y lo guardamos en la variable: dobleDeValor

La estructura general de un método Java es la siguiente:

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

Los elementos que aparecen entre corchetes son opcionales.

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

**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 ha de devolver. El dato se devuelve mediante la instrucción return. Si el método no devuelve ningún valor este tipo será void.

**nombreMetodo**: es el nombre que se le da al método. Para crearlo hay que seguir las mismas normas que para crear nombres de variables.

**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 son obligatorios aunque estén vacíos.

**throws listaExcepciones** (opcional): indica las excepciones que puede generar y manipular el método.

**return**: se utiliza para devolver un valor. La palabra clave return va seguida de una expresión que será evaluada para saber el valor de retorno. Esta expresión puede ser compleja o puede ser simplemente el nombre de un objeto, una variable de tipo primitivo o una constante.

El tipo del valor de retorno 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.

Un método puede devolver un tipo primitivo, un array, un String o un objeto.

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 la instrucción return.

La instrucción return puede aparecer en cualquier lugar dentro del método, no tiene que estar necesariamente al final.

### **Ej. método que suma dos números enteros.**

El método se llama sumar y recibe dos números enteros a y b. En la llamada al método los valores de las variables numero1 y numero2 se copian en las variables a y b. El método suma los dos números y guarda el resultado en c. Finalmente devuelve mediante la instrucción return la suma calculada.

```
{
    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);
    }

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

**Ej. Método que no devuelve ningún valor.** El método `cajaTexto` recibe un `String` y lo muestra rodeado con un borde.

El tipo devuelto es `void` y es opcional escribir la sentencia `return`. El método acaba cuando se alcanza la llave final.

```
{
    public static void main(String[] args) {
        Scanner cin = new Scanner(System.in);
        String cadena;
        System.out.print("Introduce cadena de texto: ");
        cadena = cin.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();
        for (int i = 0; i < n + 4; i++){
            System.out.print("#");
        }
        System.out.println();
        System.out.println("# " + str + " #");
        for (int i = 0; i < n + 4; i++){
            System.out.print("#");
        }
        System.out.println();
    }
}
```

- **Práctica 1:** Crear un programa para calcular el máximo común divisor ( mcd ) de dos números enteros recibidos por teclado. Se deberá crear un método llamado `mcd` que recibe los dos números y devuelve el máximo común divisor. El método `main` del programa es:

```
public static void main(String[] args) {
    Scanner cin = new Scanner(System.in);
    System.out.println("Cálculo de MCD para dos números");
    System.out.print("Número 1: ");
    int num1 = cin.nextInt();
    System.out.print("Número 2: ");
    int num2 = cin.nextInt();

    String solucion = "MCD: " + mcd(num1,num2);
    System.out.println(solucion);
}
```

● **Práctica 2:** Utilizando el método `mcd()` creado en la práctica anterior crear un programa que calcule el mcm. Recordar que:  $mcm(a,b) = a*b/mcd(a,b)$

● **Práctica 3:** Crear un programa con un método llamado `aleatorio()` que reciba dos números enteros positivos y devuelva un número aleatorio que esté entre esos dos números. El `main()` del programa:

```
public static void main(String[] args) {  
    int num;  
    do{  
        num = aleatorio(25,45);  
        System.out.println(num);  
    }while( num != 35 );  
}
```

● **Práctica 4** Crear un método llamado `numeroValido()`. El usuario debe introducir un número entre 20 y 50 y ser múltiplo de 3. El `main()` del programa:

```
public static void main(String[] args) {  
    int num;  
    Scanner cin = new Scanner(System.in);  
    do{  
        System.out.println("Introducir un número: ");  
        num = cin.nextInt();  
    }while( !numeroValido(num) );  
    System.out.println("El número cumple los requisitos. Se puede continuar");  
}
```

● **Práctica 5:** Crear un método llamado `factorial()` que obtenga el factorial de un número. Hacer uso de él en un programa que el usuario introduzca por teclado un número y se le muestre el factorial de ese número

● **Práctica 6:** Crear un método llamado `combinacion()` que se le pasen dos números y obtenga la combinación de esos dos números. Este método debe hacer uso del método `factorial` creado en la práctica anterior. La fórmula de la combinación es:

**`combinacion(n,r) = factorial(n) / ( factorial(n-r) * factorial(r)`**

## Enumeraciones (Enum)

