



UNIDAD 1

INTRODUCCIÓN A LA PROGRAMACIÓN.

1. INTRODUCCIÓN.
2. LA INFORMACIÓN.
 - 2.1. Unidades de Medida.
 - 2.2. Representar Información.
 - 2.3. Otros sistemas de Numeración.
3. ALGORITMOS.
 - 3.1. Pseudocódigo para Algoritmos.
 - 3.1.1. Instrucciones de Asignación.
 - 3.1.2. Instrucciones de entrada / salida.
 - 3.1.3. Instrucciones de control de flujo.
 - 3.1.4. Subalgoritmos / subprocesos / subprogramas.
4. HARDWARE BÁSICO DE UNA COMPUTADORA.
5. LENGUAJES DE PROGRAMACIÓN.
6. CONSTRUIR PROGRAMAS.
7. PRIMER CONTACTO CON JAVA.
 - 7.1. La Orientación a Objetos.
 - 7.2. Evolución de Java.
8. DESARROLLO Y UML.
9. EJERCICIOS.





1.1. INTRODUCCIÓN.

La RAE define un programa como un "Conjunto de instrucciones que permite a un ordenador realizar funciones diversas, como el tratamiento de textos, el diseño de gráficos, la resolución de problemas matemáticos, el manejo de bancos de datos, etc.".

Para analizar cualquier cosa, se puede adoptar la estrategia de la **caja negra** para estudiarla, es decir, ni sabemos ni nos preocupa como es o como funciona su interior, para nosotros es algo oscuro y desconocido. Para entender lo que es, simplemente observamos como se relaciona con el exterior. Usando este enfoque, podríamos ver a un programa tal y como describe la siguiente figura:



Figura 1: Un programa en ejecución (enfoque de Caja Negra)

Sin saber nada de su interior, cuando el programa se ejecuta, normalmente coge información del exterior (LEE) con ayuda del hardware. En su interior la manipula (la transforma o la mueve de lugar...) y expulsa al exterior (ESCRIBE) más información.

¿De dónde lee información un programa? De un teclado, de un disco duro, de un micrófono, de una tarjeta de red, de un sensor, ...

¿Dónde escribe un programa información? En una pantalla, en un disco duro, en una impresora, en una red, en una tarjeta de sonido, en un servo (actuador), ...



1DAM PROG UNIDAD 1. Introducción a la Programación.

Como la información es la materia prima con la que trabaja un programa de ordenador, vamos a comenzar este tema revisándola. Aprender o repasar que es la información, como se mide y como se representa cualquier cosa con ella. Incluso el propio programa es información.

Luego veremos el concepto de algoritmo, que representa el trabajo que el programa hace con la información con la que trabaja.

Más tarde, examinamos el hardware que participa en la ejecución de los programas.

También comentamos el proceso de generar programas, como los lenguajes de programación han ido evolucionando a lo largo de la historia y por último, haremos una primera toma de contacto con el lenguaje de programación que vamos a usar en el curso: Java.

Pero antes vamos a intentar definir unos primeros términos para que podamos entender sin ambigüedad de lo que hablamos cuando veas los apuntes. En informática se entiende por:

- **Programa:** un conjunto de instrucciones ejecutables por un ordenador o dispositivo similar.
- **Proceso:** cuando un programa se está ejecutando en un dispositivo se le llama proceso, porque tiene un estado en cada momento (la sentencia que está ejecutando, los datos que tiene, la pila, los ficheros que usa, etc.)
- **Aplicación:** Software formado por uno o más programas, la documentación de los mismos y los archivos necesarios para su funcionamiento, de modo que el conjunto completo **forman una herramienta de trabajo en un ordenador**. Normalmente en el lenguaje cotidiano no se distingue entre aplicación y programa.



1.2. LA INFORMACIÓN.

Un ordenador maneja información de todo tipo. Como seres humanos, también nosotros manejamos habitualmente información de muchos tipos: números, texto, imágenes, sonidos, diagramas, códigos, etc. Sin embargo, al tratarse de una máquina digital, un ordenador sólo es capaz de trabajar con **información binaria**. Por este motivo, todos los ordenadores necesitan tener codificada la información del mundo real a algo equivalente como código binario que pueda manipular.

1.2.1. UNIDADES DE MEDIDA DE LA INFORMACIÓN.

La **teoría de la información**, es una propuesta teórica presentada por [Claude E. Shannon](#) y [Warren Weaver](#) a finales de la década de los años 1940. Esta teoría está relacionada con las leyes matemáticas que rigen la transmisión y el procesamiento de la información y se ocupa de la medición y representación, así como también de la capacidad de los sistemas de comunicación para transmitirla y procesarla.

Nota: La teoría de la información es una rama de la [teoría de la probabilidad](#) y de las [ciencias de la computación](#) que estudia la [información](#) y todo lo relacionado con ella: canales, [compresión de datos](#) y [criptografía](#), entre otros.

De ahí provienen las unidades que se usan actualmente para medir la cantidad de información y que ahora describimos:

La unidad mínima de información es el **dígito binario**, abreviado como **bit** que puede representar una de las respuestas a una pregunta que tenga dos posibles respuestas. **No puede haber una información más pequeña.** Se suele decir que un bit puede tener un valor NO/SI, APAGADO/ENCENDIDO, FALSO/VERDADERO ó 0/1.

Como un bit es una cantidad de información muy pequeña, también



1DAM PROG UNIDAD 1. Introducción a la Programación.

usamos cantidades mayores como **el Byte (se pronuncia /bait/)**. Un **byte son 8 bits juntos**. Para representar una letra que usa un idioma, podríamos usar 1Byte (si usamos código ASCII). También lo podemos usar para escribir números enteros positivos desde 0 hasta 255.

Para cantidades mayores de información se usan abreviaturas similares al sistema métrico internacional (Kilo, Mega, Giga, etc.) pero la equivalencia en vez de ir de 1000 en 1000, van de 1024 en 1024, por eso recientemente se han definido como Kilo informático, Mega informático, etc. En esta tabla aparecen algunas unidades (hay más):

UNIDAD	ABREVIATURA	INFORMACIÓN
bit	b	1 bit
Byte	B	8 bits
Kilo i Byte	KiB (antes KB)	1024 Bytes
Mega i Byte	MiB (antes MB)	1024 KiB
Giba i Byte	GiB (antes GB)	1024 MiB
Tera i Byte	TiB (antes TB)	1024 GiB

EJEMPLO 1: ¿Cuántas letras podríamos almacenar en una memoria RAM de 1GiB si cada letra necesita 8 bits?

$$1\text{GiB} = 1024 \text{ MiB} = 1024 * 1024 \text{ KiB} = 1024 * 1024 * 1024 \text{ Bytes} = 1073741824 \text{ Bytes}$$

Como 8 bits = 1 Byte -> y 1 letra es 1 Byte -> pueden almacenarse 1073741824 letras

1.2.2. REPRESENTAR LA INFORMACIÓN EN BINARIO.

1.2.2.1. SISTEMAS DE NUMERACIÓN.

En general, a lo largo de la historia han existido numerosos sistemas de numeración. Cada cultura o civilización ha usado el suyo propio. Para simplificar, dividiremos a todos estos sistemas en dos tipos:



1DAM PROG UNIDAD 1. Introducción a la Programación.

- **Sistemas no posicionales**. utilizan símbolos cuyo valor numérico es siempre el mismo independientemente de su posición en un número. Es lo que ocurre con la numeración romana, donde el símbolo **I** siempre tiene el valor uno independientemente de su posición.: **I** = 1. **II** = 2 (el primer I vale 1 y el segundo tb.)
- **Sistemas posicionales**. los símbolos tienen un mismo valor pero cambian de peso en función de la posición que ocupan. Es el caso de nuestra numeración decimal, donde el símbolo 2, en la cifra 12 vale dos y pesa 1 (aporta 2), mientras que en la cifra 21 vale dos pero pesa 10 (y aporta veinte).

Como los sistemas posicionales facilitan realizar cálculos matemáticos, han sustituido a los no posicionales en todo el mundo.

Todos **los sistemas posicionales tienen una base**, que es **el número total de símbolos (letras) que utiliza para escribir los números**. En el caso de la numeración decimal, la base es 10, porque usamos 10 letras para escribir todos los números: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

Cada letra tiene un valor fijo, pero un peso que cambia con la posición y que aumenta cuanto más a la izquierda aparezca en el número:

Peso	1000	100	10	1
Número	1	2	3	7

El valor representado por un número es el resultado de hacer la suma del valor de cada letra por su peso (subrayado en el ejemplo).

$$1 \times \underline{1000} + 2 \times \underline{100} + 3 \times \underline{10} + 7 \times \underline{1}$$

Los pesos de cada cifra van cambiando en potencias de la base:

Peso	1000	100	10	1
Peso como potencias de 10	10^3	10^2	10^1	10^0



1DAM PROG UNIDAD 1. Introducción a la Programación.

Nota: recuerda que una potencia entera base^{exponente} (base elevado a exponente) es multiplicar la base por sí misma, tantas veces como diga el exponente.

$$4^3 = 4 \cdot 4 \cdot 4 = 64 \quad \text{--> la base es 4 y el exponente es 3}$$

$$2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16 \quad \text{--> la base es 2 y el exponente es 4.}$$

- Cualquier base elevada a 0 es 1: $2^0 = 1$; $7^0 = 1$; $10^0 = 1$
- Una base elevada a exponente negativo es $1/\text{base}^{\text{exponente}}$

El **Teorema Fundamental de la Numeración** permite saber el valor decimal que tiene cualquier número en cualquier base (B). Dicho teorema utiliza esta fórmula:

$$\dots + X_3 \cdot B^3 + X_2 \cdot B^2 + X_1 \cdot B^1 + X_0 \cdot B^0 + X_{-1} \cdot B^{-1} + X_{-2} \cdot B^{-2} + \dots$$

Donde:

- X_i Es el símbolo (letra) que se encuentra en la posición número i del número que se está convirtiendo a decimal. Teniendo en cuenta que la posición de las unidades es la 0 (la posición -1 sería la del primer decimal, la -2 la segunda decimal, etc.).
- **B** Es la base del sistema que se utiliza para representar al número. Por ejemplo, si tenemos el número 153,6 utilizando el sistema octal (base 8), el paso a su valor decimal se haría:

$$1 \cdot 8^2 + 5 \cdot 8^1 + 3 \cdot 8^0 + 6 \cdot 8^{-1} = 64 + 40 + 3 + 6/8 = 107,75$$

Mientras que si fuese un número en decimal (base 10) el valor sería el que estamos acostumbrados:

$$1 \cdot 10^2 + 5 \cdot 10^1 + 3 \cdot 10^0 + 6 \cdot 10^{-1} = 100 + 50 + 3 + 6/10 = 153,6$$

SISTEMA BINARIO DE NUMERACIÓN

Los ordenadores al ser máquinas digitales, solo pueden trabajar con información binaria. Por tanto, para trabajar con números necesitamos



expresarlos en base 2 (binario). Usamos un alfabeto con los símbolos {0, 1} y por tanto la base es 2.

1.2.2.2. NÚMEROS ENTEROS (SIN DECIMALES) Y POSITIVOS.

CONVERTIR DE BINARIO A DECIMAL.

El teorema fundamental de la numeración se puede aplicar para saber el valor decimal representado por un número escrito en binario. Así para el número binario 11011011_2 la conversión se haría:

$$1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 219$$

Simplemente hay que sumar las potencias de los dígitos 1 sin hacer caso de los 0. Se puede convertir asignando un peso a cada dígito. De modo que la primera cifra (la que está más a la derecha) se corresponde con la potencia 2^0 o sea 1, la siguiente 2^1 (2), la siguiente 2^2 que es cuatro y así sucesivamente el doble de la anterior, de esta forma, por ejemplo para el número 10100101_2 , se haría:

Pesos:	128	64	32	16	8	4	2	1
	1	0	1	0	0	1	0	1

se suman los pesos que tienen un 1: $128 + 32 + 4 + 1 = 165$

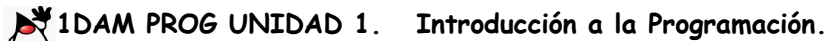
Cuando hay varios bits juntos (1 Byte, 2 Bytes, etc.) Se suele hablar de los bit más significativos o menos significativos.

- **Bits más significativos:** los que tienen un peso mayor. Es decir, los que más a la izquierda están.
- **Bits menos significativos:** Aquellos que tienen un menor peso, los que más a la derecha están.

EJERCICIO 1: Convierte a decimal estos números binarios:

a) 10101_2 b) 100111_2 c) 111010_2 d) 11011101_2

EJERCICIO 2: Escribe los siguientes números en binario:



- • •



decimal.

Otra posibilidad más rápida es mediante restas sucesivas. En este caso se colocan los pesos (todas las potencias de 2) hasta sobrepasar el número decimal a convertir. El primer peso que sea menor al número decimal a convertir será un 1 en el número binario. Se resta el peso al número. Se ponen a 0 todas los pesos mayores. Se vuelve a aplicar el método sucesivamente.

EJEMPLO 2: convertir a binario el número decimal 179

1) Obtener potencias de 2 hasta sobrepasar el número:

256 128 64 32 16 8 4 2 1

2) Proceder a las restas (observar de abajo a arriba):

Potencias	Pesos	Restas
1	1	$1-1=0$
2	1	$3-2=1$
4	0	
8	0	
16	1	$19-16=3$
32	1	$51-32=19$
64	0	
128	1	$179-128=51$
256	0	

El número binario es el 10110011_2 , porque los ceros a la izquierda no aportan nada.

1.2.2.3. NÚMEROS ENTEROS (SIN DECIMALES) NEGATIVOS.

Si se representan números enteros tanto positivos como negativos, entonces se suelen usar dos métodos. El primero consiste en dedicar un bit para indicar el signo del número. Un 0 indica positivo y un 1



1DAM PROG UNIDAD 1. Introducción a la Programación.

indica negativo. En el caso de usar un byte para almacenar el número, el bit más significativo indica el signo (0 positivo y 1 negativo) y los otros 7 bits son el valor absoluto. Así:

01110101₂ es el número 117

11110101₂ es el número -117

El rango representable con 1 byte sería del -127 al 127: desde 1111111₂ hasta 0111111₂. El inconveniente de este formato es que el cero tiene dos formas de escribirse: -0 y +0 (10000000 y 00000000)

Otra posibilidad es la **representación en complemento a uno**. En esta representación, a los números negativos se les invierten todos los bits.

EJEMPLO 3: Representa -117 en complemento a 1 con 8 bits:

117 es 01110101₂ → complemento a 1 es cambiar 0 por 1 y viceversa

10001010₂ es el número -117 en complemento a uno

Pero no soluciona el problema de que hay dos representaciones para el número cero (la 00000000 y la 11111111). Por ese motivo y porque la resta de dos números se puede hacer sumando a uno el complemento a dos del otro, la representación que suele utilizarse para los números negativos es el **complemento a dos**. Esta representación consiste en sumar 1 al complemento a uno. De este modo se añade al rango de valores el número -128 que se representa como 10000000 y el cero ya no tiene dos formas de escribirse.

EJEMPLO 4: Reperesentar -117 en complemento a 2 con 8 bits.

117 es 01110101₂ → complemento a 1 es cambiar 0 por 1 y viceversa

10001010₂ es el número -117 en complemento a uno

El complemento a 2 es sumar 1 al complemento a 1:

10001011_2 es el número -117 en complemento a dos

EJERCICIO 4: Escribe estos números negativos con 8 bits representándolos en formato bit de signo, complemento a 1 y complemento a 2.

a) $1 \cdot y^{-1}$ b) $2 \cdot y^{-2}$ c) $3 \cdot y^{-3}$ d) $8 \cdot y^{-8}$ e) 0

1.2.2.4. OPERACIONES CON NÚMEROS ENTEROS BINARIOS.

Vamos a ver dos operaciones aritméticas (suma y resta) y varias operaciones lógicas (AND, OR y NOT). Aunque hay más, estas nos sirven para comprobar que trabajar en binario (como hacen los circuitos de los ordenadores) es perfectamente posible.

SUMA ARITMÉTICA

La suma se efectúa igual que con números decimales, sólo que usando cifras binarias. Cuando en decimal una cifra y otra alcanzan o superan al valor de la base, decimos que nos llevamos una (en informática se llama **acarreo**). Ejemplos en decimal:

$$0 + 0 = 0 \quad 1 + 0 = 1 \quad \dots$$

:

•

$0 + 9 = 9$ $1 + 9 = 0$ y me llevo una (el acarreo)

Las sumas de posibles valores en binario se recoge en esta tabla:

Suma	Resultado
0 + 0	0
0 + 1	1
1 + 0	1
1 + 1	0 (más un acarreo de 1)

Ejemplo:

$$\begin{array}{cccccccc}
 & & & & 1 & 1 & 1 & \\
 + & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
 \hline
 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1
 \end{array}$$



EJERCICIO 5: Convierte a binario y realiza la suma en binario.

- a) $7 + 5$ b) $6 + 9$ c) $20 + 4$ d) $14 + 16$

RESTA ARITMÉTICA

Funciona como la resta normal en decimal, por ejemplo $3 - 2$ es 1. Y cuando el número que resta es mayor que al que le quitas, pides prestado uno a la posición siguiente, ejemplo: $11 - 3$:

$$\begin{array}{r} 21 \\ -13 \\ \hline 8 \end{array}$$

Dices: 1 menos 3 no cabe (el 1 es más pequeño que 3)

Así que pides 1 prestado a la siguiente posición y haces:
 $11 - 3 = 8$

Te llevas 1 a lo que resta de la siguiente posición (para devolver lo que usaste prestado) y queda $2 - 1 - 1 = 0$

La diferencia de la resta con la suma es que los acarreos se usan en el segundo número, en el que resta, mientras que en la suma no importa donde lo pongas. La siguiente tabla resume la resta binaria:

Resta	Resultado
0 - 0	0
0 - 1	1 (más un acarreo de 1)
1 - 0	1
1 - 1	0

Ejemplo:

$$\begin{array}{r} 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0 \\ - 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1 \\ \hline 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1 \end{array} \quad \text{acarreos}$$

EJERCICIO 6: Convierta a binario y realice su resta.

- a) $8 - 4$ b) $15 - 8$ c) $22 - 3$ d) $64 - 16$

Otra posibilidad es pasar el valor que resta a complemento a dos y hacer la suma, descartando el último acarreo. Así lo hacen todas las CPU's para ahorrarse los circuitos de la resta.

EJERCICIO 7: Repite el ejercicio 6 usando suma en complemento a 2.

OPERACIÓN LÓGICA AND

Los bits de dos valores se mezclan bit a bit interpretando cada bit como un valor de verdad (verdadero si es 1 y falso si es 0). La



1DAM PROG UNIDAD 1. Introducción a la Programación.

operación AND de dos bits es 1 solamente si los dos bits son 1. Es como la multiplicación bit a bit ($0 \times 1 = 0$; $1 \times 1 = 1$). La siguiente tabla contempla todas las posibilidades:

AND	Resultado
0 AND 0	0
0 AND 1	0
1 AND 0	0
1 AND 1	1

OPERACIÓN LÓGICA OR

El resultado es 1 si al menos uno de los dos bits es 1, y es 0 solamente si los dos bits son 0. La siguiente tabla resume todas las posibilidades:

OR	Resultado
0 OR 0	0
0 OR 1	1
1 OR 0	1
1 OR 1	1

OPERACIÓN LÓGICA NOT

Se aplica a un solo valor, y consiste en cambiar ceros por unos y viceversa, es la negación bit a bit.

NOT	Resultado
NOT 0	1
NOT 1	0

EJERCICIO 8: Convierta a binario de 1 Byte y realiza operaciones.

a) 18 OR 16

b) 192 AND 64

c) NOT 3

1.2.2.5. NÚMEROS CON DECIMALES.

Los números que tienen decimales se suelen representar de dos formas: **punto fijo** y **punto flotante** (o **coma fija/coma flotante**).



NOTACIÓN EN PUNTO FIJO

Una cantidad fija de los bits se dedican a la parte entera del número y otra cantidad se dedican a la parte decimal. Por ejemplo en 1 byte decidimos que los 4 más significativos son para la parte entera y los 4 menos significativos son para la parte decimal. Los pesos serían:

Nota: recuerda que $2^{-1} = 1/2 = 0.5$ y $2^{-2} = 1/(2 \times 2) = 1/4 = 0.25$, etc.

	Bits de la parte entera				Bits de la parte decimal			
Pesos:	8	4	2	1	0.5	0.25	0.125	0.06125
cifras	1	0	0	1	1	0	0	1

El valor decimal de este número sería: 9.56125

La representación en punto fijo tiene el inconveniente de definir un rango de posibles valores muy limitado y tiene problemas para trabajar con números muy grandes o muy pequeños. Pero tiene la ventaja de que un error de aproximación al valor exacto en una operación, no se encadena y aumenta tanto en las siguientes operaciones.

