



# Programación

02 Programación Estructurada y Modular

José Luis González Sánchez





# Contenidos

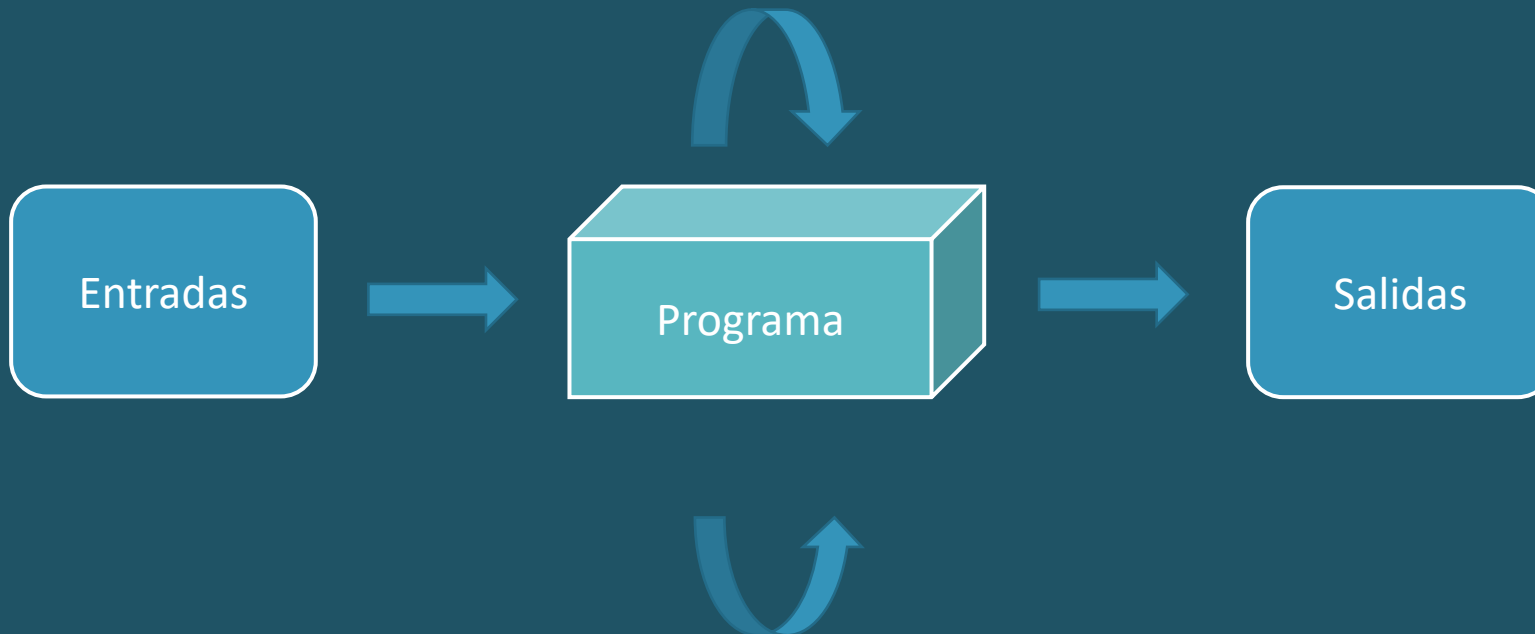
1. Tipos y Operaciones
2. Constantes y Variables
3. Algoritmos
4. Programación Estructurada
5. Programación Modular
6. Recursividad

# Tipos y Operaciones

Todo es más simple de lo que parece

# Tipos y Operaciones

- Antes de empezar a programar debemos tener en cuenta que:
  - Un programa procesa un conjunto de datos de entrada (o información de entrada) mediante una serie de operaciones/pasos para obtener unos datos o información de salida.
  - Es por ello que debemos aprender a cómo estructurar y definir nuestra información de entrada y salida, es decir nuestros datos y las operaciones que realizamos sobre ellos para obtener los resultados deseados



# Tipos y Operaciones

- **Los tipos de datos nos sirven para clasificar nuestra información de entrada y salida y trabajar con ellos.**
- **Tipos simples:** Se llama tipo de dato a una clase concreta de objetos o valores. Cada tipo de datos tiene asignadas unas determinadas operaciones:
  - Números enteros: Con signo o sin él (7, -9, 125...)
  - Números reales: Con decimal (7.9, 15.26...)
  - Caracteres o letras: Alfanumérico entre comillas simples ('z', 'a', '2'...)
  - Cadenas: secuencia de caracteres entre comillas dobles ("casa", "programación")
  - Lógicos: verdadero o falso (F/V)
- **Los tipos nos ayuda a clasificar la información y los datos y a saber cuánta memoria vamos a necesitar para procesar dicha información.**
- Además los tipos nos ayuda a una nueva clasificación de lenguajes: Tipados vs No Tipados.
- Así que nuestra primera pregunta a procesar una información o dato es **¿Cuál es su tipo?** Y con ello sabremos qué podemos hacer con ello.
- **Tenemos los datos compuestos:** arrays, listas, etc, que los veremos más adelante.

# Tipos y Operaciones

- Llamamos **lenguajes tipados (O fuertemente tipados)** a los lenguajes donde cada dato a usar se le debe asignar el tipo que le corresponde. Con ello nos aseguramos de utilizar la memoria y las operaciones que este tipo tiene asignada.
  - Ventajas: se conoce su naturaleza desde el momento de la compilación y con ello las operaciones a realizar. Nos aporta seguridad a la hora de desarrollar y un control extra en lo que hacemos. Están muy unido a lenguajes compilados.
  - Inconvenientes: muchas veces no conocemos el tipo o debemos cambiar de un tipo a otro. Debemos hacer el Casting de uno a otro. Esto puede complicar el desarrollo cuando comenzamos.
  - Ejemplos: JAVA, C++, TypeScript
- Llamamos **lenguajes NO tipados o débilmente tipados o con tipado dinámico** (no son exactamente lo mismo) a los lenguajes donde aunque existiendo tipos de datos, no los especificamos a la hora de declarar variables, si no es el propio lenguaje el que asigna y cambia el tipo según la información a usar.
  - Ventajas: no nos preocupamos de definir un tipo, simplemente trabajamos con la información. El propio lenguaje realiza las conversiones oportunas según se necesiten. Están unidos a los lenguajes interpretados
  - Inconvenientes: podemos perder el control del tipo y con ello tener errores que no detectemos. Debemos aprender las reglas de conversión del lenguaje.
  - Ejemplos: Python, PHP, Ruby, JavaScript

# Tipos y Operaciones

- **Los datos** que participan en una operación se llaman **operandos** y el símbolo de la operación se llama **operador**. Existen dos tipos de operaciones: **aritméticas y lógicas**.
  - **Aritméticas: solo se pueden realizar con números enteros y reales:**
    - Suma +
    - Resta -
    - Multiplicación \*
    - División entera div (devuelve el cociente entero, solo con enteros)
    - División / (devuelve el cociente con decimales)
    - Modulo (resto) % ó mod (devuelve el resto. solo con enteros)
    - Exponenciación ^

# Tipos y Operaciones

- **Relacionales: son operaciones comparativas.**
  - Menor que <
  - Mayor que >
  - Igual que o doble igual ==
  - Menor o igual que <=
  - Mayor o igual que >=
  - Distinto de !=
- **Lógicas: aplacan una lógica o condición para evaluar**
  - Conjunción Y: &&
  - Disyunción O: ||
  - Negación NO: !

AND, $p \wedge q$ , $p * q$ , $p \text{ y } q$ Conjunción lógica Y			OR, $p \vee q$ , $p + q$ , $p \text{ o } q$ Disyunción lógica O		
$p$	$q$	$p \text{ AND } q$	$p$	$q$	$p \text{ OR } q$
V	V	V	V	V	V
V	F	F	V	F	V
F	V	F	F	V	V
F	F	F	F	F	F

NOT, $\neg p$ , $\bar{p}$ , $\neg p$ , $\sim p$ Negación lógica	
$p$	NOT $p$
V	F
F	V



# Tipos y Operaciones

	Operaciones	Operandos	Resultados	Ejemplo
Aritméticas	+, -, *, /, ^	Números enteros (Z) Números reales (R)	$Z * Z = Z$ $Z * R = R$ $R * R = R$	$7.4 + 2 = 9.4$ $3 + 4 = 7$
Relacionales	>, >=, <, <=, !=, ==	Números enteros (Z) Números reales (R) Caracteres	Lógicos (V/F)	$3 < 7 = V$ $'a' > 'z' = F$
Lógicas	Y (and), O (or), No (not)	Lógicos (V/F)	Lógico (V/F)	$V \text{ ó } F = V$ $(2 > 7) \text{ Y } ('a' < 'z') \quad F$ $F \quad \quad \quad V$

