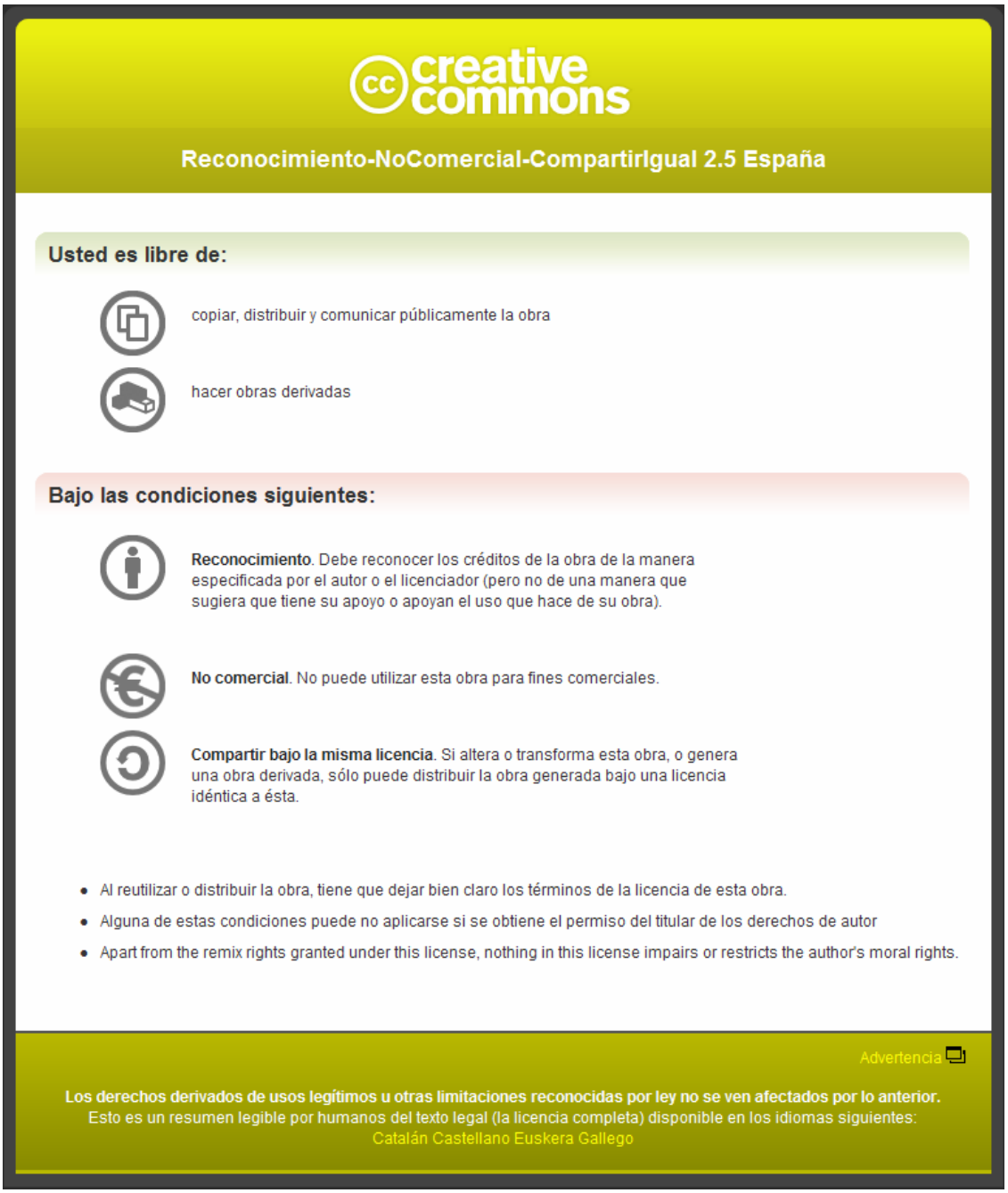
Apuntes de Fundamentos de Programación

Curso 2006/2007

1er Curso de Administración de Sistemas Informáticos

Autor: Jorge Sánchez Asenjo (año 2007) http://www.jorgesanchez.net
e-mail:info@jorgesanchez.net



Para ver una copia completa de la licencia, acudir a la dirección
http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es

(Unidad 1) Algoritmos y Programas

(1.1) computadora y sistema operativo

(1.1.1) computadora

Según la **RAE** (Real Academia de la lengua española), una computadora es una ***máquina electrónica, analógica o digital, dotada de una memoria de gran capacidad y de métodos de tratamiento de la información, capaz de resolver problemas matemáticos y lógicos mediante la utilización automática de programas informáticos***.

Sin duda esta máquina es la responsable de toda una revolución que está cambiando el panorama económico, social e incluso cultural. Debido a la importancia y al difícil manejo de estas máquinas, aparece la **informática** como la ciencia orientada al proceso de información mediante el uso de computadoras.

Una computadora consta de diversos componentes entre los que sobresale el procesador, el componente que es capaz de realizar las tareas que se requieren al ordenador o computadora. En realidad un procesador sólo es capaz de realizar tareas sencillas como:

- Operaciones aritméticas simples: suma, resta, multiplicación y división
- Operaciones de comparación entre valores
- Almacenamiento de datos

Algunos de los componentes destacables de un ordenador son:

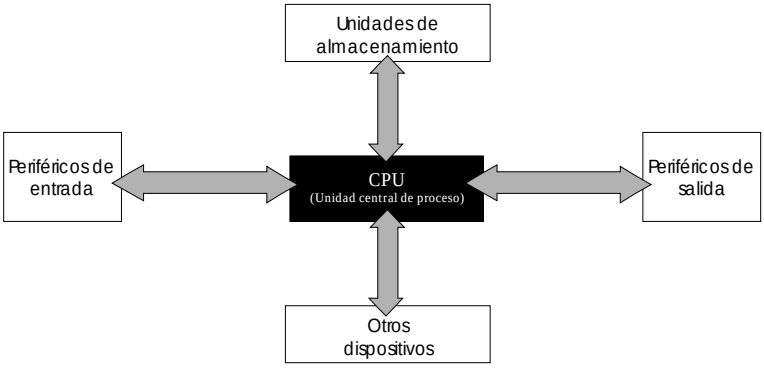


Ilustración 1, componentes de un ordenador desde un punto de vista lógico

Este desglose de los componentes del ordenador es el que interesa a los programadores. Pero desde un punto de vista más físico, hay otros componentes a señalar:

- **Procesador.** Núcleo digital en el que reside la CPU del ordenador. Es la parte fundamental del ordenador, la encargada de realizar todas las tareas.
- **Placa base.** Circuito interno al que se conectan todos los componentes del ordenador, incluido el procesador.
- **Memoria RAM.** Memoria interna formada por un circuito digital que está conectado mediante tarjetas a la placa base. Su contenido se evapora cuando se desconecta al ordenador. Lo que se almacena no es permanente.
- **Memoria caché.** Memoria ultrarrápida de características similares a la RAM, pero de velocidad mucho más elevada por lo que se utiliza para almacenar los últimos datos utilizados.
- **Periféricos.** Aparatos conectados al ordenador mediante tarjetas o ranuras de expansión (también llamados puertos). Los hay de **entrada** (introducen datos en el ordenador: teclado, ratón, escáner,...), de **salida** (muestran datos desde el ordenador: pantalla, impresora, altavoces,...) e incluso de **entrada/salida** (módem, tarjeta de red).
- **Unidades de almacenamiento.** En realidad son periféricos, pero que sirven para almacenar de forma permanente los datos que se deseen del ordenador. Los principales son el **disco duro** (unidad de gran tamaño interna al ordenador), la **disquetera** (unidad de baja capacidad y muy lenta, ya en desuso), el **CD-ROM** y el **DVD.**

(1.1.2) hardware y software

hardware

Se trata de todos los componentes físicos que forman parte de un ordenador: procesador, RAM, impresora, teclado, ratón,...

software

Se trata de la parte conceptual del ordenador. Es decir los datos y aplicaciones que maneja y que permiten un grado de abstracción mayor. Cualquier cosa que se pueda almacenar en una unidad de almacenamiento es software (la propia unidad sería hardware).

(1.1.3) Sistema Operativo

Se trata del software (programa) encargado de gestionar el ordenador. Es la aplicación que oculta la física real del ordenador para mostrarnos un interfaz que permita al usuario un mejor y más fácil manejo de la computadora.

funciones del Sistema Operativo

Las principales funciones que desempeña un Sistema Operativo son:

- Permitir al usuario comunicarse con el ordenador. A través de comandos o a través de una interfaz gráfica.
- Coordinar y manipular el hardware de la computadora: memoria, impresoras, unidades de disco, el teclado,...
- Proporcionar herramientas para organizar los datos de manera lógica (carpetas, archivos,...)
- Proporcionar herramientas para organizar las aplicaciones instaladas.
- Gestionar el acceso a redes
- Gestionar los errores de hardware y la pérdida de datos.
- Servir de base para la creación de aplicaciones, proporcionando funciones que faciliten la tarea a los programadores.
- Administrar la configuración de los usuarios.
- Proporcionar herramientas para controlar la seguridad del sistema.

algunos sistemas operativos

- **Windows.** A día de hoy el Sistema Operativo más popular (instalado en el 95% de computadoras del mundo). Es un software propiedad de Microsoft por el que hay que pagar por cada licencia de uso.

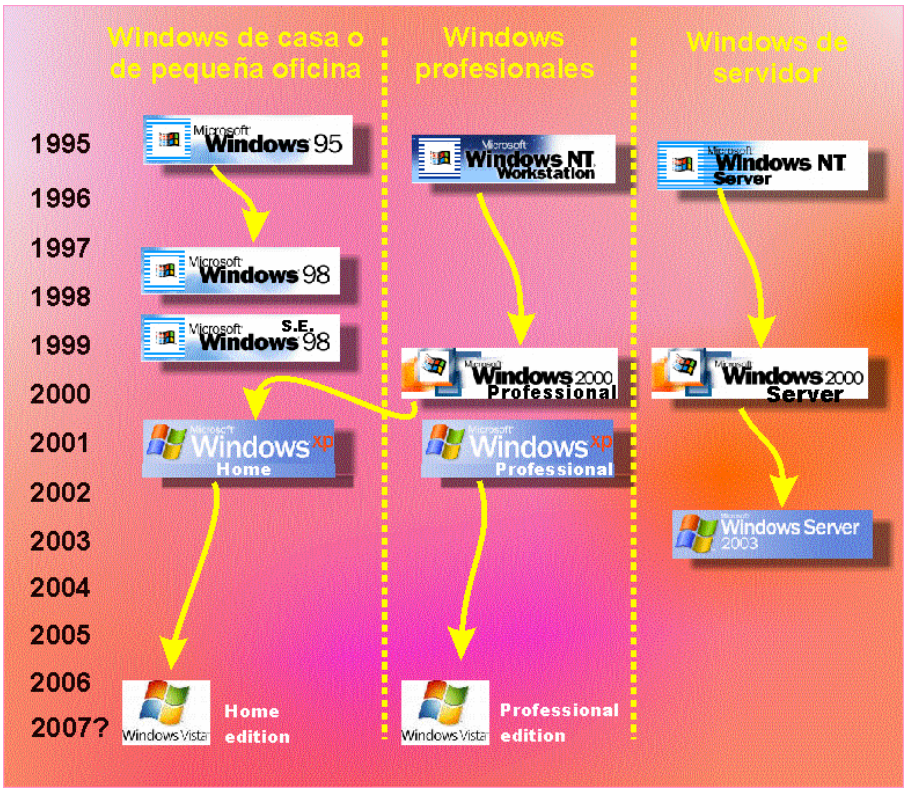


Ilustración 2, Versiones actuales de Windows

- **Linux.** Sistema operativo de código abierto. Posee numerosas distribuciones (muchas de ellas gratuitas) y software adaptado para él (aunque sólo el 15% de ordenadores tiene instalado algún sistema Linux). Fundamentalmente su éxito está en grandes máquinas o servidores. Actualmente las distribuciones Linux más conocidas son: **Red Hat**, **Fedora** (versión gratuita de Red Hat), **Debian**, **Ubuntu** (variante de Debian de libre distribución), **Mandriva** y **SUSE**.
- **MacOs.** Sistema operativo de los ordenadores **MacIntosh**.
- **Unix.** Sistema operativo muy robusto para gestionar redes de todos los tamaños. Actualmente en desuso debido al uso de Linux (que está basado en Unix), aunque sigue siendo muy utilizado para gestionar grandes redes (el soporte sigue siendo una de las razones para que se siga utilizando)
- **Solaris.** Versión de Unix para sistemas **Sun**.

(1.2) codificación de la información

(1.2.1) introducción

Sin duda una de las informaciones que más a menudo un ordenador tiene que manipular son los números. Pero también el ordenador necesita codificar otro tipo de información, como por ejemplo caracteres, imágenes, sonidos,...

EL problema es que para el ordenador toda la información debe estar en formato binario (unos y ceros). Por ello se necesita traducir todos los datos a ese formato.

(1.2.2) sistemas numéricos

En general, a lo largo de la historia han existido numerosos sistemas de numeración. Cada cultura o civilización se ha servido en la antigüedad de los sistemas que ha considerado más pertinentes. Para simplificar, dividiremos a todos los sistemas en dos tipos:

- **Sistemas no posicionales**. En ellos se utilizan símbolos cuyo valor numérico es siempre el mismo independientemente de donde se sitúen. Es lo que ocurre con la numeración romana. En esta numeración el símbolo **I** significa siempre ***uno*** independientemente de su posición.
- **Sistemas posicionales**. En ellos los símbolos numéricos cambian de valor en función de la posición que ocupen. Es el caso de nuestra numeración, el símbolo ***2***, en la cifra ***12*** vale ***2***; mientras que en la cifra ***21*** vale veinte.

La historia ha demostrado que los sistemas posicionales son mucho mejores para los cálculos matemáticos por lo que han retirado a los no posicionales. La razón: las operaciones matemáticas son más sencillas utilizando sistemas posicionales.

Todos los sistemas posicionales tienen una **base**, que es el número total de símbolos que utiliza el sistema. En el caso de la numeración decimal la base es 10; en el sistema binario es 2.

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

```
...+ X3·B3 + X2·B2 + X1·B1 + X0·B0 + X-1·B-1 + X-2·B-2+...
```

Donde:

- **Xi** Es el símbolo que se encuentra en la posición número i del número que se está convirtiendo. Teniendo en cuenta que la posición de las unidades es la posición 0 (la posición -1 sería la del primer decimal)
- **B** Es la base del sistemas que se utiliza para representar al número

Por ejemplo si tenemos el número ***153,6*** utilizando e sistema octal (base ocho), el paso a decimal se haría:

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

(1.2.3) sistema binario

introducción

Los números binarios son los que utilizan las computadoras para almacenar información. Debido a ello hay términos informáticos que se refieren al sistema binario y que se utilizan continuamente. Son:

- **BIT (de *Binary diGIT*)**. Se trata de un dígito binario, el número binario 1001 tiene cuatro BITS.
- **Byte**. Es el conjunto de 8 BITS.
- **Kilobyte**. Son 1024 bytes.
- **Megabyte**. Son 1024 Kilobytes.
- **Gigabyte**. Son 1024 Megabytes.
- **Terabyte**. Son 1024 Gigabytes.
- **Petabyte**. Son 1024 Terabytes.

conversión binario a decimal

Utilizando el teorema fundamental de la numeración, por ejemplo para el número binario 10011011011 el paso sería (los ceros se han ignorado):

$$1 \cdot 2^{10} + 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1243$$

conversión decimal a binario

El método más utilizado es ir haciendo divisiones sucesivas entre dos. Los restos son las cifras binarias. Por ejemplo para pasar el 39:

```
39:2 = 19 resto 1
19:2 = 9 resto 1
9:2 = 4 resto 1
4:2 = 2 resto 0
2:2 = 1 resto 0
1:2 = 0 resto 1
```

Ahora las cifras binarias se toman al revés. Con lo cual, el número 100111 es el equivalente en binario de 39.

operaciones aritméticas binarias

suma

Se efectúa igual que las sumas decimales, sólo que cuando se suma un uno y otro uno, ese dice que tenemos un acarreo de uno y se suma a la siguiente cifra. Ejemplo (suma de 31, en binario 10011, y 28, en binario, 11100)

Acarreo	**1**	**1**			
	1	1	1	1	1
	1	1	1	0	0
1	**1**	**1**	**0**	**1**	**1**

El resultado es 111011, 59 en decimal.

resta

El concepto es parecido sólo que en el caso de la resta es importante tener en cuenta el signo. No se explica en el presente manual ya que se pretende sólo una introducción a los números binarios. En la actualidad la resta se hace sumando números en **complemento a 2**[1].

operaciones lógicas

Se trata de operaciones que manipulan BITS de forma lógica, son muy utilizadas en la informática. Se basan en una interpretación muy utilizada con los números binarios en la cual el dígito **1** se interpreta como **verdadero** y el dígito **0** se interpreta como falso.

operación AND

La operación AND (en español ***Y***), sirve para unir expresiones lógicas, se entiende que el resultado de la operación es verdadero si alguna de las dos expresiones es verdadero (por ejemplo la expresión ***ahora llueve y hace sol*** sólo es verdadera si ocurren ambas cosas).

En el caso de los dígitos binarios, la operación AND opera con dos BITS de modo que el resultado será uno si ambos bits valen uno.

	0	**1**
0	0	0
1	0	1

La tabla superior se llama **tabla de la verdad** y sirve para mostrar resultados de operaciones lógicas, el resultado está en la parte blanca, en la otra parte se representan los operadores . El resultado será 1 si ambos operadores valen 1

[1] Se trata de una forma avanzada de codificar números que utiliza el primer BIT como signo y utiliza el resto de forma normal para los números positivos y cambiando los unos por los ceros para los números negativos.

operación OR

OR (***O*** en español) devuelve verdadero si cualquiera de los operandos es verdadero (es decir, si valen 1). La tabla es esta:

	0	1
0	0	1
1	1	1

operación NOT

Esta operación actúa sobre un solo BIT y lo que hace es invertirle; es decir, si vale uno valdrá cero, y si vale cero valdrá uno.

	0	1
	1	0

codificación de otros tipos de datos a binario

texto

Puesto que una computadora no sólo maneja números, habrá dígitos binarios que contengan información que no es traducible a decimal. Todo depende de cómo se interprete esa traducción.

Por ejemplo en el caso del texto, lo que se hace es codificar cada carácter en una serie de números binarios. El código **ASCII** ha sido durante mucho tiempo el más utilizado. Inicialmente era un código que utilizaba 7 bits para representar texto, lo que significaba que era capaz de codificar 127 caracteres. Por ejemplo el número 65 (1000001 en binario) se utiliza para la A mayúscula.

Poco después apareció un problema: este código es suficiente para los caracteres del inglés, pero no para otras lenguas. Entonces se añadió el octavo bit para representar otros 128 caracteres que son distintos según idiomas (Europa Occidental usa unos códigos que no utiliza Europa Oriental).

Eso provoca que un código como el 190 signifique cosas diferentes si cambiamos de país. Por ello cuando un ordenador necesita mostrar texto, tiene que saber qué juego de códigos debe de utilizar (lo cual supone un tremendo problema).

Una ampliación de este método de codificación es el código **Unicode** que puede utilizar hasta 4 bytes (32 bits) con lo que es capaz de codificar cualquier carácter en cualquier lengua del planeta utilizando el mismo conjunto de códigos. Poco a poco es el código que se va extendiendo; pero la preponderancia histórica que ha tenido el código ASCII, complica su popularidad.

otros datos

En el caso de datos más complejos (imágenes, vídeo, audio) se necesita una codificación más compleja. Además en estos datos no hay estándares, por lo que hay decenas de formas de codificar.

En el caso, por ejemplo, de las imágenes, una forma básica de codificarlas en binario es la que graba cada **píxel** (cada punto distinguible en la imagen) mediante tres bytes: el primero graba el nivel de rojo, el segundo el nivel de azul y el tercero el nivel de verde. Y así por cada píxel.

(1.2.4) sistema hexadecimal

Es un sistema que se utiliza mucho para representar números binarios. Un problema (entre otros) de los números binarios es que ocupan mucho espacio para representar información. El sistema hexadecimal es la forma de representar números en base 16. de modo que en los dígitos del 0 al 9 se utilizan los mismos símbolos que en el sistema decimal y a partir del 10 se utiliza la letra A y así hasta la letra F que simboliza el 15.

Así el número hexadecimal CA3 sería:

$$C \cdot 16^2 + A \cdot 16^1 + 3 \cdot 16^0 = 12 \cdot 256 + 10 \cdot 16 + 3 = 3235$$

Como se observa pasar de hexadecimal a decimal es complejo. La razón del uso de este sistema es porque tiene una equivalencia directa con el sistema binario. De hecho en una cifra hexadecimal caben exactamente 4 bits. Por ello la traducción de hexadecimal a binario se basa en esta tabla:

Hexadecimal	Binario
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Así el número hexadecimal C3D4 sería el binario 1100 0011 1101 0100. Y el binario 0111 1011 1100 0011 sería el hexadecimal 7BC3

(1.3) algoritmos

(1.3.1) noción de algoritmo

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

Los algoritmos, como indica su definición oficial, son una serie de pasos que permiten obtener la solución a un problema. La palabra algoritmo procede del matemático Árabe **Mohamed Ibn Al Kow Rizmi**, el cual escribió sobre los años 800 y 825 su obra ***Quitad Al Mugabala***, donde se recogía el sistema de numeración hindú y el concepto del cero. **Fibonacci**, tradujo la obra al latín y la llamó: ***Algoritmi Dicit***.

El lenguaje algorítmico es aquel que implementa una solución teórica a un problema indicando las operaciones a realizar y el orden en el que se deben efectuarse. Por ejemplo en el caso de que nos encontremos en casa con una bombilla fundida en una lámpara, un posible algoritmo sería:

(1) Comprobar si hay bombillas de repuesto

(2) En el caso de que las haya, sustituir la bombilla anterior por la nueva

(3) Si no hay bombillas de repuesto, bajar a comprar una nueva a la tienda y sustituir la vieja por la nueva

Los algoritmos son la base de la programación de ordenadores, ya que los **programas de ordenador** se puede entender que son algoritmos escritos en un código especial entendible por un ordenador.

Lo malo del diseño de algoritmos está en que no podemos escribir lo que deseemos, el lenguaje ha utilizar no debe dejar posibilidad de duda, debe recoger todas las posibilidades.

Por lo que los tres pasos anteriores pueden ser mucho más largos:

[1] Comprobar si hay bombillas de repuesto

(1.1) Abrir el cajón de las bombillas

(1.2) Observar si hay bombillas

[2] Si hay bombillas:

(2.1) Coger la bombilla

(2.2) Coger una silla

(2.3) Subirse a la silla

(2.4) Poner la bombilla en la lámpara

[3] Si no hay bombillas

(3.1) Abrir la puerta

(3.2) Bajar las escaleras....

Cómo se observa en un algoritmo las instrucciones pueden ser más largas de lo que parecen, por lo que hay que determinar qué instrucciones se pueden utilizar y qué instrucciones no se pueden utilizar. En el caso de los algoritmos preparados para el ordenador, se pueden utilizar sólo instrucciones muy concretas.

(1.3.2) características de los algoritmos

características que deben de cumplir los algoritmos obligatoriamente

- **Un algoritmo debe resolver el problema para el que fue formulado**. Lógicamente no sirve un algoritmo que no resuelve ese problema. En el caso de los programadores, a veces crean algoritmos que resuelven problemas diferentes al planteado.
- **Los algoritmos son independientes del ordenador**. Los algoritmos se escriben para poder ser utilizados en cualquier máquina.
- **Los algoritmos deben de ser precisos**. Los resultados de los cálculos deben de ser exactos, de manera rigurosa. No es válido un algoritmo que sólo aproxime la solución.
- **Los algoritmos deben de ser finitos**. Deben de finalizar en algún momento. No es un algoritmo válido aquel que produce situaciones en las que el algoritmo no termina.
- **Los algoritmos deben de poder repetirse**. Deben de permitir su ejecución las veces que haga falta. No son válidos los que tras ejecutarse una vez ya no pueden volver a hacerlo por la razón que sea.

