Métodos en Java.

Llegados a este punto, ya hemos utilizado unos cuantos métodos del **API de Java** en nuestros ejemplos:

- **Métodos estáticos de Clase:**
  Math.abs () – **Math**.round () – **Math**.max ()– **Math**.min () – **Math**.random() … etc
  Arrays.sort() – **Arrays**.binarySearch()

- **Métodos de objeto:**
  Scanner sc ⟶ sc.nextInt () – sc.nextLine () – sc.next ()

  String frase ⟶ frase.length () – frase.charAt () – frase.subString () – frase.valueOf ()
  frase.toUpperCase () – frase.equalsIgnoreCase () … etc

Los hemos utilizado|llamado|invocado de esas **2 formas**, según si son **Estáticos o No estáticos.**

- **MÉTODOS ESTÁTICOS.** Son métodos que invocamos directamente usando el **nombre de su clase**, sin necesidad de instanciar objetos de la clase a la que pertenecen. Para ello han sido declarados como estáticos **(static)** dentro de su clase**.** Esto es característico de las llamadas clases **"helper".** Las clases **Math y Arrays son** un buen ejemplo de clases **"helper" o de utilidad**.

  Para llamar a métodos **estáticos** de clases "helper" usamos **NombreClase.nombreMétodo**:

  Math.pow(3,3) – Math.round(total) – Math.random() – Arrays.sort() … etc

- **METODOS NO ESTÁTICOS.** Hasta ahora hemos utilizado métodos de la clase **Scanner y String** en nuestros ejemplos. **SIEMPRE** hemos necesitado antes **declarar e instanciar** un **OBJETO** de la **CLASE** correspondiente. Los métodos **NO** estáticos han de ser llamados por un objeto **"vivo".** Los métodos no estáticos dan forma al **comportamiento** de los objetos de una clase.

  Para llamar a métodos NO estáticos, **es preciso declarar e instanciar objetos de su clase**. Después usaremos **nombreobjeto.nombreMétodo**:

  String frase = new String ("Estamos aprendiendo Java");

  frase.length() - frase.charAt(5) – frase.subString(0,7)
  24 **'o'** **"Estamos"**

Hasta el momento, sólo hemos USADO métodos del API de Java.
En adelante aprenderemos a CREAR nuestros propios métodos.

Aprender a crear métodos en Java, y usar correctamente la gran cantidad de métodos del API de Java y de APIs de terceros, es de vital importancia para progresar en Programación

¿Cuándo hay que crear MÉTODOS propios?

SIEMPRE!! En programación orientada a objetos (POO), veremos que **NO** hay otra forma de definir el **comportamiento** de un **OBJETO** que no sea a través de sus **MÉTODOS**. Por lo tanto, la importancia de definir correctamente y aprender a llamar|invocar **métodos** en Java es **fundamental**

Tradicionalmente, los métodos surgieron con los conceptos de **programación modular y reutilización de código**. Siempre se han creado módulos de código independientes **(en Java se llaman métodos)** cuando nos encontramos con alguna funcionalidad que necesitamos ejecutar en repetidas ocasiones dentro de un programa. En vez de escribir todo el código **cada vez que necesitemos esa funcionalidad**, la idea es crear un módulo de código independiente **(método)** que haga lo que se necesita. Posteriormente sólo tendremos que llamar/invocar a ese método las veces que necesitemos para ejecutar la funcionalidad deseada.

EJEMPLO: Durante este curso hemos utilizado el método **length,** que nos devuelve la longitud (caracteres) de un String. Es una funcionalidad que trabajando con Strings se necesita a menudo, de ahí que Java incluya ese método en la **clase String**. Es un método sencillo. **Si no existiera en el API de java. ¿podríamos hacerlo nosotros? …** una tosca aproximación podría sería algo así:

```
import java.util.Scanner;
public class Utilidades {

   public static void main(String[] args)
   {
         Scanner sc = new Scanner(System.in);
         System.out.println("Teclea el String: ");
         String cadena = new String (sc.nextLine());
         System.out.println(" la longitud de la cadena es: " + longitud(cadena));

   }

   public static int longitud(String s)
   {
      int contador=0;
      char c = '\u00a5';              //añadimos al String un caracter unicode no existente en teclado
      String aux = s + c;
      while (aux.charAt(contador) != c)        // buscamos el carácter añadido artificialmente
      {
           contador++;                         //vamos contando las posiciones del String
      }
      return contador;       // devolvemos el valor calculado
   }
}
```

- En Java un método pertenece a una clase. **No podemos escribir métodos fuera de una clase**. En java **TODO** bloque de código va incluido en alguna clase.
- La instrucción **return** se usa para devolver el valor calculado. **return** puede aparecer en cualquier parte del método, no tiene que estar necesariamente al final. Cuando un método no retorna ningún valor devuelve **void** y no es necesario el **return**.
- Desde un método se puede invocar a otro/s método/s. (En el ejemplo la llamada a **charAt(i)**)

  (Si el método es estático sólo se puede llamar a otros métodos también estáticos)
- En el **return**, Java termina la ejecución del método, retorna al punto de llamada y continúa a partir de ese punto.