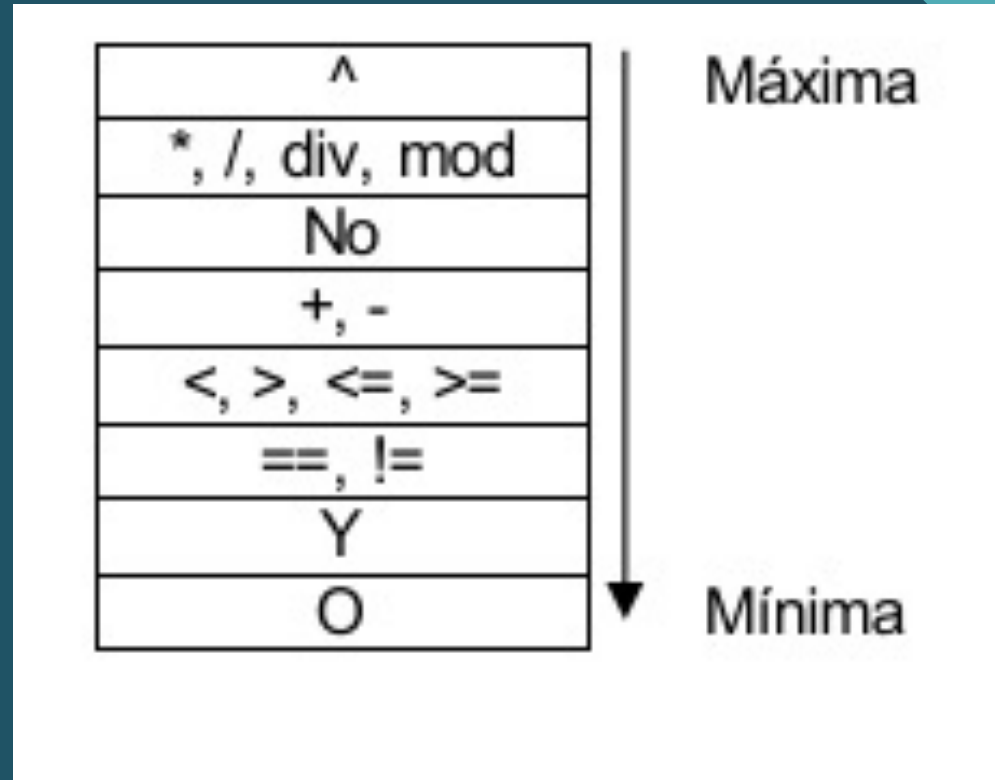
# Tipos y Operaciones

- Además de las operaciones que hemos visto, los lenguajes de programación disponen de una serie de operadores predefinidos que hacen operaciones más complejas. A estas se les llaman funciones de librería.

Función	Descripción	Tipo de dato	Tipo de resultado
abs(x)	Valor absoluto de x	Real o Entero	Real o Entero
sen(x)	Seno de x	Real o Entero	Real
cos(x)	Coseno de x	Real o Entero	Real
exp(x)	$e^x$	Real o Entero	Real
ln(x)	Logaritmo neperiano de x	Real o Entero	Real
log10(x)	Logaritmo decimal de x	Real o Entero	Real
redondeo(x)	Redondea el numero x al valor más próximo	Real	Entero
trunc(x)	Trunca el numero x, es decir, le elimina la parte decimal	Real	Entero
raíz(x)	raíz cuadrada de x	Real o Entero	Real
cuadrado(x)	$X^2$	Real o Entero	Real o Entero
aleatorio(x)	Genera un numero al azar entre x y 0	Entero	Entero

# Tipos y Operaciones

- Proridad de los operadores

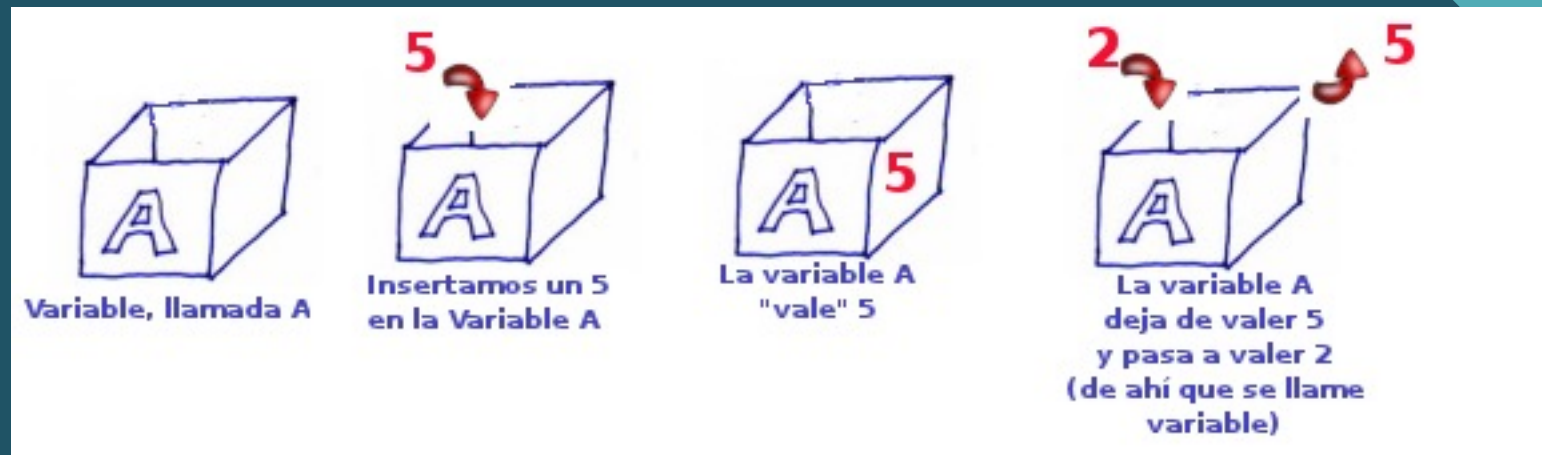
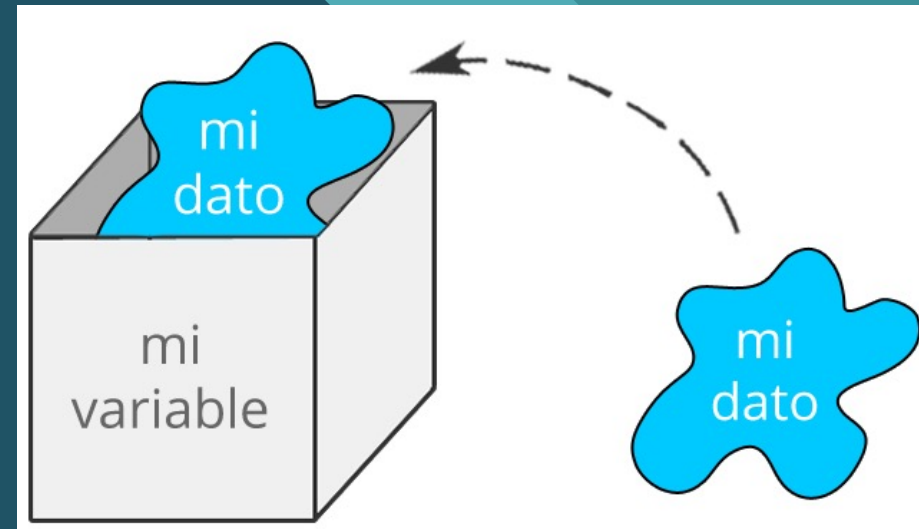


# Constantes y Variables

Almacenando nuestros datos

# Tipos y Operaciones

- Se define una **constante** como un dato del programa que no cambia mientras se está ejecutando, siempre es el mismo, por otro lado, el valor de una **variable**, si puede cambiar, y por lo tanto si puede cambiar durante la ejecución.
- Podemos imaginar que una variable es un contenedor de memoria para almacenar datos de un tipo determinado (ya sea indicado de forma explícita o implícita)
- **Características de las constantes y variables:** constan de un **nombre**, un **tipo de dato base** (lo que puede almacenar esa variable), y un **valor** (valor que tenga en ese instante la variable).



# Tipos y Operaciones

- **Reglas para poner nombre:**

- 1. No pueden empezar por numero
- 2. No pueden contener operadores (+ - \*...)
- 3. No pueden tener espacios
- 4. No pueden empezar por caracteres especiales (@ ñ ` '...)
- 5. Si es constante va todo en mayúscula y si son dos o más palabras se separan con guión bajo: NUMERO\_PI
- 6. Si es variable, se escribe en minúscula y si son dos o más palabras, todo junto, la primerapalabra en minúscula y la primera letra del resto de palabras en mayúscula: edadUsuario
- 7. El nombre de la variable debe representar el dato que almacena (código limpio)
- 8. Nuestra nomenclatura para declararlas es: Tipo variableNombre: Valor

# Tipos y Operaciones

- **Declaración y asignación de variables:**
  - **Declaración.** Definir cómo es una variable y si queremos otorgarle un valor inicial obligatorios y lógicos:
    - Caracter letraAlfabeto: 'Z'
    - Cadena ciudadResidencia: "Leganés"
    - Lógico esRepetidor: V
    - Real NUMERO\_PI: 3,1415
    - Real notaMedia: 7,8
    - Entero edadAlumno: 20
  - **Asignación:** dar valor a una variable a lo largo de un programa despues de definirla
    - notaMedia = 9,25
    - edadAlumno = 18
    - provinciaResidencia = "Madrid"
- **Expresiones:** una expresión es una combinación de constantes, variables y funciones.
  - Ej:  $(6+4)/3$   $(6+4) \text{ div } 3$

