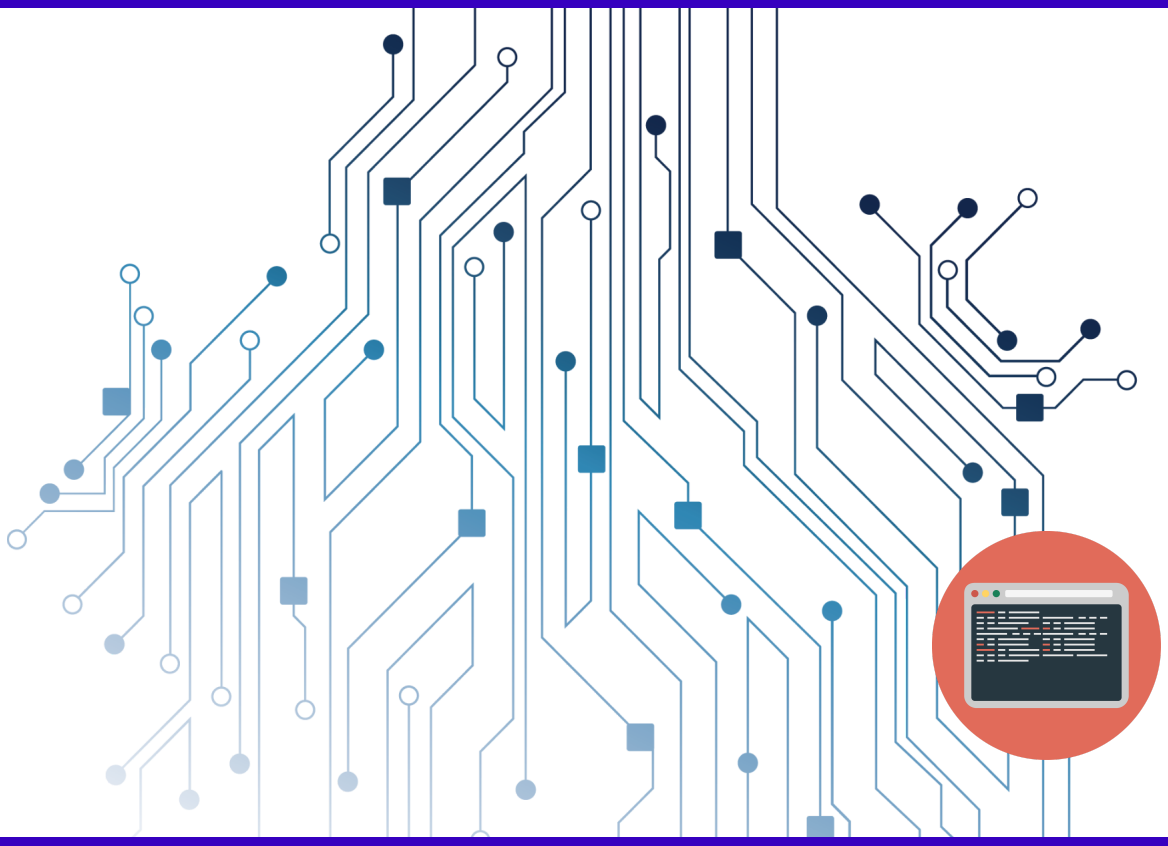
CFGS APLICACIONES MULTIPLATAFORMA - PROGRAMACIÓN

UNIDAD 2

ELEMENTOS LÓGICOS



DIEGO VALERO ARÉVALO
BASADO EN APUNTES Y EJERCICIOS PROPORCIONADOS POR
Mª CARMEN DÍAZ GONZÁLEZ - IES VIRGEN DE LA PALOMA

U2 - **ELEMENTOS LÓGICOS**
ÍNDICE

2.1 - ESTRUCTURAS DE CONTROL

Las **estructuras de control** dirigen el flujo de ejecución lineal del código y permiten alterarlo según las necesidades del programa.

Estas estructuras van de la mano con la **lógica de programación** y por tanto necesitan **variables** para poder funcionar. Para poder manipular ese flujo haremos **operaciones lógicas** con esas variables que definirán unos **comportamientos u otros** dependiendo de las condiciones que establezcamos.

Cada estructura se compone por una **cabecera** que la define y un **ámbito** donde pondremos las **instrucciones**. Este ámbito es la **zona delimitada por llaves** (**{ }**) justo después de la cabecera de la estructura.

Las llaves se pueden **omitir** si dentro de una estructura **sólo hay una única instrucción**.

JAVA dispone de diferentes estructuras de control, entre las que encontramos:

Selección o condicionales	Repetición o bucles	De salto

>> 2.1.1- ESTRUCTURAS DE SELECCIÓN (CONDICIONALES)

Las **estructuras de selección o condicionales** nos permiten establecer **caminos alternativos dependiendo de si se cumplen o no una o más condiciones**.