NOTACIÓN EN PUNTO FLOTANTE

El estándar del [IEEE](#) para aritmética en coma flotante (IEEE 754) es una [norma o estándar](#) para representar valores en [coma flotante](#), establecida en 1985. La idea de este formato es descomponer los números en 3 partes:

- Un bit de signo: 0 positivo y 1 negativo.
- Una mantisa que contiene los dígitos del número. En simple precisión son 24 bits, 23 que se almacenan explícitamente y 1 bit es implícito. La mantisa representa siempre un valor fraccionario.
- Un exponente que indica dónde se coloca el punto decimal en relación al inicio de la mantisa. Su bit más significativo indica signo

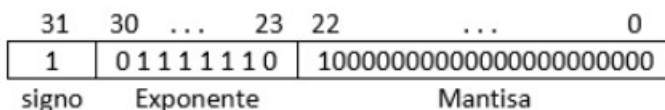


del exponente. Cuando el exponente es negativo representa a un número menor que uno. Suele estar desplazado a 127.

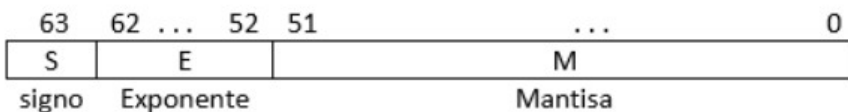
El valor decimal almacenado puede calcularse así en simple precisión:

$$v = (-1)^{b_{31}} \times (1, b_{22}b_{21} \dots b_0)_2 \times 2^{e-127}$$

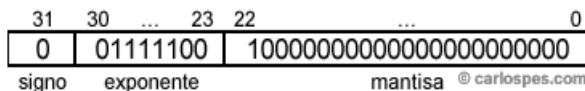
IEEE754 de 32 bits (simple precisión):



IEEE744 de 64 bits (doble precisión):



EJEMPLO 5: Averiguar el valor del siguiente número decimal expresado en IEEE 754 de 32 bits con exponente en exceso de 2^{n-1}



El exponente y la mantisa son positivos (bit de signo a 0). Pasamos el exponente a base 10:

$$01111100_2 - 127_{10} = 124_{10} - 127_{10} = -3$$

Para escribir el número en notación científica, la mantisa se debe escribir con un bit implícito (1), seguido de la coma decimal (.) y de los bits de la mantisa (100000000000000000000000), teniendo en cuenta que los



1DAM PROG UNIDAD 1. Introducción a la Programación.

ceros por la derecha se pueden despreciar (porque son la parte decimal). Por tanto, el número es:

$$1,1 \times 2^{-3}$$

Para expresar el número en base 10, hay dos formas. La primera:

$$1,1 \times 2^{-3} = 0,0011_2 = (2^{-3} + 2^{-4})_{10} = 0,125_{10} + 0,0625_{10} = 0,1875_{10}$$

y la segunda:

$$1,1 \times 2^{-3} = ((2^0 + 2^{-1}) \times 2^{-3})_{10} = ((1 + 0,5) \times 0,125)_{10} = (1,5 \times 0,125)_{10} = 0,1875_{10}$$

Por tanto,

$$3E40000_{\text{CFL (PRECISIÓN SIMPLE)}} = 1,1 \times 2^{-3} = 0,0011_2 = 0,1875_{10}$$

Nota: Los programadores para representar los números reales en estos formatos, usan el sistema hexadecimal. Afortunadamente, normalmente trabajamos a más alto nivel y no tendremos que manipular estos formatos directamente.

Nota: sin embargo estas representaciones pueden influir en nuestros programas de varias formas:

- operaciones sencillas pueden generar errores de aproximación y hacer hasta fallar nuestros programas si no lo tenemos en cuenta.
- Si trabajamos con cantidades de dinero, la aparición de estos errores puede ser inaceptable y tendremos que buscar formas alternativas de hacer cálculos que los eviten o minimicen.

En general la representación en coma flotante permite representar números muy pequeños y muy grandes sin consumir una gran cantidad de memoria, pagando a cambio el precio de perder precisión y



cometiendo errores de representación y aproximación.

1.2.2.6. REPRESENTACIÓN DE TEXTO.

Para representar las letras de un texto se usan códigos: Se asocia una combinación de bits diferente a cada posible letra.

ASCII (American Standard Code International Interchange)

Inicialmente era un código que utilizaba 7 bits para representar cada letra, lo que significa que era capaz de codificar 128 caracteres. Por ejemplo el número 65 (1000001 en binario) se utiliza para la A mayúscula. Poco después apareció un problema: este código es suficiente para los caracteres del inglés pero no para otros idiomas.

Entonces se añadió el octavo bit para representar otros 128 caracteres que son distintos según idiomas (Europa Occidental usa unos códigos que no utiliza Europa Oriental). Eso provoca que un código como el 190 signifique cosas diferentes si cambiamos de país. Por ello cuando un ordenador necesita mostrar texto, tiene que saber qué juego de códigos debe de utilizar (tremendo problema). Cada letra siempre necesita 1 Byte de información.

EJERCICIO 9: Si escribes tu primer apellido en ASCII

- a) ¿Cómo se vería en binario?
- b) ¿Cuántos bytes necesita?
- c) ¿Cuántas veces puedes escribirlo en una memoria de 1MiB.

UNICODE

Puede utilizar hasta 4 bytes (32 bits) por letra, eso le permite codificar cualquier carácter de cualquier lengua del planeta utilizando el mismo conjunto de códigos. Poco a poco es el código que se va extendiendo, pero la preponderancia histórica que ha tenido el código ASCII complica su popularidad.



CÓDIGO ASCII

Caracteres ASCII de control			Caracteres ASCII imprimibles			ASCII extendido										
00	NULL	(carácter nulo)	32	espacio	64	@	96	.	128	Ç	160	á	192	Ł	224	Ó
01	SOH	(inicio encabezado)	33	!	65	A	97	a	129	ü	161	í	193	ł	225	ô
02	STX	(inicio texto)	34	"	66	B	98	b	130	é	162	ó	194	Ł	226	Ô
03	ETX	(fin de texto)	35	#	67	C	99	c	131	â	163	ú	195	ł	227	Ò
04	EOT	(fin transmisión)	36	\$	68	D	100	d	132	ä	164	ñ	196	—	228	õ
05	ENQ	(consulta)	37	%	69	E	101	e	133	à	165	Ñ	197	†	229	Ö
06	ACK	(reconocimiento)	38	&	70	F	102	f	134	á	166	ª	198	ä	230	µ
07	BEL	(timbre)	39	'	71	G	103	g	135	ç	167	º	199	Ä	231	þ
08	BS	(retroceso)	40	(72	H	104	h	136	ê	168	¿	200	Ł	232	ƒ
09	HT	(tab horizontal)	41)	73	I	105	i	137	ë	169	©	201	ł	233	ú
10	LF	(nueva línea)	42	*	74	J	106	j	138	è	170	¬	202	Ł	234	Û
11	VT	(tab vertical)	43	+	75	K	107	k	139	ï	171	½	203	ł	235	Ü
12	FF	(nueva página)	44	,	76	L	108	l	140	î	172	¾	204	ł	236	ý
13	CR	(retorno de carro)	45	-	77	M	109	m	141	ï	173	¿	205	=	237	Ý
14	SO	(desplaza afuera)	46	.	78	N	110	n	142	Ā	174	«	206	≠	238	—
15	SI	(desplaza adentro)	47	/	79	O	111	o	143	Ā	175	»	207	≠	239	'
16	DLE	(esc.vínculo datos)	48	0	80	P	112	p	144	É	176	€	208	ð	240	≡
17	DC1	(control disp. 1)	49	1	81	Q	113	q	145	æ	177	€	209	Ð	241	±
18	DC2	(control disp. 2)	50	2	82	R	114	r	146	Æ	178	€	210	Ê	242	—
19	DC3	(control disp. 3)	51	3	83	S	115	s	147	ô	179	€	211	Ë	243	¾
20	DC4	(control disp. 4)	52	4	84	T	116	t	148	ö	180	€	212	Ë	244	ŋ
21	NAK	(conf. negativa)	53	5	85	U	117	u	149	ò	181	À	213	Ì	245	§
22	SYN	(inactividad sinc)	54	6	86	V	118	v	150	û	182	Ā	214	Í	246	÷
23	ETB	(fin bloque trans)	55	7	87	W	119	w	151	ü	183	Ā	215	Î	247	°
24	CAN	(cancelar)	56	8	88	X	120	x	152	ÿ	184	©	216	Ï	248	°
25	EM	(fin del medio)	57	9	89	Y	121	y	153	Ō	185	€	217	Ĵ	249	"
26	SUB	(sustitución)	58	:	90	Z	122	z	154	Ū	186	€	218	Ŕ	250	•
27	ESC	(escape)	59	;	91	[123	{	155	ø	187	€	219	Ŗ	251	ˆ
28	FS	(sep. archivos)	60	<	92	\	124		156	£	188	€	220	Ÿ	252	˜
29	GS	(sep. grupos)	61	=	93]	125	}	157	Ø	189	€	221	ˆ	253	ˆ
30	RS	(sep. registros)	62	>	94	^	126	~	158	×	190	¥	222	ˆ	254	■
31	US	(sep. unidades)	63	?	95	_			159	f	191	γ	223	■	255	nbsp
127	DEL	(suprimir)														

Figura 2: Tabla de códigos ASCII.

Unicode incluye todos los caracteres de uso común en la actualidad. La versión 11.0 contiene 137374 caracteres de alfabetos, sistemas



1DAM PROG UNIDAD 1. Introducción a la Programación.

ideográficos y colecciones de símbolos (matemáticos, técnicos, musicales, iconos...). La cifra crece con cada versión.

Unicode incluye sistemas de escritura modernos como: árabe, braille, copto, cirílico, griego, [sinogramas](#) ([hanja](#) coreano, [hanzi](#) chino y [kanji](#) japonés), [silabarios japoneses](#) ([hiragana](#) y katakana), [hebreo](#) y [latino](#), escrituras históricas extintas, para propósitos académicos, como por ejemplo: cuneiforme, griego antiguo, [lineal B](#) micénico, [fenicio](#) y [rúnico](#).

Entre los caracteres no alfabéticos incluidos en Unicode se encuentran símbolos musicales y matemáticos, fichas de juegos como el dominó, flechas, iconos, etc.

$$\begin{array}{ccc} \tilde{n} & \equiv & n + \tilde{\circ} \\ \text{U+00F1} & & \text{U+006E} \quad \text{U+0303} \end{array}$$

Figura 3: Ejemplo de carácter UNICODE.

Además, Unicode incluye los [signos diacríticos](#) como caracteres independientes que pueden ser combinados con otros caracteres y dispone de versiones predefinidas de la mayoría de letras con símbolos diacríticos en uso en la actualidad, como las vocales acentuadas del español. Es un estándar en constante evolución y se agregan nuevos caracteres continuamente. Se han descartado ciertos alfabetos, propuestos por distintas razones, como por ejemplo el alfabeto [klingon](#).

UTF-8 (8-bit Unicode Transformation Format)

Es un formato de codificación de caracteres [Unicode](#) e [ISO 10646](#) utilizando símbolos de longitud variable. UTF-8 fue creado por [Robert C. Pike](#) y Kenneth L. Thompson. Está definido como estándar por la [RFC 3629](#) de la [Internet Engineering Task Force](#) (IETF). Actualmente es una de las tres posibilidades de codificación reconocidas por Unicode y



1DAM PROG UNIDAD 1. Introducción a la Programación.

lenguajes web en ISO 10646. Sus características principales:

- Es capaz de representar cualquier carácter Unicode.
- Usa símbolos de longitud variable (de 1 a 4 bytes por carácter).
- Incluye la especificación [US-ASCII](#) de 7 bits, por lo que cualquier mensaje ASCII se representa sin cambios.
- Incluye sincronía. Es posible determinar el inicio de cada símbolo sin reiniciar la lectura desde el principio de la comunicación.
- No superposición. Los conjuntos de valores que puede tomar cada byte de un carácter multibyte, son disjuntos, por lo que no es posible confundirlos entre sí.

Estas características lo hacen atractivo en la codificación de correos electrónicos y páginas web. El [IETF](#) requiere que todos los protocolos de [Internet](#) indiquen qué [codificación](#) utilizan para los textos y que UTF-8 sea una de las codificaciones contempladas. El [Internet Mail Consortium](#) (IMC) recomienda que todos los programas de correo electrónico sean capaces de crear y mostrar mensajes codificados utilizando UTF-8.

ñ (Unicode) = U+00F1

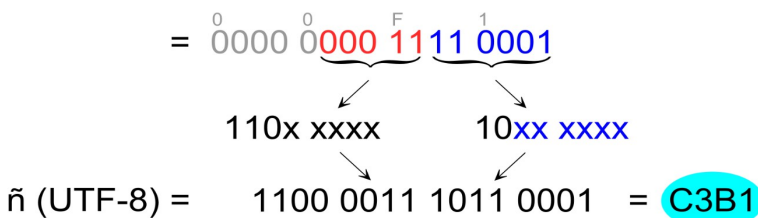


Figura 4: Ejemplo de codificación en UTF-8.

UTF-16

Significa en [ISO/IEC 10646:2003](#) "UCS Transformation Format for 16



1DAM PROG UNIDAD 1. Introducción a la Programación.

Planes of Group 00", es una forma de codificación de caracteres UCS y Unicode utilizando símbolos de longitud variable. Sus características:

- Es capaz de representar cualquier carácter Unicode.
- Utiliza símbolos de longitud variable: 1 o 2 palabras de 16 bits por carácter Unicode (2 o 4 bytes). La unidad de información es la palabra de 16 bits.
- Está optimizado para representar caracteres del plano básico multilingüe (BMP) y caracteres del rango U+0000 a U+FFFF. El BMP contiene la gran mayoría de caracteres y sistemas de escritura en uso en la actualidad. Cuando se limita al plano básico multilingüe, UTF-16 puede ser considerado una forma de codificación con símbolos de tamaño fijo (16 bits).
- No superposición: Los símbolos de 1 palabra (16 bits) utilizan un subconjunto de valores que no puede utilizarse en símbolos de 2 palabras (32 bits).

EJEMPLO 6: Codificación del carácter Unicode U+1D11E, Clave de sol. El carácter está fuera del plano básico (BMP) y por tanto requiere el uso de pares subrogados.



(Unicode) = U+1D11E

1D11E - 10000 = 0D11E

= $\overset{0}{0000} \overset{D}{1101} \overset{0}{0001} \overset{1}{0001} \overset{E}{1110}$

110110 yyyyyyyyyy

110111 xxxxxxxxxx



(UTF-16) = 1101100000110100 1101110100011110 = **D834, DD1E**

Figura 5: Ejemplo de codificación en UTF-16.



1.2.2.7 DATOS MULTIMEDIA: IMÁGENES, AUDIO...

Estos datos son más complejos y necesitan una codificación también más compleja. Además hay más estándares, decenas de formas diferentes de codificar, cada una con sus ventajas e inconvenientes.

IMÁGENES

En el caso, de las imágenes, hay dos grandes tipos: **mapas de bits** y **vectoriales**. Los mapas de bits codifican en binario el color de cada **píxel** (cada punto distinguible en la imagen). Cuantos más bits se usen en cada píxel más colores diferentes puede tener la imagen y más realista será. Con 3 bytes para cada píxel (el primero graba el nivel de rojo, el segundo el nivel de azul y el tercero el nivel de verde) se obtiene true color. Y así por cada píxel. Por ejemplo un punto en una imagen de color rojo puro sería:

11111111 00000000 00000000

Naturalmente en un fichero imagen no solo se graban los píxeles sino el tamaño de la imagen, el modelo de color (aparte de RGB hay otros), quizás varios bits por píxel que indiquen el nivel de transparencia de cada uno, o una paleta de colores y muchas otras cosas...

La imágenes de mapa de bits cuando se amplían o reducen pierden calidad (sobre todo si se amplían), se dice que **se pixelan**, porque cambia la resolución y por tanto el tamaño de los píxels.

Este problema no lo padecen las imágenes vectoriales, donde no se guarda información de cada punto de la imagen, si no la fórmula que generan las figuras de la imagen, por tanto el tamaño no importa. Se usan en diseño gráfico sobre todo.



PROGRAMAS

Son información y por tanto, igual que cualquier otra cosa, se representan mediante ceros y unos. Hablaremos más de ellas después de ver el hardware que ejecuta los programas.

SONIDO

Al igual que las imágenes, hay que digitalizarlo. Y esto genera una gran cantidad de bits. Se suele usar una frecuencia de muestreo (se toma una medida del volumen del sonido cierto número de veces por segundo (Hz) y cada medida se anota en binario y representa la fuerza de la onda sonora.

A mayor frecuencia de muestreo (Hz) y mayor cantidad de diferentes valores que puedan anotarse en cada medición (bits de la muestra), mejor calidad tendrá el sonido y más bits necesitará. Igual que para las imágenes, hay cientos de formatos diferentes y son complejos.

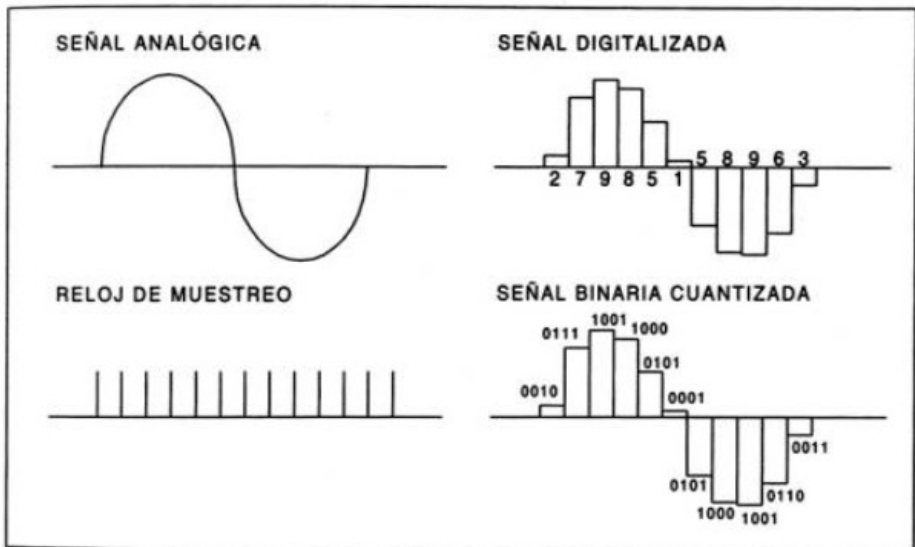


Figura 6: Digitalizar onda sonora analógica.



VÍDEO

El vídeo es una mezcla de muchas imágenes cada segundo (frames) y sonido. Todos los datos multimedia generan mucha información, por eso los formatos de almacenar esta información casi siempre deben recurrir a trucos para reducir esa enorme cantidad de bits (métodos de compresión, etc.). Aquí se distinguen formatos contenedores como JPEG, AVI ... (como se coloca cada cosa) y codificaciones concretas (de audio y vídeo) ¿Te suenan los codecs?

1.2.3. OTROS SISTEMAS DE NUMERACIÓN.

Aparte del sistema binario, hay otros sistemas de numeración útiles que se utilizan a menudo en informática (por administradores, programadores, algunos protocolos de red, etc.). Ejemplos:

OCTAL (base 8)

Las cifras de un número expresado en octal van desde 0 a 7. Así que el número 8 en decimal, se escribe 10 en octal. Lo interesante de este sistema es que tiene una traducción directa y rápida con el sistema binario, ya que una cifra octal se corresponde exactamente con 3 cifras binarias (3 bits). La razón es que el número 8 es 2 elevado a 3. De ahí que 3 bits binarios sean una cifra octal.

EJEMPLO 7: escribe en octal el número binario 1100011.

Se agrupan los bits de 3 en 3 (empezando por la derecha), es decir 1 100 011 y cada grupo se pasa a octal: 143.

EJEMPLO 8: Pasa a binario el número octal 217.

Cada cifra octal se escribe con 3 bits: 010 001 111

EJERCICIO 10: Escribe tu primer apellido en ASCII en octal.

HEXADECIMAL (BASE 16)

Tiene la ventaja de que una cifra hexadecimal representa exactamente



1DAM PROG UNIDAD 1. Introducción a la Programación.

4 bits (lo que se conoce como un **nibble**). De este modo 2 cifras hexadecimales representan un byte (8 bits). Las letras que se utilizan para escribir los 16 valores son {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F} donde A vale 10, B vale 11, C vale 12, D vale 13, E 14 y F es 15.