- **Validez.** Un algoritmo es válido si carece de errores. Un algoritmo puede resolver el problema para el que se planteó y sin embargo no ser válido debido a que posee errores
- **Eficiencia.** Un algoritmo es eficiente si obtiene la solución al problema en poco tiempo. No lo es si es lento en obtener el resultado.
- **Óptimo.** Un algoritmo es óptimo si es el más eficiente posible y no contiene errores. La búsqueda de este algoritmo es el objetivo prioritario del programador. No siempre podemos garantizar que el algoritmo hallado es el óptimo, a veces sí.

(1.3.3) elementos que conforman un algoritmo

- **Entrada.** Los datos iniciales que posee el algoritmo antes de ejecutarse.
- **Proceso.** Acciones que lleva a cabo el algoritmo.
- **Salida**. Datos que obtiene finalmente el algoritmo.

(1.3.4) fases en la creación de algoritmos

Hay tres fases en la elaboración de un algoritmo:

(1) **Análisis.** En esta se determina cuál es exactamente el problema a resolver. Qué datos forman la entrada del algoritmo y cuáles deberán obtenerse como salida.

(2) **Diseño**. Elaboración del algoritmo.

(3) **Prueba**. Comprobación del resultado. Se observa si el algoritmo obtiene la salida esperada para todas las entradas.

(1.4) aplicaciones

(1.4.1) programas y aplicaciones

- **Programa.** La definición de la **RAE** es: ***Conjunto unitario de instrucciones que permite a un ordenador realizar funciones diversas, como el tratamiento de textos, el diseño de gráficos, la resolución de problemas matemáticos, el manejo de bancos de datos***, etc. Pero normalmente se entiende por programa un **conjunto de instrucciones ejecutables por un ordenador.**

Un **programa estructurado** es un programa que cumple las condiciones de un algoritmo (finitud, precisión, repetición, resolución del problema,...)

- **Aplicación.** Software formado por uno o más programas, la documentación de los mismos y los archivos necesarios para su funcionamiento, de modo que el conjunto completo de archivos forman una herramienta de trabajo en un ordenador.

Normalmente en el lenguaje cotidiano no se distingue entre aplicación y programa; en nuestro caso entenderemos que la aplicación es un software completo que cumple la función completa para la que fue diseñado, mientras que un programa es el resultado de ejecutar un cierto código entendible por el ordenador.

(1.4.2) historia del software. La crisis del software

Los primeros ordenadores cumplían una única programación que estaba definida en los componentes eléctricos que formaban el ordenador.

La idea de que el ordenador hiciera varias tareas (ordenador programable o multipropósito) hizo que se idearan las **tarjetas perforadas.** En ellas se utilizaba código binario, de modo que se hacían agujeros en ellas para indicar el código 1 o el cero. Estos "primeros programas" lógicamente servían para hacer tareas muy concretas.

La llegada de ordenadores electrónicos más potentes hizo que los ordenadores se convirtieran en verdaderas máquinas digitales que seguían utilizando el 1 y el 0 del código binario pero que eran capaces de leer miles de unos y ceros. Empezaron a aparecer los primeros lenguajes de programación que escribían código más entendible por los humanos que posteriormente era convertido al código entendible por la máquina.

Inicialmente la creación de aplicaciones requería escribir pocas líneas de código en el ordenador, por lo que no había una técnica especificar a la hora de crear programas. Cada programador se defendía como podía generando el código a medida que se le ocurría.

Poco a poco las funciones que se requerían a los programas fueron aumentando produciendo miles de líneas de código que al estar desorganizada hacían casi imposible su mantenimiento. Sólo el programador que había escrito el código era capaz de entenderlo y eso no era en absoluto práctico.

La llamada **crisis del software** ocurrió cuando se percibió que se gastaba más tiempo en hacer las modificaciones a los programas que en volver a crear el software. La razón era que ya se habían codificado millones de líneas de código antes de que se definiera un buen método para crear los programas.

La solución a esta crisis ha sido la definición de la **ingeniería del software** como un oficio que requería un método de trabajo similar al del resto de ingenierías. La búsqueda de una metodología de trabajo que elimine esta crisis parece que aún no está resuelta, de hecho los métodos de trabajo siguen redefiniéndose una y otra vez.

(1.4.3) el ciclo de vida de una aplicación

Una de las cosas que se han definido tras el nacimiento de la ingeniería del software ha sido el ciclo de vida de una aplicación. El ciclo de vida define los pasos que sigue el proceso de creación de una aplicación desde que se propone hasta que finaliza su construcción. Los pasos son:

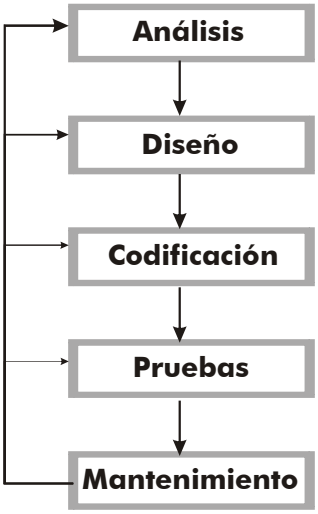


Ilustración 3, Ciclo de vida de una aplicación

(1) **Análisis.** En esta fase se determinan los requisitos que tiene que cumplir la aplicación. Se anota todo aquello que afecta al futuro funcionamiento de la aplicación. Este paso le realiza un analista

(2) **Diseño.** Se especifican los esquemas de diseño de la aplicación. Estos esquemas forman los ***planos*** del programador, los realiza el analista y representan todos los aspectos que requiere la creación de la aplicación.

(3) **Codificación.** En esta fase se pasa el diseño a código escrito en algún lenguaje de programación. Esta es la primera labor que realiza el programador

(4) **Pruebas.** Se trata de comprobar que el funcionamiento de la aplicación es la adecuada. Se realiza en varias fases:

a) **Prueba del código.** Las realizan programadores. Normalmente programadores distintos a los que crearon el código, de ese modo la prueba es más independiente y generará resultados más óptimos.

b) **Versión alfa.** Es una primera versión terminada que se revisa a fin de encontrar errores. Estas pruebas conviene que sean hechas por personal no informático.

c) **Versión beta.** Versión casi definitiva del software en la que no se estiman fallos, pero que se distribuye a los clientes para que

encuentren posibles problemas. A veces está versión acaba siendo la definitiva (como ocurre con muchos de los programas distribuidos libremente por Internet).

(5) **Mantenimiento.** Tiene lugar una vez que la aplicación ha sido ya distribuida, en esta fase se asegura que el sistema siga funcionando aunque cambien los requisitos o el sistema para el que fue diseñado el software. Antes esos cambios se hacen los arreglos pertinentes, por lo que habrá que retroceder a fases anteriores del ciclo de vida.

(1.5) errores

Cuando un programa obtiene una salida que no es la esperada, se dice que posee errores. Los errores son uno de los caballos de batalla de los programadores ya que a veces son muy difíciles de encontrar (de ahí que hoy en día en muchas aplicaciones se distribuyan ***parches*** para subsanar errores no encontrados en la creación de la aplicación).

tipos de errores

- **Error del usuario.** Errores que se producen cuando el usuario realiza algo inesperado y el programa no reacciona apropiadamente.
- **Error del programador.** Son errores que ha cometido el programador al generar el código. La mayoría de errores son de este tipo.
- **Errores de documentación**. Ocurren cuando la documentación del programa no es correcta y provoca fallos en el manejo
- **Error de interfaz**. Ocurre si la interfaz de usuario de la aplicación es enrevesada para el usuario impidiendo su manejo normal. También se llaman así los errores de protocolo entre dispositivos.
- **Error de entrada / salida o de comunicaciones.** Ocurre cuando falla la comunicación entre el programa y un dispositivo (se desea imprimir y no hay papel, falla el teclado,...)
- **Error fatal.** Ocurre cuando el hardware produce una situación inesperado que el software no puede controlar (el ordenador se cuelga, errores en la grabación de datos,...)
- **Error de ejecución.** Ocurren cuando la ejecución del programa es más lenta de lo previsto.

La labor del programador es predecir, encontrar y subsanar (si es posible) o al menos controlar los errores. Una mala gestión de errores causa experiencias poco gratas al usuario de la aplicación.

(1.6) lenguajes de programación

(1.6.1) breve historia de los lenguajes de programación

inicios de la programación

Charles Babbage definió a mediados del siglo XIX lo que él llamó la **máquina analítica**. Se considera a esta máquina el diseño del primer ordenador. La realidad es que no se pudo construir hasta el siglo siguiente. El caso es que su colaboradora **Ada Lovelace** escribió en tarjetas perforadas una serie de instrucciones que la máquina iba a ser capaz de ejecutar. Se dice que eso significó el inicio de la ciencia de la programación de ordenadores.

En la segunda guerra mundial debido a las necesidades militares, la ciencia de la computación prospera y con ella aparece el famoso **ENIAC** (***Electronic Numerical Integrator And Calculator***), que se programaba cambiando su circuitería. Esa es la primera forma de programar (que aún se usa en numerosas máquinas) que sólo vale para **máquinas de único propósito**. Si se cambia el propósito, hay que modificar la máquina.

código máquina. primera generación de lenguajes (1GL)

No mucho más tarde apareció la idea de que las máquinas fueran capaces de realizar más de una aplicación. Para lo cual se ideó el hecho de que hubiera una memoria donde se almacenaban esas instrucciones. Esa memoria se podía rellenar con datos procedentes del exterior. Inicialmente se utilizaron tarjetas perforadas para introducir las instrucciones.

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

Sólo se ha utilizado por los programadores en los inicios de la informática. Su incomodidad de trabajo hace que sea impensable para ser utilizado hoy en día. Pero cualquier programa de ordenador debe, finalmente, ser convertido a este código para que un ordenador puede ejecutar las instrucciones de dicho programa.

Un detalle a tener en cuenta es que el código máquina es distinto para cada tipo de procesador. Lo que hace que los programas en código máquina no sean portables entre distintas máquinas.

lenguaje ensamblado. segunda generación de lenguajes (2GL)

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

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

La idea es la siguiente: si en el código máquina, el número binario 0000 significa sumar, y el número 0001 significa restar. Una instrucción máquina que sumara el número 8 (00001000 en binario) al número 16 (00010000 en binario) sería:

```
0000 00001000 00010000
```

Realmente no habría espacios en blanco, el ordenador entendería que los primeros cuatro BITS representan la instrucción y los 8 siguientes el primer número y los ocho siguientes el segundo número (suponiendo que los números ocupan 8 bits). Lógicamente trabajar de esta forma es muy complicado. Por eso se podría utilizar la siguiente traducción en ensamblador:

```
SUM  8 16
```

Que ya se entiende mucho mejor.

Ejemplo[2] (programa que saca el texto “Hola mundo” por pantalla):

```
DATOS SEGMENT
     saludo db "Hola mundo!!!","$"
DATOS ENDS
CODE SEGMENT
     assume cs:code,ds:datos
START PROC
     mov ax,datos
     mov ds,ax
     mov  dx,offset saludo
     mov  ah,9
     int  21h
     mov  ax,4C00h
     int  21h
START ENDP
CODE ENDS
     END START
```

[2] Ejemplo tomado de la página **http://www.victorsanchez2.net**

Puesto que el ensamblador es una representación textual pero exacta del código máquina; cada programa sólo funcionará para la máquina en la que fue concebido el programa; es decir, no es **portable**.

La ventaja de este lenguaje es que se puede controlar absolutamente el funcionamiento de la máquina, lo que permite crear programas muy eficientes. Lo malo es precisamente que hay que conocer muy bien el funcionamiento de la computadora para crear programas con esta técnica. Además las líneas requeridas para realizar una tarea se disparan ya que las instrucciones de la máquina son excesivamente simples.

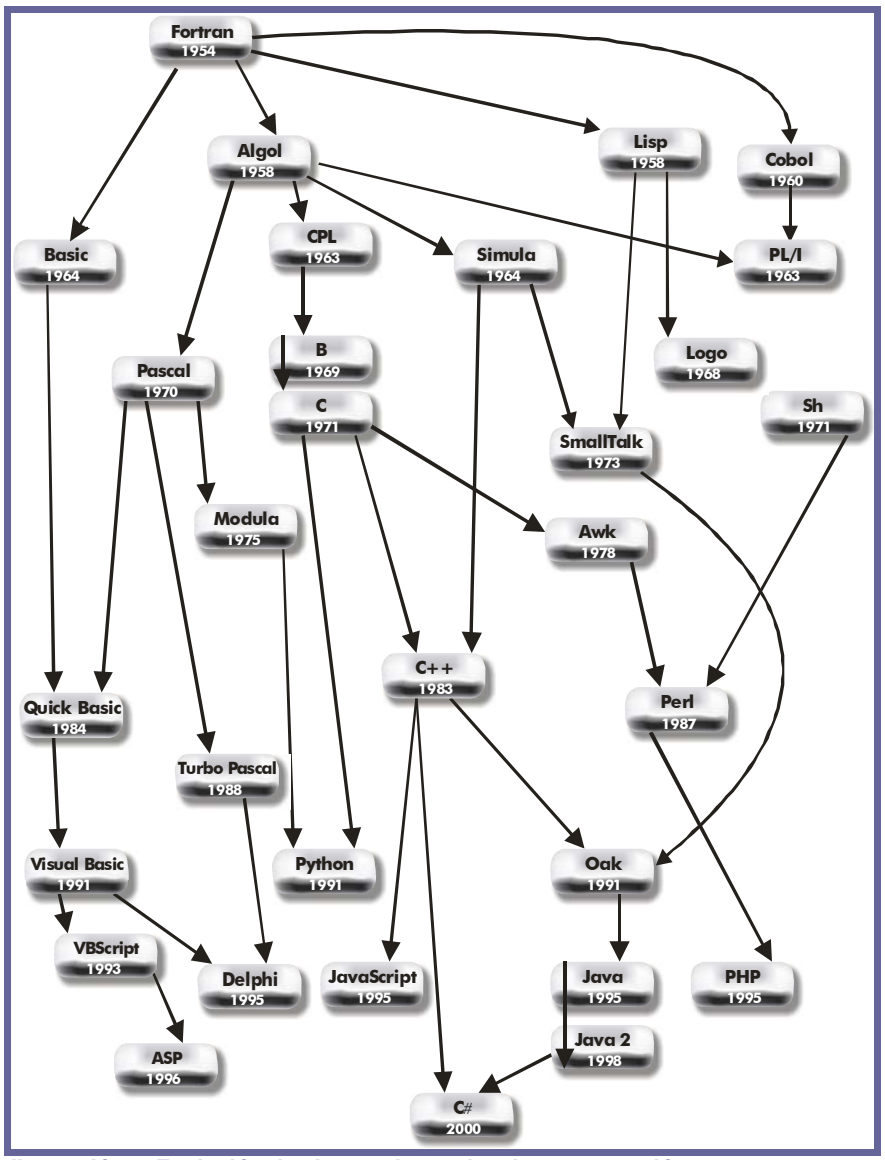


Ilustración 4, Evolución de algunos lenguajes de programación

lenguajes de alto nivel. lenguajes de tercera generación (3GL)

Aunque el ensamblador significó una notable mejora sobre el código máquina, seguía siendo excesivamente críptico. De hecho para hacer un programa sencillo requiere miles y miles de líneas de código.

Para evitar los problemas del ensamblador apareció la tercera generación de lenguajes de programación, la de los lenguajes de alto nivel. En este caso el código vale para cualquier máquina pero deberá ser traducido mediante software especial que adaptará el código de alto nivel al código máquina correspondiente. Esta traducción es necesaria ya que el código en un lenguaje de alto nivel no se parece en absoluto al código máquina.

Tras varios intentos de representar lenguajes, en 1957 aparece el que se considera el primer lenguaje de alto nivel, el **FORTRAN** (***FORmula TRANslation)***, lenguaje orientado a resolver fórmulas matemáticos. Por ejemplo la forma en FORTRAN de escribir el texto ***Hola mundo*** por pantalla es:

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

Poco a poco fueron evolucionando los lenguajes formando lenguajes cada vez mejores (ver). Así en 1958 se crea **LISP** como lenguaje declarativo para expresiones matemáticas.

Programa que escribe ***Hola mundo*** en lenguaje LISP:

```
(format t "¡Hola, mundo!")
```

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

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
MAIN SECTION.
DISPLAY "Hola mundo"
STOP RUN.
```

BASIC se creo en el año 1964 como lenguaje de programación sencillo de aprender en 1964 y ha sido, y es, uno de los lenguajes más populares. En 1968 se crea **LOGO** para enseñar a programar a los niños. **Pascal** se creo con la misma idea académica pero siendo ejemplo de lenguaje estructurado para

programadores avanzados. El creador del Pascal (**Niklaus Wirdth**) creo **Modula** en 1977 siendo un lenguaje estructurado para la programación de sistemas (intentando sustituir al **C**).

Programa que escribe por pantalla ***Hola mundo*** en lenguaje Pascal):

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

lenguajes de cuarta generación (4GL)

En los años 70 se empezó a utilizar éste término para hablar de lenguajes en los que apenas hay código y en su lugar aparecen indicaciones sobre qué es lo que el programa debe de obtener. Se consideraba que el lenguaje SQL (muy utilizado en las bases de datos) y sus derivados eran de cuarta generación. Los lenguajes de consulta de datos, creación de formularios, informes,... son lenguajes de cuarto nivel. Aparecieron con los sistemas de base de datos

Actualmente se consideran lenguajes de éste tipo a aquellos lenguajes que se programan sin escribir casi código (lenguajes visuales), mientras que también se propone que éste nombre se reserve a los lenguajes orientados a objetos.

lenguajes orientados a objetos

En los 80 llegan los lenguajes preparados para la programación orientada a objetos todos procedentes de **Simula** (1964) considerado el primer lenguaje con facilidades de uso de objetos. De estos destacó inmediatamente **C++**.

A partir de **C++** aparecieron numerosos lenguajes que convirtieron los lenguajes clásicos en lenguajes orientados a objetos (y además con mejoras en el entorno de programación, son los llamados lenguajes **visuales**): **Visual Basic**, **Delphi** (versión orientada a objetos de Pascal), **Visual C++,...**

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

El programa ***Hola mundo*** en C# sería:

```
using System;

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

lenguajes para la web

La popularidad de Internet ha producido lenguajes híbridos que se mezclan con el código **HTML** con el que se crean las páginas web. HTML no es un lenguaje en sí sino un formato de texto pensado para crear páginas web. Éstos lenguajes se usan para poder realizar páginas web más potentes.

Son lenguajes interpretados como **JavaScript** o **VB Script**, o lenguajes especiales para uso en servidores como **ASP**, **JSP** o **PHP**. Todos ellos permiten crear páginas web usando código mezcla de página web y lenguajes de programación sencillos.

Ejemplo, página web escrita en lenguaje HTML y JavaScript que escribe en pantalla "Hola mundo" (de color rojo aparece el código en JavaScript):

```
<html>
<head>
   <title>Hola Mundo</title>
</head>
<body>
   <script type="text/javascript">
      document.write("¡Hola mundo!");
   </script>
</body>
</html>
```

(1.6.2) tipos de lenguajes

Según el estilo de programación se puede hacer esta división:

- **Lenguajes imperativos.** Son lenguajes donde las instrucciones se ejecutan secuencialmente y van modificando la memoria del ordenador para producir las salidas requeridas. La mayoría de lenguajes (**C, Pascal, Basic, Cobol,** ...son de este tipo. Dentro de estos lenguajes están también los lenguajes orientados a objetos (**C++, Java, C#,...)**
- **Lenguajes declarativos.** Son lenguajes que se concentran más en el qué, que en el cómo (cómo resolver el problema es la pregunta a realizarse cuando se usan lenguajes imperativos). Los lenguajes que se programan usando la pregunta ¿qué queremos? son los declarativos. El más conocido de ellos es el lenguaje de consulta de Bases de datos, **SQL**.
- **Lenguajes funcionales.** Definen funciones, expresiones que nos responden a través de una serie de argumentos. Son lenguajes que usan expresiones matemáticas, absolutamente diferentes del lenguaje usado por las máquinas. El más conocido de ellos es el **LISP.**
- **Lenguajes lógicos.** Lenguajes utilizados para resolver expresiones lógicas. Utilizan la lógica para producir resultados. El más conocido es el **PROLOG**.

(1.6.3) intérpretes

A la hora de convertir un programa en código máquina, se pueden utilizar dos tipos de software: **intérpretes** y **compiladores**.

En el caso de los intérpretes se convierte cada línea a código máquina y se ejecuta ese código máquina antes de convertir la siguiente línea. De esa forma si las dos primeras líneas son correctas y la tercera tiene un fallo de sintaxis, veríamos el resultado de las dos primeras líneas y al llegar a la tercera se nos notificaría el fallo y finalizaría la ejecución.

El intérprete hace una simulación de modo que parece que la máquina entiende directamente las instrucciones del lenguaje, pareciendo que ejecuta cada instrucción (como si fuese código máquina directo).

El **BASIC** era un lenguaje interpretado, se traducía línea a línea. Hoy en día la mayoría de los lenguajes integrados en páginas web son interpretados, la razón es que como la descarga de Internet es lenta, es mejor que las instrucciones se vayan traduciendo según van llegando en lugar de cargar todas en el ordenador. Por eso lenguajes como **JavaScript** (o incluso, en parte, **Java**) son interpretados.

proceso

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

(1) Lee la primera instrucción

(2) Comprueba si es correcta

(3) Convierte esa instrucción al código máquina equivalente

(4) Lee la siguiente instrucción

(5) Vuelve al paso 2 hasta terminar con todas las instrucciones

ventajas

- Se tarda menos en crear el primer código máquina. El programa se ejecuta antes.
- No hace falta cargar todas las líneas para empezar a ver resultados (lo que hace que sea una técnica idónea para programas que se cargan desde Internet)

desventajas

- El código máquina producido es peor ya que no se optimiza al valorar una sola línea cada vez. El código optimizado permite estudiar varias líneas a la vez para producir el mejor código máquina posible, por ello no es posible mediante el uso de intérpretes.

- Todos los errores son errores en tiempo de ejecución, no se pueden detectar antes de lanzar el programa. Esto hace que la depuración de los errores sea más compleja.
- El código máquina resultante gasta más espacio.
- Hay errores difícilmente detectables, ya que para que los errores se produzcan, las líneas de errores hay que ejecutarlas. Si la línea es condicional hasta que no probemos todas las posibilidades del programa, no sabremos todos los errores de sintaxis cometidos.

(1.6.4) compiladores

Se trata de software que traduce las instrucciones de un lenguaje de programación de alto nivel a código máquina. La diferencia con los intérpretes reside en que se analizan todas las líneas antes de empezar la traducción.

Durante muchos años, los lenguajes potentes han sido compilados. El uso masivo de Internet ha propiciado que esta técnica a veces no sea adecuada y haya lenguajes modernos interpretados o semi interpretados, mitad se compila hacia un código intermedio y luego se interpreta línea a línea (esta técnica la siguen **Java** y los lenguajes de la plataforma **.NET** de Microsoft).

ventajas

- Se detectan errores antes de ejecutar el programa (errores de compilación)
- El código máquina generado es más rápido (ya que se optimiza)
- Es más fácil hacer procesos de depuración de código

desventajas

- El proceso de compilación del código es lento.
- No es útil para ejecutar programas desde Internet ya que hay que descargar todo el programa antes de traducirle, lo que ralentiza mucho su uso.

(1.7) programación

(1.7.1) introducción

La programación consiste en pasar algoritmos a algún lenguaje de ordenador a fin de que pueda ser entendido por el ordenador. La programación de ordenadores comienza en los años 50 y su evolución a pasado por diversos pasos.

La programación se puede realizar empleando diversas **técnicas** o métodos. Esas técnicas definen los distintos tipos de programaciones.

(1.7.2) programación desordenada

Se llama así a la programación que se realizaba en los albores de la informática (aunque desgraciadamente en la actualidad muchos programadores siguen empleándola). En este estilo de programación, predomina el ***instinto*** del programador por encima del uso de cualquier método lo que provoca que la corrección y entendimiento de este tipo de programas sea casi ininteligible.