Las condiciones siempre deben dar como resultado un **valor booleano** (`true` o `false`), en cualquier otro caso se producirá un error de compilación. Tenemos **cuatro** estructuras principales:

`if`	`if-else`	`if-else if-else`	`switch`

· 2.1.1.1- IF

`if` permitirá ejecutar las instrucciones que contenga sólo **si se dan las condiciones** que se le indican.

```
if(condición){
   instrucciones...
}
```

EJEMPLO

.java

```
1  if(llueve){
2     ...
3  }
```

En este caso, sólo se ejecutarán las instrucciones si el valor de `llueve` es `true`.

Puedes establecer las condiciones igualando a true o false o de **manera abreviada**. Esta última es una manera de optimizar la escritura de código:

```
if(cond==true) = if(cond)
if(cond==false) = if(!cond)
```

· 2.1.1.2- IF-ELSE

Si queremos considerar una condición y también el caso de que no se cumpla, podemos añadir la estructura **`else`**, que se ejecutará en caso de que no se cumpla la condición de **`if`**. El **`else`** no es nunca obligatorio, dependerá de nuestro código.

```
if(condición){
   instrucciones...
}
else{
   instrucciones...
}
```

EJEMPLO

.java

```
1  if(llueve){
2     ...
3  }
4  else{
5     ...
6  }
```

En este caso, si no se cumple que **`llueve`** sea **`true`**, no ejecutará las instrucciones dentro del **`if`** y saltará directamente a las del **`else`**.

· 2.1.1.3- IF-ELSE IF-ELSE

Si queremos considerar más de una condición a la vez usaremos la estructura **`else if`**, en la cual podemos establecer diferentes condiciones para considerar casos específicos. Esta estructura también puede acompañarse con **`else`** si lo necesitamos.

```
if(condición){
   instrucciones...
}
else if(condición){
   instrucciones...
}
...
else{
   instrucciones...
}
```

EJEMPLO

.java

```
1  if(llueve){
2     ...
3  }
4  else if(!llueve  && frio){
5     ...
6  }
7  else{
8     ...
9  }
```

En este caso, si no se cumple que **`llueve`** sea **`true`**, comprobará si es **`false`** y si además (**`&&`**) **`frio`** es **`true`**. Si no se cumple tampoco, bajaría hasta el **`else`** directamente.

· 2.1.1.4- CONDICIONALES ANIDADOS

Gran parte de las estructuras que componen un código se pueden **anidar**, es decir, se pueden incluir dentro de otras estructuras. Los condicionales pueden anidarse para

```
if(condición){
   instrucciones...
}
```

comprobar otra condición si se ha dado una en específico.

```
else{
    if(condición){
        instrucciones...
    }
}
```

EJEMPLO

.java

```
1  if(x == y){
2      System.out.println("x e y son iguales");
3  }
4  else{
5      if(x > y){
6          System.out.println("x es mayor que y");
7      }
8      else{
9          System.out.println("y es mayor que x");
10     }
11 }
```

Seguramente ya hayas pillado cómo funciona el flujo de código a través de los condicionales. En este ejemplo puedes ver mejor diferentes casos y qué código se ejecutaría según cada uno.

Si `x` valiese `4` e `y` valiese `7`, ¿qué salida del programa tendríamos?

Si quisiéramos optimizar el ejemplo anterior, podríamos hacerlo de la siguiente manera:

EJEMPLO

.java

```
1  if(x == y){
2      System.out.println("x e y son iguales");
3  }
4  else if(x > y){
5      System.out.println("x es mayor que y");
6  }
7  else{
8      System.out.println("y es mayor que x");
9  }
```

En este otro ejemplo podemos comprobar el funcionamiento de los condicionales:

EJEMPLO

.java

```
1  if(x > 0){
2      if(x < 10){
3          System.out.println("x está entre 0 y 10");
4      }
5  }
```

Que podemos simplificar más aún con el uso de los **operadores lógicos**:

EJEMPLO

.java

```
1  if((x > 0) && (x < 10)){
2      System.out.println("x está entre 0 y 10");
3  }
```

Al proceso de optimizar un código se le llama **refactorización**. Existen muchas maneras de refactorizar u optimizar un programa. Con estos ejemplos has podido comprobar cómo podemos

reducir en líneas escritas y agrupar condiciones para, aparte de que el código sea más óptimo, que al leerlo nos cueste lo mínimo comprender lo que hace cada estructura. **Ten este objetivo siempre en mente al programar**.

· 2.1.1.5- SWITCH

¿Qué pasa si queremos **comprobar muchos valores distintos para la misma variable**? ¿Hacemos trescientos `if-else if`? No es necesario (ni óptimo). Para estos casos tenemos una estructura que nos lo resuelve:

```
switch(var){
   case valor:
      instrucciones…
      break;
   ...
   default:
      instrucciones...
      break;
}
```

La estructura `switch` permite usar una variable y definir casos específicos para la misma en una sola estructura. Se compone de los siguientes elementos:

`switch(var){}`	Cabecera de la estructura `switch`, donde colocamos la **variable** (`var`) de la que queremos comprobar su valor.