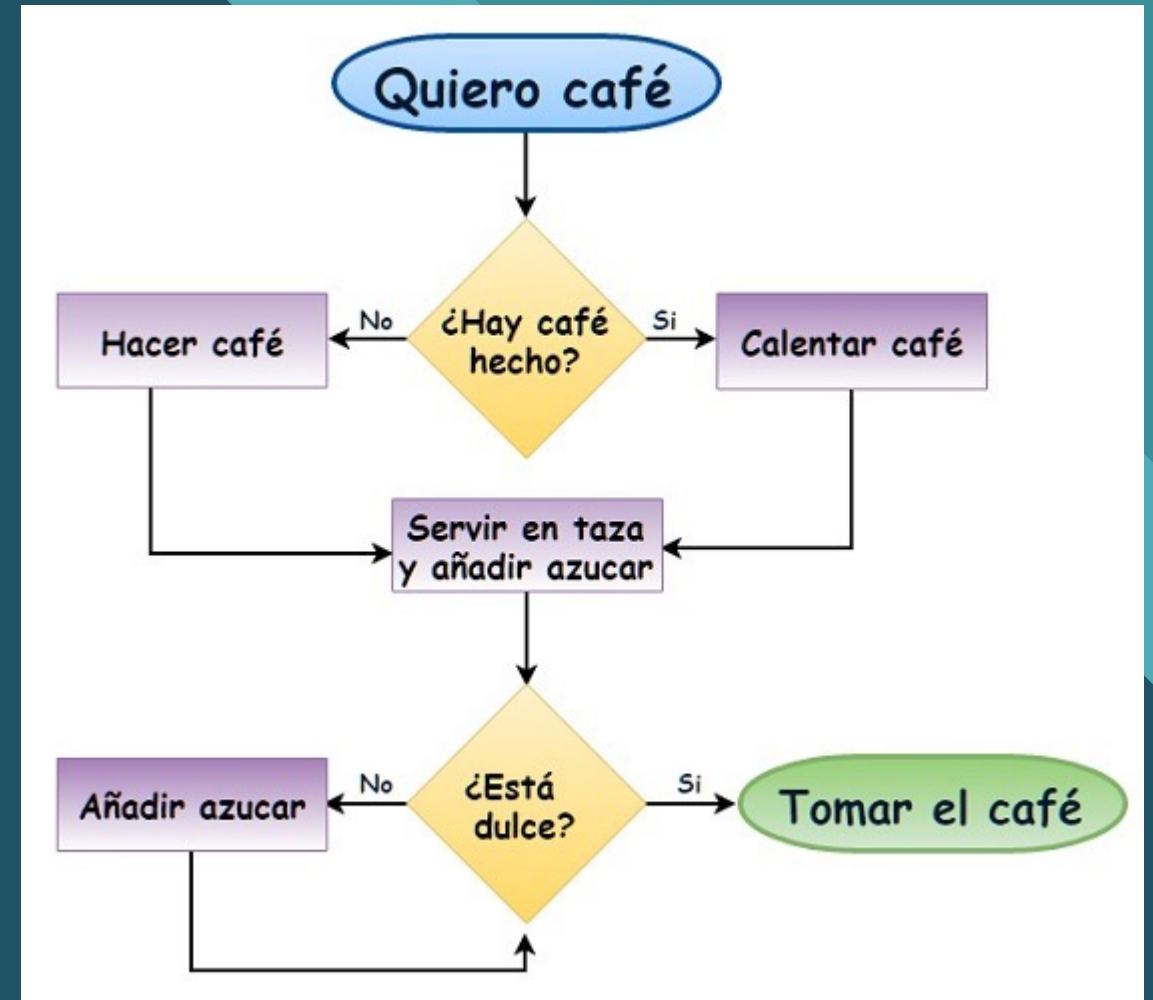
# Algoritmo

¿Cómo resolver nuestros problemas y representarlos?

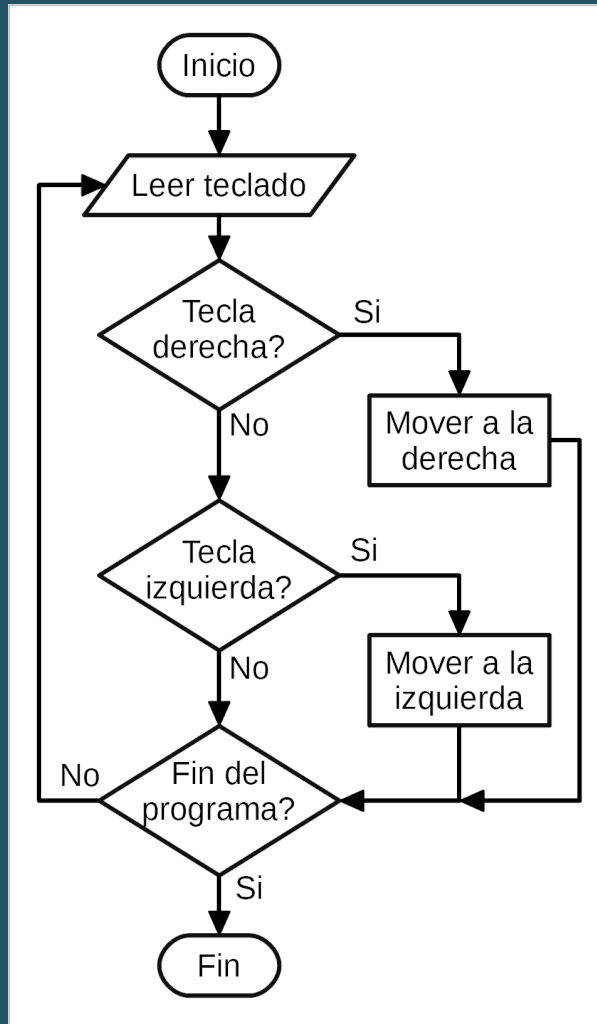





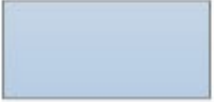

# Algoritmos

- Un **algoritmo** es una secuencia ordenada de pasos que conducen a la solución de un problema. Tienen tres características:
  - Son precisos en el orden de realización de los pasos.
  - Están bien definidos de forma que usando un algoritmo varias veces con los mismos datos, dé la misma solución.
  - Son finitos, deben acabarse en algún momento.
- Los algoritmos deben representarse de forma independiente del lenguaje de programación que luego usaremos.
- Usaremos ordinogramas o diagramas de flujo para representarlos y pseudocódigo



# Algoritmos



Símbolo	Nombre	Función
	Inicio / Final	Representa el inicio y el final de un proceso
	Línea de Flujo	Indica el orden de la ejecución de las operaciones. La flecha indica la siguiente instrucción.
	Entrada / Salida	Representa la lectura de datos en la entrada y la impresión de datos en la salida
	Proceso	Representa cualquier tipo de operación
	Decisión	Nos permite analizar una situación, con base en los valores verdadero y falso

# Algoritmos

- El **pseudocódigo** es un lenguaje de descripción de algoritmos que usa un lenguaje común pero con ciertas normas, se usarán palabras reservadas que serán unas palabras que indican una función concreta que se haría en el ordenador.

Algoritmo Ejemplo

Declaración

Real A, B  
Real S

Inicio

leer (A)

leer (B)

$S = A * B$

escribir (S)

Fin

**Comenzamos con Algoritmo o Programa y el nombre del programa usando CamelCase**

**Declaración:** palabra reservada que indica las variables que usaré y el tipo de dato.

**Inicio / Fin:** Palabras que indican el comienzo y final de las instrucciones que va a realizar el algoritmo, estas instrucciones se realizarán paso a paso desde Inicio hasta Fin.

**Instrucciones:** En un primer momento tenemos dos tipos:

a) **Asignación:** son del tipo  $A=B$ ,  $C=3+D...$

b) **Entrada / Salida:**

**leer (X):** el algoritmo introducirá en x un valor que nos viene desde el exterior (teclado, flash...).

**escribir (X):** con esta instrucción, el algoritmo saca el valor de X al exterior (monitor, impresora...), para poner una frase o palabra a mostrar: escribir ("palabra")

# Algoritmos

- **Comentarios.** Se usan para explicar determinados fragmentos de un algoritmo o aclarar el conjunto de acciones o pasos realizados.
- Existen dos tipos de comentarios:

- **Comentarios de varias líneas.** Los comentarios de varias líneas se incluyen entre los símbolos `/*` y `*/`

```
/*
```

```
Este es un ejemplo de  
un comentario de varias  
líneas.
```

```
*/
```

- **Comentarios de una sola línea.** Para comentar una sola línea se utiliza la doble diagonal `//`. El comentario se inicia cuando se encuentra la doble diagonal y continua hasta el final de la línea.

```
// Este es un comentario de una sola línea
```

```
//Este es otro comentario
```