Ejemplo de uso de esta programación (listado en Basic clásico):

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

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

(1.7.3) programación estructurada

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

- **Secuencias** (instrucciones que se generan secuencialmente)
- **Alternativas** (sentencias ***if***)
- **Iterativas** (bucles condicionales)

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

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

La ventaja de esta programación está en que es más legible (aunque en este caso el código es casi más sencillo en su versión desordenada). **Todo programador debería escribir código de forma estructurada.**

(1.7.4) programación modular

Completa la programación anterior permitiendo la definición de módulos independientes cada uno de los cuales se encargará de una tarea del programa. De este forma el programador se concentra en la codificación de cada módulo haciendo más sencilla esta tarea. Al final se deben integrar los módulos para dar lugar a la aplicación final.

El código de los módulos puede ser invocado en cualquier parte del código. Realmente cada módulo se comporta como un subprograma que, partir de unas determinadas entradas obtienen unas salidas concretas. Su funcionamiento no depende del resto del programa por lo que es más fácil encontrar los errores y realizar el mantenimiento.

(1.7.5) programación orientada a objetos

Es la más novedosa, se basa en intentar que el código de los programas se parezca lo más posible a la forma de pensar de las personas. Las aplicaciones se representan en esta programación como una serie de objetos independientes que se comunican entre sí.

Cada objeto posee datos y métodos propios, por lo que los programadores se concentran en programar independientemente cada objeto y luego generar el código que inicia la comunicación entre ellos.

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

Ejemplo (código Java):

```
/**
 *Calcula los primos del 1 al 1000
*/
public class primos {
    /** Función principal */
    public static void main(String args[]){
        int nPrimos=10000;
        boolean primo[]=new boolean[nPrimos+1];
        short i;
        for (i=1;i<=nPrimos;i++) primo[i]=true;
        for (i=2;i<=nPrimos;i++){
            if (primo[i]){
                 for (int j=2*i;j<=nPrimos;j+=i){
                     primo[j]=false;                }
```

```
                }
            }
            for (i=1;i<=nPrimos;i++) {
                System.out.print(" "+i);
            }
        }
}
```

(Unidad 2) Metodología de la programación

(2.1) metodologías

(2.1.1) introducción

Se entiende por metodología el conjunto de reglas y pasos estrictos que se siguen para desarrollar una aplicación informática completa. Hay diversas metodologías, algunas incluso registradas (hay que pagar por utilizarlas).

Independientemente de la metodología utilizada suele haber una serie de pasos comunes a todas ellas (relacionados con el ciclo de vida de la aplicación):

(1) **Análisis**

(2) **Diseño**

(3) **Codificación**

(4) **Ejecución**

(5) **Prueba**

(6) **Mantenimiento**

(2.1.2) análisis

Al programar aplicaciones siempre se debe realizar un análisis. El análisis estudia los requisitos que ha de cumplir la aplicación. El resultado del análisis es una **hoja de especificaciones** en la que aparece los requerimientos de la aplicación. Esta hoja es redactada por el o la analista, la persona responsable del proceso de creación de la aplicación.

En la creación de algoritmos sencillos, el análisis consistiría únicamente en:

- **Determinar las entradas**. Es decir, los datos que posee el algoritmo cuando comienza su ejecución. Esos datos permiten obtener el resultado.

- **Determinar las salidas**. Es decir, los datos que obtiene el algoritmo como resultado. Lo que el algoritmo devuelve al usuario.
- **Determinar el proceso**. Se estudia cuál es el proceso que hay que realizar.

(2.1.3) diseño

En esta fase se crean esquemas que simbolizan a la aplicación. Estos esquemas los elaboran analistas. Gracias a estos esquemas se simboliza la aplicación. Estos esquemas en definitiva se convierte en la documentación fundamental para plasmar en papel lo que el programador debe hacer.

En estos esquemas se pueden simbolizar: la organización de los datos de la aplicación, el orden de los procesos que tiene que realizar la aplicación, la estructura física (en cuanto a archivos y carpetas) que utilizará la aplicación, etc.

La creación de estos esquemas se puede hacer en papel, o utilizar una **herramienta CASE** para hacerlo.

En el caso de la creación de algoritmos, conviene en esta fase usar el llamado **diseño descendente**. Mediante este diseño el problema se divide en módulos, que, a su vez, se vuelven a dividir a fin de solucionar problemas más concretos. Al diseño descendente se le llama también **top-down**. Gracias a esta técnica un problema complicado se divide en pequeños problemas que son más fácilmente solucionables.

Siempre existe en el diseño la **zona principal** que es el programa principal que se ejecutará cuando el programa esté codificado en un lenguaje de programación.

En la construcción de aplicaciones complejas en esta fase se utilizan gran cantidad de esquemas para describir la organización de los datos y los procedimientos que ha de seguir el programa. En pequeños algoritmos se utilizan esquemas más sencillos.

(2.1.4) codificación

Escritura de la aplicación utilizando un lenguaje de programación (C, Pascal, C++, Java,...). Normalmente la herramienta utilizada en el diseño debe ser compatible con el lenguaje que se utilizará para codificar. Es decir si se utiliza un lenguaje orientado a objetos, la herramienta de diseño debe ser una herramienta que permita utilizar objetos.

(2.1.5) ejecución

Tras la escritura del código, mediante un software especial se traduce a código interpretable por el ordenador (código máquina). En este proceso pueden detectarse errores en el código que impiden su transformación. En ese caso el software encargado de la traducción (normalmente un **compilador** o un **intérprete**) avisa de esos errores para que el programador los pueda corregir.

(2.1.6) prueba

Se trata de testear la aplicación para verificar que su funcionamiento es el correcto. Para ello se comprueban todas las entradas posibles, comprobando que las salidas son las correspondientes.

(2.1.7) mantenimiento

En esta fase se crea la documentación del programa (paso fundamental en la creación de aplicaciones). Gracias a esa documentación se pueden corregir futuros errores o renovar el programa para incluir mejoras detectadas, operaciones que también se realizan en esta fase.

(2.2) notaciones para el diseño de algoritmos

(2.2.1) diagramas de flujo

introducción

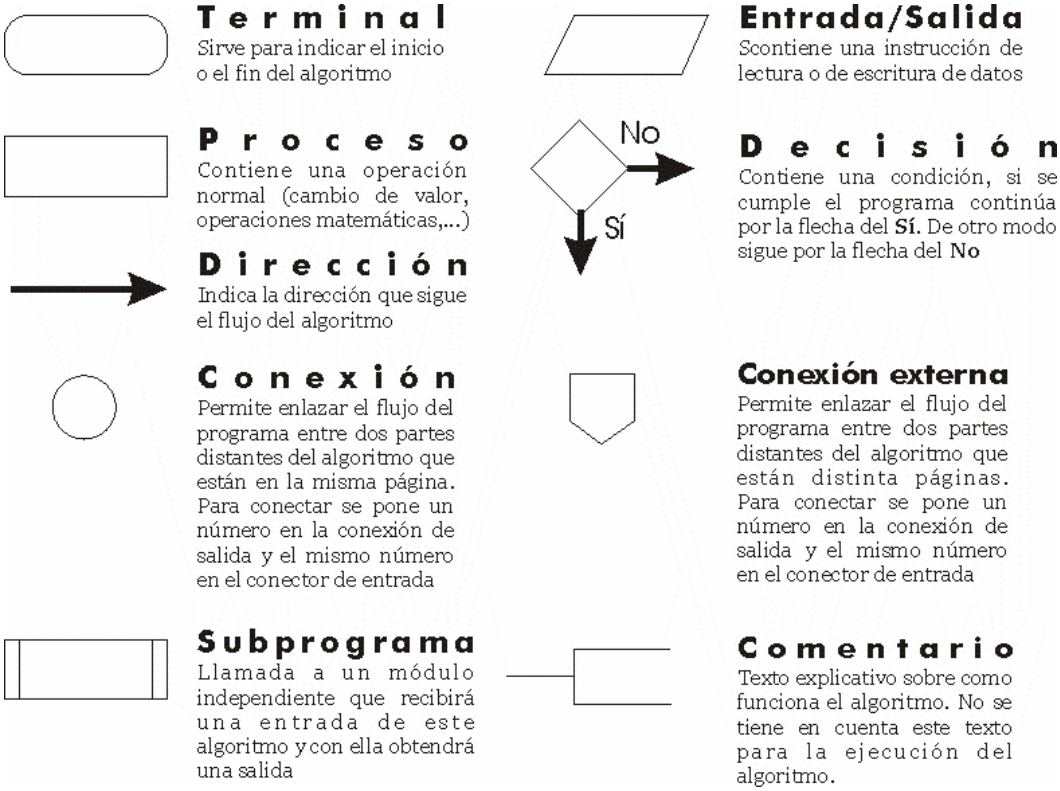
Es el esquema más viejo de la informática. Se trata de una notación que pretende facilitar la escritura o la comprensión de algoritmos. Gracias a ella se esquematiza el flujo del algoritmo. Fue muy útil al principio y todavía se usa como apoyo para explicar ciertos algoritmos. Si los algoritmos son complejos, este tipo de esquemas no son adecuados.

No obstante cuando el problema se complica, resulta muy complejo de realizar y de entender. De ahí que actualmente, sólo se use con fines educativos y no en la práctica. Pero sigue siendo interesante en el aprendizaje de la creación de algoritmos.

Los diagramas utilizan símbolos especiales que ya están normalizados por organismos de estandarización como **ANSI** e **ISO**.

símbolos principales

La lista de símbolos que generalmente se utiliza en los diagramas de flujo es:



Ejemplo:

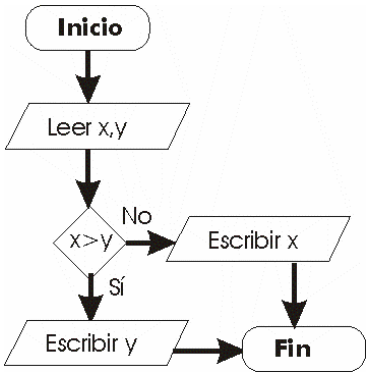


Ilustración 5, Diagrama de flujo que escribe el mayor de dos números leídos

desventajas de los diagramas de flujo

Los diagramas de flujo son interesantes como primer acercamiento a la programación ya que son fáciles de entender. De hecho se utilizan fuera de la

programación como esquema para ilustrar el funcionamiento de algoritmos sencillos.

Sin embargo cuando el algoritmo se complica, el diagrama de flujo se convierte en ininteligible. Además los diagramas de flujo no facilitan el aprendizaje de la programación estructurada, con lo que no se aconseja su uso a los programadores para diseñar algoritmos.

(2.2.2) pseudocódigo

introducción

Las bases de la programación estructurada fueron enunciadas por **Niklaus Wirdth**. Según este científico cualquier problema algorítmico podía resolverse con el uso de estos tres tipos de instrucciones:

- **Secuenciales.** Instrucciones que se ejecutan en orden normal. El flujo del programa ejecuta la instrucción y pasa a ejecutar la siguiente.
- **Alternativas.** Instrucciones en las que se evalúa una condición y dependiendo si el resultado es verdadero o no, el flujo del programa se dirigirá a una instrucción o a otra.
- **Iterativas.** Instrucciones que se repiten continuamente hasta que se cumple una determinada condición.

El tiempo le ha dado la razón y ha generado una programación que insta a todo programador a utilizar sólo instrucciones de esos tres tipos. Es lo que se conoce como programación estructurada.

El propio Niklaus Wirdth diseñó el lenguaje **Pascal** como el primer lenguaje estructurado. Lo malo es que el Pascal al ser lenguaje completo incluye instrucciones excesivamente orientadas al ordenador.

Por ello se aconseja para el diseño de algoritmos estructurados el uso de un lenguaje especial llamado pseudocódigo, que además se puede traducir a cualquier idioma (Pascal está basado en el inglés).

El pseudocódigo además permite el diseño modular de programas y el diseño descendente gracias a esta posibilidad

Hay que tener en cuenta que existen multitud de ***pseudocódigos***, es decir **no hay un pseudocódigo 100% estándar.** Pero sí hay gran cantidad de detalles aceptados por todos los que escriben pseudocódigos. Aquí se comenta el pseudocódigo que parece más aceptado en España. Hay que tener en cuenta que el pseudocódigo se basa en Pascal, por lo que la traducción a Pascal es casi directa.

El pseudocódigo son instrucciones escritas en un lenguaje orientado a ser entendido por un ordenador. Por ello en pseudocódigo sólo se pueden utilizar ciertas instrucciones. La escritura de las instrucciones debe cumplir reglas muy estrictas. Las únicas permitidas son:

- **De Entrada /Salida**. Para leer o escribir datos desde el programa hacia el usuario.
- **De proceso.** Operaciones que realiza el algoritmo (suma, resta, cambio de valor,...)
- **De control de flujo**. Instrucciones alternativas o iterativas (bucles y condiciones).
- **De declaración.** Mediante las que se crean variables y subprogramas.
- **Llamadas a subprogramas.**
- **Comentarios.** Notas que se escriben junto al pseudocódigo para explicar mejor su funcionamiento.

escritura en pseudocódigo

Las instrucciones que resuelven el algoritmo en pseudocódigo deben de estar encabezadas por la palabra **inicio** (en inglés **begin**) y cerradas por la palabra **fin** (en inglés **end**). Entre medias de estas palabras se sitúan el resto de instrucciones. Opcionalmente se puede poner delante del inicio la palabra **programa** seguida del nombre que queramos dar al algoritmo.

En definitiva la estructura de un algoritmo en pseudocódigo es:

```
programa nombreDelPrograma
inicio
  instrucciones
  ....
fin
```

Hay que tener en cuenta estos detalles:

- Aunque no importan las mayúsculas y minúsculas en pseudocódigo, se aconsejan las minúsculas porque su lectura es más clara y además porque hay muchos lenguajes en los que sí importa el hecho de hecho escribir en mayúsculas o minúsculas (C, Java, ...)
- Se aconseja que las instrucciones dejen un espacio (sangría) a la izquierda para que se vea más claro que están entre el inicio y el fin. Esta forma de escribir algoritmos permite leerlos mucho mejor.

comentarios

En pseudocódigo los comentarios que se deseen poner (y esto es una práctica muy aconsejable) se ponen con los símbolos // al principio de la línea de comentario (en algunas notaciones se escribe **). Cada línea de comentario debe comenzar con esos símbolos:

```
inicio
  instrucciones
  //comentario
  //comentario
  instrucciones
fin
```

(2.3) creación de algoritmos

(2.3.1) instrucciones

Independientemente de la notación que utilicemos para escribir algoritmos, éstos contienen instrucciones, acciones a realizar por el ordenador. Lógicamente la escritura de estas instrucciones sigue unas normas muy estrictas. Las instrucciones pueden ser de estos tipos:

- **Primitivas.** Son acciones sobre los datos del programa. Son:
  - Asignación
  - Instrucciones de Entrada/Salida
- **Declaraciones**. Obligatorias en el pseudocódigo, opcionales en otros esquemas. Sirven para advertir y documentar el uso de variables y subprogramas en el algoritmo.
- **Control**. Sirven para alterar el orden de ejecución del algoritmo. En general el algoritmo se ejecuta secuencialmente. Gracias a estas instrucciones el flujo del algoritmo depende de ciertas condiciones que nosotros mismos indicamos.

(2.3.2) instrucciones de declaración

Sólo se utilizan en el pseudocódigo. Indican el nombre y las características de las **variables** que se utilizan en el algoritmo. Las variables son nombres a los que se les asigna un determinado valor y son la base de la programación. Al nombre de las variables se le llama **identificador**.

identificadores

Los algoritmos necesitan utilizar datos. Los datos se identifican con un determinado **identificador** (nombre que se le da al dato). Este nombre:

- Sólo puede contener letras, números y el carácter _
- Debe comenzar por una letra
- No puede estar repetido en el mismo algoritmo. No puede haber dos elementos del algoritmo (dos datos por ejemplo) con el mismo identificador.
- Conviene que sea aclarativo, es decir que represente lo mejor posible los datos que contiene. ***x*** no es un nombre aclarativo, ***saldo_mensual*** sí lo es.

Los valores posibles de un identificador deben de ser siempre del mismo tipo (lo cual es lógico puesto que un identificador almacena un dato). Es decir no puede almacenar primero texto y luego números.

declaración de variables

Es aconsejable al escribir pseudocódigo indicar las variables que se van a utilizar (e incluso con un comentario indicar para qué se van a usar). En el caso de los otros esquemas (diagramas de flujo y tablas de decisión) no se utilizan (lo que fomenta malos hábitos).

Esto se hace mediante la sección del pseudocódigo llamada **var**, en esta sección se colocan las variables que se van a utilizar. Esta sección se coloca antes del **inicio** del algoritmo. Y se utiliza de esta forma:

```
programa nombreDePrograma
var
  identificador1: tipoDeDatos
  identificador2: tipoDeDatos
  ....
inicio
  instrucciones
fin
```

El tipo de datos de la variable puede ser especificado de muchas formas, pero tiene que ser un tipo compatible con los que utilizan los lenguajes informáticos. Se suelen utilizar los siguientes tipos:

- **entero.** Permite almacenar valores enteros (sin decimales).
- **real.** Permite almacenar valores decimales.
- **carácter.** Almacenan un carácter alfanumérico.
- **lógico** (o **booleano**). Sólo permiten almacenar los valores **verdadero** o **falso.**

- **texto.** A veces indicando su tamaño (**texto**(20) indicaría un texto de hasta 20 caracteres) permite almacenar texto. Normalmente en cualquier lenguaje de programación se considera un tipo compuesto.

También se pueden utilizar datos más complejos, pero su utilización queda fuera de este tema.

Ejemplo de declaración:

```
var
  numero_cliente: entero // código único de cada cliente
  valor_compra: real //lo que ha comprado el cliente
  descuento: real //valor de descuento aplicable al
cliente
```

También se pueden declarar de esta forma:

```
var
  numero_cliente: entero // código único de cada cliente
  valor_compra, //lo que ha comprado el cliente
  descuento :real //valor de descuento aplicable al
cliente
```

La coma tras ***valor_compra*** permite declarar otra variable real.

constantes

Hay un tipo especial de variable llamada constante. Se utiliza para valores que no van a variar en ningún momento. Si el algoritmo utiliza valores constantes, éstos se declaran mediante una sección (que se coloca delante de la sección **var**) llamada **const** (de constante). Ejemplo:

```
programa ejemplo1
const
  PI=3.141592
  NOMBRE=”Jose”
var
  edad: entero
  sueldo: real
inicio
....
```

A las constantes se las asigna un valor mediante el símbolo **=.** Ese valor permanece constante (pi siempre vale 3.141592). Es conveniente (aunque en absoluto obligatorio) utilizar letras mayúsculas para declarar variables.

(2.3.3) instrucciones primitivas

Son instrucciones que se ejecutan en cuanto son leídas por el ordenador. En ellas sólo puede haber:

- Asignaciones (←)
- Operaciones (+, -, * /,...)
- Identificadores (nombres de variables o constantes)
- Valores (números o texto encerrado entre comillas)
- Llamadas a subprogramas

En el pseudocódigo se escriben entre el inicio y el fin. En los diagramas de flujo y tablas de decisión se escriben dentro de un rectángulo

instrucción de asignación

Permite almacenar un valor en una variable. Para asignar el valor se escribe el símbolo ←, de modo que:

```
identificador←valor
```

El identificador toma el valor indicado. Ejemplo:

```
x←8
```

Ahora ***x*** vale 8. Se puede utilizar otra variable en lugar de un valor. Ejemplo:

```
y←9
x←y
```

x vale ahora lo que vale ***y***, es decir x vale 9.

Los valores pueden ser:

- **Números.** Se escriben tal cual, el separador decimal suele ser el punto (aunque hay quien utiliza la coma).
- **Caracteres simples.** Los caracteres simples (un solo carácter) se escriben entre comillas simples: 'a', 'c',etc.
- **Textos.** Se escriben entre comillas doble "Hola"
- **Lógicos.** Sólo pueden valer **verdadero** o **falso** (se escriben tal cual)
- **Identificadores.** En cuyo caso se almacena el valor de la variable con dicho identificador. Ejemplo:

```
x←9
y←x //y valdrá nueve
```

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

Ejemplo:

```
x←(y*3)/2
```

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

+	Suma
-	Resta o cambio de signo
*	Producto
/	División
mod	Resto. Por ejemplo 9 **mod** 2 da como resultado 1
div	División entera. 9 **div** 2 da como resultado 4 (y no 4,5)
↑	Exponente. 9↑2 es 9 elevado a la 2

Hay que tener en cuenta la prioridad del operador. Por ejemplo la multiplicación y la división tienen más prioridad que la suma o la resta. Sí 9+6/3 da como resultado 5 y no 11. Para modificar la prioridad de la instrucción se utilizan paréntesis. Por ejemplo 9+(6/3)

(2.3.4) instrucciones de entrada y salida

lectura de datos

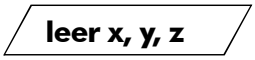
Es la instrucción que simula una lectura de datos desde el teclado. Se hace mediante la orden **leer** en la que entre paréntesis se indica el identificador de la variable que almacenará lo que se lea. Ejemplo (pseudocódigo):

```
leer(x)
```

El mismo ejemplo en un diagrama de flujo sería:

En ambos casos ***x*** contendrá el valor leído desde el teclado. Se pueden leer varias variables a la vez:

```
leer(x,y,z)
```



escritura de datos

Funciona como la anterior pero usando la palabra **escribir**. Simula la salida de datos del algoritmo por pantalla.

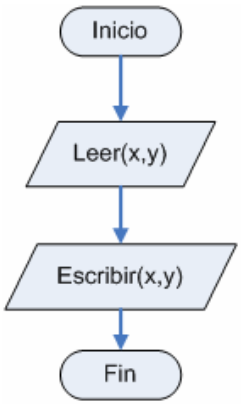
```
escribir(x,y,z)
```

ejemplo de algoritmo

El algoritmo completo que escribe el resultado de multiplicar dos números leídos por teclado sería (en pseudocódigo)

```
programa mayorDe2
var
  x,y: entero
inicio
  leer(x,y)
  escribir(x*y)
fin
```

En un diagrama de flujo:



(2.3.5) instrucciones de control

Con lo visto anteriormente sólo se pueden escribir algoritmos donde la ejecución de las instrucciones sea secuencial. Pero la programación estructurada permite el uso de decisiones y de iteraciones.

Estas instrucciones permiten que haya instrucciones que se pueden ejecutar o no según una condición (instrucciones alternativas), e incluso que se ejecuten repetidamente hasta que se cumpla una condición (instrucciones iterativas). En definitiva son instrucciones que permiten variar el flujo normal del programa.

expresiones lógicas

Todas las instrucciones de este apartado utilizan expresiones lógicas. Son expresiones que dan como resultado un valor lógico (verdadero o falso). Suelen ser siempre comparaciones entre datos. Por ejemplo x>8 da como resultado verdadero si x vale más que 8. Los operadores de relación (de comparación) que se pueden utilizar son:

>	Mayor que
<	Menor que
≥	Mayor o igual
≤	Menor o igual
≠	Distinto
=	Igual

También se pueden unir expresiones utilizando los operadores **Y** (en inglés **AND**), el operador **O** (en inglés **OR)** o el operador **NO** (en inglés **NOT**). Estos operadores permiten unir expresiones lógicas. Por ejemplo:

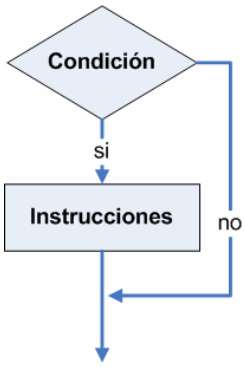
Expresión	Resultado verdadero si…
a>8 **Y** b<12	La variable ***a*** es mayor que 8 y (a la vez) la variable ***b*** es menor que 12
a>8 **Y** b<12	O la variable ***a*** es mayor que 8 o la variable ***b*** es menor que 12. Basta que se cumpla una de las dos expresiones.
a>30 **Y** a<45	La variable ***a*** está entre 31 y 44
a<30 **O** a>45	La variable ***a*** no está entre 30 y 45
NO a=45	La variable ***a*** no vale 45
a >8 **Y NO** b<7	La variable ***a*** es mayor que 8 y ***b*** es menor o igual que 7
a>45 Y a<30	Nunca es verdadero

instrucción de alternativa simple

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

```
si expresión_lógica entonces
  instrucciones
fin_si
```

Esto es equivalente al siguiente diagrama de flujo:



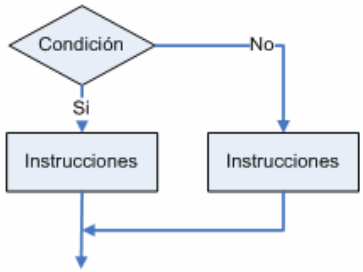
Las instrucciones sólo se ejecutarán si la expresión evaluada es verdadera.

instrucción de alternativa doble

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

```
si expresión_lógica entonces
  instrucciones //se ejecutan si la expresión es
verdadera
si_no
  instrucciones //se ejecutan si la expresión es falsa
fin_si
```

Sólo se ejecuta unas instrucciones dependiendo de si la expresión es verdadera. El diagrama de flujo equivalente es:



Hay que tener en cuenta que se puede meter una instrucción si dentro de otro si. A eso se le llama alternativas anidadas.