EJEMPLO 9: Pasar a hexadecimal el número binario **10110011**.

Agrupas cada 4 bits desde la derecha y cambias cada grupo por su valor: 1011 0011 que es **B3** en hexadecimal.

EJEMPLO 10: Pasa a binario el siguiente número hexadecimal **CA**.

C = 12 en decimal = 1100 en binario

A = 10 en decimal = 1010 en binario

CA = 1100 1010

EJERCICIO 11: transforme su apellido escrito en ASCII de binario a hexadecimal.

BASE 64.

Se usa como un sistema de codificar binario en texto. Utiliza 64 letras, por tanto, cada letra equivale a 6 bits. Por este motivo 4 letras son 3 bytes de información. Lo usan muchos protocolos de intercambio de información.

Usa las letras mayúsculas de la A-Z (valores de 0 a 25), las letras minúsculas a-z (valores 26 a 51), números (0-9) como valores de 52 a 61 y suma (+) y barra (/) para valores 62 y 63. Si faltan bits para cuadrar la correspondencia con los bytes que debe ocupar una información en un mensaje, se usa el símbolo = como relleno: ninguna vez (si cuadra), una vez (si faltan 2 bits) o dos veces (si faltan 4 bits) para alcanzar así un número exacto de bytes.

EJERCICIO 12: Escriba en Base64 la información que se guarda en un ordenador si es:



a) 00000001

b) 00000001 00000010

1.3. ALGORITMOS.

Según la RAE un algoritmo es: un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

Los algoritmos, como indica su definición oficial, son una serie de pasos que permiten obtener la solución a un problema. La palabra algoritmo procede del matemático Árabe **Mohamed Ibn Al Kow Rizmi**, el cual escribió alrededor de los años 800 y 825 su obra *Quitad Al Mugabala*, donde entre otras cosas se recogía el sistema de numeración hindú (el decimal que usamos hoy, antes era el romano), el concepto del cero y las x de las ecuaciones algebraicas también provienen de esta obra.

Fibonacci, tradujo la obra al latín y la llamó: *Algoritmi Dicit*. El lenguaje algorítmico es aquel que implementa una solución teórica a un problema, indicando las operaciones a realizar para resolverlo.

Por ejemplo en el caso de que nos encontremos en casa con el problema de una bombilla fundida en una lámpara, un posible algoritmo sería:

- (1) Comprobar si hay bombillas de repuesto.
- (2) Si hay, entonces sustituir la bombilla anterior por la nueva
- (3) Si no, bajar a comprar una nueva a una tienda y volver a (1)

Los algoritmos son la base de la programación de ordenadores, ya que los programas de ordenador pueden entenderse como algoritmos escritos en un lenguaje especial entendible por un ordenador. Lo malo del diseño de algoritmos está en que no podemos escribir lo que deseamos, el lenguaje que se utilice no debe dejar posibilidad de duda, y debe recoger todas las posibilidades. Además, el nivel de detalle en el que se describe la solución de un problema puede variar. Los tres pasos anteriores del algoritmo para cambiar la bombilla pueden ser



1DAM PROG UNIDAD 1. Introducción a la Programación.

mucho más largos:

Comprobar si hay bombillas de repuesto

[1.1] Abrir el cajón de las bombillas

[1.2] Observar si hay bombillas

Si hay bombillas:

[1.3] Coger la bombilla

[1.4] Coger una silla

[1.5] Subirse a la silla

[1.6] Quitar la actual y poner la nueva bombilla en la lámpara

Si no hay bombillas

[1.7] Abrir la puerta

[1.8] Bajar las escaleras....

En el caso de los algoritmos pensados para ejecutarlos en el ordenador, se pueden utilizar sólo ciertas instrucciones muy concretas.

Nota: a partir de ahora usamos la palabra **algoritmo** o **programa** indistintamente.

CARACTERÍSTICAS OBLIGATORIAS DE UN ALGORITMO.

- **Debe resolver el problema para el que fue formulado.** No sirve un algoritmo que no resuelve su problema.
- **Ser independiente del ordenador.** Los algoritmos se escriben para poder ser utilizados en cualquier máquina.
- **Ser preciso.** Los resultados que genera deben tener la exactitud que se necesite.
- **Ser finitos.** Deben de finalizar en algún momento. No es válido si en algunas situaciones no termina nunca.
- **Ser repetibles.** Deben poder ejecutarse una y otra vez..

CARACTERÍSTICAS DESEABLES DE UN ALGORITMO.

- **Validez.** Un algoritmo es válido si carece de errores.
- **Eficiencia.** Un algoritmo es eficiente si obtiene la solución al problema en poco tiempo (realizando pocas instrucciones). No lo es, si es lento o consume excesiva memoria para obtener el



resultado.

- **Óptimo.** Un algoritmo es óptimo si es el más eficiente posible y no contiene errores. La búsqueda de este algoritmo es el objetivo prioritario del programador. No siempre podemos garantizar que el algoritmo que usamos sea el óptimo.

ELEMENTOS QUE TIENE UN ALGORITMO.

- **Entrada.** datos iniciales que tiene/lee antes de ejecutarse.
- **Proceso.** Acciones o instrucciones que lleva a cabo el algoritmo.
- **Salida.** Datos que obtiene finalmente el algoritmo.

CÓMO SE REPRESENTAN LOS ALGORITMOS.

Un algoritmo se puede expresar con un lenguaje **formal** (como el matemático) o en **pseudocódigo**: con más o menos detalle usando un lenguaje con unas instrucciones o sentencias muy reducidas para evitar ambigüedades. Ejemplos:

Algoritmos en pseudocódigo para calcular la raíz cuadrada de un entero x.	
CON SALTOS INCONDICIONALES(amarillo)	PROGRAMACIÓN ESTRUCTURADA (sin saltos incondicionales)
PASO 1: Leer x PASO 2: $b \leftarrow x$ PASO 3: SI b no es igual a x/b ENTONCES PASO 3.1 $b \leftarrow (x/b + b) / 2$ PASO 3.2 SALTA AL PASO 3 FIN SI PASO 4: Escribir raíz cuadrada = b	PASO 1: Leer x PASO 2: $b \leftarrow x$ PASO 3: MIENTRAS b distinto de x/b REPITE $b \leftarrow (x/b + b) / 2$ FIN REPITE PASO 4: Escribir raíz cuadrada = b

También se pueden dibujar mediante unos símbolos predefinidos por la ISO que crean unos diagramas llamados **diagramas de flujo**.

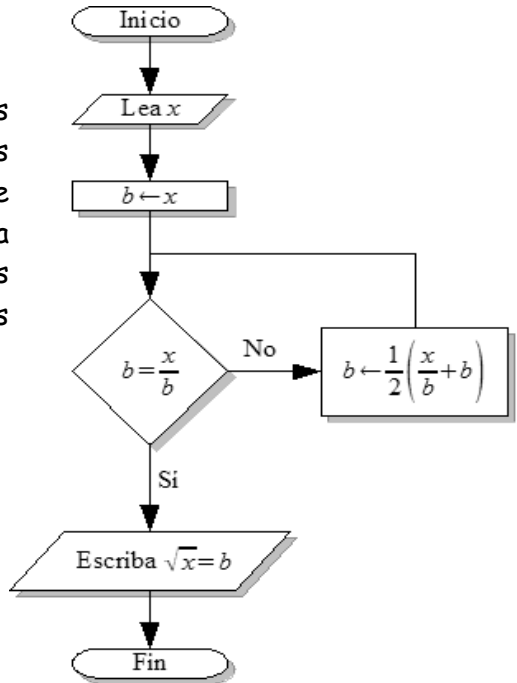
Vemos el diagrama de flujo del algoritmo anterior en la siguiente figura.



SÍMBOLOS DE DIAGRAMAS DE FLUJO:

En los diagramas de flujo los pasos aparecen como diferentes figuras unidas por flechas que indican el siguiente paso a ejecutar. Como hay diferentes tipos de pasos, se usan estos símbolos:

- **Óvalo:** indica el principio o un final del algoritmo.
- **Paralelograma:** indica una operación de lectura (entrada de datos) o escritura (salida de datos).
- **Rectángulo:** indican procesamiento (operaciones).
- **Rombo:** indica la comprobación de una condición que puede ser cierta o falsa. Si la condición es cierta se continúa ejecutando unas sentencias y si es falsa se continúa ejecutando otras sentencias diferentes.



Inicio/fin

Lectura/escritura

operación

selección

subprograma

Fichero

Fichero

Los diagramas de flujo son útiles para visualizar el comportamiento de algoritmos con pocas sentencias. Cuando un algoritmo tiene muchas instrucciones, los diagramas son difíciles de entender.



TÉCNICAS DE DISEÑO DE ALGORITMOS:

- **Algoritmos determinísticos (comportamiento lineal):** si lo ejecutas varias veces con la misma entrada, devuelve el mismo resultado. Normalmente son los que se utilizan.
- **Algoritmos voraces (greedy):** seleccionan los elementos más prometedores del conjunto de posibles soluciones, hasta encontrar una solución aceptable. En la mayoría de los casos, la solución no es la óptima (la mejor).
- **Algoritmos paralelos:** dividen un problema en subproblemas que se solucionan a la vez (en paralelo).
- **Algoritmos probabilísticos:** hay pasos que usan valores pseudoaleatorios (al azar). Dos ejecuciones con la misma entrada podrían generar salidas diferentes.
- **Algoritmos no determinísticos:** en alguno de sus pasos no se puede predecir el paso que se ejecutará a continuación (no lineales).
- **Divide y vencerás:** dividen el problema en subconjuntos disjuntos obteniendo una solución parcial de cada uno de ellos para después unirlos, logrando así la solución al problema completo.
- **Metaheurísticas:** encuentran soluciones aproximadas (no óptimas) de problemas basándose en un conocimiento anterior (a veces llamado experiencia) de los mismos.
- **Programación dinámica:** intenta resolver problemas bajando su coste computacional (cantidad de instrucciones a ejecutar → tiempo) a cambio de aumentar el coste espacial (aumentar el consumo de memoria).



1DAM PROG UNIDAD 1. Introducción a la Programación.

- **Ramificación y acotación:** se basa en la construcción de las soluciones al problema mediante un árbol, que se recorre de forma controlada, encontrando las mejores soluciones.
- **Vuelta atrás (backtracking):** se construye el espacio de soluciones de un problema dando pasos sucesivos, cuando un paso no conduce a una solución aceptable se retrocede a un paso anterior (back) y si es posible se sigue por otro camino, almacenando las soluciones menos costosas en cada paso y recordando el camino seguido (tracking) se obtiene una solución.

TRAZA DE UN ALGORITMO.

Para comprobar el funcionamiento de un algoritmo hay que usarlo. Hacer una traza significa comprobar las instrucciones que se van ejecutando para ciertos datos de entrada, ver las sentencias que ejecuta y como van cambiando los datos que no son fijos (los que pueden cambiar) desde que comienza hasta que acaba el algoritmo. Esto nos permite comprobar si el resultado que devuelve es correcto y detectar los errores en caso de no serlo.

EJEMPLO 11: vamos a hacer la traza del algoritmo que calcula la raíz cuadrada de un número x usando como dato de entrada el número 4. Haremos una tabla donde la primera columna identifica la instrucción que ejecuta el algoritmo, y el resto de columnas tiene el valor de las variables (datos que cambian) y las condiciones que usa para decidir que ejecutar después:

PASO 1:	Leer x
PASO 2:	$b \leftarrow x$
PASO 3:	SI b distinto de x/b ENTONCES
PASO 3.1	$b \leftarrow (x/b + b) / 2$
PASO 3.2	SALTA AL PASO 3
	FIN SI
PASO 4:	Escribir "La raíz cuadrada de ", x , " es ", b



PASOS	DATOS Y EXPRESIONES DEL ALGORITMO				
	x	b	x/b	b distinto de x/b	(x/b + b)/2
1	4				
2		4			
3			4/4=1	4 distinto de 1? cierto	
3_1		2,5			2,5
3			1,6	2.5 distinto de 1.6? cierto	
3_1		2,05			2,05
3			1,95	2.05 distinto de 1.95? cierto	
3_1		2			2
3			2	2 distinto de 2? falso	
4		2			

EJERCICIO 13: Este algoritmo tiene varios problemas. El primero lo podemos detectar si hacemos la traza de un número negativo. Haz la traza si le damos un n° negativo como -4. (La raíz cuadrada de un n° negativo no existe) ¿Cómo se comportará el algoritmo? ¿Lo hará bien? Indica lo que ocurre al hacer la traza y piensa como solucionarlo.

EJERCICIO 14: En matemáticas se llaman productos notables a ciertas expresiones que siempre se cumplen con independencia de los valores que se emplean (al contrario que las ecuaciones que solamente son ciertas para algunos valores concretos). Por ejemplo es sabido que:

$$(a + b)(a - b) = a^2 - b^2 \text{ para cualesquiera valores de } a, b.$$

El siguiente algoritmo que trabaja con números en coma flotante debería ser por tanto infinito. ¿Lo es? ¿Acaso no se cumplen las matemáticas? Si las matemáticas se cumplen, ¿Porqué acaba?



```
Algoritmo ejercicio14
  definir a, b, inc como real;
  inc <- 0.1;
  a <- 1;
  b <- 2;
  mientras (a + b) * (a - b) == a * a - b * b hacer
    escribir a, " ", b;
    a <- a + inc;
    b <- b + inc;
  fin mientras
  escribir "Si llego aquí, las matemáticas no se cumplen?";
fin algoritmo
```

1.3.1. PSEUDOCÓDIGO PARA ALGORITMOS.

Hay que tener en cuenta que **no hay un pseudocódigo estándar**. Nosotros vamos a usar una versión que sigue la filosofía de la **programación estructurada**.

Nota: Las bases de la programación estructurada las propuso **Niklaus Wirdth**. Según este científico, cualquier algoritmo se puede expresar usando 3 tipos de instrucciones:

- **Secuenciales.** Se ejecutan en el orden en que se escriben, una detrás de otra.
- **Alternativas.** Evalúan (calculan) una condición (expresión que devuelve true/false) y según el resultado, el flujo del programa se dirige a unas instrucciones o a otras.
- **Iterativas.** Repiten continuamente la ejecución de algunas instrucciones hasta que se cumple una determinada condición.

El tiempo le ha dado la razón y ha generado un estilo universal que exige a todo programador que utilice sólo instrucciones de estos tres tipos. Es lo que se conoce como **programación estructurada**. Las únicas sentencias permitidas en nuestro pseudocódigo son:

- **De Entrada/Salida.** Para leer/escribir datos del/al exterior.



1DAM PROG UNIDAD 1. Introducción a la Programación.

- **De proceso.** Operaciones (sumas, restas, cambiar un dato, ...)
- **De declaración.** Definen datos y subprogramas.
- **Llamadas a subprogramas:** permiten reutilizar código.
- **Comentarios.** Notas que se escriben junto al pseudocódigo para aclarar cosas y explicar mejor su funcionamiento.
- **De control de flujo.** Alternativas o iterativas.

Reglas Útiles para Escribir Pseudocódigo

Las instrucciones que expresan un algoritmo en pseudocódigo pueden estar entre las palabras **inicio** (begin) y **final** (end). Opcionalmente se pone delante del inicio la palabra **Algoritmo** o **proceso** seguida de un nombre que queramos darle. En definitiva la estructura de un algoritmo en pseudocódigo puede ser (entre otras) de estas 2 formas:

inicio	algoritmo nombre_del_algoritmo
instrucciones...	instrucciones...
fin	Fin algoritmo

Hay que ir adquiriendo buenos hábitos al redactar código:

- **Si es posible, use minúsculas:** no importa usar mayúsculas o minúsculas en pseudocódigo (otros lenguajes tienen otras normas), pero se aconsejan las minúsculas (lectura más clara).
- **Indente instrucciones:** si unas instrucciones están contenidas en otras, deje al menos 4 espacios en blanco (de sangría) por su izquierda, para que se detecte visualmente que están contenidas. **Permite comprender mejor la lógica del código y encontrar fallos mucho más rápido.**
- **Escriba comentarios:** En pseudocódigo los comentarios que se quieran escribir (costumbre muy aconsejable) se ponen con los símbolos **//** al principio de la línea del comentario (en algunas notaciones se escribe ******). Un comentario es una línea que sirve para ayudar a entender el código. Cada línea de comentario debe comenzar con esos símbolos. Ejemplo:



```
Algoritmo ejemplo
// Comentario 1
instrucciones
** Comentario 2
Instrucciones
Fin algoritmo
```

Tipos de Instrucciones o sentencias

Dentro del pseudocódigo se pueden utilizar los siguientes tipos:

- **Definir datos simbólicos.** Los datos que maneja el algoritmo pueden ser **literales** o **simbólicos**. Por ejemplo en la expresión $10/100 * \text{suelo}$ el 10 y el 100 son **constantes literales** (su valor es literalmente lo que hay escrito). Pero también participa otro dato que es un valor al que se da el nombre de **suelo**. Es un **dato simbólico** (asociamos un nombre a un dato, le damos un símbolo). Si el valor de este dato puede cambiar, se dice que ese nombre es una **variable**. La variable suelo podría servir por ejemplo para almacenar el salario de una persona en un algoritmo. La instrucción de pseudocódigo que declara una variable es:

definir <lista_símbolos> **como** <tipo>;

donde <lista_símbolos> es uno o más nombres de datos simbólicos separados por comas y <tipo> indica los valores que pueden contener esos datos. En pseudocódigo <tipo> puede ser:

- **entero:** números sin decimales.
- **numero / numerico / real:** números con decimales.
- **Logico:** valor de verdad.
- **carácter / texto / cadena.** Para letras.

EJEMPLO 12: algoritmo que define/declara datos simbólicos.



1DAM PROG UNIDAD 1. Introducción a la Programación.

```
Algoritmo ejemplo12
  definir PI, sueldo como numero;
  definir nombre como cadena;
  definir edad como entero;
  definir casado como logico;
  PI ← 3.14;
  nombre ← "Jose"; // Observa que el texto va dentro de comillas!
  Leer sueldo;      // Es buena idea avisar antes de lo que pides
  casado ← verdadero;
fin algoritmo.
```

- **Instrucciones primitivas.** Son las instrucciones del algoritmo que hacen cambios a los datos o al flujo de ejecución. Se escriben entre las palabras inicio y fin. Se representan mediante cuadrados en los diagramas de flujo. Sólo pueden contener estos posibles comandos:
 - **Asignaciones:** identificador (←) expresión con: datos literales o simbólicos, operaciones (+, -, * /,...) y llamadas a subalgoritmos de tipo función.
 - **Llamadas a subalgoritmos.**
 - **Instrucciones de entrada y salida:** leer y escribir.
- **Instrucciones alternativas.** Permiten ejecutar una o más instrucciones en función de una determinada condición. Se trata de la instrucción "**si ... entonces ... [sino ...] fin si**".
- **Instrucciones iterativas.** Son una o más instrucciones cuya ejecución se realiza continuamente hasta que una determinada condición se cumple. Son las instrucciones **mientras-hacer** (while), **repetir-hasta** (repeat-until), **hacer-mientras** (do-while) y **desde-hasta-hacer** (for).

Ahora veremos estas instrucciones con mayor detalle, para comenzar a pensar como plantear y resolver problemas con esta herramienta.



1.3.1.1. INSTRUCCIONES DE ASIGNACIÓN

Permiten almacenar (guardar/escribir/recordar) un valor en una variable del algoritmo. Para asignar el valor se escribe el símbolo `<-`, de modo que:

Sintaxis	Significado
identificador <- valor	En identificador se guarda valor. Si valor es una expresión (una fórmula), primero se evalúa (calcula) y el el resultado se guarda.

EJEMPLO 13: algoritmo con sentencias de asignación.

```
algoritmo ejemplo13
  definir x, y como enteros;
  y <- 9;
  leer x;
  x <- (x + 1) * sqrt(y); // sqrt(y) es la raiz cuadrada de y
fin algoritmo.
```

Los datos o valores que aparecen en las expresiones pueden ser:

- Números. Se escriben tal cual, el separador decimal es el punto.
- Caracteres simples. Los caracteres simples (un solo carácter) se escriben entre comillas simples: 'a', 'c', '1', etc.
- Textos. Se escriben entre comillas dobles: "Hola"
- Lógicos. Sólo pueden valer **verdadero/falso**, **V/F**, **1/0**.
- Identificadores. Si es una variable, se utiliza el valor que contiene. Si es un subprograma de tipo función, al ejecutarse devuelve un valor que es el que se usa.

En las instrucciones de asignación se pueden utilizar expresiones más complejas con ayuda de **los operadores**. Ejemplo:

```
x <- y * 3 / 2
```



1DAM PROG UNIDAD 1. Introducción a la Programación.