# Programación Estructurada

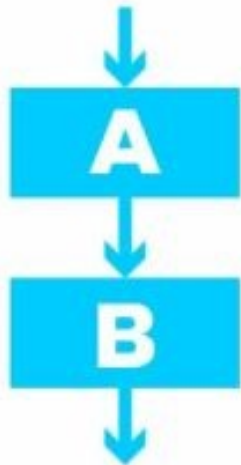
Nuestro primer paradigma

# Programación Estructurada

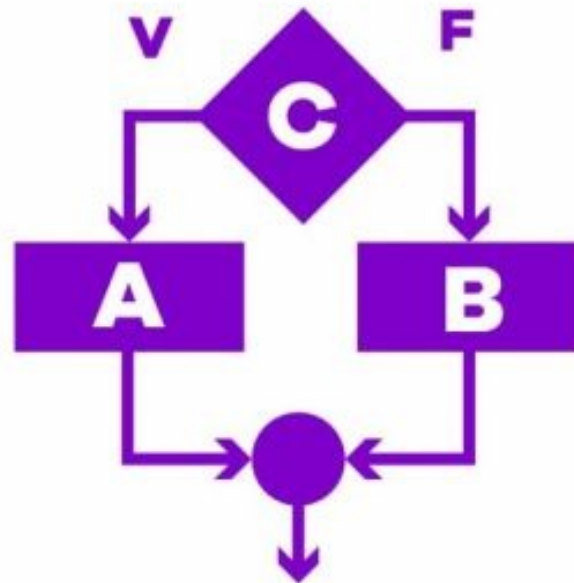
- En 1966, Bohm y Jacopini, demostraron que se puede escribir cualquier programa utilizando solo 3 tipos de estructura: secuencial, selectiva o condicional y repetitiva.
- Los programas así definidos cumplen 3 características:
  - Existe un único punto de inicio y un único final.
  - Existe al menos un camino que parta de Inicio y llegue a Fin pasando por todo el programa.
  - No deben existir bucles infinitos.
- Todo programa puede escribirse utilizando únicamente las tres instrucciones de control siguientes:
  - Secuencia.
  - Instrucción condicional: siples o múltiples.
  - Iteración (bucle de instrucciones) con condición inicial: indefinidos, definidos.
  - Solamente con estas tres estructuras se pueden escribir todos los programas y aplicaciones posibles. Si bien los lenguajes de programación tienen un mayor repertorio de estructuras de control, estas pueden ser construidas mediante las tres básicas citadas.

# Programación Estructurada

Secuencia



Selección o condicional



Iteración (ciclo o bucle)



# Programación Estructurada. Estructura Secuencial

- Es aquella a la que una acción le sigue otra.

Inicio

Acción 1

Acción 2

.....

.....

Acción n

Fin

Programa Secuencial

Variables

Cadena nombre, apellidos

Entero edad

Inicio

leer (nombre)

leer (apellidos)

leer (edad)

edad = edad + 2

escribir (nombre)

escribir (apellidos)

escribir (edad)

Fin



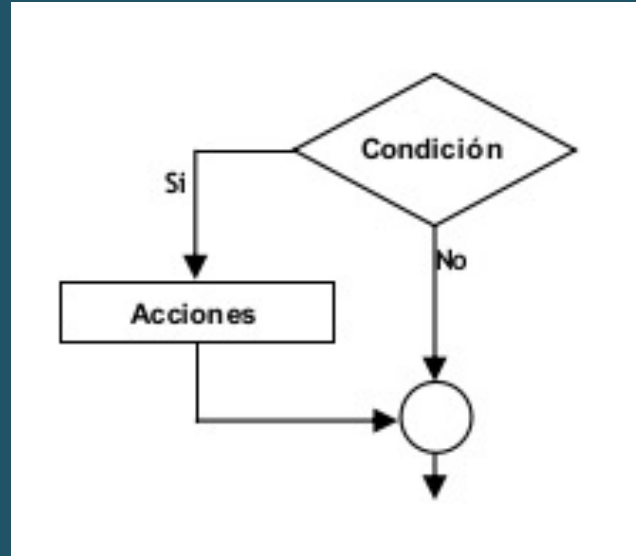
# Programación Estructurada. Estructura Condicional

- Aparece cuando tengo que elegir entre unas acciones u otras según una condición. Existen 3 tipos de estructuras selectivas: simples, dobles y múltiples.
- **Condicional simple:** Si la condición es verdadera, se harán las instrucciones que pongamos entre Inicio y Fin.

Si (condición) Entonces

Acciones 1

Fin\_si



Programa CondicionalSimple

Variables

Entero num

Inicio

leer (num)

Si (num>0) Entonces

escribir ("Es positivo")

Fin\_si

Fin

# Programación Estructurada. Estructura Condicional Doble

- **Condicional Doble:** La condición doble hará dos bloques de instrucción, uno si se cumple u otro sino se cumple.

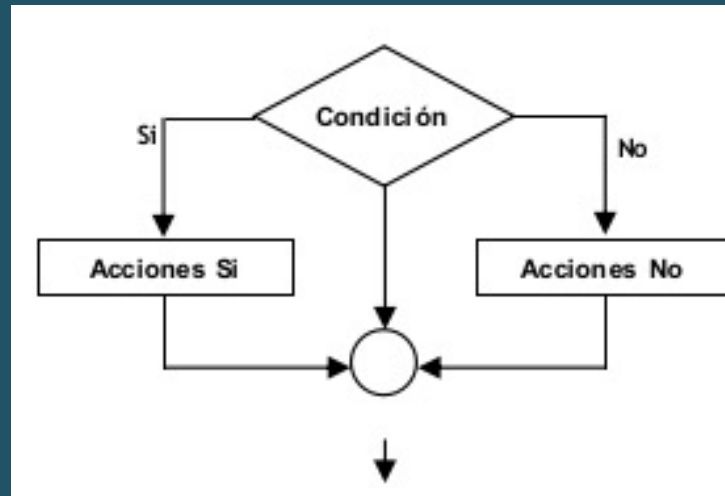
Si (condición) Entonces

Acciones si verdadero

Sino

Acciones si falso

Fin\_si



Programa CondicionalDoble

Variables

Entero año

Inicio

leer (año)

Si ((año mod 4)==0) Entonces  
escribir ("El año es bisiesto")

Sino

escribir ("El año no es bisiesto")

Fin\_si

Fin

# Programación Estructurada. Estructura Condicional Anidada

- La estructura secuencial/condicional doble puede tener varios si anidados..

```
Si (num>0) Entonces
    escribir ("Positivo")
Sino
    Si (N<0) Entonces
        escribir ("Negativo")
    Sino
        escribir ("0")
    Fin_Si
Fin_Si
```

# Programación Estructurada. Estructura Condicional Múltiple

- Existen casos en los que se debe elegir no solo entre verdadero y falso, sino entre mas opciones, para esos casos se utiliza la estructura según, cuya regla de escritura general es:

Según (variable)

Valor 1: acciones si 1

...

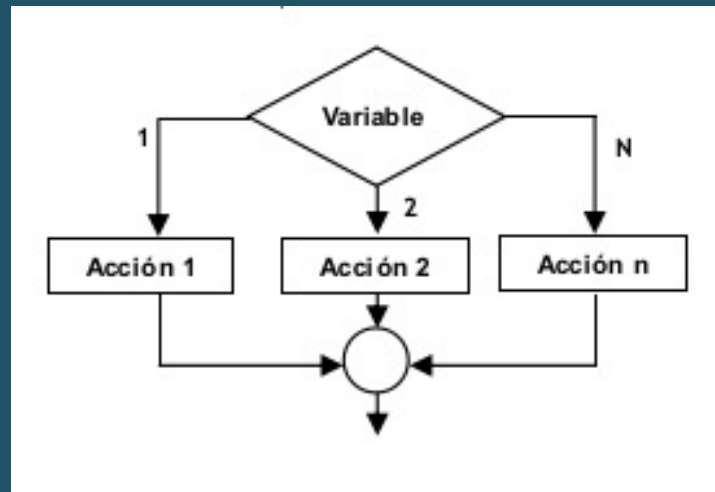
Valor 2: acciones si 2

...

Valor no: acciones si no

...

Fin\_según



Programa DiasMes

Variables

Entero mes, año

INICIO

leer (año)

leer (mes)

Según (mes)

Valor 1:

escribir ("Enero 31 días")

Valor 2:

Si ((año mod 4)==0) Entoences

escribir ("Febrero 29 días")

Sino

escribir ("Febrero 28 días")

Fin\_Si

Valor 3:

escribir ("Marzo 31 días")

Valor 12:

escribir ("Diciembre 31 días")

Sino:

escribir ("Error")

Fin\_según

FIN

# Programación Estructurada. Estructura Repetitivas

- Estas estructuras consisten básicamente en **repetir varias veces un conjunto de instrucciones**. Los bucles deben ser finitos, es decir, la repetición cesa en algún momento. Los componentes de los bucles son:
  - Cuerpo del bucle: conjunto de instrucciones que se repiten
  - Condición de salida: será la condición que pondremos para que pare la repetición.
- Tenemos los siguientes:
  - Bucles indefinidos: Mientras y Repetir
  - Bucles definidos: Para

# Programación Estructurada. Bucle Mientras

- En esta estructura, el cuerpo del bucle se repite mientras que la condición que pongamos sea verdadera.

