

Fundamentos de Informática y metodología de la programación

FORMADORES { IT }

info@formadoresfreelance.es · www.formadoresit.es



Ayuntamiento de
FUENLABRADA

Índice

Fundamentos de Informática.....	5
Conceptos generales	5
Informática e información	5
El ordenador	6
Componentes del ordenador.....	7
Unidad central de proceso (CPU)	8
Memoria.....	9
Dispositivos de entrada/salida.....	11
Teclado.....	11
Ratón	11
Scanner	12
Cámara digital.....	12
Pantalla o monitor.....	12
Impresora	13
Discos magnéticos.....	13
Pen drive o memoria USB	13
CD.....	14
DVD.....	14
El sistema operativo.....	15
Codificación de la información.....	19
Sistema binario	19
Conversión binario a decimal	19
Conversión decimal a binario	20
Sistema hexadecimal.....	21
Medidas de información.....	22
Codificación de la información	23
Código ASCII.....	23
Código Unicode	25
Introducción a la programación.....	26
Programas y algoritmos.....	26

Lenguajes de programación.....	27
Tipos de lenguajes de programación.....	28
Lenguajes de bajo nivel.....	28
Lenguajes de nivel intermedio.....	29
Lenguajes de alto nivel.....	29
Ensambladores, compiladores e intérpretes.....	32
Ensambladores.....	33
Compilador.....	33
Intérpretes.....	34
Linkado.....	35
Entornos de desarrollo integrados.....	37
Netbeans.....	37
Eclipse.....	38
Visual Studio.NET.....	38
Fases en el desarrollo de un programa.....	39
Fase de análisis.....	40
Fase de diseño.....	40
Fase de implementación.....	40
Fase de prueba.....	41
Fase de mantenimiento.....	41
Fase de documentación.....	41
Diseño de algoritmos.....	42
Datos y variables.....	42
Diagramas de flujo.....	44
Símbolo de principio y fin.....	46
Símbolo de proceso.....	46
Líneas de flujo.....	47
Símbolo de entrada/salida.....	47
Símbolo de decisión.....	48
Acumuladores y contadores.....	51
Estructuras repetitivas.....	55
Pseudocódigo.....	58

Normas en la creación de pseudocódigos.....	62
Seguimiento de algoritmos.....	64
Programación estructurada.....	70
Teorema de la estructura.....	70
Ventajas de la programación estructurada.....	70
Tipos de instrucciones.....	71
Instrucciones secuenciales.....	71
Instrucciones alternativas.....	72
Alternativas simples.....	72
Alternativa múltiple.....	73
Instrucciones repetitivas.....	76
Otros elementos de programación estructurada.....	79
Tipos de datos.....	79
Operadores.....	80
Comentarios.....	80
Ejercicios de ejemplo.....	81
Ejercicio 1.....	81
Ejercicio 2.....	81
Ejercicio 3.....	82
Ejercicio 4.....	83
Ejercicio 5.....	83
Arrays.....	84
Definición de un array.....	84
Acceso a los elementos del array.....	85
Ejercicios de ejemplo.....	85
Ejercicio 1.....	85
Ejercicio 2.....	87
Arrays multidimensionales.....	88
Ejercicio de ejemplo.....	90
Ordenación de arrays.....	92
Ejercicio resuelto.....	94
Programación Modular.....	96

Ventajas de la programación modular.....	96
Procedimientos y funciones.....	97
Procedimientos.....	98
Funciones.....	100
Programación por capas.....	102
Ejercicio resuelto.....	104
Programación Orientada a Objetos.....	106
Clases y objetos.....	106
Objetos.....	106
Clases.....	108
Creación de objetos.....	110
Constructores.....	113
Ejercicio ejemplo.....	114
Otras características de la POO.....	117
Sobrecarga de métodos.....	117
Herencia.....	118
Herencia en pseudocódigo.....	118
Sobrescritura de métodos.....	120

Conceptos generales

Informática e información

El uso eficaz de la **información** para la empresa resulta determinante, no solo para sus beneficios, sino incluso para su propia supervivencia. La manera en la que se van a tratar y manipular los **datos** resulta pues crucial.

En este contexto surge el concepto de **informática**, que hoy en día puede considerarse como la Ciencia que estudia el tratamiento, elaboración, transmisión y utilización de la información. La informática estudia el tratamiento automático de la información utilizando dispositivos electrónicos y sistemas computacionales.

La informática está estrechamente a los **computadores u ordenadores**, entendiendo como tal un sistema electrónico capaz de tratar y manipular la información, procesando los datos de manera rápida y segura.

Hoy en día los computadores se han convertido en una herramienta indispensable para la vida actual, pues la mayor parte de los aparatos que utilizamos hoy en día (ordenadores, teléfonos móviles, televisores, etc.) cuentan con capacidad de computación.

En informática aparecen involucrados diferentes facetas de las ciencias de la computación, como la programación y las metodologías para el desarrollo de software, la arquitectura de computadores, las redes de datos (como Internet) y la inteligencia artificial.

El ordenador

La aparición del ordenador y con él de la informática en la década de los años 50 estableció nuevas bases en el tratamiento y manipulación de la información, lo que ha permitido llegar a los avances tecnológicos que conocemos hoy en día.

Un ordenador es una máquina que permite automatizar el tratamiento de la información. Por si solo, el ordenador no toma decisiones, carece de inteligencia, simplemente se encarga de realizar las tareas para las que se le ha programado. Estas tareas son definidas en lo que se conoce como un programa.

Los programas establecen las tareas que el ordenador debe realizar con los datos. Básicamente, estas tareas se definen en forma de operaciones aritméticas, lógicas, almacenamiento y recuperación de datos. Los programas, conocidos también como aplicaciones, permiten a los usuarios de un ordenador realizar tareas complejas de forma sencilla y rápida que de otra manera serían tediosas o imposibles de realizar, como realizar cálculos y manipulaciones de grandes cantidades de datos, procesar textos e imágenes, almacenar y gestionar datos, etc.

En líneas generales, podemos decir que la misión de un ordenador es recoger unos datos de entrada, procesarlos en función de las instrucciones definidas por un programa y generar unos resultados legibles para el usuario.



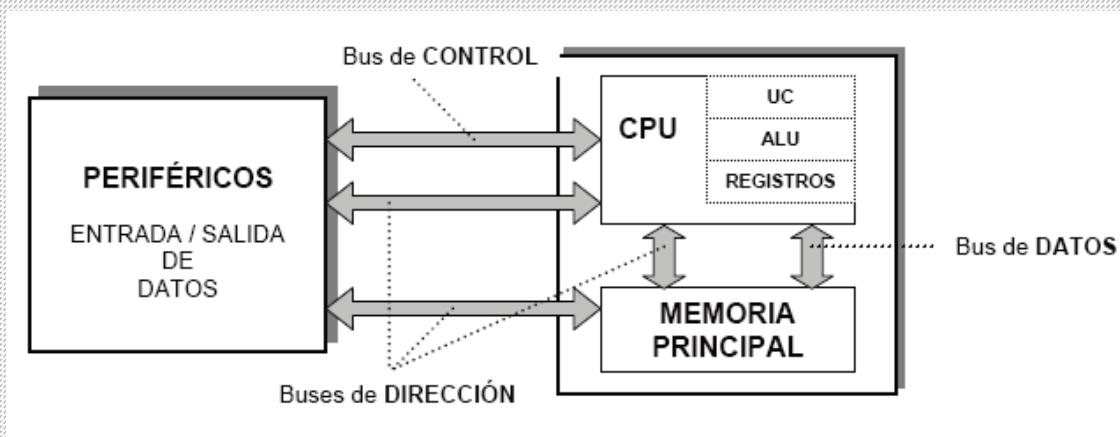
Componentes del ordenador

El ordenador está formado por dos tipos de componentes:

- **Hardware.** Es la parte física, que la componen la CPU, la memoria y los dispositivos periféricos.
- **Software.** La forman los elementos lógicos, como el sistema operativo y los programas.

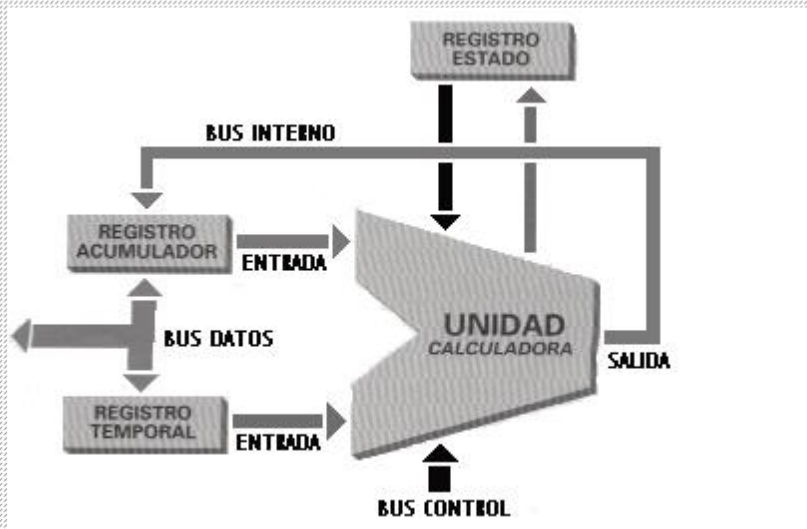
Unidad central de proceso (CPU)

Es el verdadero cerebro de la computadora. Su misión consiste en coordinar y controlar o realizar todas las operaciones del sistema. Se comunica con los otros componentes hardware a través de los buses o canales por donde fluye la información de un bloque a otro.



La CPU está formada por:

- **La unidad de control.** Como su nombre indica, es el centro de control del ordenador, desde ella se localizan y se interpretan las instrucciones a ejecutar con las correspondientes operaciones que conllevan. También gestiona la recogida de datos necesarios para la realización de operaciones, datos que son entregados a la unidad aritmético-lógica para que opere con ellos, así como el almacenamiento del resultado de dichas operaciones.
- **La unidad aritmético-lógica.** Realiza las operaciones elementales de tipo aritmético y lógico. Su función es ejecutar la operación sobre los datos que recibe de la unidad de control. Esta unidad se compone a su vez de dos partes:
 - Banco de registros generales, que es donde se almacenan los datos que vienen de la unidad de control. Uno de estos registros es el acumulador, donde se van acumulando los resultados de las operaciones y otro es el registro temporal, que como su nombre indica, almacena de forma temporal datos de entrada.
 - Operador. Es el encargado de realizar las operaciones. El operador entrega al acumulador el resultado de las operaciones.



Memoria

La memoria o memoria principal es un componente hardware que almacena las instrucciones que debe procesar la CPU así como los datos generados por esta durante el procesamiento de las instrucciones de un programa.

La memoria principal se divide en dos tipos:

- **Memoria RAM (Random Access Memory):** Esta memoria es de tipo volátil, lo que significa que la información que contiene solo permanece en ella mientras el computador está encendido. Su contenido puede ser leído y modificado por la CPU. Es de acceso aleatorio porque la CPU puede acceder directamente a cualquier posición de la misma, sin tener que recorrerla de forma secuencial. En ella se cargan partes de los programas que tienen que ser ejecutados por la CPU.

Una variante de la memoria RAM es la **memoria cache**. Se trata de una parte de la RAM a la que la CPU puede acceder de forma especialmente rápida. Está destinada a almacenar instrucciones y datos de uso común por parte de la CPU.

- **Memoria ROM (Read Only Memory):** Memoria solo de lectura, su contenido no puede ser modificado y permanece almacenado de forma permanente, incluso después de apagar el computador. Normalmente, esta memoria contiene instrucciones básicas del sistema operativo del equipo, es decir, aquellas que son fundamentales para que el ordenador

pueda funcionar, entre ellas las conocidas como BIOS (Basic Input-Output System) o instrucciones básicas de control de dispositivos.

La Memoria ROM se encuentra fijada a la placa base del ordenador, mientras que la memoria RAM se compone de uno o varios chips insertables, lo que en muchos ordenadores permite que la capacidad de este tipo de memoria pueda ser ampliable.



**Circuito de
memoria ROM**



Memoria RAM
(Expandible e intercambiable)

Pregunta de refuerzo

Las operaciones sobre los datos manejados por un programa son realizadas por:

- a) La memoria RAM
- b) La memoria cache
- c) La CPU
- d) La memoria ROM

Respuesta: La respuesta es la c. Todas las operaciones aritmético lógicas son realizadas por la Unidad aritmético-lógica, que es uno de los bloques que forman la CPU. Los diferentes tipos de memoria sirven para almacenar datos, pero no para operar con ellos.

Actividad práctica 0. Arquitecturas de ordenadores

Dispositivos de entrada/salida

Son aquellos que comunican el computador con el exterior y se comunican con la CPU y la memoria a través de los buses de entrada/salida.

Por un lado, tenemos los dispositivos de entrada, que permiten al usuario introducir datos, comandos y programas en el ordenador. El dispositivo de entrada más común es un teclado, a través del cual el usuario teclea la información, que es convertida en señales eléctricas binarias (1 y 0) que se almacenan en la memoria central.

Por otro lado están los dispositivos de salida, que permiten al usuario ver los resultados de los cálculos o de las manipulaciones de datos realizados por el ordenador. El dispositivo de salida más común es el monitor, que presenta los caracteres y gráficos en una pantalla similar a la de un televisor.

A continuación, te presentamos algunos de los dispositivos de entrada y salida más utilizados en la actualidad:

Teclado

El teclado es un dispositivo eficaz para introducir datos no gráficos, como aquellos que nos solicitan los programas para su funcionamiento. También dispone de teclas para realizar funciones especiales.



Ratón

Este dispositivo nos permite dar instrucciones a nuestra computadora a través de un cursor que aparece en la pantalla mediante la realización de un clic para que se lleve a cabo una acción determinada; a medida que el Mouse rueda sobre el escritorio, el cursor (Puntero) en la pantalla hace lo mismo. El ratón se conecta a uno de los puertos USB del ordenador (conectores universales) a través de un cable, aunque también existen los ratones ópticos que se comunican con señales ópticas y no requieren cableado.



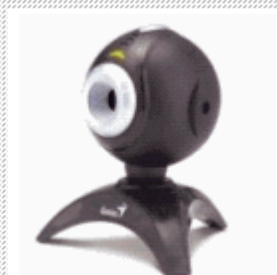
Scanner

Es dispositivo de entrada de información. Permite la introducción de imágenes gráficas al computador mediante un sistema de matrices de puntos, como resultado de un barrido óptico del documento. La información se almacena en archivos en forma de mapas de bits (bit maps), o en otros formatos más eficientes como Jpeg o Gif.



Cámara digital

Es otro dispositivo de entrada que transmite al ordenador las imágenes que capta. Un ejemplo de este tipo de dispositivos es la Webcam, que se trata de una cámara de pequeñas dimensiones y que tiene que estar conectada al PC para. Su uso es generalmente para videoconferencias por Internet, pero mediante el software adecuado, se pueden grabar videos como una cámara normal y tomar fotos estáticas.



Pantalla o monitor

Permite visualizar la información suministrada por el ordenador. En el caso más habitual se trata de un aparato basado en un tubo de rayos catódicos (CRT) como el de los televisores, mientras que en los portátiles y en ordenadores actuales es una pantalla plana de cristal líquido (LCD).



Impresora

Se utiliza para presentar información impresa en papel. Actualmente se utilizan impresoras de tinta e impresoras de toner.



Discos magnéticos

Un disco magnético está constituido por una superficie metálica, recubierta por una capa de un material magnetizable. Los datos se almacenan cambiando el sentido del campo magnético de dicha sustancia, y una cabeza de lectura y grabación por cada superficie de disco (actualmente los discos duros vienen en paquetes de varios platos), esta cabeza está conformada por un electroimán que puede inducir un campo magnético o detectar el sentido del cambio magnético mientras que el disco gira en un sentido. La información se almacena en pistas concéntricas que a su vez se dividen en sectores que a su vez se dividen en bloques



Pen drive o memoria USB

Una memoria USB (de Universal Serial Bus, en inglés pendrive o USB flash drive) es un pequeño dispositivo de almacenamiento que se conecta al ordenador a través del puerto USB y permite almacenar información de manera permanente. Actualmente, se pueden alcanzar con este tipo memorias capacidades del orden de varias decenas de gigabytes.



CD

El Compact Disc (CD) es un soporte digital óptico utilizado para almacenar cualquier tipo de información (audio, vídeo, documentos y otros datos). En un CD la información se almacena en formato digital, es decir, utiliza un sistema binario para guardar los datos.

**DVD**

Son discos que utilizan un formato de almacenamiento óptico que puede ser usado para guardar datos, incluyendo películas con alta calidad de vídeo y audio. A diferencia de los CD, todos los DVD guardan los datos utilizando un sistema de archivos denominado UDF (Universal Disk Format), el cual es una extensión del estándar ISO 9660, usado para CD de datos.



Pregunta de refuerzo.

Indica cuál de los siguientes elementos no es un dispositivo de entrada-salida:

- a) Disco duro externo
- b) Lápiz óptico
- c) Impresora 3D
- d) Memoria ROM

La respuesta es la d. La memoria ROM es uno de los componentes internos del ordenador, no puede considerarse un periférico de entrada-salida. El resto de elementos indicados en a, b y c sí son dispositivos de entrada-salida.

El sistema operativo

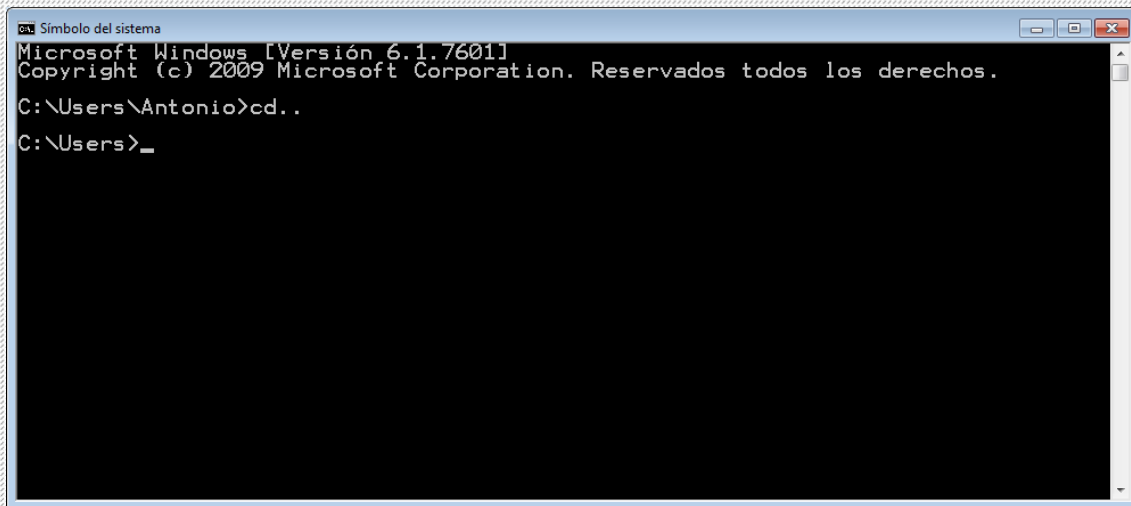
El sistema operativo es el componente software más importante del ordenador, pues se trata de un programa que hace que el propio ordenador funcione. Entre sus funciones principales están:

- Proporcionar una interfaz entre el resto de programas del ordenador, los dispositivos hardware y el usuario.
- Administrar los recursos de la máquina, como la memoria y la CPU
- Gestionar y organizar los archivos
- Controlar los dispositivos periféricos

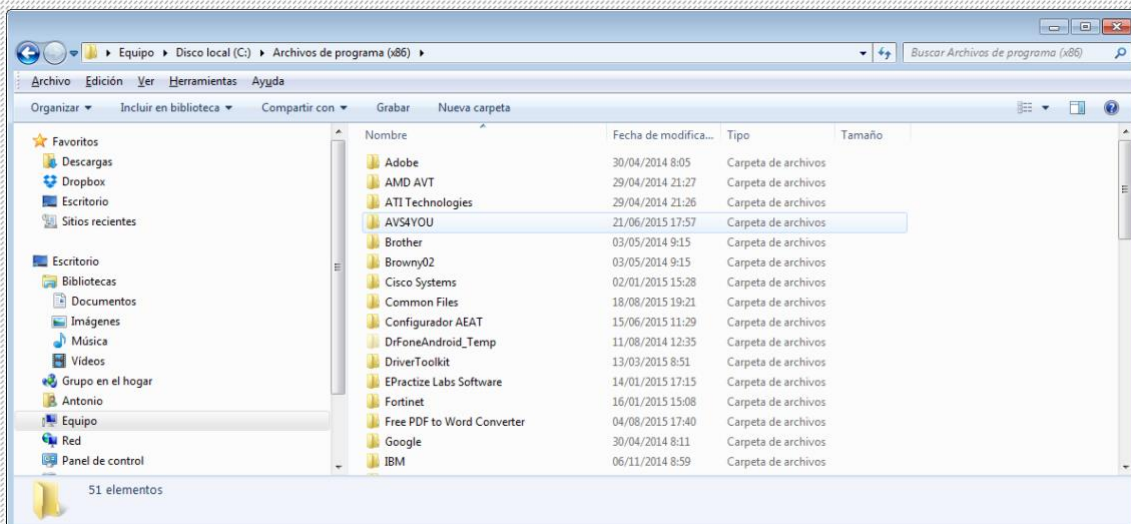
Como vemos, es el responsable de que todo funcione adecuadamente dentro del ordenador.

Existen varios tipos de sistemas operativos en función del número de usuarios simultáneos que lo pueden utilizar, del número de microprocesadores que gestionan, etc. Por ejemplo, un sistema operativo multiusuario permite que varios usuarios utilicen los programas al mismo tiempo, mientras que los multiprocesadores gestionan máquinas con varias CPU. También tenemos los sistemas operativos multitarea, que permiten ejecutar en el ordenador varios programas al mismo tiempo.

Los sistemas operativos deben proporcionar una interfaz al usuario a través de la cual éste pueda proporcionarle determinadas órdenes, como copiar o mover ficheros, eliminar ficheros, ejecutar y cancelar programas. Esta interfaz puede ser a través de un sistema de comandos



O bien a través de un entorno gráfico amigable con el que podemos interactuar a través del ratón.



Actualmente existen muchos Sistemas Operativos, entre los cuales podemos mencionar:

- DOS. Fue uno de los primeros sistemas operativos de ordenador, la interacción con el usuario es a través de comandos.
- Windows. Sistema operativo creado por Microsoft y, sin duda, el más popular. Se basa en la interacción con el usuario a través de interfaz gráfica. Desde que apareció a mediados de los 80, ha ido pasando por muchos cambios y versiones, desde el Windows 3.0 hasta el Windows 2010 actual.



- Linux. Se trata de un sistema operativo de libre distribución y es el que más ha crecido en los últimos años, hasta el punto de que actualmente es casi tan utilizado como Windows. Está basado en Unix y, aunque las primeras versiones utilizaban línea de comandos para interactuar con el usuario, las distribuciones existentes actualmente se basan en interfaz gráfica.

Linux no es el producto de una sola compañía, es el resultado de la contribución de un gran número de compañías y grupos de personas. De hecho, el *sistema GNU/Linux* es un componente central, el cual se transforma en muchos productos diferentes: las llamadas **distribuciones**. Cada distribución proporciona su propia apariencia y funcionamiento a Linux. Las hay desde grandes sistemas completos, hasta las más ligeras que entran en una memoria USB



- OS MAC. El sistema operativo OS fue creado por Apple para sus ordenadores Mac. Está diseñado sobre una sólida base UNIX para

aprovechar el hardware al máximo. Su variante IOS es el sistema operativo que llevan los teléfonos iPhone y las tabletas iPad.



- Android. Android es un sistema operativo creado por Google para ser utilizado en Smartphones y tabletas de diferentes fabricantes. La mayoría de los dispositivos de este tipo, salvo los Apple, incluyen este sistema operativo. Está basado en Linux, lo que lo hace bastante robusto y seguro. Actualmente, los programas que corren sobre este sistema operativo están escritos en lenguaje Java.



Pregunta de refuerzo.

Uno de los siguientes nombres no corresponde a un sistema operativo, indica cuál es:

- a) Eclipse
- b) Linux
- c) Windows
- d) Android

La respuesta correcta es la a. Eclipse es el nombre de un entorno de desarrollo, es decir, de un programa informático que se utiliza para crear programas. El resto de nombres indicados en b, c y d si corresponden a sistemas operativos.

Codificación de la información

A continuación, estudiaremos la manera en la que la información es codificada en el interior del ordenador para su tratamiento, así como las nomenclaturas utilizadas para referirnos a las cantidades de datos.

Sistema binario

Los datos, y toda la información en general, son codificados en el interior del ordenador utilizando un sistema de numeración conocido como binario. El sistema binario utiliza nada más que dos dígitos para representar la información, el 0 y el 1.

El sistema numérico binario fue el escogido por los ingenieros informáticos para el funcionamiento de los ordenadores, porque era más fácil para el sistema electrónico de la máquina distinguir y manejar solamente dos dígitos, en lugar de los diez dígitos (del 0 al 9), que constituyen el sistema numérico decimal. Además, la mayoría de los circuitos electrónicos que conforman un ordenador sólo puede detectar la presencia o ausencia de tensión en el circuito. A la presencia de tensión en un punto del circuito le asignamos el valor 1 y a la ausencia de la misma el valor 0.

Conversión binario a decimal

Mediante la combinación de unos y ceros podemos representar cualquier valor, por ejemplo, para representar el número dos utilizaremos la combinación 10, para el tres sería 11, mientras que para el cuatro será 100. Cada posición, representada por n , tiene un valor 2^n , siendo 0 la posición más a la derecha:

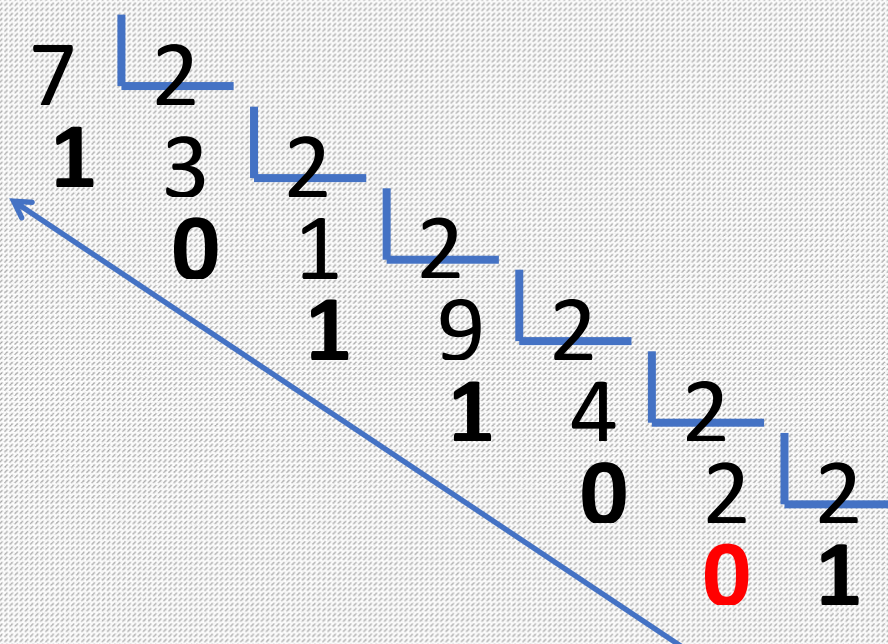
2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	0	1	1

Según esto, la combinación indicada en la imagen representaría al número:
 $1 \cdot 2^6 + 1 \cdot 2^4 + 1 \cdot 2^1 + 1 \cdot 2^0 = 64 + 16 + 2 + 1 = 83$

El sistema binario es un sistema de base dos, dos dígitos, de ahí que a los componentes que utilizan dicho sistema se les conozca también como **digitales**. A cada posición de un dígito binario se le conoce también como **bit**, así el número representado anteriormente estaría formado por 7 bits.