`case valor:`
`  instrucciones...`
`  break;`
	Dentro de un `case`, especificamos en `"valor"` el caso específico para `var`. Si se da que `var=="valor"`, entrará en el `case` y ejecutará las instrucciones. Para que el código sólo ejecute ese `case` en concreto y ninguno más, **debemos cerrarlo siempre con** `break`, si no el código continuaría hacia los que hay debajo. El `break` es una **estructura de salto** que veremos en el apartado *2.1.3*.

`default:`
`  instrucciones...`
`  break;`
	Si se diese que `var` **no tuviese ninguno de los valores guardados en los** `case`, se ejecutaría la sentencia `default`. Esta no es obligatoria aunque sí recomendada.

EJEMPLO

`.java`

```
1  switch(num){
2     case 1:
3        System.out.println("Uno.");
4        break;
5     case 2:
6        System.out.println("Dos.");
7        break;
8     case 3:
9        System.out.println("Tres.");
10       break;
11    default:
12       System.out.println("No está en los casos.");
13       break;
14 }
```

Si `num` valiese 2...

`Console`

```
Dos.
```

Si tenemos una **misma instrucción para diferentes valores**, podemos **agrupar** estos en un mismo `case`:

EJEMPLO

.java

```java
switch(nota){
   case 0, 1, 2, 3, 4:
      System.out.println("Estás suspenso.");
      break;
   case 5, 6, 7, 8, 9, 10:
      System.out.println("Estás aprobado.");
      break;
   default:
      System.out.println("Nota no válida.");
      break;
}
```

Switch permite **comprobar tanto valores numéricos como de texto**, aunque para casos de texto (`string`) existen mejores formas que veremos en otra unidad.

>> 2.1.2- ESTRUCTURAS DE REPETICIÓN (BUCLES)

Las **estructuras de repetición** o **bucles** se basan en condiciones para ejecutar un mismo bloque de código mientras se cumpla esta. Pueden usarse para establecer un número determinado de veces que queramos mostrar algo u operar ciertas instrucciones.

Se llama **iteración** a cada vez que se inician las instrucciones de un bucle hasta que se finalizan, o más sencillamente, cada **"vuelta" del bucle**.

Existen **tres** estructuras de repetición principales:

while	do-while	for

· 2.1.2.1- WHILE

`while` ejecutará las instrucciones mientras se cumpla la condición. En el momento que deje de cumplirse, saldrá del bucle para continuar con el resto del código.

```java
while(condición){
   instrucciones...
}
```

EJEMPLO

.java

```java
int valor = 0;

while(valor < 10){
   valor++;
   System.out.println("valor: "+valor);
}
```

Console

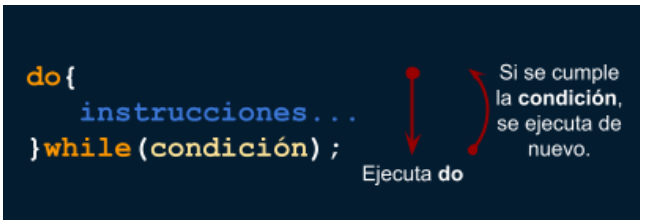
```
valor: 1
valor: 2
valor: 3
valor: 4
valor: 5
valor: 6
valor: 7
valor: 8
valor: 9
valor: 10
```

Con un bucle `while` podemos ejecutar la misma condición las veces que haga falta. Aquí por ejemplo, se mostrará el `syso` mientras que `valor` sea menor que `10`. Si seguimos el flujo, vemos que entra en el bucle porque se cumple la condición, suma 1 a `valor` e imprime la línea, y como se sigue cumpliendo la

condición, seguirá repitiéndose hasta que el valor no sea **menor que** `10`.

· 2.1.2.2- DO-WHILE

Al igual que **`while`**, es un bucle que repetirá su contenido mientras se cumpla la condición, con la variante de que **esta no se comprueba hasta que se ha hecho la primera iteración**, entrando directamente y ejecutando el código por lo menos una vez.



EJEMPLO

.java

```java
int valor = 10;

do{
   valor++;
   System.out.println("valor: "+valor);
}while(valor < 10)
```

Console

```
valor: 11
```

Aquí podemos ver que como la primera vez no se necesita condición, se ejecuta el código y después se comprueba la condición exigida. Como **`valor`** vale `11` tras esta primera iteración, no continúa pero las instrucciones de **`do`** se han ejecutado.

· 2.1.2.3- FOR

Este bucle incorpora un contador que se incrementa en cada iteración y su condición se basa en este. Los argumentos que controlan un **`for`** son **tres**:

```java
for(int i = x; i < y; i++){
   instrucciones...
}
```

`int i = x;`	En la **inicialización** declaramos la **variable que servirá de contador** con su **valor inicial**. Esta variable sólo funciona dentro del **`for`**, y se suele nombrar como **`i`**.
`i < y;`	La **condición** especifica cuántas iteraciones tendrá el **`for`**. Podemos especificar un valor numérico directamente, otra variable e incluso longitudes de **`string`** o de una **`colección`**.
`i++`	El **incremento** (o **decremento**) del bucle después de cada iteración.

