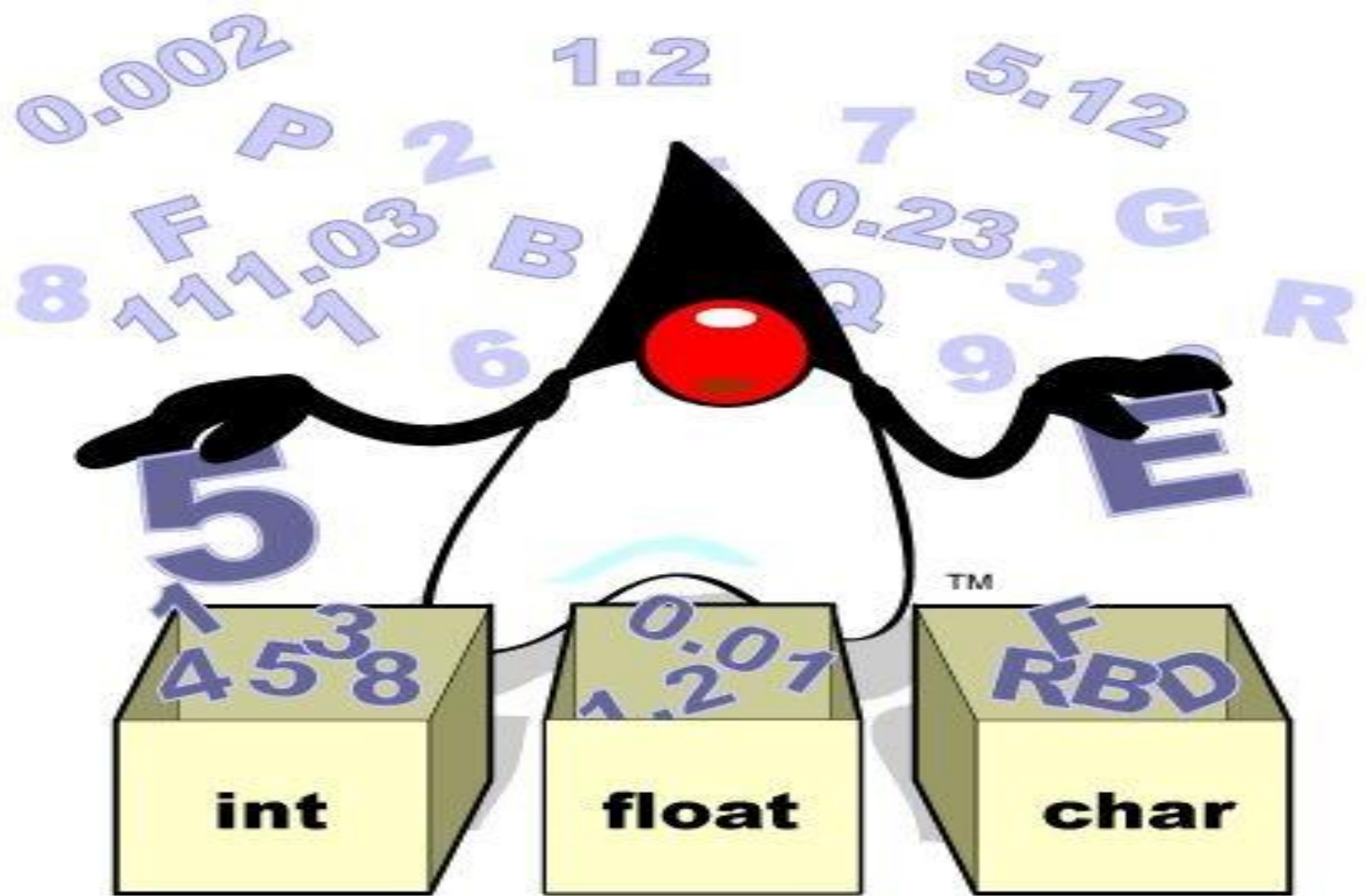


UT01. INTRODUCCIÓN A LA PROGRAMACIÓN.

Programación – 1ºDAW

Índice

- ❑ Datos, algoritmos y programas.
- ❑ Paradigmas de programación.
- ❑ Lenguajes de programación.
- ❑ Errores y calidad de los programas.
- ❑ Fases en la creación de un programa.
- ❑ Programación estructurada



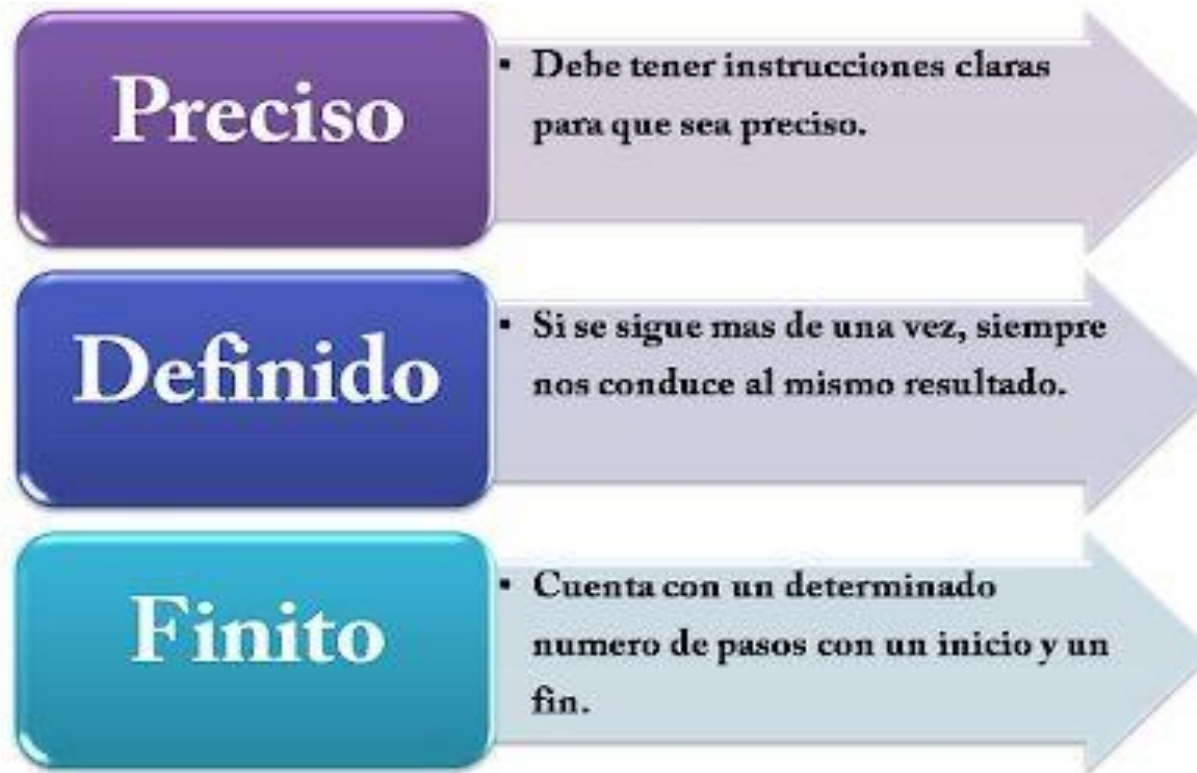
1. Algoritmos, datos y programas.

Algoritmos y programas

- Un algoritmo es un conjunto ordenado y finito de pasos o instrucciones bien definidas (sin ambigüedad) que llevan a la resolución de un problema específico.
- Los algoritmos son una parte fundamental de la programación, ya que se utilizan para diseñar programas de computadora.
- Los algoritmos deben ser **claros, lógicos** y capaces de producir un **resultado predecible** cuando se siguen correctamente.

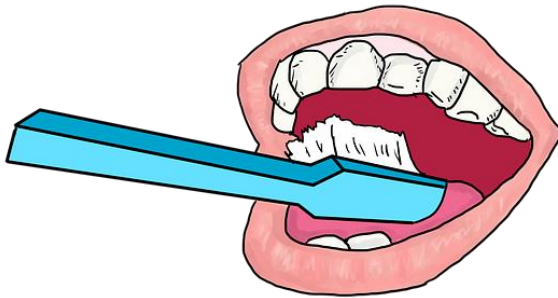
Algoritmos y programas

- **Características fundamentales que debe cumplir un algoritmo:**



Algoritmos y programas

- Los algoritmos pueden variar en complejidad, desde algoritmos simples que resuelven tareas básicas hasta algoritmos altamente sofisticados que abordan problemas complejos.
- Un ejemplo de algoritmo sencillo sería una **receta de cocina**, abrir una puerta, lavarse las manos ...
- **Ejemplo de algoritmo:** lavarnos los dientes



1. Tomar la crema dental
2. Destapar la crema dental
3. Tomar el cepillo de dientes
4. Aplicar crema dental al cepillo
5. Tapar la crema dental
6. Abrir la llave del grifo
7. Remojar el cepillo con la crema dental
8. Cerrar la llave del lavamanos
9. Frotar los dientes con el cepillo
10. Abrir la llave del lavamanos
11. Enjuagarse la boca
12. Enjuagar el cepillo
13. Cerrar la llave del lavamanos
14. Secarse la cara y las manos con una toalla

Algoritmos y programas

□ Algoritmos cotidianos:

A diferencia de los seres humanos que realizan actividades sin detenerse a pensar en los pasos que deben seguir, los computadores son muy ordenados y necesitan que, quien los programa, les diga cada uno de los pasos que deben realizar y el orden lógico de ejecución.

□ Tarea:

Enumera en orden lógico los pasos siguientes (para pescar):

- ___ El pez se traga el anzuelo.
- ___ Enrollar el sedal.
- ___ Tirar el sedal al agua.
- ___ Llevar el pescado a casa.
- ___ Quitar el Anzuelo de la boca del pescado.
- ___ Poner carnada al anzuelo.
- ___ Sacar el pescado del agua.

Algoritmos y programas

- **Tarea:** Utilizando un lenguaje cotidiano describe los pasos necesarios para realizar las siguientes tareas:

1. Envolver un regalo
2. Freír un huevo
3. Cruzar la calle

Ahora uno un poquitín más complicado:

4. Construye un avión de papel y después escribe los pasos necesarios para que cualquiera pueda construir tu mismo modelo.

Algoritmos y programas

- **Diseño de algoritmos:**

- Podremos obtener distintas soluciones (algoritmos distintos), igualmente válidas, para resolver un mismo problema.
- Es necesario tener el conocimiento de las **técnicas de programación**.

Algoritmos y programas

- Un **lenguaje de programación** es un conjunto de reglas y símbolos que se utilizan para escribir programas de ordenador.
- Se podría decir que un lenguaje de programación es como un idioma artificial diseñado para que sea fácilmente **entendible** por un **humano** e **interpretable** por una **máquina**.
- Los **algoritmos son independientes** de los **lenguajes de programación** y de las computadoras donde se ejecutan.
- Un mismo algoritmo puede ser expresado en diferentes lenguajes de programación y puede ejecutarse en diferentes dispositivos.

Algoritmos y programas

- Los **lenguajes de programación** proporcionan un **medio para expresar las acciones** que deseamos que el ordenador realice. Estas acciones pueden incluir cálculos matemáticos, manipulación de datos, toma de decisiones y más.
- Cada lenguaje de programación tiene su propia **sintaxis** (reglas específicas para escribir código) y **semántica** (significado de las instrucciones escritas).

Algoritmos y programas.

- Un **programa** es una secuencia de **instrucciones** escritas en un lenguaje de programación que le dicen al ordenador cómo realizar una tarea.
- Estas instrucciones manipulan **datos** para obtener unos **resultados** que serán la solución del problema que resuelve el programa.
- Un **programa** es un algoritmo que ha sido expresado en un determinado lenguaje de programación para poder ser ejecutado por un ordenador.