Conversión decimal a binario

Para obtener cual es la representación en binario de cualquier cantidad decimal, se toma este número y se divide entre 2, si el cociente obtenido es mayor que 2, se vuelve a dividir entre 2 y así sucesivamente. El resultado será el último cociente obtenido seguido de los restos de cada división en orden inverso a como se han obtenido. El siguiente ejemplo ilustra gráficamente lo explicado:



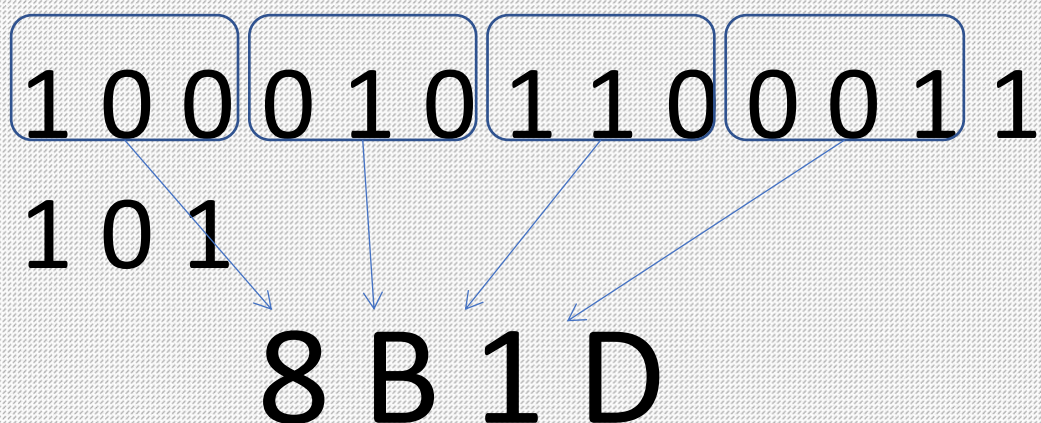
77 -> 1001101

Sistema hexadecimal

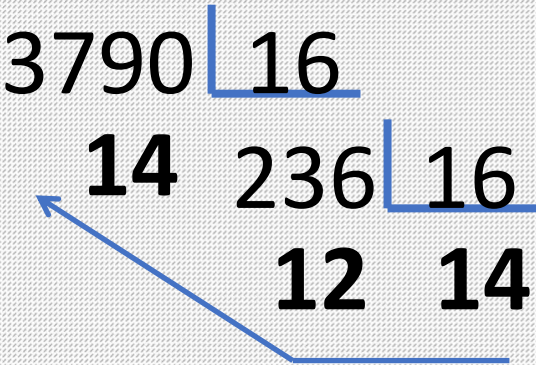
El sistema hexadecimal utiliza 16 símbolos para representar los datos. Además de los números del 0 al 9, se emplean las letras A, B, C, D, E y F, que representarían los números 10, 11, 12, 13, 14 y 15, respectivamente. Combinando estos símbolos podemos representar cualquier cantidad, teniendo en cuenta que el peso de cada posición tiene un valor de 16^n . Así pues, la combinación 5FA representaría al número:

$$5 \cdot 16^2 + 15 \cdot 16^1 + 10 \cdot 16^0 = 1280 + 240 + 10 = 1530$$

El sistema hexadecimal no es utilizado internamente por los ordenadores, que como hemos explicado, emplean el binario, sin embargo, resulta muy útil en informática para representar cantidades grandes de datos y el hecho de que 16 sea potencia de 2, simplifica la transformación de representación binaria a hexadecimal, pues bastará con transformar cada grupo de cuatro bits a su equivalente hexadecimal, comenzando por la derecha:



De la misma forma que si hizo para convertir a binario, si queremos obtener la representación hexadecimal de cualquier cantidad en decimal, debemos dividir esta por 16, así como los diversos cocientes obtenidos. El resultado en hexadecimal será el último cociente seguido de los restos en orden inverso:



Actividad práctica 1: Conversiones entre sistemas de numeración

Medidas de información

Como hemos explicado, el sistema binario es el que utilizan los computadores para representar la información. En este sistema, la unidad de medida básica es el bit, con el que podemos representar solamente dos posibles valores el 0 y el 1. Para poder referirnos a cantidades mayores, se introdujo el byte, que es la combinación de 8 bits. A partir de ahí, se utilizan múltiplos de esta unidad de medida, como el KiloByte, que son 1024 bytes, o el MegaByte que son 1.024.000 bytes. El siguiente cuadro nos muestra las diferentes unidades utilizadas:

Unidad	Abrev.	Se habla de	Representa
1 bit	bit	bits	unidad mínima
1 Byte	Byte	bytes	conjunto de 8 bits
1 kiloByte	KB	kas	1024 Bytes
1 MegaByte	MB	megas	1024 KB (1.048.576 bytes)
1 Gigabyte	GB	gigas	1024 MB (1.073.741.824 bytes)
1 Terabyte	TB	teras	1024 GB (un billón de bytes)

Dado que la información en el interior de una memoria o un disco duro se representa en binario, las unidades de mediada anteriores se utilizan también para expresar las capacidades de estos componentes. Así, cuando decimos que una memoria tiene una capacidad de 4 gigas, estamos diciendo que es capaz de almacenar más de cuatro millones de bytes de información, dicho de otra manera, podría contener hasta 4*8=32 millones de ceros y unos de información.

Codificación de la información

Los sistemas de codificación se utilizan para representar y almacenar la información en memoria.

En la década de 1960, se adoptó el código ASCII como estándar para representación de los caracteres del alfabeto latino tal como se usa en inglés moderno y en otras lenguas occidentales.

Código ASCII

En ASCII Cada carácter alfanumérico tiene asignado una combinación binaria de 8 bits (byte), con lo que utilizando este sistema de codificación podríamos representar hasta 256 símbolos. Datos y programas son codificados en este sistema dentro del ordenador.

Los símbolos que puede representar el código ASCII se pueden dividir en tres grupos:

- Caracteres de control. No representan caracteres con una representación visual, sino que, como su nombre indica, tienen funciones de control, como por ejemplo la tecla escape, el control de carro, la tabulación, etc. Este grupo de símbolos están representados con los códigos ASCII del 0 al 31 y también el 127.
- Caracteres alfanuméricos. Se trata de los números, las letras del alfabeto y otros símbolos utilizados en la escritura de texto. Están representados por los códigos que van del 32 al 126
- ASCII extendido. Se emplea para representar caracteres especiales, como letras acentuadas y con diéresis, la letra ñ, etc. Los códigos correspondientes a estos caracteres van del 128 a 255

Las siguientes tablas muestran los caracteres que forman cada uno de estos grupos con su código correspondiente:

Caracteres de control ASCII			
DEC	HEX	Símbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de línea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

Caracteres ASCII imprimibles								
DEC	HEX	Símbolo	DEC	HEX	Símbolo	DEC	HEX	Símbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	\$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(72	48h	H	104	68h	h
41	29h)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	
61	3Dh	=	93	5Dh]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	-			

ASCII extendido											
DEC	HEX	Símbolo	DEC	HEX	Símbolo	DEC	HEX	Símbolo	DEC	HEX	Símbolo
128	80h	Ç	160	A0h	á	192	C0h	Ł	224	E0h	Ó
129	81h	ù	161	A1h	í	193	C1h	ł	225	E1h	ô
130	82h	é	162	A2h	ó	194	C2h	Ł	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	ł	227	E3h	Ö
132	84h	ä	164	A4h	ñ	196	C4h	Ł	228	E4h	ö
133	85h	à	165	A5h	Ñ	197	C5h	ł	229	E5h	Õ
134	86h	á	166	A6h	ª	198	C6h	Ł	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ł	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	Ł	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	Ł	233	E9h	Û
138	8Ah	è	170	AAh	¬	202	CAh	Ł	234	EAh	Ü
139	8Bh	ï	171	ABh	½	203	CBh	Ł	235	EBh	Ù
140	8Ch	î	172	ACH	¼	204	CCh	Ł	236	ECh	Ý
141	8Dh	ì	173	ADh	¡	205	CDh	Ł	237	EDh	Ý
142	8Eh	Ā	174	AEh	«	206	CEh	Ł	238	EEh	ˆ
143	8Fh	Ā	175	AFh	»	207	CFh	Ł	239	EFh	˙
144	90h	É	176	B0h	⋮	208	D0h	Ł	240	F0h	˚
145	91h	æ	177	B1h	⋮	209	D1h	Ł	241	F1h	±
146	92h	Æ	178	B2h	⋮	210	D2h	Ł	242	F2h	¾
147	93h	ô	179	B3h	⋮	211	D3h	Ł	243	F3h	¼
148	94h	ò	180	B4h	⋮	212	D4h	Ł	244	F4h	¶
149	95h	ò	181	B5h	⋮	213	D5h	Ł	245	F5h	§
150	96h	ù	182	B6h	⋮	214	D6h	Ł	246	F6h	÷
151	97h	ù	183	B7h	⋮	215	D7h	Ł	247	F7h	¿
152	98h	ÿ	184	B8h	⋮	216	D8h	Ł	248	F8h	ˆ
153	99h	Û	185	B9h	⋮	217	D9h	Ł	249	F9h	˙
154	9Ah	Ü	186	BAh	⋮	218	DAh	Ł	250	FAh	˙
155	9Bh	ø	187	BBh	⋮	219	DBh	Ł	251	FBh	˙
156	9Ch	£	188	BCh	⋮	220	DCh	Ł	252	FCh	˙
157	9Dh	Ø	189	BDh	⋮	221	DDh	Ł	253	FDh	˙
158	9Eh	×	190	BEh	⋮	222	DEh	Ł	254	FEh	˙
159	9Fh	f	191	BFh	⋮	223	DFh	Ł	255	FFh	˙

Código Unicode

Se trata de una extensión del código ASCII, en la que se utilizan 2 bytes para representar los caracteres, por lo que se podrían representar hipotéticamente algo más de 64000 símbolos.

Además de los caracteres ya representados por el código ASCII, Unicode incluye la representación de símbolos de multitud de lenguas, como el chino, árabe o ruso. Es el sistema de codificación utilizado por las computadoras actuales.

Pregunta de refuerzo

El símbolo & se representa internamente en el ordenador:

- a) Mediante la combinación 00100110
- b) Directamente como &, pues es un símbolo ASCII
- c) Este símbolo no tiene representación
- d) Mediante la combinación 0&h

Solución: La respuesta correcta es la a, pues corresponde a la representación en binario del número 38, que es el código ASCII del símbolo &. La b no puede ser porque toda la información se almacena en binario, no tal cual. La respuesta c no tiene sentido y la d no es correcta por el mismo motivo que la b.

Introducción a la programación

Un ordenador es un sistema ciertamente complejo que cuenta con unos elementos hardware ciertamente sofisticados y potentes, pero por muy complejo que sea no es capaz de realizar ninguna tarea por sí mismo. Para que pueda llevar a cabo las numerosas funciones que estamos acostumbrados a ver habitualmente en los ordenadores modernos, como realizar complicados cálculos, procesar imágenes, datos y textos, comunicarse con otros equipos, etc., es necesario la existencia de un **programa** que transmita una serie de órdenes a bajo nivel al ordenador de modo que, al ser ejecutadas de manera conjunta y ordenada, consigan completar una determinada tarea útil para el usuario.

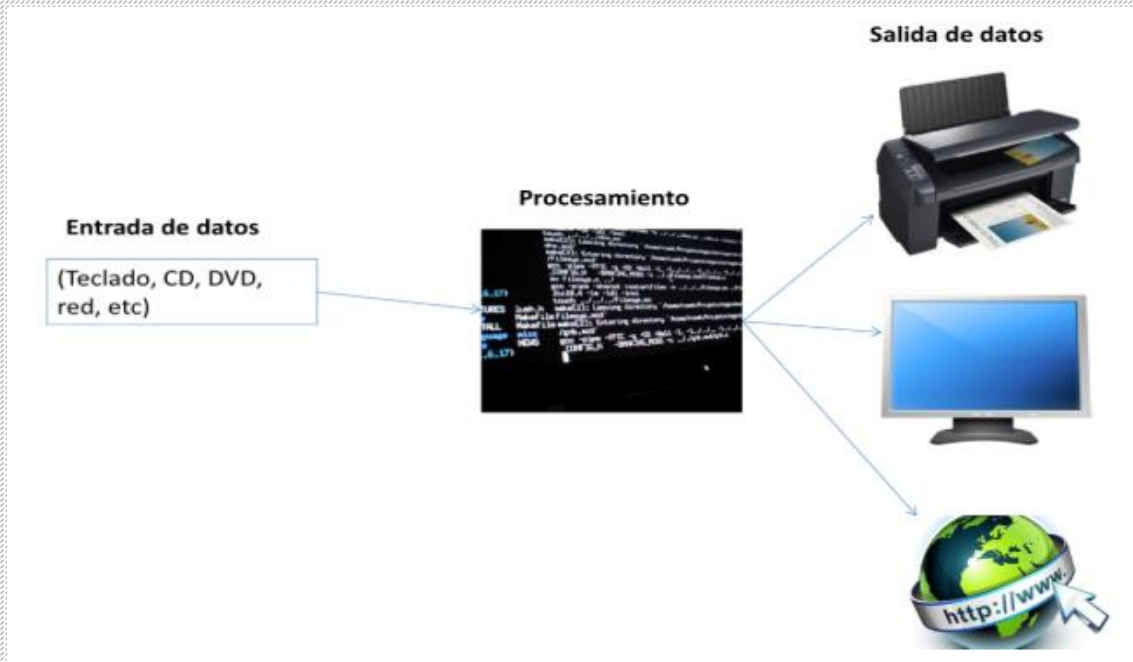
Un ordenador, es capaz de realizar únicamente tres tipos de operaciones:

- Operaciones aritméticas básicas
- Operaciones de tipo lógico (comparar dos valores)
- Almacenamiento y recuperación información.

Estas tres operaciones convenientemente ligadas entre sí **forman lo que llamamos un programa.**

Programas y algoritmos

Un programa en definitiva es un conjunto de órdenes que ejecuta el ordenador para conseguir un objetivo. Las órdenes se proporcionan a través de un **lenguaje de programación** (códigos). A estas órdenes escritas en un determinado lenguaje de programación se les llama también instrucciones. De forma general este conjunto de instrucciones toma unos datos de entrada y devuelve unos datos de salida, o resultados.



El ordenador siempre funciona bajo control de un programa, incluso las operaciones más básicas que hace el ordenador, como comunicarse con los dispositivos de entrada/salida, interactuar con el usuario, gestionar los propios recursos del ordenador, etc., son realizados por un programa llamado sistema operativo, que es el programa más importante que ejecuta un ordenador.

Los programas que se ejecutan en un ordenador se encuentran en unidades de almacenamiento permanente, como el disco duro, un disco óptico, o un pendrive, incluso la memoria ROM, donde se almacena parte del núcleo del sistema operativo. Cuando el ordenador recibe la orden de ejecutar un programa, éste, o parte de él, es cargado en la memoria RAM del ordenador para su ejecución.

Para realizar un programa, los programadores definen un **algoritmo**. Un algoritmo es la descripción exacta y sin ambigüedades de la secuencia de pasos elementales a aplicar a un proceso para que, a partir de unos datos iniciales, se obtenga la solución buscada a un problema determinado. Un programa es la expresión de un algoritmo en un lenguaje de programación entendible por el ordenador.

Lenguajes de programación

Los lenguajes de programación proporcionan la notación utilizada para la escritura de los programas. Para la escritura de los programas o aplicaciones informáticas actuales, el programador utiliza un lenguaje de programación

denominado "de alto nivel", que le permite escribir las instrucciones siguiendo una notación "entendible" para el programador, no así para el ordenador. Para que el ordenador pueda entender las órdenes contenidas en un programa cualquiera escrito por el programador en lenguaje de "alto nivel", es necesario traducir estas instrucciones a otras "de bajo nivel" que puedan ser entendidas por el ordenador. Este código de bajo nivel, conocido como "código máquina", está compuesto solamente de unos y ceros, es el único que entiende el ordenador y es el que le permite interpretar las órdenes contenidas de los programas para que las pueda ejecutar.

Tipos de lenguajes de programación

Según el nivel, es decir, la cercanía de las instrucciones del lenguaje de programación con el lenguaje humano, estos se pueden agrupar en tres tipos:

- Lenguajes de bajo nivel
- Lenguajes de nivel intermedio
- Lenguajes de alto nivel

Lenguajes de bajo nivel

Se trata de lenguajes cuyo juego de instrucciones son entendibles directamente por el hardware. El lenguaje de bajo nivel que utilizan los ordenadores es el conocido como "código máquina", y está formado por unos y ceros, es decir, código binario, lo que entiende directamente el microprocesador.



Además de la complejidad que supone escribir programas de esta manera, cada tipo de microprocesador dispone de su propio juego de instrucciones o combinaciones de ceros y unos con las que se puede indicar a este las tareas a realizar, por lo que un programa en código máquina solo puede utilizarse en la máquina para la que se programó.

Lenguajes de nivel intermedio

Se les conoce con ese nombre porque están a medio camino entre el código máquina y los lenguajes de alto nivel. El lenguaje de ensamblador fue el primer lenguaje de nivel intermedio en desarrollarse, con el objetivo de sustituir el lenguaje máquina por otro más similar a los utilizados por las personas. Cada instrucción en ensamblador equivale a una instrucción en lenguaje máquina, utilizando para su escritura palabras nemotécnicas en lugar de cadenas de bits.

El juego de instrucciones del lenguaje ensamblador está formado por palabras abreviadas procedentes del inglés (Ejemplo: MOV A, B). La programación en lenguaje ensamblador precisa de un amplio conocimiento sobre la constitución, estructura y funcionamiento interno de un ordenador, ya que maneja directamente las posiciones de memoria, registros del procesador y demás elementos físicos. El siguiente listado muestra un ejemplo de programa escrito en lenguaje ensamblador:

```
INICIO: ADD B, 1  
      MOV A, E  
      CMP A, B  
      JE FIN  
      JMP INICIO  
FIN : END
```

Al igual que en el caso del código máquina, los programas escritos en ensamblador son dependientes del procesador para el que se han creado. Aunque fue el primer lenguaje de programación que se empezó a utilizar para ordenadores, actualmente no se utiliza en la creación de programas para usuarios de ordenador, su uso está limitado a la programación de microcontroladores y dispositivos electrónicos.

Lenguajes de alto nivel

Se les llama lenguajes de alto nivel porque el conjunto de órdenes que utilizan son fáciles de entender y aprender. Además, no hay incompatibilidades entre un microprocesador y otro, por lo que un programa escrito para un ordenador puede ser utilizado en otro.

Como inconveniente destacable, está la necesidad de traducir los programas escritos en un lenguaje de alto nivel a un lenguaje máquina o ensamblador para que pueda ser ejecutado por la unidad central de proceso, lo que significa disponer necesariamente de un software traductor (ensamblador, compilador o intérprete) para cada tipo de ordenador utilizado. Más adelante, hablaremos de estos paquetes software.

El siguiente listado corresponde a un programa escrito en un lenguaje de programación de alto nivel:

```
int c=20;
int sum;
for(int i=1;i<=c;i++){
    sum=sum+i;
}
System.out.println(sum);
```

El abanico de lenguajes de programación de alto nivel existentes hoy en día es enorme y no para de crecer. Entre más utilizados en la actualidad tenemos:

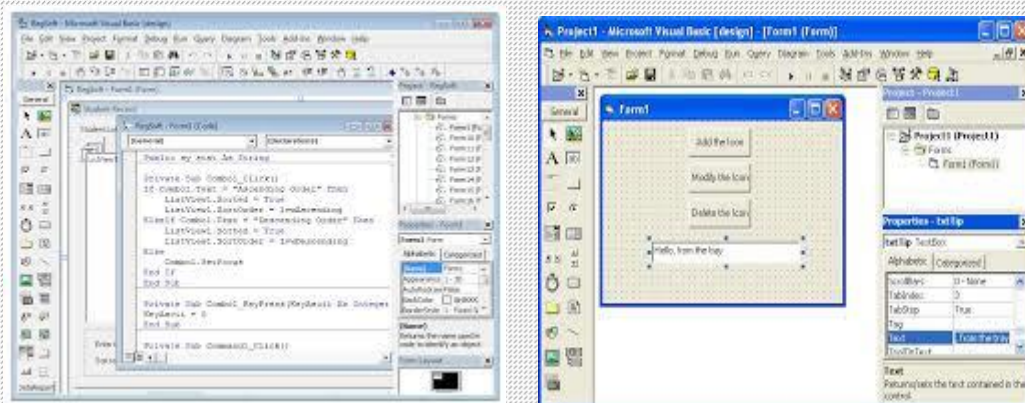
Java. Se trata de uno de los lenguajes de programación más utilizados actualmente, apareció a principios de los años 90 y desde entonces su uso no ha hecho más que extenderse. Puede ser utilizado para crear programas para muy diferentes fines, como aplicaciones de escritorio, aplicaciones para Web, incluso programas para dispositivos electrónicos como tabletas o smartphones.

Una de las características más interesantes de este lenguaje es que es multiplataforma, lo que significa que un programa compilado en Java puede ser ejecutado en diferentes sistemas operativos. Esto disparó la popularidad y uso de este lenguaje de programación y muchos fabricantes software, como IBM y Oracle se lanzaron a crear productos basados en Java.



- JavaScript. Aunque de nombre similar a Java, solo se parece a éste en los fundamentos sintácticos. JavaScript es un lenguaje interpretado, utilizado en la creación de scripts en páginas Web, es decir, bloques de código integrados dentro de una página y que son interpretados y ejecutados por el navegador Web al procesar dicha página.

- C. Es un lenguaje de programación muy popular. Se desarrollo a principios de los años 70 y se caracteriza porque permite crear código muy eficiente que optimiza los recursos del ordenador. Este lenguaje es el que se utiliza en la creación de muchos sistemas operativos, como Unix o Windows. Aunque dispone de estructuras sintácticas propias de un lenguaje de alto nivel, también incorpora instrucciones de bajo nivel que permiten un control de los recursos hardware, lo que le hace muy apropiado en entornos donde estos recursos puedan ser limitados.
- Visual Basic. Muy popular en los años 80 por su simplicidad y potencia, con pocas líneas de código se pueden realizar muchas tareas. Es el lenguaje de programación con el que se crearon las primeras aplicaciones para Windows.



- PHP. Dispone de un amplio juego de instrucciones especialmente diseñadas para la creación de programas en entorno Web. Su código es interpretado por un servidor Web que genera dinámicamente páginas en HTML, lo que le hace adecuado para este tipo de entornos. También dispone de instrucciones para acceder a bases de datos de tipo MySQL
- C#. Se trata de una versión actualizada del lenguaje C que Microsoft creo a finales de los 90 para incorporarlo a la plataforma .NET. Con el se pueden desarrollar aplicaciones Windows, Web y móviles de forma sencilla, al estilo de Visual Basic, pero con la elegancia y precisión de C
- Ruby. Es un lenguaje de programación cuya aparición se remonta a mediados de los 90, pero que es últimamente cuando está adquiriendo mayor popularidad. Se trata de un lenguaje dinámico y de código abierto y está enfocado a la productividad. Se aplica especialmente sobre la plataforma on-rails para el desarrollo de aplicaciones para Web

Pregunta de refuerzo

El lenguaje que manipula registros del procesador es:

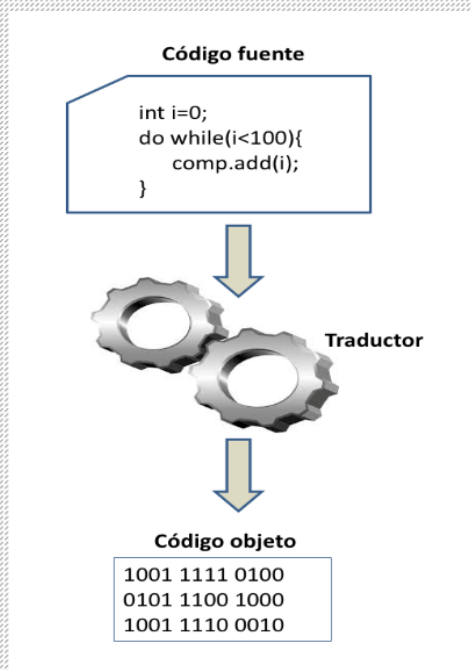
- a) Lenguaje máquina
- b) Lenguaje C
- c) Cualquier lenguaje de alto nivel
- d) Lenguaje ensamblador

Solución: La respuesta correcta es la d. El lenguaje ensamblador, que es un lenguaje de nivel medio, trata directamente con registros del microprocesador y posiciones de memoria. La a es falsa porque el lenguaje máquina es binario puro, no trata con elementos hardware, la b es falsa porque C es un lenguaje de alto nivel que se abstrae de los detalles del hardware, mientras que la c es falsa también por el mismo motivo.

Ensambladores, compiladores e intérpretes

Cuando utilizamos un lenguaje distinto al lenguaje máquina, los programas elaborados en dicho lenguaje deben ser traducidos a código binario a fin de que las instrucciones en ellos especificadas puedan ser entendidas y ejecutadas por el ordenador.

Esta tarea de traducción del programa, denominado **código fuente**, a código entendible por el microprocesador, denominado **código objeto**, es realizada por un software traductor que, dependiendo de la naturaleza del código fuente y la manera en la que realice la traducción, se denomina ensamblador, compilador o intérprete.



Ensambladores

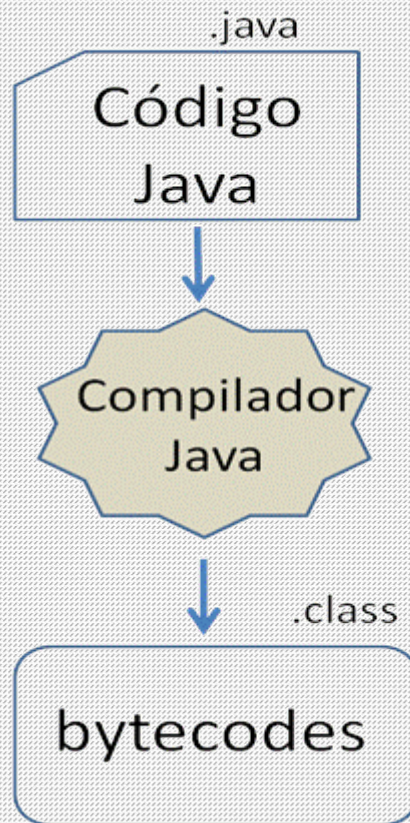
Un ensamblador es un programa que traduce el código de un programa escrito en ensamblador a código máquina entendible por el ordenador. Estos programas suelen venir ya incorporados en el propio ordenador, ya que cada tipo de microprocesador dispone de su propio juego de instrucciones en ensamblador.

Compilador

Un compilador es un software que traduce el código fuente escrito en un lenguaje de programación de alto nivel, a código ejecutable por el ordenador, normalmente código máquina, aunque en algunos casos se traduce a un código intermedio que posteriormente es interpretado durante la ejecución del programa. Cada lenguaje de programación de alto nivel requiere de su propio software compilador.

Durante la fase de compilación, el código fuente se suministra a través de un archivo de texto que contiene el conjunto de instrucciones que forma el programa. Como resultado de la compilación, se genera un nuevo archivo en código binario que, dependiendo del caso, puede ser directamente ejecutado o requerir un nuevo proceso de traducción posterior.

La siguiente imagen nos muestra un ejemplo de compilación de un programa escrito en Java. Como vemos, el código fuente se suministra en archivos de texto .java y como resultado de la generación se genera uno o varios archivos de código objeto con extensión **.class**, que deberán ser traducidos a código máquina en el momento de ejecutar el programa:



Durante la compilación de un programa tienen lugar las siguientes acciones:

- **Análisis léxico.** El programa se divide en tokens o secuencias de caracteres que tienen un significado. En esta fase, se detectan las palabras reservadas, signos de puntuación, variables etc.
- **Análisis sintáctico.** Los tokens se agrupan jerárquicamente en frases gramaticales que el compilador utiliza para sintetizar la salida. Se comprueba si lo obtenido de la fase anterior es sintácticamente correcto.
- **Análisis semántico.** Durante esta fase se revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código.
- **Síntesis.** Se genera el código objeto.
- **Optimización.** Se optimiza el código generado en la fase anterior de modo que resulte más rápido de ejecutar.

Intérpretes

Un intérprete realiza la traducción del programa fuente y lo ejecuta directamente, sin generar ningún código objeto. La traducción o interpretación y la ejecución no se realizan como procesos independientes, sino en una misma

operación e instrucción por instrucción, respetando rigurosamente el orden establecido en ellas.

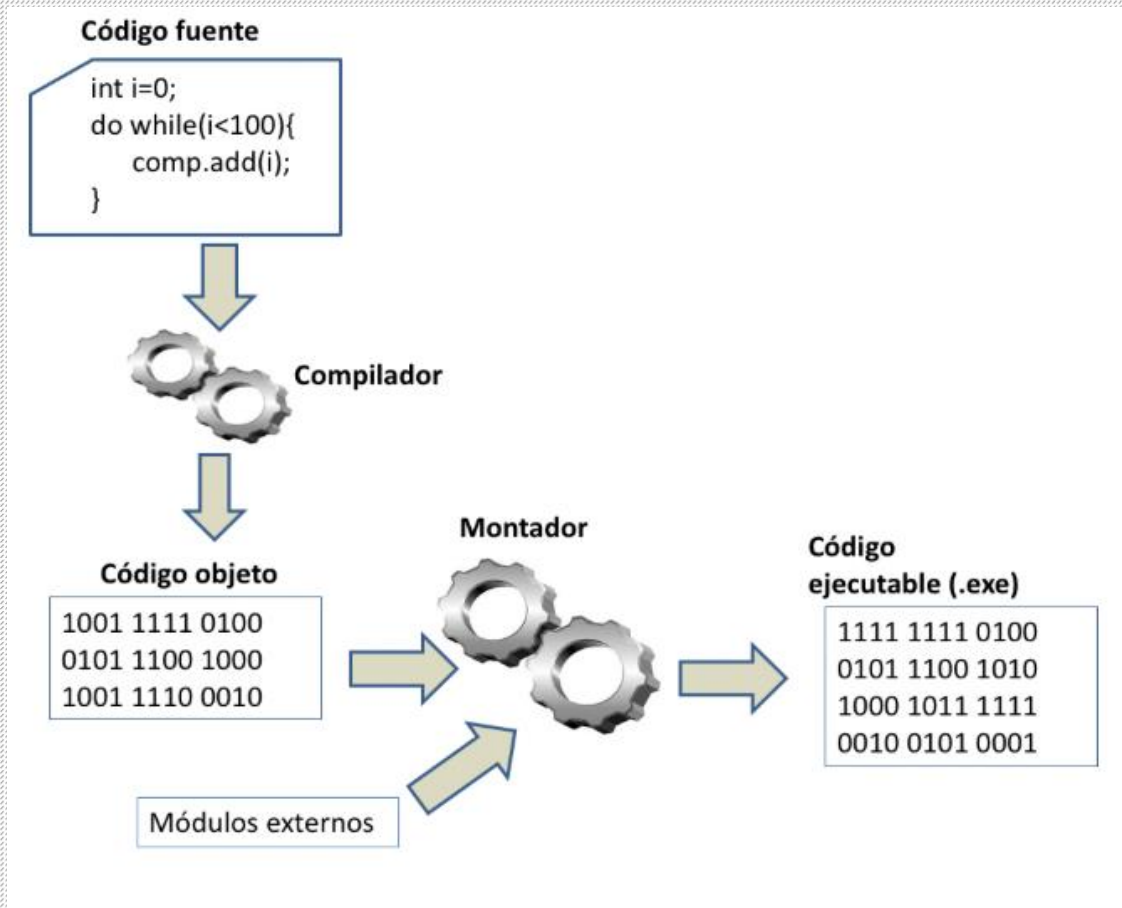
Un ejemplo de lenguaje interpretado es JavaScript. El código fuente de un programa escrito en JavaScript se encuentra embebido dentro de una página Web. Cuando la página es procesada por el navegador, éste realiza la interpretación del script, es decir, traduce y ejecuta las instrucciones secuencialmente.

También el código objeto o bytecodes, generado durante la compilación de un programa Java, sufre un proceso de interpretación durante la fase de ejecución de dicho programa. En ese momento, un software conocido como Máquina Virtual Java traduce y ejecuta cada línea de bytecodes.