Es decir, x vale el resultado de multiplicar el valor de y por tres y dividirlo entre dos. Los operadores permitidos son:

+ Suma - Resta o cambio de signo
* Producto / División mod Resto

Cuando escribas expresiones, ten en cuenta la prioridad de operadores porque el producto, división y módulo tienen más prioridad (se aplican antes) que sumas y restas.

Si escribes: $9 + 6 / 3$ da como resultado 11 y no 5. Para modificar esta prioridad, se encierra entre paréntesis lo que quieras ejecutar primero. Pero debes usarlos solamente si es necesario, o complicas el código. Ej: $(9 + 6) / 3$ sí da 5.

1.3.1.2. INSTRUCCIONES DE ENTRADA Y SALIDA

Lectura de datos Es la instrucción que lee un dato del exterior. Se hace mediante la orden **leer** en la que entre paréntesis opcionales se indica el identificador de la variable que almacenará lo que se lea. Ejemplo:

pseudocódigo	Diagrama de flujo
Leer x ó leer(x)	

Se pueden leer varias variables a la vez:

pseudocódigo	Diagrama de flujo
leer(x, y, z) ó leer x, y, z	



Escritura de datos Funciona como la anterior pero usando la palabra **escribir**. Indica la salida de datos del algoritmo hacia el exterior.

pseudocódigo	Diagrama de flujo
<pre>escribir(x, y, z) o escribir x, y, z [sin saltar]</pre>	

EJEMPLO 14: Pseudocódigo y algoritmo que escribe el resultado de multiplicar dos números leídos por teclado.

pseudocódigo	Diagrama de flujo
<pre>algoritmo producto leer n1, n2; escribir n1 * n2; fin</pre>	

1.3.1.3. INSTRUCCIONES DE CONTROL DE FLUJO.

Con las instrucciones vistas hasta ahora sólo se pueden escribir algoritmos donde la ejecución de las instrucciones sea secuencial (una detrás de otra, desde la primera a la última). Pero para resolver algunos problemas se necesita cambiar esta forma de ejecución.

La programación estructurada permite el uso de **decisiones** y de **iteraciones**. Estas instrucciones permiten que haya instrucciones que se pueden ejecutar o no, según una condición (**instrucciones alternativas**), e incluso que se ejecuten repetidamente hasta que se



cumpla una condición (**instrucciones iterativas**). En definitiva son **instrucciones que permiten cambiar el flujo de ejecución del algoritmo, por defecto secuencial**.

Expresiones lógicas

Todas las instrucciones de control de flujo utilizan **expresiones lógicas**. Son expresiones (fórmulas) que dan como resultado un **valor lógico (verdadero o falso)**.

Las expresiones lógicas son simples cuando tienen un único valor. Suelen ser siempre el resultado de comparaciones entre datos. Por ejemplo $n1 > 8$ da como resultado verdadero si $n1$ vale más que 8. Los **operadores de relación (de comparación)** que se pueden utilizar son:

> Mayor que	\geq (\geq) Mayor o igual
< Menor que	\leq (\leq) Menor o igual
\neq (\neq) Distinto	= Igual

También se pueden unir varias expresiones simples para formar otra más compleja utilizando los operadores **Y** (en inglés AND), el operador **O** (en inglés OR) o el operador **NO** (en inglés NOT). Por ejemplo:

Expresión	Resultado verdadero si...
$a > 30$ Y $a < 45$	La variable a está entre 31 y 44
$a < 30$ O $a > 45$	La variable a no está entre 30 y 45
NO $a = 45$	La variable a no vale 45
$a > 8$ Y NO $b < 7$	La variable a es mayor que 8 y b es mayor o igual que 7
$a > 45$ Y $a < 30$	Nunca es verdadero

Nota: en cuanto a la precedencia, primero el NOT, luego el AND y el último el OR (como la suma).

EJEMPLO 15: Varias expresiones lógicas.

```
diceMiau ← verdadero;  
esGato <- diceMiau;  
barato <- precio < 1;
```



1DAM PROG UNIDAD 1. Introducción a la Programación.

```
caro <- NO barato;  
peligroso <- esGato 0 velocidad > 120;  
tieneFiebre <- grados > 37.5;  
entre5y8 <- 5 <= x Y x >= 8;
```

EJEMPLO 16: La hora de tutoría es los lunes y martes a las 16 horas.

¿Qué expresión es correcta?

- a) dia = "lunes" 0 dia = "martes" Y hora == 16 // Mal
- b) (dia = "lunes" 0 dia = "martes") Y hora == 16 // Bien

EJERCICIO 15: dados el día y mes de nacimiento de una persona, esTauro será cierto si nació entre el 21/4 y el 21/5.

Instrucción de alternativa simple

La alternativa simple se crea con la instrucción **si** (en inglés **if**). Esta instrucción evalúa una determinada **expresión lógica (condición)** y dependiendo de si el resultado es verdadero o no, se ejecutan las instrucciones siguientes. Funcionamiento:

pseudocódigo	Diagrama de flujo
<pre>si expresión_lógica entonces instrucciones_si_verdadera fin_si</pre>	

Las instrucciones sólo se ejecutan si la condición es verdadera.

instrucción de alternativa doble

Se trata de una variante de la anterior en la que se ejecutan unas instrucciones si la expresión evaluada es verdadera y otras si es falsa.



pseudocódigo	Diagrama de flujo
<pre>si expresión_lógica entonces instrucciones sino instrucciones fin_si</pre>	

EJEMPLO 17: Sentencia alternativa simple.

```
si nota >= 5 entonces
    aprobado <- verdadero;
finsi;
```

EJEMPLO 18: Sentencia alternativa doble.

```
si x != 0 entonces
    Escribir "inverso de ", x, 1 / x;
sino
    Escribir "No tiene inverso";
finsi;
```

Se puede escribir una instrucción **si** dentro de otro. A eso se le llama **alternativas anidadas**.

EJEMPLO 19. Alternativas anidadas.

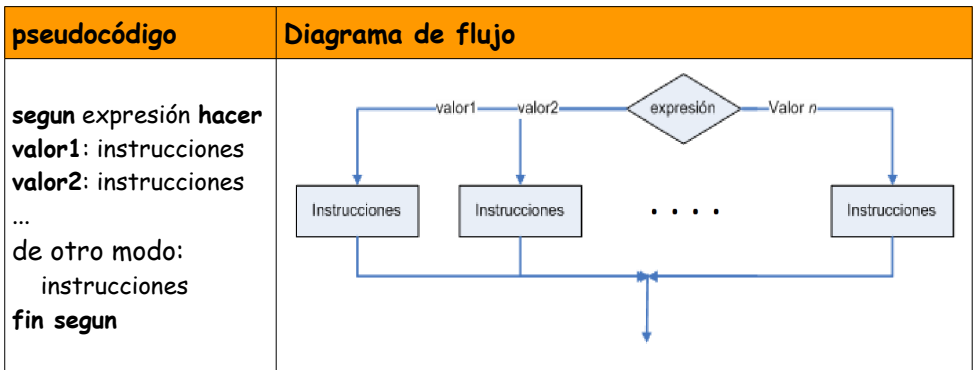
```
si a >= 5 entonces
    escribir "apto" // Observa que la indentación te
    si a >= 5 y a < 7 entonces // describe la estructura del
        escribir "nota:aprobado" // código de forma visual
    finsi
    si a ≥ 7 y a<9 entonces
        escribir "notable"
    sino
        escribir "sobresaliente"
    finsi
sino
    escribir "suspenseo"
finsi;
```



Al anidar estas instrucciones hay que tener en cuenta que hay que cerrar las instrucciones **si** interiores antes que las exteriores. Eso es una regla básica de la programación estructurada.

Instrucción alternativa compuesta

En muchas ocasiones se requiere contemplar más de dos alternativas. Para estos casos podemos encadenar o anidar sentencias **si...entonces**, o también existe una instrucción (**según_sea expresión**) que evalúa la expresión y según los valores que tome, ejecuta unas u otras.



Se evalúa la expresión y si es igual a uno de los valores interiores se ejecutan las instrucciones de ese valor. Si no coincide con ningún valor se ejecutan las instrucciones de la sección **de otro modo**.

EJEMPLO 20. Alternativas múltiples usadas para un menú.

```
Escribir "1. Leer valores a y b.";
Escribir "1. Suma a + b";
Escribir "3. Resta a - b";
Escribir "4. Multiplica a * b.";
Escribir " Que quiere hacer: " sin saltar;
Leer opcion;
segun opcion hacer
  1: Leer a, b;
  2: Escribir "suma: ", a + b;
  3: Escribir "resta: ", a - b;
  4: Escribir "producto: ", a * b;
```



de otro modo "Opción incorrecta";
finsegun;

Instrucciones iterativas de tipo mientras-hacer

Los algoritmos necesitan instrucciones **iterativas (repetitivas)**. ¿Porqué se necesitan sentencias que repitan la ejecución de otras varias veces? Imagina este ejemplo: Haz un algoritmo que imprima los números del 1 al 10:

```
algoritmo uno
  escribe 1
  ...
  escribe 10
fin
```

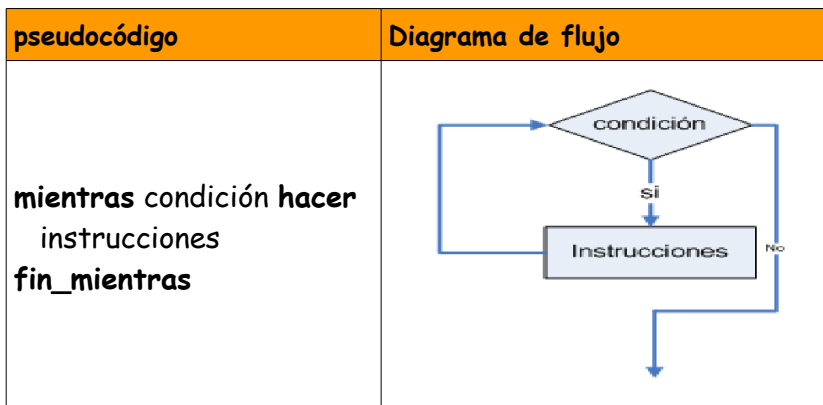
Ahora este: haz un algoritmo que lea un n° llamado n , y escriba los números del 1 hasta n (Ahora el n puede ser cualquier entero). Si no puedes repetir, no lo puedes hacer con un esfuerzo razonable.

Ahora este otro: Haz un algoritmo que lea números hasta que su valor sea mayor que 10 e indica cuantos n° s has leído hasta ese momento (si no puedes repetir, no lo puedes hacer).

La primera sentencia iterativa que comentamos es conocida como **mientras-hacer** (en inglés **while-do**). Su estructura es la de la tabla y su funcionamiento consiste en:

PASO 1: se calcula la condición.

PASO 2: si es cierta se ejecutan las sentencias del interior y se vuelve al PASO 1. Si es falsa, se acaba la ejecución de la sentencia y se continúa por la siguiente.



Significa que las instrucciones del interior se ejecutan una y otra vez mientras la condición sea verdadera. Si la condición es falsa, las instrucciones se dejan de ejecutar.

EJEMPLO 21: implementar los algoritmos propuestos en el texto.

Nºs del 1 al 10	Nºs de 1 a n	Cuantos menores de 10
<pre>Algoritmo uno definir i como entero; i ← 1; mientras i <= 10 hacer escribir i; i ← i + 1; fin_mientras fin algoritmo</pre>	<pre>Algoritmo dos // en pseudocódigo // definir datos es // opcional según conf. Escribir "Teclee n: "; Leer n; i ← 1; mientras i <= n hacer escribir i; i ← i + 1; fin_mientras fin algoritmo</pre>	<pre>Algoritmo tres contador ← 0; n ← 1 // para que entre mientras n <= 10 hacer Escribir "Teclee n: "; leer n; si n <= 10 entonces contador ← contador + 1 fin si fin mientras escribir contador; fin algoritmo</pre>

Las instrucciones interiores de **mientras** podrían incluso no ejecutarse si la condición es falsa inicialmente, por eso en el tercer ejemplo, nos aseguramos de que la primera vez entre, asignando a n un valor 1.



Instrucciones iterativas de tipo repetir-hasta

La diferencia con la instrucción anterior está en que se evalúa la condición al final (en lugar de al principio). Consiste en una serie de instrucciones que se ejecutan repetidamente **hasta que** la condición sea verdadera (**funciona al revés que el mientras**) ya que si la condición es falsa, las instrucciones se siguen ejecutando.

Nota: No todos los lenguajes de programación tienen todas las sentencias de control. Debemos tener la habilidad de intercambiarlas.

Nota: Para transformar instrucciones repetir-hasta en repetir-mientras y viceversa, invierta la condición (lo contrario).

pseudocódigo	Diagrama de flujo
repetir instrucciones hasta condición	<pre>graph TD; A[Instrucciones] --> B{condición}; B -- si --> C[]; B -- no --> A;</pre>

EJEMPLO 22: escribir números del 1 al 10 con un bucle repetir-hasta.

```
x <- 1;  
repetir  
    escribir x;  
    x ← x + 1;  
hasta x > 10;
```



Instrucciones iterativas de tipo hacer...mientras

Ejecuta unas instrucciones mientras se cumpla una condición. La condición se evalúa tras la ejecución de las instrucciones.

pseudocódigo	Diagrama de flujo
hacer instrucciones mientras condición	<pre>graph TD; A[Instrucciones] --> B{condición}; B -- si --> A; B -- no --> C[];</pre>

Es un bucle **mientras** donde las instrucciones al menos se ejecutan una vez (lo mismo que repetir-hasta salvo que la condición se interpreta al revés).

Este formato está presente en el lenguaje C y derivados (C++, Java, C#, JavaScript) y algunos otros (Python, ...), mientras que el formato de **repetir-hasta** está presente en lenguajes como Pascal, PL/SQL, etc.

Nota: Esta sentencia no la tiene el programa psInt.

EJEMPLO 23: escribir números del 1 al 10 con un bucle hacer-mientras.

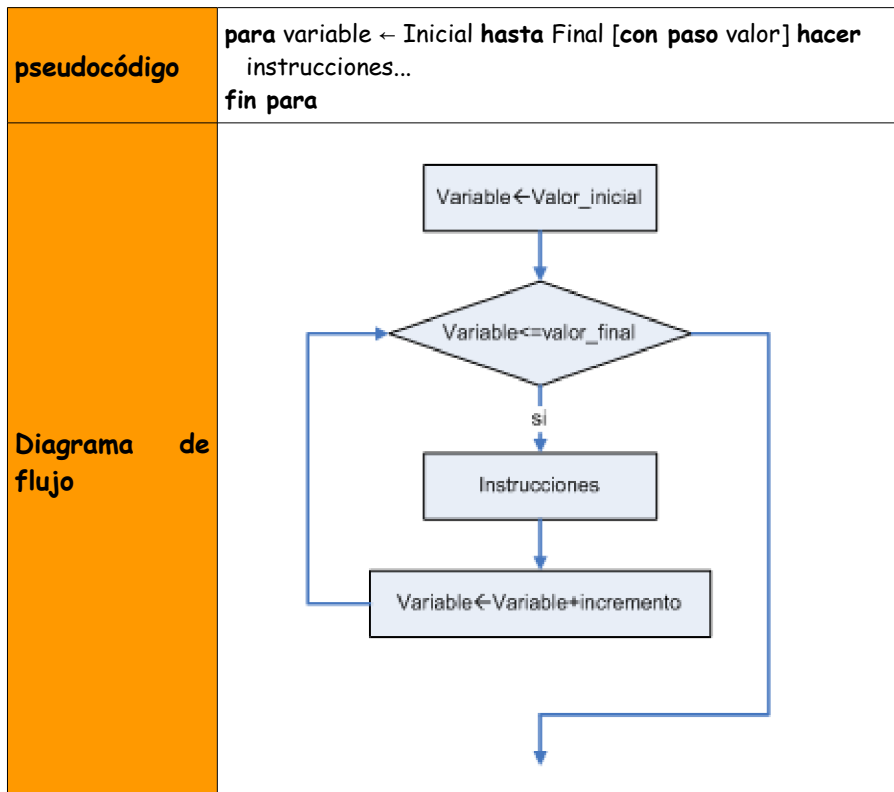
```
x ← 1
hacer
    escribir x
    x ← x + 1
mientras x <= 10
```




Instrucciones iterativas para (por contador)

Existe otro tipo de estructura iterativa. En realidad no sería necesaria porque con un **mientras** se puede hacer, pero como automatiza crear **bucles por contador**, casi todos los lenguajes de programación la tienen.

Son instrucciones que repiten continuamente las que contienen, según los valores de un contador al que se le pone un valor de inicio, un valor final y el incremento en cada iteración (repetición).





También hay otros símbolos específicos en los diagramas de flujo para esta instrucción. Cosas a tener en cuenta del cambio del contador:

- La variable contador la cambia automáticamente la sentencia.
- No es buena idea cambiarle el valor dentro del bucle.
- La cantidad de incremento de la variable contador (paso) por defecto es 1.

EJEMPLO 24: escribir números del 1 al 10 usando un bucle por contador.

```
para x <- 1 hasta 10 hacer
  escribir x;
fin para
```

1.3.1.4. SUBALGORITMOS/SUBPROCESOS/SUBPROGRAMAS.

Imagina que en un algoritmo hay una tarea de unas 20 instrucciones y hay que repetirla en siete lugares diferentes (no 7 veces en el mismo lugar del código como un bucle, sino en diferentes lugares del código).



Figura 7: Mismas sentencias en varios lugares.

Caben varias posibilidades de hacerlo y cada una tiene pros y contras:

- Primera: Repetir las mismas instrucciones. Desventajas:
 - El algoritmo es más difícil de leer (más largo, son 140 instrucciones adicionales).
 - Si pasados dos meses te das cuenta que cometiste un error, o quieres cambiar la forma de realizar ese trabajo, tendrás que revisar en 7 sitios para arreglarlo (se complica el



mantenimiento, se pierde eficiencia en el desarrollo y se aumenta la posibilidad de cometer errores).

- Segunda: **Usar una macro.** Una macro consiste en darle a ese código un nombre y cuando se escriba el nombre la macro, ese nombre en realidad representa a todas las sentencias. El programa en realidad sigue siendo igual de grande, aunque cuando lo trabajas parece más corto. La ventaja de las macros es que el código es rápido. La desventaja es que el programa real, es largo y pueden aparecer efectos colaterales.
- Tercera: **Subprograma.** A esas instrucciones se les da un nombre (identificador) y cuando queramos ejecutarlas, se hace una llamada (escribimos su nombre en el lugar donde queremos volver a ejecutarla). A estos trozos de código con nombre se les llama sub-algoritmos, subprogramas, subrutinas y hay dos grandes tipos: **funciones y procedimientos.** Una de las diferencias con las macros es que el código no se vuelve a "pegar", simplemente se ejecuta en el único lugar donde está almacenado.

La tercera opción es la idea de los subalgoritmos: una o varias instrucciones que juntas hacen una tarea muy concreta a las que se agrupa y se les da nombre dentro del algoritmo, y se pueden ejecutar cuando sean necesarias realizando una llamada (usando ese nombre).

A los subalgoritmos, si es necesario, se les puede pasar los datos con los que deben trabajar. En cada llamada, estos datos pueden cambiar. Lo que no cambia es el trabajo que hace con ellos (el código). A estos datos se les llama **parámetros (o argumentos)**. También pueden declarar constantes y variables propias denominadas **variables locales** (no existen ni se conocen fuera del subalgoritmo).



1DAM PROG UNIDAD 1. Introducción a la Programación.

Cuando se define un subalgoritmo hay que indicar cuantos parámetros necesita. Cuando se hace una llamada desde otro algoritmo, hay que pasarle esa cantidad de datos y en el orden en que los espera. Si no se dice nada, los datos pasados como parámetros son locales, de forma que cualquier cambio que se les haga, desaparece y no afecta al dato original. Pero si se indica **por referencia** al definir el subalgoritmo, significa que los cambios que se hacen al parámetro, se hacen también al dato original, no a una copia.

Hay dos tipos de subalgoritmos:

- **Procedimientos:** cuando se llaman, ejecutan sus instrucciones pero no devuelven ningún valor.
- **Funciones:** además de ejecutar sus instrucciones cuando son llamadas, devuelven un valor. Esto hace que puedan utilizarse en expresiones, porque realmente equivalen a un valor que participa en la expresión, solo que está pendiente de calcularse.

Tener subalgoritmos te permite subir el nivel lógico en el que programas, porque como unos pueden llamar a otros, digamos que puedes construir "nuevas sentencias" que al usarlas hacen trabajos más complejos, de forma que con menos código, puedes hacer más cosas.