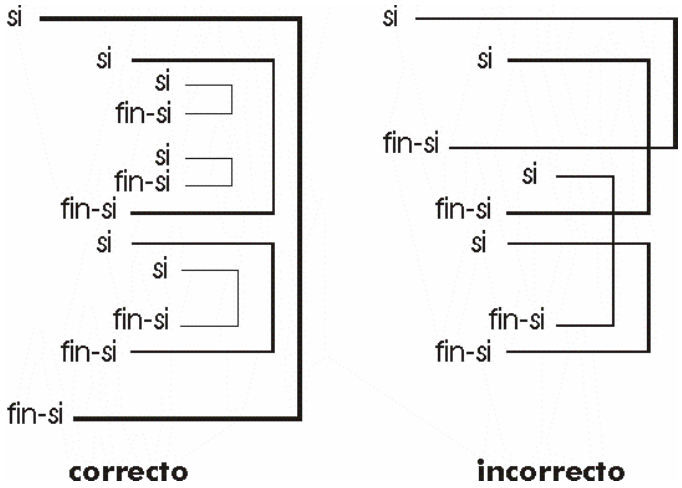
Ejemplo:

```
si a≥5 entonces
  escribe(“apto”)
  si a≥5 y a<7 entonces
       escribe(“nota:aprobado”)
```

```
  fin_si
  si a≥7 y a<9 entonces
       escribe("notable")
  si_no
       escribe("sobresaliente")
  fin_si
si_no
  escribe("suspenso")
fin_si
```

Al anidar estas instrucciones hay que tener en cuenta que hay que cerrar las instrucciones **si** interiores antes que las exteriores.

Eso es una regla básica de la programación estructurada:



alternativa compuesta

En muchas ocasiones se requieren condiciones que poseen más de una alternativa. En ese caso existe una instrucción evalúa una expresión y según los diferentes valores que tome se ejecutan unas u otras instrucciones.

Ejemplo:

```
según_sea expresión hacer
  valor1:
       instrucciones del valor1
  valor2:
       instrucciones del valor2
  ...
  si-no
       instrucciones del si_no
fin_según
```

Casi todos los lenguajes de programación poseen esta instrucción que suele ser un **case** (aunque C, C++, Java y C# usan **switch**). Se evalúa la expresión y si es igual que uno de los valores interiores se ejecutan las instrucciones de ese valor. Si no cumple ningún valor se ejecutan las instrucciones del ***si_no***.

Ejemplo:

```
programa pruebaSelMultiple
var
  x: entero
inicio
  escribe("Escribe un número del 1 al 4 y te diré si es par o impar")
  lee(x)
  según_sea x hacer
       1:
            escribe("impar")
       2:
            escribe("par")
       3:
            escribe("impar")
       4:
            escribe("par")
       si_no
            escribe("error eso no es un número de 1 a
4")
  fin_según
fin
```

El según sea se puede escribir también:

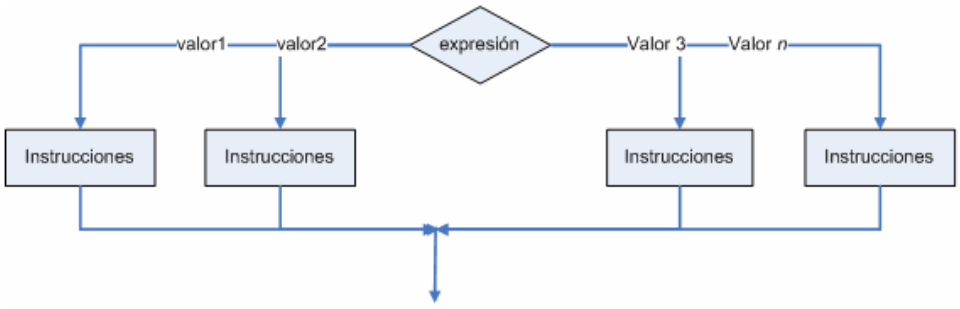
```
  según_sea x hacer
       1,3:
            escribe("impar")
       2,4:
            escribe("par")
       si_no
            escribe("error eso no es un número de 1 a
4")
  fin_según
```

Es decir el valor en realidad puede ser una lista de valores. Para indicar esa lista se pueden utilizar expresiones como:

1..3	De uno a tres (1,2 o 3)
>4	Mayor que 4
>5 Y <8	Mayor que 5 y menor que 8
7,9,11,12	7,9,11 y 12. Sólo esos valores (no el 10 o el 8 por ejemplo)

Sin embargo estas últimas expresiones no son válidas en todos los lenguajes (por ejemplo el C no las admite).

En el caso de los diagramas de flujo, el formato es:

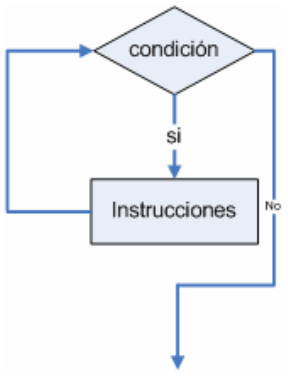


instrucciones iterativas de tipo *mientras*

El pseudocódigo admite instrucciones iterativas. Las fundamentales se crean con una instrucción llamada **mientras** (en inglés **while**). Su estructura es:

```
mientras condición hacer
  instrucciones
fin_mientras
```

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



Ejemplo (escribir números del 1 al 10):

```
x←1
mientras x<=10
  escribir(x)
  x←x+1
fin_mientras
```

Las instrucciones interiores a la palabra mientras podrían incluso no ejecutarse si la condición es falsa inicialmente.

instrucciones iterativas de tipo *repetir*

La diferencia con la anterior está en que se evalúa la condición al final (en lugar de al principio). Consiste en una serie de instrucciones que repiten continuamente su ejecución hasta que la condición sea verdadera (funciona por tanto al revés que el ***mientras*** ya que si la condición es falsa, las instrucciones se siguen ejecutando. Estructura

```
repetir
  instrucciones
hasta que condición
```

El diagrama de flujo equivalente es:

Ejemplo (escribir números del 1 al 10):

```
x←1
repetir
  escribir(x)
  x←x+1
hasta que x>10
```

instrucciones iterativas de tipo *hacer...mientras*

Se trata de una iteración que mezcla las dos anteriores. Ejecuta una serie de instrucciones mientras se cumpla una condición. Esta condición se evalúa tras la ejecución de las instrucciones. Es decir es un bucle de tipo ***mientras*** donde las instrucciones al menos se ejecutan una vez (se puede decir que es lo mismo que un bucle repetir salvo que la condición se evalúa al revés). Estructura:

```
hacer
  instrucciones
mientras condición
```

Este formato está presente en el lenguaje C y derivados (C++, Java, C#), mientras que el formato de ***repetir*** está presente en el lenguaje Pascal.

El diagrama de flujo equivalente es:

Ejemplo (escribir números del 1 al 10):

```
x←1
hacer
  escribir(x)
  x←x+1
mientras x≤10
```

instrucciones iterativas *para*

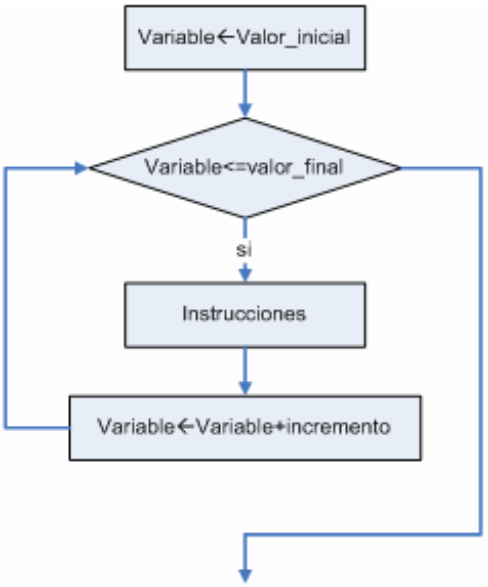
Existe otro tipo de estructura iterativa. En realidad no sería necesaria ya que lo que hace esta instrucción lo puede hacer una instrucción ***mientras***, pero facilita el uso de bucles con contador. Es decir son instrucciones que se repiten continuamente según los valores de un contador al que se le pone un valor de inicio, un valor final y el incremento que realiza en cada iteración (el incremento es opcional, si no se indica se entiende que es de uno). Estructura:

```
para variable←valorInicial hasta valorfinal hacer
  instrucciones
fin_para
```

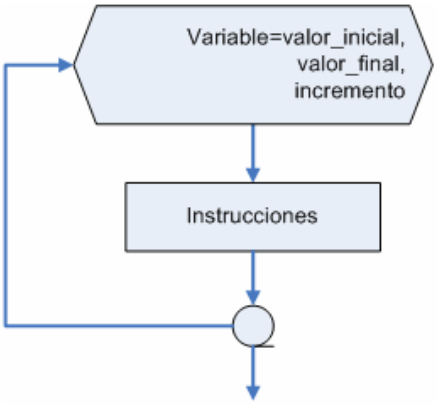
Si se usa el incremento sería:

```
para variable←vInicial hasta vFinal incremento valor
hacer
  instrucciones
fin_para
```

El diagrama de flujo equivalente a una estructura ***para*** sería:



También se puede utilizar este formato de diagrama:



Otros formatos de pseudocódigo utilizan la palabra **desde** en lugar de la palabra **para** (que es la traducción de **for**, nombre que se da en el original inglés a este tipo de instrucción).

estructuras iterativas anidadas

Al igual que ocurría con las instrucciones ***si***, también se puede insertar una estructura iterativa dentro de otra; pero en las mismas condiciones que la instrucción ***si.*** Cuando una estructura iterativa esta dentro de otra se debe cerrar la iteración interior antes de cerrar la exterior (véase la instrucción ***si***).

(2.4) otros tipos de diagramas

La programación estructurada hoy en día no se suele representar en ningún otro tipo de diagrama. Pero para expresar determinadas situaciones a tener en cuenta o cuando se utilizan otras programaciones se utilizan otros diagramas

(2.4.1) diagramas entidad/relación

Se utilizan muchísimo para representar datos que se almacenan en una base de datos. Son imprescindibles para el diseño de las bases de datos que es una de las aplicaciones más importantes de la informática.

(2.4.2) diagramas modulares

Representan los módulos que utiliza un programa y la relación que hay entre ellos. Suelen utilizarse conjuntamente con los diagramas descritos anteriormente.

(2.4.3) diagramas de estados

Representan los estados por los que pasa una aplicación, son muy importantes para planificar la aplicación.

(2.4.4) diagramas de secuencia

Indica tiempo transcurrido en la ejecución de la aplicación. Son muy útiles para diseñar aplicaciones donde el control del tiempo es crítico.

(2.4.5) diagramas de clases

Representan los elementos que componen una aplicación orientada a objetos y la relación que poseen éstos. Hoy en día el más popular es el diagrama de clases de la notación UML.

(2.4.6) UML

UML es el nombre que recibe el lenguaje de modelado universal. UML es un estándar en la construcción de diagramas para la creación de aplicaciones orientadas a objetos. Utiliza diversos tipos de esquemas para representar aspectos a tener en cuenta en la aplicación (organización de los datos, flujo de las instrucciones, estados de la aplicación, almacenamiento del código, etc.). Este tipo de diagramas serán explicados en temas posteriores.

(Unidad 3)
El lenguaje C

(3.1) historia del lenguaje C

(3.1.1) el nacimiento de C

Fue **Dennis Ritchie** quien en 1969 creo el lenguaje C a partir de las ideas diseñadas por otro lenguaje llamado **B** inventado por **Ken Thompson**, quien en los años 70 fue el encargado de desarrollar el lenguaje C.

Ritchie lo inventó para programar la computadora **PDP-11** que utilizaba el sistema **UNIX** (el propio Ritchie creo también Unix). De hecho La historia de C está muy ligada a la de UNIX, este sistema siempre ha incorporado compiladores para trabajar en C. El lenguaje C se diseñó como lenguaje pensado para programar sistemas operativos, debido a sus claras posibilidades para ello.

Pero su éxito inmediato hizo que miles de programadores en todo el mundo lo utilizaran para crear todo tipo de aplicaciones (hojas de cálculo, bases de datos,…), aunque siempre ha ten ido una clara relación con las aplicaciones de gestión de sistemas.

Debido a la proliferación de diferentes versiones de C, en 1983 el organismo **ANSI** empezó a producir un C estándar para normalizar su situación. En 1989 aparece el considerado como **C estándar** que fue aceptado por **ISO**, organismo internacional de estándares. Actualmente éste C es universalmente aceptado.

Actualmente se sigue utilizando enormemente para la creación de aplicaciones de sistemas (casi todas las distribuciones de Linux están principalmente creadas en C) y en educación se considera el lenguaje fundamental para aprender a programar.

(3.1.2) C y C++

Debido al crecimiento durante los años 80 de la programación orientada a objetos, en 1986 Bjarne Stroupstrup creo un lenguaje inspirado en Simula pero utilizando la sintaxis del lenguaje C.

C y C++ pues, comparten instrucciones casi idénticas. Pero la forma de programar es absolutamente diferente. Saber programar en C no implica saber programar en C++

(3.1.3) características del C

Se dice que el lenguaje C es un lenguaje de nivel **medio.** La razón de esta indicación está en que en C se pueden crear programas que manipulan la máquina casi como lo hace el lenguaje **Ensamblador**, pero utilizando una sintaxis que se asemeja más a los lenguajes de alto nivel. De los lenguajes de alto nivel toma las estructuras de control que permiten programar de forma estructurada.

Al tener características de los lenguajes de bajo nivel se puede tomar el control absoluto del ordenador. Además tiene atajos que gustan mucho a los programadores al tener una sintaxis menos restrictiva que lenguajes como Pascal por ejemplo), lo que le convierte en el lenguaje idóneo para crear cualquier tipo de aplicación.

Sus características básicas son:

- **Es un lenguaje estructurado y modular. Lo que facilita su compresión y escritura**
- **Es un lenguaje que incorpora manejo de estructuras de bajo nivel (punteros, bits), lo que le acerca a los lenguajes de segunda generación**
- **Permite utilizar estructuras de datos complejas (arrays, pilas, colas, textos,...)**
- **Es un lenguaje compilado**
- **Permite crear todo tipo de aplicaciones**

(3.2) entornos de programación

La programación de ordenadores requiere el uso de una serie de herramientas que faciliten el trabajo al programador. Esas herramientas (software) son:

- **Editor de código**. Programa utilizado para escribir el código. Normalmente basta un editor de texto (el código en los lenguajes de programación se guarda en formato de texto normal), pero es conveniente que incorpore:
  - **Coloreado inteligente de instrucciones**. Para conseguir ver mejor el código se colorean las palabras claves de una forma, los identificadores de otra, el texto de otra, los números de otra,....
  - **Corrección instantánea de errores**. Los mejores editores corrigen el código a medida que se va escribiendo.
  - **Ayuda en línea**. Para documentarse al instante sobre el uso del lenguaje y del propio editor.
- **Compilador**. El software que convierte el lenguaje en código máquina.
- **Lanzador**. Prueba la ejecución de la aplicación
- **Depurador**. Programa que se utiliza para realizar pruebas sobre el programa.

- **Organizador de ficheros**. Para administrar todos los ficheros manejados por las aplicaciones que hemos programado.

Normalmente todas esas herramientas se integran en un único software conocido como entorno de programación o **IDE** (***Integrate Development Environment***, Entorno de desarrollo integrado), que hace más cómodos todos los procesos relacionados con la programación.

(3.3) compilación de programas en C

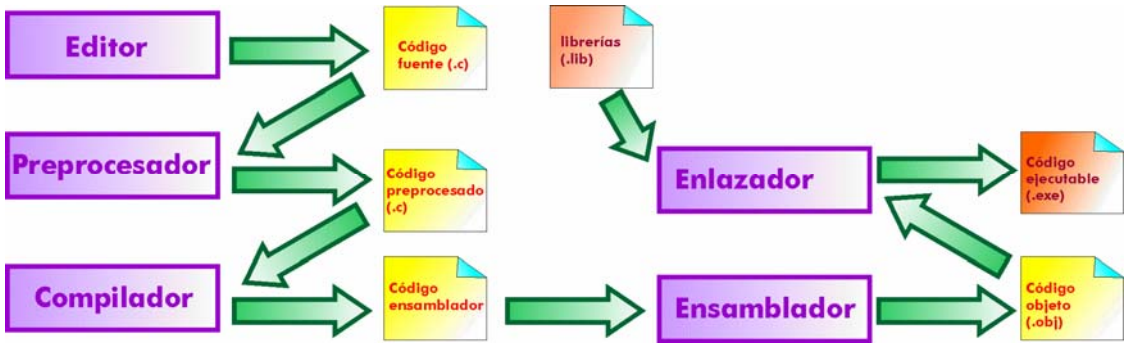


Ilustración 6, proceso de compilación de un programa C

Ya se ha explicado en unidades anteriores que la conversión del código escrito en un lenguaje de programación (**código fuente**) puede ser traducido a código máquina mediante un compilador o mediante un intérprete.

En el caso del lenguaje C se utiliza siempre un compilador (es un **lenguaje compilado**). El proceso completo de conversión del código fuente en código ejecutable sigue los siguientes pasos (ver Ilustración 6):

(1) **Edición**. El código se escribe en un editor de texto o en un editor de código preparado para esta acción. El archivo se suele guardar con extensión **.c**

(2) **Preprocesado**. Antes de compilar el código, el preprocesador lee las instrucciones de preprocesador y las convierte al código fuente equivalente.

(3) **Compilación.** El código fuente resultante en lenguaje C se compila mediante el software apropiado, obteniendo el código en ensamblador equivalente.

(4) **Ensamblado.** El código anterior se ensambla (utilizando software ensamblador, ya que el código que este paso requiere es código en ensamblador) para obtener código máquina. Éste código aún no es ejecutable pues necesita incluir el código de las librerías utilizadas. Éste es el código objeto (con extensión **obj**)

(5) **Enlazado.** El código objeto se une al código máquina de las librerías y módulos invocados por el código anterior. El resultado es un archivo ejecutable (extensión **.exe** en Windows)

El código enlazado es el que se prueba para comprobar si el programa funciona. La mayoría de entornos de programación tienen un **cargador** o **lanzador** que permite probar el programa sin abandonar el entorno. Desde el punto de vista del usuario de estos entornos, sólo hay dos pasos: **compilar** (conseguir el código ejecutable) y **ejecutar** (cargar el código máquina para que el ordenador lo ejecute)

(3.4) fundamentos de C

(3.4.1) estructura de un programa C

Un programa en C consta de una o más **funciones**, las cuales están compuestas de diversas **sentencias** o **instrucciones**. Una sentencia indica una acción a realizar por parte del programa. Una función no es más que (por ahora) un nombre con el que englobamos a las sentencias que posee a fin de poder invocarlas mediante dicho nombre. La idea es:

```
nombreDeFunción(parámetros) {
  sentencias
}
```

Los símbolos **{** y **}** indican el inicio y el final de la función. Esos símbolos permiten delimitar bloques en el código.

El nombre de la función puede ser invocado desde otras sentencias simplemente poniendo como sentencia el nombre de la función. Como a veces las funciones se almacenan en archivos externos, necesitamos incluir esos archivos en nuestro código mediante una sentencia especial **include**, que en realidad es una **directiva de preprocesador.** Una directiva de preprocesador es una instrucción para el compilador con el que trabajamos. El uso es:

```
#include <cabeceraDeArchivoExterno>
```

La directiva **include** permite indicar un archivo de cabecera en el que estará incluida la función que utilizamos. En el lenguaje C estándar los archivos de cabecera tienen extensión **h**. Los archivos de cabecera son los que permiten utilizar funciones externas (o librerías) en nuestro programa.

Una de las librerías más utilizadas en los programas, es la que permite leer y escribir en la consola del Sistema Operativo. En el caso de C esta librería está disponible en la cabecera **stdio.h**

el primer programa en C++

En todos los lenguajes de programación, el primer programa a realizar es el famoso ***Hola mundo,*** un programa que escribe este texto en pantalla. En C++ el código de este programa es:

```
#include <stdio.h>
int main()
{
      printf("Hola mundo");
      return 0;
}
```

En el programa anterior ocurre lo siguiente:

(1) La línea ***include*** permite utilizar funciones de la librería ***stdio.h*** que es la que permite leer y escribir datos por la consola del Sistema Operativo

(2) La función ***main*** es la función cuyas instrucciones se ejecutan en cuanto el programa inicia su proceso.

(3) La instrucción ***printf("Hola mundo")*** es la encargada de escribir el texto ***"Hola mundo"*** por pantalla

(4) La instrucción ***return 0*** finaliza el programa e indica (con el valor cero) que la finalización ha sido correcta.

(3.5) elementos de un programa en C

(3.5.1) sentencias

Los programas en C se basan en sentencias las cuales siempre se incluyen dentro de una función. En el caso de crear un programa ejecutable, esas sentencias están dentro de la función **main.** A está función le precede la palabra **int**.

Ahora bien al escribir sentencias hay que tener en cuenta las siguientes normas:

(1) Toda sentencia en C termina con el símbolo "punto y coma" (**;**)

(2) Los bloques de sentencia empiezan y terminan delimitados con el símbolo de llave ({ y }). Así **{** significa inicio y **}** significa fin

(3) En C hay distinción entre mayúsculas y minúsculas. No es lo mismo **main** que **MAIN**. Todas las palabras claves de C están en minúsculas. Los nombres que pongamos nosotros también conviene ponerles en minúsculas ya que el código es mucho más legible así.

(3.5.2) comentarios

Se trata de texto que es ignorado por el compilador al traducir el código. Esas líneas se utilizan para documentar el programa.

Esta labor de documentación es fundamental. De otro modo el código se convierte en ilegible incluso para el programador que lo diseñó. Tan importante como saber escribir sentencias es utilizar los comentarios. Todavía es más importante cuando el código va a ser tratado por otras personas, de otro modo una persona que modifique el código de otra lo tendría muy complicado. En C los comentarios se delimitan entre los símbolos **/*** y ***/**

```
/* Esto es un comentario
el compilador hará caso omiso de este texto*/
```

Como se observa en el ejemplo, el comentario puede ocupar más de una línea.

(3.5.3) palabras reservadas

Se llaman así a palabras que en C tienen un significado concreto para los compiladores. No se pueden por tanto usar esas palabras para poner nombre a variables o a funciones. La lista de palabras reservadas del C es ésta:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

A estas palabras a veces los compiladores añaden una lista propia. Como C distingue entre mayúsculas y minúsculas, el texto **GoTo** no es una palabra reservada.

(3.5.4) identificadores

Son los nombres que damos a las variables y a las funciones de C. Lógicamente no pueden coincidir con las palabras reservadas. Además puesto que C distingue entre las mayúsculas y las minúsculas, hay que tener cuidado de usar siempre las minúsculas y mayúsculas de la misma forma (es decir, ***nombre***, ***Nombre*** y ***NOMBRE*** son tres identificadores distintos).