Linkado

En algunos lenguajes de programación, un programa fuente que ha sido compilado y por tanto traducido a lenguaje máquina, es decir, convertido en programa objeto, es un programa que todavía no es ejecutable. Esto es así porque el código objeto generado necesita de otras librerías de código externas para poderse ejecutar.

Un programa **linkador** o montador tiene como misión resolver los direccionamientos del código objeto a las librerías de código externas y combinar el conjunto de ambos en lo que se conoce como un ejecutable, que es un archivo de código máquina o binario con extensión .exe que puedes ser ejecutado directamente en el ordenador. Normalmente, un .exe es dependiente del sistema operativo para el que se ha generado, por lo que no podría considerarse como un archivo multiplataforma, es decir, que pueda ser ejecutado en cualquier ordenador.

**Pregunta de refuerzo**

Indica cual de las siguientes es una diferencia entre un compilador y un intérprete:

- a) Un compilador genera código ejecutable puro, mientras que el intérprete genera código intermedio
- b) Un compilador genera un archivo de código objeto, mientras que el intérprete no genera ningún archivo intermedio
- c) Un compilador genera código binario, mientras que el intérprete genera código en ensamblador
- d) Un compilador solo es aplicable a lenguajes de alto nivel, mientras que el intérprete se emplea con lenguajes de código intermedio

Solución: La respuesta correcta es la *b*. Los compiladores generan archivos resultantes del proceso de compilación, mientras que el intérprete traduce y ejecuta en la misma operación, sin generar ningún código intermedio. La respuesta *a* no es correcta porque los intérpretes no generan código en la traducción, además, no en todos los casos de compilación se genera ejecutable puro. La *c* es falsa porque el intérprete, como hemos indicado, no genera

código intermedio y la *d* es falsa porque el uso de compiladores o intérpretes no tiene relación con el nivel del lenguaje.

Entornos de desarrollo integrados

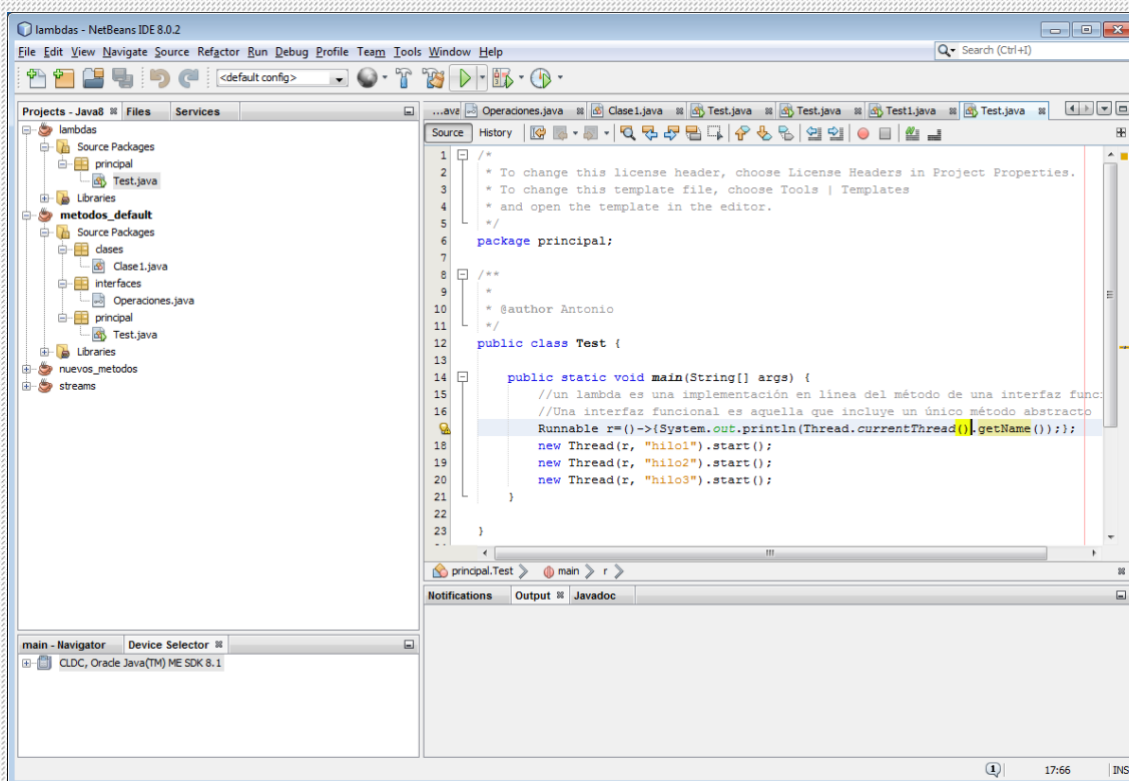
Los entornos de desarrollo integrados, conocidos como IDE, son programas informáticos creados con el objetivo de facilitar la creación, depuración y testeo de programas escritos con un determinado lenguaje de programación.

Los entornos de desarrollo ofrecen al programador todo lo necesario para que pueda realizar los programas, como un editor de texto para poder escribir el código con ayudas en línea que le informan de posibles errores de sintaxis, un compilador para poder generar el código objeto, un linkador, opciones para depurar los programas y ejecutarlos paso a paso en busca de errores, etc.

Durante el estudio de los lenguajes de programación dentro del itinerario abordaremos con detalle el manejo de un IDE en concreto, sin embargo, ahora te vamos a presentar algunos de los más utilizados en la actualidad.

Netbeans

El entorno de desarrollo Netbeans es uno de los más utilizados por la comunidad de desarrolladores. Se trata de un programa de libre distribución que puede descargarse desde la dirección netbeans.org, con el que puedes crear programas en lenguaje Java, PHP y C.

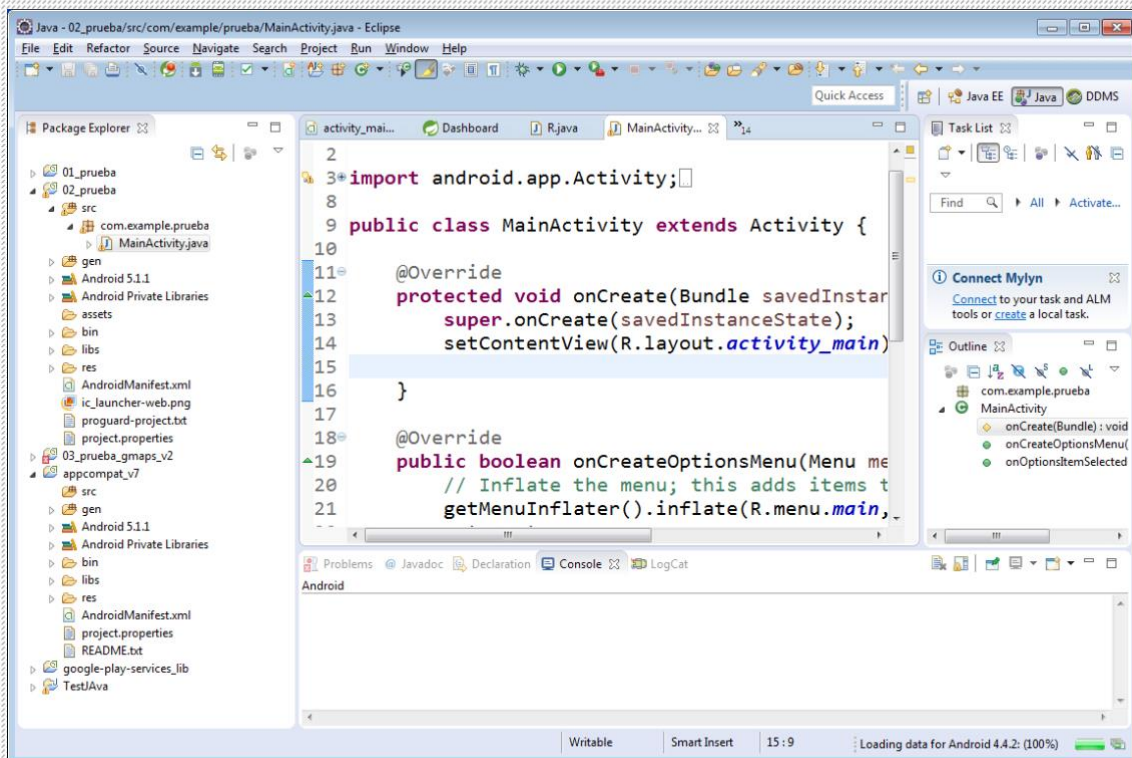


Es muy sencillo de manejar y ofrece muchas facilidades para la escritura de los programas. Permite realizar desde programas sencillos basados en la

interacción con el usuario a través de la línea de comandos, hasta complejas aplicaciones para entorno Web.

Eclipse

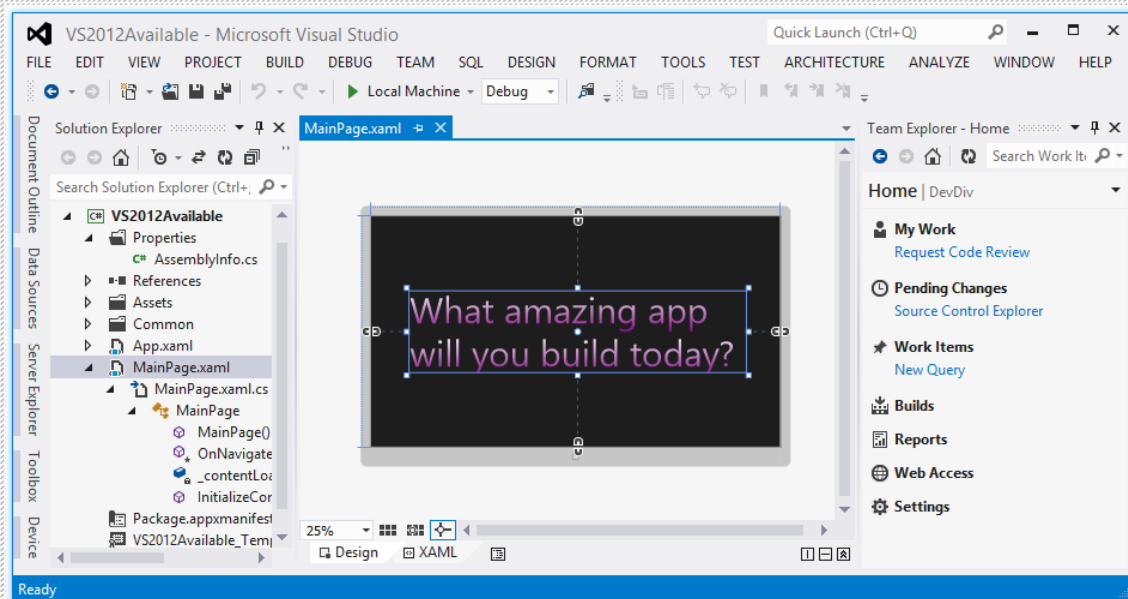
Es el entorno de desarrollo más extendido entre la comunidad de desarrolladores Java. Al igual que netbeans, es de libre distribución y podemos descargarlo desde la dirección <https://eclipse.org/downloads/>



Se trata de un IDE para creación de programas Java en todos los entornos: aplicaciones de escritorio, Web, etc. Incluso es el entorno de desarrollo más utilizado en la creación de programas para dispositivos Android.

Visual Studio.NET

Es el entorno de desarrollo de Microsoft para la creación programas para Windows y Web sobre sistema operativo Windows.

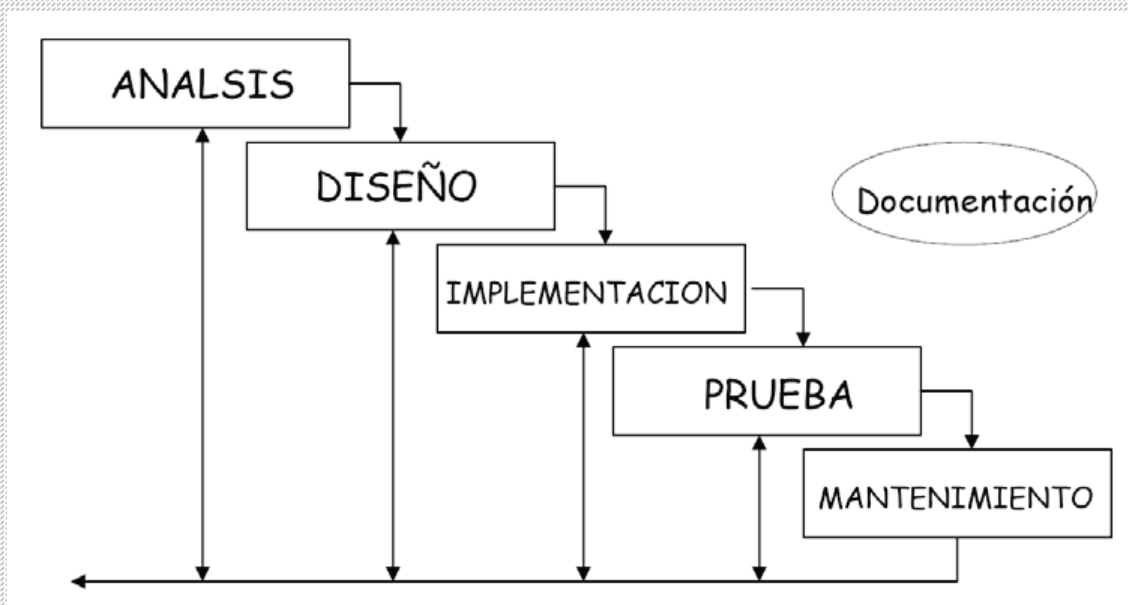


Puede utilizarse con cualquiera de los lenguajes que forman la plataforma .NET, como Visual Basic y C#.

Fases en el desarrollo de un programa

El desarrollo de cualquier programa informático sigue una serie de fases que no siempre son llevadas a cabo por la misma persona, pero que es necesario que sean realizadas todas y de forma coordinada a fin de conseguir un producto de calidad y sin errores.

El siguiente diagrama nos muestra cuales son estas fases y el orden en el que deben ser realizadas:



Fase de análisis

Su principal objetivo es establecer **qué** debe hacer el software a desarrollar y **no cómo** debe hacerlo. Suele ser misión del analista funcional.

Durante esta fase, el analista recogerá las ideas expuestas por el cliente y deberá transformarlas en especificaciones válidas para el diseñador en lo que se conoce como el **documento de análisis funcional**. Así mismo, dicho documento se deberá validar con el cliente y comprobar que las especificaciones definidas en el mismo, se ajustan a los deseos que cliente le transmitió.

Fase de diseño

En esta fase, partiendo de la información recogida en el análisis el diseñador establece **cómo** se llevan a cabo los objetivos presentados en dicho análisis. Suele ser ésta, la misión del analista orgánico.

Durante esta fase se deberá alcanzar una solución óptima, detallada y con la mayor precisión posible para el desarrollo de la aplicación. Se trata de diseñar el conjunto de algoritmos de lo que el programa debe hacer y para expresar dichos algoritmos, el analista orgánico cuenta con los siguientes elementos:

- **Diagramas de flujo.** El diagrama de flujo u ordinograma es una técnica que se basa en expresar de forma gráfica la secuencia lógica y detallada de las operaciones que necesitamos para la realización de un programa. Los ordinogramas se emplearon mucho hasta finales de los 80, y debido a la complejidad de los programas actuales, hoy en día no es la técnica más adecuada para expresar un algoritmo de programación.
- **Pseudocódigo.** El pseudocódigo consiste en expresar el algoritmo de un programa, de la forma más detallada posible, utilizando frases y palabras del lenguaje común. Las palabras y expresiones que utilizamos en pseudocódigo se asemejan a las que emplean los lenguajes de programación, de ahí que se llame pseudocódigo, pero de una manera más informal. Esto permite que la traducción del algoritmo al código final resulte una tarea relativamente sencilla.

Fase de implementación

Durante esta fase el programador debe convertir el algoritmo diseñado en la fase anterior a código escrito en un lenguaje de programación de alto nivel. El éxito de esta tarea depende en gran medida del detalle y precisión con el que se ha creado el algoritmo, por lo que la fase de diseño de un programa resulta de gran importancia en el desarrollo de una aplicación.

Fase de prueba

Una vez completado el programa, se somete a una serie de pruebas a fin de detectar fallos en la implementación. Dichos fallos nos pueden llevar de nuevo a realizar cambios y ajustes en las fases anteriores. Existen técnicas y conjuntos de utilidades especialmente creados para ayudar a los programadores a realizar un completo plan de pruebas que permita detectar cualquier fallo de diseño e implementación.

Fase de mantenimiento

En esta fase tiene lugar la corrección de errores descubiertos por los usuarios y que no fueron detectados durante la fase de pruebas. Así mismo, durante esta fase se mejora el software con la inclusión de nuevas características y modificación de otras ya existentes, con el fin de adaptar mejor el producto a las necesidades de los usuarios finales.

Fase de documentación

La fase de documentación no es una fase independiente como el resto de las presentadas, sino que se integra con todas las anteriores. Esto significa que se deben documentar todos los pasos y acciones realizadas en cada una de las fases del desarrollo. La documentación es algo imprescindible para el mantenimiento y mejora del software.

Pregunta de refuerzo

Los errores de compilación de un programa se corrigen en la fase de:

- a) Diseño
- b) Implementación
- c) Pruebas
- d) Mantenimiento

Solución. La respuesta correcta es la *b*. Los errores de compilación se producen al compilar el código del programa escrito en el lenguaje de alto nivel utilizado, por lo que se detectarán durante la fase de implementación del mismo. Estos se producen por errores de sintaxis cometidos durante la escritura del código.

Diseño de algoritmos

La fase de diseño de un programa resulta es sin duda alguna una de las más importante de todo el proceso de creación de un programa. En ella se diseña el algoritmo que describe lo que tiene que hacer el programa. Esta descripción se realiza siguiendo la **lógica de programación**.

Por lógica de programación entendemos la capacidad para expresar la secuencia de operaciones o algoritmo que debe seguir un programa informático para resolver un problema, utilizando para ello una serie de estructuras o expresiones estandarizadas.

La lógica de programación no es algo con lo que se nazca, sino que se adquiere con el tiempo y con la práctica. Por ello, el objetivo de este capítulo es presentarte todos los elementos necesarios para que, dadas unas especificaciones, seas capaz de expresar de forma adecuada la lógica de operaciones que un programa debería seguir para conseguir su objetivo. Según hemos indicado en el capítulo anterior, dos son las técnicas que se utilizan para diseñar el algoritmo de un programa:

- Diagramas de flujo
- Pseudocódigo

A continuación, analizaremos dichas técnicas con detalle y veremos numerosos ejemplos a fin de que adquieras la soltura necesaria para diseñar tus propios programas.

Pero antes de entrar en las técnicas de diseño de algoritmos, vamos a analizar uno de los elementos claves de la programación y, por tanto, del diseño de algoritmos: los datos.

Datos y variables

Durante el desarrollo de un programa informático, independientemente del lenguaje de programación utilizado, se utilizan datos y variables.

Los datos representan la información manejada por el programa, y pueden ser números, fechas, textos etc. Todo programa recibe unos datos de entrada que son manipulados por el mismo y, como resultado, generará unos datos de salida.

Durante la manipulación de los datos por parte del programa, estos son almacenados en **variables**. Una variable es una zona de memoria, a la que se le asigna un nombre o identificador, en la que se guarda un dato de un determinado tipo. Después, la variable puede utilizarse dentro del programa para realizar diferentes operaciones con el dato que contiene.

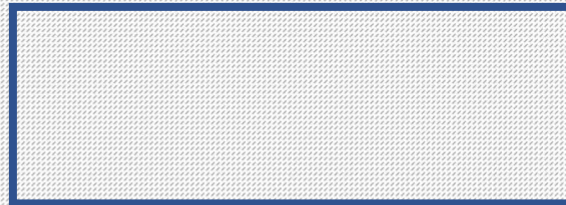
Normalmente, los lenguajes de programación obligan a declarar las variables antes de ser utilizadas. Declarar una variable es la manera de indicarle al compilador que queremos utilizar una variable, a fin de que éste pueda reservar el espacio en memoria para el almacenamiento de la misma.

La manera en la que un lenguaje declara una variable depende de la sintaxis del mismo. El siguiente ejemplo corresponde a una declaración de una variable de tipo entero en Java:

```
int num;
```

En este caso, se declara una variable de tipo entero, es decir, que solo puede almacenar datos numéricos de tipo entero, y se le asigna el identificador *num*. Gráficamente, la situación en memoria podría representarse así:

num



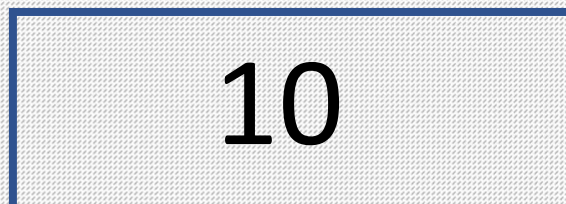
La caja sería como la zona de memoria asignada a la variable y el nombre que aparece a su izquierda sería el identificador asignado a la misma.

Una vez declarada la variable, el programa puede hacer uso de ella para asignarle datos y después realizar operaciones con los mismos. En el ejemplo anterior, la asignación de un dato a la variable se realizaría a través del signo =, tal y como se muestra en el ejemplo:

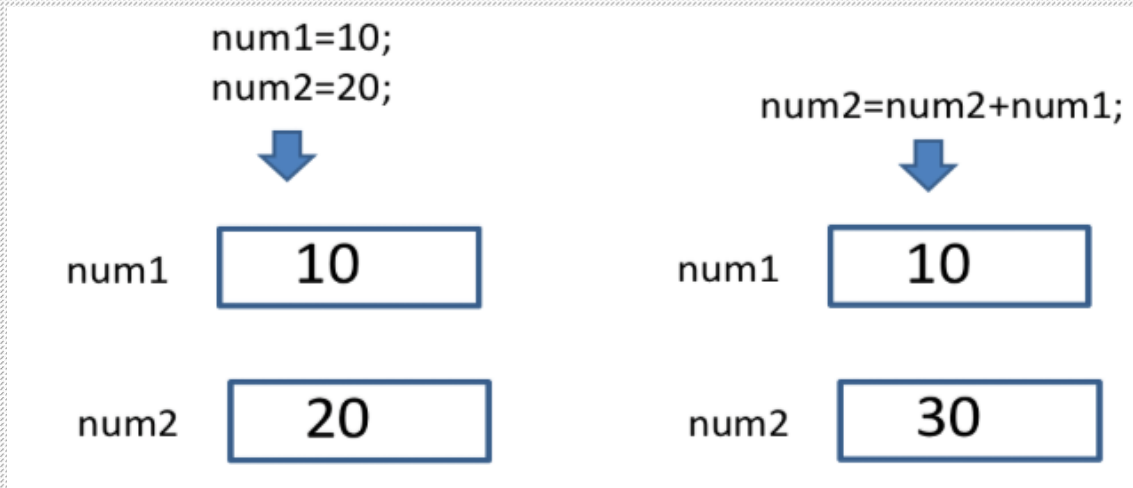
```
num=10;
```



num



Como decimos, a partir del momento en que la variable ya tiene el dato, pueden realizarse operaciones con el mismo, incluido la modificación del valor de la propia variable. El siguiente ejemplo, sumaría el contenido de dos variables y lo depositaría en una de ellas:



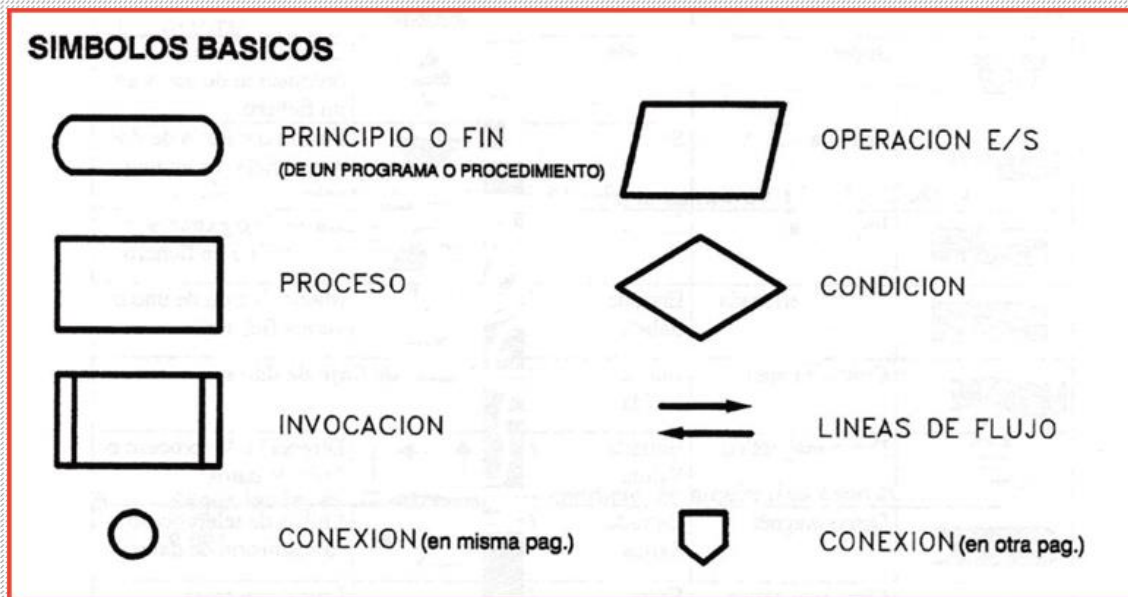
Independientemente del lenguaje de programación utilizado, los datos y las variables representan elementos importantes en la lógica de programación, por lo que serán utilizados también en la creación de algoritmos durante la fase de diseño de los programas.

Diagramas de flujo

Conocidos también como **ordinogramas**, los diagramas de flujo son una técnica que se hizo muy popular en los principios de la programación de ordenadores. Aunque actualmente prácticamente no se utilice, resulta muy útil para ir adquiriendo esa lógica de programación que nos permita diseñar y desarrollar programas, pues el uso de elementos gráficos para representar las estructuras típicas utilizadas en programación facilitará la comprensión de los mismos.

Los ordinogramas se emplean durante la fase de diseño de un programa y son independientes del lenguaje de programación utilizado para implementar el mismo, se basan en la representación gráfica, mediante una serie de símbolos, de la secuencia de operaciones lógicas a realizar por un algoritmo.

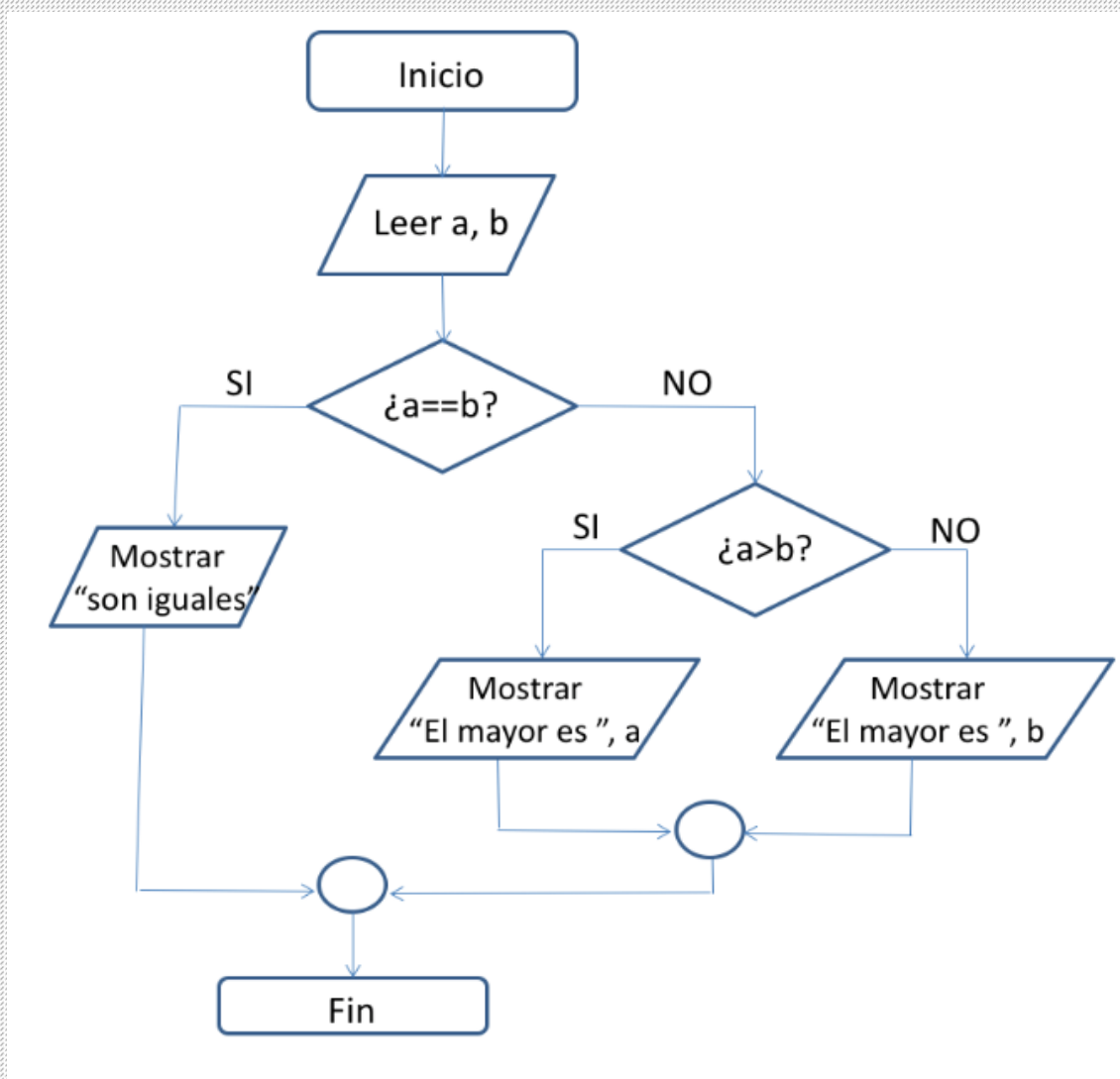
Para crear un ordinograma se utilizan símbolos bastante intuitivos y fáciles de recordar. La siguiente imagen nos muestra los símbolos básicos utilizados en la creación de un diagrama de flujo:



A la hora de utilizar estos símbolos para crear un ordinograma hay que tener en cuenta ciertas normas básicas que debemos cumplir:

- Todos los símbolos utilizados deben estar unidos por líneas de flujo solo horizontales y/o verticales.
- No se pueden cruzar las líneas de flujo. Evitar los cruces.
- No deben quedar líneas de flujo sin conectar.
- A un símbolo de proceso pueden llegarle varias líneas de flujo, pero solo puede salir una de él.
- Al símbolo de inicio no puede llegarle ninguna línea de flujo
- De un símbolo de fin no puede salir ninguna línea de flujo, pero si le pueden llegar varias.
- Se deben trazar los símbolos de manera que se pueda leer de arriba abajo y de izquierda a derecha.