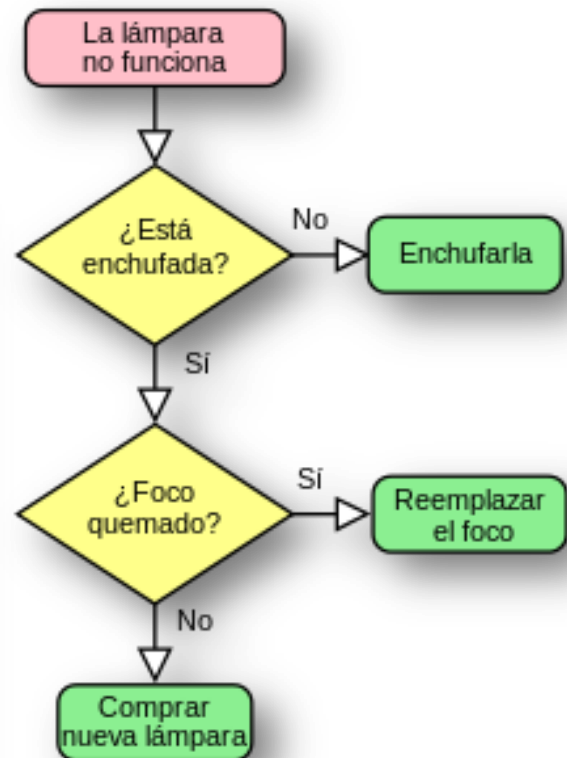
Algoritmos y programas

- Podemos representar un algoritmo, principalmente:
 - Gráficamente:** diagramas de flujo
 - Pseudocódigo:** lenguaje natural

Proceso Suma

```
Definir A,B,C como Reales;  
  
Escribir "Ingrese el primer numero:";  
Leer A;  
  
Escribir "Ingrese el segundo numero:";  
Leer B;  
  
C <- A+B;  
  
Escribir "El resultado es: ",C;
```

FinProceso



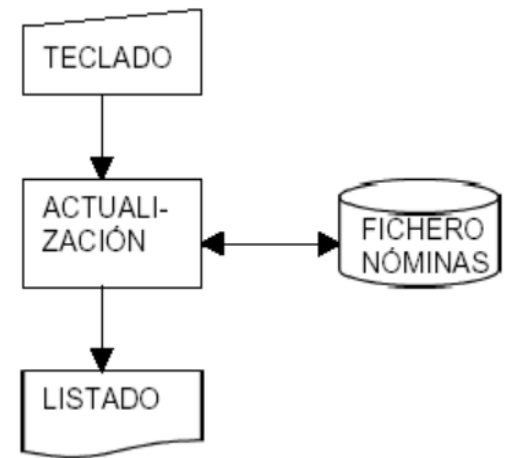
Algoritmos y programas

- **Diagramas de Flujo:** son una representación gráfica de un algoritmo.
 - ▣ Conjunto de símbolos normalizados, conectados mediante líneas de flujo que muestran la secuencia de pasos de un programa, las acciones que realiza o el origen y destino de los datos.
 - ▣ **Organigramas** (diagramas de flujo del sistema): visión general del programa, E/S del programa, dispositivos utilizados.
 - ▣ **Ordinogramas:** visión interna del programa. Es el que se utiliza en el diseño de algoritmos.

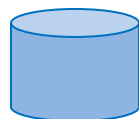
Algoritmos y programas

□ Organigramas – Normas: deben mostrar

- Datos o dispositivos de entrada (arriba)
- Soportes de entrada/salida (A Izda/dcha)
- Datos o dispositivos de salida (abajo)
- Nombre del proceso, tarea, programa (centro)
- Flujos de datos (flechas)



□ Elementos:



disco



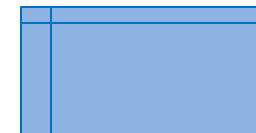
Teclado



Impresora



Pantalla



Almacenamiento interno



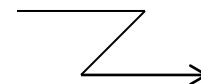
Proceso



Disp. Conexión



Línea de conexión
Salida/Entrada



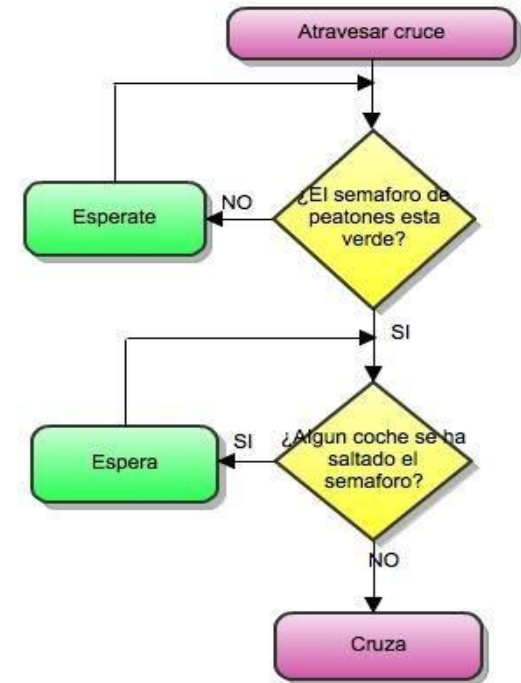
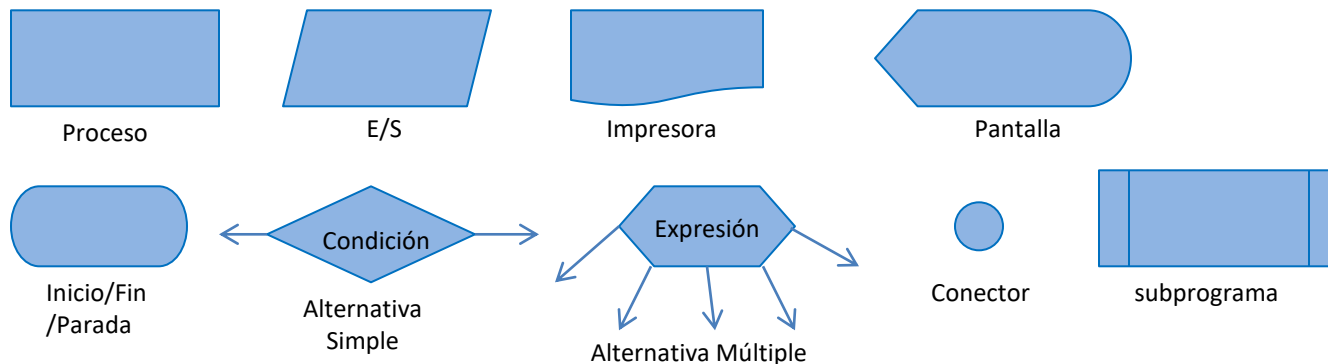
Salida/Entrada por línea de
comunicación

Algoritmos y programas

□ Ordinogramas – Normas:

- Un inicio (arriba) y un final (abajo).
- Secuencia de operaciones de arriba a abajo.
- Elementos conectados por líneas rectas de flujo de datos sin cruces.
- Se debe guardar simetría.
- Inicio-Entrada datos-Proceso-Resultados-Fin

□ Elementos:



Algoritmos y programas

□ Pseudocódigo

- ▣ Lenguaje intermedio entre el lenguaje natural y lenguaje de programación sujeto a ciertas reglas.
- ▣ Independiente del lenguaje de programación y de la máquina.
- ▣ Permite el diseño descendente (Top Down).
- ▣ No puede ser ejecutado directamente por un ordenador, por lo que tampoco es considerado como un lenguaje de programación propiamente dicho. Ahora sí (Pseint).

▣ Estructura general:

Programa: Nombre del programa

Entorno:

declaración de variables y tipos

Algoritmo

Secuencia de instrucciones

FinAlgoritmo

Subprograma: Nombre subprograma

Entorno

Algoritmo

Secuencia de instrucciones

FinAlgoritmo

- ❑ **En un algoritmo tenemos datos e instrucciones.**
- ❑ **Los datos son todo aquello que puede ser almacenado de forma independiente. Son símbolos que representan valores, objetos que deben ser tratados.**
- ❑ **Ejemplo: En una calculadora → los datos son los números y las operaciones con las que operar, así como los resultados obtenidos.**

Datos en un algoritmo. Constantes y variables.

- ❑ Un **dato constante** (**constante**) es un dato cuyo valor no cambia a lo largo del algoritmo.
- ❑ Un **dato variable** (**variable**) es un dato cuyo valor puede cambiar a lo largo del algoritmo.
- ❑ A las variables y constantes se les asigna un **identificador alfanumérico** (un nombre).
- ❑ Distinguiremos entre el identificador de la variable o constante y su valor.
- ❑ Reglas para los identificadores (pueden variar con el lenguaje de programación):
 - ❑ Deben empezar por una letra
 - ❑ No deben contener símbolos especiales (comillas, caracteres acentuados, almohadilla, ...) salvo el subrayado “_”
 - ❑ No deben coincidir con alguna palabra reservada del lenguaje de programación.

Datos en un algoritmo. Constantes y variables.

□ Declaración y asignación

- ▣ Para utilizar una variable, debemos declararla al principio del algoritmo e indicar el tipo de la misma → **declarar la variable**

```
X es entero  
Y es real  
letra es carácter
```

- ▣ Las variables tienen que ser de un **tipo de datos determinado**.
- ▣ Debemos indicar el tipo de dato de la variable al declararla.
- ▣ Para dar un valor a una variable, se emplea una sentencia de **asignación** empleando el símbolo '=' ó '←'

```
X = 5  
Y = 7.445  
LETRA = 'J'
```

- ▣ A partir de la asignación de un valor, pueden hacerse operaciones con las variables exactamente igual que se harían con datos.

Datos. Tipos de datos de las variables

- **Tipo de datos:** indica el tipo de elementos o valores que puede almacenar la variable y las operaciones que se pueden realizar con ellas.
- Cada tipo de datos tiene asociado un **conjunto de valores** y un **conjunto de operaciones** para manipularlos.
- **Tipos de datos simples**
 - Números enteros
 - Números reales
 - Caracteres
 - Lógicos (booleanos)
- **Tipos de datos complejos:** también llamados estructuras de datos, como los arrays, se verán más adelante.

Datos. Tipos de datos simples

□ Números Enteros

- Tipo más sencillo $\rightarrow \dots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \dots$
- El ordenador tiene una **memoria finita**, por lo que la cantidad de valores enteros que puede manejar también es finita y depende del número de bits que emplee para ello (va a depender del hardware)
- Los enteros comprenden **números con signo y sin signo** y dependiendo del caso, la forma de almacenarlos en binario es distinta: complemento a uno, complemento a dos, ...
- Si utilizamos 8 bits para codificar números enteros, el rango de valores permitido irá de 0 a 255 (sin signo) o de -128 a +127 (con signo).
- Con 16 bits, el rango será de 0 a 65535 sin signo y de -32768 a 32767 con signo.
- Si se utilizan 32, 64, ... bits se pueden manejar números enteros mayores.

Datos. Tipos de datos simples

□ Números reales

- Este tipo permite representar números con decimales.
- La cantidad de decimales de un número real puede ser infinita (número π), pero al ser finita la memoria del ordenador, hay que establecer un número máximo de dígitos decimales.
- La representación interna de los números reales se denomina coma flotante, que es una generalización de la notación científica convencional:
 - Consiste en definir cada número con una mantisa y un exponente
 - La notación científica es muy útil para representar números muy grandes economizando esfuerzos
 - Ejemplo: $1294390000000000000000 \rightarrow 1,29439 \times 10^{20}$
 - Aunque un ordenador lo representa siempre con un 0 a la izquierda de la coma decimal $\rightarrow 0,129439 \times 10^{21}$

Datos. Tipos de datos simples

□ Números reales

- $0,129439 \times 10^{21}$

La mantisa es el número situado en la posición decimal (129439)

El exponente es 21

- La notación científica es igualmente útil para números decimales muy pequeños

- $0,000000000000000000000000259 \rightarrow 2,59 \times 10^{-23}$

Recordemos que el ordenador lo representará así $\rightarrow 0,259 \times 10^{-22}$

Internamente, el ordenador reserva varios bits para la mantisa y otros para el exponente

- Como en el caso de los número enteros, la magnitud de los números que puede manejar el ordenador estará relacionada con el número de bits reservados para su almacenamiento.

- **Overflow o desbordamiento** \rightarrow se produce cuando al realizar operaciones con números (enteros o reales), el resultado excede el rango máximo permitido.