Mientras (condición)

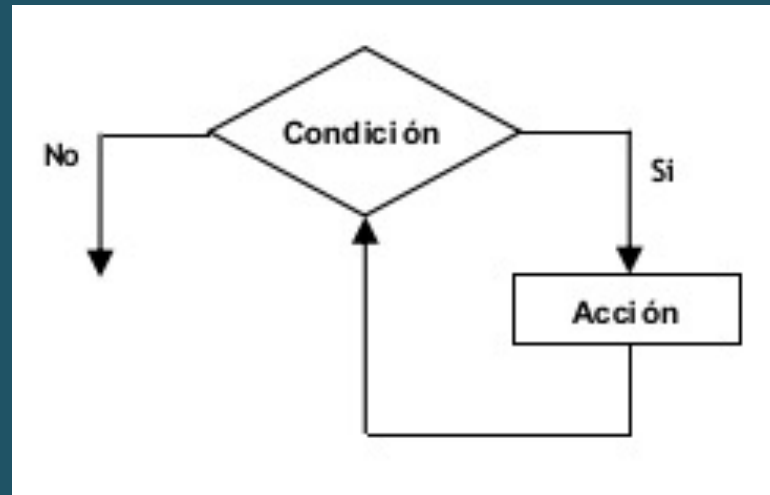
Acción 1

Acción 2

...

Acción n

Fin\_mientras



Programa SaludoRepetitivo

Variables

Entero contador

INICIO

contador=1

Mientras (cont<=10)

escribir ("Hola")

contador=contador+1

Fin\_mientras

FIN

# Programación Estructurada. Bucle Repetir

- La estructura repetir, realizará las acciones que englobe mientras que se cumpla una determinada condición. La diferencia con el mientras es que con repetir, las acciones se hacen al menos una vez.

Repetir

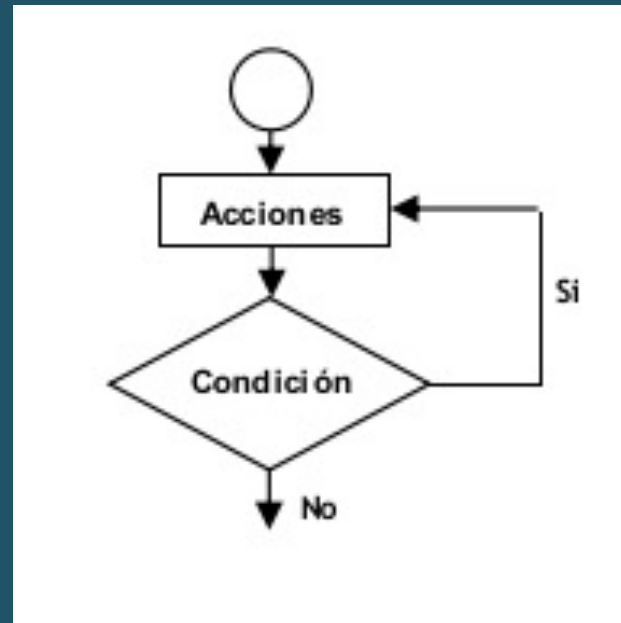
Acción 1

Acción 2

...

Acción n

Mientras (condicion)



Programa SaludoRepetitivo

Variables

Entero contador

INICIO

contador=1

Repetir

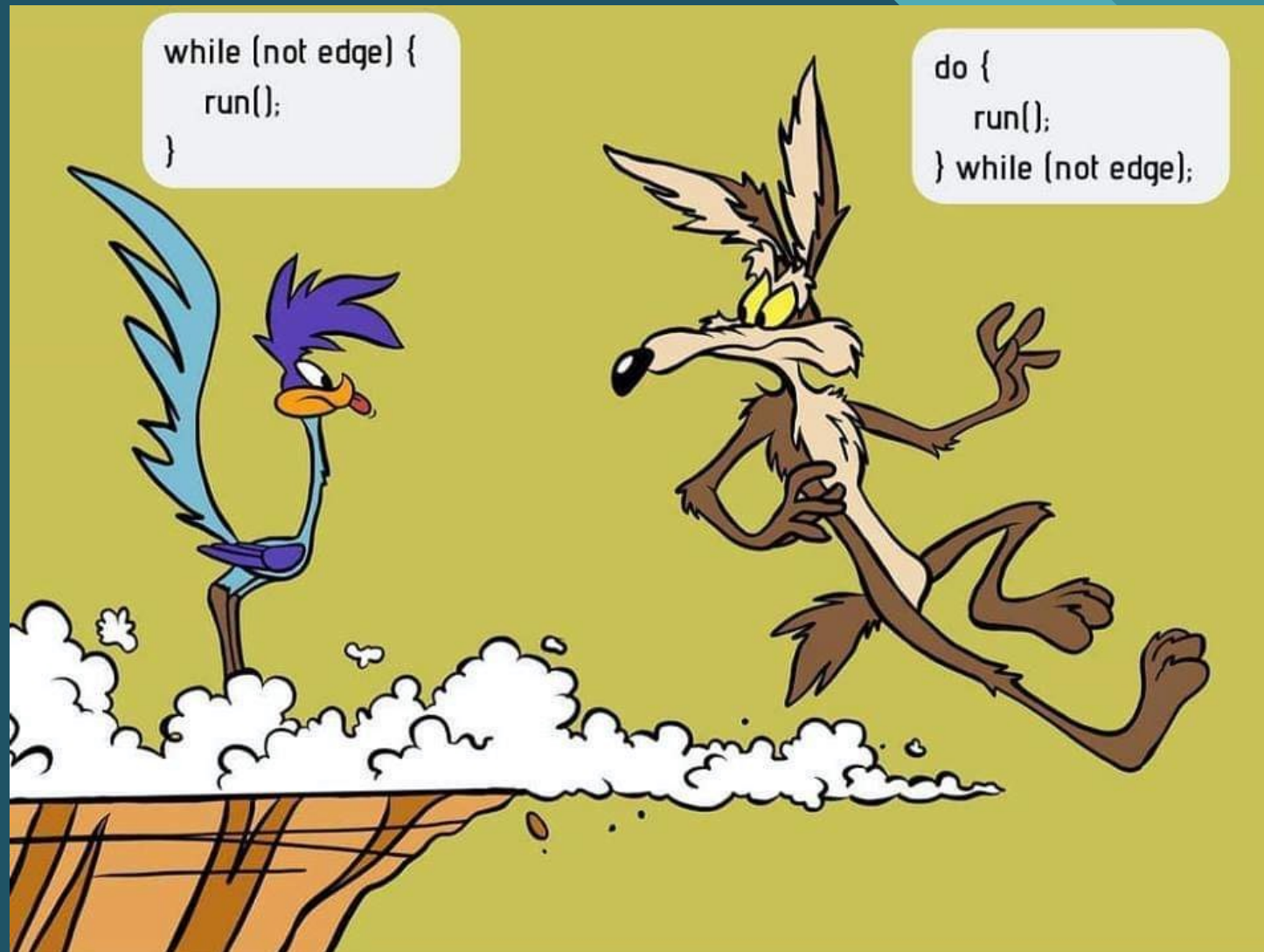
escribir ("Hola")

contador=contador+1

mientras (cont<=10)

FIN

# Programación Estructurada. Bucle Indefinidos





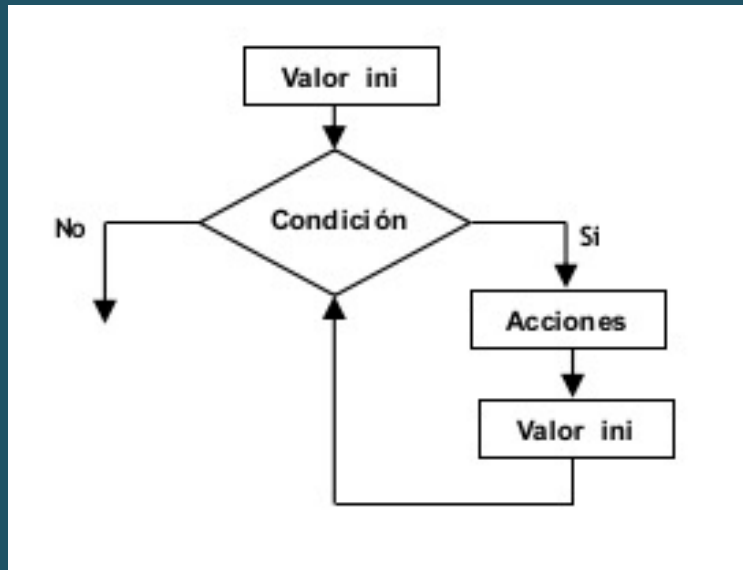
# Programación Estructurada. Bucle Para

- Esta estructura repetitiva se usa solo para los bucles definidos, es decir, cuando se sabe exactamente cuántas veces se va a repetir algo.
- Esta estructura repetirá las acciones de forma secuencial desde el valor inicial de la variable hasta el valor final incrementado la variable automáticamente según le indiquemos con Inc. Si no se pone, es que Inc = 1

Para (variable = valor\_inicial mientras valor\_final Inc 1)

Acción1  
Acción 2  
...  
Acción n

Fin\_para



Programa SaludoRepetitivo  
Variables

Entero contador

INICIO

Para (contador=1 mientras contador<=10 Inc 1)  
    escribir ("Hola")

Fin\_para

FIN

# Programación Estructurada. Banderas o Flags

- Son variables que solo pueden tomar dos valores (normalmente) y que a esos valores le asignamos un significado que nosotros, como programadores, debemos controlar.