El siguiente ordinograma de ejemplo corresponde al algoritmo de un programa encargado de leer dos números y mostrar si son iguales o, en caso de ser distintos, cuál de ellos es el mayor.



A continuación, pasaremos a comentar con más detalle los símbolos para creación de ordinogramas.

Símbolo de principio y fin

Como su nombre indica, se utilizan para especificar el principio y fin del programa. En su interior se escribe la palabra inicio o fin, según corresponda.



Símbolo de proceso

El símbolo de proceso especifica una operación realizada por el programa, como, por ejemplo, una operación aritmética con los datos recibidos en la entrada. La operación a realizar se especifica en el interior del rectángulo:

$$s=a+b$$

En el ejemplo anterior, se realiza la suma de dos datos representados por las variables *a* y *b* y el resultado se deposita en una tercera variable llamada "*s*". Es decir, la asignación se realiza de derecha a izquierda, que es como se hace en la mayoría de los lenguajes de alto nivel, utilizando el signo "=".

Para expresar las operaciones empleamos símbolos, como el signo + para la suma, el * para la multiplicación, el = para asignar un valor, etc. Estos símbolos son conocidos como **operadores**.

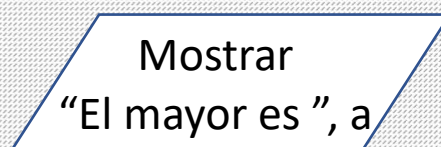
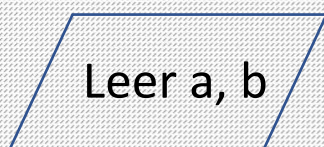
Cada lenguaje de programación tiene su propio juego de operadores, por lo que a la hora de utilizarlos en el diseño de un ordinograma no hay un convenio claro sobre su utilización; se debe procurar que sean lo más intuitivos posibles. Durante el estudio de la programación estructurada estableceremos un juego de operadores para ser utilizados durante el diseño de pseudocódigo.

Líneas de flujo

Las líneas de flujo representan el camino que sigue el algoritmo. Para representarlas se utilizan flechas, que parten de una operación y apuntan a la siguiente operación a realizar.

Símbolo de entrada/salida

Especifican una operación de entrada o salida, es decir, de entrada de datos al programa o de salida de datos desde el programa al exterior. La operación en concreto se indica en el interior del símbolo, utilizando un verbo como "leer", "mostrar", "imprimir", etc.



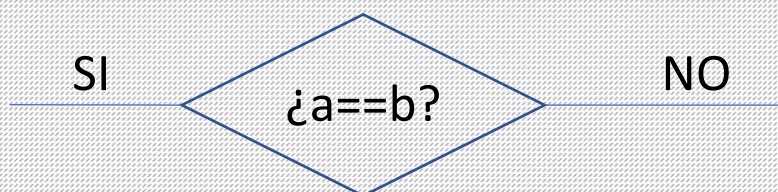
Al indicar la operación de entrada o salida en el diagrama de flujo, nos debemos abstraer del dispositivo en concreto que se utilizará para la operación. Por ejemplo, al indicar *Leer a, b*, simplemente indicamos que el programa recibe dos datos del exterior, sin especificar el dispositivo concreto utilizado para la lectura de los mismos (teclado, fichero, Web, etc.).

En el caso de las operaciones de entrada, a continuación del nombre de la operación (leer en este caso) se indican las variables donde se guardarán los datos recuperados, separadas por una coma. En el caso de las operaciones de

salida, a continuación del nombre de la operación se indicará la lista de variables con los datos a mostrar y si queremos que aparezca también una frase como en nuestro ejemplo, esta frase se escribirá entre comillas.

Símbolo de decisión

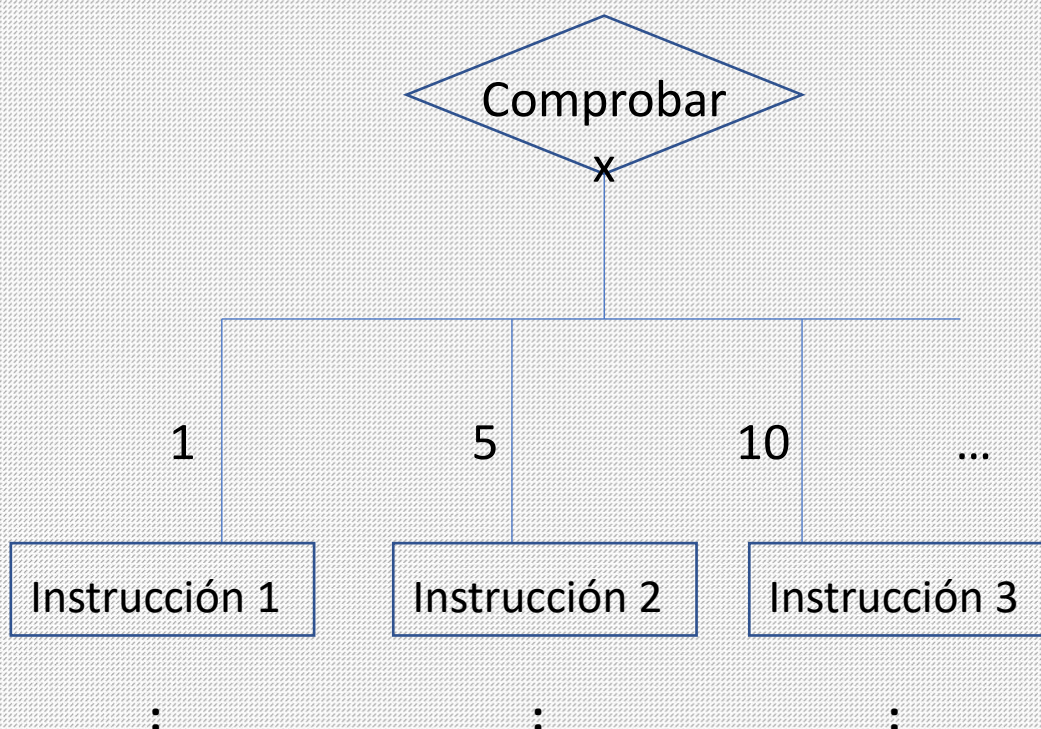
Mediante este símbolo con forma de rombo expresamos una operación de comprobación que puede alterar el flujo de ejecución de un programa. El resultado de la comprobación será de tipo verdadero/falso (SI/NO), de modo que si es verdadero el programa tomará un camino y si es falso tomará otro:



La operación de comprobación se indica entre interrogaciones en el interior del rombo y se utilizan los llamados operadores de tipo condicional, es decir, aquellos símbolos que se utilizan para comprobar la igualdad o desigualdad de datos, si un dato es mayor o menor que otro, etc. El resultado de estas comprobaciones siempre será verdadero o falso.

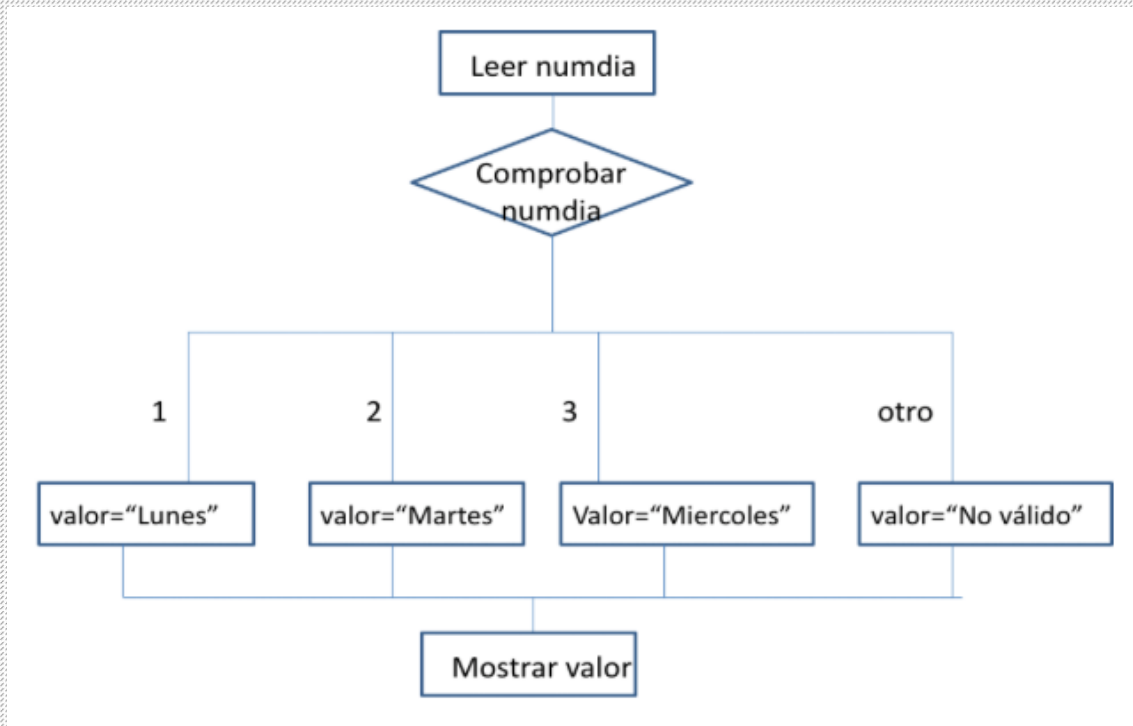
En el ejemplo que hemos presentado, se pregunta si el dato contenido en a es igual al contenido en b, para lo que usamos el operador doble igual ==, en vez del simple igual =, pues éste se reserva para operaciones de asignación.

Una variante de este tipo de operaciones son las decisiones con salida múltiple o **alternativas múltiples**. En este caso, la operación de comprobación puede dar como resultado distintos valores, no verdadero o falso, por lo que se podrán definir tantas salidas o caminos como posibles resultados queramos controlar. La forma de expresar una operación de decisión múltiple sería como se indica en el ejemplo:



Como vemos en el ejemplo, dependiendo del valor de un dato llamado "x", el programa puede tomar diferentes caminos y ejecutar diferentes grupos de instrucciones; si es 1, ejecutará las instrucciones del grupo "instrucción 1", si es 5 las del grupo "Instrucción 2", etc. Se puede definir también un camino para el caso de que el resultado de la expresión no coincida con ninguno de los valores definidos.

Después de ejecutar cada caso, el programa continuaría por un único camino. El siguiente ordinograma de ejemplo se encarga de leer un número de día y mostrar el día correspondiente, siempre que sea un número entre 1 y 3, sino mostrará que es un día no válido:

**Pregunta de refuerzo**

Queremos hacer un programa que lea un nombre de empleado y para cada empleado leído realizar una tarea diferente, ¿qué símbolo habrá que utilizar después de la lectura del nombre?:

- a) Símbolo de proceso
- b) Símbolo de decisión simple
- c) Símbolo de decisión múltiple
- d) Símbolo de entrada-salida

Solución: La respuesta correcta es la c. Como se trata de definir distintos caminos posibles en función de la variable que contiene el nombre, se debe utilizar una decisión múltiple. La *a* no es cierta porque no es un simple procesamiento, la *b* es falsa porque, como decimos, son varios posibles caminos a tomar, no solo dos y la *d* es falsa por el mismo motivo que *a*.

Acumuladores y contadores

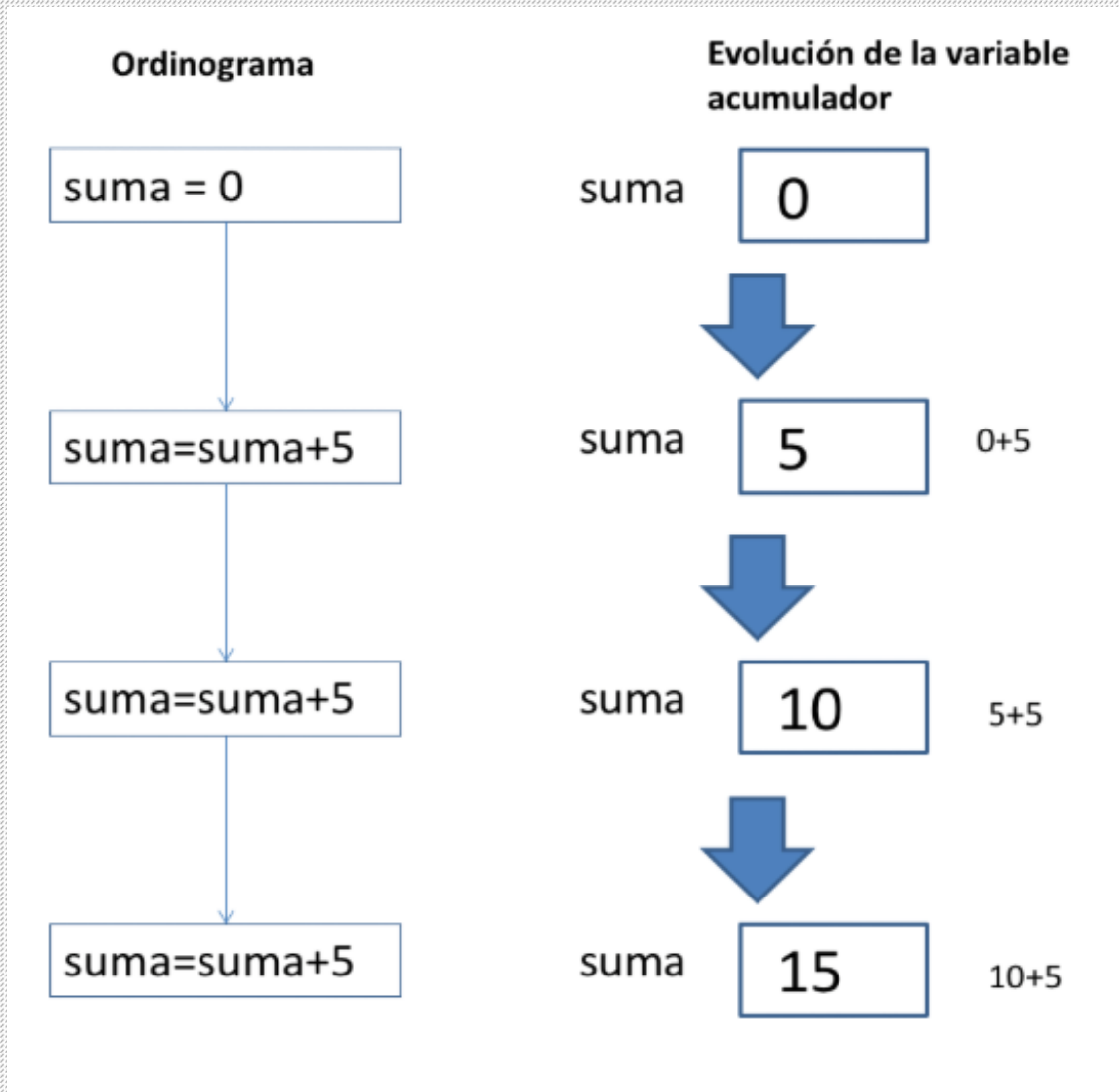
En la resolución de algoritmos nos vamos a encontrar numerosas situaciones en las que tenemos que realizar la suma o producto de varias cantidades. Estas operaciones no se realizarán en una única fase, sino que requerirán la realización de sumas o productos parciales de forma repetitiva hasta conseguir la cantidad final.

Para guardar los resultados de estas operaciones utilizaremos variables de tipo numérico, a las que se conoce en programación con el nombre de **acumuladores**, puesto que van acumulando los resultados parciales de la operación.

En cada suma parcial que se realiza con un acumulador se debe tomar el valor de la variable, sumarle o multiplicarle la cantidad y depositar el resultado de nuevo en la variable. Esto se representa en un ordinograma mediante el símbolo de proceso utilizado una expresión del tipo $\text{variable} = \text{variable} + \text{cantidad}$ o $\text{variable} = \text{variable} * \text{cantidad}$.

En el caso de los acumuladores de suma, la variable se inicializará a 0 mientras que para la multiplicación lo hará a 1.

En el siguiente ejemplo se muestra el caso de un pequeño programa que realiza la suma de los tres primeros múltiplos de 5 utilizando un acumulador:

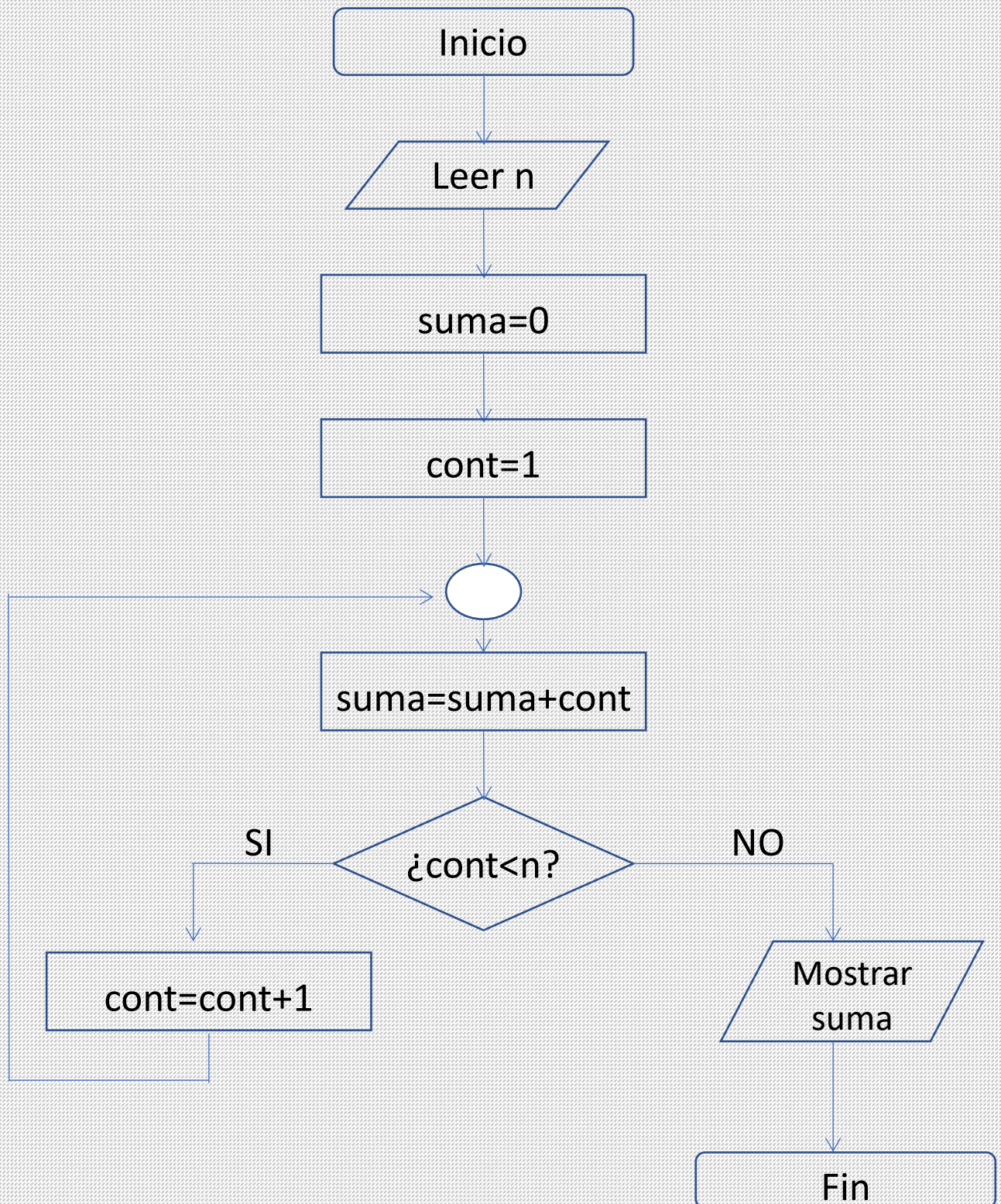


Otro tipo de variables numéricas comúnmente utilizadas en un programa informático son los **contadores**. Un contador es una variable que se va incrementando en una unidad cada vez que el programa realiza una determinada acción, a fin de ir contando el número de veces que dicha acción es realizada. También puede haber decontadores, es decir, variables que vayan restando uno a su valor repetidas veces.

La operación de incremento se representaría en un organigrama mediante el símbolo de proceso en el que se indicará una expresión del tipo $\text{variable} = \text{variable} + 1$

A continuación, te presentamos otro ejemplo de ordinograma que resuelve otro sencillo algoritmo, concretamente, se trataría de un programa que calcula la suma de todos los números naturales comprendidos entre 1 y un número leído.

Por ejemplo, si el número leído por el programa es 8, el resultado sería $1+2+3+4+5+6+7+8$. Este es su diagrama de flujo:

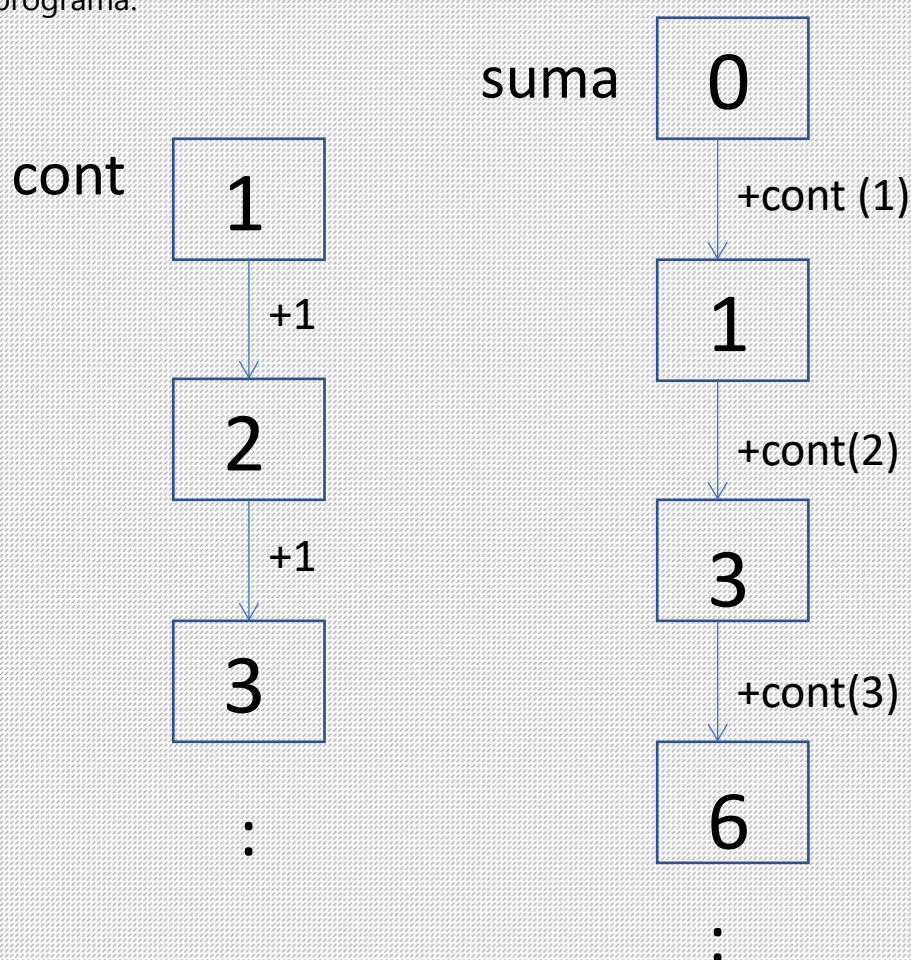


Como vemos, a parte de la variable n empleada para guardar el número leído, se utiliza una variable acumulador llamada *suma* y otra de tipo contador llamada *cont*.

En *suma* iremos acumulando los valores de las sumas parciales realizadas, mientras que *cont* irá contando los números desde 1 a *n* para ir sumándolos después con el contenido de *suma* mediante la instrucción $\text{suma} = \text{suma} + \text{cont}$. Así, cuando el programa ejecuta por primera vez la operación anterior, como *suma* se ha iniciado a 0 y *cont* a 1, la operación sería $\text{suma} = 0 + 1$, con lo que se guardaría el valor 1. Después, el programa pregunta si $\text{cont} < n$, como *cont* vale 1, el resultado de la pregunta es verdadero (suponemos que *n* es positivo y mayor que 1), por lo que se incrementa el valor de *cont* en una unidad (ahora valdrá 2) y se vuelve a la instrucción $\text{suma} = \text{suma} + \text{cont}$.

Al ejecutar por segunda vez la instrucción, tendríamos $\text{suma} = 1 + 2$ y de nuevo se volvería a realizar la consulta $\text{cont} < n$, así hasta que *cont* haya alcanzado a *n*, en cuyo caso ya se habrán acumulado en *suma* todos los números comprendidos entre 1 y *n*.

En el siguiente diagrama se ilustra la evolución de ambas variables en el programa:



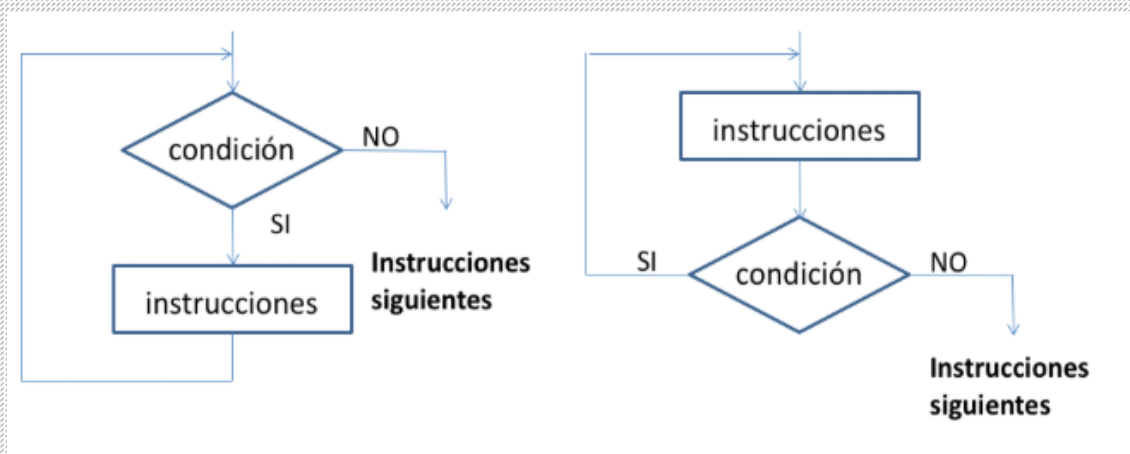
Los contadores y acumuladores siempre se utilizan en el interior de estructuras repetitivas, es decir, aquellas que contienen bloques de instrucciones que se ejecutan repetidamente.

Estructuras repetitivas

En el programa de ejemplo anterior hemos utilizado una estructura clásica en programación, que es la **estructura repetitiva**. No se trata de una operación en sí, sino de un conjunto de operaciones que se ejecutan mientras que se de una condición.

En nuestro caso, se trata de ir acumulando la suma de números naturales en la variable suma hasta alcanzar el número leído.

En una estructura repetitiva siempre encontramos una instrucción de tipo condicional, que determina si el grupo de instrucciones tiene que volver o no a ejecutarse. Se puede optar por comprobar la condición y ejecutar las instrucciones si se cumple, o ejecutar primero las instrucciones y comprobar después la condición:



En cualquiera de los casos, si la condición se cumple (resultado verdadero), el bloque de instrucciones vuelve a ejecutarse y en el momento en que deje de ejecutarse se continuará por otro camino.

Ejercicio de refuerzo

En un programa tenemos una variable llamada "intentos" que almacena el número de intentos disponibles que tenemos para poder realizar una determinada tarea. Cada vez que se falla la tarea se deberá quitar un intento, ¿qué instrucción de proceso debemos escribir para ello?:

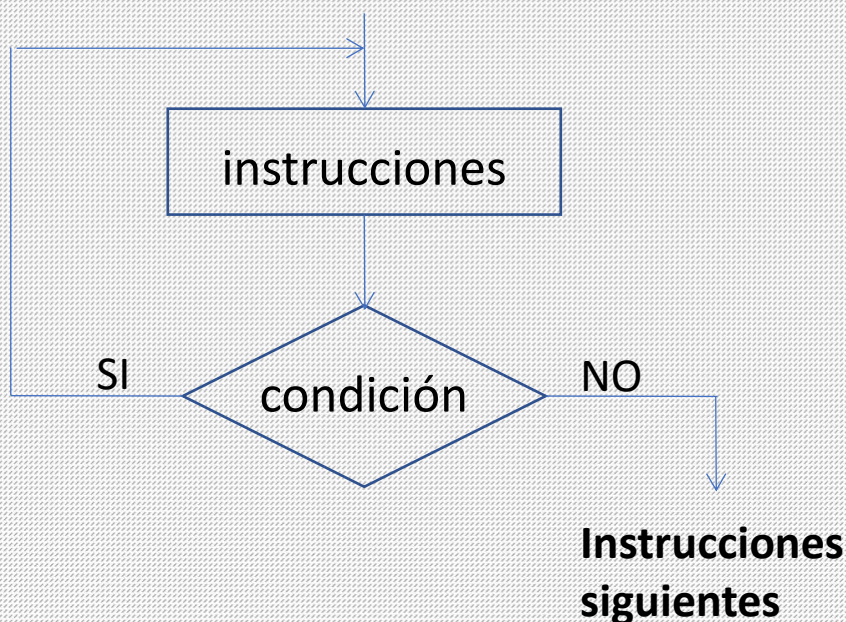
- a) `intentos=-1`
- b) `intentos=-intentos`
- c) `intentos-1`
- d) `intentos=intentos-1`

Solución: La respuesta correcta es la *d*. Esta instrucción, de tipo contador, es la que se debe indicar para decrementar en una unidad una variable. La instrucción indicada en *a* es incorrecta porque asigna directamente el valor -1 a la variable, mientras que la indicada en *b* lo que hace es asignar a la variable el valor actual, pero con signo negativo. Por su parte, la instrucción *c* resta uno a la variable, pero no actualiza el valor de la misma.