Sintaxis de creación de funciones:

```
subalgoritmo variable_devuelta <- nombre ( parámetro1, parámetro2, ... )  
  definir variable_local1 como tipo;  
  instrucciones...;  
  variable_devuelta ← expresión // Este es el valor que devuelve  
fin_subalgoritmo
```

Sintaxis de creación de procedimientos:

```
subalgoritmo nombre ( parámetro1, parámetro2 [por referencia, ...] )
```



definir variable_local1 como tipo;

instrucciones...; // no devuelve nada

fin_subalgoritmo

Nota: en pseInt se puede usar las palabras **funcion** y **subproceso** además de subalgoritmo, pero no procedimiento.

Ejercicio 16: Describe los siguientes subalgoritmos indicando si son procedimientos o funciones, si tienen argumentos, lo que hacen, donde se hacen sus llamadas y el resultado de ejecutar el programa:

```
subalgoritmo pulsaTecla
    Escribir "Pulse tecla para continuar..." sin saltar;
    Esperar Tecla;
fin subalgoritmo

subalgoritmo r <- razon(a, b)
    si b != 0 entonces
        r <- a / b;
    sino
        escribir "Error al calcular la razón";
    fin si
fin subalgoritmo

subalgoritmo e <- leeEntero(msj)
    definir e como entero;
    Escribir msj sin saltar;
    Leer e;
fin subalgoritmo

algoritmo uno
    n1 <- LeeEntero("Edad: ");
    pulsaTecla;
    n2 <- LeeEntero("Talla de pie: ");
    pulsaTecla;
    Escribir "Su pie ha crecido ", razon(n2, n1), " tallas al año";
    pulsaTecla;
fin algoritmo
```



EJEMPLO 25: Crear una función llamada **at** a la que se pasan dos parámetros, la base y la altura y devuelve el área del triángulo.

```
subalgoritmo area ← areaTri(base, altura)
    area ← base * altura / 2;
finsubalgoritmo
```

EJERCICIO 17: Haz un procedimiento que se llame *intercambia* al que pasas por referencia dos variables enteras e intercambian su valores. Haz un algoritmo que lea dos variables enteras e imprima los valores que ha leído. Luego llama a *intercambia* y vuelve a imprimir sus valores. Si los parámetros no se pasan por referencia y se pasan por valor, ¿Cuál sería el resultado? ¿Porqué?

EJERCICIO 18:
¿Qué salida genera este algoritmo si implementa los siguientes subalgoritmos en forma de funciones?

```
1  Funcion f <- f1(a)
2      a <- a + 1;
3      f <- a + b;
4  fin funcion
5
6  Funcion f <- g(a por referencia)
7      a <- a + 1;
8      f <- a + b;
9  fin funcion
10
11 Funcion f <- h(a)
12     definir n1 como entero;
13     a <- a + 1;
14     f <- a + b;
15 fin funcion
16
17 Algoritmo ejercicio18
18     definir nada como entero;
19     n <- 3;
20     Escribir "Antes de f1(n): ", n;
21     nada <- f1(n);
22     Escribir "Después de f1(n): ", n;
23     Escribir "Antes de g(n): ", n;
24     nada <- g(n);
25     Escribir "Después de g(n): ", n;
26     Escribir "Antes de h(n): ", n;
27     nada <- h(n);
28     Escribir "Después de h(n): ", n;
29 finalgoritmo
```



1.4. HARDWARE DE UN COMPUTADOR.

Según la RAE, una computadora es una máquina electrónica, analógica o digital, dotada de una memoria de gran capacidad y de métodos de tratamiento de la información, capaz de resolver problemas matemáticos y lógicos mediante la utilización automática de programas informáticos. Todos tenemos una imagen clara de lo que es una computadora, o como se la conoce popularmente, un ordenador.

Nota: cualquier dispositivo (como un móvil, o un electrodoméstico), que sea capaz de ejecutar programas, puede asimilarse a esta definición.

Inicialmente, las primeras computadoras eran máquinas basadas en el funcionamiento de relés o de ruedas. Por eso sólo eran capaces de realizar una única tarea. A finales de los años cuarenta, **Von Newman** escribió en un artículo las bases del funcionamiento de los ordenadores (seguidos en su mayor parte hasta el día de hoy).

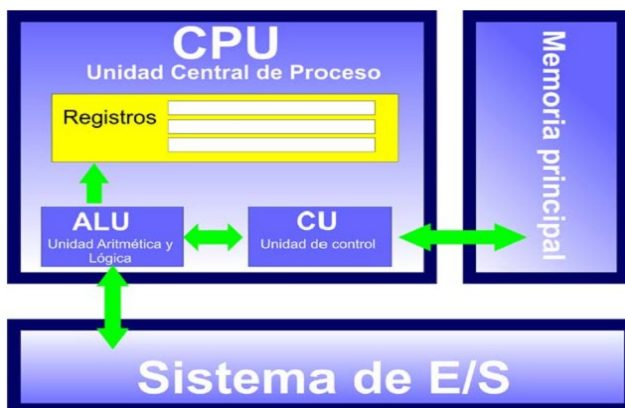


Figura 8: Elementos fundamentales del ordenador.

Las mejoras que consiguió este modelo (entre otras) fueron:



1DAM PROG UNIDAD 1. Introducción a la Programación.

- Incluir el modelo de **Programa Almacenado** (fundamental para que el ordenador pueda realizar más de una tarea).
- Aparece el concepto de programa como secuencia de instrucciones secuenciales (aunque pueden incluir bifurcaciones y saltos).

El modelo no ha cambiando excesivamente hasta la actualidad, los ordenadores basados en el silicio lo siguen. De los componentes internos del ordenador, cabe destacar **el procesador (o microprocesador o CPU)**. Se trata de un chip que contiene todos los elementos de la **Unidad Central de Proceso** y que es capaz de realizar e interpretar instrucciones. En realidad un procesador sólo es capaz de realizar tareas sencillas como:

- **Operaciones aritméticas y lógicas simples:** suma, resta, multiplicación y división y AND, OR, NOT, XOR.
- **Operaciones de comparación entre valores y saltar a otras instrucciones, etc.**
- **Almacenamiento y movimiento de datos**

En definitiva, los principales componentes actualmente son:

- **Procesador.** Núcleo digital en el que reside la CPU del ordenador. Es la parte fundamental.
- **Placa base.** Circuito interno al que se conectan todos los componentes del ordenador, incluido el procesador.
- **Memoria RAM.** Memoria principal del ordenador, formada por circuitos digitales conectados mediante tarjetas a la placa base. Su contenido se pierde cuando se desconecta el ordenador. Lo que se almacena no es permanente. Mientras el ordenador está funcionando contiene todos los programas y datos con los que el ordenador trabaja.



1DAM PROG UNIDAD 1. Introducción a la Programación.

- **Memoria caché.** Memoria muy rápida similar a la RAM, pero de velocidad mucho más elevada, por lo que se utiliza para almacenar los últimos datos utilizados de la memoria RAM.
- **Periféricos.** Aparatos conectados al ordenador mediante tarjetas o ranuras de expansión (puertos). Los hay de **entrada** (introducen datos: teclado, ratón, escáner, ...), de **salida** (sacan datos desde el ordenador: pantalla, impresora, altavoces, ...) e incluso de **entrada/salida** (módem, tarjeta de red).
- **Unidades de almacenamiento.** En realidad son periféricos, pero que sirven para almacenar de forma permanente los datos que se deseen del ordenador. Los principales son el **disco duro** (unidad de gran tamaño interna al ordenador), **disqueteras** (en desuso), **CD-ROM**, **DVD**.

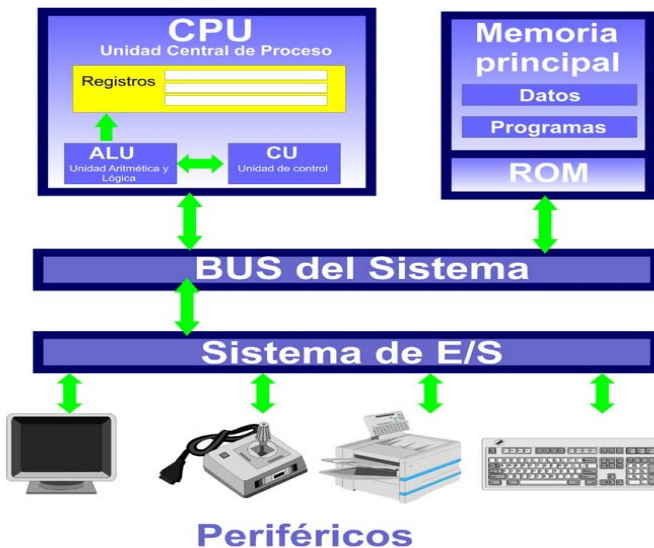


Figura 9: Arquitectura de Von Neuman.

Vamos a comentar el funcionamiento de la CPU para entender como se ejecutan los programas.



ELEMENTOS DE UNA CPU

Internamente una CPU tiene:

- **Registros:** memorias muy rápidas y de poca capacidad.
 - De datos: almacenan los datos que manipula un programa.
 - De segmento: ayudan a definir donde comienzan ciertas zonas de memoria (**segmentos**). Se usan para evitar mezclar instrucciones y datos (muy peligroso). El programa define un segmento de datos (donde almacena datos), un segmento de programa (almacena el programa), un segmento de pila (donde guarda las direcciones de retorno después de ejecutar una subrutina, sus parámetros y variables locales...)
 - De flags (banderas): cuando se realiza una operación se activan ciertos bits que sirven de chivatos o avisadores. Por ejemplo, si se restan dos números y el resultado es cero, se activa el flag Z (resultado 0).
 - Apuntadores: apuntan a zonas de memoria. Como PC/IP (apunta a la siguiente instrucción a ejecutar), SP (puntero de pila, indica hasta donde llega la pila), BP (puntero de base de pila....)
- **ALU (Unidad Aritmético Lógica):** circuitos que hacen operaciones aritméticas y lógicas.
- **UC (Unidad de Control):** se encarga de ejecutar las instrucciones.
- **Buses Internos:** interconectan los elementos dentro de la CPU.
 - Bus de direcciones: indica la dirección de memoria RAM que la CPU quiere leer o escribir.
 - Bus de datos: permite que se muevan los datos entre un elemento de origen hasta el destino.
 - Bus de control: la UC da las órdenes y consulta el estado.



PARÁMETROS IMPORTANTES:

- Reloj: el ritmo al que trabajan los componentes. Se mide en Hz.
- Tamaño de palabra: cuantos bytes tiene el mayor dato con el que trabaja.
- Tamaño de bus de direcciones: se mide en bits y a más bits, más memoria RAM puede usar.

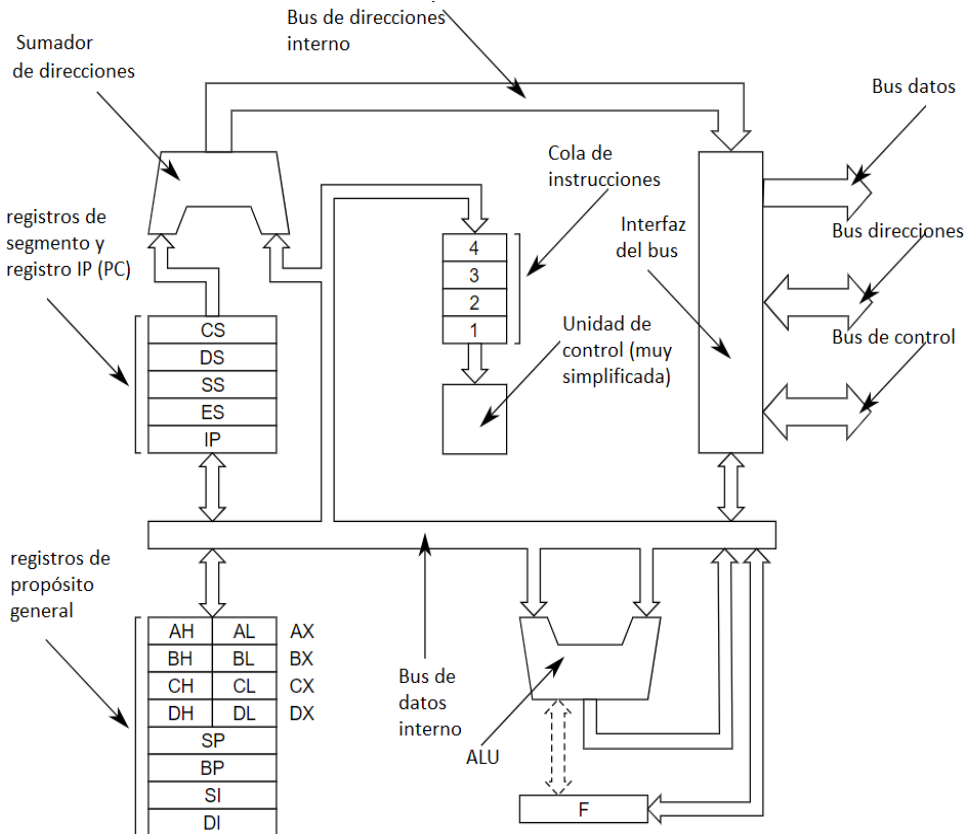


Figura 10: Arquitectura interna de la CPU Intel 8086.

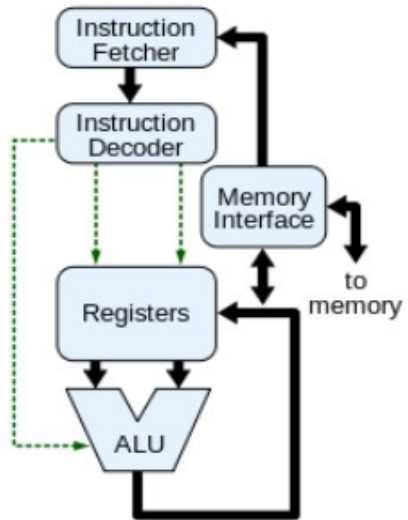
La operación fundamental de la mayoría de las CPU es ejecutar una secuencia de instrucciones almacenadas en una memoria llamadas



1DAM PROG UNIDAD 1. Introducción a la Programación.

«programa». El programa está representado en binario y almacenado a partir de cierta dirección de memoria RAM. Hay instrucciones que son de 1 byte, otras necesitan dos bytes, otras 3 bytes o más. Hay cuatro pasos que casi todos las CPU de la [arquitectura de von Neumann](#) usan en su operación: leer, decodificar, ejecutar y escribir.

- **Leer:** se recupera una instrucción (una secuencia de bytes) de la memoria. La CPU sabe donde está almacenada (la dirección en RAM) esa instrucción porque uno de sus registros (llamado [contador de programa PC](#), o [puntero de instrucción IP](#)) sigue el rastro de donde se almacena la instrucción siguiente que hay que ejecutar. Después de leer una instrucción, el PC se incrementa para apuntar al lugar donde está la siguiente.



- **Decodificar:** la instrucción es dividida en partes que tienen significado para otras unidades de la CPU. A menudo, un grupo de bits en la instrucción, llamados opcode, indican qué operación realizar. Las partes restantes proporcionan información adicional, como operandos de una operación. Tales operandos se pueden dar como un valor constante (llamado valor inmediato), o como un lugar donde localizar un valor, que puede ser un [registro](#) o una dirección de memoria. En diseños antiguos las unidades de la CPU responsables de decodificar la instrucción eran dispositivos hardware fijos. Sin embargo, en CPUs más



1DAM PROG UNIDAD 1. Introducción a la Programación.

abstractos y complicados, es frecuentemente usar un microprograma (**firmware**), es decir cada instrucción es en realidad un programa escrito en firmware.

- **Ejecución:** Durante este paso, varias unidades del CPU intervienen en realizar la operación indicada por la instrucción. Si, por ejemplo, se pide una operación suma, una ALU se conecta a un conjunto de entradas y un conjunto de salidas. Las entradas proporcionan los números a ser sumados, y las salidas contendrán la suma final. Los flags se actualizan....
- **Escribir:** «escribe» los resultados del paso de ejecución a una cierta forma de memoria. Muy a menudo, los resultados son escritos a algún registro interno de la CPU. En otros casos pueden ser escritos en RAM (más lenta pero más barata y más grande). Algunos tipos de instrucciones manipulan el contador de programa en lugar de directamente producir datos de resultado. Estas son llamadas generalmente "saltos" (jumps) y facilitan comportamientos como bucles, la ejecución condicional de programas (con el uso de saltos condicionales), y funciones en programas.

Después de la ejecución de la instrucción y la escritura de los datos resultantes, el proceso entero se repite con el siguiente ciclo de instrucción, normalmente leyendo la siguiente instrucción en secuencia debido al valor incrementado en el contador de programa.

Si la instrucción completada era un salto, el contador de programa será modificado para contener la dirección de la instrucción a la cual se saltó, y la ejecución del programa continúa normalmente. En CPUs más complejos que el descrito aquí, múltiples instrucciones pueden ser leídas, decodificadas, y ejecutadas simultáneamente.



1.5. LENGUAJES DE PROGRAMACIÓN.

Actualmente hay más de 1220 lenguajes de programación (sin contar sus variantes). Sus relaciones las puedes conocer haciendo clic en el enlace para acceder al universo de los lenguajes <https://exploring-data.com/vis/programming-languages-influence-network-2014/>

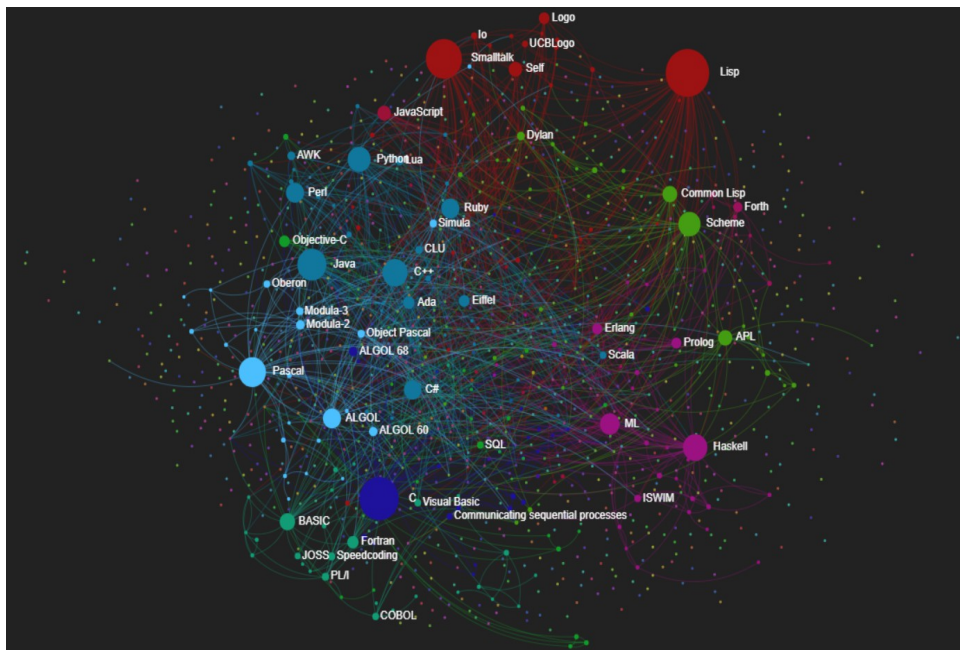


Figura 11: Algunos de los lenguajes de programación y su relación.

BREVE HISTORIA DE LOS LENGUAJES DE PROGRAMACIÓN

INICIOS DE LA PROGRAMACIÓN

Charles Babbage definió a mediados del siglo XIX lo que él llamó la **máquina analítica**. Se considera a esta máquina el diseño del primer ordenador. La realidad es que no se pudo construir hasta el siglo siguiente. El caso es que su colaboradora **Ada Lovelace** escribió en



1DAM PROG UNIDAD 1. Introducción a la Programación.

tarjetas perforadas una serie de instrucciones que la máquina iba a ser capaz de ejecutar. Se dice que eso significó el inicio de la programación de ordenadores. En la segunda guerra mundial debido a las necesidades militares, la ciencia de la computación prospera y con ella aparece el famoso **ENIAC** (Electronic Numerical Integrator And Calculator), que se programaba cambiando su circuitería. Esa es la primera forma de programar (aún se usa en algunas máquinas) que sólo vale para **máquinas de único propósito**. Si se cambia el propósito, hay que modificar la máquina.

LENGUAJE MÁQUINA (PRIMERA GENERACIÓN DE LENGUAJES).

No mucho más tarde apareció la idea de que las máquinas fueran capaces de realizar más de una aplicación. Para lo cual se ideó que hubiera una memoria donde se almacenan esas instrucciones. Esa memoria se podía rellenar con datos procedentes del exterior.