Los **MÉTODOS TRADICIONALMENTE** han sido parte básica de 2 conceptos fundamentales:

- **MODULARIDAD.**
- **REUTILIZACIÓN DE CÓDIGO.**

Ambos conceptos están íntimamente relacionados, y lo que persiguen es reutilizar componentes ya desarrollados para construir un producto mayor.

Modularidad hay en todos los procesos productivos que nos rodean. Todo se hace hoy en día "por piezas", **la programación también**.

Un claro ejemplo de **modularidad** es el **hardware** El fabricante de un portátil o equipo de sobremesa no fabrica el producto al completo, ni mucho menos. Utiliza una gran cantidad de componentes desarrollados por otros fabricantes (discos, memorias, procesadores, microchips genéricos, fuentes de alimentación, etc.) Los reúne en un proyecto, al que aporta también componentes propios, y el resultado es un ordenador de la marca X, pero producido de forma modular con componentes de otros muchos fabricantes. En Programación también se trabaja así.

Estructura de un programa JAVA:

Componentes del API Java + [Componentes de terceros] + Componentes propios
(MODULARIDAD + REUTILIZACIÓN DE CÓDIGO)

La **reutilización de código** es otro concepto fundamental en el desarrollo de aplicaciones. Componentes de Software que ya estaban hechos, y funcionando bien, serán reutilizadas en otros proyectos. A veces tal cuál, y otras con leves modificaciones/ampliaciones. Volviendo al ejemplo de cómo se montan/fabrican los equipos informáticos. ¿Quién no ha **reutilizado** alguna vez componentes de un ordenador para emplearlos en otro? … (discos, módulos de memoria RAM, cables, disipadores …)

La Modularidad, desarrollar proyectos en base a componentes modulares, facilita mucho la **reutilización**. En el mundo del desarrollo de aplicaciones ambos conceptos van siempre cogidos de la mano.

En **POO** la misión fundamental de los **métodos es dar** forma al **comportamiento** de los objetos de una Clase

Antes de la **P**rogramación **O**rientada a **O**bjetos, los métodos se usaban únicamente con fines de **"utilidad".** Se organizaba el código de forma modular y se fomentaba la reutilización.

- La función de "**utilidad**" en **Java** la realizan los **métodos static de clases "helper",** como **las clase Math, Arrays y muchas otras.** (En este documento nos centraremos en este tipo de métodos - **static**)

- La otra función de los métodos en **Java** (la primordial sin duda) viene dada por la orientación a objetos del lenguaje. Los métodos son los que dan forma al **comportamiento** de los objetos, los que construyen objetos (**constructores**) y los que acceden a los atributos de un objeto (**selectores y mutadores**). Todo esto lo veremos en la **Unidad 3 de este curso**.

¿Cómo es un método Java?

Un método en Java es un bloque de código independiente dentro de una clase.
Su 1º línea o cabecera se llama **FIRMA,** y contiene su **nombre**, modificadores de acceso, parámetros que ha de recibir para realizar su función, y el valor que devuelve, o **void** si el método no devuelve ningún valor.

Volvamos al ejemplo del método **longitud** que hemos visto antes:

```
public static int longitud (String s)
{
   int contador=0;
   char c = '\u00a5';
   String aux = s + c;
   while (aux.charAt(contador) != c)
   {
        contador++;
   }
   return contador;
}
```

```
[modificadores ACCESO|Especificadores]  tipo nombreMetodo([lista parámetros]) [throws Excepciones]
{
   // código del método
   [return valor;]
}
```

[Los elementos que aparecen entre corchetes son opcionales]