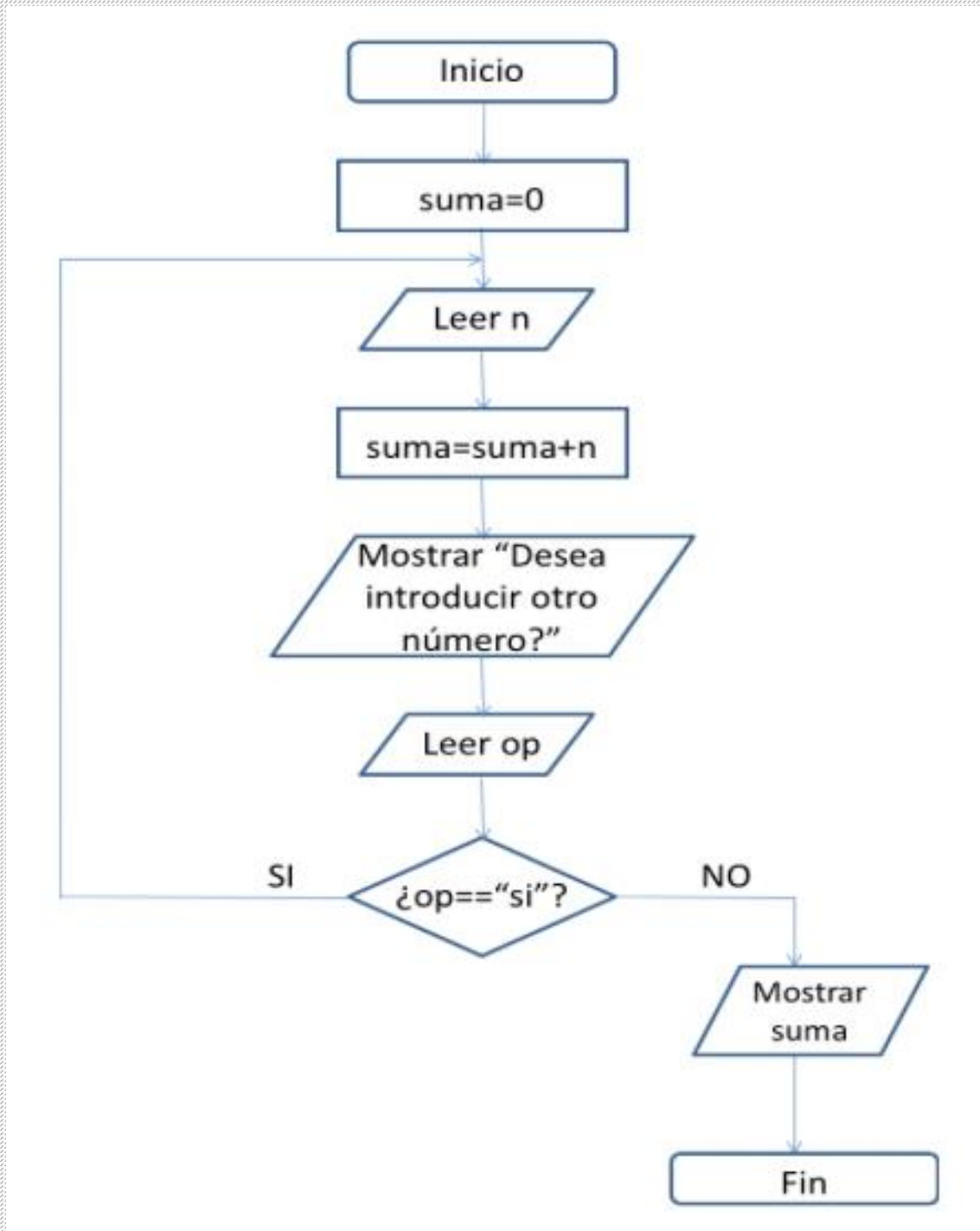
Ejercicio resuelto

A continuación, vamos a realizar otro ordinograma, consistente en un algoritmo que realice la lectura de números y los sume, hasta que el usuario del programa indique que ya no quiere introducir más números, momento en el cual se presentará la suma de todos los números leídos.

La idea es utilizar una estructura repetitiva que vaya leyendo los números, sumándolos y preguntando si se quiere realizar una nueva lectura, en cuyo caso se volverá a repetir el proceso. En este caso, el bloque de instrucciones se deberá ejecutar primero y consultar la condición al final:



Este sería el ordinograma correspondiente al algoritmo indicado:



En el ordinograma presentado, se utiliza una variable acumulador llamada "suma", donde se suman todos los números que se vayan leyendo. Las operaciones de lectura de número y suma en el acumulador se van repitiendo mientras el usuario responda "si" a la pregunta de si quiere introducir otro número.

Actividad práctica 2. Análisis de ordinograma

Actividad práctica 3. Realización de ordinograma

Pseudocódigo

El pseudocódigo consiste en expresar mediante un lenguaje coloquial las operaciones que describen el algoritmo asociado a un programa informático. Básicamente, se trata de traducir las operaciones que se describen en un organigrama a frases del lenguaje común, sin emplear símbolos.

El lenguaje utilizado para describir los algoritmos mediante pseudocódigo, debe ser fácil de interpretar por parte de una persona. Las palabras y expresiones utilizadas en los mismos deberán indicar de forma clara y precisa las tareas a realizar.

Y es que, el objetivo del pseudocódigo es intentar describir un algoritmo de la forma más detallada posible y, a su vez, próxima a un lenguaje de programación de alto nivel, de manera que la traducción final al código real durante la fase de implementación sea lo más sencilla posible, de ahí que se le llame *pseudocódigo*.

Seguidamente vamos a ver unos ejemplos que nos van a ir aclarando como plantear un pseudocódigo y las estructuras típicas que se utilizan. En este primer listado te presentamos la versión en pseudocódigo del primer ejemplo que realizamos con ordinogramas, correspondiente al algoritmo para leer dos números y mostrar el mayor de los dos:

Inicio

```
leer a, b
si (a==b) entonces
    mostrar "Son iguales"
sino
    si (a>b) entonces
        mostrar "El mayor es", a
    sino
        mostrar "El mayor es", b
    fin si
fin si
```

Fin

Como vemos, el esquema es similar al que se sigue en los ordinogramas, pero sin la utilización de símbolos. En el caso de estructuras complejas, como las

instrucciones alternativas, utilizamos palabras especiales como **si..entonces..sino..fin** si, para definir las operaciones y delimitar su ámbito. Al no seguir un esquema gráfico como el ordinograma, durante la definición de un pseudocódigo es conveniente indentar (escribir con espacios o tabulaciones a la izquierda) aquellas instrucciones o bloques de instrucciones que forman parte de otras, a fin de aclarar la estructura del algoritmo.

Aunque hay lenguajes de programación de alto nivel que si lo hacen, nosotros no haremos distinción entre mayúsculas y minúsculas a la hora de escribir un pseudocódigo.

En este otro ejemplo tenemos el bloque de pseudocódigo que describe el algoritmo a seguir por un programa encargado de mostrar la suma de todos los números naturales comprendidos entre 1 y un número leído:

Inicio

```
suma=0
cont=1
leer n
etiqueta1:
suma=suma+cont
si(cont<n) entonces
    cont=cont+1
    ir a etiqueta1
sino
    mostrar suma
fin si
```

Fin

En el pseudocódigo anterior vemos el uso de etiquetas, como la expresión "etiqueta 1:", que sirven para marcar ciertas partes del programa, de modo que podamos utilizar dichas etiquetas como referencia a la hora pasar el control del programa a ese punto.

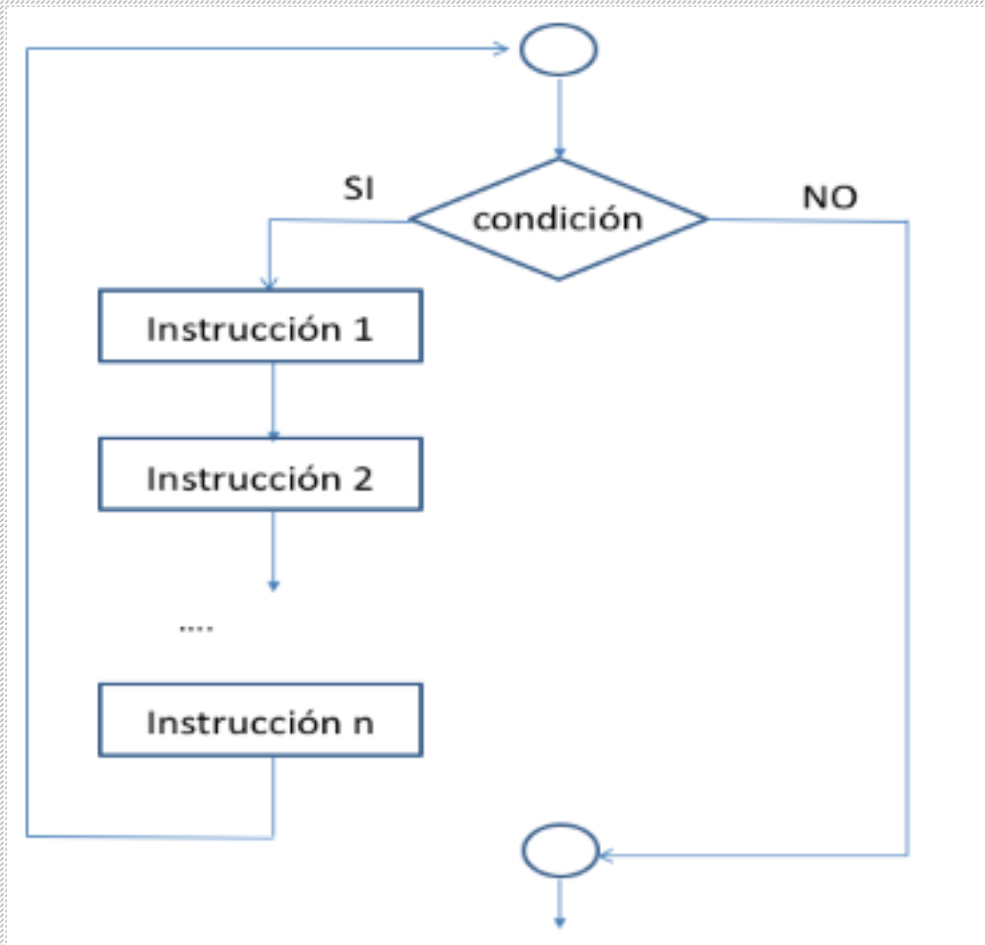
Sin la utilización de saltos, que muchas veces dificultan el seguimiento de un algoritmo, el programa anterior podría haberse escrito de esta otra manera utilizando una estructura repetitiva:

Inicio

```
suma=0
cont=1
leer n
mientras (cont<=n)
    suma=suma+cont
    cont=cont+1
fin mientras
mostrar suma
```

Fin

Con la **estructura repetitiva** *mientras*, lo que queremos expresar es que mientras se cumpla la condición indicada en el paréntesis, el programa tendrá que ejecutar el bloque de sentencias codificadas en su interior (las que aparecen entre *mientras* y *fin mientras*). Su funcionamiento es el siguiente: cuando el programa llega a la instrucción *mientras*, se comprueba la condición y, si esta es cierta, se ejecutará el bloque de sentencias. Después de ejecutar la última sentencia del bloque, el programa vuelve a comprobar la condición del *mientras* y, si vuelve a cumplirse, de nuevo se ejecuta el conjunto de instrucciones. Así sucesivamente hasta que la condición sea falsa, en cuyo caso el programa continuará con las instrucciones situadas después de *fin mientras*. Según indicamos en el apartado anterior, la estructura *mientras* tendría la siguiente forma representándola con este sistema:



Como ya hemos visto durante el estudio de los ordinogramas y analizaremos con más detalle durante el estudio de las técnicas de programación estructurada, existen distintas variantes a la hora de implementar una estructura repetitiva, como que la condición sea evaluada después de la ejecución de las instrucciones en vez de antes, o que el bloque de sentencias se ejecute un número definido de veces controlado por un contador.

El siguiente pseudocódigo corresponde al ejercicio resuelto presentado en el apartado anterior, consistente en un algoritmo que realiza la lectura y suma de números hasta que el usuario indica que no quiere introducir más números:

Inicio

 suma=0

 Hacer

 leer n

 suma=suma+n

 mostrar "¿Desea introducir otro número?"

 leer op

 Mientras (op!="sí")

 Mostrar suma

Fin

En este caso utilizamos la expresión *Hacer...Mientras*, para ejecutar primero el bloque de sentencias que se deben repetir y preguntas al final por la condición, de modo que, si esta condición se cumple, se volverá a ejecutar de nuevo el conjunto de instrucciones indicadas dentro de *hacer*.

Pregunta de refuerzo

Indica cual será el valor de la variable *l* al finalizar el siguiente bloque de instrucciones:

l = 1

Mientras(*l* < 10) hacer

l = *l* + 2

 Si(*l* = 7) entonces

l = 1

 Fin si

Fin Mientras

- a) 7
- b) 10
- c) 1
- d) Bucle infinito

Solución: La respuesta es la *d*. El bloque nunca termina, pues en el momento en que la variable *l* vale 7, se vuelve a poner a 1. Por este motivo, *a*, *b* y *c* son falsas.

Normas en la creación de pseudocódigos

Aunque cada persona es libre de definir sus propias normas a la hora de crear pseudocódigo, a fin de conseguir un pseudocódigo limpio y organizado, es necesario seguir una serie de reglas básicas y de sentido común, entre las que podríamos destacar:

- **Disponer de un juego limitado de instrucciones.** Es importante que a la hora de definir un algoritmo mediante pseudocódigo, utilicemos siempre el mismo juego de instrucciones y con ellas resolver cualquier problema de programación.
- **Utilizar estructuras lógicas de control.** Además de lo que serían las instrucciones de proceso (asignación de datos a una variable, operación aritmética entre variables, etc.), se debe disponer de un juego de instrucciones que expresen las operaciones de control de flujo que se llevan a cabo habitualmente en cualquier programa, como las alternativas simples y múltiples o las de tipo repetitivo, evitando saltos dentro del código.

- **Separación de datos y código.** Cuando se van a manejar varios datos en un programa, conviene separar la declaración de esos datos a utilizar de lo que serían las instrucciones de manipulación de los mismos.

Según la última de las reglas que acabamos de presentar, conviene a la hora de diseñar el pseudocódigo de un algoritmo definir previamente las variables de los datos que se van a manipular y, cuando proceda, inicializar las mismas. Esto permite, antes de empezar con el código, aclarar con qué datos vamos a trabajar.

A la hora de declarar las variables, se indicará el nombre de la variable seguido del tipo de dato con el que vamos a trabajar:

Nombre_variable tipo

A continuación, mostramos una versión del programa anterior en el que se realiza una separación entre datos y código:

Inicio

Datos:

suma entero

cont entero

n entero

Código:

suma=0

cont=1

leer n

mientras (cont<=n)

 suma=suma+cont

 cont=cont+1

fin mientras

mostrar suma

Fin

Los nombres de los tipos de datos los indicaremos según nuestro criterio, aunque será conveniente estandarizarlos. Más adelante daremos unas indicaciones sobre los tipos de datos habituales que podemos encontrarnos en un programa.

Veamos otro algoritmo de ejemplo en pseudocódigo. Corresponde al de un programa encargado de leer un número y mostrar el factorial de dicho número.

El factorial de un número se calcula multiplicando todos los números naturales menores de ese número hasta 1. Por ejemplo, el factorial del 5 se calcularía:
 $5*4*3*2*1$

He aquí el algoritmo:

Inicio

Datos:

factorial entero

cont entero

n entero

Código:

factorial=1

Leer n

cont=n

Mientras (cont >= 1)

 factorial=factorial*cont

 cont=cont-1

Fin mientras

mostrar factorial

Fin

En este ejemplo mostrado se trata de ir multiplicando los números naturales comprendidos entre 1 y el número leído.

Una vez más, para ir ejecutando repetidas veces una tarea tenemos que echar mano de la instrucción mientras.

Para controlar el número de multiplicaciones realizadas, utilizamos una variable contador que se inicializa al número leído y que con cada multiplicación se decrementa en una unidad. La variable *factorial* se utiliza como acumulador de las multiplicaciones.

Seguimiento de algoritmos

El seguimiento de algoritmos es un ejercicio mental que, como su nombre indica, consiste en seguir la lógica de un algoritmo, en nuestro caso escrito mediante pseudocódigo, tal y como lo haría un ordenador, a fin de determinar el valor final de ciertas variables.

Este ejercicio mental nos va ayudar a comprender el funcionamiento de las estructuras lógicas utilizadas en programación, lo que sin duda va a contribuir enormemente a adaptar nuestra mente a la lógica de programación.

Seguidamente, presentaremos una serie de algoritmos en pseudocódigo, tendrás que analizarlos, hacer un seguimiento de las operaciones que va realizado y responder a las cuestiones que en cada uno se plantean. Después de cada uno, te daremos la solución a las cuestiones planteadas con un pequeño razonamiento en cada caso.

Algoritmo 1

Determina el valor de las variables a y b al finalizar el siguiente programa:

Inicio

Datos:

a entero

b entero

a=7

b=5

Código:

si (a>b) entonces

a=a+2

sino

b=b-2

Fin si

si(a<=b) entonces

a=a+3

sino

b=b-5

fin si

Fin

Solución:

Los valores de las variables serán a=9 y b=0

Dado que el valor inicial de a es mayor que el de b, se cumpliría la primera condición y por tanto, a la variable a se le sumaría 2 ($a=7+2$) y b quedaría intacta. Después, la segunda condición resultaría falsa (9 no es menor que o igual que 5), por lo que entraría a ejecutar el bloque *sino*, dejando a la variable b a 0.

Algoritmo 2

Al finalizar el siguiente programa, ¿qué valor se imprimiría de la variable I?

Inicio

Datos:

A entero

I entero

A=-5

I=4

Código:

Mientras (I<=30) hacer

I=I*3

Si (I<5) entonces

I=A*2

Sino

I=I+A

Fin si

Fin mientras

Mostrar "El valor de I es ", I

Fin

Solución:

Se mostrará: "El valor de I es 43". Al llegar por primera vez a la condición del mientras, dicha condición se cumple al ser I (vale 4) menor que 30, por tanto en la variable I se guardará el valor 12 (resultado de multiplicar 4, que es el valor de I, por 3). Después, se comprueba la condición, y como no es cierta (12 no es menor que 5), se ejecuta $I=12-5$, resultando el valor 7 dentro de I. Al llegar al final de mientras, el programa vuelve de nuevo arriba a evaluar la condición de mientras, que vuelve a cumplirse, almacenándose en I el valor 21 (que es $7*3$).

De nuevo no se cumple la condición del si, y en la variable I se almacena $21-5$, que son 16. Y otra vez a comprobar la condición del mientras que vuelve a cumplirse, almacenándose en I el valor 48 (que es $16*3$), sigue sin cumplirse la condición del si, y se resta de nuevo el valor 5 a la variable, quedándose en 43. Pero ahora, al volver a la condición del mientras, esta ya no se cumple, por lo que se sale de la instrucción repetitiva y se va a la operación de imprimir el valor de la variable

Algoritmo 3

Indica los valores de las variables R, T y X al finalizar el siguiente algoritmo:

Inicio

Datos:

X entero

B entero

R entero

T entero

Código:

X=14

B=10

R=5

T=1

Mientras (R>50) hacer

X=X+B

R=R*X

T=T+1

Fin mientras

Fin

Solución:

Los valores de las variables serán R=5, T=1 y X=14. Los valores de las variables permanecen intactos, pues desde el primer momento la condición R>50 no se cumple y el programa no entra a ejecutar las instrucciones definidas dentro del bloque mientras, provocando la finalización del programa.

Algoritmo 4

Indica los valores de las variables R, T y X al finalizar el siguiente algoritmo:

Inicio

Datos:

X entero

B entero

R entero

T entero

Código:

X=18

B=7

R=2

T=1

Mientras (R<50) hacer

X=X+B

R=R*X

T=T+1

Fin mientras

Fin

Solución:

Los valores de las variables serán R=50, T=2 y X=25. En este caso, cuando el programa llega a evaluar la condición del mientras, esta se cumple ($2 < 50$), por lo que la variable X toma el valor 25 ($18 + 7$), R adquiere el valor 50 ($25 * 2$) y T el valor 2. Al llegar al final del mientras, se vuelve a evaluar la condición de este y, como ya no se cumple, no se entra a ejecutar sus instrucciones por lo que el programa termina con los valores indicados de las variables.

Algoritmo 5

Indica los valores de las variables X, Y y Z al finalizar el siguiente proceso. El símbolo <> se utiliza para indicar "distinto de".

Inicio

Datos:

X entero

Y entero

Z entero

Código:

X=0

Y=7

Z=-4

Mientras (X <> Z) hacer

Si (Y<25) entonces

Y=Y+4

Sino

Si(Z<0) entonces

Z=Z+2

X=X+1

Sino

Z=Z+1

X=X-1

Fin si

Y=y+3

Fin si

Fin mientras

Fin

Solución:

Los valores de las variables serán X=1, Y=16 y Z=1. Al llegar a la instrucción mientras, como la condición se cumple se entra en el bloque y se evalúa la condición del si, que no se cumple, por lo que se entra en el bloque sino. La condición del si anidado también se cumple, por lo que Z toma el valor -2 y X el valor 1. Tras la ejecución del si anidado, se ejecuta Y=Y+3, tomando esta variable el valor 10. De nuevo se evalúa la condición del mientras y vuelve a cumplirse, pero no la del primer si, por lo que se entra de nuevo en el sino y también en el si anidado, quedando las variables Z=0, X=2 e Y=13. De nuevo se comprueba la condición de mientras que se cumple una vez más, pero no la del si y, en este caso, tampoco la del si anidado, por lo que Z toma el valor 1 y X

también, quedando $Y=16$. En la siguiente iteración de mientras, la condición deja de cumplirse y el programa finaliza.

Actividad práctica 4. Seguimiento de algoritmos

Actividad práctica 5. Realización de algoritmos en pseudocódigo

Programación estructurada

Como ya hemos visto, un algoritmo es un conjunto de operaciones que resuelven un problema específico. Pero muchos problemas de programación, especialmente los que tienen cierta complejidad, pueden resolverse con diferentes algoritmos, algunos de ellos más óptimos y eficientes que otros. Esta situación se nos planteó durante la realización de los ejemplos de pseudocódigo que vimos anteriormente, cuando el programa que se encargaba de calcular la suma entre 1 y un número leído lo resolvimos de dos maneras, una utilizando saltos de un punto a otro del programa y la otra mediante el uso de una instrucción repetitiva, siendo esta última la más eficiente. De ahí que a finales de los años sesenta surgiera una forma de programar que permitía crear programas fiables y eficientes, además de fáciles de comprender. Esta nueva forma de programar se definía a través del teorema del programa estructurado o **teorema de la estructura**.

Teorema de la estructura

Según el teorema de la estructura, todo programa, por muy complejo que sea, puede ser diseñado con sólo tres tipos de instrucciones básicas:

- Instrucciones secuenciales
- Instrucciones alternativas o condicionales
- Instrucciones repetitivas

Este teorema, deja fuera de este grupo a instrucciones del tipo salto incondicional, que dificultan el seguimiento de los programas y los hace poco fiables.

Ventajas de la programación estructurada

Con la programación estructurada, diseñar algoritmos para resolver un problema informático sigue siendo una labor que requiere esfuerzo, creatividad y precaución, pero aplicando sus principios podemos obtener las siguientes ventajas:

- Programas más fáciles de comprender. Al carecer de saltos a otras partes del código, el algoritmo se puede seguir secuencialmente, lo que facilita su comprensión
- Estructura de programa clara. Los tres tipos de instrucciones que se utilizan tienen un funcionamiento claro y definido.
- Reducción de errores. Al limitar el juego de instrucciones a utilizar y tener una estructura definida, se cometen menos errores al diseñar el algoritmo y por tanto, al programar
- Facilidad en la detección de errores. No solamente se comenten menos errores, sino que los que se cometen son más fáciles de detectar y corregir.
- Programas más simples y rápidos. Estructurar un programa garantiza la solución más sencilla en cuanto a cantidad de código utilizado, además de más eficiente.

Tipos de instrucciones

A continuación, vamos a ir analizando los tres tipos de estructuras lógicas que se utilizan en programación estructurada, para aplicarlos después en diversos ejercicios de ejemplo que iremos realizando.

Instrucciones secuenciales

Instrucciones secuenciales son aquellas que se ejecutan en secuencia, es decir, una detrás de otra. Realmente, en un programa estructurado todas las sentencias son secuenciales, solo que unas pueden ser simples, como la inicialización de variables, operaciones aritméticas entre datos, lectura de números, y otras pueden ser complejas, como una instrucción alternativa o repetitiva.

Centrándonos en las operaciones simples, como hemos indicado, pueden consistir en la inicialización de alguna variable:

I=0

Operaciones entre datos:

I=I+4

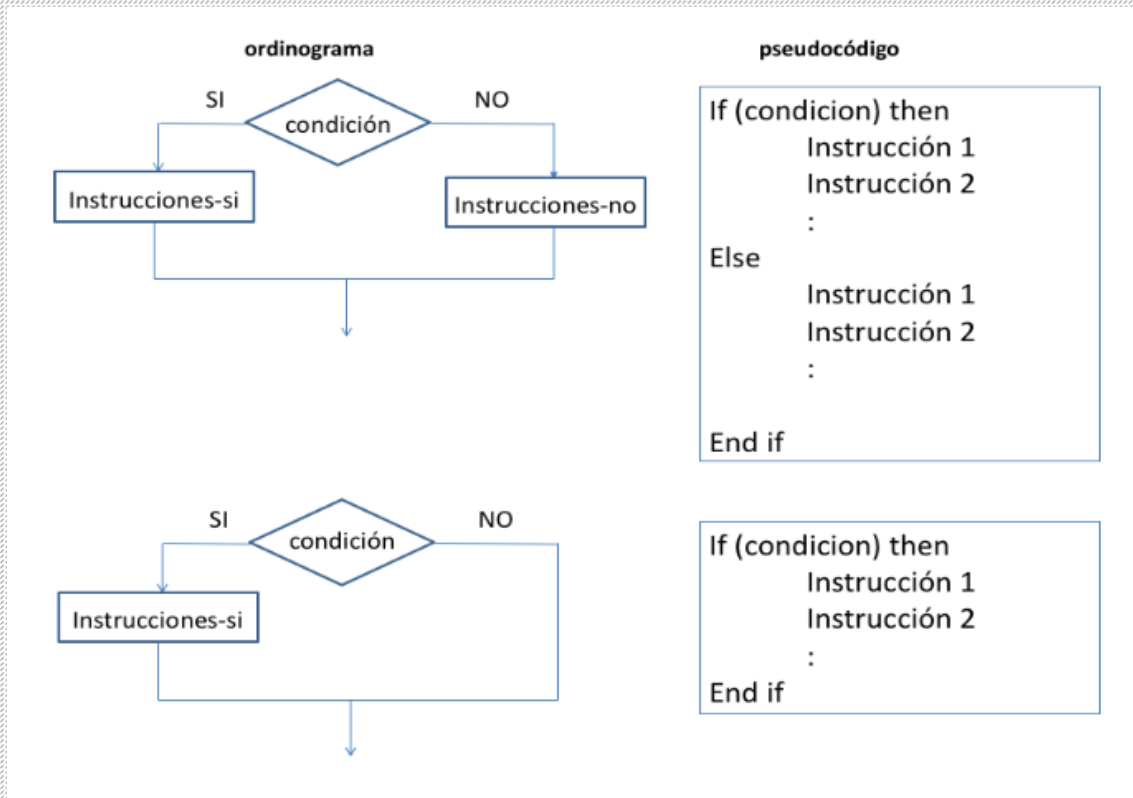
*A=B*I*

O operaciones de entrada y salida:

Leer N

Imprimir Suma

En la siguiente imagen se ilustra cómo se representan estas instrucciones, tanto en un ordinograma como mediante pseudocódigo:



Instrucciones alternativas

Las instrucciones alternativas o condicionales nos permiten tomar decisiones en un programa, de modo que ante una condición el programa pueda tomar un camino en caso de que dicha condición se cumpla, o ejecutar sentencias diferentes si no se cumple.

Las instrucciones alternativas pueden ser de dos tipos:

- Alternativas simples
- Alternativas múltiples

Alternativas simples

En este tipo de alternativas tenemos solamente dos caminos posibles, el que se tomará si la condición evaluada es verdadera y el que se seguirá si es falsa.

En la siguiente imagen tenemos su representación tanto en ordinograma como en pseudocódigo:

Como puedes observar, hay dos variantes de esta instrucción, en la primera se evalúa la condición, si es verdadera se ejecutan unas instrucciones y si es falsa otras diferentes. Al final, el programa continúa por el mismo camino.

En la otra variante, si la condición es falsa no se ejecuta ninguna instrucción especial, el programa continúa por el mismo camino por el que seguiría después de ejecutar las instrucciones para la condición verdadera.