```
public enum Demarcacion
{
    PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO
}
```

Por convenio los nombres de los enumerados se escriben en mayúsculas.

Los tipos de datos enumerados son una forma de declarar una variable con un conjunto restringido de valores. Por ejemplo, los días de la semana, las estaciones del año, los meses, etc. Es como si definiéramos nuestro propio tipo de datos.

La forma de declararlos es con la palabra reservada `enum`, seguida del nombre de la variable y la lista de valores que puede tomar entre llaves. A los valores que se colocan dentro de las llaves se les considera como constantes, van separados por comas y deben ser valores únicos.

La lista de valores se coloca entre llaves, porque un tipo de datos `enum` no es otra cosa que una especie de clase en Java, y todas las clases llevan su contenido entre llaves.


Veamos un ejemplo:

```
public class tiposenumerados {
    public enum DiaSemanal {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO};

    public static void main(String[] args) {
        // codigo de la aplicacion
        DiaSemanal diaActual = DiaSemanal.MARTES;
        DiaSemanal diaSiguiente = DiaSemanal.MIERCOLES;

        System.out.print("Hoy es: ");
        System.out.println(diaActual);
        System.out.println("Mañana\nes\n"+diaSiguiente);

    } // fin main
} // fin tiposenumerados
```

 **Práctica 7:** Crear un programa con un tipo enumerado para las estaciones y mostrar en pantalla el dato enumerado correspondiente a la actual estación.





# Arrays unidimensionales

Los arrays permiten almacenar una colección de objetos o datos del mismo tipo. Son muy útiles y su utilización es muy simple:

Declaración del array. La declaración de un array consiste en decir "esto es un array" y sigue la siguiente estructura:

```
tipo[] nombre; //esta es la opción estandar  
tipo nombre[]; //esta también se acepta
```

el tipo será un tipo de variable o una clase ya existente, de la cual se quieran almacenar varias unidades.

Creación del array. La creación de un array consiste en decir el tamaño que tendrá el array, es decir, el número de elementos que contendrá, y se pone de la siguiente forma:

```
nombre=new tipo[dimension];
```

donde dimensión es un número entero positivo que indicará el tamaño del array. Una vez creado el array este no podrá cambiar de tamaño.

Veamos un ejemplo de su uso:

```
int[] n; // Declaración del array.  
n = new int[10]; //Creación del array reservando para el un espacio en memoria.  
int[] m=new int[10]; // Declaración y creación en un mismo lugar.
```

Una vez hecho esto, ya podemos almacenar valores en cada una de las posiciones del array, usando corchetes e indicando en su interior la posición en la que queremos leer o escribir, teniendo en cuenta que la primera posición es la cero y la última el tamaño del array menos uno. En el ejemplo anterior, la primera posición sería la 0 y la última sería la 9.

La modificación de una posición del array se realiza con una simple asignación. Simplemente se especifica entre corchetes la posición a modificar después del nombre del array. Veamoslo con un simple ejemplo:

```
int[] numeros=new int[3]; // Array de 3 números (posiciones del 0 al 2).  
numeros[0]=5; // Primera posición del array.  
numeros[1]=25; // Segunda posición del array.  
numeros[2]=20; // Tercera y última posición del array.
```

El acceso a un valor ya existente dentro de una posición del array se consigue de forma similar, simplemente poniendo el nombre del array y la posición a la cual se quiere acceder entre corchetes:

```
int suma=numeros[0] + numeros[1] + numeros[2];
```

El valor de suma será 50

Los array soportan la propiedad `length` nos permite saber el tamaño de cualquier array, lo cual es especialmente útil en métodos que tienen como argumento un array.