- **[modificadores ACCESO|Especificadores].** public|private|protected – static|abstract. (Hablaremos de los ámbitos y de los especificadores más adelante en profundidad)
- **tipo**: indica el tipo del valor que devuelve el método mediante su instrucción **return**. Si el método no devuelve ningún valor se pone **void** y el **return** no es necesario, como sucede con el método **main()**.
- **nombreMetodo**: Seguiremos las mismas normas que para los nombres de variables: **todo minúsculas**, salvo si es compuesto (**longitud, longitudDeTabla, calculaMediaArray, etc**)
- **[Lista de parámetros]: 0 o más** separados por comas. **Son los datos de entrada que recibe el método** para hacer su función. En la **FIRMA** del método les llamamos **parámetros**, pero cuando se llama al método, a los datos que se le "pasan" o envían se les llama **argumentos.**
- **[throws Excepciones]:** indica las excepciones que puede generar el método. Sirve de aviso para tratar esas excepciones en el punto de llamada con un bloque **try-catch (Se verá más adelante)**
- **return:** se utiliza para devolver un valor. La ejecución de un método termina cuando se llega a su llave final } o cuando se ejecuta la instrucción **return**.

Con respecto al ejemplo del método longitud:

- **public** (público), significa que podrá ser llamado desde cualquier otra clase.
- **static**, significa que no hay que crear un objeto de su clase para utilizarlo.
- El parámetro que recibe **longitud** para hacer su trabajo es un **String.**
- El valor que calcula y devuelve el método (return), es la longitud de ese String.

ERRORES TÍPICOS al trabajar con métodos:

- **EN LA LLAMADA: la cabecera o FIRMA del método hay que cumplirla al 100%.**

  Correcto: longitud ("Hola soy el profe") – longitud(frase) … si frase es una variable de tipo String
  Incorrecto: longitud(5) – longitud () – longitud(f) si f no es una variables de tipo **String**

- **EN EL CÓDIGO: longitud** ha de devolver un dato de tipo **int.** Obligatoriamente en alguna parte del código debe de haber una instrucción **return (dato de tipo int).**

PASO POR VALOR Vs PASO POR REFERENCIA

En Java sólo existe el paso de parámetros por valor. Java crea una copia de cada valor en la pila (Stack) y envía esa copia de los **valores** al método.

Ejemplo:

Math.max(5,7)
Si enviamos a un método **valores finales,** el método recibe una copia de los valores y con eso trabaja

int num1 = 5;
int num2 = 7;
Math.max(num1,num2);
Si enviamos a un método **variables**, Java hace una **copia del valor** que contienen y se la envía al método.

TIPOS PRIMITIVOS | INMUTABLES -- IMPOSIBLE modificar la variable original en un método.

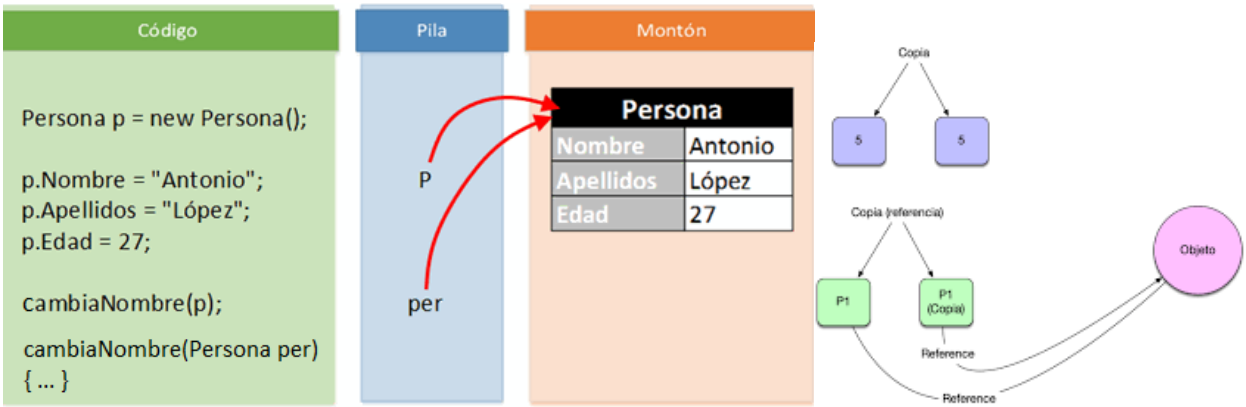
No hay que preocuparse con las variables de tipos primitivos o inmutables. No existe el paso por referencia que si existía en **otros lenguajes**. El método **max** recibe un **5** y un **7** en ambos casos y trabaja con esos valores. Pase lo que pase dentro del método, **las variables num1 y num2 no se verán afectadas y mantendrán sus valores 5 y 7 previos.** Antes de la llamada al método, Java creó una copia de num1 y num2 en la pila (Stack) y el valor original de num1 y num2 es inmutable.