Programa NumerosPrimos

Variables

Enteros contador, numero

Lógico esPrimo= V

Inicio

Leer (numero)

Para (cont=2 mientras contador < numero)

Si ((num mod cont) == 0) Entonces

esPrimo = F

Fin\_si

Fin\_para

Si (esPrimo == V) Entonces

escribir("Si")

Sino

escribir("No")

Fin\_si

Fin

# Programación Estructurada. Aleatorios

- La función `aleatorio()` devuelve un número al azar comprendido entre el 0 y el valor que se le indique. La estructura de esta función es:
  - $\text{Numero} = (\text{aleatorio} () * \text{valorMáximo})$
- Para obtener un número al azar evitando el 0, la estructura será la siguiente
  - $\text{Numero} = (\text{aleatorio} () * \text{valorMáximo}) + 1$

# Programación Estructurada. Comentarios

- Los comentarios nos sirven para aclarar o explicar fragmentos de código.
- Tenemos dos tipos
  - Línea a línea: Comienzan por //
  - En bloque: Comienzan por /\* y terminan con \*/

Programa NumerosPrimos  
Variables

Enteros contador, numero  
Lógico esPrimo= V

Inicio

```
// Aquí leo un numero
Leer (numero)
/* Esto es un comentario de bloque
   así que puedo escribir distintas líneas
*/
Para (cont=2 mientras contador < numero)
    Si ((num mod cont) == 0) Entonces
        esPrimo = F
    Fin_si
Fin_para

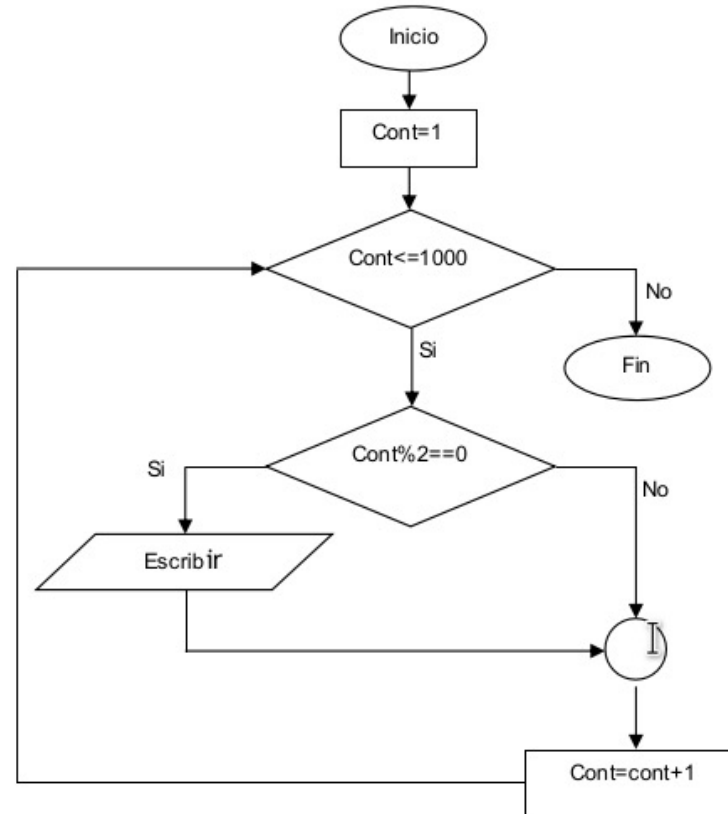
Si (esPrimo == V) Entonces
    escribir("Si") // Imprimo el resultado si es verdadero
Sino
    escribir("No")
Fin_si
```

Fin

# Programación Estructurada. Banderas o Flags

## Ejemplo:

Escribir los números pares entre 1 y 1000



# Programación Modular

Creando nuestras propias estructuras y reutilizando código

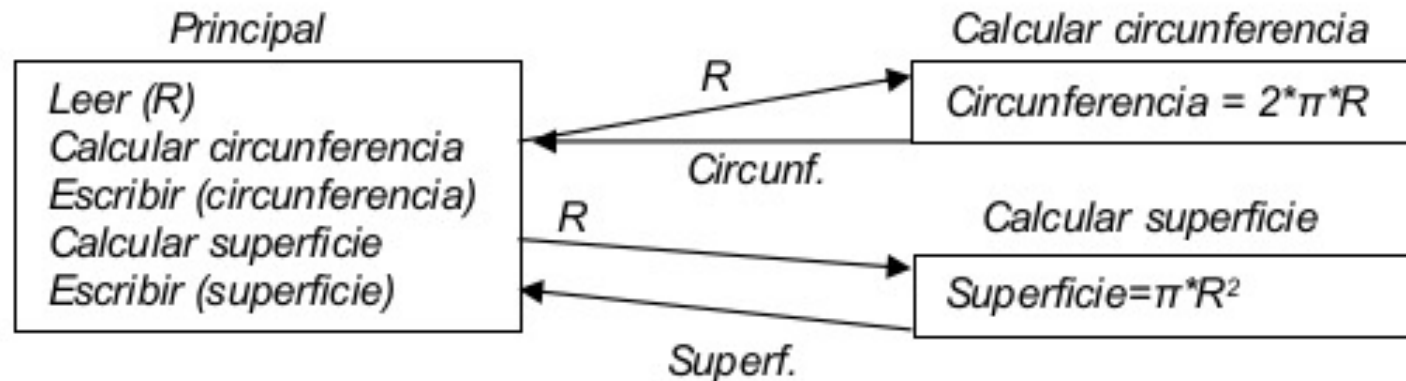
# Programación Modular

- La **programación modular** se puede definir como una programación que afronta el problema descomponiéndolo en subproblemas más simples, cada uno de los cuales se resuelve mediante un subalgoritmo que se llama modulo y que será más o menos independiente del resto del programa.
- Las ventajas que ofrece la programación modular son:
  - Facilita la resolución del problema.
  - Aumenta la claridad y legibilidad de todo el programa.
  - Permite que varios programadores trabajen en el mismo proyecto.
  - Reduce el tiempo de desarrollo ya que se pueden reutilizar esos módulos en varios programas.
  - Aumenta la fiabilidad porque es más sencillo diseñar y depurar módulos y el mantenimiento es más fácil.
- La descomposición modular se basa en la técnica “Divide y Vencerás” (DAC o Divide And Conquer), esta técnica tiene dos pasos:
  - Identificación de los subproblemas y construcción de los módulos que lo resuelven.
  - Combinación de los módulos para resolver el problema original.

# Programación Modular.

- En general, el problema principal se resuelve en un algoritmo que se llama algoritmo o **modulo principal**, mientras que los subproblemas se resuelven en los **módulos secundarios** o subalgoritmos.
- Cuando un modulo necesita que otro módulo haga una tarea, se produce lo que se llama una invocación o llamada.
- Lo habitual es que el algoritmo que invoca o que llama le pase unos valores al subalgoritmo, esos valores se llaman parámetros, y que el subalgoritmo devuelva unos resultados.

*Ejemplo:*



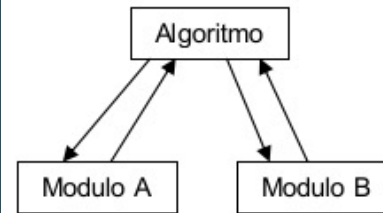


# Programación Modular.

- **Nivel de descomposición modular.** La cantidad o el número de módulos que será necesario emplear para resolver el problema principal viene dado por la cantidad de operaciones que tenga que hacer el programa.
- No existe una regla general, pero cuando un modulo tiene demasiadas líneas u observamos que hace mas de una cosa es porque se puede descomponer en dos módulos; y al contrario, si un modulo es demasiado sencillo, puede que pertenezca a un modulo general.
- Lo más importante es que el modulo haga una cosa y este bien diseñado para hacerlo.

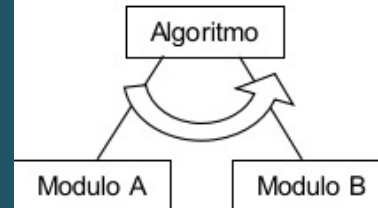
**A) Diagrama de caja:** En este diagrama se representan las llamadas a los subalgoritmos (módulos)

## Secuencial



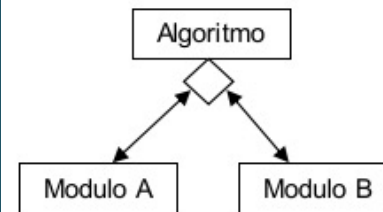
En este caso, se haría de forma secuencial, primero el modulo A y después el modulo B

## Repetitiva



En este caso, primero llama al modulo A y después al B, y vuelve al principio y llama al A y luego al B y así hasta que salga.

## Selectiva



En este caso, el algoritmo llama al modulo A ó al modulo B.