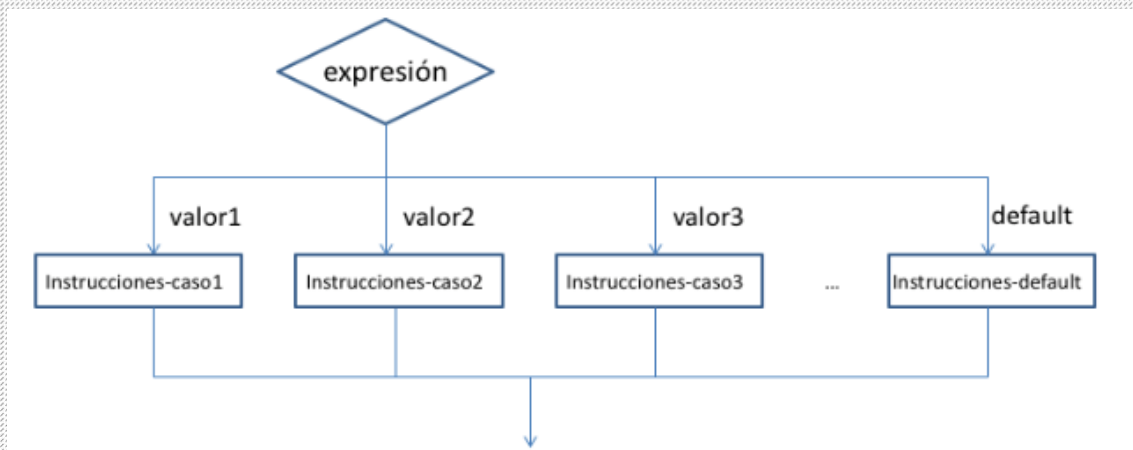
A la hora de expresar estas instrucciones mediante pseudocódigo utilizaremos el convenio de emplear palabras en inglés, de esta manera, puesto que las instrucciones que utilizan los lenguajes de programación de alto nivel para definir este tipo de estructuras son también palabras en inglés, se facilita la traducción final del diseño en pseudocódigo a código en la fase de implementación.

Alternativas múltiples

En una estructura de tipo alternativa múltiple, no es una condición de tipo verdadero o falso la que determina el camino a seguir por el programa, sino una expresión que puede dar como resultado múltiples valores.

Para cada valor que nos interese controlar se definirá un grupo de instrucciones a ejecutar diferente. Opcionalmente, se puede definir un conjunto de instrucciones a ejecutar en caso de que el resultado de la expresión no coincida con ninguno de los valores controlados. Tras la ejecución de cualquiera de los casos, el programa continuará por un camino común.

A continuación, se muestra el diagrama de flujo equivalente a esta instrucción:



En pseudocódigo, podríamos definirlo de la siguiente manera:


```
Switch(expresion)
  Case valor1:
    Instrucciones
  Case valor2:
    Instrucciones
  :
  Default:
    Instrucciones
End Switch
```

A la hora de indicar los valores de cada caso, podemos utilizar un valor concreto, una lista de valores separados por comas o un rango de valores definido por la expresión **valorinicial To valorfinal**, si bien algunos lenguajes de programación de alto nivel solo permiten indicar valores concretos a la hora de evaluar las salidas de una instrucción de este tipo.

El final de las instrucciones de cada caso vendrá determinado por el caso siguiente, y tras ejecutar un caso, el programa continuará con la ejecución de la sentencia indicada después del **End Switch**.

En el siguiente ejemplo te mostramos un pseudocódigo de un algoritmo encargado de leer una nota de un examen y mostrar la calificación correspondiente:

Inicio

Datos:

nota entero

Algoritmo:

Leer nota

Switch(nota)

Case 0 to 4

Imprimir "Suspendido"

Case 5

Imprimir "Aprobado"

Case 6

Imprimir "Bien"

Case 7, 8

Imprimir "Notable"

Case 9, 10

Imprimir "Sobresaliente"

Default

Imprimir "Nota no válida"

End Switch

Fin

Pregunta de refuerzo

Indica cual de las siguientes características no corresponde con una ventaja de la programación estructurada:

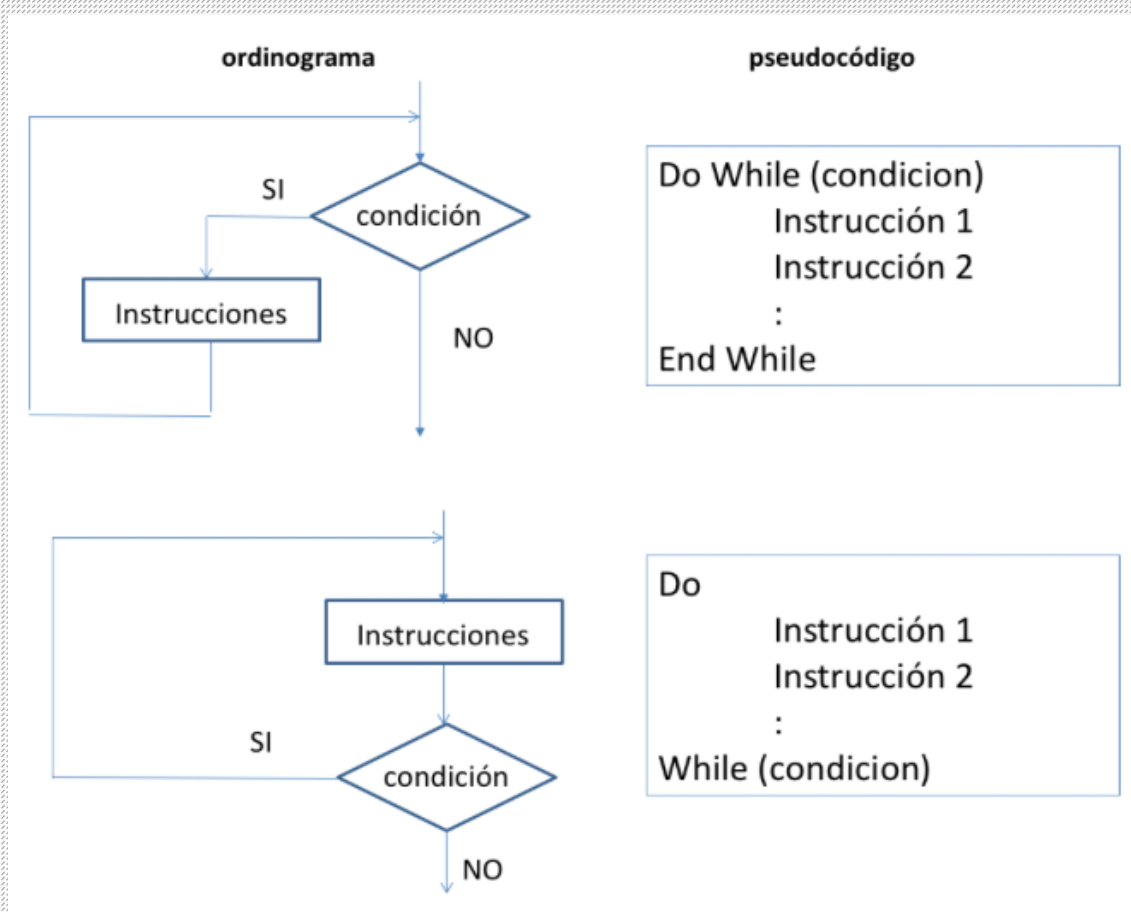
- a) Programas más fáciles de seguir
- b) Facilidad en la detección de errores
- c) Reutilización de código
- d) Eliminación de saltos en el programa

Solución: La respuesta correcta es la c. Realizar programas estructurados proporciona varias ventajas, como las indicadas en *a*, *b* y *d*, pero la reutilización de código no es una de ellas. Esta característica es proporcionada por la programación modular, no por la programación estructurada.

Instrucciones repetitivas

Las instrucciones de tipo repetitivo permiten ejecutar un bloque de sentencias varias veces en función del cumplimiento de una condición. Se trata de un tipo de instrucción muy importante en programación, con la que podemos realizar cálculos iterativos y recorrer tablas u otras estructuras de datos complejas. Utilizando adecuadamente esta instrucción, podemos evitar saltos en un programa, que es una de las claves de la programación estructurada.

En el siguiente esquema tenemos las dos formas clásicas en las que se puede presentar este tipo de instrucciones:



En el primer caso, lo primero es evaluar la condición y si esta se cumple, se ejecutará el bloque de sentencias definido entre **Do While** y **End While**. Al finalizar la ejecución de dicho bloque, se vuelve a evaluar la condición de nuevo y si vuelve a cumplirse, otra vez se ejecuta el bloque de sentencias. Es de esperar que la ejecución del bloque de sentencias actúe sobre los elementos utilizados en la condición, de forma que este deje de cumplirse en algún momento.

En el segundo caso, primero se ejecutan las instrucciones indicadas dentro del **Do.. While** y después se evalúa la condición, de modo que, si esta se cumple, el bloque de instrucciones volverá a ejecutarse de nuevo. Con esta estructura se garantiza que el bloque de instrucciones se ejecuta al menos una vez, aunque la condición no se cumpla desde el principio.

Podemos ver un ejemplo de utilización de esta segunda forma de la estructura repetitiva While en la siguiente versión del programa de las notas en la que se repite la lectura de la nota hasta que el valor introducido sea válido, es decir, entre 0 y 10:

Inicio

Datos:

nota entero

Algoritmo:

Do

Leer nota

While(nota < 0 OR nota > 10)

Switch(nota)

Case 0 To 4

Imprimir "Suspenso"

Case 5

Imprimir "Aprobado"

Case 6

Imprimir "Bien"

Case 7, 8

Imprimir "Notable"

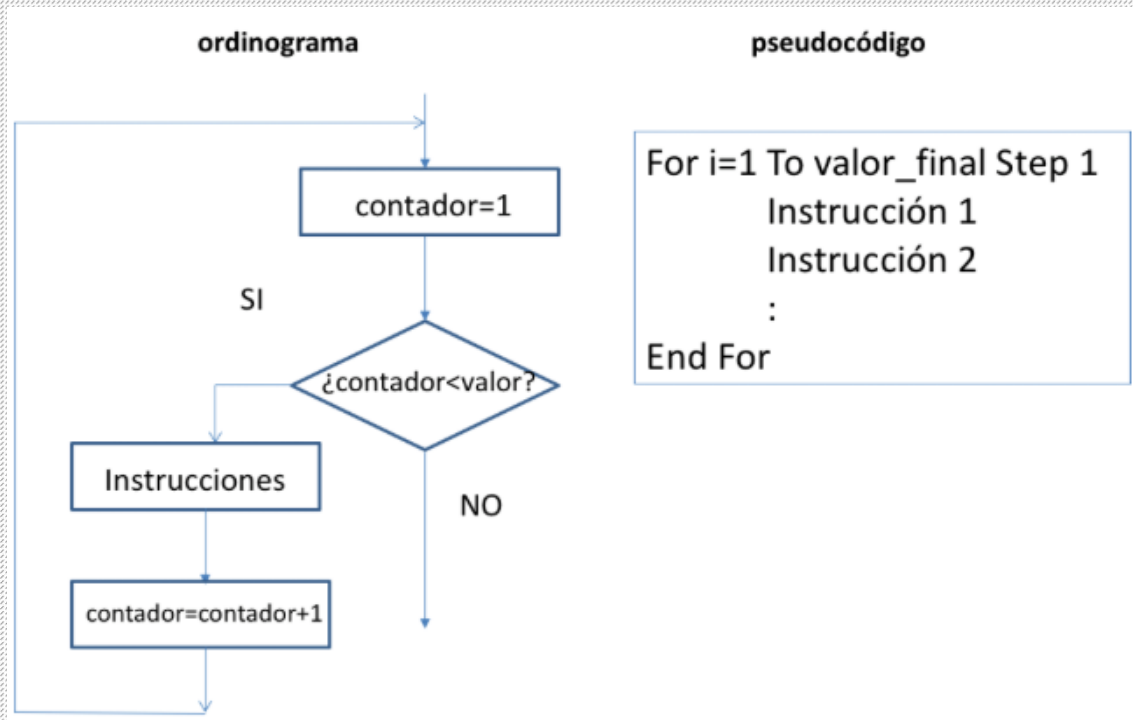
Case 9, 10

Imprimir "Sobresaliente"

End Switch

Fin

Un caso habitual de instrucción repetitiva es aquella que ejecuta el bloque de instrucciones un número concreto de veces, el cual viene determinado por una variable contador que se va incrementando en cada iteración. Este tipo de instrucciones repetitivas suele representarse en los lenguajes de programación de alto nivel mediante una instrucción llamada **For** en vez de *While*. La siguiente ilustración nos muestra su representación en forma de ordinograma y pseudocódigo:



Desde que la variable de control, llamada *i* en este caso, toma un valor inicial hasta que alcanza un valor final, se ejecutan las instrucciones del bloque. Al entrar en la instrucción *for*, la variable se inicializa al valor indicado, en este caso 1 y, si el valor de esta variable es menor o igual que el valor final, se ejecuta el bloque de sentencias definido en su interior. Al finalizar la ejecución del bloque, se incrementa en 1 la variable de control y se vuelve a comprobar si ha alcanzado el valor final, así hasta que la variable haya recorrido todos los valores enteros entre el inicial y el final.

En la instrucción *step* se indica el incremento a realizar al finalizar cada iteración y, aunque normalmente es 1, puede ser cualquier otro valor.

El algoritmo del programa encargado de calcular la suma de todos los números naturales comprendidos entre 1 y un número leído podría representarse de la siguiente manera:

Inicio

Datos:

N entero

I entero

Suma entero

Código:

Suma = 0

Leer N

For I=1 to N Step 1

Suma=Suma+I

End For

Imprimir "La suma final es ",Suma

Fin

Otros elementos de programación estructurada

Además de las instrucciones de control, a la hora de crear un programa estructurado se utilizan otros elementos de los que ya hemos hecho uso en los ejemplos de algoritmos presentados hasta el momento, pero que vamos a formalizar a continuación.

Tipos de datos

En todos los programas se manejan datos que, como ya hemos visto en los pseudocódigos realizados, se almacenan en variables y se opera con ellos a través de estas.

Cada lenguaje de programación de alto nivel tiene su propio juego de tipos de datos, pero de cara a poder definir los distintos tipos de datos con los que vamos a tratar en un pseudocódigo, vamos a definir nuestro propio conjunto de tipos. Será un conjunto básico, formado por los tipos más comunes:

Nombre tipo	Significado
Integer	Numérico entero
Decimal	Numérico decimal
String	Texto
Date	Fecha
Boolean	Lógico (true o false)

Así, en la zona de definición de datos, cuando vayamos a indicar el tipo de una variable utilizaremos los nombres indicados en la tabla anterior.

Operadores

Los operadores permiten realizar operaciones con los datos dentro de un programa. Ya hemos visto como utilizar algunos de los más habituales en los ejemplos que te hemos ido presentando a lo largo del módulo.

En la siguiente tabla resumimos los operadores que utilizaremos en el diseño de nuestros programas, clasificados por tipos:

Tipos de operadores	Operadores
Aritméticos	+, -, *, /, %
Condicionales	<, >, <=, >=, ==, <>
Lógicos	AND, OR, NOT

Los operadores aritméticos los utilizaremos con tipos numéricos, es decir, Integer y Decimal. Los cuatro primeros son para las operaciones de suma, resta, multiplicación y división, mientras que el operador "%" se utiliza para calcular el resto de la división entre dos números. Este operador ya lo utilizamos en los ejercicios donde teníamos que comprobar si un número era par o impar. Los operadores condicionales se emplean en las condiciones de las instrucciones de tipo *If* y *While*. Siempre devolverán un valor de tipo boolean (true o false). Los operadores <, >, <=, >= los emplearemos con tipos numéricos, mientras que los de igualdad y desigualdad (== y <>) los emplearemos con cualquier tipo de datos.

En cuanto a los operadores lógicos, se usarán para realizar las tres operaciones lógicas básicas:

- AND. El resultado es verdadero si las dos expresiones lo son:
(3>1 AND 5<10) // el resultado es verdadero
(4>2 AND 2>8) //el resultado es falso porque la segunda expresión //lo es
- OR. El resultado es verdadero si alguna de de las expresiones lo es:
(10>3 OR 5<1) //el resultado es verdadero
- NOT. El contrario a un valor boolean:
(NOT 4<6) //el resultado es falso

Comentarios

Los comentarios los utilizamos en un programa, y también en pseudocódigo, para incluir notas del programador que no forman parte de la sintaxis lógica del programa.

Su misión es aclarar a la persona que está leyendo el programa el uso de alguna instrucción o expresión, por lo que ayudan a entender el funcionamiento del mismo.

Los comentarios los podemos incluir en cualquier parte del pseudocódigo, y siempre irán precedidos por los símbolos //. En la explicación de los operadores lógicos hemos visto unos ejemplos de uso.

Ejercicios de ejemplo

A continuación, vamos a realizar algunos ejercicios en los que aplicaremos las estructuras de programación estructurada comentadas.

Ejercicio 1

Realizar un programa que lea dos números y muestre la suma de todos los números comprendidos entre los dos leídos.

Solución:

Inicio

Datos:

N1, N2, Mayor, Menor Integer

Suma, I Integer

Código:

Suma=0;

Leer N1

Leer N2

If (N1>N2) Then

Mayor=N1

Menor=N2

Else

Mayor=N2

Menor=N1

End If

For I=Menor To Mayor Step 1

Suma=Suma+I

End For

Imprimir "La suma final es ", Suma

Fin

Ejercicio 2

Realizar un programa que vaya leyendo números y los sume hasta la introducción de un número negativo, en cuyo momento dejará de leer números y mostrará la suma de números positivos leídos hasta entonces.

Solución:

Inicio

Datos:

N Integer

Suma Integer

Código:

Suma=0

Leer N

Do While(N>=0)

Suma=Suma+N

Leer N

End While

Imprimir "Suma de números leídos ", Suma

Fin

Ejercicio 3

Realizar un algoritmo para un programa que lea 10 números y nos diga cuántos de ellos son pares y cuantos impares.

Solución:

Inicio

Datos:

Pares, Impares, I, N Integer

Código:

Pares=0;

Impares=0;

For I=1 To 10 Step 1

Leer N

//el número es par si el resto de su división entre 2 es 0

If(N%2=0) Then

Pares=Pares+1;

Else

Impares=Impares+1;

End If

End For

Imprimir "Total de números pares ", Pares

Imprimir "Total de números impares ", impares

Fin

Ejercicio 4

En este ejercicio se pide diseñar un algoritmo para un programa que lea un texto y nos indique cuantas vocales tiene dicho texto. Para su realización, supondremos que existe una función llamada "ObtenerLetra" que, a partir de un texto y un entero nos devuelve la letra que ocupa la posición indicada por el entero (más adelante se tratará en el curso el tema de las funciones), y otra función llamada "Longitud" que nos devuelve el total de caracteres de la cadena.

Solución:

Inicio

Datos:

Letra, Cadena String

I, Vocaes, Lon Integer

Código:

Vocales=0

Leer Cadena

Lon=Longitud(Cadena)

//recorre las letras de la cadena desde la posición 1 hasta la longitud

For I=1 To Lon Step 1

Letra=ObtenerLetra(Cadena, I)

Switch(Letra)

Case "a", "e", "i", "o", "u":

Vocales=Vocales+1

End Switch

End For

Mostrar "El total de vocales es: ", Vocales

Fin

Ejercicio 5

Utilizando las funciones Longitud y ObtenerLetra comentadas en el ejercicio anterior, realizar un programa que lea un texto y lo muestre invertido. Por ejemplo, si el texto es "hola" mostrará "aloh".

Inicio

Datos:

Texto, Aux String

I Integer

Leer Texto

Código:

Aux="" //inicializa la variable de texto a cadena vacía

//Para que el contador del for vaya descontando en vez de contar


```
//el valor de step será negativo
For I= Longitud(Texto) To 1 Step -1
    //concatena, es decir, une, el contenido actual de la variable
    //con la letra devuelta por la función
    Aux=Aux+ ObtenerLetra(Texto, I)
End For
```

Actividad práctica 6. Realización de algoritmos con programación estructurada

Arrays

Un array, también llamado tabla o matriz, es un conjunto de datos de un mismo tipo al que se accede mediante una única variable. Cada dato tiene una posición asignada dentro del array:

	1	2	3	4	5	6	7
codigos	10	25	1	30	17	15	9

En el ejemplo de la figura anterior, tenemos un array de 7 elementos de tipo número entero, contenido en la variable *codigos*. En este caso, la posición asignada al primer elemento es la 1, a siguiente la 2 y así sucesivamente.

Los arrays los utilizamos cuando queremos manejar un grupo de datos de manera conjunta, realizando una serie de operaciones sobre todos los datos del array. Al acceder a todos ellos mediante la misma variable y tener un índice asociado, el array se puede recorrer mediante una instrucción de tipo repetitivo, con lo que podemos definir las operaciones dentro de la estructura repetitiva y que estas se apliquen para cada uno de los elementos del array.

Definición de un array

A la hora de declarar una variable de tipo array, lo haremos como con cualquier otra variable, solo que debemos indicar el rango de índices para establecer el tamaño del array:

codigos 1 to 7 Integer

En la instrucción anterior se ha declarado la variable *codigos* del ejemplo, consistente en un array de enteros de 7 elementos.

Hay lenguajes de programación en los que el índice del primer elemento está fijado en 0 y no se puede modificar, por lo que a la hora de declarar el array simplemente se indicará el tamaño del mismo.

Acceso a los elementos del array

Para poder acceder a una posición del array utilizaremos la expresión *variable[indice]*. Como vemos, después del nombre de la variable se indica entre corchetes la posición a la que queremos acceder. Por ejemplo, para almacenar un valor en la primera posición del array códigos sería:

```
codigos[1]=100
```

y para decirle al programa que nos muestre el valor de la última posición sería:

```
Imprimir codigos[7]
```

El siguiente programa de ejemplo almacena los 10 primeros números pares en un array de enteros y después lo recorre para mostrar su contenido:

Inicio

Datos:

nums 1 to 10 Integer

i Integer

Código:

For i=1 To 10 Step 1

 nums[i]=(i-1)*2

End For

For i=1 To 10 Step 1

 Imprimir nums[i]

End For

Fin

Ejercicios de ejemplo

Ejercicio 1

En este caso vamos a desarrollar un algoritmo algo más complejo. Se trata un programa encargado de leer las notas de unos alumnos y guardarlas en un array. Si se introduce una nota no válida (negativa o mayor de 10), se le volverá a solicitar al usuario. Serán un total de 15 notas. Posteriormente, el programa mostrará los siguientes datos:

Nota media:

Nota máxima

Nota mínima:

Total de aprobados:

Solución:

Inicio

Datos:

notas 1 to 15 Integer

i, nota, mayor, menor, media, aprobados Integer

Código:

For i=1 To 15 Step 1

Leer nota

//mediante la siguiente instrucción repetimos la lectura

//de la nota mientras no sea correcta

Do While(nota<0 OR nota>10)

Imprimir "Nota incorrecta, introdúzcala de nuevo"

Leer nota

End While

notas[i]=nota

End For

//inicialización de variables. Las variables mayor y menor se

//al valor del primer elemento del array

mayor=nota[1]

menor=nota[1]

media=0

aprobados=0

//recorremos de nuevo el array para realizar los cálculos

For i=1 To 15 Step 1

//si encuentra una nota mayor, actualiza la

//variable mayor con dicha nota

If(nota[i]>mayor) Then

mayor=nota[i]

End If

//si encuentra una nota menor, actualiza la

//variable menor con dicha nota

If(nota[i]<menor) Then

menor=nota[i]

End If

media=media+nota[i]

If(nota[i]>=5) Then

aprobados=aprobados+1


```
        End If
    End For
    Media=media/15 //para calcular la media aritmética
    Imprimir "Nota media: ", media
    Imprimir "Nota máxima:", mayor
    Imprimir "Nota mínima:", menor
    Imprimir "Total de aprobados:", aprobados
Fin
```

Ejercicio 2

En este segundo ejercicio de arrays, vamos a desarrollar un algoritmo para un programa que simule el lanzamiento de un dado 1000 veces. Tras realizar los lanzamientos, el programa nos indicará el porcentaje de veces que ha salido cada número. Para poder resolver este ejercicio, vamos a suponer la existencia de una función llamada "generarAleatorio" que nos generará un número aleatorio entre los dos que le indiquemos como argumentos.

Solución:

Inicio

```
Datos:
Lanzamientos 1 To 6 Integer
I, Num Integer
For I=1 To 6 Step 1
    //inicializamos los elementos del array a 0
    Lanzamientos[I]=0
End For
For I=1 To 1000 Step 1
    //se generan los numeros y se incrementa el elemento
    //que ocupa la posición indicada por el número obtenido
    Num=generarAleatorio(1,6)
    Lanzamientos[Num]=Lanzamientos[Num]+1
End For
//Mostramos los resultados
For I=1 To 6 Step 1
    //utilizamos el signo + para unir el texto con
    //el valor de la variable
    Mostrar "Porcentaje de "+I, Lanzamientos[I]*100/1000
End For
```


Pregunta de refuerzo

Tenemos un array cuyo tamaño está especificado por la variable *tam*, y en el que el índice del primer elemento es el 0. ¿Cuál de las siguientes instrucciones permitiría recorrer el array desde el primer elemento hasta el último?:

- a) For *i*=1 To *tam* Step 1
- b) For *i*=0 To *tam*-1 Step 1
- c) For *i*=0 To *tam* Step 1
- d) For *i*=1 To *tam*-1 Step 1

Solución: La respuesta correcta es la *b*. Si el índice del primer elemento es 0, el del último será tamaño-1. La respuesta *a* es falsa porque comenzaría por el segundo elemento, no el primero, y se saldría de los límites del array. La *c* es falsa porque se saldría de los límites del array y la *d* es falsa porque se saltaría el primer elemento.

Actividad práctica 7. Realización de algoritmos con arrays

Arrays multidimensionales

El tipo de array que hemos tratado hasta ahora es un array de una dimensión, pero podemos tener arrays de varias dimensiones. Un array de dos dimensiones, por ejemplo, sería como una tabla organizada en filas y columnas:

	1	2	3	4	5	6	7
1							
2							
3							
4							

El array de la imagen anterior sería de dos dimensiones y tendría un tamaño de 4 filas x 7 columnas = 28 elementos. Para declarar una variable de este tipo de array utilizaremos la expresión: elementos 1 to 4, 1 to 7 Integer

La variable la hemos llamado *elementos* y se deben indicar los rangos de índices de cada dimensión, separados por una coma.

Para acceder a cada posición, utilizaremos doble corchete, uno para cada índice. Por ejemplo, para almacenar un número en la quinta columna y segunda fila sería: `elementos[2][5]=30`

	1	2	3	4	5	6	7
1							
2					30		
3							
4							

Para arrays con más de dos dimensiones el proceso sería el mismo, aunque a partir de tres dimensiones no podríamos hacer una representación gráfica del mismo.

En el siguiente ejemplo declaramos un array de tres dimensiones de tamaño 4x7x3:

`datos 1 to 4, 1 to 7, 1 to 3 Integer`
`datos[2][5][1]=100`

Para recorrer un array multidimensional, se deberán utilizar estructuras repetitivas anidadas, una por cada dimensión. El siguiente algoritmo de ejemplo simula la generación de un tablero de ajedrez mediante un array bidimensional, representando los cuadrados negros con 1 y los blancos con 0:

Inicio

Datos:

Tablero 1 to 8, 1 to 8 Integer

I,K Integer

For I=1 To 8 Step 1

For K=1 To 8 Step 1

If((I+K)%2=0) Then

Tablero[I][K]=0


```

Else
    Tablero[I][K]=1
End If
End For
End For
Fin

```

Ejercicio de ejemplo

En el siguiente ejercicio, vamos a crear el algoritmo de un programa encargado de leer los datos de 10 personas y guardarlos en un array. Cada persona se caracteriza por un nombre y dirección de correo. Posteriormente, el programa solicitará una dirección de correo al usuario y realizará la búsqueda de la persona en el array, si la encuentra mostrará su nombre y si no, mostrará un mensaje indicando que no la ha encontrado:

Inicio

```

Datos:
//Array para guardar las 10 parejas de datos
Basedatos 1 to 10, 1 to 2String
Fila Integer
Nombre, Email String
Código:
For Fila=1 To 10 Step 1
    //solicita el nombre e email de cada persona
    //y lo guarda en una fila del array
    Leer Nombre
    Leer Email
    Basedatos[Fila][1]=Nombre
    Basedatos[Fila][2]=Email
End For
Leer Email
//pone la variable Nombre a cadena vacía porque
//se utilizará para guardar el nombre de la persona
// y si no la encuentra, se quedará con ese valor
Nombre=""
For Fila=1 To 10 Step 1
    If(Basedatos[Fila][2]=Email) Then
        Nombre=Basedatos[Fila][1]
    End If
End For
//comprueba si la variable es distinta de cadena vacía
//si es así, es que encontró la persona
If(Nombre<>"") Then

```



```
        Mostrar Nombre
    Else
        Mostrar "Persona no encontrada"
    End If
Fin
```

Pregunta de refuerzo

Dado el siguiente array

Parametros 1 To 7, 1 To 20, 1 To 5 Integer.

¿Cuál de las siguientes estructuras permitiría recorrer todo su contenido?

a) For I=7 To 20 Step 1

```
        For K=1 To 5 Step 1
```

b) For I=1 To 7 Step 1

```
        For K=1 To 20 Step 1
```

```
            For N=1 To 5
```

c) For I=1 To 7 Step 1

```
        For K=7 To 20 Step 1
```

```
            For N=5 To 20
```

d) For I=1 To 5 Step 1

```
        For K=1 To 7 Step 1
```

```
            For N=5 To 20
```

Solución: La respuesta correcta es la *b*. Utiliza un *for* para recorrer cada dimensión, anidiando cada uno con el *for* de la dimensión anterior. La *a* no es correcta porque solo recorre dos dimensiones. La *c* y *d* no son correctas porque recorren solo parcialmente cada dimensión.

Actividad práctica 8. Arrays multidimensionales

Ordenación de arrays

Una de las tareas más habituales que se llevan a cabo con arrays es la ordenación de los mismos. Ordenar un array consiste en colocar sus elementos siguiendo un determinado criterio de ordenación, por ejemplo, en un array de números este criterio podría consistir en colocar los elementos de menor a mayor o de mayor a menor.

En cualquier caso, para conseguir ordenar un array debemos seguir algún método que nos permita definir un algoritmo que garantice dicha ordenación.