```
System.out.println("Longitud del array: "+numeros.length);
```

Un uso habitual de los arrays es en el paso de parámetros. Para pasar como argumento un array a una función o método, esta debe tener en su definición un parámetro declarado como array. Esto es simplemente que uno de los parámetros de la función sea un array. Veamos un ejemplo:

```
int sumaArray (int[] array) {  
    int suma=0;  
    for (int i=0; i < array.length;i++)  
        suma=suma+array[i];  
    return suma;  
}
```

En el método anterior se pasa como argumento un array numérico, sobre el cual se calcula la suma de todos los números que contiene. Fijarse que especificar que un argumento es un array es igual que declarar un array, sin la creación del mismo. Para pasar como argumento un array a una función, simplemente se pone el nombre del array:

```
int suma=sumaArray(numeros);
```

En Java las variables se pasan por copia a los métodos, esto quiere decir que cuando se pasa una variable a un método, y se realiza una modificación de su valor en dicho método, el valor de la variable en el método desde el que se ha realizado la invocación no se modifica. Pero **cuidado, eso no pasa con los arrays**. Cuando dicha modificación se realiza en un array, es decir, se cambia el valor de uno de los elementos del array, si que cambia su valor de forma definitiva. Veamos un ejemplo que ilustra ambas cosas:

```
public static void main(String[] args) {
    int num=0;
    int[] vect=new int[1];
    vect[0]=0;
    System.out.println(num+"-"+vect[0]);
    modificaArray(num, vect);
    System.out.println(num+"-"+vect[0]);
}

static int modificaArray(int num, int[] vect){
    num = 5;
    vect[0] = 5;
    /* Modificación de los valores de la variable, solo afectará al array, no a
    num */
}
```

Mostrará por pantalla "0-0" antes de llamar a modificaArray() y "0-5" después. Se puede observar que las modificaciones dentro del método afectan al array y no a la variable entera.

## Inicialización del array

Se puede inicializar y declarar a la vez un array:

```
int[] array = {10, 20, 30};

String[] diasSemana= {"Lunes", "Martes", "Miércoles", "Jueves", "Viernes",
"Sábado", "Domingo"};
```

Pero cuidado, la inicialización solo se puede usar en ciertos casos. La inicialización anterior funciona cuando se trata de un tipo de dato primitivo (int, short, float, double, etc.) o un String, y algunos pocos casos más, pero no funcionará para cualquier objeto.

Una inicialización más general pasa normalmente por hacer uso de un bucle:

```
int size = 10;
int[] array;
array = new int[size];
for(int i=0; i<size;i++){
    array[i] = 0;
}
```

● **Práctica 8:** Crear un programa que, mediante un bucle, guarde 10 números en un array introducidos por el usuario. Luego, también con un bucle, muestre cada uno de esos números y el índice que le corresponde en el array

● **Práctica 9:** Crear un programa que guarde en un array 10 números aleatorios entre 1 y 99 que sean pares. Luego mostrar en pantalla los 10 números, así como el máximo y el mínimo de esos 10 números y las respectivas posiciones que ocupan en el array

● **Práctica 10:** Hacer un programa que primero solicita la cantidad de números que se van a introducir. Después de haberlos introducido muestra la media y los números que se han introducido

● **Práctica 11:** Modificar el programa anterior para que calcule también la varianza

● **Práctica 12:** Crear un programa que genere 20 números aleatorios enteros entre 1 y 100. Este array una vez se hay rellenado no se puede modificar. Crear un segundo array donde se almacenará una copia de los 5 números más pequeños del primer array. Mostrar en pantalla el contenido del array de 20 números y mostrar cuáles son los 5 números más pequeños

● **Práctica 13:** Crear una variante del programa anterior que en lugar de guardar una copia de los números más pequeños guarde la posición en la que están esos números.

● **Práctica 14:** Crear un programa que introduzca 10 números por teclado y mostrarlos ordenados de menor a mayor al finalizar

● **Práctica 15:** Crear un programa que muestre al azar una carta de la baraja española. Por ej. “sota de copas” Para ello se usarán dos array En el primero se registra:  
1,2,3,4,5,6,7,sota,caballo,rey  
En el segundo  
oros,copas,bastos,espadas  
Mediante dos aleatorios se toma un dato del primer array y un dato del segundo array para componer el nombre de la carta a mostrar

● **Práctica 16:** Crear un programa que genere 10 números aleatorios enteros entre 1 y 100. Se deben mostrar esos 10 números, la media de esos 10 números y decir cuáles de esos 10 números son mayores que la media calculada.

**Observar el siguiente array:**

[ 5 , 2 , 12 , 14 , 8 , 9 , 11 ]  
0 , 1 , 2 , 3 , 4 , 5 , 6 ← Posiciones

Imaginemos un bucle que vaya comparando la posición  $i$  con la  $i+1$  y si el número de la posición  $i$  es mayor que el de la posición  $i+1$  los intercambia

Ej. Veamos todos los pasos del bucle con el array anterior:

$i=0$ . En  $i$  y  $i+1$  tenemos 5 y 2 En este caso  $5 > 2$  entonces intercambia los dos números

[ 5, 2, 12, 14, 8, 9, 11 ] //array antes de comparar posición 0 con 1  
[ 2, 5, 12, 14, 8, 9, 11 ] //array después de comparar posición 0 con 1

$i=1$ . En  $i$  y  $i+1$  tenemos 5 y 12 En este caso  $5 \leq 12$  entonces no hace nada

[ 2, 5, 12, 14, 8, 9, 11 ] //array antes de comparar posición 1 con 2  
[ 2, 5, 12, 14, 8, 9, 11 ] //array después de comparar posición 1 con 2

$i=2$  En  $i$  y  $i+1$  tenemos 12 y 8 En este caso  $12 > 8$  entonces intercambia los números

[ 2, 5, 12, 8, 14, 9, 11 ] // array antes  
[ 2, 5, 8, 12, 14, 9, 11 ] // array después

$i=3$  En  $i$  y  $i+1$  tenemos 12 y 14 En este caso  $12 \leq 14$  entonces no hace nada

[ 2, 5, 8, 12, 14, 9, 11 ] // array antes  
[ 2, 5, 8, 12, 14, 9, 11 ] // array después

$i=4$  En  $i$  y  $i+1$  tenemos 14 y 9 En este caso  $14 > 9$  entonces intercambia los números

[ 2, 5, 8, 12, 14, 9, 11 ] // array antes  
[ 2, 5, 8, 12, 9, 14, 11 ] // array después

$i=5$  En  $i$  y  $i+1$  tenemos 14 y 11 En este caso  $14 > 11$  entonces intercambia los números

[ 2, 5, 8, 12, 9, 14, 11 ] // array antes  
[ 2, 5, 8, 12, 9, 11, 14 ] // array después

$i=6$  En este momento el bucle terminaría porque ya no es posible comparar con  $i+1$

Lo que haremos es volver a repetir todo el bucle una y otra vez hasta que se detecte que no se ha cambiado ningún elemento del array. Caso en el cuál ya estará ordenado

Veamos como quedaría el array después de la segunda ejecución completa del bucle:

[ 2, 5, 8, 9, 11, 12, 14 ]

El array ya está ordenado y cuando realice una tercera ejecución completa del bucle se dará cuenta y termina.

● **Práctica 17:** Crear un programa que reproduzca el algoritmo anterior

● **Práctica 18:** Crear un programa que introduzca 5 números y muestre cuáles son los dos números más cercanos. Por ej. Si:  
14,11,2,10,17 => 11,10

# Excepciones

El manejo de excepciones consiste en prever y capturar los posibles errores de ejecución para redirigirlos a una rutina de código preprogramada, de manera que el programa pueda recuperarse del error. Se suele usar, por lo tanto, en lugares sensibles donde podría ocurrir un error de ejecución. Normalmente, en entradas de datos del usuario o en operaciones aritméticas con valores impredecibles.

El control de excepciones en Java se programa mediante un bloque try-catch, que tiene esta sintaxis:

```
try {
    <sentencias que pueden fallar>
}
catch (excepcion-1) {
    <control de la excepción 1>
}
catch (excepcion-1) {
    <control de la excepción 2>
}
...
finally {
    <control final>
}
```

La JVM intentará ejecutar las instrucciones del bloque try. En caso de que ocurra un error de ejecución, buscará si ese error está recogido en alguno de los bloques catch. En tal caso, ejecutará el bloque catch y luego seguirá ejecutando el programa como si tal cosa.

El bloque finally es opcional. Si existe, se ejecutará siempre, sea cual sea la excepción.

Ejemplo: Escribir un programa que divida un número a entre otro b y muestre el resultado.