El límite de tamaño de un identificador es de 32 caracteres (aunque algunos compiladores permiten más tamaño). Además hay que tener en cuenta que los identificadores deben de cumplir estas reglas:

- Deben comenzar por una letra o por el signo de subrayado (aunque comenzar por subrayado se suele reservar para identificadores de funciones especiales del sistema).
- Sólo se admiten letras del abecedario inglés, no se admite ni la ñ ni la tilde ni la diéresis, números y el carácter de subrayado

(3.5.5) líneas de preprocesador

Las sentencias que comienzan con el símbolo "#" y que no finalizan con el punto y coma son líneas de preprocesador (o directivas de compilación). Son indicaciones para el preprocesador, pero que en el lenguaje C son sumamente importantes. La más utilizada es

#include

Que permite añadir utilizar funciones externas en nuestro código, para lo que se indica el nombre del archivo de cabecera que incluye a la función (o funciones) que queremos utilizar.

(3.6) variables

(3.6.1) introducción

Las variables sirven para identificar un determinado valor que se utiliza en el programa. En C es importante el hecho de tener en cuenta que una variable se almacena en la memoria interna del ordenador (normalmente en la **memoria RAM**) y por lo tanto ocupará una determinada posición en esa memoria. Es decir, en realidad identifican una serie de bytes en memoria en los que se almacenará un valor que necesita el programa.

Es decir si ***saldo*** es un identificador que se refiere a una variable numérica que en este instante vale 8; la realidad interna es que *saldo* realmente es una dirección a una posición de la memoria en la que ahora se encuentra el número 8.

(3.6.2) declaración de variables

En C hay que declarar las variables antes de poder usarlas. Al declarar lo que ocurre es que se reserva en memoria el espacio necesario para almacenar el contenido de la variable. No se puede utilizar una variable sin declarar. Para declarar una variable se usa esta sintaxis:

```
tipo identificador;
```

Por ejemplo:

```
int x;
```

Se declarara *x* como variable que podrá almacenar valores enteros.

En C se puede declarar una variable en cualquier parte del código, basta con declararla antes de utilizarla por primera vez. Pero es muy buena práctica hacer la declaración al principio del bloque o función en la que se utilizará la variable. Esto facilita la comprensión del código.

También es buena práctica poner un pequeño comentario a cada variable para indicar para qué sirve. A veces basta con poner un identificador claro a la variable. Identificadores como ***a, b*** o ***c*** por ejemplo, no aclaran la utilidad de la variable; no indican nada. Identificadores como ***saldo***, ***gastos***, ***nota***,... son mucho más significativos.

(3.6.3) tipos de datos

Al declarar variables se necesita indicar cuál es el tipo de datos de las variables los tipos básicos permitidos por el lenguaje C son:

tipo de datos	rango de valores posibles	tamaño en bytes
char	0 a 255 (o caracteres simples del código ASCII)	1
int	-32768 a 32767 (algunos compiladores consideran este otro rango: -2.147.483.648 a -2.147.483.647	2 (algunos 4)
float	3.4E-38 a 3.3E+38	4
double	1.7E-308 a 1.7E+308	8
void	sin valor	0

Hay que tener en cuenta que esos rangos son los clásicos, pero en la práctica los rangos (sobre todo el de los enteros) depende del computador, procesador o compilador empleado.

tipos enteros

Los tipos **char** e **int** sirven para almacenar enteros y también valen para almacenar caracteres. Normalmente los números se almacenan en el tipo **int** y los caracteres en el tipo **char;** pero la realidad es que cualquier carácter puede ser representado como número (ese número indica el código en la tabla **ASCII**). Así el carácter ***'A'*** y el número ***65*** significan exactamente lo mismo en lenguaje C.

tipos decimales

En C los números decimales se representan con los tipos **float** y **double.** La diferencia no solo está en que en el **double** quepan números más altos, sino en la precisión.

Ambos tipos son de **coma flotante**. En este estilo de almacenar números decimales, la precisión es limitada. Cuántos más bits se destinen a la precisión, más preciso será el número. Por eso es más conveniente usar el tipo **double** aunque ocupe más memoria.

tipos lógicos

En C estándar el uso de valores lógicos se hace mediante los tipos enteros. Cualquier valor distinto de cero (normalmente se usa el uno) significa verdadero y el valor cero significa falso. Es decir en el código:

```
int x=1;
```

Se puede entender que *x* vale 1 o que *x* es verdadera. Ejemplo:

```
int x=(18>6);
printf("%d",x); /*Escribe 1, es decir verdadero */
```

En C++ se añadió el tipo **bool** (booleano o lógico) para representar valores lógicos, las variables **bool** pueden asociarse a los números cero y uno, o mejor a las palabras **true** (verdadero) y **false** (falso)**.** Ejemplo:

```
bool b=true; //Eso es el equivalente C++ del C int b=1;
```

En C para conseguir el mismo efecto se utilizan trucos mediante la directiva **#define** (cuyo uso se explica más adelante)

modificadores de tipos

A los tipos anteriores se les puede añadir una serie de modificadores para que esos tipos varíen su funcionamiento. Esos modificadores se colocan por delante del tipo en cuestión. Son:

- **signed**. Permite que el tipo modificado admita números negativos. En la práctica se utiliza sólo para el tipo char, que de esta forma puede tomar un rango de valores de -128 a 127. En los demás no se usan ya que todos admiten números negativos
- **unsigned**. Contrario al anterior, hace que el tipo al que se refiere use el rango de negativos para incrementar los positivos. Por ejemplo el unsigned int tomaría el rango 0 a 65535 en lugar de -32768 a 32767
- **long**. Aumenta el rango del tipo al que se refiere.
- **short**. El contrario del anterior. La realidad es que no se utiliza.

Las combinaciones que más se suelen utilizar son:

tipo de datos	rango de valores posibles	tamaño en bytes
signed char	-128 a 127	1
unsigned int	0 a 65335	2
long int (o **long** a secas)	-2147483648 a 2147483647	4

long long	-9.223.372.036.854.775.809 a 9.223.372.036.854.775.808 (casi ningún compilador lo utiliza, casi todos usan el mismo que el **long**)	8
long double	3.37E-4932 a 3,37E+4932	10

Una vez más hay que recordar que los rangos y tamaños depende del procesador y compilador. Por ejemplo, el tipo ***long double*** no suele incorporarlo casi ningún compilador (aunque sí la mayoría de los modernos).

Aún así conviene utilizar esas combinaciones por un tema de mayor claridad en el código.

(3.6.4) asignación de valores

Además de declarar una variable. A las variables se las pueden asignar valores. El operador de asignación en C es el signo "=". Ejemplo:

```
x=3;
```

Si se utiliza una variable antes de haberla asignado un valor, ocurre un error. Pero es un error que no es detectado por un compilador. Por ejemplo el código:

```
#include <stdio.h>
int main(){
    int a;
    printf("%d",a);
  return 0; }
```

Este código no produce error, pero como a la variable *a* no se le ha asignado ningún valor, el resultado del ***printf*** es un número sin sentido. Ese número representa el contenido de la zona de la memoria asociada a la variable ***a***, pero el contenido de esa zona no le hemos preparado, por ello el resultado será absolutamente aleatorio.

Hay que destacar que se puede declarar e inicializar valores a la vez:

```
int a=5;
```

E incluso inicializar y declarar varias variables a la vez:

```
int a=8, b=7, c;
```

También es válida esta instrucción:

```
int a=8, b=a, c;
```

La asignación *b=a* funciona si la variable *a* ya ha sido declarada (como es el caso)

(3.6.5) literales

Cuando una variable se asigna a valores literales (17, 2.3,etc.) hay que tener en cuenta lo siguiente:

- Los números se escriben tal cual (17, 34, 39)
- El separador de decimales es el punto (18.4 se interpreta como 18 *coma* 4)
- Si un número entero se escribe anteponiendo un cero, se entiende que está en notación octal. Si el número es 010, C entiende que es el 8 decimal
- Si un número entero se escribe anteponiendo el texto 0x (***cero equis***), se entiende que es un número hexadecimal. El número 0x10 significa 16.
- En los números decimales se admite usar notación científica: 1.23e+23 (eso significa $(1{,}23 \cdot 10^{23})$
- Los caracteres simples van encerrados entre comillas simples, 'a'
- Los textos (***strings***) van encerrados entre comillas dobles "Hola"
- Los enteros se pueden almacenar como caracteres 'A' o como enteros cortos. Es más 'A' es lo mismo que 65. Eso vale tanto para los tipos **char** como para los **int**.

secuencias de escape

En el caso de los caracteres, hay que tener en cuenta que hay una serie de caracteres que son especiales. Por ejemplo como almacenamos en una variable *char* el símbolo de la comilla simple, si la propia comilla simple sirve para delimitar, es decir:

```
char a='';/*Error*/
```

Eso no se puede hacer ya que el compilador entiende que hay una mala delimitación de caracteres. Para resolver este problema y otros se usan los caracteres de escape, que representan caracteres especiales.

Todos comienzan con el signo "\" (***backslash***) seguido de una letra minúscula, son:

código	significado
\a	Alarma (*beep* del ordenador)
\b	Retroceso
\n	Nueva línea
\r	Retorno de carro
\t	Tabulador
\'	Comilla simple
\"	Comilla doble
\\	Barra inclinada invertida o *backslash*

(3.6.6) ámbito de las variables

Toda variable tiene un **ámbito**. Esto es la parte del código en la que una variable se puede utilizar. De hecho las variables tienen un ciclo de vida:

(1) En la declaración se reserva el espacio necesario para que se puedan comenzar a utilizar (digamos que se avisa de su futura existencia)

(2) Se la asigna su primer valor (la variable ***nace***)

(3) Se la utiliza en diversas sentencias (no se puede utilizar su contenido sin haberla asignado ese primer valor.

(4) Cuando finaliza el bloque en el que fue declarada, la variable ***muere***. Es decir, se libera el espacio que ocupa esa variable en memoria. No se la podrá volver a utilizar.

variables locales

Son variables que se crean dentro de un bloque (se entiende por bloque, el código que está entre { y }). Con el fin del bloque la variable es eliminada. La mayoría son locales a una determinada función, es decir sólo se pueden utilizar dentro de esa función. Ejemplo:

```
void func1(){
  int x;
  x=5;
}
void func2(){
  int x;
  x=300;
}
```

Aunque en ambas funciones se usa *x* como nombre de una variable local. En realidad son dos variables distintas, pero con el mismo nombre. Y no podríamos usar *x* dentro de la función *func2* porque estamos fuera de su ámbito.

Otro ejemplo:

```
void func(){
  int a;
a=13;
  {
       int b;
       b=8;
  }/*Aquí muere b*/
  a=b; /*Error! b está muerta*/
}/*Aquí muere a*/
```

En la línea *a=b* ocurre un error de tipo "*Variable no declarada*", el compilador ya no reconoce a la variable *b* porque estamos fuera de su ámbito.

variables globales

Son variables que se pueden utilizar en cualquier parte del código. Para que una variable sea global basta con declararla fuera de cualquier bloque. Normalmente se declaran antes de que aparezca la primera función:

```
#include <stdio.h>
int a=3; //La variable "a" es global
int main(){
    printf("%d",a);
  return 0;
}
```

En C no se permite declarar en el mismo bloque dos variables con el mismo nombre. Pero sí es posible tener dos variables con el mismo nombre si están en bloques distintos. Esto plantea un problema, ya que cuando se utiliza la variable surge una duda: ¿qué variable utilizará el programa, la más local o la más global? La respuesta es que siempre se toma la variable declarada más localmente. Ejemplo:

```
#include <stdio.h>
int a=3;
int main(){
  int a=5;
  {
     int a=8;
```

```
        printf("%d",a); //escribe 8. No hay error
    }
    return 0;
    }
```

En el código anterior se han declarado tres variables con el mismo nombre (*a*). Cuando se utiliza la instrucción ***printf*** para escribir el valor de a, la primera vez escribe 8, la segunda vez escribe 5 (ya que ese ***printf*** está fuera del bloque más interior).

Es imposible acceder a las variables globales si disponemos de variables locales con el mismo nombre. Por eso no es buena práctica repetir el nombre de las variables.

(3.6.7) conversión de tipos

En numerosos lenguajes no se pueden asignar valores entre variables que sean de distinto tipo. Esto significaría que no podemos asignar a una variable **char** valores de una variable **int.**

En C no existe esta comprobación. Lo que significa que los valores se convierten automáticamente. Pero eso también significa que puede haber problemas indetectables, por ejemplo este programa:

```
#include <stdio.h>
int main(){
    char a;
    int b=300;
    a=b;
    printf("%d %d",a,b);
}
/* Escribe el contenido de a y de b. Escribe 44 y 300 */
```

En ese programa el contenido de *a* debería ser 300, pero como 300 sobrepasa el rango de las variables *char*, el resultado es 44. Es decir, no tiene sentido, esa salida está provocada por el hecho de perder ciertos bits en esa asignación.

En la conversión de *double* a *float* lo que ocurre normalmente es un redondeo de los valores para ajustarles a la precisión de los *float*.

(3.6.8) modificadores de acceso

Los modificadores son palabras que se colocan delante del tipo de datos en la declaración de las variables para variar su funcionamiento (al estilo de *unsigned*, *short* o *long*)

modificador *extern*

Se utiliza al declarar variables globales e indica que la variable global declarada, en realidad se inicializa y declara en otro archivo. Ejemplo

Archivo 1	Archivo 2

`int x,y;`
`int main(){`
`  x=12;`
`  y=6;`
`}`
`void funcion1(void){`
`  x=7;`
`}`
	`extern int x,y;`
	`void func2(void){`
	`  x=2*y;`
	`}`

El segundo archivo utiliza las variables declaradas en el primero

modificador *auto*

En realidad las variables toman por defecto este valor (luego no hace falta utilizarle). Significa que las variables se eliminan al final del bloque en el que fueron creadas.

modificador *static*

Se trata de variables que no se eliminan cuando el bloque en el que fueron creadas finaliza. Así que si ese bloque (normalmente una función), vuelve a invocarse desde el código, la variable mantendrá el último valor anterior.

Si se utiliza este modificador con variables globales, indica que esas variables sólo pueden utilizarse desde el archivo en el que fueron creadas.

modificador *register*

Todos los ordenadores posee una serie de memorias de pequeño tamaño en el propio procesador llamadas **registros**. Estas memorias son mucho más rápidas pero con capacidad para almacenar muy pocos datos.

Este modificador solicita que una variable sea almacenada en esos registros para acelerar el acceso a la misma. Se utiliza en variables *char* o *int*a las que se va a acceder muy frecuentemente en el programa (por ejemplo las variables contadoras en los bucles). Sólo vale para variables locales.

```
register int cont;
for (cont=1;cont<=300000;cont++){
...
```

modificador *const*

Las variables declaradas con la palabra **const** delante del tipo de datos, indican que son sólo de lectura. Es decir, constantes. Las constantes no pueden cambiar de valor, el valor que se asigne en la declaración será el que permanezca (es obligatorio asignar un valor en la declaración). Ejemplo:

```
const float pi=3.141592;
```

modificador *volatile*

Se usa para variables que son modificadas externamente al programa (por ejemplo una variable que almacene el reloj del sistema).

(3.7) entrada y salida por consola

Aunque este tema será tratado con detalle más adelante. Es conveniente conocer al menos las funciones *printf* y *scanf* que permiten entradas y salidas de datos de forma muy interesante.

función *printf*

La función *printf* permite escribir datos en la consola de salida (en la pantalla). Si lo que se desea sacar es un texto literal se utiliza de esta manera:

```
printf("Hola");
```

Lógicamente a veces se necesitan sacar el contenido de las variables en la salida. Para ello dentro del texto que se desea mostrar se utilizan unas indicaciones de formato, las cuales se indican con el signo"%" seguido de una letra minúscula. Por ejemplo:

```
int edad=18;
...
printf("Tengo %i años",edad);
/*escribe tengo 18 años*/
```

El código **%i** sirve para recoger el valor de la variable que se indique entendiendo que esa variable tiene valores enteros. Esa variable va separada del texto por una coma. Si se usan más variables o valores, éstos se separan con comas entre sí.

Ejemplo:

```
int edad=18, edadFutura=19;
printf("Tengo %i años, el año que viene
%i",edad,edadFutura);
/*Escribe:
 Tengo 18 años, el año que viene 19
*/
```

Además del código %d, hay otros que se pueden utilizar:

código	significado
%i	Entero
%d	Entero

%c	Un carácter
%f	Punto flotante (para *double* o *float*)
%e	Escribe en notación científica
%g	Usa %e o %f, el más corto de los dos
%o	Escribe números enteros en notación octal
%x	Escribe números enteros en notación hexadecimal
%s	Cadena de caracteres
%u	Enteros sin signo
%li o %ld	Enteros largos (long)
%p	Escribe un puntero
%%	Escribe el signo %

Ejemplo:

```
char c='H';
int n=28;
printf("Me gusta la letra %c y el número %i,
%s",c,n,"adiós");
/* Escribe: Me gusta la letra H y el número 28, adiós */
```

Mediante *printf*, también se pueden especificar anchuras para el texto. Es una de sus características más potentes. Para ello se coloca un número entre el signo % y el código numérico (*i, d, f*,...). Ese número indica anchura mínima. Por ejemplo:

```
int a=127, b=8, c=76;
printf("%4d\n",a);
printf("%4d\n",b);
printf("%4d\n",c);
/* Sale:
 127
   8
  76
*/
```

En el ejemplo se usan 4 caracteres para mostrar los números. Los números quedan alineados a la derecha en ese espacio, si se desean alinear a la izquierda, entonces el número se pone en negativo (**%-4d**).

Si se coloca el número delante del número, el espacio sobrante se rellena con ceros en lugar de con espacios.

Ejemplo:

```
int a=127, b=8, c=76;
printf("%04d\n",a);
printf("%04d\n",b);
printf("%04d\n",c);
/* Sale:
0127
0008
0076
*/
```

También se pueden especificar los decimales requeridos (para los códigos decimales, como *f* por ejemplo). De esta forma el código **%10.4f** indica que se utiliza un tamaño mínimo de 10 caracteres de los que 4 se dejan para los decimales.

(3.7.2) función *scanf*

Esta función es similar a la anterior. sólo que ésta sirve para leer datos. Posee al menos dos parámetros, el primero es una cadena que indica el formato de la lectura, el segundo (y siguientes) son las zonas de memoria en las que se almacenará el resultado.

Por ejemplo, para leer un número entero por el teclado y guardarlo en una variable de tipo *int* llamada *a,* se haría:

```
scanf("%d",&a);
```

El símbolo &a, sirve para tomar la dirección de memoria de la variable *a*. Esto significa que *a* contendrá el valor leído por el teclado.

Este otro ejemplo:

```
scanf("%d %f",&a,&b);
```

Lee dos números por teclado. El usuario tendrá que colocar un espacio entre cada número. El primero se almacenará en *a* y el segundo (decimal) en *b.*

Otro ejemplo:

```
scanf("%d,%d",&a,&b);
```

En la lectura de textos se pueden especificar formatos más avanzados. Pero eso lo veremos en el tema 5.

(3.8) operadores

Se trata de uno de los puntos fuertes de este lenguaje que permite especificar expresiones muy complejas gracias a estos operadores.

(3.8.1) operadores aritméticos

Permiten realizar cálculos matemáticos. Son:

operador	significado
+	Suma
-	Resta
*	Producto
/	División
%	resto (módulo)
++	Incremento
--	Decremento

La suma, resta, multiplicación y división son las operaciones normales. Sólo hay que tener cuidado en que el resultado de aplicar estas operaciones puede ser un número que tenga un tipo diferente de la variable en la que se pretende almacenar el resultado.

El signo "-" también sirve para cambiar de signo (*-a* es el resultado de multiplicar a la variable *a* por -1*)*.

El incremento (++), sirve para añadir uno a la variable a la que se aplica. Es decir *x*++ es lo mismo que *x=x+1*. El decremento funciona igual pero restando uno. Se puede utilizar por delante (**preincremento**) o por detrás (**postincremento**) de la variable a la que se aplica (*x*++ ó ++*x*). Esto último tiene connotaciones. Por ejemplo:

```
int x1=9,x2=9;
int y1,y2;
y1=x1++;
y2=++x2;
printf("%i\n",x1); /*Escribe 10*/
printf("%i\n",x2); /*Escribe 10*/
printf("%i\n",y1); /*¡¡¡Escribe 9!!! */
printf("%i\n",y2); /*Escribe 10*/
```

La razón de que *y1* valga 9, está en que la expresión *y1=x1*++, funciona de esta forma:

```
y1=x1;
x1=x1+1;
```

Mientras que en y2=++x2, el funcionamiento es:

```
x2=x2+1;
y2=x2;
```

Es decir en x2++ primero se asigno y luego se incremento. Si el incremento va antes se realiza al contrario.

(3.8.2) operadores relacionales

Son operadores que sirven para realizar comparaciones. El resultado de estos operadores es verdadero o falso (uno o cero). Los operadores son:

operador	significado
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual que
!=	Distinto de

(3.8.3) operadores lógicos

Permiten agrupar **expresiones lógicas**. Las expresiones lógicas son todas aquellas expresiones que obtienen como resultado verdadero o falso. Estos operadores unen estas expresiones devolviendo también verdadero o falso. Son:

operador	significado
&&	Y (AND)
||	O (OR)
¡	NO (NOT)

Por ejemplo: *(18>6) && (20<30)* devuelve verdadero (1) ya que la primera expresión (18>6) es verdadera y la segunda (20<30) también. El operador Y (&&) devuelve verdadero cuando las dos expresiones son verdaderas. El operador O (||) devuelve verdadero cuando cualquiera de las dos es verdadera.

Finalmente el operador NO (!) invierte la lógica de la expresión que le sigue; si la expresión siguiente es verdadera devuelve falso y viceversa. Por ejemplo !(18>15) devuelve falso (0).

(3.8.4) operadores de bits

Permiten realizar operaciones sobre los bits del número, o números, sobre los que operan. Es decir si el número es un *char* y vale 17, 17 en binario es 00010001. Estos operadores operan sobre ese código binario. En este manual simplemente se indican estos operadores:

operador	significado
&	AND de bits
|	OR de bits
~	NOT de bits
^	XOR de bits
>>	Desplazamiento derecho de los bits
<<	Desplazamiento izquierdo de bits

(3.8.5) operador de asignación

Ya se ha comentado que el signo "=" sirve para asignar valores. Se entiende que es un operador debido a la complejidad de expresiones de C. Por ejemplo:

```
int x=5,y=6,z=7;
x=(z=y++)*8;
printf("%d",x); //Escribe 48
```

En C existen estas formas abreviadas de asignación. Esto sirve como abreviaturas para escribir código. Así la expresión:

```
x=x+10;
```

Se puede escribir como:

```
x+=10;
```

Se permiten estas abreviaturas:

operador	significado
+=	Suma y asigna
-=	Resta y asigna
*=	Multiplica y asigna
/=	Divide y asigna
%=	Calcula el resto y asigna

Además también se permiten abreviar las expresiones de bit: &=, |=, ^=, >>=, <<=

(3.8.6) operador ?

Permite escribir expresiones condicionales. Su uso es el siguiente:

```
Expresión_a_valorar?Si_verdadera:Si_falsa
```

Ejemplo:

```
x=(y>5?'A':'B');
```

Significa que si la variable *y* es mayor de 5, entonces a *x* se le asigna el carácter 'A', sino se le asignará el carácter 'B'.

Otro ejemplo:

```
printf("%s",nota>=5?"Aprobado":"Suspenso");
```

En cualquier caso hay que utilizar este operador con cautela. Su dominio exige mucha práctica.

(3.8.7) operadores de puntero & y *

Aunque ya se le explicará más adelante con detalle, conviene conocerle un poco. El operador & sirve para obtener la dirección de memoria de una determinada variable. No tiene sentido querer obtener esa dirección salvo para utilizar punteros o para utilizar esa dirección para almacenar valores (como en el caso de la función *scanf*).

El operador * también se utiliza con punteros. Sobre una variable de puntero, permite obtener el contenido de la dirección a la que apunta dicho puntero.