Datos. Tipos de datos simples

□ Caracteres y cadenas

- El tipo **carácter** sirve para representar **datos alfanuméricos**.
- El conjunto de elementos que puede representar un ordenador está estandarizado según el **código ASCII**, que es una combinación de bits asociada a un carácter alfanumérico concreto.
 - Las combinaciones de 8 bits dan lugar a un total de 256 valores → cantidad de caracteres diferentes que se pueden utilizar.
- **Datos de tipo carácter válidos:**
 - ▣ Las letras minúsculas: 'a', 'b', 'c', ... , 'z'
 - ▣ Las letras mayúsculas: 'A', 'B', 'C', ... , 'Z'
 - ▣ Los dígitos: '1', '2', '3', ... , '9', '0'
 - ▣ Caracteres especiales: '\$', '%', '¿', ...
- Los variables de tipo carácter sólo pueden contener **UN carácter**.
- Si queremos representar una serie de **varios caracteres**, por ejemplo para escribir una palabra, usaremos el **tipo cadena de caracteres** → tipo de datos complejo que veremos más adelante.

Datos. Tipos de datos simples

ASCII Hex Símbolo

0	0	NUL
1	1	SOH
2	2	STX
3	3	ETX
4	4	EOT
5	5	ENQ
6	6	ACK
7	7	BEL
8	8	BS
9	9	TAB
10	A	LF
11	B	VT
12	C	FF
13	D	CR
14	E	SO
15	F	SI

ASCII Hex Símbolo

16	10	DLE
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN
25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
30	1E	RS
31	1F	US

ASCII Hex Símbolo

32	20	(space)
33	21	!
34	22	"
35	23	#
36	24	\$
37	25	%
38	26	&
39	27	'
40	28	(
41	29)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/

ASCII Hex Símbolo

48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

ASCII Hex Símbolo

64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O

ASCII Hex Símbolo

80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D]
94	5E	^
95	5F	_

ASCII Hex Símbolo

96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

ASCII Hex Símbolo

112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	
125	7D	}
126	7E	~
127	7F	

Datos. Tipos de datos simples

□ Datos lógicos

- El tipo de dato **lógico** o **booleano**, es un dato que sólo puede tomar un valor entre dos posibles → **verdadero** o **falso**.
- Se utiliza para representar alternativas del tipo **sí/no**.
- Los datos lógicos contienen **información binaria** (0 o 1), por lo que son bastante importantes. El valor **true** se representa con el **número 1** y el valor **false** con el **número 0**.
- Además, son el resultado de todas las **operaciones lógicas y relacionales** que utilizaremos en nuestros programas.

Datos. Operaciones con los datos

- Cada **tipo de dato** tiene asociado un **conjunto de operaciones**.
- Los datos de una operación se llaman **operandos** y el símbolo de la operación se denomina **operador**.
 - En la operación $5 + 3$, los operandos son 5 y 3 y “+” es el operador.
- Las **operaciones básicas** con datos se pueden clasificar en dos grandes grupos:
 - **Operaciones aritméticas**
 - **Operaciones lógicas**

Datos. Operaciones con los datos

□ Operaciones aritméticas

- Son análogas a las operaciones matemáticas convencionales, aunque cambian los símbolos.
- Sólo se emplean con datos de tipo entero o real (aunque puede haber alguna excepción).
- No todos los operadores existen en todos los lenguajes de programación (Fortran no tiene división entera, C no tiene exponenciación, en Pascal el operador % se escribe “mod”)....

□ Suma	+
□ Resta	-
□ Producto	*
□ Potencia	^
□ División entera	\ o div
□ División real	/
□ Módulo	mod o %

Datos. Operaciones con los datos

□ Operaciones aritméticas

- El tipo del resultado de cada operación dependerá del tipo de los operandos.
- Las operaciones “div” (división entera) y “%” (resto) sólo se pueden hacer con números enteros, no con reales.
- La operación “/” sólo se puede hacer con reales, no con enteros.
- El operador “-” también se usa para preceder a los números negativos, como en el álgebra convencional.

Operandos	Operador	Operación	Resultado
35 y 9 (enteros)	+	35 + 9	44 (entero)
35 y 9 (enteros)	-	35 - 9	26 (entero)
35 y 9 (enteros)	*	35 * 9	315 (entero)
35 y 9 (enteros)	div	35 div 9	3 (entero)
35 y 9 (enteros)	%	35 % 9	8 (entero)
35 y 9 (enteros)	^	35 ^ 9	overflow
8,5 y 6,75 (reales)	+	8,5 + 6,75	15,25 (real)
8,5 y 6,75 (reales)	-	8,5 - 6,75	1,75 (real)
8,5 y 6,75 (reales)	*	8,5 * 6,75	57,375 (real)
8,5 y 6,75 (reales)	/	8,5 / 6,75	1,259 (real)
8,5 y 6,75 (reales)	^	8,5 ^ 6,75	1,877 x 10 ⁶ (real)

Datos. Operaciones con los datos

□ Operaciones lógicas (o booleanas)

- Estas operaciones sólo pueden dar como **resultado verdadero o falso**.
- Tenemos dos tipos de operadores en este tipo de operaciones: **operadores de relación** y **operadores lógicos**.
- **Operadores de relación:**

- Muchos lenguajes usan “<>” en lugar de “!=” para el operador “distinto de”
- Se pueden usar con todos los tipos de datos simples: entero, real, carácter o lógico.

□ Igual	=
asignación)	(diferencia con la
□ Menor	<
□ Menor o igual	<=
□ Mayor	>
□ Mayor o igual	>=
□ Distinto	!= o <>

Datos. Operaciones con los datos

□ Operaciones lógicas (o booleanas)

□ Ejemplos de operadores de relación:

Operandos	Operador	Operación	Resultado
35, 9 (enteros)	>	35 > 9	verdadero
35, 9 (enteros)	<	35 < 9	falso
35, 9 (enteros)	==	35 == 9	falso
35, 9 (enteros)	!=	35 != 9	verdadero
5, 5 (enteros)	<	5 < 5	falso
5, 5 (enteros)	<=	5 <= 5	verdadero
5, 5 (enteros)	!=	5 != 5	falso
"a", "c" (caracteres)	==	'a' == 'c'	falso
"a", "c" (caracteres)	>=	'a' >= 'c'	falso
"a", "c" (caracteres)	<=	'a' <= 'c'	verdadero

□ Para operadores de relación con datos lógicos se considera que “falso” es menor que “verdadero”, por lo tanto:

Operandos	Operador	Operación	Resultado
verdadero, falso	>	verdadero > falso	verdadero
verdadero, falso	<	verdadero < falso	falso
verdadero, falso	==	verdadero == falso	falso

Datos. Operaciones con los datos

□ Operaciones lógicas (o booleanas)

- Los operadores lógicos son “and” (y), “or” (o) y “not” (no)
- Sólo se pueden emplear con tipos de datos lógicos.

■ Y o AND

■ O u OR

■ NO o NOT

Tablas de verdad de los operadores lógicos.

A	no A
V	F
F	V

A	B	A AND B
F	F	F
F	V	F
V	F	F
V	V	V

A	B	A OR B
F	F	F
F	V	V
V	F	V
V	V	V

Datos. Operaciones con los datos

Prioridad de los operadores

- Es habitual encontrar varias operaciones en una misma línea.
- En estos casos es imprescindible conocer la **prioridad de los operadores**, ya que **dependiendo del orden puede variar el resultado de la operación**.
 - ▣ Ejemplo: $6 + 4 / 2 \rightarrow$ podría dar 5 u 8
- La prioridad de cálculo respeta las **reglas generales del álgebra**:
 - ▣ La **división** y la **multiplicación** tienen **más prioridad** que la **suma** o la **resta**.
 - ▣ Las prioridades del cálculo se pueden alterar usando **paréntesis**, como en el álgebra.
 - ▣ A igualdad de prioridad entre operadores, la operación se calcula de **izquierda a derecha**, en el sentido de lectura de los operandos.
- **Pero el resto de prioridades pueden variar de un lenguaje de programación a otro.**

Datos. Operaciones con los datos

Prioridad de los operadores

Operador	Prioridad
()	Mayor
^	
-	Signo, operador unario
* , / , div , mod	/ división real \ o div división entera mod es el resto de la div
+ -	Incluida la concatenación
< <= > >=	
== !=	
No	
Y	
O	Menor

Operación	Resultado
$6 + 4 / 2$	8
$(6 + 4) / 2$	5
$(33 + 3 * 4) / 5$	9
$2 \wedge 2 * 3$	12
$3 + 2 * (18 - 4 \wedge 2)$	7
$5 + 3 < 2 + 9$	verdadero
$2 + 3 < 2 + 4$ y $7 > 5$	verdadero
$"A" > "Z" \text{ o } 4 / 2 + 4 > 6$	falso
$"A" > "Z" \text{ o } 4 / (2 + 2) \leq 6$	verdadero

Datos. Expresiones

- Una **expresión** es una combinación de constantes, variables, operadores y funciones.
 - ▣ Se trata de operaciones aritméticas o lógicas como las vistas con anterioridad, pero que además pueden contener variables.
 - ▣ **Ejemplo: $(5 + x) \text{ div } 2$**
 - ▣ En esta expresión aparecen dos constantes (5 y 2), una variable (x) y dos operadores (+ y div)
 - ▣ Los paréntesis sirven para alterar la prioridad de los operadores.
 - ▣ Para resolver la expresión (averiguar su resultado), debemos conocer el valor de la variable x
 - ▣ **Otro ejemplo: $(-b + \text{raiz}(b^2 - 4 * a * c) / (2 * a))$**
- La forma más habitual de encontrar una expresión es combinada con una sentencia de asignación a una variable.
 - ▣ $x1 = (-b + \text{raiz}(b^2 - 4 * a * c) / (2 * a))$
 - ▣ En estos casos, la expresión (a la derecha del “=”) se evalúa y su resultado se asigna a la variable situada a la izquierda del “=”

Datos. Expresiones

Dadas las siguientes variables y constantes:

$X=1$, $Y=4$, $Z=10$, $PI=3.14$, $E=2.71$

evaluar las expresiones:

- a) $2 * X + 0.5 * Y - 1 / 5 * Z$
- b) $PI * X ^ 2 > Y \text{ OR } 2 * PI * X <= Z$
- c) $E ^ (X - 1) / (X * Z) / (X / Z)$
- d) $\text{"DON"} + \text{" JUAN"} = \text{"DON JUAN"} \text{ OR } \text{"A"} = \text{"a"}$
- e) $((3 + 2) ^ 2 - 15) / 2 * 5$
- f) $5 - 2 > 4 \text{ AND NOT } 0.5 = 1 / 2$

Datos. Expresiones

Dadas las siguientes variables y constantes:

$X=1$, $Y=4$, $Z=10$, $PI=3.14$, $E=2.71$ evaluar las expresiones:

- a) $2 * X + 0.5 * Y - 1 / 5 * Z$ SOL: 2
- b) $PI * X^2 > Y \text{ OR } 2 * PI * X \leq Z$ SOL: VERDADERO
- c) $E^{(X - 1) / (X * Z) / (X / Z)}$ SOL: 1
- d) "DON" + " JUAN" = "DON JUAN" OR "A" = "a"
SOL: VERDADERO
- e) $((3 + 2)^2 - 15) / 2 * 5$ SOL: 25
- f) $5 - 2 > 4 \text{ AND NOT } 0.5 = 1 / 2$ SOL: FALSO

Datos. Operaciones con los datos

Funciones

- **Funciones de biblioteca:** son proporcionadas por los lenguajes de programación para operaciones más complejas, como:
 - ▣ Raíces cuadradas, logaritmos, senos, cosenos, ...
- **Funciones propias:** el programador también puede escribir sus propias funciones:
 - ▣ Reciben cero, uno o varios argumentos (datos) y devuelven resultados.
- **PSeInt** permite definir funciones propias y además dispone de una biblioteca de funciones.

Datos. Operaciones con los datos

Funciones

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 número x al valor entero más próximo	Real	Entero
trunc(x)	trunca el número x, es decir, le elimina la parte decimal	Real	Entero
raiz(x)	raiz cuadrada de x	Real o Entero	Real
cuadrado(x)	x^2	Real o Entero	Real o Entero
aleatorio(x)	genera un número al azar entre 0 y x	Entero	Entero

Operación	Resultado
abs(-5)	5
abs(6)	6
redondeo(5.7)	6
redondeo(5.2)	5
trunc(5.7)	5
trunc(5.2)	5
cuadrado(8)	64
raiz(64)	8



2. Paradigmas de programación

Paradigmas de Programación

- Un **paradigma de programación** es un modelo o técnica para el **diseño e implementación de programas**.
- Este modelo determinará cómo será el proceso de diseño y la estructura final del programa.
- Cada uno de los paradigmas existentes tienen sus ventajas e inconvenientes, algunos son más o menos apropiados para resolver determinados tipos de problemas, pero **no es correcto decir que un paradigma es mejor que otro**.
- Existen múltiples paradigmas, incluso hay lenguajes de programación que utilizan varios paradigmas (multiparadigma) como java.
- Utilizar un paradigma de programación derivará en un programa fácil de mantener, entender y corregir.