TIPOS NO INMUTABLES | OBJETOS -- ES POSIBLE modificar la variable original en un método.

Otra cosa **muy distinta** es cuando pasamos a un método **OBJETOS** o cualquier tipo de dato **no inmutable**. ¿Qué pasa cuando instanciamos un nuevo objeto en la memoria **HEAP?**

```
String frase = new String();
Persona p=new Persona();
```

Java crea las variables **frase y p** en la **pila**(stack) … pero esas variables no contienen el **String frase**, ni el objeto **Persona p**, sino **la referencia** que apunta a la dirección de memoria (HEAP- "Montón") dónde "viven" esos objetos.



Al pasar objetos a métodos, Java crea una copia del "valor" de ese objeto en el stack. El "valor" de ese objeto en realidad es la referencia a la posición de memoria HEAP del objeto. Es decir, el método recibe **una copia de la REFERENCIA**, y por tanto trabaja **DIRECTAMENTE SOBRE EL OBJETO**, pudiendo modificar su contenido original. **Esto es muy versátil, pero hay que tenerlo MUY claro** y saber manejarlo correctamente.

Referencias a Objetos

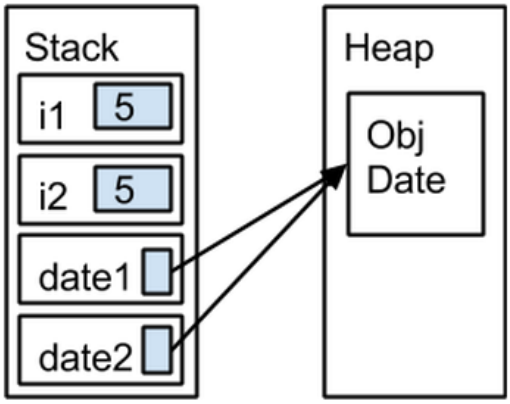
Una variable es un lugar en la memoria donde se guarda un dato. Para ser exacto, este lugar en la memoria es la *Pila o Stack*. En el caso de los datos primitivos, como en "int i = 5" hay cuatro bytes en la *Pila* donde se almacena el número 5.

Cuando se crea un objeto en Java, como en "new Date()" , el objeto se guarda en una parte de la memoria llamada *Heap*. Cuando asignamos el objeto a una variable como en "date = new Date()", lo que guardamos en *date* es la dirección de memoria *Heap* donde está el objeto.

Veamos las implicaciones de este referenciamiento indirecto con un ejemplo de código.

```
int i1 = 5;
int i2 = i1;
Date date1 = new Date();
Date date2 = date1;
```

Cómo queda en la memoria después de ejecutar las asignaciones anteriores, se ve en el gráfico siguiente. En la Pila o Stack vemos que se guardan las 4 variables. *i1 e i2* tienen copias independientes del valor 5. Diferente es el caso de *date1* y *date2,* donde estas variables guardan un puntero al mismo objeto.



Lo importante aquí, es entender que cualquier modificación dentro del objeto Date, se verá reflejada tanto en *date1* como *date2*. Este comportamiento suele traer confusiones y errores. El problema no existe en el caso de *i1 e i2*, si modificamos *i1* por ejemplo "i1 = 3", *i2* seguirá valiendo 5.

En este documento vamos a hacer una serie de ejemplos de métodos a los que pasaremos Arrays como argumentos para hacer cosas con ellos. **Hay que dejar claro que**, si modificamos un Array u otro objeto no inmutable dentro de un método, estas modificaciones **afectan al original**.