# Programación Modular. Funciones

- Las funciones resuelven un problema sencillo y devuelven siempre un único resultado al algoritmo que las llamó. Las funciones pueden tener argumentos o parámetros aunque no es obligatorio.
- **Las funciones, como devuelven siempre un valor, es conveniente que ese valor se almacene en una variable.**
- numAleatorio=aleatorio (): Aleatorio es una función del sistema que no tiene argumentos y que devuelve un resultado que se guarda en un variable
- resultado=raíz (9): Esta función si tiene argumento (9).

# Programación Modular. Funciones

- **Declaración de Funciones.** Definimos cómo será nuestra función en base al tipo que devuelve, los parámetros que usa y las acciones a realizar. Si tiene varias palabras seguimos la norma de las variables.

Tipo\_resultado\_devuelto nombreFuncion(argumentos con su tipo)

Variables y constantes

---

Inicio

---

---

---

Devolver (el valor que sea)

Fin

Entero suma(Entero x, y)

Variables

Entero resultado

Inicio

resultado=x+y

Devolver (resultado)

Fin

# Programación Modular. Funciones

- La llamada o invocación consiste en una mención al nombre de la función seguida entre paréntesis de los valores que se desean asignar a los argumentos. Deben aparecer tantos valores como argumentos haya y deben coincidir en tipo. Estos valores se asignarán a los argumentos y podrán usarse dentro de la función como variables.

Programa EjemploFuncion

Variables

Entero a, b

Inicio

leer (a)

leer (b)

escribir (suma(a,b))

Fin

Entero suma (Entero x, y)

Variables

Entero resultado

Inicio

resultado=x+y

Devolver (resultado)

Fin

Programa EjemploFuncion

Variables

Entero a, b, c

Inicio

leer (a)

leer (b)

c=suma(a,b)

escribir (v)

Fin

Entero suma(Entero x, y)

Variables

Entero resultado

Inicio

resultado=x+y

Devolver (resultado)

Fin

# Programación Modular. Procedimientos

- Los procedimientos son tipos de subalgoritmos (o funciones) que no devuelven ningún valor. Se suelen utilizar para ejercicios en los que no hay que calcular nada o al contrario, para subrutinas/subalgoritmos que calculan un valor o más.

# Programación Modular. Procedimientos

- Declaración. Similar a la función, pero sin tipo de devolución.

nombreProcedimiento(lista de argumentos)

Variables

...

Inicio

...

...

Fin

# Programación Modular. Procedimientos

- La llamada a un procedimiento, como no devuelve nada, se realiza simplemente con el nombre del procedimiento y los argumentos que se le pasarán.
- Ejemplo: escribir todos los pares entre 1 y N

Programa EjemploNumerosPares

Variables

Entero numero

Inicio

leer(numero)

obtenerPares(numero)

Fin

obtenerPares (Entero n)

Variables

Entero contador

Inicio

Para (contador=1 Mientras que contador<=n)

Si ((contador mod 2)==0) Entonces

escribir(contador)

Fin\_si

Fin\_Para

Fin

# Programación Modular. Paso de Parámetros

- Con “Paso de parámetros” nos referimos al modo en el que le pasamos a los subalgoritmos las variables argumento.
- Tendríamos dos modos de hacerlo:
  - Paso por valor
  - Paso por referencia



# Programación Modular. Paso de Parámetros

- **Paso por Valor**
- Es el modelo que hemos estado utilizando hasta ahora y significa que las variables del algoritmo principal que le pasamos al subalgoritmo, no pueden ser modificadas en su valor por dicho subalgoritmo.

Programa ParametrosPorValor

Variables

Entero num

Inicio

num = 7

pasoValor(num)

escribir(num) // saldría 7

Fin

pasoValor (Entero n)

Variables

Inicio

n=n\*2

escribir(n) // saldría 14

Fin

# Programación Modular. Paso de Parámetros

- **Paso por Referencia**

- En estas llamadas de este tipo se produce una ligadura o una asociación entre la variable que usamos en la llamada y el parámetro del subalgoritmo, es decir, las modificaciones que sufra el parámetro, se reflejan en la variable que usamos en la llamada. En pseudocódigo reflejaremos esta situación con la palabra “Ref”. El paso por referencia se usa sobre todo cuando el subalgoritmo tiene que devolver más de un valor.

Programa ParametrosPorReferencia

Variables

Entero num

Inicio

num = 7

pasoReferencia((Ref) num)

escribir(num) // saldría 14

Fin

pasoReferencia (Entero (Ref) n)

Variables

Inicio

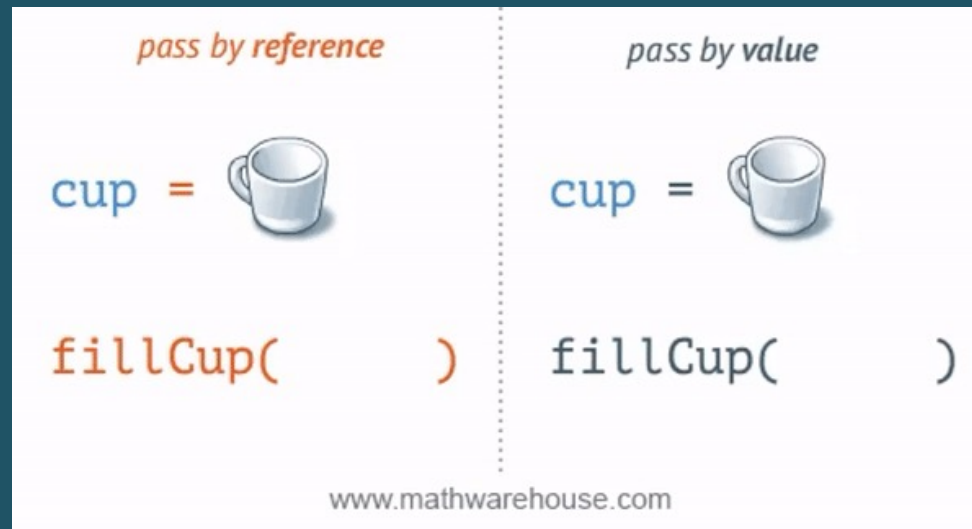
n=n\*2

escribir (n) // saldría 14

Fin

# Programación Modular. Paso de Parámetros

- Diferencias entre paso por valor y por referencia
- Como hemos visto, la principal utilidad del paso por referencia, es que a un subalgoritmo (procedimiento o función), puede “devolver” más de un valor. Por lo tanto las diferencias son:
  - El paso por valor se puede decir que sigue un camino unidireccional, es decir, los datos van del algoritmo principal al subalgoritmo, pero no al contrario.
  - En los pasos por referencia, los datos viajan de forma bidireccional.



# Programación Modular. Ámbito de Variables

- Se llama ámbito de una variable a la parte del programa o del algoritmo donde se puede utilizar.
- **Variables locales.** Se denomina variable local, aquella variable que existe únicamente dentro de un algoritmo o de un subalgoritmo.
- Las variables a, b y suma son variables locales del algoritmo principal y solo existen dentro de él, por lo tanto son inaccesibles desde el subalgoritmo función.
- Las variables x, y, resultado son variables locales del subalgoritmo función e inaccesibles desde el algoritmo principal.

Programa VariableLocal  
Variables

Enteros a, b, suma

Inicio

leer(a)

leer(b)

suma = multiplicar(a, b)

escribir (suma)

Fin

Entero multiplicar(Entero x, y)  
Variables

entero resultado

Inicio

resultad=x\*y

Devolver (resultado)

Fin

# Programación Modular. Ámbito de Variables

- **Variables globales.** Son aquellas que se pueden usar en todos los sitios del programa y que se suelen definir en el algoritmo principal. Se definen como Global

```
Programa VariableGlobal
Variables
    Entero a, b
    Entero Global suma

Inicio

    leer(a)
    leer(b)
    multiplica(a, b)
    escribir (suma)

Fin
```

```
multplica (Entero x, y)
Inicio

    suma=x+y

Fin
```

# Programación Modular. Ámbito de Variables

- **Efectos laterales**
- El uso de **variables globales** está **prácticamente prohibido**, solo se utilizarán para casos muy puntuales que no se puedan resolver de otra manera.
- Se denominan efectos laterales a la comunicación de datos entre un algoritmo y un subalgoritmo al margen de los canales habituales (parámetros por referencia y devolución de funciones)
- No se deben usar variables globales por las siguientes razones:
  - Complica el código haciéndolo muy difícil de interpretar y de seguir, así como de modificar.
  - Va en contra de la programación estructurada, obteniendo como resultado, el código “spaghetti”, es decir, es muy difícil de seguir y mantener.

# Programación Modular. Paquete de Librerías o Módulos

- **Paquete de Módulos o Librerías**

- Llamamos paquete de módulos o librerías de módulos a un conjunto de funciones o procedimientos que se agrupan y que realizan operaciones relacionadas, por ejemplo funciones matemáticas o de entrada y salida.
- Importando estas librerías o módulos tenemos disponibles en nuestro programa dicha funcionalidad.
- Pueden venir hechas por el sistema o los podemos realizar nosotros.