EJEMPLO

.java

```java
for(int i = 1; i <= 10; i++){
   System.out.println("valor: "+i);
}
```

Console

```
valor: 1
valor: 2
valor: 3
valor: 4
valor: 5
valor: 6
valor: 7
valor: 8
valor: 9
valor: 10
```

Como ves, hemos hecho lo mismo que en el ejemplo del bucle **`while`**, pero **el contador está en el bucle** y no nos hace falta escribirlo como instrucción.

¡CUIDADO! Si una condición nunca deja de cumplirse, el programa nunca saldrá del bucle y éste se ejecutará indefinidamente. Esto se conoce como **bucle infinito**.

¿Es malo? **No siempre**, a veces queremos que el programa finalice sólo con una condición específica, por ejemplo en un **menú** donde se piden valores para diferentes acciones y hasta que el usuario no lo decida el programa no se cierra.

>> 2.1.3- ESTRUCTURAS DE SALTO

Las **estructuras de salto** son palabras reservadas que permiten manipular la ejecución de cualquiera de los bucles que hemos visto. Existen dos:

`break`	`continue`

· 2.1.3.1- BREAK

Con la estructura **`break`** hacemos que se "rompa" la ejecución del bucle y el programa salga del mismo hasta la siguiente sentencia de código.

```
break;
```

EJEMPLO

.java

```
1  for(int i = 1; i <= 10; i++){
2     if(i == 5){
3        break;
4     }
5     System.out.println(" n"+i+",");
6  }
```

Console

```
n1, n2, n3, n4,
```

Si **`i`** tiene valor **`5`**, entra en el **`if`**, que ejecuta **`break`** y la ejecución sale del bucle.

· 2.1.3.2- CONTINUE

`continue` lo que hace es saltarse todas las instrucciones siguientes del bucle y comenzar en la siguiente iteración.

```
continue;
```

EJEMPLO

.java

```
1  for(int i = 1; i <= 10; i++){
2     System.out.println(" n");
3
4     if(i == 5){
5        continue;
6     }
7
8     System.out.println(i+",");
9  }
```

Console

```
n1, n2, n3, n4, n n6, n7, n8, n9, n10,
```

Si **`i`** tiene valor **`5`**, entra en el **`if`**, que ejecuta **`continue`**, pasando directamente a la siguiente vuelta.

Observa que la **línea 2**, que es anterior al `continue`, sí que se ejecuta en cualquier caso.

2.2 - MÉTODOS O FUNCIONES

Si queremos hacer un programa que realice **diferentes tareas**, lo mejor es hacer caso al dicho "*divide y vencerás*".

Un **método** o **función** es un **bloque de código que se dedica exclusivamente a realizar una de esas tareas** y está **separado del código principal (`main`)**, desde el cual podemos **invocar o "llamar"** al método cuando lo necesitemos. Separar el código en métodos también permite **reutilizar código** en vez de copiar-pegar o reescribirlo, algo que es muy útil a la hora de optimizar la ejecución de un programa.

Como hemos dicho, estos métodos van a realizar tareas diferentes, que en general será **devolver un resultado** en forma de **tipo de dato** (un número, un valor booleano,...).

Para comprender mejor los métodos, podemos echar un vistazo a la API de JAVA que incluye muchos y diferentes métodos de los cuales ya hemos visto algunos, como **Random()**, **print()**, **next()**,...

>> 2.2.1- TRABAJAR CON MÉTODOS

· 2.2.1.1- DECLARACIÓN DE MÉTODOS

Para declarar un método debemos usar la siguiente estructura:

```
public / private [static] tipo / void nombreDelMetodo([parámetro/s]){
   instrucciones…
   [return value;]
}
```

`public / private [static]`	Los **modificadores** definen el uso del método. Por ahora nos centraremos en usar `public static`. Más adelante veremos más tipos.
`tipo / void`	Debemos indicar el **tipo de dato** que devolverá el método. Si es un método que no devuelve datos, se utiliza `void`.
`nombreDelMetodo`	El **nombre** que queremos dar al método. Debe ser un **nombre único** y **no puede llamarse como la clase en la que estamos trabajando** (por ahora).
`[parámetro/s]`	Los **parámetros** son los **datos que le llegarán desde fuera al método y con los que podrá trabajar**, que se indican con el **tipo** y un **nombre** que le queramos dar a cada uno. `(tipo1 nombre1, ..., tipoN nombreN)` Un método **puede necesitar o no parámetros**, y lo ideal es que **no use más de tres**.

	Cuando se declaran los parámetros de un método, al llamarlo **se deben escribir estos en el mismo orden**.
`{ instrucciones... }`	Las **instrucciones** o **cuerpo** contiene el **código para realizar la tarea asignada**. Puede usar las **variables de los parámetros** e incluso **crear nuevas variables** para usarlas sólo en el método.
`[return value;]`	La sentencia `return` se debe escribir **sólo si el método no es** `void`, y se sitúa al **final del método**. Es la línea que devuelve el valor que nos interesa después de haber ejecutado el método.

⚠ Las **variables** de los métodos **no pueden usar variables externas a no ser que se le pasen como parámetros**, ni las variables **creadas dentro de un método pueden usarse fuera de él**. En la **unidad 3, *Programación Orientada a Objetos (POO)*** veremos más sobre esto, llamado **Principio de Encapsulación**.