(3.8.8) operador sizeof

Este operador sirve para devolver el tamaño en bytes que ocupa en memoria una determinada variable. Por ejemplo:

```
int x=18;
printf("%i",sizeof x); /*Escribe 2 (o 4 en algunos compiladores)*/
```

Devuelve 2 o 4, dependiendo del gasto en bytes que hacen los enteros en la máquina y compilador en que nos encontremos.

(3.8.9) operador *coma*

La coma "," sirve para poder realizar más de una instrucción en la misma línea. Por ejemplo:

```
y=(x=3,x++);
```

La coma siempre ejecuta la instrucción que está más a la izquierda. Con lo que en la línea anterior primero la *x* se pone a 3; y luego se incrementa la *x* tras haber asignado su valor a la variable *y* (ya que es un postincremento).

(3.8.10) operadores especiales

Todos ellos se verán con más detalle en otros temas. Son:

- **El operador "." (punto).** Este operador permite hacer referencia a campos de un registro. Un registro es una estructura de datos avanzada que permite agrupar datos de diferente tipo.
- **El operador flecha "->"** que permite acceder a un campo de registro cuando es un puntero el que señala a dicho registro.
- **Los corchetes "[]",** que sirven para acceder a un elemento de un array. Un array es una estructura de datos que agrupa datos del mismo tipo
- **Molde o cast**. Que se explica más adelante y que sirve para conversión de tipos.

(3.8.11) orden de los operadores

En expresiones como:

```
x=6+9/3;
```

Podría haber una duda. ¿Qué vale *x*? Valdría 5 si primero se ejecuta la suma y 9 si primero se ejecuta la división. La realidad es que valdría 9 porque la división tiene preferencia sobre la suma. Es decir hay operadores con mayor y menor preferencia.

Lógicamente el orden de ejecución de los operadores se puede modificar con paréntesis.

Por ejemplo:

```
x=(6+9/3;
y=(x*3)/((z*2)/8);
```

Como se observa en el ejemplo los paréntesis se pueden anidar.

Sin paréntesis el orden de precedencia de los operadores en orden de mayor a menor precedencia, forma 16 niveles. Los operadores que estén en el mismo nivel significa que tienen la misma precedencia. En ese caso se ejecutan primero

los operadores que estén más a la izquierda. El orden es (de mayor a menor precedencia):

(1) ( ) [ ] . ->

(2) Lo forman los siguientes:

- NOT de expresiones lógicas: !
- NOT de bits: ~
- Operadores de punteros: * &
- Cambio de signo: -
- sizeof
- (*cast*)
- Decremento e incremento: ++ --

(3) Aritméticos prioritarios: / * %

(4) Aritméticos no prioritarios (suma y resta): + -

(5) Desplazamientos: >> <<

(6) Relacionales sin igualdad: > < >= <=

(7) Relacionales de igualdad: == !=

(8) &

(9) ^

(10) |

(11) &&

(12) ||

(13) ?:

(14) = *= /* += -= %= >>= <<= |= &= ^=

(15) , (coma)

(3.9) expresiones y conversión de tipos

(3.9.1) introducción

Operadores, variables, constantes y funciones son los elementos que permiten construir expresiones. Una expresión es pues un código en C que obtiene un determinado valor (del tipo que sea).

(3.9.2) conversión

Cuando una expresión utiliza valores de diferentes tipos, C convierte la expresión al mismo tipo. La cuestión es qué criterio sigue para esa conversión. El criterio, en general, es que C toma siempre el tipo con rango más grande. En ese sentido si hay un dato **long double,** toda la expresión se convierte a long double, ya que ese es el tipo más grande. Si no aparece un long double entonces el tipo más grande en el que quepan los datos.

El orden de tamaños es:

(1) long double

(2) double

(3) float

(4) unsigned long

(5) long

(6) int

Es decir si se suma un *int* y un *float* el resultado será *float*.

En una expresión como:

```
int x=9.5*2;
```

El valor 9.5 es *double* mientras que el valor 2 es *int* por lo que el resultado (19) será *double.* Pero como la variable *x* es entera, el valor deberá ser convertido a entero finalmente.

(3.9.3) operador de molde o *cast*

A veces se necesita hacer conversiones explícitas de tipos. Para eso está el operador *cast.* Este operador sirve para convertir datos. Su uso es el siguiente, se pone el tipo deseado entre paréntesis y a la derecha el valor a convertir. Por ejemplo:

```
x=(int) 8.3;
```

x valdrá 8 independientemente del tipo que tenga, ya que al convertir datos se pierden decimales.

Este ejemplo (comprar con el utilizado en el apartado anterior):

```
int x=(int)9.5*2;
```

Hace que *x* valga 18, ya que al convertir a entero el 9.5 se pierden los decimales.

(Unidad 4) Programación estructurada en C

(4.1) expresiones lógicas

Hasta este momento nuestros programas en C apenas pueden realizar programas que simulen, como mucho, una calculadora. Lógicamente necesitamos poder elegir qué cosas se ejecutan según unas determinada circunstancias.

Todas las sentencias de control de flujo se basan en evaluar una expresión lógica. Una expresión lógica es cualquier expresión que pueda ser evaluada con verdadero o falso. En C (o C++) se considera verdadera cualquier expresión distinta de 0 (en especial el uno, valor **verdadero**) y falsa el cero (**falso**).

(4.2) sentencia *if*

(4.2.1) sentencia condicional simple

Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta una serie de sentencias en caso de que la expresión lógica sea verdadera. Su sintaxis es:

```
if(expresión lógica) {
  sentencias
}
```

Si sólo se va a ejecutar una sentencia, no hace falta usar las llaves:

```
if(expresión lógica) sentencia;
```

Ejemplo:

```
if(nota>=5){
   printf("Aprobado");
   aprobados++;
}
```

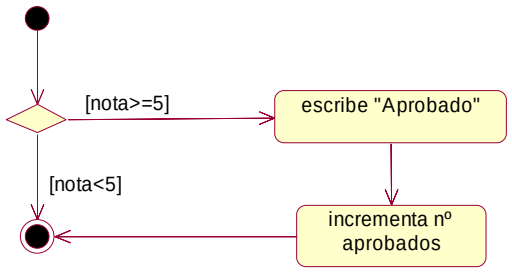


Ilustración 7, Diagrama de actividad del ejemplo anterior

(4.2.2) sentencia condicional compuesta

Es igual que la anterior, sólo que se añade un apartado **else** que contiene instrucciones que se ejecutarán si la expresión evaluada por el **if** es falsa. Sintaxis:

```
if(expresión lógica){
   sentencias
}
else {
   sentencias
}
```

Las llaves son necesarias sólo si se ejecuta más de una sentencia. Ejemplo:

```
if(nota>=5){
   printf("Aprobado");
   aprobados++;
}
else {
   printf("Suspensos");
   suspensos++;
}
```

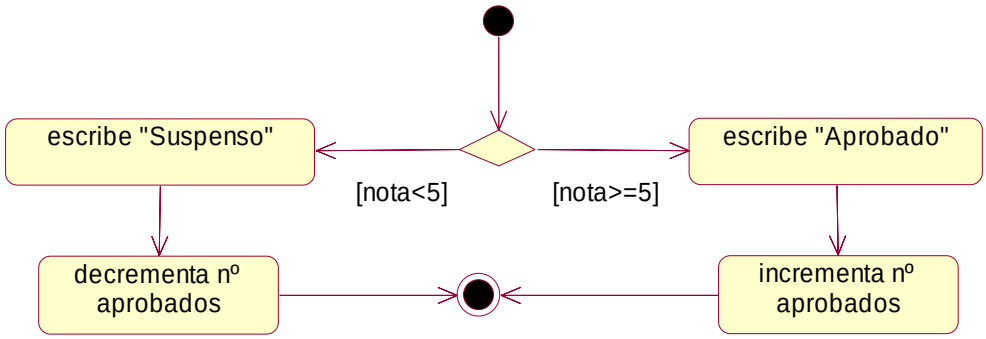


Ilustración 8, diagrama UML de actividad del ejemplo anterior

(4.2.3) anidación

Dentro de una sentencia **if** se puede colocar otra sentencia **if**. A esto se le llama **anidación** y permite crear programas donde se valoren expresiones complejas.

Por ejemplo en un programa donde se realice una determinada operación dependiendo de los valores de una variable, el código podría quedar:

```
if (x==1) {
  sentencias
  …
}
else {
  if(x==2) {
       sentencias
       …
  }
  else {
       if(x==3) {
            sentencias
            …
       }
  }
}
```

Pero si cada **else** tiene dentro sólo una instrucción **if** entonces se podría escribir de esta forma (que es más legible), llamada ***if-else-if***:

```
if (x==1) {
  instrucciones
  …
}
else if (x==2) {
  instrucciones
  …
}
else if (x==3) {
  instrucciones
  …
}
```

(4.3) sentencia *switch*

Se trata de una sentencia que permite construir alternativas múltiples. Pero que en el lenguaje C está muy limitada. Sólo sirve para evaluar el valor de una variable entera (o de carácter, ***char***).

Tras indicar la expresión entera que se evalúa, a continuación se compara con cada valor agrupado por una sentencia **case.** Cuando el programa encuentra un ***case*** que encaja con el valor de la expresión se ejecutan todos los ***case*** siguientes. Por eso se utiliza la sentencias **break** para hacer que el programa abandone el bloque ***switch.*** Sintaxis***:***

```
switch(expresión entera){
  case valor1:
       sentencias
       break; /*Para que programa salte fuera del switch
                de otro modo atraviesa todos los demás
                case */
  case valor2:
       sentencias
  ...
  default:
       sentencias
}
```

Ejemplo:

```
switch (diasemana) {
  case 1:
       printf(”Lunes”);
       break;
  case 2:
       printf(”Martes”);
       break;
  case 3:
       printf(”Miércoles”);
       break;
  case 4:
       printf(”Jueves”);
       break;
  case 5:
       printf(”Viernes”);
       break;

case 6:
       printf(”Sábado”);
       break;
  case 7:
       printf(”Domingo”);
       break;
  default:
       std::cout<<”Error”;
}
```

Sólo se pueden evaluar expresiones con valores concretos (no hay una ***case >3*** por ejemplo). Aunque sí se pueden agrupar varias expresiones aprovechando el hecho de que al entrar en un case se ejecutan las expresiones de los siguientes.

Ejemplo:

```
switch (diasemana) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:
       printf("Laborable");
       break;
  case 6:
  case 7:
       printf("Fin de semana");
       break;
  default:
       printf("Error");
}
```

(4.4) bucles

A continuación se presentan las instrucciones C que permiten realizar instrucciones repetitivas (bucles).

(4.4.1) sentencia *while*

Es una de las sentencias fundamentales para poder programar. Se trata de una serie de instrucciones que se ejecutan continuamente mientras una expresión lógica sea cierta.

Sintaxis:

```
while (expresión lógica) {
  sentencias
}
```

El programa se ejecuta siguiendo estos pasos:

(1) Se evalúa la expresión lógica

(2) Si la expresión es verdadera ejecuta las sentencias, sino el programa abandona la sentencia **while**

(3) Tras ejecutar las sentencias, volvemos al paso 1

Ejemplo (escribir números del 1 al 100):

```
int i=1;
while (i<=100){
  printf("%d",i);
  i++;
}
```

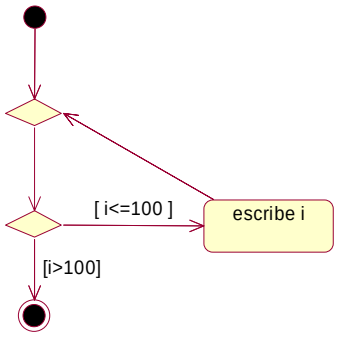


Ilustración 9, diagrama UML de actividad del bucle anterior

(4.4.2) sentencia do..while

La única diferencia respecto a la anterior está en que la expresión lógica se evalúa después de haber ejecutado las sentencias. Es decir el bucle al menos se ejecuta una vez. Es decir los pasos son:

(1) Ejecutar sentencias

(2) Evaluar expresión lógica

(3) Si la expresión es verdadera volver al paso 1, sino continuar fuera del while

Sintaxis:

```
do {
  sentencias
} while (expresión lógica)
```

Ejemplo (contar del 1 al 1000):

```
int i=0;
do {
  i++;
  printf("%d",i);
} while (i<=1000);
```

(4.4.3) sentencia *for*

Se trata de un bucle especialmente útil para utilizar contadores. Su formato es:

```
for(inicialización;condición;incremento){
  sentencias
}
```

Las sentencias se ejecutan mientras la condición sea verdadera. Además antes de entrar en el bucle se ejecuta la instrucción de inicialización y en cada vuelta se ejecuta el incremento. Es decir el funcionamiento es:

(1) Se ejecuta la instrucción de inicialización

(2) Se comprueba la condición

(3) Si la condición es cierta, entonces se ejecutan las sentencias. Si la condición es falsa, abandonamos el bloque ***for***

(4) Tras ejecutar las sentencias, se ejecuta la instrucción de incremento y se vuelve al paso 2

Ejemplo (contar números del 1 al 1000):

```
for(int i=1;i<=1000;i++){
  printf("%d",i);
}
```

La ventaja que tiene es que el código se reduce. La desventaja es que el código es menos comprensible. El bucle anterior es equivalente al siguiente bucle **while:**

```
i=1; /*sentencia de inicialización*/
while(i<=1000) { /*condición*/
  printf("%d",i);
  i++; /*incremento*/
}
```

(4.5) sentencias de ruptura de flujo

No es aconsejable su uso ya que son instrucciones que rompen el paradigma de la programación estructurada. Cualquier programa que las use ya no es estructurado. Se comentan aquí porque en algunos listados de código puede ser útil conocerlas.

(4.5.1) sentencia *break*

Se trata de una sentencia que hace que el flujo del programa abandone el bloque en el que se encuentra.

```
for(int i=1;i<=1000;i++){
  printf("%d",i);
  if(i==300) break;
}
```

En el listado anterior el contador no llega a 1000, en cuanto llega a 300 sale del *for*

(4.5.2) sentencia *continue*

Es parecida a la anterior, sólo que en este caso en lugar de abandonar el bucle, lo que ocurre es que no se ejecutan el resto de sentencias del bucle y se vuelve a la condición del mismo:

```
for(int i=1;i<=1000;i++){
  if(i%3==0) continue;
  printf("%d",i);
}
```

En ese listado aparecen los números del 1 al 1000ç, menos los múltiplos de 3 (en los múltiplos de 3 se ejecuta la instrucción ***continue*** que salta el resto de instrucciones del bucle y vuelve a la siguiente iteración.

El uso de esta sentencia genera malos hábitos, siempre es mejor resolver los problemas sin usar ***continue.*** El ejemplo anterior sin usar esta instrucción quedaría:

```
for(int i=1;i<=1000;i++){
  if(i%3!=0)      printf("%d",i);
}
```

La programación estructurada prohíbe utilizar las sentencias **break** y **continue**

(Unidad 5) Funciones

(5.1) funciones y programación modular

(5.1.1) introducción

En cuanto los programas resuelve problemas más complejos, su tamaño empieza a desbordar al programador. Para mitigar este problema apareció la **programación modular.** En ella el programa se divide en módulos de tamaño manejable. Cada módulo realiza una función muy concreta y se pueden programar de forma independiente.

En definitiva la programación modular implementa el paradigma ***divide y vencerás***, tan importante en la programación. El programa se descompone en esos módulos, lo que permite concentrarse en problemas pequeños para resolver los problemas grandes.

En C los módulos se llaman **funciones.** En los temas anteriores hemos usado algunas funciones implementadas en las librerías del lenguaje C (**printf** y **scanf** por ejemplo). El lenguaje C proporciona diversas funciones ya preparadas (se las llama **funciones estándar**) para evitar que el programador tenga que codificar absolutamente todo el funcionamiento del programa.

Las funciones son **invocadas** desde el código utilizando su nombre. Cuando desde el código se invoca a una función, entonces el flujo del programa salta hasta el código de la función invocada. Después de ejecutar el código de la función, el flujo del programa regresa al código siguiente a la invocación.

(5.1.2) uso de las funciones

Todo programa C se basa en una función llamada **main** que contiene el código que se ejecuta en primer lugar en el programa. Dentro de ese ***main*** habrá llamadas a funciones ya creadas, bien por el propio programador o bien funciones que forman parte de las bibliotecas de C o de bibliotecas privadas (una **biblioteca** o **librería** no es más que una colección de funciones).

Así por ejemplo:

```
int main(){
  printf("%lf", pow(3,4));
}
```

Ese código utiliza dos funciones, la función ***printf*** que permite escribir en la pantalla y la función ***pow*** que permite elevar un número a un exponente (en el ejemplo 3^4).

Para poder utilizarlas las funciones tienen que estar definidas en el código del programa o en un archivo externo. Si están en un archivo externo (como ocurre en el ejemplo) habrá que incluir la cabecera de ese archivo para que al crear el ejecutable de nuestro programa se enlacen los archivos compilados de las funciones externas que utilice el programa.

Es decir hay que indicar en qué archivos se definen esas funciones. En el ejemplo, habría que incluir al principio esta líneas:

```
#include <stdio.h>
#include <math.h>
```

(5.1.3) crear funciones

Si creamos funciones, éstas deben definirse en el código. Los pasos para definir una función son:

- Crear una línea en la que se indica el nombre de la función, el tipo de datos que devuelve dicha función y los parámetros que acepta. Los parámetros son los datos que necesita la función para trabajar
- Indicar las variables locales a la función
- Indicar las instrucciones de la función
- Si es preciso, indicar el valor que devuelve

Sintaxis:

```
tipo nombreDeLaFunción(parámetros){
  definiciones
  instrucciones
}
```

La primera línea de la sintaxis anterior es lo que se llama el **prototipo de la función**. Es la definición formal de la función. Desglosamos más su contenido:

- ***tipo***. Sirve para elegir el tipo de datos que devuelve la función. Toda función puede obtener un resultado. Eso se realiza mediante la instrucción **return**. El tipo puede ser: **int, char, long, float, double**,.... y también **void**. Éste último

se utiliza si la función no devuelve ningún valor (a estas funciones se las suele llamar procedimientos).

- ***nombreDeLafunción.*** El identificador de la función. Debe cumplir las reglas ya comentadas en temas anteriores correspondientes al nombre de los identificadores.
- ***parámetros.*** Su uso es opcional, hay funciones sin parámetros. Los parámetros son una serie de valores que la función puede requerir para poder ejecutar su trabajo. En realidad es una lista de variables y los tipos de las mismas. Son variables cuya existencia está ligada a la función

Ejemplo de función:

```
double potencia(int base, int exponente){
  int contador;
  int negativo; /* Indica si el exponente es negativo*/
  double resultado=1.0;

  if (exponente<0) {
       exponente=-exponente;
       negativo=1;
  }
}
```

```
for(contador=1;contador<=exponente;contador++)
       resultado*=base;

  if (negativo) resultado=1/resultado;
  return resultado;
}
```

En los parámetros hay que indicar el tipo de cada parámetro (en el ejemplo se pone **int *base*, int *exponente*** y no **int *base, exponente***), si no se pone el tipo se suele tomar el tipo **int** para el parámetro (aunque esto puede no ser así en algunos compiladores).

prototipos de las funciones

En realidad fue el lenguaje **C++** el que ideó los prototipos, pero **C** los ha aceptado para verificar mejor el tipo de los datos de las funciones.

Gracias a estos prototipos el compilador reconoce desde el primer momento las funciones que se van a utilizar en el programa. Los prototipos se ponen al principio del código (tras las instrucciones **#include**) delante del **main**. Ejemplo (programa completo):

```
#include <stdio.h>
#include <conio.h>

/*Prototipo de la función*/
double potencia(int base, int exponente);

int main(){
  int b, e;
  do{
    printf("Escriba la base de la potencia (o cero para salir");
    scanf("%d",&b);
    if (b!=0){
         printf("Escriba el exponente");
          scanf("%d",&e);
         printf("Resultado: %lf", potencia(b,e));
    }
  }while(b!=0);
  getch();
}
/*Cuerpo de la función*/
double potencia(int base, int exponente){
  int contador;
  int negativo=0;
  double resultado=1.0;
  if (exponente<0) {
       exponente=-exponente;
       negativo=1;
  }
  for(contador=1;contador<=exponente;contador++)
       resultado*=base;
  if (negativo) resultado=1/resultado;
  return resultado;
}
```

Aunque no es obligatorio el uso de prototipos, sí es muy recomendable ya que permite detectar errores en compilación (por errores en el tipo de datos) que serían muy difíciles de detectar en caso de no especificar el prototipo.

(5.1.4) funciones void

A algunas funciones se les pone como tipo de datos el tipo **void.** Son funciones que no devuelven valores. Sirven para realizar una determinada tarea pero esa

tarea no implica que retornen un determinado valor. Es decir son funciones sin instrucción **return.**

A estas funciones también se las llama **procedimientos.** Funcionan igual salvo por la cuestión de que al no retornar valore, no tienen porque tener instrucción **return.** Si se usa esa instrucción dentro de la función, lo único que ocurrirá es que el flujo del programa abandonará la función (la función finaliza). No es muy recomendable usar así el ***return*** ya que propicia la adquisición de muy malos hábitos al programar.

(5.2) parámetros y variables

(5.2.1) introducción

Ya se ha comentado antes que las funciones pueden utilizar parámetros para almacenar valores necesarios para que la función pueda trabajar.

La cuestión es que los parámetros de la función son variables locales que recogen valores enviados durante la llamada a la función y que esas variables mueren tras finalizar el código de la función. En el ejemplo anterior, ***base*** y ***exponente*** son los parámetros de la función ***potencia***. Esas variables almacenan los valores utilizados en la llamada a la función (por ejemplo si se llama con ***potencia(7,2)***, ***base*** tomará el valor 7 y ***exponente*** el valor2).

(5.2.2) paso de parámetros por valor y por referencia

Se explicará con detalle más adelante. En el ejemplo:

```
int x=3, y=6;
printf("%lf",potencia(x,y);
```

Se llama a la función ***potencia*** utilizando como parámetros los valores de x e y. Estos valores se asocian a los parámetros ***base*** y ***exponente***. La duda está en qué ocurrirá con ***x*** si en la función se modifica el valor de ***base***.

La respuesta es, nada. Cuando se modifica un parámetro dentro de una función no ocurre nada. En el ejemplo, ***base*** tomará una copia de ***x***. Por lo que ***x*** no se verá afectada por el código de la función. Cuando ocurre esta situación se dice que el parámetro se pasa por **valor.**

Sin embargo en la función **scanf** ocurre lo contrario:

```
scanf("%d",&x);
```

La variable ***x*** sí cambia de valor tras llamar a scanf. Esto significa que ***x*** se pasa **por referencia**. En realidad en C sólo se pueden pasar valores, otros lenguajes (como Pascal por ejemplo) admiten distinción. En C para usar el paso por referencia hay que pasar la dirección de la variable (&x), o lo que es lo mismo un puntero a la variable.

Es decir la llamada **scanf("%d",&x)** envía la dirección de ***x*** a la función ***printf***, esa función utiliza dicha función para almacenar el número entero que el usuario escriba por teclado.

En temas posteriores se profundizará este tema, en concreto en el tema dedicado a los punteros.

(5.2.3) variables static

Las variables ***static*** son variables que permanecen siempre en la memoria del ordenador. Su uso fundamental es el de variables locales a una función cuyo valor se desea que permanezca entre llamadas.

Para entender mejor su uso, veamos este ejemplo:

```
#include <stdio.h>
#include <conio.h>

void prueba();

int main(){
  int i;
  for(i=1;i<=10;i++) prueba;
  getch();
}

void prueba(){
  int var=0;
  var++;
  printf("%d\n",var);
}
```

En el ejemplo, se llama 10 veces seguidas a la función ***prueba()***, esta función escribe siempre lo mismo, el número 1. La razón está en que la variable ***var*** se crea e inicializa de nuevo en cada llamada a la función y muere cuando el código de la función finaliza. Con lo que siempre vale 1.