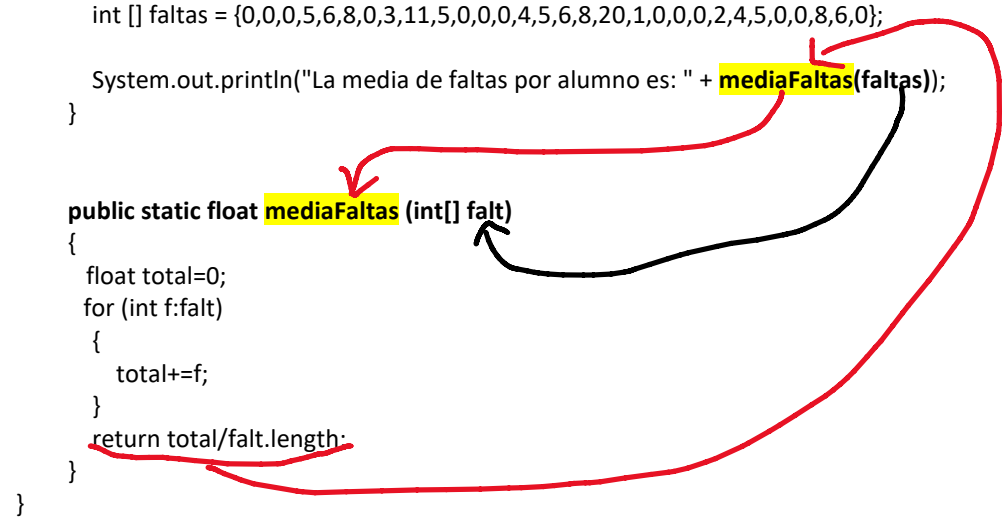
Cuando pasamos un **Objeto** a un **método**, lo que le llega al método es una copia de la ubicación del objeto en la memoria **HEAP**. Esto permite al método acceder al objeto para leer/modificar su estado. **Cualquier modificación que hagamos AFECTARÁ AL OBJETO ORIGINAL**

EJEMPLO 1: método para calcular la media de los valores contenidos en un ARRAY de enteros. La media calculada puede tener decimales así que el dato de retorno será de tipo float.

```
public class UtilsArrays1{
    public static void main(String[] args) {
       int [] faltas = {0,0,0,5,6,8,0,3,11,5,0,0,0,4,5,6,8,20,1,0,0,0,2,4,5,0,0,8,6,0};

       System.out.println("La media de faltas por alumno es: " + mediaFaltas(faltas));
    }


    public static float mediaFaltas (int[] falt)
    {
      float total=0;
      for (int f:falt)
       {
          total+=f;
       }
       return total/falt.length;
    }
}
```



En este ejemplo el array original **int[] faltas** **no se modifica** en el método. Sólo se consultan sus valores para calcular la media.

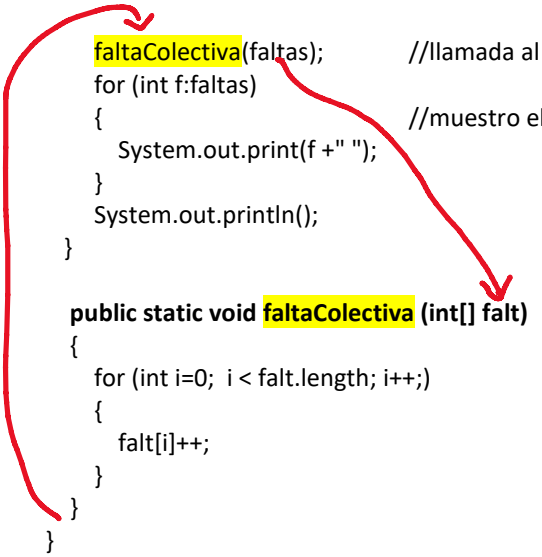
EJEMPLO 2: método para aplicar una falta colectiva a todas las posiciones del Array faltas. En este ejemplo el array original **SE MODIFICA** dentro del método.

NO tendremos el mismo Array **antes y después** de la llamada al método. Se ve al recorrer y mostrar valores después de la llamada. El paso de **OBJETOS** a métodos **PERMITE LA MODIFICACIÓN DEL ORIGINAL. No así con los tipos primitivos u objetos inmutables como los Strings.**

```
public class UtilsArrays2 {
   public static void main(String[] args) {
      int[] faltas = {0,0,0,5,6,8,0,3,11,5,0,0,0,4,5,6,8,20,1,0,0,0,2,4,5,0,0,8,6,0};

      faltaColectiva(faltas);          //llamada al metodo faltacolectiva
      for (int f:faltas)
      {                                //muestro el Array
         System.out.print(f +" ");
      }
      System.out.println();
   }

   public static void faltaColectiva (int[] falt)
   {
      for (int i=0;  i < falt.length; i++;)
      {
         falt[i]++;
      }
   }
}
```



No hay **return,** pues es un método que no devuelve valor **(void)**. Simplemente hace su labor, termina y se continúa la ejecución justo a partir del punto dónde se produjo la llamada al método.