Paradigmas de Programación

- Los paradigmas se pueden clasificar en **dos grandes grupos**:



- **El paradigma imperativo:** (del latín Imperare, *ordenar*) consiste en una sucesión de instrucciones que describen paso a paso **lo que el programa debe hacer**.
- **El paradigma declarativo:** consiste en describir el resultado final que se busca, es decir, en expresar **qué debe hacer** el programa pero **no cómo hacerlo**. El código es más difícil de entender por el alto grado de abstracción.

Paradigma Imperativo

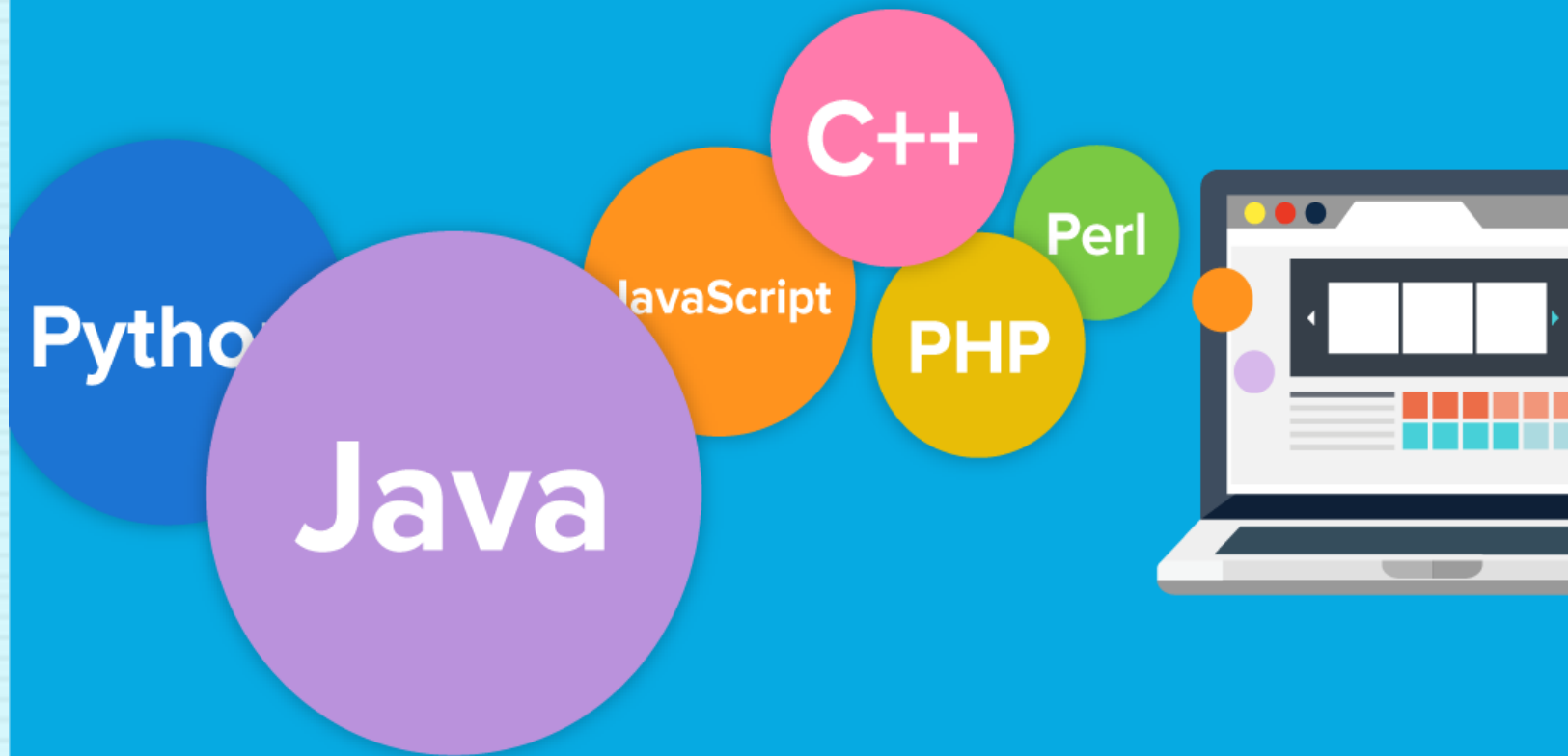
- **Programación convencional o no estructurada:** basado en instrucciones de salto (GOTO) que dificultaban la modificación del código y su reutilización. Un programa era un único archivo.
- **Programación estructurada:** surge para resolver los problemas de la programación convencional. Permite utilizar estructuras que facilitan la modificación, reutilización y mantenimiento de los programas, como los bucles o estructuras de control (Teorema de la P.E). Ej : C, Pascal, Fortran...
 - **Ventajas:**
 - Los programas son fáciles de leer, sencillos y rápidos.
 - Depuración y mantenimiento de los programas es sencillo.
 - **Inconvenientes:**
 - Todo el programa se concentra en un único bloque.
 - No permite reutilización eficaz de código => **programación modular.**
- **Programación procedimental:** amplía el anterior permitiendo desglosar un algoritmo en porciones manejables (**procedimientos**) con la intención de que el código sea más claro.

Paradigma Imperativo

- **Programación modular:** El código se divide en bloques más pequeños e independientes (**módulos**).
- **Programación orientada a objetos:** los programas se componen de objetos **independientes** y **reutilizables** que tienen propiedades y pueden hacer acciones.
 - Está basada en técnicas como herencia, polimorfismo, encapsulamiento...
 - No es una programación tan intuitiva como la estructurada.
 - El código es reutilizable.
 - Facilidad para localizar y depurar un error en un objeto.
 - Lenguajes como Java, C++, PHP son ejemplos.
- **Programación visual, orientada a eventos.**

Paradigma Declarativo

- **Programación funcional:** un programa consta de llamadas a funciones concatenadas en las que cada parte del programa se interpreta como una función. Por ejemplo, LISP, SQL...
- **Programación lógica:** se basa en la lógica matemática. En este paradigma se especifica **qué hacer y no cómo hacerlo**. Se utiliza en inteligencia artificial. Por ejemplo Prolog, Mercury ...
- Algunos lenguajes de programación se incluyen en un único paradigma como **Smalltalk** que es orientado a objetos, otros como **Python** o **Java** son multiparadigma.



3. Lenguajes de Programación

Lenguajes de Programación

- Un **lenguaje de programación** es un conjunto de **reglas sintácticas y semánticas, símbolos y palabras especiales** establecidas para la construcción de programas.
- Está formado por:
 - ▣ **Alfabeto:** conjunto de símbolos permitidos.
 - ▣ **Sintaxis:** reglas y estructura para escribir las instrucciones del lenguaje, es decir, para escribir el código del mismo.
 - ▣ **Semántica:** significado de las instrucciones del lenguaje.
- Un lenguaje de programación es un **lenguaje artificial** creado por el ser humano para escribir programas.

Lenguajes de Programación

- **Clasificación según la cercanía al lenguaje humano**
 - ▣ **De alto nivel:** más próximos al razonamiento humano.
 - ▣ **De bajo nivel:** más próximos al funcionamiento interno de la computadora: (**lenguaje máquina** y ensamblador).
- **Clasificación según la técnica o paradigma de programación utilizado:** (algunos lenguajes pertenecen a varios paradigmas)
 - ▣ **Imperativos:** Pascal, C, Java, C++, etc.
 - ▣ **Orientados a objetos:** Java, PHP, etc.
 - ▣ **Visuales:** Visual Basic.Net, Borland Delphi, etc.
 - ▣ **Declarativos:** LISP, Prolog, etc.

Lenguajes de bajo nivel.

□ Lenguaje Máquina:

- Sus instrucciones son combinaciones de unos y ceros (binario)
- Único lenguaje que entiende directamente el ordenador (no necesita traducción).
- Depende del procesador.
- Fue el primer lenguaje utilizado, pero hoy nadie programa en este lenguaje.
- **Inconvenientes:**
 - Cada programa era válido para un tipo de procesador solamente.
 - Su lectura por el ser humano era extremadamente difícil, por tanto era difícil de mantener.

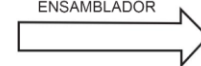
□ Lenguaje Ensamblador: https://es.wikipedia.org/wiki/Lenguaje_ensamblador

- Sustituyó al lenguaje máquina para facilitar la labor de programación. Es difícil de utilizar.
- En lugar de unos y ceros se programa usando mnemotécnicos (instrucciones complejas).
- Necesita **traducción (Programa Ensamblador)** al lenguaje máquina para poder ejecutarse.
- Sus instrucciones son sentencias que hacen referencia a la ubicación física de los datos en el equipo (direcciones de memoria).
- Dependen del hardware de la máquina.
- **Inconvenientes:**
 - Los programas siguen dependiendo del hardware.
 - Los programadores debían conocer la máquina donde programaban (memoria, registros...)

LENGUAJE ENSAMBLADOR

```
ADD CL, AL
MOV CL, 325
SUB CL, 45
```

ENSAMBLADOR



LENGUAJE MÁQUINA

```
0001 1001 0100 1101
0110 0011 0001 1110
1100 0000 0001 1010
```

Lenguajes de alto nivel.

- **Lenguaje de alto nivel basados en código:**
 - ▣ Sustituyeron al lenguaje ensamblador para facilitar la labor de programación
 - ▣ Fáciles de aprender y de utilizar => se reduce el tiempo de desarrollo.
 - ▣ Son independientes del hardware.
 - ▣ En lugar de mnemotécnicos, utilizan sentencias derivadas del idioma inglés.
 - ▣ Necesita traducción al lenguaje máquina => Más lentos.
 - ▣ Son utilizados hoy día, aunque la tendencia es que cada vez menos.
 - **Lenguajes visuales u orientados a eventos:**
 - ▣ Están sustituyendo a los lenguajes de alto nivel basados en código.
 - ▣ En lugar de sentencias escritas, se programa gráficamente usando el ratón y diseñando directamente la apariencia del software.
 - ▣ Su correspondiente código se genera automáticamente.
 - ▣ Necesitan traducción al lenguaje máquina.
 - ▣ Son completamente portables de un equipo a otro.
- <http://www.larevistainformatica.com/LENGUAJES-DE-PROGRAMACION-listado.html>

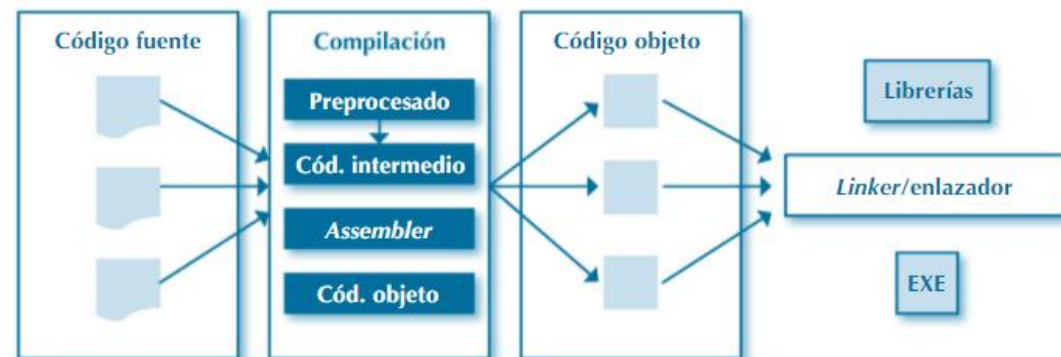
Lenguajes compilados e interpretados.