# Programación Modular. Reutilización

- **Reutilización de módulos**
- Una de las principales ventajas de los módulos que vimos al principio era la reutilización, es decir, una vez definido un subalgoritmo, lo podemos usar para resolver diversos problemas. Para que un modulo sea reutilizable, debe cumplir dos características:
  - Que tenga bien definida la tarea que realiza, que no se sea ambigua.
  - Que sea independiente, es decir, que se use siempre igual, sin importar ni donde, ni cuantas veces lo usemos.



# Recursividad

Llamándonos a nosotros mismos

# Recursividad

- **Recursividad:** Se llama recursividad a un proceso mediante el que una función se llama a sí misma de forma repetida, hasta que se satisface alguna determinada condición. El proceso se utiliza para computaciones repetidas en las que cada acción se determina mediante un resultado anterior. Se pueden escribir de esta forma muchos problemas iterativos.
- Se deben satisfacer dos condiciones para que se pueda resolver un problema recursivamente:
  - Primera: El problema se debe escribir en forma recursiva. Es decir, una función debe llamarse así misma.
  - Segunda: La sentencia del problema debe incluir una condición de fin, para salir de la recursividad
- Cuando ejecutamos un programa recursivo, las llamadas recursivas no se ejecutan inmediatamente. Lo que se hace es colocarlas en una pila hasta que la condición de término se encuentra. Entonces se ejecutan las llamadas a la función en orden inverso a como se generaron, como si se fueran sacando de la pila.

# Recursividad. Ejemplo del Factorial

- ¿Qué es la función factorial?
- La función factorial se representa con un signo de exclamación “!” detrás de un número. Esta exclamación quiere decir que hay que multiplicar todos los números enteros positivos que hay entre ese número y el 1.
- $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$

- **Versión iterativa**

```
Entero factorial (entero num) {  
  INICIO
```

```
    Variables
```

```
        Entero t, res
```

```
        res=1;
```

```
    para (t=1 mientras t<= num)
```

```
        res = res * t;
```

```
    Fin_para
```

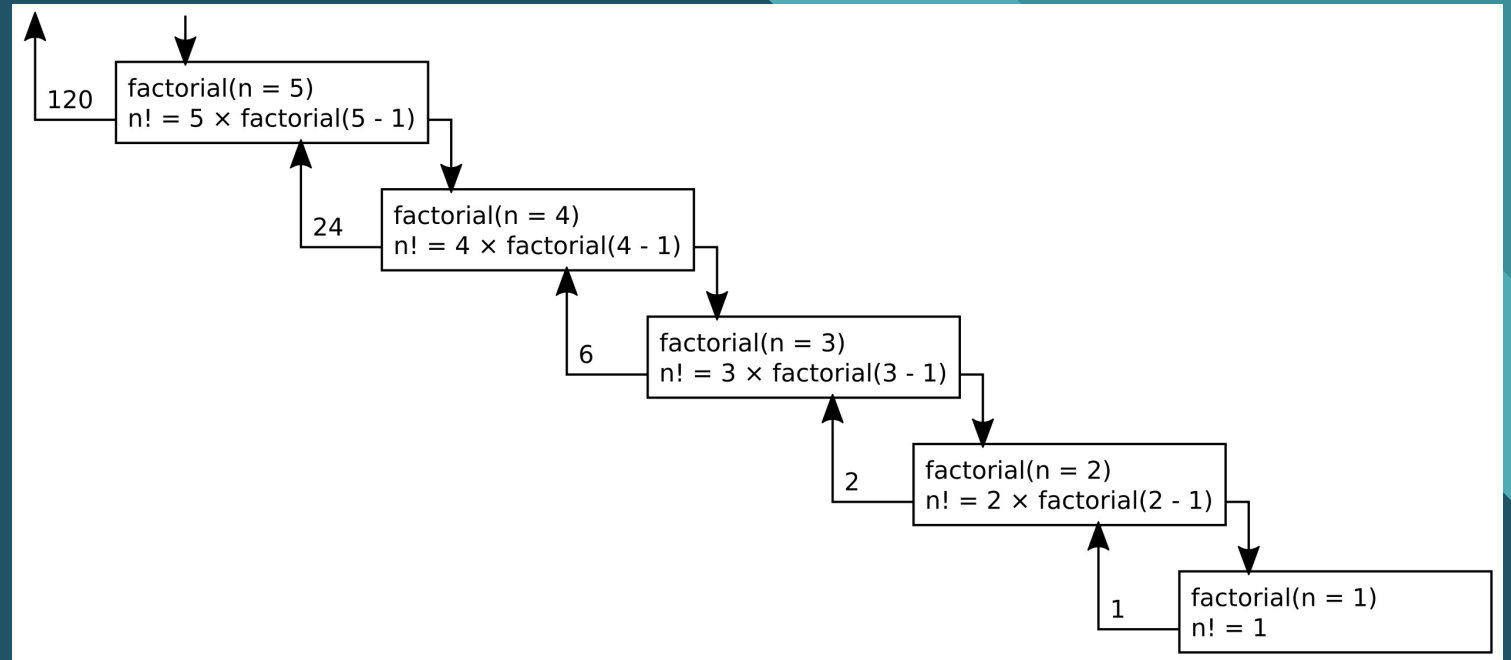
```
    Devolver (res)
```

```
FIN
```

# Recursividad. Ejemplo del Factorial

- **Versión recursiva**

```
Entero factorial (entero num) {  
  INICIO  
    SI (num <= 1)  
      Devuelve(1)  
    sino  
      devuelve(num * factorial (num-1))  
  FIN
```



# Recursividad. Ejemplo del Factorial

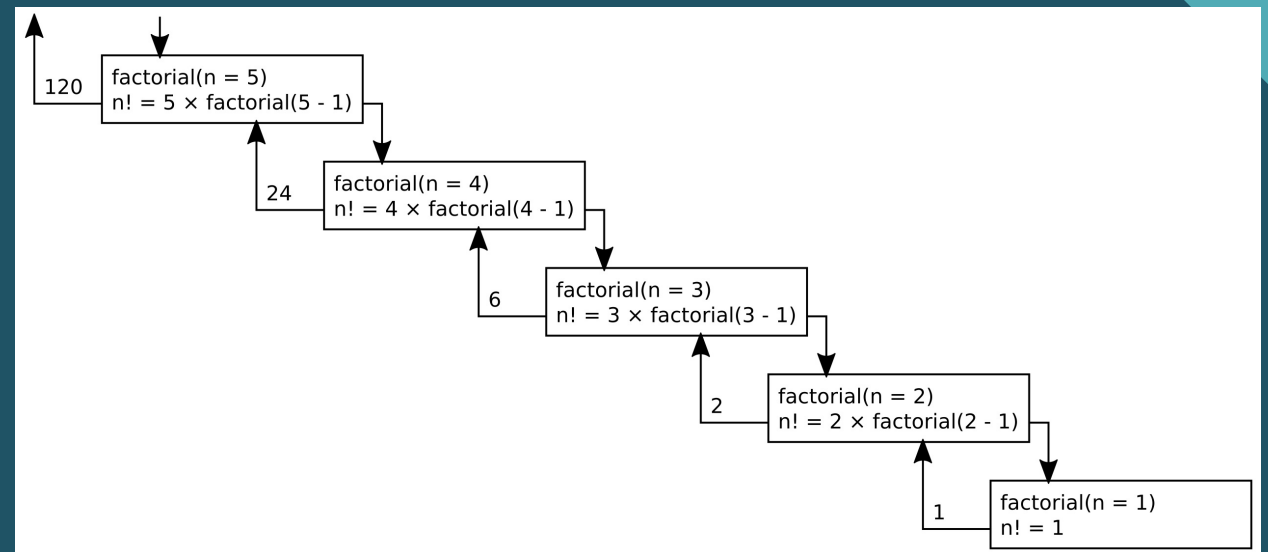
- Cuando ejecutamos un programa recursivo, las llamadas recursivas no se ejecutan inmediatamente. Lo que se hace es colocarlas en una pila hasta que la condición de término se encuentra. Entonces se ejecutan las llamadas a la función en orden inverso a como se generaron, como si se fueran sacando de la pila, por tanto el orden sería algo así:

```
1º      n! = n * (n-1)!
2º      (n-1)! = (n-1) * (n-2)!
3º      (n-2)! = (n-2) * (n-3)!
.....
.....
Último  2! = 2 * 1!
         1!      1
```

# Recursividad. Ejemplo del Factorial

- Los valores reales se devolverán en orden inverso, es decir:

1º         $1! = 1$   
2º         $2! = 2 * 1! = 2 * 1 = 2$   
3º         $3! = 3 * 2! = 3 * 2 = 6$   
.....  
.....  
Último    $n! = n * (n-1)! = \dots$





“

"Los programas deben ser escritos para que los lean las personas, y sólo incidentalmente, para que lo ejecuten las máquinas"

- Abelson and Sussman



”

# Recursos

- Twitter: <https://twitter.com/joseluisgonsan>
- GitHub: <https://github.com/joseluisgs>
- Web: <https://joseluisgs.github.io>
- Discord: <https://discord.gg/uv7GcytM>
- Aula Virtual: <https://aulavirtual33.educa.madrid.org/ies.luisvives.leganes/course/view.php?id=245>





# Gracias

José Luis González Sánchez