Si modificamos las cuatro últimas líneas así:

```
void prueba(){
  static int var=0;
  var++;
  printf("%d\n",var);
}
```

Ahora el resultado es absolutamente distinto. al ser estática la variable ***var*** se crea en la primera llamada y permanece en memoria, es decir no se elimina al

finalizar la función. En la segunda llamada no se crea de nuevo la variable, es más la instrucción de creación se ignora, ya que la función ya está creada. El resultado ahora es absolutamente distinto. Aparecerán en pantalla los números del 1 al 10.

(5.2.4) ámbito de las variables y las funciones

Ya se comentó en temas precedentes. Pero se vuelve a comentar ahora debido a que en las funciones es donde mejor se aplica este concepto.

Toda variable tiene un ciclo de vida que hace que la variable sea declarada, inicializada, utilizada las veces necesarias y finalmente eliminada. En la declaración además de declarar el tipo, implícitamente estamos declarando el ámbito, de modo que una variable que se declara al inicio se considera global, significa pues que se puede utilizar en cualquier parte del archivo actual.

Si una variable se declara dentro de una función, ésta muere en cuanto la función termina. Lo mismo ocurre con los parámetros, en realidad los parámetros son variables locales que copian el valor de los valores enviados en la llamada a la función, pero su comportamiento es el mismo que el de las variables locales.

Las variables ***static*** son variables locales que mantienen su valor entre las llamadas, es decir no se eliminan pero sólo se pueden utilizar en la función en la que se crearon.

El ámbito más pequeño lo cubren las variables declaradas dentro de un bloque ({}) de una función. Estas variables sólo se pueden utilizar dentro de las llaves en las que se definió la variables, tras la llave de cierre del bloque (}), la variable muere.

Es importante tener en cuenta que **todo lo comentado anteriormente es aplicable a las funciones.** Normalmente una función es global, es decir su prototipo se coloca al principio del archivo y eso hace que la función se pueda utilizar en cualquier parte del archivo, pero lo cierto es que **se puede definir una función dentro de otra función.** En ese último caso dicha función sólo se puede utilizar dentro de la función en la que se declara.

(5.3) recursividad

(5.3.1) introducción

La recursividad es una técnica de creación de funciones en la que una función se llama a sí misma. Hay que ser muy cauteloso con ella (incluso evitarla si no es necesario su uso), pero permite soluciones muy originales , a veces, muy claras a ciertos problemas. De hecho ciertos problemas (como el de las torres de Hanoi, por ejemplo) serían casi imposibles de resolver sin esta técnica.

La idea es que la función resuelva parte del problema y se llame a sí misma para resolver la parte que queda, y así sucesivamente. En cada llamada el problema debe ser cada vez más sencillo hasta llegar a una llamada en la que la función devuelve un único valor.

Es fundamental tener en cuenta cuándo la función debe dejar de llamarse a sí misma, es decir cuando acabar. De otra forma se corre el riesgo de generar infinitas llamadas, lo que bloquearía de forma grave el PC en el que trabajamos.

Como ejemplo vamos a ver la versión recursiva del factorial.

```
double factorial(int n){
  if(n<=1) return 1;
  else return n*factorial(n-1);
}
```

La última instrucción (***return n*factorial(n-1))*** es la que realmente aplica la recursividad. La idea (por otro lado más humana) es considerar que el factorial de nueve es nueve multiplicado por el factorial de ocho; a su vez el de ocho es ocho por el factorial de siete y así sucesivamente hasta llegar al uno, que devuelve uno.

Para la instrucción **factorial(4);** usando el ejemplo anterior, la ejecución del programa generaría los siguientes pasos:

(1) Se llama a la función factorial usando como parámetro el número 4 que será copiado en la variable-parámetro ***n***

(2) Como n>1, entonces se devuelve 4 multiplicado por el resultado de la llamada **factorial(3)**

(3) La llamada anterior hace que el nuevo ***n*** (variable distinta de la anterior) valga 3, por lo que esta llamada devolverá 3 multiplicado por el resultado de la llamada **factorial(2)**

(4) La llamada anterior devuelve 2 multiplicado por el resultado de la llamada **factorial(1)**

(5) Esa llamada devuelve 1

(6) Eso hace que la llamada **factorial(2)** devuelva 2*1, es decir 2

(7) Eso hace que la llamada **factorial(3)** devuelva 3*2, es decir 6

(8) Por lo que la llamada **factorial(4)** devuelve 4*6, es decir **24** Y ese es ya el resultado final

(5.3.2) ¿recursividad o iteración?

Ya conocíamos otra versión de la función factorial resuelta por un bucle **for** en lugar de por recursividad. La cuestión es ¿cuál es mejor?

Ambas implican sentencias repetitivas hasta llegar a una determinada condición. Por lo que ambas pueden generar programas que no finalizan si la condición nunca se cumple. En el caso de la iteración es un contador o un centinela el que permite determinar el final, la recursividad lo que hace es ir simplificando el problema hasta generar una llamada a la función que devuelva un único valor.

Para un ordenador es más costosa la recursividad ya que implica realizar muchas llamadas a funciones en cada cual se genera una copia del código de la misma, lo que sobrecarga la memoria del ordenador. Es decir, **es más rápida y menos voluminosa la solución iterativa de un problema recursivo.**

¿Por qué elegir recursividad? De hecho si poseemos la solución iterativa, no deberíamos utilizar la recursividad. **La recursividad se utiliza sólo si:**

- **No encontramos la solución iterativa a un problema**
- **El código es mucho más claro en su versión recursiva**

(5.3.3) recursividad cruzada

Hay que tener en cuenta la facilidad con la que la recursividad genera bucles infinitos. Por ello una función nunca ha de llamarse a sí misma si no estamos empleando la recursividad. Pero a veces estos problemas de recursividad no son tan obvios. Este código también es infinito en su ejecución:

```
int a(){
  int x=1;
  x*=b();
  return x;
}

int b(){
  int y=19;
  y-=a();
  return y;
}
```

Cualquier llamada a la función ***a*** o a la función ***b*** generaría código infinito ya que ambas se llaman entre sí sin parar jamás. A eso también se le llama recursividad, pero recursividad cruzada.

(5.4) bibliotecas

(5.4.1) introducción

En cualquier lenguaje de programación se pueden utilizar bibliotecas (también llamadas librerías) para facilitar la generación de programas. Estas bibliotecas en realidad son una serie de funciones organizadas que permiten realizar todo tipo de tareas.

Cualquier entorno de programación en C permiten el uso de las llamadas bibliotecas estándar, que son las que incorpora el propio lenguaje de programación.

Hay varias librerías estándar en C. Para poder utilizarlas se debe incluir su archivo de cabecera en nuestro código avisando de esa forma al compilador que se debe enlazar el código de dicha librería al nuestro para que nuestro programa funcione.

Eso se realizar mediante la directiva de procesador **#include** seguida del nombre del archivo de cabecera de la librería. Ese archivo suele tener extensión **.h** y contiene las declaraciones de las funciones de dicha librería.

(5.4.2) ejemplo de librería estándar. *math*

Math es el nombre de una librería que incorpora funciones matemáticas. Para poder utilizar dichas funciones hay que añadir la instrucción

```
#include <math.h>
```

al principio del código de nuestro archivo. Cada directiva ***include*** se coloca en una línea separada y no requiere punto y coma al ser una indicación al compilador y no una instrucción de verdad.

Esta librería contiene las siguientes funciones (se indican los prototipos y uso):

Prototipo	Descripción
int abs(int *n*)	Devuelve el valor absoluto del número entero indicado
int ceil(double *n*)	Redondea el número decimal ***n*** obteniendo el menor entero mayor o igual que ***n.*** Es decir ***ceil(2.3)*** devuelve 3 y ***ceil(2.7)*** también devuelve 3
double cos(double *n*)	Obtiene el coseno de ***n*** (***n*** se expresa en radianes)
double exp(double *n*)	Obtiene e***n***
double fabs(double *n*)	Obtiene el valor absoluto de ***n*** (siendo ***n*** un número ***double***)
int floor(double *n*)	Redondea el número decimal ***n*** obteniendo el mayor entero menor o igual que ***n.*** Es decir ***floor(2.3)*** devuelve 2 y ***floor(2.7)*** también devuelve 2
double fmod(double *x*, double *y*)	Obtiene el resto de la división entera de ***x/y***
double log(double *n*)	Devuelve el logaritmo neperiano de ***n***
double log10(double *n*)	Devuelve el logaritmo decimal de ***n***
double pow(double *base*, double *exponente*)	Obtiene $base^{exponente}$
double sin(double *n*)	Obtiene el seno de ***n*** (***n*** se expresa en radianes)
double sqrt(double *n*)	Obtiene la raíz cuadrada de ***n***
double sin(double *n*)	Obtiene la tangente de ***n*** (***n*** se expresa en radianes)

(5.4.3) números aleatorios

Otras de las funciones estándar más usadas son las de manejo de números aleatorios. Están incluidas en **stdlib.h**.

La función **rand()** genera números aleatorios entre 0 y una constante llamada RAND_MAX que está definida en stdlib.h (normalmente vale 32767). Así el código:

```
for(i=1;i<=100;i++)
  printf("%d\n",rand());
```

Genera 100 números aleatorios entre 0 y RAND_MAX. Para controlar los números que salen se usa el operador del módulo (%) que permite obtener en este caso números aleatorios del en un rango más concreto.

Por ejemplo:

```
for(i=1;i<=100;i++)
  printf("%d\n",rand()%10);
```

Obtiene 100 números aleatorios del 0 al 9. Pero ocurre un problema, los números son realmente ***pseudoaleatorios***, es decir la sucesión que se obtiene siempre es la misma. Aunque los números circulan del 0 al 9 sin orden ni sentido lo cierto es que la serie es la misma siempre.

La razón de este problema reside en lo que se conoce como semilla. La semilla es el número inicial desde el que parte el programa. La semilla se puede generar mediante la función **srand** a la que se le pasa como parámetro un número entero que será la semilla. Según dicha semilla la serie será distinta.

Para que sea impredecible conocer la semilla se usa la expresión:

```
srand(time(NULL));
```

Esa expresión toma un número a partir de la fecha y hora actual, por lo que dicho número es impredecible (varía en cada centésima de segundo). Así realmente el código:

```
srand(time(NULL));
for(i=1;i<=100;i++)
printf("%d\n",rand()%10);
```

genera cien números del 0 al 9 siendo impredecible saber qué números son, ya que la semilla se toma del reloj del sistema.

(5.4.4) crear nuestras propias librerías

Bajo el paradigma de reutilizar el código, normalmente las funciones interesantes que crean los usuarios, es conveniente agruparlas en archivos que luego se pueden reutilizar.

Para ello se hacen los siguientes pasos:

(1) Se crea en un archivo (generalmente con extensión .h) las funciones. En esos archivos se agrupan las funciones según temas. Esos archivos **no**

tendrán ninguna función *main*, sólo se incluyen las funciones de nuestra librería

(2) Se crea el archivo principal con su función ***main*** correspondiente.

(3) En dicho archivo se utiliza la instrucción ***include*** seguida de la ruta al archivo que contiene alguna de las funciones que deseamos usar. Esa ruta debe ir entre comillas dobles (de otra forma se busca el archivo en la carpeta de librerías estándar). Cada archivo de funciones requiere una instrucción ***include*** distinta.

(Unidad 6)
Estructuras estáticas

(6.1) introducción

En programación se llaman estructuras estáticas a datos compuestos de datos simples (enteros, reales, caracteres,...) que se manejan como si fueran un único dato y que ocupan un espacio concreto en memoria.

Las estructuras estáticas son:

- **Arrays.** También llamadas listas estáticas, matrices y arreglos (aunque quizá lo más apropiado es no traducir la palabra array). Son una colección de datos del mismo tipo.
- **Cadenas.** También llamadas **strings**. Se trata de un conjunto de caracteres que es tratado como un texto completo.
- **Punteros**. Permiten definir variables que contienen posiciones de memoria; son variables que se utilizan para apuntar a otras variables
- **Estructuras**. Llamadas también **registros.** Son datos compuestos de datos de distinto tipo. Una estructura podría estar compuesta de un entero, un carácter y un array por ejemplo.

(6.2) arrays

(6.2.1) introducción

Imaginemos que deseamos leer las notas de una clase de 25 alumnos. Desearemos por tanto almacenarlas y para ello con lo que conocemos hasta ahora no habrá más remedio que declarar 25 variables.

Eso es tremendamente pesado de programar. Manejar esas notas significaría estar continuamente manejando 25 variables. Por ello en C (como en casi todos los lenguajes) se pueden agrupar una serie de variables del mismo tipo en un tipo de datos más complejo que facilita su manipulación.

Los arrays son una colección de datos del mismo tipo al que se le pone un nombre (por ejemplo ***nota***). Para acceder a un dato de la colección hay que utilizar su índice (por ejemplo ***nota[4]*** leerá la cuarta nota).

Tras leer las 25 notas el resultado se almacena en la variable ***nota*** y se podrá acceder a cada valor individual usando ***nota[i],*** donde ***i*** es el elemento al que queremos acceder. Hay que tener en cuenta que en los arrays el primer elemento es el 0; es decir ***nota[4]*** en realidad es el quinto elemento.

(6.2.2) declaración de arrays

Una array ocupa un determinado espacio fijo en memoria. Para que el ordenador asigne la cantidad exacta de memoria que necesita el array, hay que declararle. En la declaración se indica el tipo de datos que contendrá el array y la cantidad de elementos. Por ejemplo:

```
int a[7];
```

Esa instrucción declara un array de siete elementos. Lo que significa que en memoria se reserva el espacio de siete enteros (normalmente 2 o 4 bytes por cada entero). Los elementos del array irán de ***a[0]*** a ***a[6]*** (hay que recordar que en C el primer elemento de un array es el cero).

Hay que tener en cuenta que **los arrays contienen una serie finita de elementos**, es decir, de antemano debemos conocer el tamaño que tendrá el array. El valor inicial de los elementos del array será indeterminado (dependerá de lo que contenga la memoria que se ha asignado al array, ese valor no tiene ningún sentido).

(6.2.3) utilización de arrays

asignación de valores

El acceso a cada uno de los valores se realiza utilizando un **índice** que permite indicar qué elemento del array estamos utilizando. Por ejemplo:

```
a[2]=17;
printf("%d",a[2]);
```

a[2] es el tercer elemento del array, se le asigna el valor 17 y ese valor se muestra luego en la pantalla.

Para rellenar los datos de un array se suelen utilizar bucles, en especial el bucle **for**. Por ejemplo en el caso de tener que leer 21 notas, gracias a los arrays no necesitamos 21 instrucciones, sino que basta con un bucle con contador que vaya recorriendo cada elemento.

Ejemplo (lectura de 21 notas):

```
/* Array de 21 elementos */
int nota[21];
/* Contador para recorrer el array */
int i;

for(i=0;i<21;i++) {
  printf("Escriba la nota %d:",i);
  scanf("%d",&nota[i]);
}
```

Al final del bucle anterior, las notas estarán rellenadas. Por ejemplo con estos valores:

componente	nota[0]	nota[1]	nota[2]	...	nota[19]	nota[20]
valor	5	6	4	...	7	4

Otro ejemplo (array que almacena y muestra los 20 primeros factoriales);

```
double factorial[21];
int i;

factorial[0]=1;
for(i=1;i<21;i++){
  factorial[i]=i*factorial[i-1];
}
/*Mostrar el resultado*/
for(i=0;i<21;i++){
  printf("El factorial de %d es .0f\n",i,factorial[i]);
}
```

En C hay que tener un cuidado especial con los índices, por ejemplo este código es erróneo:

```
int a[4];
a[4]=13;
```

No hay elemento ***a[4]*** por lo que ese código no tiene sentido. Sin embargo **en C no se nos advierte del fallo**, de hecho el uso de ***a[4]*** permite almacenar el número 13 en la posición del que sería el quinto elemento del array. Como eso no tiene sentido, la instrucción escribe en una zona de memoria que podría estar

utilizando otra variable, lo que provocaría que el programa no funcione correctamente (aunque no se nos avise del error).

Hay que tener mucho cuidado con este tipo de errores, conocidos como errores de **desbordamiento de índice de array**. Ya que hace que manipulemos la memoria sin control alguno. Los programas fallan en su ejecución, pero será muy difícil detectar por qué.

Una buena práctica de programación consiste en utilizar constantes en lugar de números para indicar el tamaño del array. En el ejemplo de las notas ocurre que si nos hemos equivocado y en lugar de 21 notas había 30, tendremos que cambiar todos los 21 por 30. En su lugar es más conveniente:

```
#define TAMANIO 20
int nota[TAMANIO];
/* Contador para recorrer el array */
int i;

for(i=0;i<TAMANIO;i++) {
  printf("Escriba la nota %d:",i);
  scanf("%d",&nota[i]);
}
```

La directiva **#define** permite definir macros. El compilador sustituirá el nombre TAMANIO por 20 antes de compilar. Si resulta que el número de notas es otra, basta cambiar sólo la línea del ***define*** y no el resto.

iniciar arrays en la declaración

C posee otra manera de asignar valores iniciales al array. Consiste en poner los valores del array entre llaves en la declaración del mismo.

Ejemplo:

```
int nota[10]={3,5,6,7,3,2,1,7,4,5};
```

nota[0] valdrá 3, ***nota[1]*** valdrá 5, ***nota[2]*** valdrá 6 y así sucesivamente. Las llaves sólo se pueden utilizar en la declaración. Si hay menos elementos en las llaves de los que posee el array entonces se rellenan los elementos que faltan con ceros, ejemplo:

```
int nota[10]={3,5};
int i=0;
for(i=0;i<10;i++) printf("%3d",nota[i]);
```

El resultado es:

```
5  0  0  0  0  0  0  0  0
```

es decir, ***nota[0]*** valdrá 3, ***nota[1]*** valdrá 5, y el resto se rellena con ceros. En el caso de que haya más elementos en las llaves de los que declaramos en el array, por ejemplo:

```
int a[2]={3,5,4,3,7};
```

Esa instrucción produce un error.

Otra posibilidad es declarar el array sin indicar el tamaño y asignarle directamente valores. El tamaño del array se adaptará al tamaño de los valores:

```
int a[]={3,4,5,2,3,4,1}; /* a es un array int[7] */
```

(6.2.4) pasar un array como parámetro de una función

Los arrays son un elemento muy interesante para utilizar conjuntamente con las funciones. Al igual que las funciones pueden utilizar enteros, reales o caracteres como argumentos, también pueden utilizar arrays; pero su tratamiento es diferente.

En el tema anterior (dedicado a las funciones), ya se comentó la que significaba pasar como valor o pasar por referencia una variable. Cuando se pasa por valor, se toma una copia del valor del parámetro; por referencia se toma la dirección del dato. La función **scanf** requiere que el segundo argumento se pase por referencia para almacenar los valores del teclado en una dirección concreta.

Cuando las variables son simples, la dirección de la variable se toma mediante el operador **&.** Por ejemplo, si ***a*** es una variable entera, ***&a*** devuelve la dirección de dicha variable.

En el caso de los arrays hay que tener en cuenta que la dirección en la que comienza el array es la dirección de su primer elemento. Es decir si ***nota*** es un array, los valores de ese array se empieza a almacenar en ***&nota[0],*** es decir en la dirección de la primera nota; consecutivamente se irán almacenando el resto. Lo importante es que la variable que hace referencia a todo el array (por ejemplo ***nota***) en realidad apunta al primer elemento del array (más adelante diremos que es un ***puntero***).

Con lo dicho anteriormente hemos de tener claro que ***nota*** y ***&nota[0]*** se refieren a la dirección de memoria en la que se almacena el primer valor del array.:

```
int nota[8]={4,6,7,6,8,9,8,4};
printf("%d\n",&nota[0]); /* Escribe una dirección de memoria */
printf("%d\n",nota); /* Escribe la misma dirección */
printf("%d\n",&nota); /* De nuevo escribe la misma dirección */
```

En el ejemplo anterior, la tercera línea que escribe ***&nota,*** denota que ***&nota*** y ***nota*** es la misma cosa.

De esta forma hay que tener en cuenta que cuando se utiliza un array como parámetro de una función, lo que ésta recibe es una referencia al primer elemento. Por ejemplo, supongamos que queremos calcular la media aritmética

de un array de enteros. Crearemos entonces la función ***media*** que recibe dos parámetros: un array de enteros y el tamaño de dicho array. El tamaño es necesario ya que desde la función no sabremos cuál será dicho tamaño.

El código será:

```
double media(int numero[], int tamanio){
  double res;
  int i;
  for(i=0;i<tamanio;i++){
       res+=numero[i]; /* Se acumula en res la suma
                          total del array */
  }
  return res/tamanio; /* res/tamanio es la media */
}
```

Hay que tener en cuenta que si en una función se cambia el valor de un elemento del array, ese cambio de valor afecta al array original. Esto se debe al hecho de que el array se pase por referencia y no por valor. Si una función utiliza un parámetro de tipo **int** y se cambia su valor en la función, eso no significa que se cambie el valor original. Es decir en el ejemplo:

```
#include <stdio.h>
void prueba(int x);
int main(){
  int a=12;
  prueba(a); /*Se llama a la función prueba con el valor
a */
  printf("%d",a); /* escribirá el valor 12 */
}
void prueba(int x){
  x=3;
}
```

Al llamar a la función prueba, el parámetro ***x*** toma una copa de ***a.*** A esa copia le pone el valor 3. Cuando el programa regresa de la función, el valor de ***a*** no ha cambiado.

En este otro caso:

```
#include <stdio.h>
void prueba(int[] x);
int main(){
  int a[4];
  a[0]=12;
  prueba(a); /*Se llama a la función prueba con el valor
a */
  printf("%d",a[0]); /* escribirá el valor 3 */
}

void prueba(int[] x){
  x[0]=3;
}
```

aquí el valor del elemento a[0] sí que cambia por que en la función prueba, **x** no es una copa de ***a***; ***x*** y ***a*** son la misma cosa, la dirección del mismo array.

El modificador **const** permite restringir la escritura de un array cuando es utilizado como parámetro. Si se modifica el array ocurrirá un error en la ejecución. Ejemplo:

```
void prueba(const int x[],int n){
  x[0]=4; /* Error no se puede modificar x al usar const */
}
```

(6.2.5) algoritmos de búsqueda y ordenación en arrays

búsqueda en arrays

En muchas ocasiones se necesita comprobar si un valor está en un array. Para ello se utilizan funciones que indican si el valor está o no en el array. Algunas devuelve verdadero o falso, dependiendo de si el valor está o no; y otras (mejores aún) devuelven la posición del elemento buscado en el array (o un valor clave si no se encuentra, por ejemplo el valor -1).

Hay que distinguir entre el hecho de si el array está ordenado o no.

búsqueda lineal

En este caso el array no está ordenado. La función de búsqueda busca un valor por cada elemento del array. Si le encuentra devuelve la posición del mismo y si no devuelve -1.

```
/* parámetros:
   vector: Array que contiene todos los valores
   tamanio: Tamaño del mismo
   valor: valor a buscar
*/
int busquedaLineal(int vector[], int tamanio, int valor){
   int i=0;
   int enc=0; /* Indica si se ha encontrado el valor */
   while (enc==0 && i< tamanio){
         if(vector[i]==valor) enc=1;
         else i++;
   }
   if(enc==1) return i;
   else return -1;
}
```

búsqueda binaria

Se basa en que el array que contiene los valores está ordenado-. En ese caso la búsqueda anterior es excesivamente lenta.

Este algoritmo consiste en comparar el valor que buscamos con el valor que está en el centro del array; si ese valor es menor que el que buscamos, ahora buscaremos en la mitad derecha; si es mayor buscaremos en la mitad derecha. Y así hasta que lo encontremos.

Una variable (en el código es la variable **central**) se encarga de controlar el centro del array que nos queda por examinar, las variables **bajo** y **alto** controlan los límites (inferior y superior respectivamente) que nos quedan por examinar.

En caso de que el valor no se encuentre, entonces los límites **bajo** y **alto** se cruzan. Entonces el algoritmo devuelve -1 para indicar que el valor no se encontró. Esta búsqueda es rapidísima ya que elimina muchas comparaciones al ir quitando la mitad de lo que nos queda por examinar.