```
public static void main(String[] args) {
    int a = 10, b = 0, c;
    c = a / b;
    System.out.println("Resultado: " + c);
}
```

En tiempo de ejecución obtendremos:

Exception in thread "main" java.lang.ArithmeticException: / by zero at test.main (test.java:6)

Estamos dividiendo un número por cero. Y esa es la excepción que lanza. Para eso se usa el control de excepciones.

## Ejemplo completo de gestión de Excepción.

Un lugar donde suele tener lugar control de excepciones es con la entrada de datos del usuario.

Sea este código:

```
{
    System.out.println("Introducir un número: ");
    Scanner sc = new Scanner(System.in);
    int numero=0, division=0;
    numero = sc.nextInt();
    division = 5/numero;
    System.out.println("El número introducido es: " + numero
        + " y la división de 5/" + numero + " da: " + division);
}
```

**Práctica 19:** Crear un programa que incluya el código anterior e introducir texto en lugar de un número cuando el programa lo solicite ¿qué ocurre, hay un error? Si es así ¿cuál es el tipo de error/excepción que se desencadena? (tomar captura de pantalla del error y escribir el tipo de la excepción que se pudiera generar de haberla)

La ejecución de nuestro programa se ha paralizado. Si controláramos la entrada del usuario podríamos evitar la finalización del programa. Ejecuta ahora esta variante del código:

```
{
    int numero=0, division=0;
    try{
        System.out.println("Introducir un número: ");
        Scanner sc = new Scanner(System.in);
        numero = sc.nextInt();
    }catch(Exception ex){
        System.out.println("Se ha generado una excepción del tipo: " + ex);
    }
    System.out.println("El número introducido es: " + numero
        + " y la división de 5/" + numero + " da: " + division);
}
```

Podemos observar que a diferencia de antes se llega a ejecutar el último println. Pero podemos también ver que si ponemos texto en lugar de un número obtendremos un mensaje falso que diga que el número que hemos introducido es 0 (cuando sabemos que hemos introducido texto)

Un camino sería aprovechar el uso que podemos hacer de `catch()` para que determine si debemos continuar en un bucle que le siga pidiendo un número válido al usuario o continuar:

```
{
    int numero=0;
    boolean tengoNumeroValido=false;
    int division=0;
    do{
        try{
            System.out.println("Introducir un número: ");
            Scanner sc = new Scanner(System.in);
            numero = sc.nextInt();
            division = 5/numero;
            tengoNumeroValido = true;
        }catch(Exception ex){
            System.out.println("Se debe introducir un número válido");
            tengoNumeroValido=false;
        }
    }while(!tengoNumeroValido);
    System.out.println("El número introducido es: " + numero
        + " y la división de 5/" + numero + " da: " + division);
}
```

Nos mantenemos en el bucle mientras el usuario no ponga un número por teclado

El problema es que pueden darse varios tipos de excepciones y eso complica un poco la forma en la que se deben manejar. En el código anterior hemos usado: `Exception` que es una clase genérica para todas las excepciones. Así el error de entrada por teclado de un número es una hija de `Exception`, así como todas las demás excepciones



Observar el siguiente código que pretende hacer una división, controlando la entrada de un número válido del usuario:

```
{
    int numero=0;
    boolean tengoNumeroValido=false;
    int division=0;
    do{
        try{
            System.out.println("Introducir un número: ");
            Scanner sc = new Scanner(System.in);
            numero = sc.nextInt();
            division = 5/numero;

            tengoNumeroValido = true;
        }catch(Exception ex){
            System.out.println("Se debe introducir un número válido");
            tengoNumeroValido=false;
        }
    }while(!tengoNumeroValido);
    System.out.println("El número introducido es: " + numero
        + " y la división de 5/" + numero + " da: " + division);
}
```

● **Práctica 20:** Crear un programa que incluya el código anterior e introducir el número 0 cuando el programa lo solicite ¿ qué ocurre, hay un error ? Si es así ¿ cuál es el tipo de error/excepción que se desencadena ? ( tomar captura de pantalla del error y escribir el tipo de la excepción que se pudiera generar de haberla ) ¿ es lógico el mensaje recibido?

La división de un número partido por cero da infinito y eso desencadena una excepción. Esta excepción evidentemente no es la misma que la de introducir texto en lugar de un número entonces debiera recibir un mensaje apropiado. Así pues debiéramos controlar cada excepción y poner el mensaje que corresponde:

```
{
    int numero=0;
    boolean tengoNumeroValido=false;
    int division=0;
    do{
        try{
            System.out.println("Introducir un número: ");
            Scanner sc = new Scanner(System.in);
            numero = sc.nextInt();
            division = 5/numero;

            tengoNumeroValido = true;
        }catch(InputMismatchException ex){
            System.out.println("Se debe introducir un número válido ");
            tengoNumeroValido=false;
        }catch(ArithmeticException ex){
            System.out.println("No se puede dividir por cero");
            tengoNumeroValido=false;
        }
    }while(!tengoNumeroValido);
    System.out.println("El número introducido es: " + numero
        + " y la división de 5/" + numero + " da: " + division);
}
```

De la forma anterior podemos controlar las entradas de usuario y manejar las excepciones.

## Uso de la cláusula finally

finally tiene lugar siempre, indistintamente de que haya o no tenido lugar una excepción  
Ej

```
public static void main(String[] args) {  
    int a = 10, b = 0, c;  
    try {  
        c = a / b;  
    }  
    catch (ArithmeticExcepcion e) {  
        c = 0;  
    }finally{  
        System.out.println("Se ha intentado hacer la operación");  
    }  
}
```

Nosotros también podemos forzar al lanzamiento de una excepción. Por ejemplo, la división por 0.0 no lanza una excepción. Esto es por la forma en la que se ha establecido en el estandar de los double. Double recoge los siguientes valores extremos para esa división:

- Double.POSITIVE\_INFINITY → Ej. 3/0.0
- Double.NEGATIVE\_INFINITY → Ej. -2/0.0
- Double.NaN → Ej 0/0.0

Veamos como podemos forzar a que lance una excepción cuando de infinito:

```
static void main(String[] args){  
    double resultado = 0;  
    try{  
        resultado = division(3,0.0);  
    }catch(Exception ex){  
        System.out.println(ex);  
    }  
}  
  
static double division(double a, double b) throws ArithmeticException{  
    if( Double.isInfinite(a/b) ){  
        throw new ArithmeticException("Dividido por cero");  
    }  
    return a/b;  
}
```

Aquí vemos dos nuevas palabras reservadas: throw, throws

Mediante **throw** lanzamos una nueva excepción:

**throw new Exception();**

Mediante **throws** decimos que delegamos el tratamiento de la excepción al método que nos ha llamado:

**Al implementar un método, hay que decidir si las excepciones se propagarán hacia arriba (throws) o se captura en el propio método (catch)**

1. Un método que propaga una excepción que es manejada por main()

```
public static void f() throws IOException
{
    // Fragmento de código que puede
    // lanzar una excepción de tipo IOException
}


public static void main(String[] args){
    try {
        f();
    } catch (IOException error) {
        // Tratamiento de la excepción
    } finally {
        // Liberar recursos (siempre se hace)
    }
}
```

2. Un método equivalente que no propaga la excepción:

```
public static void f()
{
    // Fragmento de código libre de excepciones
    try {
        // Fragmento de código que puede
        // lanzar una excepción de tipo IOException
        // (p.ej. Acceso a un fichero)
    } catch (IOException error) {
        // Tratamiento de la excepción
    } finally {
        // Liberar recursos (siempre se hace)
    }
}

public static void main(String[] args){
    f();
}
```

Como se ve, la diferencia está en quién trata la excepción. Si lo hace el propio método donde se genera, se utilizará un bloque try catch en ese mismo método. Si queremos que quién lo gestione es el método que nos ha llamado, tenemos que poner throws en la definición de nuestro método

 **Práctica 21:** Crear un programa que reciba dos números del usuario. Si el primer número no es un entero entre 1 y 100 lanzará una excepción que mostrará en pantalla “El número debe ser un entero entre 1 y 100”



# Recursividad

La recursividad es una técnica muy utilizada en programación informática. Se suele utilizar para resolver problemas cuya solución se puede hallar resolviendo el mismo problema, pero para un caso de tamaño menor.

Cuando en informática se escribe un programa con un algoritmo recursivo, en las propias sentencias del algoritmo hay una llamada a sí mismo, es decir una de las sentencias llama al algoritmo recursivo en el que está insertada, aunque para solucionar un caso más sencillo.

Los métodos recursivos se pueden usar en cualquier situación en la que la solución pueda ser expresada como una sucesión de pasos o transformaciones gobernadas por un conjunto de reglas claramente definidas.

## Algoritmos recursivos

Un algoritmo recursivo es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva o recurrente.

Generalmente, si la primera llamada al subprograma se plantea sobre un problema de tamaño u orden  $N$ , cada nueva ejecución recurrente del mismo se planteará sobre problemas, de igual naturaleza que el original, pero de un tamaño menor que  $N$ . De esta forma, al ir reduciendo progresivamente la complejidad del problema que resolver, llegará un momento en que su resolución sea más o menos trivial (o, al menos, suficientemente manejable como para resolverlo de forma no recursiva). En esa situación diremos que estamos ante un caso base de la recursividad.

Las claves para construir un subprograma recurrente son:

- Cada llamada recurrente se debería definir sobre un problema de menor complejidad (algo más fácil de resolver).
- Ha de existir al menos un caso base para evitar que la recurrencia sea infinita.

## Ejemplos

### Factorial

La definición de factorial de un número entero positivo es:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1$$

Así, un método que resuelve sería:

```
int factorial(int n){
    int resultado=1;
    for(int i=n; i>0 ; i--)
        resultado *= i;
    return resultado;
}
```

Sin embargo podemos encontrar una solución recursiva.

Lo que hay que observar es que:

**factorial(n) = n\*factorial(n-1)**

Así que podemos escribir:

```
static int factorial(int n){  
    int resultado = n * factorial(n - 1);  
    return resultado;  
}
```

Ahora bien, si ponemos en el IDE ese código podemos observar que nos avisa que hay una recursión infinita. No hemos puesto una condición de parada y estaría eternamente llamándose a sí mismo.

La **condición de parada** suele coincidir con el caso base. En nuestro caso el caso base es:  
 $1! = 1$

o incluso  $0! = 1$  que está ( por definición matemática ) establecido así

Fijarse en la expresión usada: “condición de parada” todo nos apunta a que tenemos que poner un condicional if:

```
static int factorial(int n){  
    int resultado;  
    if( n > 1 )  
        resultado = n * factorial(n - 1);  
    else  
        resultado = 1;  
    return resultado;  
}
```

¿ Cómo se resolverá cuándo se llame al método factorial() ?

Si el número enviado es  $\leq 1$  devolverá 1

En otro caso se llamará a sí mismo para n-1 y cuando se resuelva esa llamada devolveremos la multiplicación de n por lo que nos hayan devuelto

### **Búsqueda de soluciones recursivas: cuatro preguntas básicas.**

- 1. ¿Se puede definir el problema cómo uno o más problemas más pequeños y con el mismo tipo del original?
- 2. ¿Qué es lo que hará de caso/s base?
- 3. A medida que que el problema se reduce ¿se alcanzará el caso base?
- 4. ¿Cómo hacer uso de los casos base para construir una respuesta correcta al problema inicial?

### **En el caso del factorial las respuestas serían:**

1:  $\text{factorial}(n) = n * \text{factorial}(n-1)$

2:  $\text{factorial}(1)=1$ ,  $\text{factorial}(0)=1$

3: necesariamente habrá un momento en que al ir bajando de  $n$  a  $n-1$  en cada iteración alcanzaremos el  $\text{factorial}(1)$  que es condición base

4:  $\text{factorial}(2) = 2 * \text{factorial}(1) \Rightarrow 2 * 1 \Rightarrow 2$

- **Ventajas de la Recursión ya conocidas**
  - Soluciones simples, claras.
  - Soluciones elegantes.
  - Soluciones a problemas complejos.
- **Desventajas de la Recursión: INEFICIENCIA**
  - Sobrecarga asociada con las llamadas a subalgoritmos
    - Una simple llamada puede generar un gran numero de llamadas recursivas. ( $\text{Fact}(n)$  genera  $n$  llamadas recursivas)
    - ¿La claridad compensa la sobrecarga?
    - El valor de la recursividad reside en el hecho de que se puede usar para resolver problemas sin fácil solución iterativa.
  - La ineficiencia inherente de algunos algoritmos recursivos.



## Sucesión de Fibonacci

La sucesión de Fibonacci se define así:

$$\begin{cases} a_1 &= 1 \\ a_2 &= 1 \\ a_n &= a_{n-1} + a_{n-2} \end{cases}$$

Eso nos genera la siguiente sucesión de números:

1, 1, 2, 3, 5, 8, 13, 21,...

Ahora bien, ¿ cómo haríamos un programa para que el usuario pida el término n-esimo y se le muestre su correspondiente valor de la sucesión ?

Hay aproximaciones matemáticas para poder hacer este problema sin recursión, pero éste es sin duda, un problema que por su propia naturaleza recursiva, se presta a usar una solución algorítmica de recursividad

```
static int fibonacci(int n){  
    if( n == 1 || n == 0)  
        return 1;  
    else  
        return fibonacci(n-1) + fibonacci(n-2);  
}
```

Tenemos una condición de parada en n=1 y n=0 y el resto es recursivos

- **Práctica 22:** Imaginemos que java no supiera realizar una multiplicación de dos números enteros positivos. En ese caso aún podríamos realizar una multiplicación usando recursividad aprovechando la siguiente propiedad de la multiplicación:

$$n * a = (n-1) * a + a$$

Así pues podríamos crear un método que se llame multiplicar() donde:

$$n * a = \text{multiplicar}(n, a) = \text{multiplicar}(n-1, a) + a$$

**la condición base es que  $1 * a = a$**

Hacer un programa que resuelva las multiplicaciones utilizando recursión

● **Práctica 23:** Invertir las cifras de un número mediante recursión: 123347 → 743321

● **Práctica 24:** Sumar las cifras de un número mediante recursión: 135 → 1+3+5 → 9

● **Práctica 25:** Sumar todos los elementos de un array mediante recursión

● **Práctica 26:** Multiplicar todos los elementos de un array mediante recursión

● **Práctica 27:** Calcular la potencia de forma recursiva:  $2^7 \rightarrow 128$

● **Práctica 28:** para cualesquier cifra de dinero  $\geq 12$  que cueste una carta ocurre que se puede conseguir franquear con sellos de 4 y 5. Ej  $12 = 4, 4, 4$   $13 = 4, 4, 5$   $14 = 4, 5, 5$   
Encontrar un algoritmo recursivo que resuelva el problema

● **Práctica 29:** sin usar bucles for, while, do while obtener el número menor de un array

● **Práctica 30:** Partamos de la siguiente serie:  
 $1 - 1/2 + 1/3 - 1/4 + 1/5 - \dots - 1/2^n + 1/2^{n+1} - \dots$   
Podemos diseñar un procedimiento recursivo para calcular la suma de los n primeros elementos de la serie, de modo que usemos una función diferente para los elementos pares e impares.