Veamos algunos ejemplos de métodos:

EJEMPLO

.java

```java
public static void imprimeHolaMundo(){
    System.out.println("Hola Mundo");
}
```

Este es un método muy sencillo, que al llamarlo simplemente sacaría por consola las palabras `Hola Mundo`. Como no necesita ningún dato externo para funcionar, **no tiene parámetros**, y como no necesitamos que nos devuelva ningún dato, su tipo es `void`.

EJEMPLO

.java

```java
public static void imprimePalabra(String palabra){
    System.out.println("Tu palabra: "+palabra);
}
```

En este otro método, vemos que **le pasamos un parámetro** y después lo usamos como parte del código.

⚠ Los métodos que ves aquí son ejemplos ilustrativos, pero ten en cuenta que un buen método en una clase externa al `main` **nunca tendrá en sus instrucciones impresión de líneas por consola ni pedirá datos por teclado**, porque si son métodos que se van a usar para múltiples cosas, es mejor que sólo devuelvan el dato y ya trabajar con él en el código principal.

EJEMPLO

.java

```java
public static double maximo(double v1, double v2){
    double max;
    if(v1 > v2){
        max = v1;
    }
    else{
        max = v2;
    }
    return max;
}
```

Esta función se llama `maximo`, tiene dos parámetros de entrada de tipo `double` llamados `v1` y `v2` y devuelve un dato de tipo `double`. Cuando la llamemos **calculará el máximo entre `v1` y `v2` y lo devolverá**.

· 2.2.1.2- LLAMAR A MÉTODOS

Para **llamar a un método** simplemente debemos escribir su nombre en una línea que esté dentro de otro método, por ejemplo desde el **`main`**. Si necesita parámetros, debemos escribir las **variables o datos** que queremos usar **entre los paréntesis**, y si el método es **`void`**, los dejaremos vacíos.

```
miMetodo([parametro]);
```

Al ejecutar un programa con métodos, el flujo **saldrá por la línea donde llamamos a uno y llegará hasta el mismo**, hará las instrucciones correspondientes y al terminar volverá a la misma línea para seguir con el resto del código principal.

Si queremos **conservar** el resultado de un método para usarlo más adelante, es importante que lo **recojamos en una variable**. Los métodos se pueden usar **aislados**, **combinados** e incluso **sobrecargados**.

EJEMPLO

miClase.java

```
1  public class MiClase{
2     public static void main(String[] args){
3        int a = 10;
4        int b = 40;
5        int calculo1;
6        int calculo2;
7        calculo1 = doble(a);
8        calculo2 = suma(a, b);
9        System.out.println(calculo1+", "+calculo2);
10    }
11
12    public static int doble(int num){
13       int resultado;
14       resultado = num*2;
15       return resultado;
16    }
17
18    public static int suma(int num1, int num2){
19       int resultado;
20       resultado = num1+num2;
21       return resultado;
22    }
23 }
```

Console

```
20, 50
```

Vemos que en esta clase hay un método llamado **`doble`**, que al llamarlo desde el **`main`** ejecuta un código que **multiplica por 2** el valor que le pasemos; y otro llamado **`suma`**, que como dice, **suma dos valores** pasados por parámetros.

Si ya has practicado programando, prueba a hacer
REFACTORIZACIÓN AVANZADA

¿Hay alguna manera de **optimizar este código** para **no usar tantas variables**? Echa un vistazo:

miClase.java

```
1  public class MiClase{
2     public static void main(String[] args){
3        int a = 10;
4        int b = 40;
5        System.out.println(doble(a)+", "+suma(a, b)); → Esto se llama
6                                   sobrecarga de métodos. Es cuando llamamos a unos métodos
7                                dentro de otros. Lo veremos mejor en la unidad 4, Continuación de POO.
8     }
9
10    public static int doble(int num){
11       return num*2;
```

```java
    }

    public static int suma(int num1, int num2){
        return num1+num2;
    }
}
```

>> 2.2.2- RECURSIVIDAD

La **recursividad** es una técnica de programación que permite ejecutar un bloque de código dentro de sí mismo tantas veces como queramos.

Cuando un método se llama a sí mismo, se asigna espacio en la **pila de ejecución** para las nuevas variables locales y parámetros. Al volver de una llamada recursiva, se recuperan de la pila las variables locales y los parámetros antiguos y la ejecución se reanuda en el punto de la llamada al método.

Siempre se debe establecer dentro del método un **punto de parada**, ya que si no entraríamos en un **bucle infinito**.

Veamos un ejemplo para intentar entender mejor el concepto de recursividad:

EJEMPLO

Queremos crear un método que **calcule el factorial de un número**, que si no recuerdas cómo es, es multiplicar un número por sí mismo y todos sus anteriores hasta el 1:

5! = 5x4x3x2x1

Así podemos sacar el **punto de parada**: que el número a multiplicar sea igual a 1, o de otra manera, que si no es mayor de 1 devuelva este mismo.