- Las instrucciones escritas por el programador en un determinado lenguaje de programación se denominan **código fuente**.
- El código fuente ha de ser traducido al único lenguaje que entiende el ordenador, llamado **lenguaje máquina** (en binario) dando lugar a lo que denominados **código objeto**.
- Este proceso de traducción de **código fuente** a **código objeto o lenguaje máquina** depende del lenguaje de programación y puede ser de dos tipos:
 - Compilación.
 - Interpretación.

Lenguajes compilados e interpretados.

□ Compilación

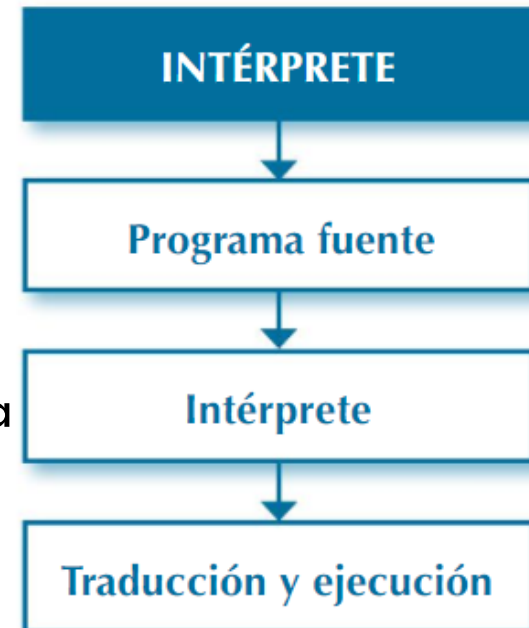
- El **compilador** es un programa que traduce el código fuente a código objeto.
- El compilador debe ser específico del S.O. donde se vaya a ejecutar el programa.
- El compilador realiza la traducción e informa de los **errores de sintaxis**.
- El código objeto generado puede ser ejecutado por la máquina directamente o puede necesitar otros pasos como ensamblado, enlazado y carga.
- Una vez finalizado obtenemos el **código ejecutable**.
- Una vez se haya obtenido el programa ejecutable no se necesita software específico para su ejecución.



Lenguajes compilados e interpretados.

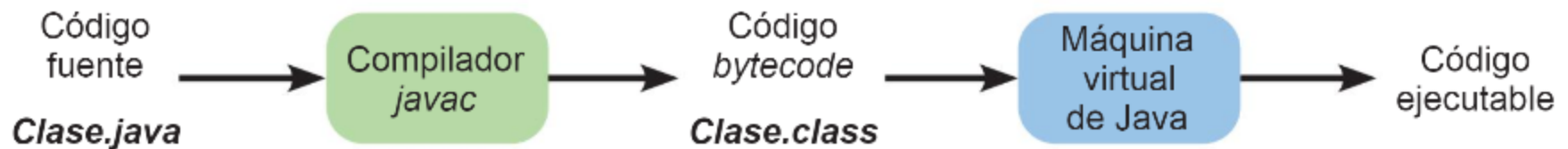
□ Interpretación

- ▣ El intérprete es un programa que se encarga de **analizar** y **ejecutar** el programa escrito en un L.P.
- ▣ El intérprete **lee** una a una las instrucciones, las **analiza, traduce** a código máquina directamente (si no hay error) y **ejecuta** instrucción a instrucción del código fuente.
- ▣ **NO se genera código objeto ni programa ejecutable.**
- ▣ Generan programas de **menor tamaño** pero son **más lentos**.
- ▣ Los lenguajes interpretados necesitan disponer en máquina donde se van a ejecutar de un programa intérprete, mientras que los compilados no.
- ▣ Ejemplos: PHP, JavaScript, Python, ...



Lenguajes compilados e interpretados.

- Hay lenguajes como Java que son **pseudo-compilados** o **pseudo-interpretados**.
- En Java el código fuente es compilado a **bytecodes** (estructura similar a instrucciones máquina) que son **independientes del hardware**.
- Java requiere de una **máquina virtual** que hace de **intérprete** que traduce los bytecodes a código de la máquina en cuestión.





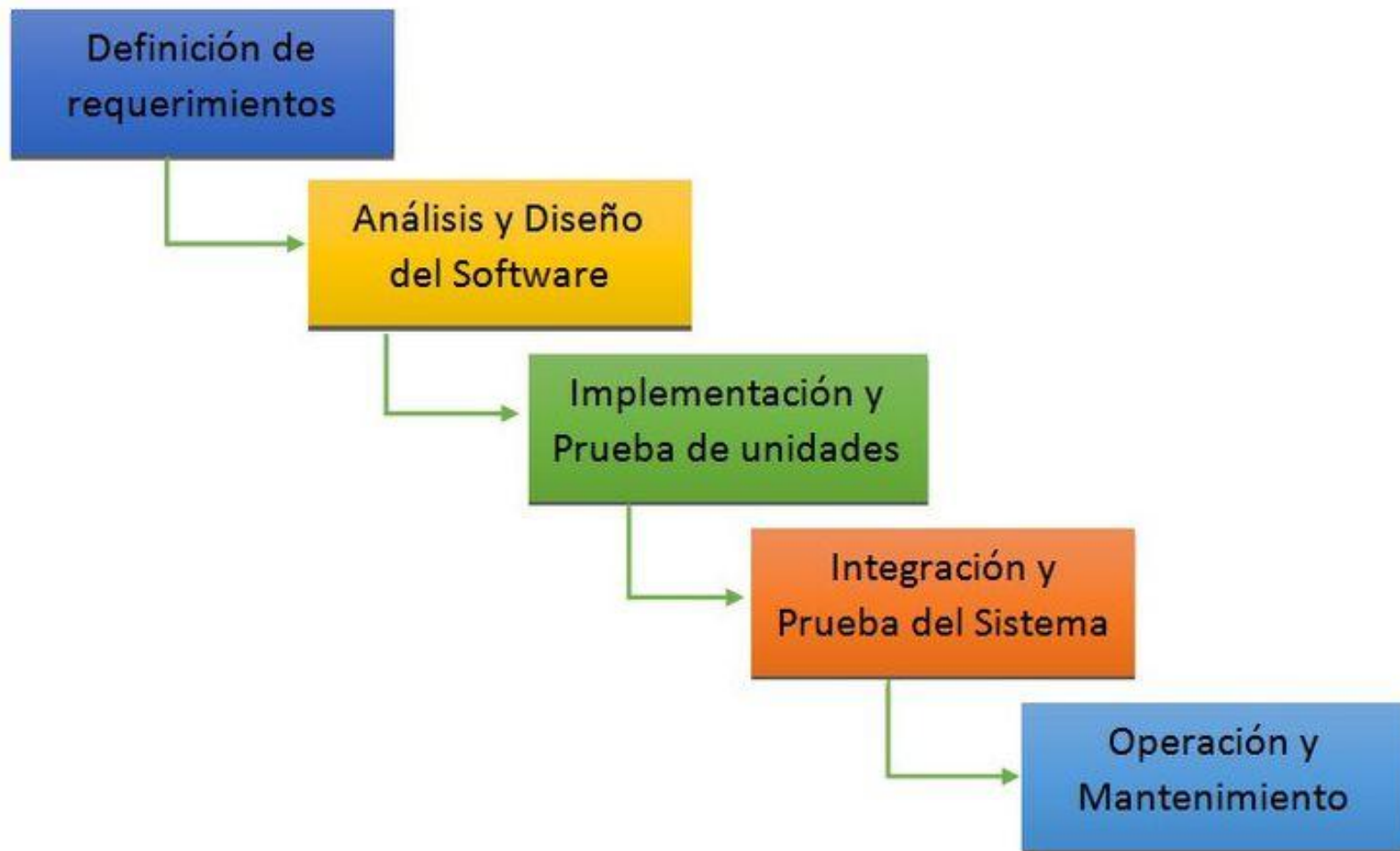
4. Errores y calidad de los programas.

Tipos de errores

- **Tipos de Errores.** Según la fase en la que se detecten los errores pueden ser:
 - Errores de **compilación** (sintácticos).
 - Errores de **ejecución**. Más difíciles de detectar porque dependen de los datos de entrada.
 - Errores de **lógica**. Cuando se obtienen resultados incorrectos. Para detectarlos se emplean ejecuciones de prueba con varios grupos de datos de ensayo. Después se comprueban los resultados con los que se deben obtener.
 - Errores de **especificación**: El peor y el más costoso de corregir, y se debe a la realización de unas especificaciones incorrectas motivadas por una mala comunicación entre el analista y quien plantea el problema. Hay que repetir el trabajo.

Calidad de los programas

- Las características generales que debe reunir un programa son:
 - ▣ **Legibilidad:** ha de ser claro y sencillo para una fácil lectura y comprensión.
 - ▣ **Fiabilidad:** ha de ser robusto, es decir, capaz de recuperarse frente a errores o usos inadecuados.
 - ▣ **Portabilidad:** su diseño debe permitir la codificación en diferentes lenguajes de programación, así como su instalación en diferentes sistemas.
 - ▣ **Modificabilidad:** ha de facilitar su mantenimiento, esto es, las modificaciones y actualizaciones necesarias para adaptarlo a una nueva situación. Relacionado con la legibilidad.
 - ▣ **Eficiencia:** se deben aprovechar al máximo los recursos de la computadora minimizando la memoria utilizada y el tiempo de proceso de ejecución. Esto hoy en día no se cumple: Windows, Office, ...



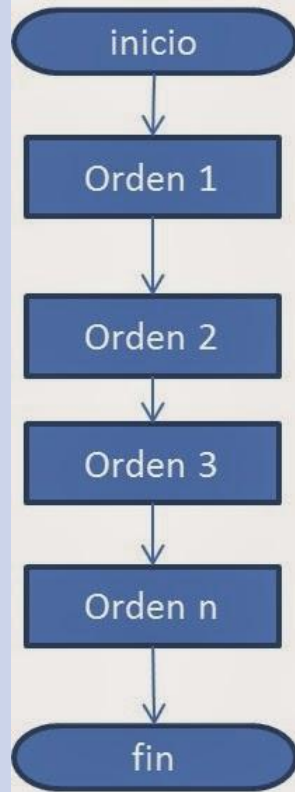
5. Fases en la creación de un programa

Fases de un programa

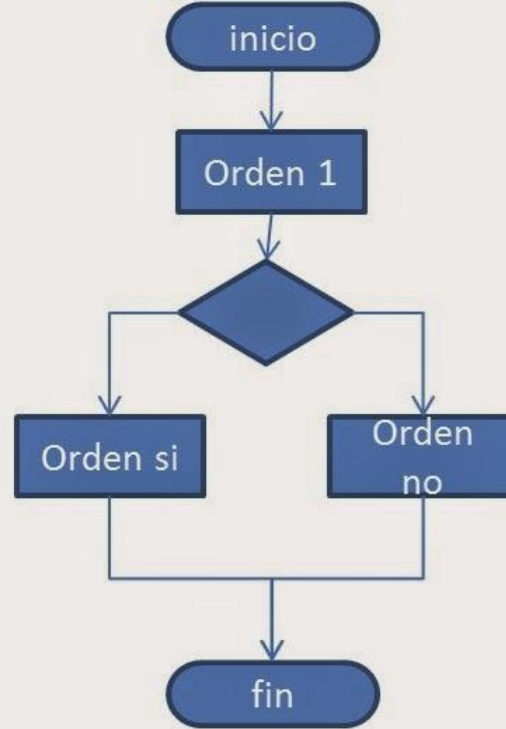
- Todo programa evoluciona a través de una serie de etapas o fases denominadas en su conjunto como **Ciclo de vida del software**: actividades y las tareas involucradas en el desarrollo, explotación y el mantenimiento de un producto software, abarcando la vida del sistema desde la definición de los requisitos hasta la finalización de su uso.