IMPORTANTE: En estos ejemplos **estamos llamando a los métodos directa y simplemente por su nombre**. **Lógico**. El método main() y los métodos ejemplo que estamos haciendo **están dentro de la misma clase.**

Esto no es lo habitual. Sólo estamos empezando. Más adelante, el 99% de las veces llamaremos a métodos poniendo delante el nombre de una clase o un objeto, como cuando llamamos a un método de la clase Math o **String.** **Math.pow(5,2) - Math.max(2,3) - faltas.length - frase.charAt(0) – frase.toUppercase() - etc**

A veces también nos veremos obligados a incluir "**imports**" porque los métodos que vamos a utilizar están empaquetados en librerías del API o de terceros no accesibles directamente desde nuestro proyecto.

Aprenderemos a crear nuestras propias clases "**helper**" con métodos propios que empaquetaremos en un package y compilaremos en un archivo .jar, que posteriormente podremos reutilizar en otros proyectos (nosotros/as u otro/as programadores/as). Por supuesto también aprenderemos a crear clases que dan vida a objetos, y cómo veremos todo su comportamiento se describe con métodos. Desde los **constructores** hasta los **selectores** y **mutadores**, o cualquier otro comportamiento del que queramos dotar al objeto. **TODO SERÁN MÉTODOS**.

EJEMPLO 3: similar al anterior, pero permite aplicar una o más faltas colectivas a todas las posiciones del Array faltas en una sola llamada. El número de faltas a sumar se envía como parámetro. Otra vez el array original **SE MODIFICA** en el método. **NO** tenemos el mismo Array **antes y después** de la llamada al método.

```
import java.util.Scanner;

public class UtilsArray3 {
   public static void main(String[] args) {
      int[] faltas = {0,0,0,5,6,8,0,3,11,5,0,0,0,4,5,6,8,20,1,0,0,0,2,4,5,0,0,8,6,0};

      Scanner sc = new Scanner(System.in);
      System.out.println("Cuantas faltas quieres sumar a todas las posiciones del Array: ");
      int numFaltas = sc.nextInt();

      faltaColectiva(faltas,numFaltas);

      System.out.println("Listado de faltas:");
      for (int f:faltas)
      {
         System.out.print(f +" ");
      }
   }

   public static void faltaColectiva (int[] falt, int num)
   {
      for (int i=0;  i < falt.length; i++;)
      {
         falt[i]+=num;
      }
   }
}
```

EJEMPLO 4: Otra variante, que permite aplicar un número de faltas determinado a una posición del Array faltas (alumno) determinado. El número de faltas y la posición del alumno se envía como parámetro. Otra vez el array original **SE MODIFICA** en el método. **NO** tenemos el mismo Array **antes y después** de la llamada al método.

```
import java.util.Scanner;
public class UtilsArray4 {
   public static void main(String[] args) {
      int[] faltas = {0,0,0,5,6,8,0,3,11,5,0,0,0,4,5,6,8,20,1,0,0,0,2,4,5,0,0,8,6,0};

      Scanner sc = new Scanner(System.in);
      System.out.println("A qué alumn@/posición del Array le quieres añadir faltas?  ");
      int pos = sc.nextInt();
      System.out.println("Cuantas faltas le quieres poner?  ");
      int numFaltas = sc.nextInt();

      faltaAlumn(faltas,pos,numFaltas);
      System.out.println("Listado de faltas: ");
      for (int f:faltas)
      {
         System.out.print(f +" ");
      }
   }

   public static void faltaAlumn (int[] falt, int p, int num)
   {
         falt[p]+=num;
   }
}
```

EJEMPLO 5: Programa para calcular números aleatorios en el rango que nosotros queramos. De eso se encargará un **método** al que le enviamos el extremo inferior y superior para el cálculo de un número aleatorio … entre 1-10 1-1000 50-60 0-999 ... lo que queramos.
Se pasan como argumentos 2 valores enteros (inf,sup).

```
import java.util.Scanner;
public class AzarConMetodo {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.println("Teclea el intervalo para generar enteros aleatorios");
      System.out.print("Primero el extremo inferior: ");
      int inf = sc.nextInt();
      System.out.print("Ahora el extremo superior: ");
      int sup = sc.nextInt();
      System.out.print("¿Cuantos enteros aleatorios Quieres: ");
      int n = sc.nextInt();
      for (int i = 1; i <= n; i++) {
         System.out.print(intAleatorio(inf,sup) + " ");
      }
   }
   public static int intAleatorio(int inferior, int superior)
   {
      int a = inferior + (int) (Math.random() * ((superior - inferior)+1));
      return a;
    }
}
```