Inicialmente se utilizaron tarjetas perforadas para introducir las instrucciones. Durante mucho tiempo esa fue la forma de programar, teniendo en cuenta que las máquinas ya entendían sólo código binario, consistía en introducir la programación de la máquina mediante unos y ceros. El llamado **código máquina**. En realidad los ordenadores es el único código que entienden, por lo que cualquier forma de programar debe de ser convertida a código máquina.

El código máquina sólo se ha utilizado por los programadores en los inicios de la informática. Su incomodidad hace que sea impensable usarlo actualmente. Pero cualquier programa de ordenador debe, finalmente, ser convertido a este código para que un ordenador pueda ejecutar las instrucciones. Un detalle a tener en cuenta es que el código máquina es distinto para cada tipo de procesador. Lo que hace que los programas en código máquina no sean portables entre distintas máquinas.



ENSAMBLADOR (SEGUNDA GENERACIÓN DE LENGUAJES, 2GL)

En los años 40 se intentó generar un lenguaje más simbólico que permitiera no tener que programar utilizando código máquina. Apareció el **lenguaje ensamblador**, que es la traducción del código máquina a una forma más simbólica. Cada tipo de instrucción se asocia a una palabra nemotécnica (como SUM para sumar por ejemplo), de forma que cada palabra tiene traducción directa a una instrucción del código máquina.

Tras escribir el programa en ensamblador, un programa (llamado también ensamblador) se encargará de traducir el lenguaje ensamblador a código máquina. Esta traducción es rápida puesto que cada línea en ensamblador tiene equivalente directo en código máquina (en los lenguajes modernos no ocurre esto).

La idea es la siguiente: si en el código máquina, el número binario 0000 significa sumar, y el número 0001 significa restar. Una instrucción máquina que sumara el número 8 (00001000 en binario) al número 16 (00010000 en binario) sería: 0000 00001000 00010000 Realmente no habría espacios en blanco, el ordenador entendería que los primeros cuatro BITS representan la instrucción y los 8 siguientes el primer número y los ocho siguientes el segundo número (suponiendo que los números ocupan 8 bits). Lógicamente trabajar de esta forma es muy complicado. Por eso se podría utilizar la siguiente traducción en ensamblador: SUM 8 16 Que ya se entiende mucho mejor.

EJEMPLO 26: programa que saca el texto "Hola mundo" por pantalla.

```
DATOS SEGMENT
saludo db "Hola mundo!!!", "$"
DATOS ENDS
CODE SEGMENT
assume cs:code, ds:datos
START PROC
mov ax, datos
```




1DAM PROG UNIDAD 1. Introducción a la Programación.

```
mov ds,ax
mov dx,offset saludo
mov ah,9      ; servicio de sacar por pantalla un texto
int 21h       ; llamada a la BIOS del PC
mov ax,4C00h
int 21h
START ENDP
CODE ENDS
END START
```

Puesto que el ensamblador es una representación textual pero exacta del código máquina, cada programa sólo funcionará para la máquina en la que fue concebido el programa, es decir, no es **portable**. La ventaja de este lenguaje es que se puede controlar absolutamente el funcionamiento de la máquina, lo que permite crear programas muy eficientes. Lo malo es precisamente que hay que conocer muy bien el funcionamiento de la computadora para crear programas con esta técnica. Además las líneas requeridas para realizar una tarea se disparan ya que las instrucciones de la máquina son excesivamente simples.

LENGUAJES DE ALTO NIVEL (TERCERA GENERACIÓN, 3GL)

Aunque el ensamblador significó una notable mejora sobre el código máquina, seguía siendo excesivamente críptico. De hecho, hacer un programa sencillo requiere miles de líneas de código. Para evitar los problemas del ensamblador apareció la tercera generación de lenguajes de programación, la de los **lenguajes de alto nivel**. En este caso el código vale para cualquier máquina pero deberá ser traducido mediante software traductor que adaptará el código de alto nivel al código máquina correspondiente. Esta traducción es necesaria ya que el código en un lenguaje de alto nivel no se parece en absoluto al código máquina. Tras varios intentos de representar lenguajes, en 1957 aparece el que se considera el primer lenguaje de alto nivel, el **FORTRAN (FORMula TRANslation)**, lenguaje orientado a resolver



1DAM PROG UNIDAD 1. Introducción a la Programación.

fórmulas matemáticas. Por ejemplo la forma en FORTRAN de escribir el texto Hola mundo por pantalla es:

```
PROGRAM HOLA
  PRINT *, '¡Hola, mundo!'
END
```

Nota: *si os gusta el cine, os aconsejo que veáis la película basada en hechos reales **Figuras Ocultas** de 2016, que narra la historia de tres mujeres científicas afroamericanas que trabajaron en la NASA a comienzos de los años sesenta (en plena carrera espacial y en mitad de la lucha por los derechos civiles de los negros estadounidenses) en el ambicioso proyecto de poner en órbita al astronauta John Glenn. La NASA compra ordenadores IBM y ...*



En 1958 se crea **LISP** como lenguaje declarativo para expresiones matemáticas. Programa que escribe Hola mundo en lenguaje LISP:

```
(format t "¡Hola, mundo!")
```

En 1960 en la conferencia **CODASYL** se creó el **COBOL** como lenguaje de gestión. En 1963 se creó **PL/I**, el primer lenguaje que admite la multitarea y la programación modular. En COBOL el programa Hola mundo sería éste (como se ve es un lenguaje más declarativo):

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```



1DAM PROG UNIDAD 1. Introducción a la Programación.

```
MAIN SECTION.  
DISPLAY "Hola mundo"  
STOP RUN.
```

BASIC se creo en el año 1964 como lenguaje de programación sencillo de aprender y es uno de los lenguajes más populares. En 1968 se crea **LOGO** para enseñar a programar a los niños. **Pascal** se creo con la misma idea académica pero siendo ejemplo de lenguaje estructurado para programadores avanzados. El creador del Pascal (**Niklaus Wirdth**) también crea **Modula** en 1977 siendo un lenguaje estructurado para la programación de sistemas (intentando sustituir al **C**). Programa que escribe por pantalla Hola mundo en lenguaje Pascal):

```
PROGRAM HolaMundo;  
BEGIN  
    Writeln( '¡Hola, mundo!' );  
END.
```

LENGUAJES DE CUARTA GENERACIÓN (4GL)

En los años 70 se empezó a utilizar éste término para hablar de lenguajes en los que se minimiza el código y en su lugar aparecen indicaciones sobre qué es lo que el programa debe de obtener. Se consideraba que el lenguaje **SQL** (muy utilizado en bases de datos) y sus derivados eran de cuarta generación. Los lenguajes de consulta de datos, creación de formularios, informes,... son lenguajes de cuarto nivel. Aparecieron con los sistemas de base de datos. Actualmente se consideran lenguajes de éste tipo a aquellos lenguajes que se programan sin escribir casi código (lenguajes visuales), mientras que también se propone que éste nombre se reserve a los lenguajes orientados a objetos.

LENGUAJES ORIENTADOS A OBJETOS.

En los 80 llegan los lenguajes preparados para la programación



1DAM PROG UNIDAD 1. Introducción a la Programación.

orientada a objetos, todos procedentes de **Simula** (1964) considerado el primer lenguaje con facilidades de uso de objetos. De estos destacó inmediatamente **C++**. A partir de **C++** aparecieron numerosos lenguajes que convirtieron los lenguajes clásicos en lenguajes orientados a objetos (y además con mejoras en el entorno de programación, son los llamados lenguajes **visuales**): **Visual Basic**, **Delphi** (versión orientada a objetos de Pascal), **Visual C++**, ...

En 1995 aparece Java como lenguaje totalmente orientado a objetos y en el año 2000 aparece **C#** un lenguaje que toma la forma de trabajar de **C++** y del propio Java. El programa Hola mundo en **C#** sería:

```
using System;
class MainClass {
    public static void Main(){
        Console.WriteLine("¡Hola, mundo!");
    }
}
```

TIPOS DE LENGUAJES DE PROGRAMACIÓN

Según el estilo de programación se puede hacer esta división:

- **Lenguajes imperativos.** Son lenguajes donde las instrucciones se ejecutan secuencialmente y van modificando la memoria del ordenador para producir las salidas requeridas. La mayoría de lenguajes (**C**, **Pascal**, **Basic**, **Cobol**, ...) son de este tipo. Dentro están también los lenguajes orientados a objetos (**C++**, **Java**, **C#**, ...)
- **Lenguajes declarativos.** Son lenguajes que se concentran más en el qué, que en el cómo (cómo resolver el problema es la pregunta a realizarse cuando se usan lenguajes imperativos). Los lenguajes que se programan usando la pregunta ¿qué



queremos? son los declarativos. El más conocido de ellos es el lenguaje de consulta de Bases de datos, **SQL**.

- **Lenguajes funcionales.** Definen funciones, expresiones que nos responden a través de argumentos. Usan expresiones matemáticas, absolutamente diferentes del lenguaje usado por las máquinas. El más conocido de ellos es el **LISP**. **Muchos lenguajes como Java (y otros nuevos como scala, ...) están potenciando esta característica de programación porque sube el nivel en el que se programa.**
- **Lenguajes lógicos.** Utilizan la lógica para producir resultados. El más conocido es el **PROLOG**.

1.6. COMO SE CONSTRUYEN LOS PROGRAMAS.

Hemos mencionado anteriormente que salvo que haya un buen motivo que lo justifique, nadie usa ensamblador y mucho menos código máquina para programar y sin embargo, el código máquina es el único lenguaje que una CPU sabe ejecutar. Actualmente los programas se escriben en **lenguajes de alto nivel**. Un programa escrito y almacenado en disco no es más que un fichero de texto que contiene las sentencias del lenguaje de alto nivel en el que se escribe. A este código se le llama **código fuente**.

A partir de este lenguaje se obtiene el código máquina ejecutable por la CPU, con ayuda de otros programas como traductores y linkadores. Hay diferentes tipos de traductores que clasifican a los lenguajes según el tipo que usen.

INTÉRPRETES

Se convierte cada sentencia de código fuente a código máquina y se ejecuta ese código máquina antes de convertir la siguiente sentencia. De esa forma, si por ejemplo las dos primeras sentencias son



1DAM PROG UNIDAD 1. Introducción a la Programación.

correctas y la tercera tiene un fallo de sintaxis, veríamos el resultado de las dos primeras líneas y al llegar a la tercera se nos notificaría el fallo y finalizaría la ejecución. El intérprete hace una simulación de modo que parece que la máquina entiende directamente las instrucciones del lenguaje, pareciendo que ejecuta cada instrucción (como si fuese código máquina directo). El **BASIC** era un lenguaje interpretado, se traducía línea a línea. Hoy en día la mayoría de los lenguajes integrados en páginas web son interpretados, como **JavaScript** (o incluso, en parte, **Java**) son interpretados. También Python, PHP y muchos otros.

Un programa que se convierte a código máquina mediante un intérprete sigue estos pasos:

- (1) Lee la primera instrucción
- (2) Comprueba que es correcta. Si no lo es informa y para.
- (3) Convierte esa instrucción al código máquina y la ejecuta.
- (4) Lee la siguiente instrucción.
- (5) Vuelve al paso 2 hasta terminar con todas las instrucciones

Ventajas

- Se tarda menos en crear el primer código máquina. El programa se ejecuta antes.
- No hace falta cargar todas las líneas para empezar a ver resultados (idónea para programas que se cargan desde Internet, o para hacer prototipos de aplicaciones).

Desventajas

- El código máquina producido es peor ya que no se optimiza al valorar una sola línea cada vez. El código optimizado permite estudiar varias líneas a la vez para producir el mejor código máquina posible, no es posible con intérpretes.
- Todos los errores son en tiempo de ejecución, no se pueden



1DAM PROG UNIDAD 1. Introducción a la Programación.

detectar antes de lanzar el programa. Esto hace que la depuración de los errores sea más compleja.

- El código máquina resultante gasta más espacio.
- Hay errores difícilmente detectables, ya que para que los errores se produzcan, las sentencias con errores hay que ejecutarlas. Si la sentencia es condicional hasta que no probemos todas las posibilidades del programa, no sabremos todos los errores cometidos.
- Para ejecutar el programa debes usar el traductor, si no lo tienes, no puedes ejecutarlo.

COMPILADORES

Traducen las instrucciones de un lenguaje de programación de alto nivel a código máquina. Analizan todas las líneas del código fuente (análisis léxico, análisis sintáctico, análisis semántico) antes de empezar la traducción y por último incorpora optimizaciones y mejoras). Durante muchos años, los lenguajes potentes han sido compilados. El uso masivo de Internet ha propiciado que esta técnica a veces no sea adecuada y haya lenguajes modernos interpretados o semi-interpretados (se compila en un código intermedio que se interpreta) como **Java** y los lenguajes de la plataforma **.NET** de Microsoft).

Ventajas

- Se detectan errores antes de ejecutar el programa (**errores de compilación**).
- El código máquina generado es más rápido (se optimiza).
- Es más fácil hacer procesos de depuración de código.
- Más estrictos a la hora de programar (crean buenos hábitos).

Desventajas

- El proceso de compilación del código es lento.



1DAM PROG UNIDAD 1. Introducción a la Programación.

- No es útil para ejecutar programas desde Internet ya que hay que descargar todo el programa antes de traducirlo, lo que ralentiza mucho su uso.
- Más estrictos a la hora de programar (pueden ser frustrantes).

FILOSOFÍA QUE SIGUE UN LENGUAJE DE PROGRAMACIÓN

La programación consiste en expresar algoritmos a algún lenguaje de programación. Esta tarea comienza en los años 50 y su evolución a pasado por diversas fases. La programación se puede realizar empleando diversas **técnicas** o métodos. Esas técnicas definen los distintos tipos de lenguajes de programación según las sentencias que ofrecen.

PROGRAMACIÓN DESORDENADA

Se llama así a la programación que se realizaba en los albores de la informática. En este estilo de programación, predomina el instinto del programador por encima del uso de cualquier método, lo que provoca que la corrección y entendimiento de este tipo de programas sea difícil. [Ejemplo \(listado en Basic clásico\):](#)

```
10 X= RANDOM()* 100 + 1;  
20 PRINT "escribe el número que crees que guardo"  
30 INPUT N  
40 IF N > X THEN PRINT "mi numero es menor" GOTO 20  
50 IF N < X THEN PRINT "mi numero es mayor" GOTO 20  
60 PRINT "¡Acertaste!"
```

El código anterior crea un pequeño juego que permite intentar adivinar un número del 1 al 100.

PROGRAMACIÓN ESTRUCTURADA

En esta programación se utiliza una técnica que genera programas que sólo permiten utilizar tres tipos de sentencias:

- **Secuencias** (instrucciones que se ejecutan secuencialmente)



1DAM PROG UNIDAD 1. Introducción a la Programación.

- **Alternativas** (sentencias if)
- **Iterativas** (bucles condicionales o por contador)

El listado anterior en un lenguaje estructurado sería (listado en Pascal):

```
PROGRAM ADIVINANUM;
USES CRT;
VAR
    x,n:INTEGER;
BEGIN
    X= RANDOM() * 100 + 1;
    REPEAT
        WRITE("Escribe el número que crees que guardo");
        READ(n);
        IF (n > x) THEN WRITE("Mi número es menor");
        IF (n < x) THEN WRITE("Mi número es mayor");
    UNTIL n = x;
    WRITE("Acertaste");
END.
```

La ventaja de esta programación está en que es más legible (aunque en este caso el código es más corto en su versión desordenada).

PROGRAMACIÓN MODULAR

Completa la programación estructurada, permitiendo la definición de módulos (trozos de código) independientes, cada uno de los cuales se encargará de una tarea del programa. De este forma el programador se concentra en la codificación de cada módulo, haciendo más sencilla esta tarea. Al final se deben integrar los módulos para dar lugar a la aplicación final. El código de los módulos puede ser invocado en cualquier parte del código. Cada módulo se comporta como un subprograma que, partir de unas determinadas entradas obtienen unas salidas concretas. Su funcionamiento no depende del resto del programa por lo que es más fácil encontrar los errores y realizar el mantenimiento.



PROGRAMACIÓN ORIENTADA A OBJETOS

Se basa en intentar que el código de los programas se parezca lo más posible a la forma de pensar de las personas. Las aplicaciones se representan en esta programación como una serie de objetos independientes que se comunican entre sí. Cada objeto posee datos y métodos propios, por lo que los programadores se concentran en programar independientemente cada objeto y luego generar el código que inicia la comunicación entre ellos.

Es la programación que ha revolucionado las técnicas de programación ya que han resultado un importante éxito gracias a la facilidad que poseen de encontrar fallos, de reutilizar el código y de documentar fácilmente el código.

EJEMPLO 27. Programa en Java.

```
/**
 * Imprime Hola mundo por pantalla
 */
public class Ejemplo27 {

    /** Función principal */
    public static void main(String args[]){
        System.out.print("Hola mundo!");
    }

} // fin de la clase
```

1.7 PRIMER CONTACTO CON JAVA.

1.7.1 LA ORIENTACIÓN A OBJETOS.

En este apartado vamos a comentar los principales conceptos que se utilizan en cualquier lenguaje de programación orientado a objetos (POO) como clase, objeto, ocultación de información, herencia, interfaz, paquete, polimorfismo...



¿QUÉ ES UN OBJETO?

Es la clave para comprender este paradigma de programación. Si miras a tu alrededor encontrarás muchos ejemplos de objetos del mundo real: tu perro, tu mesa, tu televisor, tu bicicleta, ... Los objetos del mundo real tienen dos características: un estado y un comportamiento.

- **Estado:** son características que los describen. Un perro por ejemplo tiene un nombre, un color de pelo, una raza, un peso, su humor. Una bicicleta también tiene un estado: marcha actual, cadencia de pedales, velocidad actual...
- **Comportamiento:** las acciones que son capaces de realizar. Un perro puede comer, mover la cola, ladrar y una bicicleta puede cambiar de marcha, cambiar la cadencia de pedaleo, frenar...

Identificar el estado y el comportamiento de los objetos del mundo real es una forma de pensar en términos de POO.

EJERCICIO 19: Mira a tu alrededor y escoge un objeto del mundo real. Piensa en qué estado y comportamiento podría definirlo.

Habrás observado que en el mundo real hay objetos muy sencillos y otros más complejos. También hay objetos que contienen a otros muchos objetos. Todas estas cuestiones se trasladan del mundo real al mundo de la programación orientada a objetos.

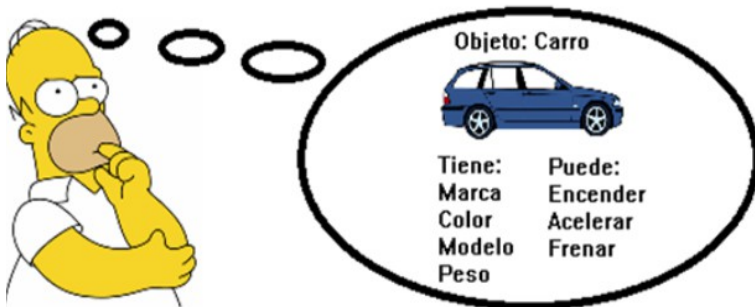


Figura 12: Objeto = estado (atributos) + comportamiento (métodos).



Un objeto software es conceptualmente equivalente a un objeto del mundo real. También tienen un estado y un comportamiento. Un objeto almacena su estado en **variables de instancia** y exponen su comportamiento mediante **métodos (funciones o procedimientos)**. Los métodos de un objeto trabajan con sus variables y sirven para que unos objetos se comuniquen con otros. Ocultando su estado interno, obligan a que interactúe con ellos a través de sus métodos. Este mecanismo de protección se conoce como **encapsulación de datos**.

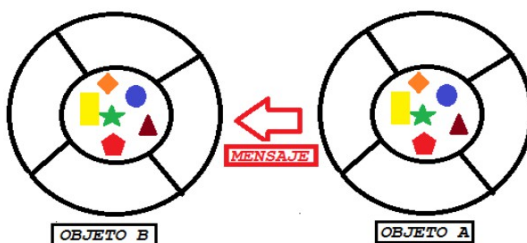


Figura 13: Un objeto encapsula su estado (lo oculta).

Considera una bicicleta. De su estado nos interesa la velocidad actual, la cadencia de pedaleo actual y la marcha que usa y de métodos tendremos los que cambian su estado. Por ejemplo si una bicicleta tiene 6 marchas, de la 1 a la 6, si intentan que tenga una marcha fuera de ese rango, no lo hará. Diseñar los programas de esta forma aporta las siguientes ventajas:

1. **Modularidad:** el código fuente que define un objeto (estado y comportamiento) puede ser escrito y actualizado independientemente del código de otros objetos. Una vez creado puede integrarse fácilmente en cualquier aplicación.
2. **Ocultamiento de información:** al interactuar solamente a través de sus métodos, los detalles internos de la implementación quedan ocultos al exterior.
3. **Reutilización de código:** si un objeto ya existe (quizás lo



1DAM PROG UNIDAD 1. Introducción a la Programación.

haya escrito otro programador), puedes usarlo en tus programas sin tener que volver a programar lo ya programado.