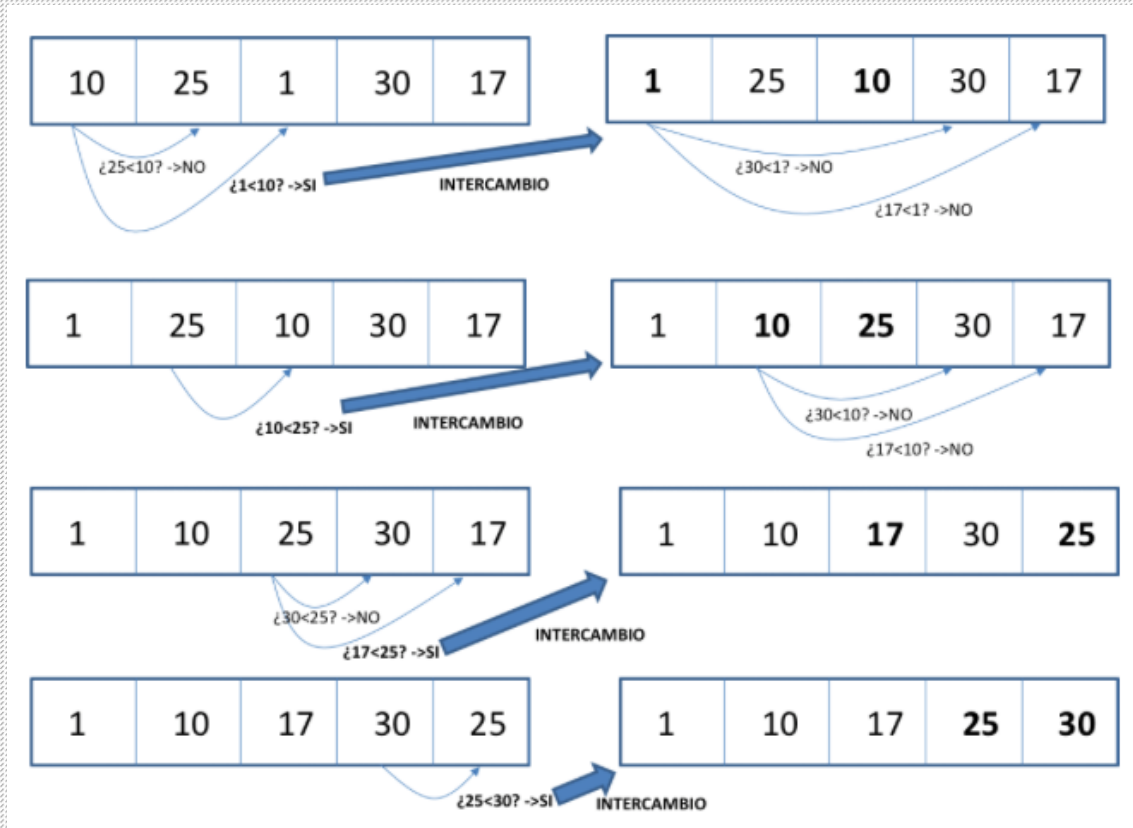
Uno de esos métodos es el conocido como **método de la burbuja**.

El método de la burbuja consiste en ir realizando ordenaciones parciales de cada elemento del array, de manera que al final conseguiremos tener ordenados todos los elementos.

Más concretamente, el procedimiento a seguir para aplicar este método en el caso de un array numérico que deba ser ordenado de menor a mayor, consistirá en lo siguiente: para cada elemento del array, se comprueba si alguno de los siguientes elementos es menor que él, si es así realizamos el intercambio de los elementos y seguimos con las comprobaciones, si encontramos un nuevo elemento menor, volvemos a intercambiar, y así hasta llegar al último.

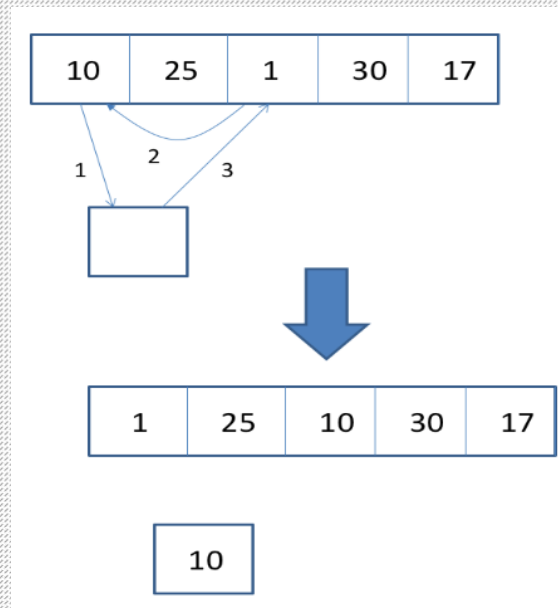
Si el proceso anterior se realiza con todos los elementos del array, excepto el último, se garantiza que el array quedará ordenado al finalizar todas las comprobaciones.

En la siguiente imagen se ilustra gráficamente el proceso de ordenación de un array de ejemplo, formado por los números 10, 25, 1, 30 y 17, utilizando el método de la burbuja que se acaba de describir:



La primera fila representa las comprobaciones del primer elemento del array, en la segunda fila se indican las comprobaciones con el segundo elemento, y así sucesivamente. Cuando se cumple la condición, se realiza el intercambio de los elementos y se continúa comprobando con los siguientes.

De cara a aplicar mediante pseudocódigo el método de la burbuja para ordenar un array, debemos utilizar dos instrucciones for anidadas, la primera para recorrer cada uno de los elementos del array y la segunda para realizar las comprobaciones entre cada elemento y los siguientes. Así mismo, se necesitará una variable auxiliar para poder realizar el intercambio de las posiciones, tal y como se ilustra en la siguiente imagen:



Para poder intercambiar el contenido de dos posiciones del array, primero se salva el contenido de una de las posiciones en la variable auxiliar, después se vuelca en dicha posición el contenido de la segunda y en tercer lugar, el valor de la variable auxiliar se lleva a la posición que fue volcada en la primera.

El siguiente algoritmo realiza la ordenación del array de notas que utilizamos en el ejercicio de ejemplo presentado al principio del apartado, utilizando la variable *aux* para realizar el intercambio de los datos:

```

For I=1 To 15 Step 1
  For K=I+1 To 15 Step 1
    If(notas[K]<notas[I]) Then
      //realización del intercambio
      aux=notas[I]
      notas[I]=notas[K]
      notas[K]=aux
    End If
  End For
End For

```

Ejercicio resuelto

En el siguiente ejercicio resolveremos un algoritmo para un programa que realizará la lectura de números y los guardará en un array, siempre que el número sea múltiplo de 5, si no, volverá a realizar la lectura del número hasta que cumpla esa condición. Este proceso se repetirá hasta conseguir un total de 10 múltiplos de 5, momento en el cual el programa mostrará los números recibidos, ordenados de mayor a menor.

Según las especificaciones indicadas, el algoritmo quedará como se indica en el siguiente listado:

Inicio

Datos:

I, K Integer

Numeros 1 to 10 Integer

Num,aux Integer

Código:

For I=1 To 10 Step 1

Mostrar "Introduce Numero"

//realiza la lectura del número y si no es

//múltiplo de 5 vuelve a leerlo

Leer Num

Do While (Num%5<>0)

Mostrar "No es múltiplo de 5, vuelve a introducirlo!"

Leer Num

End While

Numeros[I]=Num

End For

//ordenación del array

For I=1 To 10 Step 1

For K=I+1 To 10 Step 1

If(Numeros [K]> Numeros [I]) Then

//realización del intercambio

aux=Numeros[I]

Numeros [I]= Numeros [K]

Numeros [K]=aux

End If

End For

End For

//recorre de Nuevo el array para mostrar su contenido

For I=1 To 10 Step 1

Mostrar Numeros[I]

End For

Fin

Pregunta de refuerzo

Si queremos intercambiar los valores de las posiciones 1 y 5 de un array Datos, utilizando la variable auxiliar T, la manera de hacerlo sería:

- a) `Datos[1]=Datos[5], T=Datos[5], Datos[1]=T`
- b) `Datos[5]=Datos[1], T=Datos[1], Datos[5]=T`
- c) `T=Datos[1], Datos[1]=Datos[5], Datos[5]=T`
- d) `T=Datos[5], Datos[1]=Datos[5], Datos[5]=T`

Solución: La respuesta correcta es la c. Primero se salva el valor de la primera posición en la variable auxiliar, después se vuelca el valor de la posición 5 en la primera y finalmente, se guarda el valor de T (que contiene el valor inicial de la posición 1) en la posición 5. En la respuesta a ambas posiciones quedarían con el valor de Datos[5], al igual que en la d, mientras que en la respuesta b, las dos quedarían con el valor de Datos[1]

Actividad práctica 9. Realización de pseudocódigo de un programa con ordenación de arrays

Programación Modular

La programación modular representa la evolución natural de la programación estructurada. Basada en la idea de "divide y vencerás", la programación modular consiste en dividir un programa grande en bloques más pequeños (módulos), a fin de que el desarrollo pueda acometerse con mayor facilidad.

Ventajas de la programación modular

Son numerosas las ventajas que obtenemos al dividir un programa grande en bloques más pequeños, entre ellas destacan:

- **Simplificación del desarrollo.** Al dividir un problema grande en problemas más pequeños, podemos concentrarnos en acometer esos problemas pequeños de manera independiente, facilitando su resolución.
- **Menor número de errores.** Dividiendo el desarrollo en programas más simples, la probabilidad de cometer errores en estos mini programas es menor que en un programa grande. Además, los errores cometidos son más fáciles de detectar y, por tanto, de resolver.
- **Reutilización de código.** Se trata de una ventaja muy interesante, pues al disponer de módulos encargados de realizar una determinada tarea, estos módulos pueden utilizarse en distintas partes del programa donde

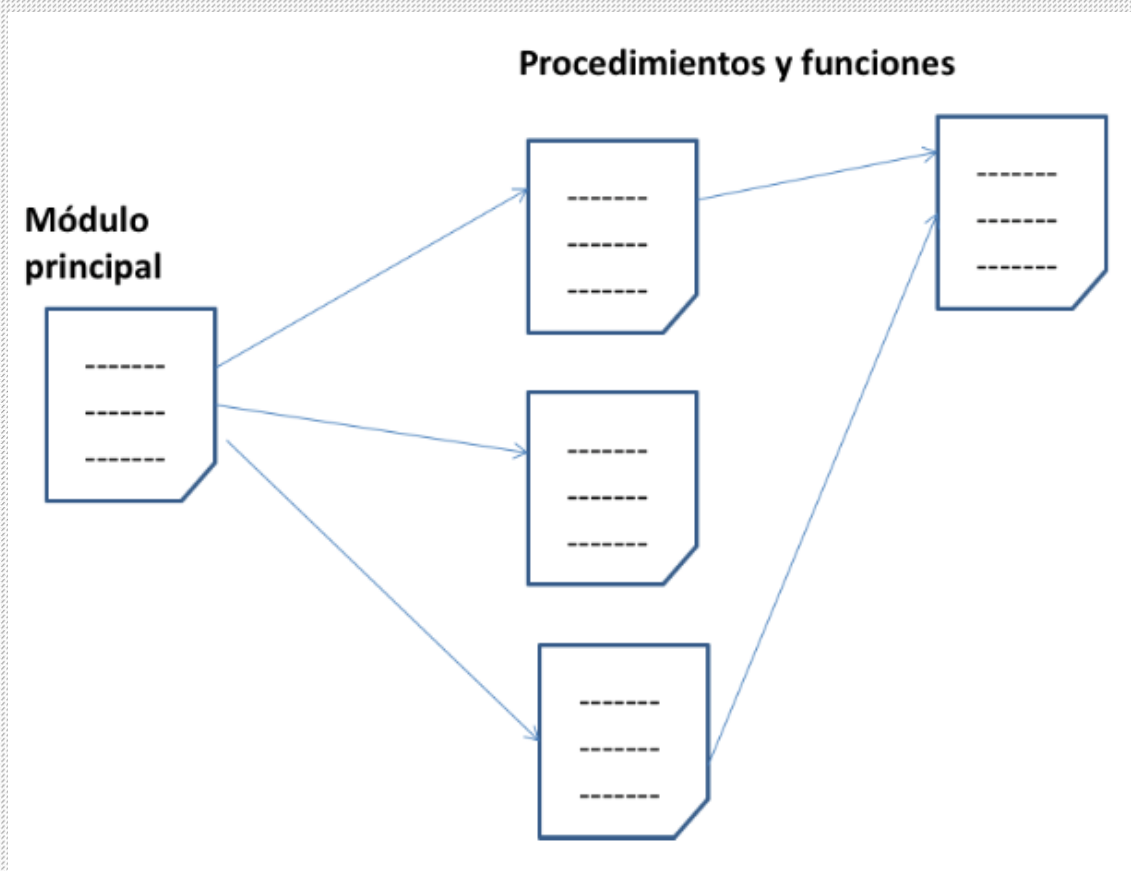
se requiera realizar dicha tarea, evitando tener que reescribir las instrucciones de nuevo.

- **Separación entre capas.** Con la programación modular se facilita la separación entre capas de la aplicación, es decir, que cada capa se pueda programar por separado. La división de una aplicación en capas permite dividir la aplicación en funcionalidades, por ejemplo, la capa de presentación se encargaría de todo lo relativo a entrada y salida de datos, mientras que la capa de lógica de aplicación se encargaría de procesar los datos y realizar los cálculos con los mismos.

Procedimientos y funciones

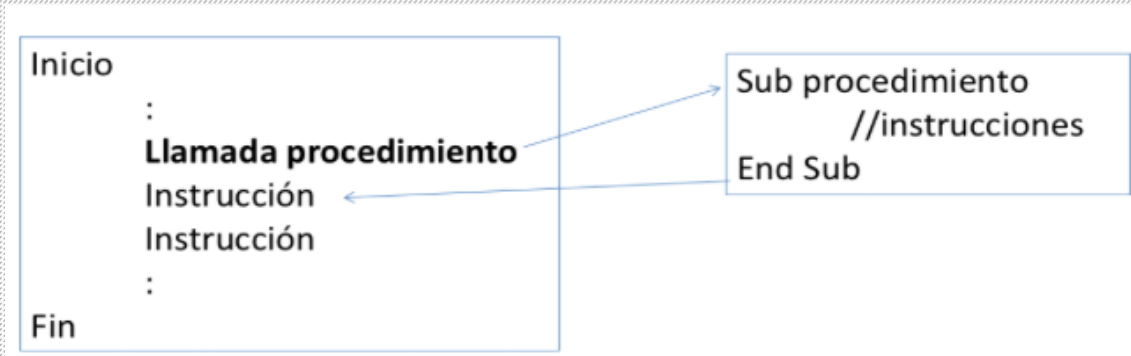
La manera que tenemos en programación de aplicar la modularidad es a través de los **procedimientos y las funciones**. Ambos representan bloques de código que realizan alguna tarea y que pueden ser llamados desde otra parte del programa.

Por ejemplo, las diferentes tareas a realizar pueden ser implementadas en procedimientos y funciones y desde un módulo principal hacer llamadas a los mismos cuando se requiera ejecutar la tarea que llevan asociada:



Procedimientos

Un procedimiento es un bloque de código que se ejecuta cuando es llamado desde otra parte del programa. Tras finalizar su ejecución, el control de la ejecución se pasa a la línea siguiente a la que provocó la llamada.



En notación de pseudocódigo, un procedimiento lo definiremos de la siguiente manera:

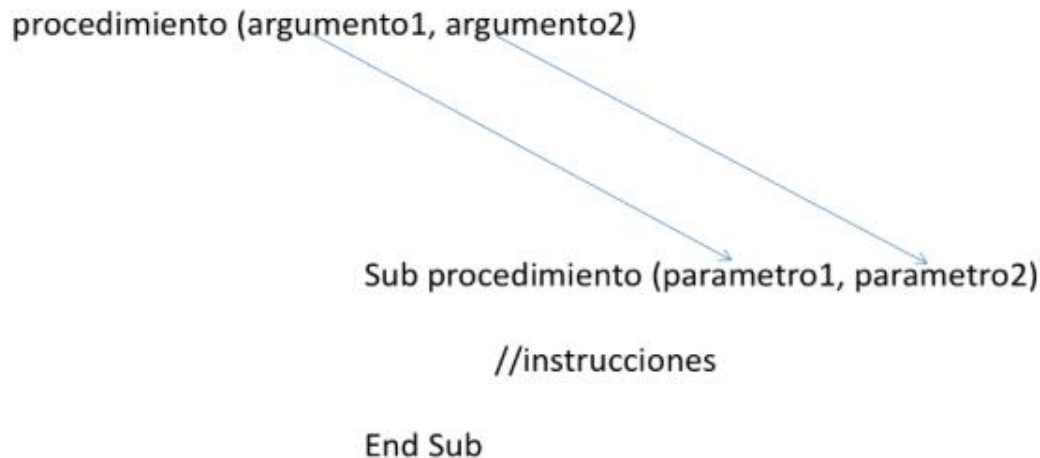
```
Sub nombre_procedimiento(parametro1, parametro2, ...)
//instrucciones
End Sub
```

La palabra *Sub* indica el inicio del procedimiento, a continuación se indica el nombre del mismo y entre paréntesis los parámetros, que no son más que las variables donde se volcarán los datos que el procedimiento recibirá en la llamada. De cara a indicar los parámetros, se especificará también el tipo de datos que representan.

Para llamar a un procedimiento desde otra parte del programa utilizaremos el nombre del mismo, seguido de la lista de argumentos o valores de llamada entre paréntesis:

```
nombre_procedimiento (argumento1, argumento2,...)
```

Los argumentos son los datos que se pasan en la llamada al procedimiento, los cuales **se volcarán en los parámetros** indicados en el mismo.



procedimiento (argumento1, argumento2)

Sub procedimiento (parametro1, parametro2)

//instrucciones

End Sub

The diagram illustrates the relationship between a procedure call and its definition. Two blue arrows originate from the parameters 'argumento1' and 'argumento2' in the call 'procedimiento (argumento1, argumento2)'. These arrows point to the parameters 'parametro1' and 'parametro2' in the definition 'Sub procedimiento (parametro1, parametro2)'. Below the definition, the lines '//instrucciones' and 'End Sub' are shown.

El argumento utilizado en la llamada a un procedimiento puede ser una variable o un literal.

Veamos un ejemplo. A continuación, definimos un sencillo procedimiento cuya misión es mostrar el mayor de los dos números recibidos como parámetros:

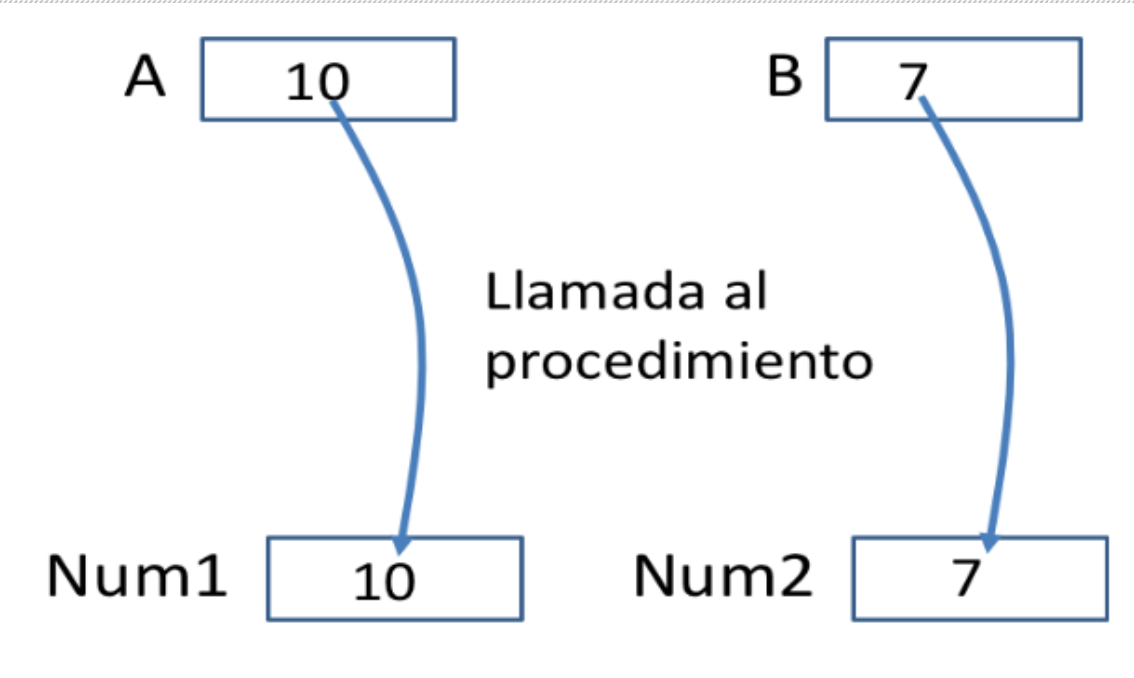
```
Sub MostrarMayor(num1 Integer, num2 Integer)
    If(num1>num2) Then
        Mostrar "El mayor es ", num1
    Else
        If((num2>num1) Then
            Mostrar "El mayor es ", num2
        Else
            Mostrar "Los números son iguales"
        End If
    End If
End Sub
```

Como vemos en el ejemplo anterior, la declaración de los parámetros de un procedimiento se realiza igual que la declaración de variables en un programa, solo que al declararse entre los paréntesis del procedimiento, **solo pueden utilizarse en el interior de éste**.

Si desde otra parte del programa se leen dos números y se quiere indicar cuál de los dos es el mayor, utilizando el procedimiento descrito anteriormente, el pseudocódigo sería simplemente:

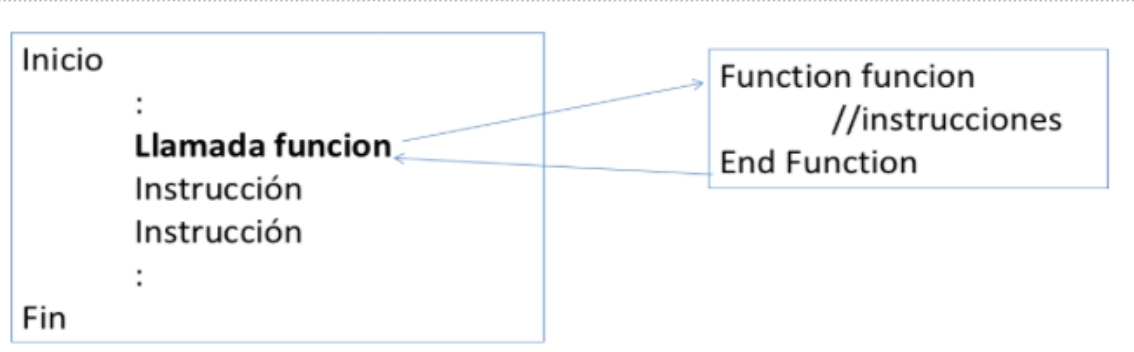
Leer A, B
MostrarMayor(A, B)

En este caso se pasan como argumentos en la llamada al procedimiento los valores contenidos en las variables A y B. Estos se volcarán en las variables parámetro num1 y num2, que son con las que trabajará el procedimiento.



Funciones

Al igual que un procedimiento, una función es un bloque de código que se ejecuta cuando es llamada y puede recibir también unos parámetros de entrada, pero a diferencia de aquel, devuelve un resultado al punto de llamada:



Para definir una función en pseudocódigo utilizaremos la palabra **Function** en vez de Sub:


```

Function nombre_funcion (parametro1, parametro2, ...)
    //instrucciones
    Return resultado
End Function

```

La instrucción *Return* se utiliza para devolver el resultado al punto de llamada, donde se recogerá y será procesado de alguna manera.
La instrucción de llamada seguirá el siguiente formato:

```
variable=nombre_funcion(argumento1, argumento2,...)
```

Cuando un programa ejecute esta instrucción, se realizará una llamada a la función indicada y se le pasarán los argumentos especificados entre paréntesis. El control del programa pasará entonces a la función, que tras ejecutar el *Return*, devolverá el control al punto de llamada y el resultado se volcará en la variable.

Como ejemplo, presentamos una función encargada de devolver el mayor de dos números recibidos:

```

Function ObtenerMayor(num1 Integer, num2 Integer)
    //declara la variable donde guardará el resultado
    Datos:
    resultado Integer
    Código:
    If(num1>num2) Then
        resultado=num1
    Else
        resultado=num2
    End If
    Return resultado
End Function

```

Como hemos visto en el listado anterior, una función o procedimiento también tendrá su zona de declaración de variables y zona de código.
Utilizando la función anterior, el siguiente bloque de código se encargaría de leer dos números y mostrar un mensaje indicando cual es el mayor:

```

Leer N1, N2
res=ObtenerMayor(N1, N2)
Mostrar "El mayor de los números es:", res

```


Se podría realizar la llamada a la función en la misma instrucción de salida sin necesidad de utilizar la variable *res*. Hay que tener en cuenta, que cuando una llamada a una función se realiza dentro de otra instrucción, se pasa el control a dicha función y cuando finaliza se completa la ejecución de la instrucción donde se encuentra la llamada a la función, utilizando el valor devuelto por esta:

Leer N1, N2

Mostrar "El mayor de los números es:", ObtenerMayor(N1, N2)

Actividad práctica 10. Realización de algoritmo basado en programación modular

Pregunta de refuerzo

Tenemos un procedimiento cuya cabecera está definida de la siguiente manera:
Sub Ejemplo (var1, var2).

Indica cual de las siguientes instrucciones de llamada al procedimiento es correcta:

- a) Ejemplo()
- b) Ejemplo(5, 9)
- c) Resultado=Ejemplo(4,2)
- d) Resultado=Ejemplo()

Solución: La respuesta correcta es la *b*. Para llamar a un procedimiento, se indica el nombre del mismo y entre paréntesis los argumentos que se le van a enviar. Dado que no devuelve resultado, no se puede realizar la llamada al mismo en una sentencia de asignación, por lo que *c* y *d* no son correctas. La *a* no es correcta porque el procedimiento declara dos parámetros, luego necesita que se le pasen dos argumentos en la llamada.

Programación por capas

Como ya hemos indicado, la programación modular permite dividir un programa grande en pequeñas partes más fáciles de desarrollar.

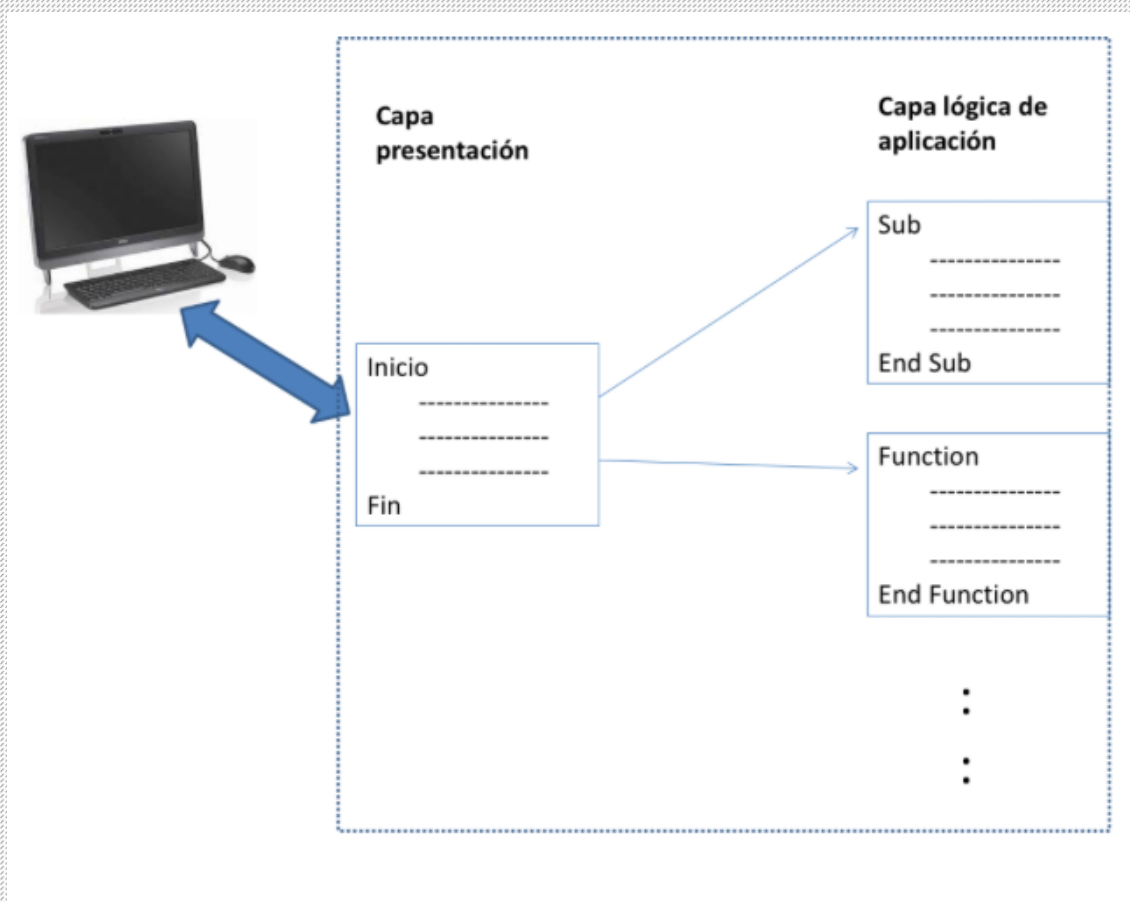
A la hora de aplicar esta metodología en el diseño de algoritmos mediante pseudocódigo, lo que haremos será definir en una serie de procedimientos y funciones los algoritmos encargados de realizar las distintas tareas de procesamiento de datos del programa.

En el módulo principal, que es el que delimitaremos entre las etiquetas *Inicio* y *Fin*, nos encargaremos de la lectura de datos del exterior y la presentación de

resultados. Desde éste llamaremos a los procedimientos y funciones creados para realizar el procesamiento de los datos.

Con esta forma de diseñar los programas, conseguimos separar el desarrollo del pseudocódigo (y también el del código final) en dos capas:

- **Capa de presentación.** Será la encargada de la interacción con el usuario, es decir, de solicitar datos al mismo y presentar los resultados obtenidos del procesamiento de dichos datos. El algoritmo correspondiente a esta capa se define dentro del módulo principal.
- **Capa de lógica de aplicación.** Realiza las operaciones con los datos y genera los resultados del procesamiento de los mismos. Se implementa a través de funciones y procedimientos, que son llamados desde el módulo principal. En esta capa no se codifican instrucciones de entrada y salida de datos; dichas operaciones son realizadas desde la capa de presentación.