Fases de un programa

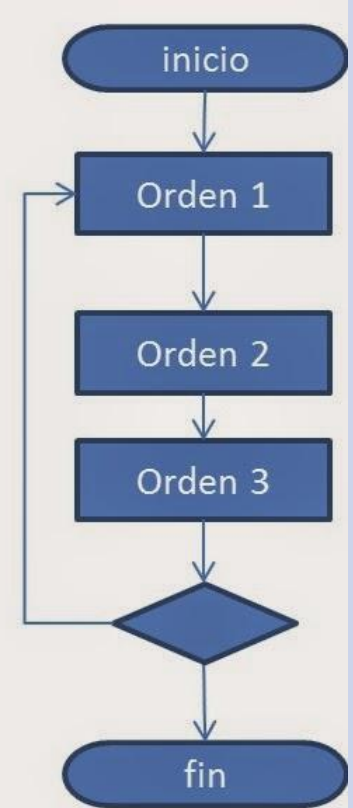
1. **Análisis.** Se trata de analizar las necesidades del usuario del software. Construye un modelo de requisitos. Documentación entendible, completa y fácil de verificar y modificar. **Qué hacer.**
2. **Diseño. Cómo hay que hacerlo.** Se traducen los requisitos en componentes software (tablas de la B.D, programas, funciones, clases con atributos y métodos, interfaces de usuario, etc)
3. **Codificación.** Traducir el diseño realizado al lenguaje de programación escogido. Se obtiene **código ejecutable.**
4. **Pruebas.** Comprobar que se cumplen criterios de corrección y calidad. Garantizan el correcto funcionamiento del sistema.
5. **Explotación.** Instalación y puesta en marcha en el entorno del cliente.
6. **Mantenimiento.** Después de la entrega del software. Adaptarse a los cambios. Cambios por errores. Adaptación al entorno (p.ej. cambio de SO), mejoras funcionales....
7. **Documentación:** en cada etapa se generan documentos. Cada etapa tiene como entrada los documentos de la etapa anterior y obtiene nuevos documentos.



Lineal



selectiva



cíclica

6. Programación estructurada

Programación estructurada

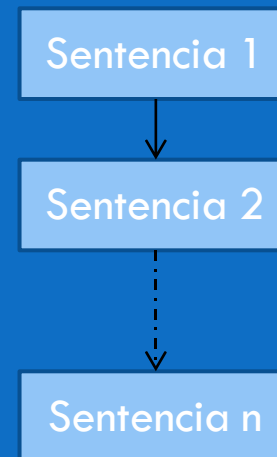
- En 1966, Böhm y Jacopini demostraron que se puede escribir cualquier programa utilizando sólo tres tipos de estructuras de control (instrucciones). Se llama el **Teorema de la programación estructurada**:
 - secuencial
 - selectiva (o condicional)
 - repetitiva (bucles)
- **Características de un programa**:
 - Posee un sólo punto de inicio y un sólo punto de fin .
 - Existe al menos un camino que parte del inicio y llega hasta el fin.
 - No existen bucles infinitos.

Programación estructurada: instrucciones

- **Instrucciones Secuenciales:**
 - Una instrucción sigue a otra

Sentencia 1;
Sentencia 2;
.....;
Sentencia n;

Pseudocódigo



Ordinograma

Programación estructurada: instrucciones

- **Instrucciones Secuenciales:**
 - Ejemplo: sumar dos números

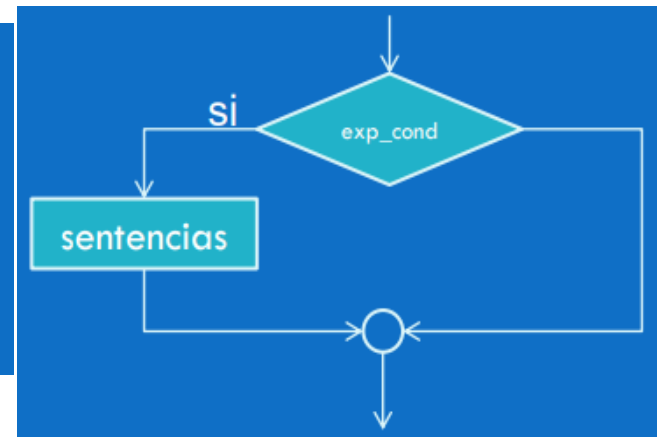
```
algoritmo suma
variables
  A, B, suma son enteros
inicio
  leer (A)
  leer (B)
  suma = A + B
  escribir (suma)
fin
```

Programación estructurada: instrucciones

- Instrucciones **Alternativas**: las instrucciones se ejecutan si se cumple una determinada condición
 - ▣ **Simple**

```
si <expresión_condicional> entonces  
    sentencias;  
finSi;
```

Pseudocódigo



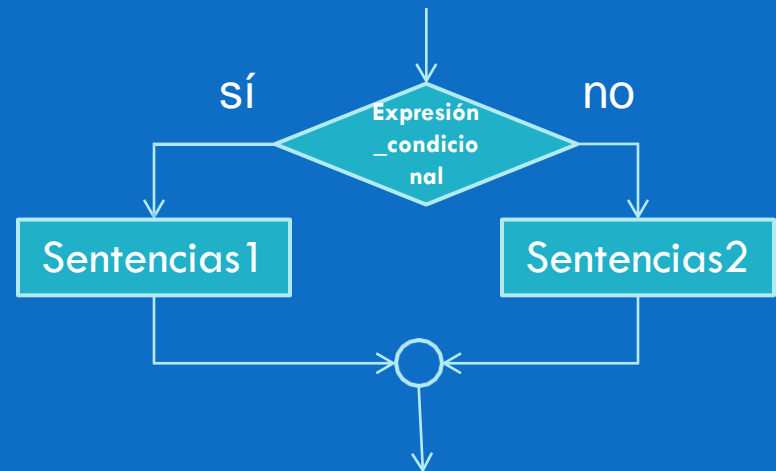
Ordinograma

Programación estructurada: instrucciones

- Instrucciones **Alternativas**: si se cumple una condición se ejecutan una sentencias y si no, otras.
 - ▣ **Doble.**

```
si <expresión_condicional> entonces  
    sentencias1;  
si no  
    sentencias2;  
finSi;
```

Pseudocódigo

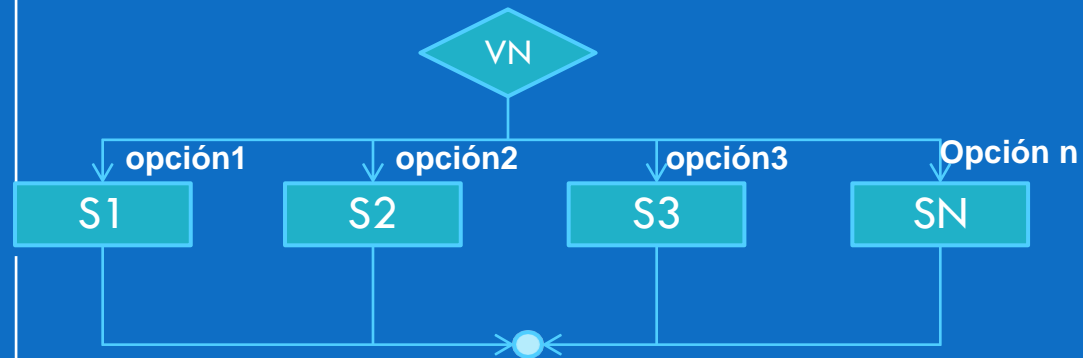


Ordinograma

Programación estructurada: instrucciones

- Instrucciones **Alternativas**: se barajan múltiples condiciones.
 - ▣ **Múltiple.**

```
Según <variable_numérica> hacer  
  <opción1>:  
    <setencias1>;  
  <opción2>:  
    <setencias2>;  
  .....  
  de otro modo:  
    <setenciasn>;  
finSegun
```



Pseudocódigo

Ordinograma

Programación estructurada: instrucciones

- Instrucciones **Repetitivas** o iterativas (Bucles):
 - permiten la ejecución repetida de un conjunto de sentencias.
- Se componen de:
 - ▣ **Cuerpo del bucle** → instrucciones que se ejecutan repetidamente
 - ▣ **Condición de salida** → debe evitarse bucles infinitos

Programación estructurada: instrucciones

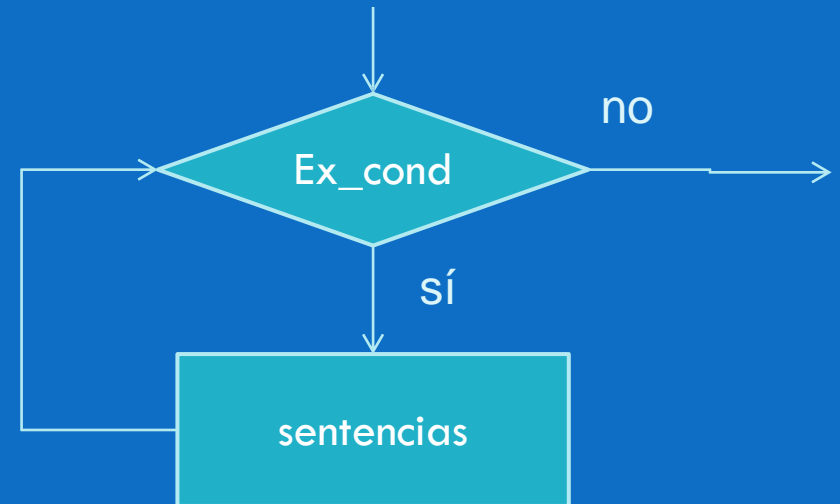
□ Instrucciones **Repetitivas** o iterativas (Bucles

▣ **Mientras**

La condición de entrada al bucle se evalúa en primer lugar.
El cuerpo del bucle puede llegar a no ejecutarse nunca.

```
Mientras <expresión-condicional> hacer  
    <sentencias>  
FinMientras;
```

Pseudocódigo



Ordinograma

Programación estructurada: instrucciones

- Instrucciones **Repetitivas** o iterativas (Bucles
 - ▣ **Mientras**

Ejemplo *Escribir un algoritmo que muestre en la pantalla todos los números enteros entre 1 y 100*

```
algoritmo contar
variables
  cont es entero
inicio
  cont = 0
  mientras (cont <= 100) hacer
    inicio
      cont = cont + 1
      escribir (cont)
    fin
  fin
fin
```

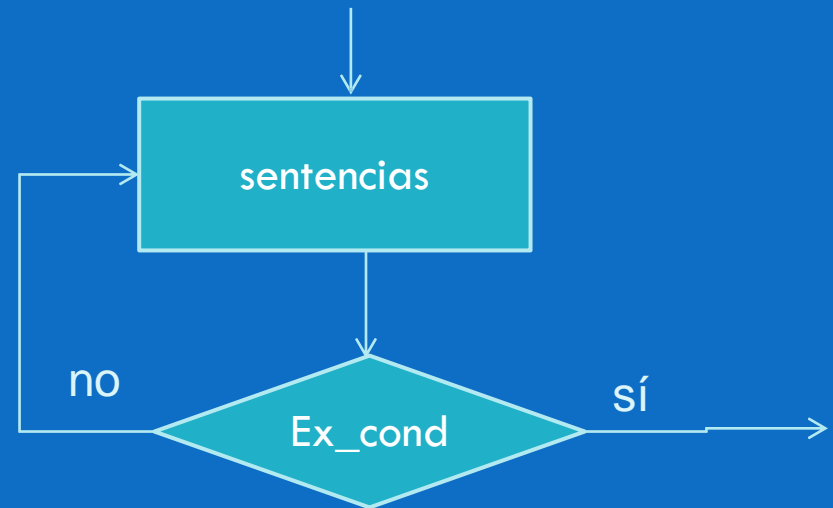
Programación estructurada: instrucciones

- Instrucciones **Repetitivas** o cíclicas o iterativas.

- ▣ **Repetir ... hasta.**

La condición de entrada al bucle se evalúa al final del cuerpo del bucle. El cuerpo del bucle siempre se ejecuta como mínimo la primera vez.

Repetir
 <sentencias>
Hasta <expresión-condicional>;



Pseudocódigo

Ordinograma

Programación estructurada: instrucciones

- Instrucciones **Repetitivas** o cíclicas o iterativas.
 - ▣ **Repetir ... hasta.**

Ejemplo *Diseñar un algoritmo que escriba todos los números enteros entre 1 y 100, pero esta vez utilizando un bucle "repetir" en lugar de un bucle "mientras"*

```
algoritmo contar
variables
  cont es entero
inicio
  cont = 0
  repetir
    inicio
      cont = cont + 1
      escribir (cont)
    fin
  mientras que (cont <= 100)
fin
```

Programación estructurada: instrucciones

□ Instrucciones **Repetitivas** o cíclicas o iterativas.