.java

```java
public int factorial(int num){
    int valor;
    if(num>1){
        valor = num * factorial(num-1);
    }
    else{
        valor = 1;
    }
    return valor;
}
```

El orden de ejecución para `factorial(5)` sería el siguiente:

`valor = 5 * factorial(5-1)` > Ejecutamos el factorial de 4
`valor = 4 * factorial(4-1)` > Ejecutamos el factorial de 3
`valor = 3 * factorial(3-1)` > Ejecutamos el factorial de 2
`valor = 2 * factorial(2-1)` > Ejecutamos el factorial de 1
`valor = 1` > Como no se cumple que `num>1`, hacemos que valor sea 1, y se empieza a devolver el resto de valores de la pila que estaban en espera.
`valor = 2 * 1` > Se calcula 2 * el valor devuelto de `factorial(1)`
`valor = 3 * 2` > Se calcula 3 * el valor devuelto de `factorial(2)`
`valor = 4 * 6` > Se calcula 4 * el valor devuelto de `factorial(3)`
`valor = 5 * 24` > Se calcula 5 * el valor devuelto de `factorial(4)`

Las llamadas al método por recursividad se van acumulando en la pila de ejecución del programa hasta que se cumple la condición de salida y entonces se devuelven los métodos que están en espera.

U2 - **BATERÍA DE EJERCICIOS**

1	Crear un programa que indique si un número introducido es positivo o negativo.
2	Crear un programa que solicite dos números e indique cuál es el mayor de los dos.
3	Crear un programa que indique si un número introducido por teclado es par y/o divisible entre 5.
4	Crear un programa que solicite dos números y los reste, de forma que siempre se reste el menor al mayor, quedando un número positivo. Si los números son iguales debe mostrar un mensaje y no realizar ninguna operación.
5	Crear un programa que solicite dos números y los muestre ordenados de mayor a menor.
6	Crear un programa que solicite tres números enteros y muestre cual es el mayor utilizando if-else. Tener en cuenta que los números introducidos pueden ser iguales entre ellos.
7	Crear un programa que solicite un número entre 0 y 999, y muestre cuántas cifras tiene.
8	Crear un programa al que se le pasa una temperatura corporal en grados centígrados y devuelve los siguientes mensajes dependiendo del valor: ◦ Si la temperatura es menor que 34 o mayor de 41, muestra “Vuelva a tomar la temperatura, posible error”. ◦ Si la temperatura está entre 34,1 y 37,1 muestra “temperatura correcta”. ◦ Si la temperatura está entre 37,2 y 38,3 muestra “Acuda al médico, puede estar enfermo”. ◦ Si la temperatura está entre 38,4 y 40,9 muestra “URGENTE: acuda al médico”. Utilizar la estructura if-else.
9	Crear un programa en el que solicite un número entero del 1 al 7, y nos muestre por pantalla el correspondiente día de la semana. Utilizar la estructura if-else.
10	Crear un programa que pida una hora por teclado y que muestre buenos días, buenas tardes o buenas noches según la hora. Se utilizarán los tramos de 6 a 12, de 13 a 20 y de 21 a 5 respectivamente. Sólo se tienen en cuenta las horas, los minutos no se deben introducir por teclado
11	Crear un programa en el que solicite un número entero del 1 al 7, y nos muestre por pantalla el correspondiente día de la semana. Utilizar la estructura switch.
12	Crear un programa en el que solicite un número entero del 1 al 12, y nos muestre por pantalla a qué mes corresponde. Utilizar la estructura switch.
13	Crear un programa que solicite una nota del 1 al 10 al usuario, y le diga si es aprobado si 5 o más o suspenso menos de 5. Utilizar la estructura switch.
14	Crear un menú de una calculadora donde las opciones sean: ◦ 1. Sumar ◦ 2. Restar ◦ 3. Multiplicar ◦ 4. Dividir. Se pedirán dos números al usuarios y según la opción seleccionada se realizará la operación y mostrará el resultado por pantalla.
15	Crea un programa que genere números aleatorios en [0,30] y los muestre por pantalla hasta que salga uno mayor que 25.
16	Crear un programa que muestre los números pares del 1 al 100 utilizando bucle for.
17	Crear un programa que muestre los números pares del 1 al 100 utilizando bucle while.