Ejercicio resuelto

A continuación, basándonos en la aplicación de la programación modular, vamos a desarrollar un algoritmo para un programa encargado de solicitar al usuario la introducción de 10 notas de alumnos. Al finalizar la introducción de estas, se mostrará la nota media, nota inferior y superior y número de aprobados.

Para el desarrollo de este algoritmo aplicaremos la separación por capas definida en el apartado anterior, es decir, el programa principal se encargará de realizar la lectura de las notas y la presentación de los cálculos, que serán implementados por cuatro funciones independientes.

Como las notas deberán ser almacenadas en un array para su posterior procesamiento, este será el parámetro que las funciones deberán recibir en la llamada.

A continuación, te presentamos los cinco módulos que componen el desarrollo de este algoritmo:

//programa principal

Inicio

Datos:

Notas 1 To 10 Integer

N, I Integer

Código:

For I=1 To 10 Step 1

Leer N

Notas[I]=N

End For

//llamadas a las funciones y presentación de resultados

Mostrar "Nota media: ", RecuperarMedia(Notas)

Mostrar "Nota superior:", ObtenerMayor(Notas)

Mostrar "Nota inferior:", ObtenerMenor(Notas)

Mostrar "Aprobados:", ObtenerAprobados(Notas)

Fin

//Funciones

Function RecuperarMedia (Notas 1 To 10 entero)

Datos:

Media, I Integer

Código:

Media =0

For I=1 To 10 Step 1

Media = Media +Notas[I]

End For


```

    Media = Media /10
    Return Media
End Function
Function ObtenerMayor (Notas 1 To 10 entero)
    Datos:
    Mayor, I entero
    Código:
    Mayor =Notas[I] //se inicializa con la primera de las notas
    For I=1 To 10 Step 1
        If(Notas[I]> Mayor) Then
            Mayor =Notas[I]
        End If
    End For
    Return Mayor
End Function
Function ObtenerMenor (Notas 1 To 10 entero)
    Datos:
    Menor, I entero
    Código:
    Menor =Notas[I] //se inicializa con la primera de las notas
    For I=1 To 10 Step 1
        If(Notas[I]< Menor) Then
            Menor =Notas[I]
        End If
    End For
    Return Menor
End Function
Function ObtenerAprobados (Notas 1 To 10 entero)
    Datos:
    Aprobados, I entero
    Código:
    Aprobados=0
    For I=1 To 10 Step 1
        If(Notas[I]>=5) Then
            Aprobados=Aprobados+1
        End If
    End For
    Return Aprobados
End Function

```

Como comentario al ejercicio que te acabamos de presentar, fíjate que en las funciones desarrolladas hemos definido el parámetro array de notas con el nombre Notas, igual que el nombre que hemos utilizado en la declaración del

array de notas del módulo principal. Esto no tiene porque ser así, **argumento y parámetro no tienen porque llamarse igual**, lo importante es que el valor contenido en uno se volcará en el otro.

Actividad práctica 11. Realización de algoritmo para aplicación en capas

Programación Orientada a Objetos

La mayoría de los lenguajes de programación actuales son orientados a objetos. El hecho de que un lenguaje sea orientado a objetos no solo implica que trabaje con objetos, sino que además se pueden aplicar una serie de características propias de la orientación a objetos, encaminadas a reutilizar código en las aplicaciones, a que sean más fáciles de mantener y más fiables.

La programación orientada a objetos (abreviadamente POO), es pues la evolución natural de la programación estructurada. Y es que cuando las aplicaciones son muy grandes, la aplicación de las técnicas básicas de la programación estructurada resultan insuficientes para poder desarrollar programas fiables y fáciles de mantener.

Seguidamente, vamos a presentar los conceptos fundamentales en los que se basa este modelo de programación.

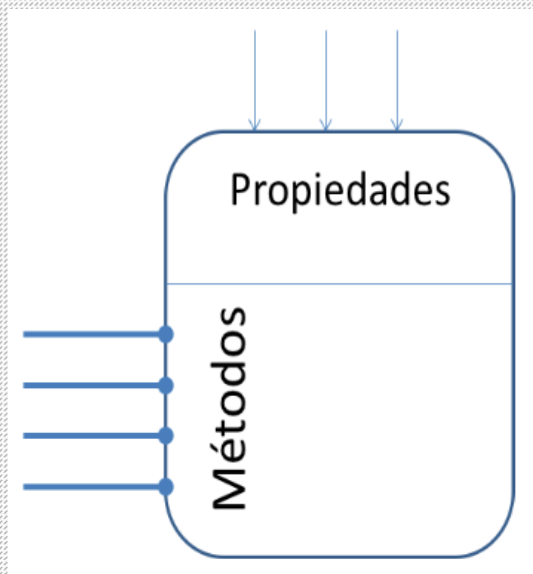
Clases y objetos

Un lenguaje orientado a objetos se basa en el uso de clases y objetos para implementar la funcionalidad de la aplicación, por eso estos conceptos serán los primeros que vamos a abordar en este capítulo.

El concepto de objeto es quizá el más importante de la programación orientada a objetos, por eso es el primero que vamos a presentarte.

Objetos

Un objeto podemos imaginarlo como una “caja negra” que expone al exterior una serie de funciones con las que poder realizar operaciones, así como una serie de parámetros que permiten configurar características del objeto. A las funciones y procedimientos que expone el objeto se les llama **métodos**, mientras que a las características se les conoce como **propiedades**.



Para entender mejor el concepto, pongamos un ejemplo del mundo real, que está lleno de objetos. Pensemos, por ejemplo, en un coche. Un coche es un tipo de objeto que expone una serie de *propiedades* como el color, la matrícula, el peso, etc., además de una serie de *métodos* para operar con el mismo, como arrancar, acelerar, frenar,...

Como este tenemos muchos otros ejemplos de objetos del mundo real. Un objeto televisor tiene una serie de propiedades como el número de pulgadas, si es o no 3D, etc., y también unos métodos como encender, apagar o cambiarCanal.

Llevado este concepto al mundo de la programación, el objeto es un elemento residente en memoria y al que accedemos desde el programa a través de una variable. A través de las propiedades podemos configurar características del objeto utilizando la siguiente sintaxis:

```
variable_objeto.propiedad=valor //asignación de valor a una propiedad  
o  
a= variable_objeto.propiedad //recuperación del valor de una propiedad
```

En ambos casos, para asignar un valor a una propiedad y para leer el valor de una propiedad, se utiliza la variable que contiene el objeto seguido del **operador punto** (".") y el nombre de la propiedad.

Dado que los métodos representan operaciones sobre el objeto, la llamada a estos ejecutará algún código internamente en el objeto. Para llamar a los métodos de un objeto utilizamos la sintaxis:

```
variable_objeto.metodo(argumento1, argumento2, ..)
```


Donde *argumento1*, *argumento2* son los valores que el método requiere para poder ejecutar la tarea.

Como vemos, la llamada a un método de un objeto es muy similar a las llamadas a procedimientos y funciones, y es que, como veremos a continuación, **un método se implementa mediante procedimientos y/o funciones**. Esto significa que puede haber métodos, aquellos que se creen mediante una función y devuelvan un resultado al punto de llamada, en cuyo caso se deberá recoger el valor devuelto por el mismo:

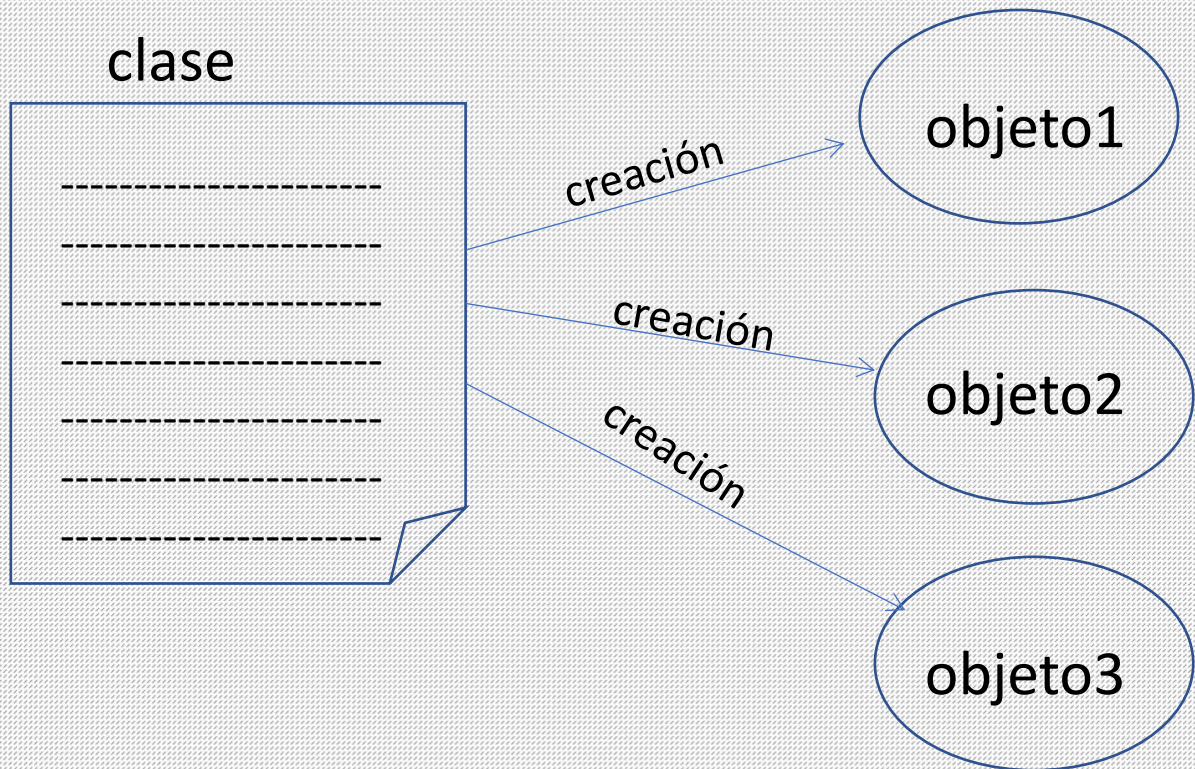
```
res = variable_objeto.metodo(argumento1, argumento2, ..)
```

Clases

Los objetos disponen de métodos, que son implementados mediante funciones y procedimientos, y de propiedades, que vienen a ser variables donde se almacenan las características de los objetos.

El lugar donde se define el código de estos métodos y propiedades es la **clase**. A través de estos métodos y propiedades, la clase establece el comportamiento de un determinado tipo de objeto.

A partir de la clase se crean los objetos y sobre ellos se aplican los métodos y propiedades definidos en la misma. La clase es, por tanto, el molde a partir del cual se pueden construir los objetos, que son los elementos que utilizaremos en los programas para poder hacer uso de los métodos y propiedades.



Cada objeto podrá tener sus propios valores de propiedades, mientras que el comportamiento de los métodos será el mismo en cada objeto.

Para definir una clase mediante pseudocódigo, utilizaremos la siguiente sintaxis:

```
Class NombreClase
    //definición del contenido de la clase
End Class
```

El contenido de la clase está formado por propiedades, que son simples variables declaradas con la palabra *property* delante, y métodos que son definidos mediante Sub y Function.

Veamos un sencillo ejemplo de creación de una clase Calculadora con la que podemos realizar operaciones básicas con dos números, como Sumar, Restar, Multiplicar y Dividir. Además de cuatro métodos para realizar las citadas operaciones, dispondrá de dos propiedades para almacenar los operandos con los que van a trabajar los métodos de la clase.

Esta sería la implementación en pseudocódigo de la clase Calculadora:

```
Class Calculadora
    Property operando1 Integer
    Property operando2 Integer
```



```
Function Sumar()  
    Return operando1+operando2  
End Function  
Function Restar()  
    Return operando1-operando2  
End Function  
Function Multiplicar()  
    Return operando1*operando2  
End Function  
Function Dividir()  
    Return operando1/operando2  
End Function  
End Class
```

Según se puede apreciar, los cuatro métodos se definen mediante el mismo tipo de funciones que estudiamos durante la programación modular, estas funciones operan con los datos almacenados en las propiedades, que no son más que variables a las que se les añade la palabra **property**.

Si se quiere definir en la clase una variable que no queremos exponer como propiedad, simplemente que sirva para almacenar algún valor compartido por los métodos, se definirá como una variable normal, sin utilizar la palabra **property** delante.

Creación de objetos

La clase define los métodos y propiedades, pero para poder hacer uso de los mismos se necesita un objeto sobre el que poder aplicarlos.

Para crear un objeto de una clase, utilizaremos un operador llamado **New**, que es el que se emplea en la mayoría de los lenguajes de programación. La manera de utilizarlo sería la siguiente:

```
variable_objeto=New NombreClase()
```

Como vemos, a continuación del operador se indica el nombre de la clase de la que queremos crear el objeto, seguido de los paréntesis. Esta instrucción creará un objeto en memoria y lo guardará en la variable indicada a la izquierda del signo igual. Esta variable tendrá que ser declarada con el tipo de la clase. A partir de ahí, ya se puede utilizar la variable objeto para acceder a propiedades y métodos.

Para ilustrarlo con un ejemplo, vamos a hacer un programa principal que se encargue de leer dos números y, utilizando un objeto de la clase Calculadora

definida anteriormente, muestre los resultados de las cuatro operaciones básicas sobre los números:

Inicio

Datos:

Oper Calculadora //define una variable del tipo de la clase,
//en ella se guardará el objeto

N1, N2 Integer

Sum, Res, Multi, Div Integer

Código:

Leer N1, N2

Oper=New Calculadora() //crea el objeto

//asigna valores leídos a las propiedades

Oper.operando1=N1

Oper.operando2=N2

//llama a los métodos para realizar los cálculos

Sum=Oper.Sumar()

Res=Oper.Restar()

Multi=Oper.Multiplicar()

Div=Oper.Dividir()

Mostrar "Suma:", Sum

Mostrar "Resta:", Res

Mostrar "Multiplicación:", Multi

Mostrar "División:", Div

Fin

En el ejemplo anterior podríamos haber prescindido de las variables Sum, Res, Multi y Div:

Mostrar "Suma:", Oper.Sumar()

Mostrar "Resta:", Oper.Restar()

Mostrar "Multiplicación:", Oper.Multiplicar()

Mostrar "División:", Oper.Dividir()

Ejercicio de refuerzo

Tenemos una clase llamada "Mesa" con una propiedad con una propiedad llamada "ancho". Si queremos aplicar el ancho 100 a un objeto de esta clase tendríamos que hacer:

a) M Mesa

M=New Mesa()

M.ancho=100

b) Mesa.ancho=100

c) Mesa = New Mesa()

Mesa.ancho=100

d) Mesa = New Mesa()

Mesa.ancho(100)

Solución: La respuesta correcta es la *a*. Se debe declarar una variable del tipo de objeto (es decir, del tipo de la clase), crear un objeto de la misma y guardarlo en la variable, para después poder llamar a la propiedad. La respuesta *b* es incorrecta porque no se puede aplicar la propiedad sobre la clase, solo sobre un objeto de la misma. La *c* es incorrecta porque la creación del objeto Mesa es incorrecta. La *d* es incorrecta por el mismo motivo que *c* y porque, además *ancho* es una propiedad, no un método.

Constructores

Hemos visto en el ejemplo anterior que para poder utilizar los métodos que realizan las operaciones es necesario primero establecer ciertas propiedades del objeto.

En muchos casos estas propiedades se establecen nada más crear el objeto, por lo que sería muy cómodo poder hacerlo en la misma instrucción de creación del objeto, en vez de tener que asignar explícitamente cada propiedad. Esto es posible hacer utilizando **constructores**.

Un constructor es un bloque de código que se define dentro de una clase y que se ejecuta en el momento en que se crea un objeto de dicha clase. Por tanto, es el lugar ideal para inicializar propiedades de los objetos.

Los constructores se definen de la siguiente manera:

```
NombreClase(parametro1, parametro2)
```

```
End
```

Como vemos, son parecidos a los procedimientos, pero sin la palabra Sub.

Además, el nombre del constructor debe ser igual al de la clase.

Por ejemplo, si queremos definir la clase Calculadora utilizando constructor, quedaría de la siguiente manera:

```
Class Calculadora
    Property operando1 Integer
    Property operando2 Integer
    //constructor
    Calculadora(op1 Integer, op2 Integer)
        operando1=op1
        operando2=op2
    End
    Function Sumar()
        Return operando1+operando2
    End Function
    Function Restar()
        Return operando1-operando2
    End Function
    Function Multiplicar()
        Return operando1*operando2
    End Function
    Function Dividir()
```



```
        Return operando1/operando2
    End Function
End Class
```

Por su parte, la instrucción de creación del objeto quedaría de la siguiente manera:

```
Oper=New Calculadora(N1, N2)
```

Y ya no sería necesario asignar explícitamente los valores N1 y N2 a las propiedades Operando1 y Operando2.

Una clase puede disponer de más de un constructor, lo que ofrecería la posibilidad de inicializar objetos de distinta forma. Todos los constructores tendrán el mismo nombre, que es el de la clase, pero deberán diferenciarse en el número de parámetros o en el tipo de los mismos.

Por ejemplo, podríamos incluir en la clase Calculadora dos constructores, uno para inicializar cada propiedad con un valor y otro que inicializaría las dos propiedades del objeto con el mismo valor:

```
Calculadora(op1 Integer, op2 Integer)
    operando1=op1
    operando2=op2
End
Calculadora(op Integer)
    operando1=op
    operando2=op
End
```

Al hecho de poder definir más de un constructor en una clase se le conoce como **sobrecarga de constructores**.

Ejercicio ejemplo

Vamos a realizar un ejercicio en el que vamos a poner en práctica la programación modular con separación en capas, utilizando una clase para la implementación de la lógica de aplicación.

El ejercicio consiste en diseñar un algoritmo para un programa que simula (de forma rudimentaria) el funcionamiento de un cajero. Al iniciarse el programa se solicitará al usuario el saldo inicial de la cuenta, después aparecerá el siguiente menú:

- 1.- Ingresar
- 2.- Extraer
- 3.- Ver saldo
- 4.- Salir

Si se seleccionan las opciones 1 y 2, se le solicitará al usuario la cantidad a ingresar/extraer, se realizará la operación y volverá de nuevo a aparecer el menú. La opción 3 mostrará el saldo de la cuenta, y a continuación, aparecerá de nuevo el menú. Cuando se elija la opción 4 el programa finalizará.

Para desarrollar el algoritmo, como indicamos antes, vamos a separar el diseño en capas. La capa de lógica de aplicación será implementada con una clase llamada Cuenta, que tendrá como propiedad el saldo y dispondrá de los métodos necesarios para operar con la cuenta. Este sería el código de la clase:

Class Cuenta

```

property saldo Decimal
//mediante el constructor se da la posibilidad de inicializar el saldo
Cuenta(s decimal)
    saldo=s
End
Sub Ingresar(cantidad Decimal)
    saldo=saldo+cantidad
End Sub
Sub Extraer(cantidad Decimal)
    saldo=saldo-cantidad
End Sub

```

End Class

En cuanto al algoritmo del programa principal, que es donde se define la capa de presentación, quedaría de la siguiente manera:

Inicio

```

Datos:
Ct Cuenta //variable donde guardaremos un objeto Cuenta
Op Integer
Cant Integer
Inicial Decimal
Código:
Mostrar "Introduce saldo inicial"
Leer Inicial
Ct=New Cuenta(Inicial) //crea el objeto cuenta
Do

```



```
Mostrar "1.- Ingresar"
Mostrar "2.- Extraer"
Mostrar "3.- Ver saldo"
Mostrar "4.- Salir"
Leer Op
Switch(Op)
    Case 1:
        Mostrar "Introduce cantidad"
        Leer Cant
        Ct.Ingresar(Cant)
    Case 2:
        Mostrar "Introduce cantidad"
        Leer Cant
        Ct.Extraer(Cant)
    Case 3:
        //muestra el valor de la propiedad saldo
        Mostrar "El saldo es ", Ct.saldo
End Switch

While(Op<>4)
```

Fin

Pregunta de refuerzo

Una clase llamada "Ejemplo" tiene un constructor con el siguiente formato:
Ejemplo(d Integer, a String)

Indica cual de las siguientes instrucciones de creación de objetos Ejemplo es correcta:

- a) New Ejemplo(10, 200)
- b) New Ejemplo()
- c) New Ejemplo(200)
- d) New Ejemplo(1, "set")

Solución: La respuesta correcta es la *d*. En esta instrucción es la única en la que se pasan los argumentos coincidentes con los parámetros definidos en el constructor. En las respuestas *a*, *b* y *c* el paso de argumentos es incorrecto.

Actividad práctica 12. Realización de aplicación basada en el uso de clases

Otras características de la POO

Como hemos indicado, los lenguajes orientados a objetos ofrecen una serie de características de gran valor durante el desarrollo de las aplicaciones. Aunque tendrás la oportunidad de profundizar en el uso de dichas características durante el estudio del lenguaje de programación específico de tu itinerario, te presentamos a continuación los fundamentos de las más interesantes.

Sobrecarga de métodos

Ya hemos comentado anteriormente el concepto de sobrecarga aplicado a constructores. Pues bien, este concepto es aplicable también a los métodos en general, es decir, una clase puede sobrecargar no solo constructores, sino también métodos.

La sobrecarga de métodos consiste en definir en una misma clase varios métodos con el mismo nombre, si bien deben diferenciarse en el número o tipo de parámetros.

Por ejemplo, en la clase Calculadora que creamos anteriormente podríamos tener los siguiente métodos suma definidos:

```
Function Sumar()  
Function Sumar(a Integer)  
Function Sumar (a Integer, b Integer)
```

Lo que **no** podríamos es tener por ejemplo los siguientes métodos definidos en la misma clase:

```
Function Restar(m Integer)  
Function Restar(k Integer)
```

Pues, aunque les hayamos dado distintos nombres, los tipos de los parámetros coinciden, luego están incumpliendo la regla de la sobrecarga.

La sobrecarga de métodos hace más cómodo el uso de una clase cuando esta dispone de varios métodos que realizan la misma tarea. Y es que, si vamos a tener tres métodos que van a realizar la operación Sumar, pero con diferentes parámetros, es preferible que se llamen igual, dado que van a hacer la misma función, que darles nombres diferentes.

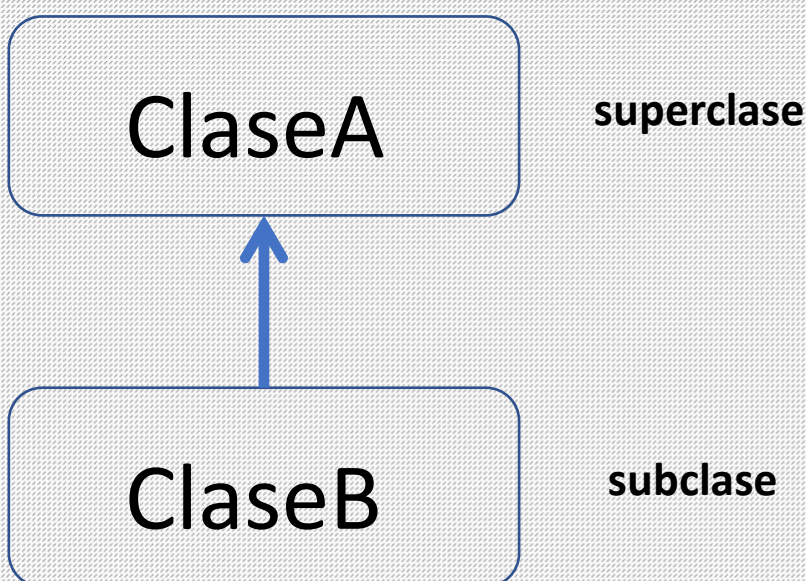
Herencia

Después del concepto de clase y objeto, la herencia es la característica más importante de los lenguajes orientados a objetos.

La herencia permite crear clases que adquieran (hereden) todos los métodos y propiedades de otras clases ya existentes. Esta característica la aplicaremos cuando queramos crear unas clases, que debe contener la misma funcionalidad que otra clases ya existente, pero que desea incorporar características y métodos adicionales.

En vez de codificar de nuevo los métodos y propiedades en la nueva clase, la herencia nos permite reutilizar el código de la clase ya existente.

La herencia se representa de forma gráfica de la siguiente manera:



A la clase padre o clase heredada se le con el nombre de **superclase** o clase base, mientras que la que hereda es conocida como **subclase** o clase derivada.

Herencia en pseudocódigo

De cara a crear una clase que herede otra clase, cada lenguaje de programación orientado a objetos dispone de su propia notación. En el caso de pseudocódigo, vamos a emplear la palabra `extends` (es la que utilizan lenguajes como Java) para indicar que una clase va a heredar otra clase:

```
Class NuevaClase extends ClasePadre  
    //código de la nueva clase  
End Class
```

Como ejemplo de aplicación, vamos a crear una clase que herede a la clase Cuenta del ejercicio anterior, a la que añadiremos una nueva propiedad llamada "clave" que represente una clave asociada a la cuenta. Por otro lado, crearemos

un nuevo método llamado "resetear" que pondrá el saldo de la cuenta a 0 cuando sea llamado:

```
Class CuentaClave extends Cuenta
    Property clave Integer
    CuentaClave(c Integer)
        clave=c
    End
    Sub Resetear()
        saldo=0 //acceso a propiedad heredada
    End Sub
End Class
```

Como vemos, la nueva clase solamente define los métodos y propiedades nuevos que incorpora, todo lo definido en la clase Cuenta también pertenece a CuentaClave, de ahí que podamos acceder directamente a la propiedad saldo. Si creásemos un objeto de CuentaClave podríamos acceder directamente a todos sus métodos:

```
Cc CuentaClave
Cc=New CuentaClave(100) //le pasa la clave de la cuenta al constructor
//llamada a métodos
Cc.Ingresar(20)
Cc.Extrear(10)
Cc.Resetear()
```

Sin embargo, la clase CuentaClave no está completa. Al crear el objeto no se establece ningún valor para el saldo, por lo que originariamente estará a 0. Lo habitual es que en el constructor de una clase se inicialice tanto las propiedades propias como las heredadas, aprovechando el constructor de la superclase para esto último.

A continuación, te presentamos la versión completa de la clase:

```
Class CuentaClave extends Cuenta
    Property clave Integer
    CuentaClave(c Integer, s Decimal):base(s)
        clave=c
    End
    Sub Resetear()
        saldo=0 //acceso a propiedad heredada
    End Sub
End Class
```


Si observas la definición del constructor verás que a continuación del paréntesis de cierre aparece la expresión `:base(s)`

Lo que hacemos con esta expresión es una llamada al constructor de la clase `base`, al que le pasamos uno de los parámetros recibidos en el constructor de `CuentaClave`, concretamente el saldo. Es realmente una llamada a `Cuenta(s Decimal)`.

Así pues, al crear un objeto `CuentaClave`, antes de ejecutarse el constructor de dicha clase, la expresión `:base()` hace que se produzca primero una llamada al constructor de la superclase, concretamente a aquel que coincida con los argumentos indicados en `base()`. Una vez que este constructor se ha ejecutado, el algoritmo continúa en el constructor de la subclase (`CuentaClave`).

Sobrescritura de métodos

Hay ocasiones en las cuales alguno de los métodos que hereda una clase está implementado de manera que no se adapta a la funcionalidad que se le quiere dar a la nueva clase, o simplemente se desea realizar una mejora en el mismo. En estos casos lo que se puede hacer es sobrescribir el método heredado en la nueva clase.

La sobrescritura de un método consiste en volver a definir en una subclase un método que ha sido heredado de la superclase. Durante la sobrescritura, se debe respetar el formato (nombre y parámetros) del método original.

Por ejemplo, supongamos que en la clase `CuentaClave` que hereda `Cuenta` quisiéramos tener una versión mejorada del método `extraer()`, de forma que solo se pudiera extraer dinero en caso de que hubiera saldo suficiente. Lo que tendríamos que hacer en este caso es volver a definir el método `extraer()`, es decir, sobrescribirlo, en la clase `CuentaClave`. El código actualizado de la clase quedaría:

```
Class CuentaClave extends Cuenta
    Property clave Integer
    CuentaClave(c Integer, s Decimal):base(s)
        clave=c
    End
    Sub Extraer(cantidad Decimal)
        If(cantidad<saldo) Then
            saldo=saldo-cantidad
        End If
    End Sub
    Sub Resetear()
```



```
        saldo=0 //acceso a propiedad heredada
    End Sub
End Class
```

Como vemos, a la hora de sobrescribir un método en pseudocódigo no utilizaremos ninguna palabra especial, simplemente se definirá el método con el mismo nombre y parámetros que el original.

Si quisiéramos crear un objeto CuentaClave y llamar al método extraer:

```
Cc CuentaClave
Cc=New CuentaClave(111, 200)
Cc.Extraer(300)
Mostrar "Saldo: ", Cc.saldo
```

Tras ejecutar el código anterior se mostrará el mensaje **Saldo 200**.

Y es que, dado que la cantidad que se quiere extraer (300) es superior al saldo (200), al llamar a la versión sobrescrita del método extraer(), la operación de descuento de la cantidad al saldo no se producirá.

Actividad práctica 13. Realización de pseudocódigos basados en el uso de clases con herencia

Pregunta de refuerzo

Una clase llamada Clase1 define un método con el siguiente formato: Imprimir (texto String). Dicha clase es heredada por Clase2, donde se define un método con el formato Imprimir(texto String, cod Integer). Lo que se ha realizado en Clase2 con Imprimir es:

- a) Una sobrescritura de método
- b) Una sobrecarga de método
- c) Una nueva definición de método, sin sobrecarga ni sobrescritura
- d) Un error de programación

La respuesta correcta es la *b*. Se ha creado un nuevo método con el mismo nombre que otro ya existente (el método Imprimir de Clase1 es heredado por Clase2) con distinto número de parámetros, luego es un caso de sobrecarga. La *a* no es correcta porque el nuevo método no coincide en número y tipo de parámetros con el heredado. Al ser *b* correcta, *c* y *d* no pueden serlo.