▣ **Para.**

La condición de entrada al bucle se evalúa en primer lugar. Este tipo de bucles se utiliza cuando sabemos de antemano cuántas veces debe repetirse el cuerpo del bucle

```
Para <Vc> = <Vi> hasta <Vf> incremento <valor>  
    <sentencias>  
FinPara;
```

Pseudocódigo



Ordinograma

Programación estructurada: instrucciones

- Instrucciones **Repetitivas** o cíclicas o iterativas.
 - ▣ **Para.**

Ejemplo 1 *Diseñar un algoritmo que escriba todos los números enteros entre 1 y 100, utilizando un bucle "para"*

```
algoritmo contar
variables
  cont es entero
inicio
  para cont desde 1 hasta 100 hacer
    inicio
      escribir (cont)
    fin
  fin
fin
```

Contadores, acumuladores, conmutadores

Contadores

- Un **contador** es una variable (casi siempre de tipo entero) cuyo valor se incrementa o decrementa un número determinado en cada repetición de un bucle

- El contador se suele utilizar de este modo:

- Se inicializa antes de que comience el bucle

$\text{cont} \leftarrow 1$

- Se modifica dentro del cuerpo del bucle (lo más habitual es que se incremente su valor en una unidad)

$\text{cont} \leftarrow \text{cont} + 1$

- Se utiliza en la condición asociada al bucle (normalmente se compara con un valor máximo (o mínimo))

Contadores, acumuladores, conmutadores

Ejemplo: Contadores

Calcular el número de aprobados entre 50 notas que se piden al usuario

```
Programa: APROBADOS
Entorno:
    NUMERO (contador) es numérico entero
    APROB (contador de aprobados) es numérico entero
    NOTA es numérico entero
Algoritmo:
    APROB ← 0
    NUMERO ← 0
    Mientras NUMERO < 50 hacer
        NUMERO ← NUMERO + 1
        Leer NOTA
        Si NOTA >= 5 entonces
            APROB ← APROB + 1
        Finsi
    Finmientras
    Escribir APROB
Finalgoritmo
```

Contadores, acumuladores, conmutadores

Acumuladores

- Un **acumulador** es una variable cuyo objetivo es acumular datos procedentes de algún cálculo previo.
 - Deben ser inicializadas.
 - Cuando se usan dentro del cuerpo del bucle, normalmente:

$$\text{acum} \leftarrow \text{acum} + N$$

- Ejemplo: sumar los 10 primeros números naturales.

Contadores, acumuladores, conmutadores

Ejemplo: Acumuladores

Calcular la suma y el producto de los 10 primeros números naturales.

```
Programa: SUMANAT
Entorno:
    SUMA (acumulador) es numérico entero
    PRODUCTO (acumulador) es numérico entero
    CONTA es numérico entero
Algoritmo:
    SUMA ← 0
    PRODUCTO ← 1
    CONTA ← 1
    Mientras CONTA ≤ 10 hacer
        SUMA ← SUMA + CONTA
        PRODUCTO ← PRODUCTO * CONTA
        CONTA ← CONTA + 1
    Finmientras
    Escribir SUMA, PRODUCTO
Finalgoritmo
```

Contadores, acumuladores, conmutadores

Conmutadores

- Un **conmutador** (o interruptor) es una variable que sólo puede tomar dos valores (booleano)
- Recibirá uno de los dos valores posibles antes de entrar en el bucle.
- Dentro del cuerpo del bucle debe cambiarse ese valor bajo ciertas condiciones.
- Utilizando el conmutador en la condición de salida del bucle, puede controlarse el número de repeticiones del mismo.
- **Ejemplo:** Escribir un algoritmo que sume todos los números positivos introducidos por el usuario a través del teclado. Para terminar de introducir números, el usuario tecleará un número negativo (variable booleana conmutador).

Contadores, acumuladores, conmutadores

```
Algoritmo UsandoConmutador
    suma, num Es Entero
    esPositivo Es Logico
    suma <- 0
    esPositivo <- Verdadero
    Escribir "Introduce un número"
    Leer num
    Si num Es Positivo Entonces
        Mientras esPositivo es Verdadero Hacer
            suma <- suma + num
            Escribir "Introduce otro numero"
            Leer num
            si num no Es Mayor Que 0 Entonces
                esPositivo <- Falso
            FinSi
        FinMientras
    FinSi
    Escribir suma
FinAlgoritmo
```

Contadores, acumuladores, conmutadores

Conmutadores

- A veces, el conmutador puede tomar más de dos valores.
- En ese caso, uno de los valores de la variable es el que hace que el bucle termine, por lo que se denomina a ese valor “centinela”.
- Ejemplo: Escribir un algoritmo que sume todos los números positivos introducidos por el usuario a través del teclado. Para terminar de introducir números, el usuario tecleará un número negativo (un valor negativo será el centinela)

Contadores, acumuladores, conmutadores

Algoritmo UsandoCentinela

 suma, num Es Entero

 suma <- 0

 Escribir "Introduce un numero"

 Leer num

 Mientras num es Mayor que 0 Hacer

 suma <- suma + num

 Escribir "Introduce otro numero"

 Leer num

 FinMientras

 Escribir suma

FinAlgoritmo

Contadores, acumuladores, conmutadores

Conmutadores

```
Programa: NOTAS
Entorno:
    NOTA es numérico entero
    NOTA10 booleano (switch)
Algoritmo:
    NOTA10 ← FALSO
    Leer NOTA
    Mientras NOTA <> -1 hacer
        Si NOTA = 10 entonces
            NOTA10 ← CIERTO
        Finsi
        Leer NOTA
    Finmientras
    Si NOTA10 = CIERTO entonces
        Escribir "Hubo 10"
    Sino
        Escribir "No hubo 10"
    Finsi
Finalgoritmo
```

Programación Estructurada. Reglas de Estilo.

Reglas de Estilo

- La escritura de un algoritmo debe ser siempre lo más clara posible, ya se esté escribiendo en pseudocódigo o en un lenguaje de programación real.
- ¿Por qué? → los algoritmos pueden llegar a ser muy complejos, si además la escritura es “sucia” y desordenada → ininteligibles
- Es importante acostumbrarse a respetar ciertas reglas básicas de estilo.
- Cada programador puede luego desarrollar su estilo propio, pero siempre dentro de un marco aceptado por la mayoría.

Programación Estructurada. Reglas de Estilo.

1. Partes de un algoritmo

- Los algoritmos deberían tener siempre una estructura de 3 partes:
 - **Cabecera** → contiene el nombre del programa o algoritmo
 - **Declaraciones** → contiene las declaraciones de variables y constantes que se usan en el algoritmo
 - **Acciones** → son el cuerpo en sí del algoritmo, es decir, las instrucciones
- Algunos lenguajes, como por ejemplo C, son lo bastante flexibles como para permitir saltarse esta estructura, pero es una buena costumbre respetarla siempre.
- En lenguajes orientados a objetos, esta estructura tiene sus matices, pero es bastante similar.

Programación Estructurada. Reglas de Estilo.

2. Documentación

- ❑ La documentación comprende el conjunto de información interna y externa que facilita el mantenimiento de un programa.
- ❑ **Documentación externa:** guías de instalación, guías de usuario, ...
- ❑ **Documentación interna:**
 - ❑ Se suele plasmar mediante comentarios significativos que acompañan a las instrucciones del algoritmo
 - ❑ Dependiendo del lenguaje de programación los comentarios se introducen utilizando unos determinados símbolos
 - ❑ Java → `//` para comentarios de una línea y `/* ...*/` para multilinea
 - ❑ Buena costumbre → incluir un comentario al principio de cada algoritmo (procedimiento, función) que explique bien lo que hace y si es necesario el autor, fecha de modificación, ...
 - ❑ OJO! comentar un programa en exceso puede ser contraproducente

Programación Estructurada. Reglas de Estilo.

Documentación

- Ejemplo de algoritmo demasiado comentado

Ejemplo *Escribir un algoritmo que sume todos los números naturales de 1 hasta 1000*

```
algoritmo sumar1000
/* Función: Sumar los números naturales entre 1 y 1000
  Autor:   Jaime Tralleta
  Fecha:   12-12-04 */

variables
  cont es entero                /* variable contador */
  suma es entero                /* variable acumulador */
  N es entero

inicio
  suma = 0                        /* se pone el acumulador a 0 */
  para cont desde 1 hasta 1000 hacer /* repetir 1000 veces */
  inicio
    suma = suma + cont           /* los números se suman al acumulador */
  fin
  escribir (suma)
fin
```


Programación Estructurada. Reglas de Estilo.

3. Estilo de escritura

- Sangrías
 - Cada bloque de código que pertenezca a otro deberá tener una sangría → los bloques están marcados por: inicio-fin, { }, ..., dependiendo del lenguaje de programación.
 - Asimismo, un algoritmo es más fácil de leer si los comentarios tienen todos la misma sangría.

Ejemplo *Escribir un algoritmo que determine, entre dos números A y B, cuál es el mayor o si son iguales. Observa bien las sangrías de cada bloque de instrucciones, así como la posición alineada de los comentarios.*

```
algoritmo comparar
// Función: Comparar dos números A y B
variables
  A, B son enteros
inicio
  leer (A)                // leemos los dos números del teclado
  leer (B)
  si (A = B) entonces      // los números son iguales
  inicio
    escribir ('Los dos números son iguales')
  fin
  si_no                    // los números son distintos, así que
  inicio                  // vamos a compararlos entre sí
    si (A > B) entonces
    inicio
      escribir ('A es mayor que B')
    fin
    si_no
    inicio
      escribir ('B es mayor que A')
    fin
  fin
fin
```

Programación Estructurada. Reglas de Estilo.

3. Estilo de escritura

- Tipografía
 - Conviene utilizar una fuente de ancho fijo (tipo Courier o Lucida Console)
- Espacios
 - Otro elemento que aumenta la legibilidad es espaciar suficientemente (pero no demasiado) los elementos de cada instrucción
 - Por la misma razón conviene dejar líneas en blanco entre bloques de instrucciones que estén relacionadas

```
si (a>b)y(c>d*raiz(k))entonces a=k+5.7*b
```

```
si (a > b) y (c > d * raiz(k) ) entonces a = k + 5.7 * b
```

Programación Estructurada. Reglas de Estilo.

3. Estilo de escritura

□ **Identificadores**

- En los **identificadores de variables** (o de constantes) es muy importante utilizar nombres que sean significativos (deben dar idea de la información que almacena una variable)
- También es importante que no sean excesivamente largos
- Lo anterior también es aplicable a los nombres de los algoritmos, procedimientos y funciones.
- Muchos lenguajes de programación distinguen entre mayúsculas y minúsculas (Java)
- Identificadores en minúscula
- Notación “camel case”

Programación Modular

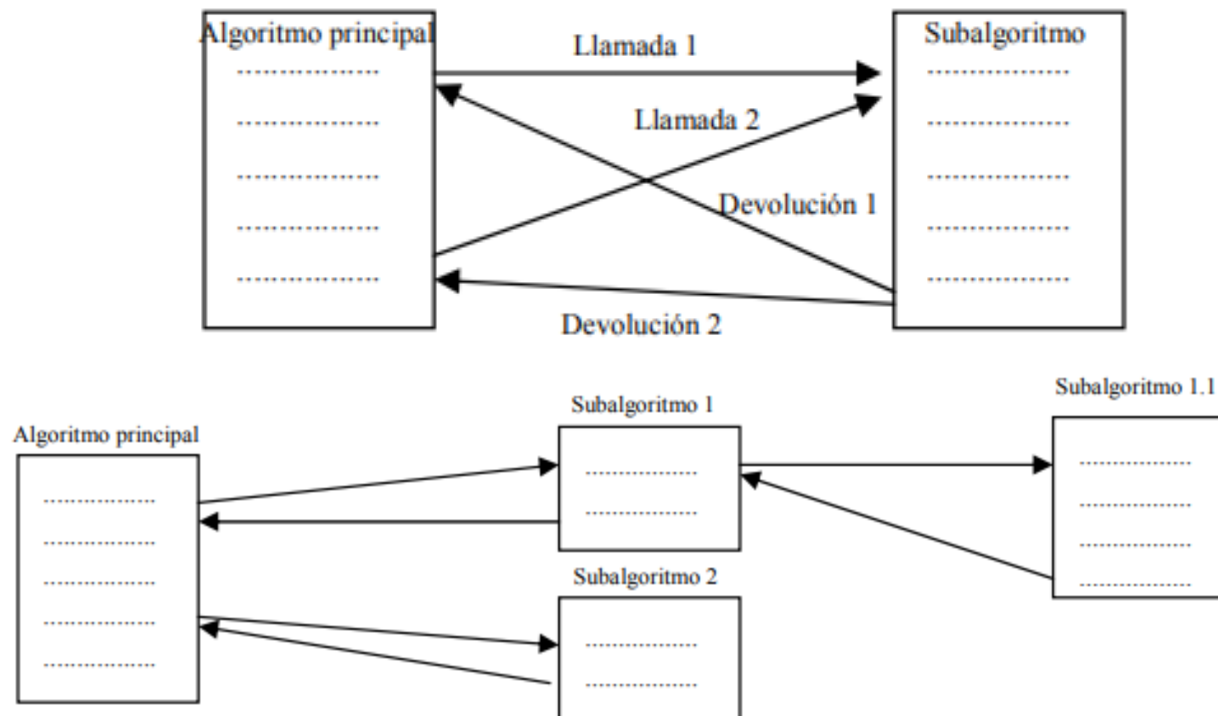
- Resuelve un problema descomponiéndolo en **subproblemas** más simples
→ cada **subproblema** se resuelve mediante un módulo independiente del resto.
- **Ventajas:**
 - Facilita la comprensión del problema
 - Mejora la legibilidad y claridad de los programas
 - Permite que varios programadores trabajen a la vez en el mismo programa
 - Reduce el tiempo de desarrollo ya que se favorece la reutilización de los módulos de código
 - Mejora la fiabilidad ya que es más sencillo diseñar y depurar módulos más pequeños
 - Facilita el mantenimiento de programas
- La Programación Estructurada y Modular no son incompatibles, son complementarias.