4. Facilidad de depuración e intercambio: si un objeto de tu aplicación da problemas, puedes eliminarlo y cambiarlo por un objeto equivalente de una forma muy sencilla (como en el mundo real, si me falla una lámpara del coche, no tengo que fabricar otro coche, simplemente cambio la lámpara, la pieza que falla).

¿QUÉ ES UNA CLASE?

En el mundo real, si analizas un objeto verás que comparte muchas cosas en común con otros objetos similares. Por ejemplo, si observas dos bicicletas, verás que tienen cosas en común (estado y comportamiento). Cada bicicleta concreta que ves, es como una existencia, un ejemplo o una **instancia** de todos los objetos bicicleta. La idea de bicicleta es como una **plantilla o molde que recoge las características comunes** de todas las bicicletas (estado y comportamiento). Eso es una **clase**, la **plantilla que describe las cosas comunes de todos los objetos que son de esa clase**. **Mientras que un objeto es una existencia real de esa clase.**

Clase:
Coche



♦ **Objeto:** *Ferrari*

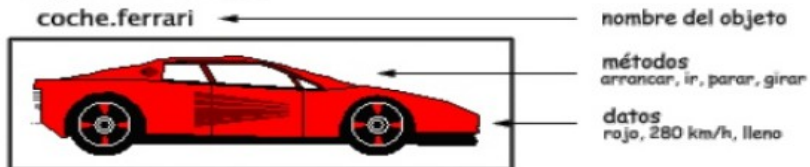


Figura 14: Clase es lo abstracto, objeto es lo concreto.



EJEMPLO 28: En Java, las clases se definen con la palabra **class** seguida de un nombre de clase, el símbolo de comienzo algo "{" y el símbolo de fin de algo "}". Los nombres de las clases, por convención (un acuerdo, algo no forzoso pero que se suele hacer) **comienzan por mayúscula**. Vamos a definir la clase Bicicleta:

```
class Bicicleta {
    int cadencia = 0;
    int velocidad = 0;
    int marcha = 1;

    void cambiaCadencia(int nuevoValor) { cadencia= nuevoValor; }

    void cambiaMarcha(int nuevoValor) { marcha= nuevoValor; }

    void subeVelocidad(int incremento) {
        velocidad= velocidad + incremento;
    }

    void aplicaFrenos(int decremento) {
        velocidad = velocidad - decremento;
    }

    void imprimeEstado() {
        System.out.println("cadencia:" + cadencia +
                           " velocidad:" + velocidad +
                           " marcha:" + marcha );
    }
}
```

EJEMPLO 29: Ahora usamos la clase creada en el ejemplo anterior para crear dos objetos de esa clase y luego usar sus métodos para cambiar el estado. Por cierto, **un programa en Java también es una clase**, nuestro programa se llamará BiciDemo:

```
public class BiciDemo {
    public static void main(String[] args) {
        // Crea dos objetos Bicicleta
        Bicicleta bici1 = new Bicicleta();
        Bicicleta bici2 = new Bicicleta();
    }
}
```



1DAM PROG UNIDAD 1. Introducción a la Programación.

```
// Llamar a los métodos de esos objetos
bici1.cambiaCadencia(50);
bici1.subVelocidad(10);
bici1.cambiaFrenos(2);
bici1.imprimeEstado();
bici2.cambiaCadencia(50);
bici2.cambiaMarcha(3);
bici2.imprimeEstado();
    }
}
```

¿QUÉ ES LA HERENCIA?

Diferentes clases de objetos tienen con frecuencia alguna cantidad de cosas en común con otras. Las bicicletas de montaña, las bicicletas de carretera y las bicicletas de paseo por ejemplo, comparten todas las características de las bicicletas, aunque también tienen cosas que son diferentes (las bicicletas de montaña suelen llevar un plato más para reducir el ratio de giro, las de carretera no tienen guardabarros, las de paseo pueden llevar portaequipajes, ...).

La programación orientada a objetos permite que unas clases puedan heredar las características de otra clase "padre" al mismo tiempo que se puede añadir cosas que la diferencian de la clase padre. La clase que hace de padre se llama **superclase** y la clase que es su hija y hereda su estado y comportamiento se llama **subclase** o **clase derivada** o **clase extendida**. En Java para indicar que una clase es hija de otra, se usa la palabra **extends** y el nombre de la clase padre después de su nombre.

EJEMPLO 30: una clase que hereda de Bicicleta.

```
class BiciMontaña extends Bicicleta {
    // nuevos campos y métodos, porque los de Bicicleta los tiene
}
```

Si creas un objeto de clase BiciMontaña, ya tendrá todo el código y variables de la clase que extiende (los hereda). Esto hace que **la nueva**



clase sea fácil de hacer y entender, pero también obliga a conocer todas sus clases padre.

¿QUÉ ES UNA INTERFACE?

Los objetos definen la interacción con el exterior a través de los métodos que exponen. Estos métodos forman la interfaz del objeto con el mundo exterior. Ocurre igual en el mundo real, por ejemplo, los botones de tu televisor son la interfaz entre tú y los componentes eléctricos del televisor.

En programación orientada a objetos, una interfaz es un grupo de métodos relacionados entre sí, con cuerpos vacíos (sin instrucciones). Por ejemplo, el comportamiento de una bicicleta se puede especificar como una interfaz así:

```
interface Bicicleta {  
    void cambiaCadencia(int nuevoValor);  
    void cambiaMarcha(int nuevoValor);  
    void aumentaVelocidad(int incremento);  
    void aplicaFrenos(int decremento);  
}
```

Para implementar esta interfaz, hay que cambiar el nombre de la clase (no pueden coincidir). Por ejemplo, a *ACMEBicicleta*. Hay que indicar detrás de la clase, la interfaz que quiere implementar. Ejemplo:

```
class ACMEBicicleta implements Bicicleta{  
    int cadencia = 0;  
    int velocidad = 0;  
    int marcha = 1;  
  
    void cambiaCadencia(int nuevoValor) { cadencia= nuevoValor; }  
  
    void cambiaMarcha(int nuevoValor) { marcha= nuevoValor; }  
  
    void subeVelocidad(int incremento) {  
        velocidad= velocidad + incremento;  
    }  
}
```




1DAM PROG UNIDAD 1. Introducción a la Programación.

```
void aplicaFrenos(int decremento) {  
    velocidad = velocidad - decremento;  
}  
  
void imprimeEstado() {  
    System.out.println("cadencia:" + cadencia +  
        " velocidad:" + velocidad +  
        " marcha:" + marcha );  
}
```

Si una clase implementa una interfaz, es como si hiciese una promesa o se comprometiera a implementar ese comportamiento. Este compromiso se vigila en tiempo de compilación, si se incumple, la clase no se compila.

***Nota:** si quisieras compilar la clase `ACMEBicicleta`, necesitas añadir la palabra `public` al principio de los métodos de la clase `ACMEBicicleta`.*

¿QUÉ ES UN Package?

Un package es un espacio de nombres que organiza un conjunto de clases e interfaces relacionadas. Conceptualmente es como las carpetas de un ordenador, podrías dejar todos los documentos .html en una carpeta, los scripts en otra, las imágenes en otra...

Los programas en Java pueden estar formados por centenares o miles de clases, por eso tiene sentido organizarlas almacenando las que estén relacionadas en paquetes.

La plataforma Java ya tiene una enorme librería de clases (un conjunto de paquetes) para que las uses en tus aplicaciones. Esta librería se conoce como la "`Application Programming Interface`", o "**API**". Sus packages contienen las tareas más habituales asociadas a la programación general. Por ejemplo, tiene **String** para cadenas de caracteres. Una clase **File** contiene todo lo que un programador



necesita crear, borrar, insertar, inspeccionar, comparar o modificar un fichero del sistema de ficheros. Un objeto **Socket** permite a un programa comunicarse con otros por la red. Hay varios objetos que son controles GUI, como botones y checkboxes. Como hay cientos de clases ya programadas, un programador que necesite programar un aplicación, si las conoce no tiene que programar un montón de código, ya dispone de él, eso le permite concentrarse en la lógica de su programa, ahorrando tiempo y esfuerzo.

El documento **The Java Platform API Specification** contiene la lista completa de packages, interfaces, classes, campos y métodos aportados en la plataforma Java SE.

1.7.2. EVOLUCIÓN DE JAVA.

En 1991, la empresa Sun Microsystems crea el lenguaje **Oak** (proyecto Green). Mediante este lenguaje se pretendía crear un sistema de televisión interactiva. Este lenguaje sólo se llegó a utilizar de forma interna en la empresa. Su propósito era crear un lenguaje independiente de la plataforma para uso en dispositivos electrónicos. Se intentaba con este lenguaje paliar uno de los problemas fundamentales del C++, que consiste en que al compilar se produce un fichero ejecutable cuyo código sólo vale para la plataforma en la que se realizó la compilación.

Nota: *plataforma se entiende como la combinación de hardware y sistema operativo.*

Sun deseaba un lenguaje para programar pequeños dispositivos electrónicos. La dificultad de estos dispositivos es que cambian continuamente y para que un programa funcione en el siguiente dispositivo, hay que reescribir el código. Por eso Sun quería crear un lenguaje independiente del dispositivo.

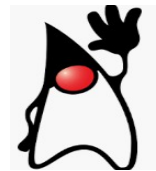


1DAM PROG UNIDAD 1. Introducción a la Programación.

En 1995 Oak pasa a llamarse Java. Java es un importante exportador de café, por eso en EEUU se conoce como Java al café, tomarse una taza de Java es tomarse una taza de café (aunque no sea de Java). Parece que los desarrolladores de Java tomaron muchas tazas. Ese año se da a conocer al público y adquiere notoriedad rápidamente.



Durante estos años se ha mejorado y revisado. La versión 1.2 modificó tanto Java que se la llamó Java 2 y también a sus descendientes (Java 1.3 y Java 1.4). Actualmente la última versión se conoce como Java 11.



Nota: otro logo de Java es Duke, el puntero de ratón que habrás visto alguna vez por ahí.

En general la sintaxis de Java es similar a C y C++. Pero posee estas diferencias:

- No hay punteros (más seguro y fácil de manejar)
- Es totalmente orientado a objetos.
- Muy preparado para ser utilizado en redes TCP/IP y especialmente en Internet.
- Implementa excepciones (control de errores) de forma nativa
- Es un lenguaje interpretado (acelera su ejecución remota aunque genera aplicaciones un poco más lentas).
- Permite múltiples hilos de ejecución, es decir que se ejecuten varias tareas en paralelo.
- Admite firmas digitales
- Tipos de datos y control de sintaxis más rigurosa que los lenguajes C y C++, lo que facilita la gestión de errores.
- Es independiente de la plataforma, ejecutable en cualquier



sistema con máquina virtual.

1.7.3. LA EJECUCIÓN DE PROGRAMAS JAVA.

LA "COMPILACIÓN" EN JAVA

En Java el código fuente no se traduce a código ejecutable. El proceso se conoce como **precompilación** y sirve para producir un archivo (de extensión **.class**) que contiene código que no es código máquina directamente ejecutable. Es un código intermedio llamado **bytecode** (también se le llama **Jcode**). Al no ser ejecutable, el archivo class no puede ejecutarse directamente con un doble clic en el sistema. El bytecode tiene que ser interpretado (es decir, traducido línea a línea) por una aplicación conocida como la **máquina virtual de Java (JVM)**.

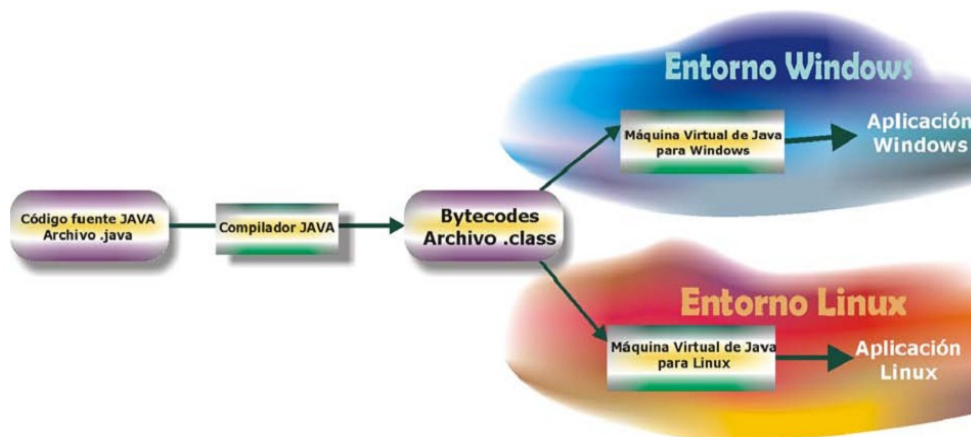


Figura 15: Proceso de generación de aplicaciones con Java.

Si solamente necesitas el intérprete (no desarrollas programas) debes descargar e instalar el **JRE (Java Runtime Environment**, entorno de ejecución de Java) para tu plataforma. La gran ventaja es que el entorno de ejecución de Java se fabrica para todas las plataformas, lo que significa que un archivo class se puede ejecutar en cualquier



1DAM PROG UNIDAD 1. Introducción a la Programación.

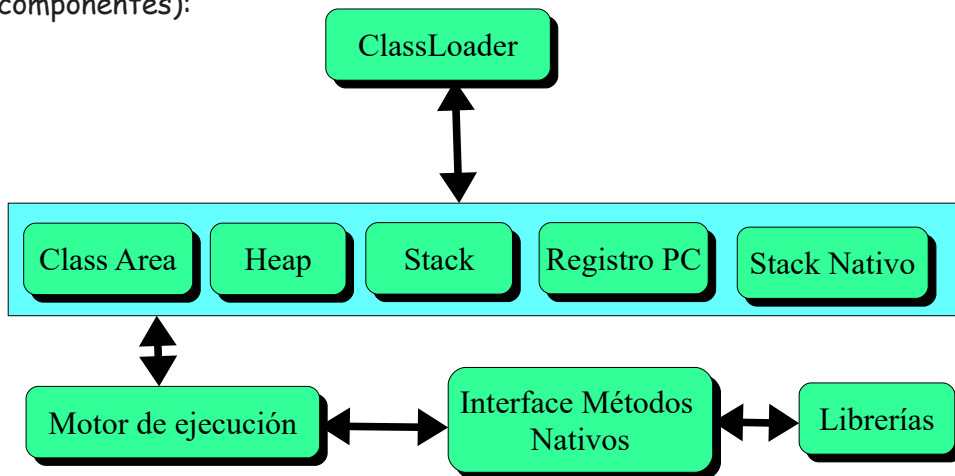
ordenador o máquina que incorpore el JRE. Sólo hay una pega, si programamos utilizando por ejemplo la versión 1.6 de Java, el ordenador en el que queramos ejecutar el programa deberá incorporar el JRE al menos de la versión 1.6. El JRE o la máquina virtual de Java son programas muy pequeños y se distribuyen gratuitamente para prácticamente todos los sistemas operativos.

A esta forma de producir código final de Java se la llama **JIT (Just In Time)**, justo en el momento) ya que el código ejecutable se produce sólo en el instante de ejecución del programa. Es decir, no hay en ningún momento código ejecutable.

LA JVM (JAVA VIRTUAL MACHINE)

Es como un ordenador abstracto (no real, sino virtual). Es una especificación que ofrece un entorno de ejecución en el que los bytecodes pueden ejecutarse.

Cuando está en funcionamiento tiene esta arquitectura (estos componentes):



Esta especificación está implementada en el JRE de cada plataforma



1DAM PROG UNIDAD 1. Introducción a la Programación.

(cada hardware + Sistema Operativo tiene su propio JRE). Cuando quieres ejecutar un programa en Java, se crea un proceso (una instancia) de la JVM.

¿Qué hace la JVM? Carga el código, Lo verifica, Lo ejecuta y le aporta el entorno de ejecución.

El ClassLoader es la parte de la JVM encargada de cargar los ficheros de clases. En realidad tiene 3 cargadores preconstruidos:

1. **Bootstrap classLoader**: Carga el fichero rt.jar que tiene todas las clases de Java como las del paquete java.lang, java.net, java.util, java.sql, etc.
2. **Extension ClassLoader**: Es una clase hija del Bootstrap y padre del System ClassLoader. Carga los ficheros jar del directorio JAVA_HOME/jre/lib/ext.
3. **System ClassLoader**: Carga los ficheros de clases de CLASSPATH. Por defecto configurado a la carpeta actual (se puede cambiar) o puedes indicarlo con la opción -cp o -classpath.

EJEMPLO 30. Vamos a hacer un programa en Java que imprima el nombre de los ClassLoader que lo han ejecutado.

```
public class EjemploClassLoader {  
  
    public static void main(String[] args) {  
        // Imprime nombre del System ClassLoader que carga esta clase  
        Class c = EjemploClassLoader.class;  
        System.out.println( c.getClassLoader() );  
        // El loader que ha cargado la clase String es el Bootstrap  
        System.out.println( String.class.getClassLoader() );  
    }  
}
```

La Class Area es un trozo de memoria donde se guarda una estructura



1DAM PROG UNIDAD 1. Introducción a la Programación.

por cada clase que se cargue en tiempo de ejecución y que contiene el código de los métodos y los datos de la clase, las constantes que se definan...

El Heap es otra zona de memoria que almacena los objetos que se creen durante tiempo de ejecución.

Stack es la zona de memoria que guarda marcos de ejecución. Un marco de ejecución se crea cuando se produce una llamada a un método. En el marco se almacenan las variables locales, los resultados a devolver de un método, los parámetros, la sentencia de código donde volver tras ejecutarse, etc. Cada thread tiene su propia stack (pila).

Registro PC (Contador de programa) contiene la dirección de memoria de la sentencia del programa que tiene la JVM que debe ejecutarse a continuación.

Stack Nativa, Los métodos nativos usados en la aplicación utilizan una pila propia para sus variables en cada proceso.

Execution Engine contiene:

- Una CPU virtual.
- Un intérprete: lee bytecodes y ejecuta las sentencias.
- Un compilador JIT: para mejorar el rendimiento.

TIPOS DE APLICACIONES JAVA

- **applets**: programas Java pensados para ejecutarse dentro de una página web. Actualmente en desuso.
- **Aplicaciones Desktop de consola o GUI**: Programas independientes al igual que los creados con los lenguajes tradicionales que pueden usar un terminal o una interfaz gráfica de usuario para interactuar.
- **Aplicaciones gráficas (GUI)**: Aquellas que utilizan las clases



con capacidades gráficas (como awt por ejemplo).

- **Aplicaciones web:** Aplicaciones que se ejecutan en la parte del servidor y generan de forma dinámica páginas web. Se usan tecnologías como **JSP (Java Server page)**.
- **Aplicaciones Enterprise:** se usan tecnologías como **EJB (Enterprise Java Beans)**, son aplicaciones distribuidas que se ejecutan parte en un servidor de aplicaciones. Ofrecen alto nivel de seguridad, balance de carga, clustering, etc.
- **Midlet:** Aplicación creada con Java para su ejecución en sistemas de propósito simple o dispositivos móviles. Los juegos Java creados para teléfonos móviles son midlets.

PLATAFORMAS DE DESARROLLO

Actualmente hay tres ediciones de Java. Cada una de ellas se corresponde con una plataforma que incluye una serie de funciones, paquetes y elementos del lenguaje (es decir la API, Application Program Interface).

- **Java SE (Java Standard Edition).** Permite escribir código Java relacionado con la creación de aplicaciones y applets en lenguaje Java común. Es decir, es el Java normal. La última versión del kit de desarrollo de aplicaciones en esta plataforma en el momento de escribir estos apuntes es el JSE 12.
- **Java EE (Java Enterprise Edition).** Todavía conocida como J2EE. Pensada para la creación de aplicaciones Java empresariales y del lado del servidor. Su última versión es la 8.
- **Java ME (Java Mobile Edition).** También conocida como J2ME. Pensada para la creación de aplicaciones Java para dispositivos móviles.
- **Java E (Java Embedded):** para aplicaciones para dispositivos tipo IOT, smart-cards, robots, electrodomésticos, etc.