18	Crear un programa que muestre los números pares del 1 al 100 utilizando bucle do-while.
19	Crear un programa que muestre los números del 550 al 340, contando de 20 en 20 hacía atrás utilizando bucle for.
20	Crear un programa que muestre los números del 550 al 340, contando de 20 en 20 hacía atrás utilizando bucle while.
21	Crear un programa que muestre los números del 550 al 340, contando de 20 en 20 hacía atrás utilizando bucle do-while .
22	Crear un programa para acertar un número aleatorio de 3 cifras, para lo que tendremos 5 oportunidades.
23	Crear un programa que muestre la tabla de multiplicar de un número que se introduce por teclado.
24	Crear un programa que calcule la media de un conjunto de números positivos introducidos por teclado. A priori, el programa no sabe cuántos números se introducirán. El usuario indicará que ha terminado de introducir los datos cuando meta un número negativo.
25	Crear un programa que muestre los n primeros términos de la serie de Fibonacci. El primer término de la serie de Fibonacci es 0, el segundo es 1 y el resto se calcula sumando los dos anteriores, por lo que tendríamos que los términos son 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144... El número n se debe introducir por teclado.
26	Crear un programa que pida una base y un exponente (entero positivo) y que calcule la potencia. (No usar la clase Math, utilizar bucles).
27	Crear un programa que diga si un número introducido por teclado es o no primo. Un número primo es aquel que sólo es divisible entre él mismo y la unidad.
28	Crear un programa que obtenga los números enteros comprendidos entre dos números introducidos por teclado y validados como distintos, el programa debe empezar por el menor de los enteros introducidos e ir incrementando de 7 en 7.
29	Crear un programa que pinte una pirámide por pantalla. La altura se debe pedir por teclado. El carácter con el que se pinta la pirámide también se debe pedir por teclado.
30	Realiza un programa que vaya pidiendo números hasta que se introduzca un numero negativo y nos diga cuantos números se han introducido, la media de los impares y el mayor de los pares. El número negativo sólo se utiliza para indicar el final de la introducción de datos pero no se incluye en el cómputo.
31	Escribe un programa que permita ir introduciendo una serie indeterminada de números mientras su suma no supere el valor 10000. Cuando esto último ocurra, se debe mostrar el total acumulado, el contador de los números introducidos y la media.
32	Crear una calculadora con un menú donde las opciones sean: 1. Sumar 2. Restar 3. Multiplicar 4. Dividir 0. Salir. Se pedirán dos números al usuarios y según la opción seleccionada se realizará la operación y mostrará el resultado por pantalla. Cuando realize la operación, deberá mostrar al usuario el menú, y volver a realizar la operación indicada, hasta que el usuario pulse 0.
33	Crear un programa que obtenga independientemente (no en el mismo bucle) la suma de los números pares y de los impares dentro de los valores del 1 al 40.
34	Escribir un programa en Java que lea un número entero por el teclado e imprima todos los números impares menores que él.

35 Crear un programa que muestre por pantalla todos los números del 1 al 10 excepto el 5. Utilizar la sentencia continue.

36 Crear un programa que muestre los números pares del 1 al 10. Utilizar la sentencia continue.

37 Crear un programa para acertar un número aleatorio del 1 al 10, para lo que tendremos 5 oportunidades. Utilizar la sentencia break.

38 Escriba un programa que solicite del usuario un número N y luego muestre por pantalla la siguiente ejecución:

```
1
1 2
1 2 3
12 3 4
.........
1 2 3 4 5 6 ...... N
```

39 Escriba un programa que lea un mes en número (1 para enero, 2 para febrero, etc.) y un año e indique el número de días de ese mes. Recuerde que un año es bisiesto si es divisible por cuatro, excepto cuando es divisible por 100, a no ser que sea divisible por 400. Así, 1900 no fue bisiesto, pero el año 2000 sí lo fue.

40 Realizar un programa que muestre por pantalla los primeros 5 números pares a partir de un número dado.

41 Transforma el siguiente bucle for en un bucle while:

```
for (int i=5; i<15;i++){
   System.out.println(i);
}
```

42 Ir pidiendo por teclado una serie de números enteros e irlos sumando. Se deja de pedir números al usuario cuando la cantidad supera el valor 50. Escribir por pantalla la suma fianl de todos los números introducidos.

43 Algoritmo que pida caracteres e imprima 'VOCAL' si son vocales y 'NO VOCAL' en caso contrario, el programa termina cuando se introduce un espacio.

44 Escribe un programa que pida el limite inferior y superior de un intervalo. Si el límite inferior es mayor que el superior lo tiene que volver a pedir. A continuación se van introduciendo números hasta que introduzcamos el 0. Cuando termine el programa dará las siguientes informaciones
◦ La suma de los números que están dentro del intervalo (intervalo abierto).
◦ Cuantos números están fuera del intervalo.
◦ Informa si hemos introducido algún número igual a los límites del intervalo.