Programación Modular

- ❑ La técnica **Divide y Vencerás** permite dividir un problema complejo en **subproblemas** de forma sucesiva hasta que obtengamos problemas lo suficientemente sencillos llamados **MÓDULOS**.
- ❑ Una vez resueltos los subproblemas deben ser combinados para dar solución al problema original (PROGRAMA PRINCIPAL). Desde el **programa principal** invocamos a dichos módulos para que se ejecuten.
- ❑ Esta técnica también se conoce como **Diseño Descendente** o Top-down.
- ❑ La mayoría de Lenguajes de programación admiten esta técnica denominando a los módulos de código procedimientos o funciones.

Programación Modular

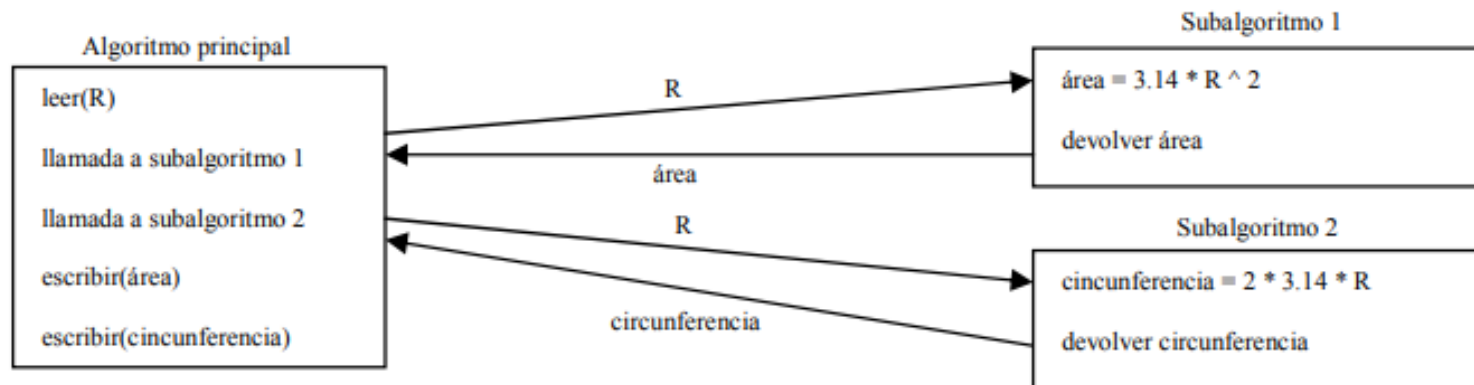
- El problema principal se resuelve en un algoritmo o **módulo principal** mientras que los **subproblemas** se resuelven en los subalgoritmos o **módulos**.
- El algoritmo principal invoca a los módulos a través de su nombre:



Programación Modular

- **Ejemplo:** Diseñar un algoritmo que calcule el área y la circunferencia de un círculo cuyo radio se lea por teclado. Se trata de un problema muy simple que puede resolverse sin aplicar el método divide y vencerás, pero lo utilizaremos como ilustración.

Dividiremos el problema en dos subproblemas más simples: por un lado, el cálculo del área, y, por otro, el cálculo de la circunferencia. Cada subproblema será resuelto en un subalgoritmo, que se invocará desde el algoritmo principal. La descomposición en algoritmos y subalgoritmos sería la siguiente (se indican sobre las flechas los datos que son intercambiados entre los módulos):



Programación Modular

Algoritmo calculaAreaCircunferencia

Definir radio, area, circunferencia Como Real

Escribir "Introduce el radio del circulo"

Leer radio

$area \leftarrow PI * radio * radio$

Escribir "Area ", area

$circunferencia \leftarrow 2 * PI * radio$

Escribir "Circunferencia ", circunferencia

FinAlgoritmo

Algoritmo calculaAreaCircunferenciav2

Definir radio, area, circunferencia Como Real

Escribir "Introduce el radio del circulo"

Leer radio

Escribir "Area ", calcula_area(radio)

$circunferencia \leftarrow calcula_circunferencia(radio)$

Escribir "Circunferencia ", circunferencia

FinAlgoritmo

Funcion area \leftarrow calcula_area (r)

$area \leftarrow PI * r * r$

FinFuncion

Funcion circunferencia \leftarrow calcula_circunferencia(r)

$circunferencia \leftarrow 2 * PI * r$

FinFuncion

Programación Modular

- **¿Hasta dónde se debe descomponer un algoritmo?**
 - La respuesta nos la dará el sentido común y la experiencia al diseñar algoritmos, pero como norma general ningún módulo debería constar de más de 30-40 líneas de código (**Ojo!** No es una regla aplicable a todos los casos)
 - Tampoco es conveniente una descomposición excesiva, módulos de 2-3 líneas de código, aunque dependerá del problema general.
- Codificamos los algoritmos y subalgoritmos en programas o subprogramas respectivamente en un determinado lenguaje de programación.
- Dependiendo de dicho lenguaje los subalgoritmos podrán ser de 2 tipos:
 - **Funciones:** pueden recibir argumentos (no obligatorio) y SIEMPRE devuelven un valor como resultado.
 - **Procedimientos:** pueden recibir argumentos (no obligatorio) y no devuelven ningún valor como resultado.

Programación Modular: Funciones

□ Pseudocódigo de la declaración de una función:

```
tipo_resultado función nombre_función(lista_de_argumentos)
constantes
    lista_de_constantes
variables
    lista_de_variables
inicio
    acciones
    devolver (expresión)
fin
```

- tipo_resultado es el tipo de datos del resultado que devuelve la función
- nombre_función es el identificador de la función
- lista_de_argumentos es una lista con los parámetros que se le pasan a la función: param_1 es tipo_datos, param2 es tipo_datos, ...
- devolver (expresión) devuelve el resultado de evaluar la expresión cuyo tipo de dato será el declarado

Ejemplo *Declaración de una función que calcule el área de un círculo. El radio se pasa como argumento de tipo real.*

```
real función área_círculo (radio es real)
variables
    área es real
inicio
    área = 3.14 * radio ^ 2
    devolver (área)
fin
```

Programación Modular: Funciones

- **Invocación a la función desde el programa principal:**
 - Al invocar a las funciones se debe recoger el resultado que devuelven

Ejemplo 1 *Escribir un algoritmo que calcule el área de un círculo mediante el empleo de la función vista en el ejemplo anterior.* La función `área_círculo()` que acabamos de ver puede ser invocada desde otro módulo, igual que invocamos las funciones de biblioteca como `raiz()` o `redondeo()`

```
algoritmo círculo
variables
    A, B, R son reales
inicio
    leer(R)
    A = área_círculo(R)
    escribir(A)
fin
```

Programación Modular: Funciones

□ Invocación a la función desde el programa principal:

Ejemplo 2 *Escribir un algoritmo que calcule el cuadrado y el cubo de un valor X introducido por teclado, utilizando funciones. Aunque el algoritmo es simple y podría resolverse sin modularidad, forzaremos la situación construyendo dos funciones, `cuadrado()` y `cubo()`:*

```
algoritmo cuadrado_cubo
variables
    N, A, B son reales
inicio
    leer(N)
    A = cuadrado(N)
    B = cubo(N)
    escribir("El cuadrado es ", A)
    escribir("El cubo es ", B)
fin

real función cuadrado (número es real)    // Devuelve el cuadrado de un número
inicio
    devolver (número ^ 2)
fin

real función cubo (número es real)        // Devuelve el cubo de un número
inicio
    devolver (número ^ 3)
fin
```

Programación Modular: Procedimientos

□ Pseudocódigo de la declaración de un procedimiento:

```
procedimiento nombre_procedimiento(lista_de_argumentos)
  constantes
    lista_de constantes
  variables
    lista_de variables
  inicio
    acciones
  fin
```

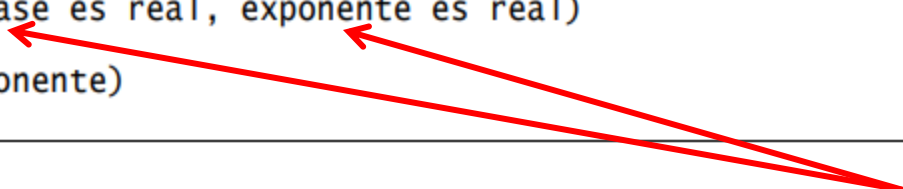
- Los **procedimientos** no devuelven ningún valor, pero a veces es necesario que una función devuelva más de un valor. En ese caso podemos hacer que los procedimientos devuelvan 0,1 ó más resultados a través de los parámetros → Paso de parámetros:
- **Por valor:** forma más sencilla; no permite al subalgoritmo devolver resultados en los parámetros.
- **Por referencia:** es más complejo; permite a los subalgoritmos devolver resultados en los parámetros.

Programación Modular: Procedimientos

□ Paso de parámetros por valor:

Ejemplo *Una función que calcula la potencia de un número elevado a otro*


```
real función potencia(base es real, exponente es real)
inicio
  devolver (base ^ exponente)
fin
```



En la declaración base y exponente se denominan **parámetros formales**

- En la invocación a la función, se pasan valores en los parámetros a la función y se denominan parámetros actuales

```
A = 5
B = 3
C = potencia(A,B)
```



- Al invocar el subalgoritmo, los **parámetros actuales** son asignados a los parámetros formales en el mismo orden en el que fueron escritos. Se hace una copia del de los parámetros actuales a los parámetros formales. Dentro del subalgoritmo, los parámetros se pueden utilizar como si fueran variables. Cualquier cambio sobre los parámetros formales dentro de la función **NO** afecta al valor de los parámetros actuales en el módulo principal.

Programación Modular: Procedimientos

□ Paso de parámetros por referencia:

- En el paso de parámetros por referencia se produce una ligadura entre el parámetro actual y el parámetro formal, de modo que si el parámetro formal se modifica dentro del subalgoritmo, el parámetro actual, propio del algoritmo principal, también será modificado.
- Por defecto, los argumentos pasan sus parámetros por valor excepto cuando indiquemos que el paso es por referencia colocando el símbolo * (asterisco) delante del nombre del argumento.

Ejemplo Escribiremos el mismo subalgoritmo de antes, pero utilizando un procedimiento (que, en principio, no devuelve resultados) en lugar de una función.

```
procedimiento potencia(base es real, exponente es real, *resultado es real)
inicio
    resultado = base ^ exponente
fin
```

- La invocación del subalgoritmo se hace del mismo modo que hasta ahora, pero delante del parámetro que se pasa por referencia debe colocarse el símbolo &:

```
A=5  B=3  C=0
potencia (A,B,&C)
```