1.8. EL PROCESO DE DESARROLLO Y UML.

Una aplicación (uno o varios programas que implementan una funcionalidad) no se fabrican, como un coche, **se desarrolla**. La aplicación pasa por una serie de pasos relacionados con el ciclo de vida de la aplicación. Pasos:

- (1) Análisis
- (2) Diseño
- (3) Codificación o implementación
- (4) Prueba
- (5) Mantenimiento

Esos pasos imponen un método de trabajo, lo que se conoce como **metodología**. Las metodologías definen los pasos y tareas que hay que realizar para completar un determinado proyecto. Aplicadas a la programación, definen la forma exacta de desarrollar el ciclo de vida de una aplicación. Una metodología marca la forma de realizar todas las fases de creación de un proyecto informático, en especial las relacionadas con el análisis y diseño. Las metodologías marcan los siguientes elementos:

- **Pasos exactos a realizar** en la creación de la aplicación. Se indica de forma exacta qué fases y en qué momento se realiza cada una.
- **Protocolo**. Normas y reglas a seguir escrupulosamente en la realización de la documentación y comunicación en cada fase.
- **Recursos humanos**. Personas encargadas de cada fase y responsables de las mismas.

La metodología a elegir por parte de los desarrolladores es tan importante que algunas metodologías son de pago, pertenecen a empresas que poseen sus derechos. En muchos casos, estas empresas



fabrican las herramientas (**CASE**) que permiten gestionar de forma informática el seguimiento del proyecto mediante la metodología empleada.

FASES EN LAS METODOLOGÍAS

Todas las metodologías imponen una serie de pasos o fases a realizar en el proceso de creación de aplicaciones, lo que ocurre es que estos pasos varían de unas a otras. En general los pasos siguen el esquema inicial planteado en el punto anterior, pero varían los pasos para recoger aspectos que se consideran que mejoran el proceso. La mayoría de metodologías se diferencian fundamentalmente en la parte del diseño. De hecho algunas se refieren sólo a esa fase (como las famosas metodologías de **Jackson** o **Warnier**). Por ejemplo la metodología **OMT** de **Rumbaugh** propone estas fases:

- Análisis
- Diseño del sistema
- Diseño de objetos
- Implementación

Naturalmente con esto no se completa el ciclo de vida, para el resto sería necesario completar el ciclo con otro método. En la metodología **MERISE** desarrollada para el gobierno francés (y muy utilizada de forma docente en las universidades españolas), se realizan estas fases:

- Esquema Director
- Estudio Previo
- Estudio Detallado
- Estudio Técnico
- Realización
- Mantenimiento

Otra propuesta de método indicada en varios libros:

- Investigación Preliminar



1DAM PROG UNIDAD 1. Introducción a la Programación.

- Determinación de Requerimientos.
- Diseño del Sistema
- Desarrollo del Software
- Prueba del Sistema
- Implantación y Evaluación

En definitiva las fases son distintas pero a la vez muy similares ya que el proceso de creación de aplicaciones siempre tiene pasos por los que todas las metodologías han de pasar.

Independientemente de la metodología utilizada, siempre ha habido un problema al utilizar metodologías. El problema consiste en que desde la fase de análisis, el cliente no ve la aplicación hasta que esté terminada, independientemente de si hay fallos en una fase temprana o no.

Diseño

En esta fase se crean esquemas que simbolizan a la aplicación. Estos esquemas los elaboran analistas. Gracias a estos esquemas se facilita la implementación de la aplicación. Estos esquemas en definitiva se convierten en la documentación fundamental para plasmar en papel lo que el programador debe hacer. En estos esquemas se pueden simbolizar: la organización de los datos de la aplicación, el orden de los procesos que tiene que realizar la aplicación, la estructura física (en cuanto a archivos y carpetas) que utiliza la aplicación, etc.

Cuanto más potentes sean los esquemas utilizados (que dependerán del modelo utilizado), más mecánica y fácil será la fase de implementación. La creación de estos esquemas se puede hacer en papel (raramente), o utilizar una herramienta CASE para hacerlo.

Nota: Las herramientas CASE (Computer Aided Software Engineering, Ingeniería de Software Asistida por Ordenador) son herramientas software pensadas para facilitar la



1DAM PROG UNIDAD 1. Introducción a la Programación.

realización de la fase de diseño en la creación de una aplicación. Con ellas se realizan todos los esquemas necesarios en la fase de diseño e incluso son capaces de escribir el código equivalente al esquema diseñado.

Mediante este diseño el problema se divide en módulos, que, a su vez, se vuelven a dividir a fin de solucionar problemas más concretos. Al diseño descendente se le llama también **top-down**. Gracias a esta técnica un problema complicado se divide en pequeños problemas que son más fácilmente solucionables. Es el caso de las metodologías de Warnier y la de Jackson. Hoy en día en esta fase se suelen utilizar modelos orientados a objetos como la metodología OMT o las basadas en **UML**. Durante el diseño se suelen realizar estas fases:

- (1) **Elaborar el modelo funcional.** Diseñar el conjunto de esquemas que representan el funcionamiento de la aplicación.
- (2) **Elaborar el modelo de datos.** Se diseñan los esquemas que representan la organización de los datos. Pueden ser esquemas puros de datos (sin incluir las operaciones a realizar sobre los datos) o bien esquemas de objetos (que incluyen datos y operaciones).
- (3) **Creación del prototipo del sistema.** Creación de un prototipo que simbolice de la forma más exacta posible la aplicación final. A veces incluso en esta fase se realiza más de un prototipo en diferentes fases del diseño. Los últimos serán cada vez más parecidos a la aplicación final.
- (4) **Aprobación del sistema propuesto.** Antes de pasar a la implementación del sistema, se debe aprobar la fase de diseño (como ocurre con la de análisis).

UML

UML es la abreviatura de **Universal Modelling Language** (Lenguaje de Modelado Universal). No es una metodología, sino una forma de



1DAM PROG UNIDAD 1. Introducción a la Programación.

escribir esquemas con pretensiones de universalidad (que ciertamente ha conseguido). La idea es que todos los analistas y diseñadores utilizaran los mismos esquemas para representar aspectos de la fase de diseño.

UML no es una metodología sino una forma de diseñar (de ahí lo de lenguaje de modelado) el modelo de una aplicación. Su vocación de estándar está respaldada por ser la propuesta de OMG (Object Management Group) la entidad encargada de desarrollar estándares para la programación orientada a objetos. Lo cierto es que sí se ha convertido en un estándar debido a que une las ideas de las principales metodologías orientadas a objetos: OMT de Rumbaugh, OOD de Grady Booch y Objectory de Jacobson (conocidos como Los Tres Amigos). Los objetivos de UML son:

- (1) Poder incluir en sus esquemas todos los aspectos necesarios en la fase de diseño.
 - (2) Facilidad de utilización.
 - (3) Que no se preste a ninguna ambigüedad, es decir que la interpretación de sus esquemas sea una y sólo una.
 - (4) Que sea soportado por multitud de herramientas CASE.
 - (5) Usable independientemente de la metodología utilizada.
- Evidentemente las metodologías deben de ser orientadas a objetos.

PROCESO UNIFICADO

Los tres amigos llegaron a la conclusión de que las metodologías clásicas no estaban preparadas para asimilar las nuevas técnicas de programación. Por ello definieron el llamado Proceso Unificado de Rational. Rational es una empresa dedicada al mundo del Análisis. De hecho es la empresa en la que recabaron Rumbaugh, Booch y Jacobson (actualmente pertenece a IBM).



1DAM PROG UNIDAD 1. Introducción a la Programación.

Dicha empresa fabrica algunas de las herramientas CASE más populares (como **Rational Rose** por ejemplo). En definitiva el proceso unificado es una metodología para producir sistemas de información.

Los modelos que utiliza para sus diagramas de trabajo, son los definidos por UML. Es una metodología iterativa y por incrementos, de forma que en cada iteración los pasos se repiten hasta estar seguros de disponer de un modelo UML capaz de representar el sistema correctamente.

DIAGRAMAS UML

Lo que realmente define UML es la forma de realizar diagramas que representen los diferentes aspectos a identificar en la fase de diseño. Actualmente estamos en la versión 2.1.1 de UML, aunque el UML que se utiliza mayoritariamente hoy en día sigue siendo el primero. Los diagramas a realizar con esta notación son:

- **Diagramas que modelan los datos.**
 - Diagrama de clases. Representa las clases del sistema y sus relaciones.
 - Diagrama de objetos. Representa los objetos del sistema.
 - Diagrama de componentes. Representan la parte física en la que se guardan los datos.
 - Diagrama de despliegue. Modela el hardware utilizado en el sistema.
 - Diagrama de paquetes. Representa la forma de agrupar en paquetes las clases y objetos del sistema.
- **Diagramas que modelan comportamiento** (para el modelo funcional del sistema).
 - Diagrama de casos de uso. Muestra la relación entre los usuarios (actores) y el sistema en función de las posibles situaciones (casos) de uso que los usuarios hacen.
 - Diagrama de actividades. Representa los flujos de trabajo



1DAM PROG UNIDAD 1. Introducción a la Programación.

del programa.

- Diagrama de estados. Representa los diferentes estados por los que pasa el sistema.
- Diagrama de colaboración. Muestra la interacción que se produce en el sistema entre las distintas clases.
- Diagramas de secuencia. Muestran la actividad temporal que se produce en el sistema. Representa como se relacionan las clases, pero atendiendo al instante en el que lo hacen.

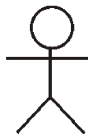
Cabe decir que esto no significa que al modelar un sistema necesitemos todos los tipos de esquema. A continuación veremos dos tipos de diagramas UML a fin de entender lo que aporta UML para el diseño de sistemas. Sin duda el diagrama más importante y utilizado es el de clases (también el de objetos).

DIAGRAMA DE CASOS DE USO

Este diagrama es sencillo y se suele utilizar para representar la primera fase del diseño. Con él se detalla la utilización prevista del sistema por parte del usuario, es decir, los casos de uso del sistema.

En estos diagramas intervienen dos protagonistas:

- **Actores.** Siempre son humanos (no interesa el teclado, sino quién le golpea) y representan usuarios del sistema. La forma de representarlos en el diagrama es ésta:



- **Casos de uso.** Representan tareas que tiene que realizar el sistema. A dichas tareas se les da un nombre y se las coloca dentro de una elipse.

El diagrama de casos de uso representa la relación entre los casos de



uso y los actores del sistema. Fuera del sistema están los actores y dentro del sistema los casos de uso que el sistema tiene que proporcionar. Este tipo de diagramas se suele crear en la toma de requisitos (fase de análisis) con el fin de detectar todos los casos de uso del sistema (Identificar que funcionalidad debe implementar la aplicación).

Después, cada caso de uso se puede subdividir (diseño descendente) en esquemas más simplificados.

EJEMPLO 31: Un diagrama de casos de uso:

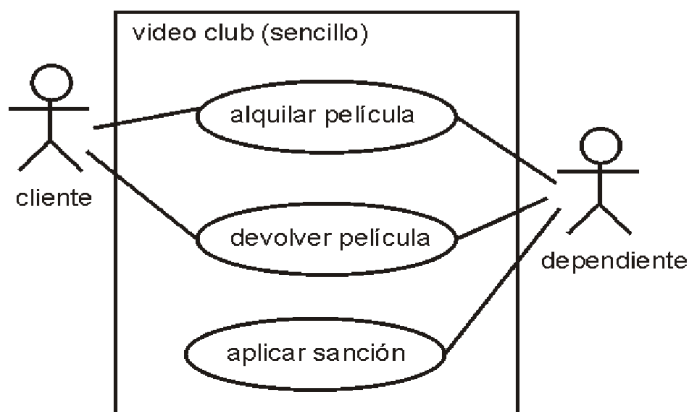


Figura 16: el rectángulo representa el sistema a crear (video club).

También se debe describir el caso de uso como texto para explicar exactamente su funcionamiento. En un formato como éste:



1DAM PROG UNIDAD 1. Introducción a la Programación.

Nombre del caso de uso:	Alquilar película
Autor:	Fulanito
Fecha:	14/10/2007
Descripción:	Permite realizar el alquiler de la película
Actores:	Cliente, dependiente
Precondiciones:	El cliente debe de ser socio
Funcionamiento normal:	<ol style="list-style-type: none">1) El cliente deja la película en el mostrador junto con su carnet2) El dependiente anota en el sistema el número de carnet para comprobar sanciones3) Sin sanciones, se anota la película y el alquiler se lleva a cabo4) El sistema calcula la fecha de devolución y el dependiente se lo indica al cliente
Funcionamiento alternativo	Si hay sanción, no se puede alquilar la película
Postcondiciones	El sistema anota el alquiler

Figura 17: descripción detallada de un caso de uso (video club).

DIAGRAMAS DE ACTIVIDAD

Hay que tener en cuenta que todos los diagramas UML se basan en el uso de clases y objetos. Representa el funcionamiento de una determinada tarea del sistema (normalmente un caso de uso).

En realidad, el diagrama de estado y el de actividad son muy parecidos, salvo que cada uno se encarga de representar un aspecto del sistema. Por ello si se conoce la notación de los diagramas de actividad es fácil entender los diagramas de estado. Elementos de los diagramas de actividad



1DAM PROG UNIDAD 1. Introducción a la Programación.







Actividad: 	Representada con un rectángulo de esquinas redondeadas dentro del cual se pone el nombre de la actividad. Cada actividad puede representar varios pasos y puede iniciarse tras la finalización de otra actividad.
Transición: 	Representada por una flecha, indican el flujo de desarrollo de la tarea. Es decir unen la finalización de una actividad con el inicio de otra (es decir, indican qué actividad se inicia tras la finalización de la otra)
Barra de sincronización: 	Barra gruesa de color negro. Sirve para coordinar actividades. Es decir si una actividad debe de iniciarse tras la finalización de otras, la transición de las actividades a finalizar irán hacia la barra y de la barra saldrá una flecha a la otra actividad.
Diamante de decisión: 	Representado por un rombo blanco, se utiliza para indicar alternativas en el flujo de desarrollo de la tarea según una determinada condición
Creación: 	Indican el punto en el que comienza la tarea
Destrucción: 	Indica el punto en el que finaliza el desarrollo del diagrama

Figura 18: Símbolos de los diagramas de actividades (video club).

Ejemplo de diagrama de actividad (para representar el funcionamiento del alquiler de una película del videoclub):

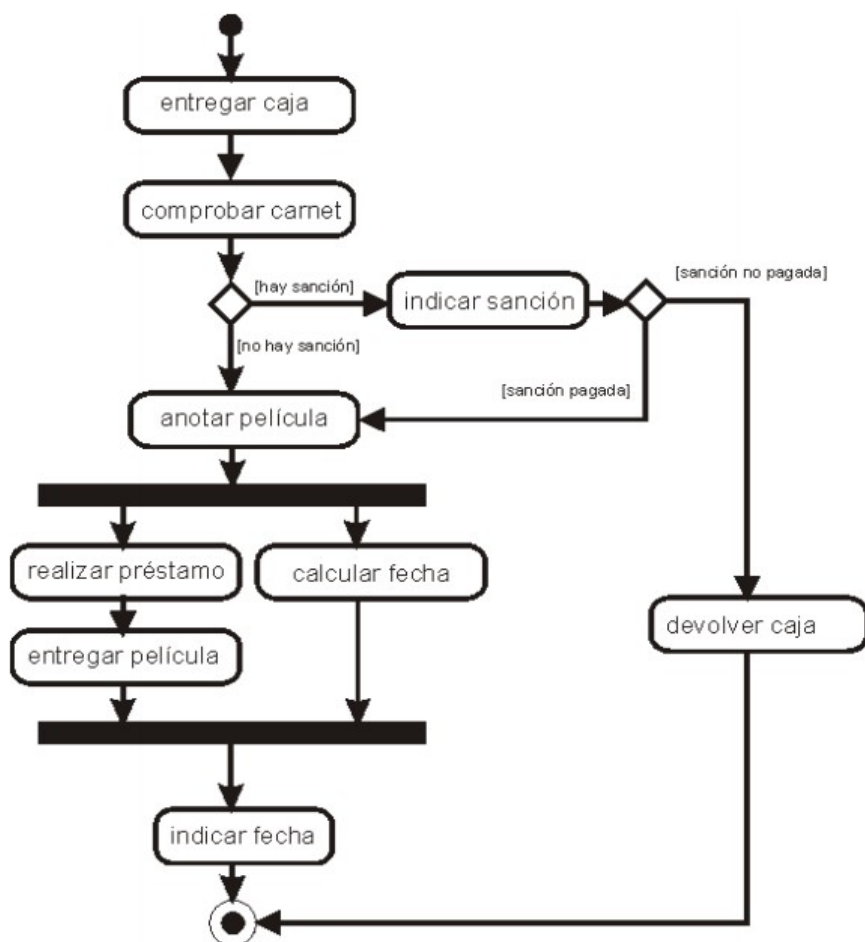


Figura 19: Diagrama de actividad de alquilar película (video club).



1.9. EJERCICIOS.

E1. Lee la base y la altura de un triángulo y muestra el área:

$$\text{área} = \text{base} \times \text{altura} / 2$$

- a) Escribe el pseudocódigo usando pseint.
- b) Genera el diagrama de flujo usando pseint.

E2. Crea un algoritmo llamado comparar que lea dos números enteros positivos $n1$ y $n2$. Si el primero es mayor ($n1 > n2$) escribir 1, si el segundo es mayor ($n2 > n1$) escribir 2 y si son iguales escribir 0.

- a) Escribe el pseudocódigo del algoritmo usando pseint.
- b) Genera el diagrama de flujo usando pseint.

E3. Crea un algoritmo llamado producto que lea dos números ($n1$ y $n2$) enteros positivos y calcule y muestre su multiplicación sin usar el operador de multiplicar (*). Pista: $n1 \times n2$ es sumar $n2$ veces el número $n1$.

- a) Escribe el pseudocódigo del algoritmo usando pseint.
- b) Genera el diagrama de flujo usando pseint.

E4. Convierte el código del ejercicio E3 en un subalgoritmo de tipo función de nombre producto. Luego crea un algoritmo que lea la base y la altura de un triángulo y calcule y muestre su área usando la función producto: $\text{area} = \text{base} * \text{altura} / 2$

- a) Escribe el pseudocódigo del algoritmo usando pseint.
- b) Genera el diagrama de flujo usando pseint.

E5. Leer un n° llamado n y aproximar su raíz cuadrada (sin decimales) probando desde el 0 hasta el número r cuyo cuadrado no exceda a n , podría tener el siguiente algoritmo:



1DAM PROG UNIDAD 1. Introducción a la Programación.

```
PASO 0: Leer n
PASO 1: r <- 0
PASO 2: Si r * r es n
    entonces
        escribo r
    sino
        si r * r es menor que n
            r <- r + 1
            Ir al paso 2
        sino
            escribir la raíz aproximada es r-1
        fin si
    fin si
```

- a) Adapata a la programación estructurada el algoritmo y escribe un pseudocódigo usando pseint.
- b) Genera el diagrama de flujo usando pseint.

E6. Crea un algoritmo llamado leerVarios, que no deje de leer repetidamente números enteros positivos y cuente los que no sean cero. Deja de leer números cuando lea el primer 0. Al final escribe cuantos números distintos de cero ha leído.

- a) Escribe el pseudocódigo del algoritmo usando pseint.
- b) Genera el diagrama de flujo usando pseint.

E7. Leer entero n y calcular su sumatorio: $1 + 2 + 3 + \dots + n$

$$\text{Sumatorio}(n) = \sum_{i=1}^n i$$

E8. Repite el ejercicio 7 pero implementa un subalgoritmo (una función) que acepte n de parámetro y devuelva el sumatorio.

E9. Repite el ejercicio 8 pero transforma la función en recursiva:

$$\text{sumatorio}(n) = \begin{cases} 0 & \text{si } n = 0 \\ n + \text{sumatorio}(n-1) & \text{en otro caso} \end{cases}$$



E10. En una línea de autobús, el billete tiene ciertos descuentos según la edad del viajero. Si el viajero es un niño menor de 3 años, el descuento es del 100% (no pagan). Si es menor de 12 años tienen un 30% de descuento. Si son mayores de 65 tienen un 40%. Un programador piensa el siguiente algoritmo para calcular el precio a pagar por un billete en sus autobuses:

```
Leer precio // Es el precio del viaje
Leer edad
si edad menor de 3 entonces descuento es 100% fin si
si edad < 12 entonces descuento es 30% fin si
si edad > 65 entonces descuento es 40% fin si
Escribir "debes pagar ", precio menos el % aplicado
Escribir "Aplicado un ", descuento%
```

Debes implementarlo en pseint, e intenta mejorarlo (que cuando pregunte indique lo que quiere leer, que declare los datos que se usen y su tipo, que cada sentencia acabe con un punto y coma, indenta las sentencias, verifica que funcione en todos los casos, etc.)

E11. ¿Qué fallo ha cometido el programador de este meme?