45 Leer por teclado un número entero N no negativo y mostrar el factorial de todos los números desde 0 hasta N.

46 El factorial de un número entero se expresa mediante el símbolo '!' y se define de la siguiente forma:

◦ Si n = 0 entonces 0! = 1

◦ Si n > 0 entonces

◦ n! = n * (n – 1) * (n – 2) * …. * 3 * 2 * 1

◦ Por ejemplo, n = 5, entonces 5! = 5 * 4 * 3 * 2 * 1 = 120

```
Introduce un número > 0: 10
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

47 Define un array de 10 números enteros con nombre num y asigna los valores según la tabla que se muestra a continuación. Muestra el contenido de todos los elementos del array. ¿Qué sucede con los valores de los elementos que no han sido inicializados?

Valor	39	4			-5				32	
Índice	0	1	2	3	4	5	6	7	8	9

48 Escribe un programa que lea 10 números por teclado y que luego los muestre en orden inverso, es decir, el primero que se introduce es el último en mostrarse y viceversa.

49 Crea un programa en el que se definan 2 arrays de 15 números, en el primero de ellos se almacenarán números aleatorios del 1 al 100, y el segundo se rellenará con el cuadrado de los números del primer array. Finalmente, mostrar el contenido de los dos arrays dispuesto en dos columnas: Número — Cuadrado.

50 Realiza un programa que pida 12 números enteros y que luego muestre esos números junto con la palabra “par” o “impar” según proceda.

51 Escribe un programa que genere 20 números enteros aleatorios entre 1 y 100 y que los almacene en un array. El programa debe ser capaz de pasar todos los números pares a las primeras posiciones del array (del 0 en adelante) y todos los números impares a las celdas restantes. Utiliza arrays auxiliares si es necesario.

52 Crear un programa para acertar un número aleatorio del 1 al 10, para lo que tendremos 5 oportunidades. Al final independientemente de si se ha acertado o no, se deberán mostrar todos los números que el usuario haya introducido (se pueden guardar en un array) y se mostrará también el número aleatorio generado.

53 Crear un programa en el que se genere un array con 50 números aleatorios del 1 al 1000, y muestre por pantalla el array completo cual es el número menor y el mayor.

54 Crear un programa que pida 10 números reales por teclado, los almacene en un array, y muestre la suma de todos los valores.

55 Crea un programa que pida 8 números enteros por teclado, los almacene en un array y luego muestre por separado la suma de todos los valores positivos y negativos.

56 Crea un programa que pida dos valores enteros N y M, luego cree un array de tamaño N, escriba M en todas sus posiciones y lo muestre por pantalla.

57 Crea un programa que pida dos valores enteros P y Q, luego cree un array que contenga todos los valores desde P hasta Q, y lo muestre por pantalla.

58 Crea un programa que cree un array con 100 números aleatorios entre 0 y 50, y luego le pida al usuario un valor R (entre 0 y 50). Por último, mostrará cuántos valores del array son igual o superiores a R.

59 Crea un programa que cree dos arrays de enteros de tamaño 100. Luego introducirá en el primer array todos los valores del 1 al 100. Por último, deberá copiar todos los valores del primer array al segundo array en orden inverso, y mostrar ambos por pantalla.

60 Crea un programa que cree un array de números enteros e introduzca la siguiente secuencia de valores: `1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5,` etc. hasta introducir 10 diez veces, y luego la muestre por pantalla.

Nota: Ninguna función mostrará nada por pantalla a no ser que se diga lo contrario.

61 Realiza una función que elimine los decimales de un número decimal. Pruébalo generando un número aleatorio con decimales y mostrando por pantalla el número sin decimales.

62 Escribe un programa que pida la edad por teclado y muestre por pantalla si eres mayor de edad o no. Implementa y utiliza la función:

	`boolean esMayorEdad(int a)` **Devuelve verdadero si a>=18, falso en caso contrario**
63	Escribe un programa que pida dos números enteros por teclado y muestre por pantalla cual es el mínimo. Implementa y utiliza la función: `int minimo(int a, int b)` **Devuelve el menor entre a y b (no utilizar la clase Math)**
64	Escribe un programa que pida cinco precios y muestre por pantalla el precio de venta de cada uno tras aplicarle un 21% de IVA. Implementa y utiliza la función: `double precioConIVA(double precio)` **Devuelve el precio tras sumarle un 21% de IVA**

65	Escribe un programa que pida el ancho y alto de un rectángulo y muestre por pantalla su área y su perímetro. Implementa y utiliza las funciones:
	`double perimetroRectangulo(double ancho, double alto)`
	Devuelve el perímetro double
	`areaRectangulo(dou ble ancho, double alto)`
	Devuelve el área

66	Realiza un programa que lea una fecha introduciendo el día, mes y año por separado y nos diga si la fecha es correcta o no. Supondremos que todos los meses tienen 30 días. Se debe crear una función donde le pasemos los datos y devuelva si es correcta o no.
67	Realiza un programa que escriba la tabla de multiplicar de un número introducido por teclado. Para ello implementa una función que reciba como parámetro un número entero y muestre por pantalla la tabla de multiplicar de dicho número.
68	Escribe un programa que imprima las tablas de multiplicar del 1 al 10. Implementa una función que reciba un número entero como parámetro e imprima su tabla de multiplicar.

69	Escribe una función que muestre por pantalla un triángulo como el del ejemplo. Deberá recibir dos parámetros: el carácter que se desea imprimir y el número de líneas del triángulo.
	`A`
	`AAA`
	`AAAAA`

70	Escribe un programa que cree un array de tamaño 100 con los primeros 100 números naturales. Luego muestra la suma total y la media. Implementa una función que calcule la suma de un array y otra que calcule la media de un array
71	Realiza una función que calcule el factorial de un número. En el programa, se solicitará al usuario el número sobre el que quiere calcular el factorial (el método solo realiza el factorial, no interacciona con el usuario).

>> EJERCICIOS DE RECURSIVIDAD

1	Sumar los números naturales hasta N (se lo damos nosotros) de forma recursiva.
2	Recorrer un array cualquiera de forma recursiva.
3	Calcular el valor de la posición fibonacci X usando recursividad.