Código de búsqueda binaria:

```
/* parámetros:
   vector: Array que contiene todos los valores
   tamanio: Tamaño del array
   valor: valor a buscar
*/
int busquedaBinaria(int vector[], int tamanio, int
valor){
   int central,bajo=0,alto=tamanio-1;

   while(bajo<=alto){
      central=(bajo+alto)/2;
      if(valor==vector[central])
         return central;
      else if (valor<vector[central])
         /* Se busca en la mitad izquierda */
         alto=central-1;
      else
         /* Se busca en la mitad derecha */
         bajo=central + 1;
   }/* Fin del while */

   return -1; /* No se encontró */
}
```

ordenación de arrays

Se trata de una de las operaciones más típicas de la programación. Un array contiene datos sin ordenar, y tenemos que ordenarle.

método por inserción directa

Consiste en que el array está en todo momento ordenado y cada vez que se añade un elemento al array, éste se coloca en la posición que corresponda.

Para ello habrá que desplazar elementos. Si el array es:

1	2	3	5	6	7		

y ahora añadimos el número 4, el resultado será:

			4	5	6	7	

se han tenido que desplazar los valores 5,6 y 7 para hacer hueco al 4 en su sitio.

Para un array de enteros la función que inserta un nuevo valor en dicho array sería:

```
/* Parámetros de la función:
  vector: Array que contiene los datos ya ordenados
  tamanio: Tamaño actual del array, es el tamaño de los
       datos ordenados. cero significa que aún no se ha
       insertado ningún valor
  valor: El nuevo valor a insertar en el array
*/
void insercion(int vector[], int  tamanio, int valor){
  /* valor es el valor que se inserta */
  int i,pos=0;
  int enc=0; /* indica si hemos encontrado la posicion
del
            elemento a insertar*/

  if (tamanio==0) /*Si no hay datos añadimos este y
salimos*/
       vector[0]=valor;
  else{ /* Hay datos */
       /*Localizar la posición en la que irá el nuevo
valor*/
      while(enc==0 && pos<tamanio ){
         if(valor<vector[pos]) enc=1;
         else pos++;
       }
       /* Desplazar los elementos necesarios para hacer
hueco
            al nuevo */
      for(i=tamanio-1;i>=pos;i--)
           vector[i+1]=vector[i];

      /*Se coloca el elemento en su posición*/
      if(pos<=tamanio) vector[pos]=valor;

   }/*fin del else */
}
```

método burbuja

Para ordenar un array cuando ya contiene valores y éstos no están ordenados, no sirve el método anterior.

Para resolver esa situación, el algoritmo de ordenación de la **burbuja** o de **intercambio directo** es uno de los más populares. Consiste en recorrer el array ***n*** veces, donde ***n*** es el tamaño del array. De modo que en cada recorrido vamos empujando hacia el principio del array los valores pequeños (si ordenamos en ascendente). Al final, el array está ordenado. La función que ordenaría mediante el método de la burbuja un array de enteros de forma ascendente.

Código:

```
void burbuja(int vector[], int tamanio){
  int i,j;
  int aux; /* Para intercambiar datos del array */
  for(i=0;i<tamanio;i++){
       for(j=0;j<tamanio-1;j++){
            if(vector[j]>vector[j+1]){
                 aux=vector[j];
                 vector[j]=vector[j+1];
                 vector[j+1]=aux;
            }
       }
  }
}
```

método de ordenación rápida o quicksort

Es uno de los métodos más rápidos de ordenación. Se trata de un algoritmo recursivo que permite ordenar indicando primero que parte del array se quiere ordenar. Inicialmente se indica ordenar desde el primero al último elemento del array., consiste en dividir lo que se desea ordenar en varios trozos que se van ordenando recursivamente.

```
/* Parámetros de la función:
  a: Array que contiene los datos ya
  izda: Posición inicial desde la que se ordena el array
  dcha: Posición final hastala que se ordena el array
*/
void quicksort(int a[], int izda, int dcha)
{
  /* Función auxiliar que sirve para intercambiar dos
elementos dentro del array*/
  void swap(int a[],int i,int j){
    int aux;
    aux=a[i];
    a[i]=a[j];
    a[j]=aux;
  }


int i,j,v,aux;
  if(dcha > izda)
  {
        v = a[dcha]; i = izda-1; j = dcha;
        for(;;){
             while(a[++i] < v && i < dcha);
             while(a[--j] > v && j > izda);
             if(i >= j)
                  break;
             swap(a,i,j);
        }
        swap(a,i,dcha);
        quicksort(a,izda,i-1);
        quicksort(a,i+1,dcha);}
}
```

La llamada a esta función para ordenar un array es ***quicksort(0,tamanio)***, donde ***tamanio*** es el tamaño del array menos uno.

(6.2.6) arrays multidimensionales

En muchas ocasiones se necesitan almacenar series de datos que se analizan de forma tabular. Es decir en forma de tabla, con su fila y su columna. Por ejemplo:

	columna 0	columna 1	columna 2	columna 3	columna 4	columna 5
fila 0	4	5	6	8	9	3
fila 1	5	4	2	8	5	8
fila 2	6	3	5	7	8	9

Si esos fueran los elementos de un array de dos dimensiones (por ejemplo el array **a**) el elemento resaltado (de verde) sería el ***a[3][1].***

La declaración de arrays de dos dimensiones se realiza así:

```
int a[3][6];
```

a es un array de tres filas y seis columnas. Otra forma de declarar es inicializar los valores:

```
int a[][]={{2,3,4},{4,5,2},{8,3,4},{3,5,4}};
```

En este caso ***a*** es un array de cuatro filas y tres columnas.

Al igual que se utilizan arrays de dos dimensiones se pueden declarar arrays de más dimensiones. Por ejemplo imaginemos que queremos almacenar las notas de 6 aulas que tienen cada una 20 alumnos y 6 asignaturas. Eso sería un array de una dimensión de 720 elementos, pero es más claro el acceso si hay tres dimensiones: la primera para el aula, la segunda para el alumno y la tercera para la asignatura.

En ese caso se declararía el array así:

```
int a[6][20][6];
```

Y para colocar la nota en el aula cuarta al alumno cinco de la asignatura 3 (un ocho es la nota):

```
a[3][4][2]=8;
```

Cuando un array multidimensional se utiliza como parámetro de una función, entonces se debe especificar el tamaño de todos los índices excepto del primero. La razón está en que de otra forma no se pueden calcular los índices correspondientes por parte de la función. Es decir:

```
void funcion(int a[][]);
```

Eso es incorrecto, en cuanto hiciéramos uso de un acceso a a[2][3], por ejemplo, no habría manera de saber en qué posición de memoria está ese valor ya que sin

saber cuántas columnas forman parte del array, es imposible determinar esa posición. Lo correcto es:

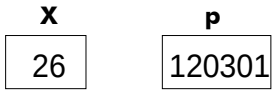
```
void funcion(int a[][5]);
```

(6.3) punteros

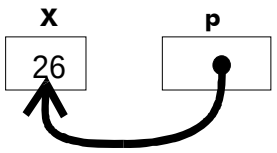
(6.3.1) introducción

Se trata de una de las herramientas más importantes de C. Se trata de una variable cuyo contenido es la dirección al contenido de otra variable. Son la base de las estructuras dinámicas.

En general una variable contiene un valor que es con el que se opera, en el caso de los punteros no es un valor directo sino que es la dirección de memoria en la que se encuentra el valor.



Si x es una variable normal y p es un puntero, 120301 será una dirección de memoria. Supongamos que esa dirección de memoria es la de la variable x. Entonces p apunta a x:



(6.3.2) declaración de punteros

Un puntero señala a una dirección de memoria. Esa dirección contendrá valores de un determinado tipo. Por ello al declarar un puntero hay que indicar de qué tipo es el puntero; o, lo que es lo mismo, el tipo de valores a los que apunta. La sintaxis es:

```
tipo *nombrePuntero;
```

El asterisco es el que indica que lo que tenemos es un puntero. El **tipo** es el tipo de valores a los que señala el puntero. Ejemplo:

```
int *ptrSuma;
```

ptrSuma es un puntero a valores enteros.

(6.3.3) operaciones con punteros

Para asignar valores a un puntero muchas veces se utiliza el operador **&** (ya comentado en los temas anteriores) que sirve para obtener la dirección de una variable. Ejemplo:

```
int *ptrSuma, suma=18;
ptrSuma=&suma;
```

Desde la última instrucción **ptrSuma** contendrá la dirección de la variable **suma**; o dicho de otra forma: **ptrSuma** apunta a la variable **suma**.

Cuando un puntero tiene un determinado valor (apunta a una determinada dirección), a menudo desearemos acceder al contenido de la dirección a la que apunta. Para ello se utiliza el operador "*****", que permite acceder al contenido del puntero. Ejemplo:

```
int *ptrSuma, suma=18;
ptrSuma=&suma;
printf("%d",*ptrSuma); /*Escribe 18*/
```

Ese mismo operador se utiliza para cambiar el valor de la variable a la que apunta el puntero.

Ejemplo:

```
int *ptrSuma, suma=18;
ptrSuma=&suma;
*ptrSuma=11;
printf("%d",x); /* Escribe 11, ahora x vale 11*/
```

(6.3.4) punteros como argumento

Un puntero puede ser un argumento de una función, en ese caso basta con indicar el tipo del puntero (**int *, char *,...**). Los punteros se suelen utilizar para llevar a cabo el paso por referencia de las variables.

Como ejemplo supongamos que queremos calcular el cubo de un número. Lo normal es que la función que calcula el cubo fuera esta:

```
double cubo(double n){
  return n*n*n;
}
```

Y en este código:

```
double x=8;
double y=cubo(x)
```

y tiene el cubo de la variable ***x*** (***8^3***).

Pero otra forma podría ser usando el cubo con paso por referencia. La función podría ser:

```
void cubo(double *ptrX){
  *ptrX=*ptrX * *ptrX * *ptrX;
}
```

A esta función se la pasa un puntero a un número **double** y el contenido de ese puntero es multiplicado por si mismo tres veces. La llamada ala función podría ser:

```
double y=8.0;
cubo(&y);
```

en esta llamada **y** pasa su dirección a la función **cubo**. De esta forma cubo tiene un puntero a **y** con el que modifica su contenido. Este es un buen ejemplo de variable pasada por referencia.

(6.3.5) aritmética de punteros

A los punteros se les puede aplicar operaciones aritméticas sin que ello produzca errores de ningún tipo. Pero hay que tener en cuenta que esta aritmética es un tanto peculiar. De hecho en este programa:

```
int *p;
p=180000; /* p apunta a la dirección 18000 de memoria */
printf("%d\n",p); /* Escribe 18000 */
printf("%d",p+2); /* Escribe 18008 */
```

Es toda una sorpresa la última línea: lo esperable sería que ***p+2*** sumara dos a la dirección que contiene **p**. Pero le suma 8, aunque en muchos compiladores sumaría 4. De hecho lo que suma son dos veces el tamaño de los enteros (se puede saber esta cifra usando ***sizeof(int)***).

Es decir si ***p*** apunta a un entero, ***p+1*** apunta al siguiente entero de la memoria. De hecho, la expresión ***p++*** hace que se incremente ***p*** para que apunte al siguiente entero en memoria. Lo mismo vale para punteros de cualquier tipo de datos que no sean enteros.

También se pueden restar punteros. El resultado es el número de bytes que separan aun puntero de otro. Por ejemplo:

```
int *p1, *p2, x=8,y=7;
p1=&x;
p2=&y;
printf("Enteros que se paran a p1 de p2=%d",p2-p1);
```

Esa diferencia es la correspondiente al número de posiciones en memoria para enteros que separan a ***p1*** de ***p2***. Es decir, si la diferencia es de 2 significará que hay 8 bytes entre los dos punteros (suponiendo que un entero ocupe 4 bytes).

(6.3.6) punteros y arrays

La relación entre punteros y arrays es muy directa, ya que los arrays son referencias a direcciones de memoria en la que se organizan los datos. Tanto es así, que **se puede hacer que un puntero señale a un array**. Por ejemplo:

```
int A[]={2,3,4,5,6,7,8};
int ptr;
ptr=A;
```

Esa última instrucción hace que el puntero señale al primer elemento del array. Desde ese momento se pueden utilizar los índices del array o la aritmética de punteros para señalar a determinadas posiciones del array. Es decir: es lo mismo ***A[3]*** que ***ptr+3***. De esta forma también son equivalentes estas instrucciones:

```
x=A[3]; x=*(ptr+3);
```

Pero la analogía entre punteros y arrays es tal, que incluso se pueden utilizar índices en el puntero:

```
x=ptr[3];
```

También sería válido (con lo cual se observa ya la gran equivalencia entre arrays y punteros):

```
x=*(A+3);
```

Es decir, son absolutamente equivalentes los arrays y los punteros.

Una de las grandes utilidades de esta capacidad es utilizar esta equivalencia para pasar los arrays a las funciones

(6.4) cadenas de caracteres

(6.4.1) introducción

Las cadenas de caracteres son los textos formados por varios caracteres. En C las cadenas (o **strings**) son arrays de caracteres, pero que tienen como particularidad que su último elemento es el carácter con código cero (es decir '\0').

En sentido estricto una cadena de C es un puntero al primer carácter de un array de caracteres que señala al primer carácter. Sólo que esa lista de caracteres finaliza en el código 0.

(6.4.2) declarar cadenas

Las cadenas de caracteres no son exactamente arrays de caracteres (es importante tener en cuenta que las cadenas terminan con el carácter nulo).

Por ello su declaración puede ser una de las siguientes:

```
char nombre1[]={'J','u','á','n','\0'};
char nombre2[]="Juán";
char *nombre3="Juán";
```

Las dos primeras declaraciones, declaran un array cuyo contenido es en ambos casos el texto ***Juán*** finalizando con el carácter 0. La segunda definición creará el texto ***Juán*** (con su terminación en nulo) en algún lugar de memoria, a cuyo primer elemento señala el puntero ***nombre3***.

La tercera forma de declarar e inicializar cadenas, puede tener problemas en algunos compiladores (como ***Dev C++*** por ejemplo), ya que la cadena se coloca en zonas de la memoria en las que no se pueden modificar. En esos compiladores es mejor inicializar cadenas utilizando siempre un array (usando la segunda forma en el ejemplo).

Al utilizar arrays para guardar cadenas, convienen que éstos sean lo suficientemente grandes, de otro modo tendríamos problemas ya que al sobrepasar el array invadiríamos el espacio de otras variables. Si no queremos dar valores iniciales a la cadena, conviene declararla en forma de array con tamaño suficiente. Por ejemplo:

```
char cadena[80];
```

Lo cual hace que se reserven en memoria 80 caracteres para esa variable.

Los punteros que señalan a cadenas se utilizan mucho ya que, puesto que las cadenas finalizan con el delimitador cero, en casi todos los algoritmos con textos, son los punteros los que recorren y operan con los caracteres.

```
char cadena[80]="Esto es una prueba";
char *p=cadena; /* El puntero señala al primer carácter de la
                   cadena */
puts(p); /*escribe: Esto es una prueba */

while (*p!=0){
  printf("%c\n",*p); /*Escribe cada carácter en una línea*/
  p++;
}
```

(6.4.3) leer y escribir cadenas

Para escribir una cadena de caracteres por la pantalla, basta con utilizar el modificador %s:

```
printf("%s",nombre1);
```

También es posible utilizar el método **puts** al que se le pasa una cadena. Este método escribe el texto de la cadena y el cambio de línea. Ejemplo:

```
puts(s);
```

De igual manera se puede leer una cadena por teclado:

```
scanf("%s" ,nombre1);
```

Mejor aún se puede utilizar la función **gets** que sirve exclusivamente para leer cadenas (es más aconsejable). Ejemplo:

```
gets(s);
```

No hace falta utilizar el signo &, ya que toda cadena es un array de caracteres y por lo tanto ***nombre1*** es la dirección inicial del texto.

Por otro lado hay dos funciones de lectura de caracteres: **getchar** y **getch.** Ambas funcionan igual sólo que con **getchar** hay que pulsar la tecla ***Intro*** para acabar la lectura, mientras que **getch** lee sin esperar la pulsación de la tecla ***Intro***. Ejemplo:

```
char c;
c=getchar(); /*habrá que escribir el carácter y pulsar
              Intro*/
c=getch(); /*nada más pulsar la tecla con el carácter, se
            almacena*/
```

La función **putchar** escribe en pantalla el carácter que se le pase como argumento.

Otra función interesante es **sprintf** que funciona igual que **printf** sólo que en lugar de imprimir por pantalla el texto, lo almacena en el primer argumento. Ejemplo:

```
int i=5;
char s[50];
sprintf(s,"El entero vale %d",i);
/* s vale "El entero vale 5"*/
```

(6.4.4) comparar cadenas

La comparación entre cadenas es problemática (como lo es también la comparación entre arrays). La razón está en que aunque dos cadenas tengan el mismo contenido, como sus direcciones serán distintas, la comparación con el operador "==" saldrá falsa.

La comparación de cadenas necesita comparar carácter a carácter o utilizar la función **strcmp** la librería **string.h** que devuelve 0 si dos cadenas tienen el mismo contenido, -1 si en el orden de la tabla ASCII la primera es menor que la segunda y 1 si la segunda es menor que la primera.

Hay que tener en cuenta que **strcmp** no comprueba las mayúsculas y que las letras como la eñe o las acentuadas causan problemas al comparar el orden alfabético.

(6.4.5) funciones y cadenas

En el caso de las funciones que reciben como parámetro cadenas, siempre utilizan punteros (**char ***) ya que permiten un mejor recorrido de los mismos. Por ejemplo esta función se encargaría de realizar una copia de una cadena en otra:

```
void copiaCadena(char *cadena1, const char *cadena2){
  while(*cadena2!=0){
        *cadena1=*cadena2;
        cadena1++;
        cadena2++;
  }
  *cadena1=0;
}
```

La palabra **const** sirve para indicar al compilador que la **cadena2** no puede ser modificada en el interior de la función. Es importante poner el delimitador cero al final de la primera cadena (tras haber copia el texto, es la instrucción ***cadena1=0**).

La llamada a esa función podría ser:

```
char s1[50]; /*s1 reserva 50 caracteres, eso es
                importante*/
char s2[]="Prueba";
copiaCadena(s1,s2);
puts(s1); /* escribe Prueba */
```

Una función puede devolver un puntero que señale a un carácter (una función que devuelve **char ***), pero si una función crea una cadena, al salir de la función, esta cadena se elimina. Esto significa que una **función no debe devolver la dirección de una cadena creada dentro de la función.**

(6.4.6) arrays de cadenas

Ya se ha comentado que una cadena es un array de caracteres en el que hay delimitador de fin de cadena que es el código cero . Luego todo lo comentado para los arrays y los punteros vale también para las cadenas.

La declaración:

```
char *s[10];
```

Declara un array de 10 punteros a caracteres (10 cadenas). Para inicializar con valores:

```
char *s[4]={"Ana","Pablo","Julián","Roberto"};
puts(s[2]); /* Escribe Julián */
```

No obstante esta declaración también genera problemas en numerosos compiladores (otra vez en **Dev C++** por ejemplo) ya que al modificar esas cadenas, como se almacenan en la zona de constantes, tendríamos problemas. Es mejor (si se van a modificar los textos) declarar reservando memoria, por ejemplo:

```
char s[4][50]={"Ana","Pablo","Julián","Roberto"};
puts(s[2]); /* Escribe Julián */
```

La declaración en este caso crea un array bidimensional de 4 x 50 caracteres. Eso se puede interpretar como cuatro arrays de caracteres de 50 caracteres como máximo cada uno. Así los textos utilizados en el ejemplo se copian en el área reservada por el array y sí se podrían modificar.

Si deseamos un puntero señalando a la segunda cadena del array:

```
char *p=s[2];
puts(p); /*También escribe Julián */
```

Eso es válido ya que ***s[2]*** es el tercer elemento del array ***s***, es un array de 50 caracteres. Es decir, ***s[2]*** es la tercera cadena del array de cadenas ***s***.

(6.4.7) funciones de uso con cadenas

funciones de manipulación de caracteres

Están en la librería **ctype.h** y trabajan con caracteres, pero son muy útiles para usar en algoritmos para cadenas. Las funciones son:

prototipo de la función	descripción
int **isdigit**(int carácter)	Devuelve verdadero si el carácter es un dígito de 0 a 9

prototipo de la función	descripción
int **isalpha**(int carácter)	Devuelve verdadero (un valor distinto de 0) si el carácter es una letra (del alfabeto inglés)
int **isalnum**(int carácter)	Devuelve verdadero si el carácter es una letra o un número.
int **islower**(int carácter)	Devuelve verdadero si el carácter es una letra minúscula (según el alfabeto inglés).
int **isupper**(int carácter)	Devuelve verdadero si el carácter es una letra mayúscula (según el alfabeto inglés).
int **tolower**(int carácter)	Convierte el carácter a minúsculas
int **toupper**(int carácter)	Convierte el carácter a mayúsculas
int **isspace**(int carácter)	Devuelve verdadero si el carácter es el espacio en blanco
int **iscntrl**(int carácter)	Devuelve verdadero si el carácter es de control
int **ispunct**(int carácter)	Devuelve verdadero si el carácter es de puntuación
int **isprint**(int carácter)	Devuelve verdadero si el carácter no es de control
int **isgraph**(int carácter)	Devuelve verdadero si el carácter no es de control y no es el espacio

funciones de conversión de cadenas

La librería **stdlib.h** contiene varias funciones muy interesantes para procesar cadenas son:

prototipo de la función	descripción
double **atof**(char *s)	Convierte la cadena **s** a formato **double**. Si la cadena no contiene datos que permitan la conversión (por ejemplo si contiene texto), su comportamiento es impredecible.
double **atoi**(char *s)	Convierte la cadena **s** a formato **int**. Si la cadena no contiene datos que permitan la conversión (por ejemplo si contiene texto), su comportamiento es impredecible.
double **atol**(char *s)	Convierte la cadena **s** a formato **long**. Si la cadena no contiene datos que permitan la conversión (por ejemplo si contiene texto), su comportamiento es impredecible.

funciones de manipulación de cadenas

La librería **string.h** proporciona las funciones más interesantes para manipular cadenas:

prototipo de la función	descripción
char * strcat(char *s1, **char** *s2)	Añade ***s2*** al final de la cadena ***s1***. Devuelve la propia ***s1***

prototipo de la función	descripción
char * strcpy(char *s1, char *s2)	Copia *s2* en *s1*. Devuelve la propia *s1*
int strcmp(char *s1, char *s2)	Compara *s1* con *s2*. Si *s1* es mayor devuelve un valor positivo, si es menor un valor negativo y si son iguales devuelve 0.
char * strchr(char *s1, int carácter)	Busca la primera vez que aparece el carácter dentro de la cadena *s1* si le encuentra devuelve un puntero al mismo, sino devuelve el valor NULL.
char * strrchr(char *s1, int carácter)	Busca la última vez que aparece el carácter dentro de la cadena *s1* si le encuentra devuelve un puntero al mismo, sino devuelve el valor NULL.
char * strstr(char * s1, char * s2)	Busca la primera vez que aparece el texto *s2* de la cadena *s1*. Si le encuentra devuelve un puntero al primer carácter de *s2* dentro de *s1*, sino devuelve el valor NULL.
int * strlen(char * s)	Devuelve el tamaño del texto *s*.

(6.5) estructuras

Son colecciones de variables de distinto tipo agrupadas bajo un mismo nombre. Son equivalentes a los **registros** de otros lenguajes. Representan fichas de datos referidas a la misma cosa.

(6.5.1) definición

Se trata de un tipo de datos formado por medio de variables de otros tipos. Por ejemplo:

```
struct persona{
  char nombre[25];
  char apellidos[50];
  char dni[10];
int edad;
};
```

Esa instrucción define un tipo de estructura, pero no declara variable alguna.

Las variables de estructura se definen así:

```
struct persona pepe;
```

pepe es una variable de tipo **persona.** La definición de la estructura se suele hacer en la zona global (antes del **main)** para que sea accesible a cualquier función del programa.

(6.5.2) typedef

Una forma mucho más interesante de utilizar estructuras, consiste en declararlas como definición de tipos. El ejemplo anterior podría utilizarse de esta forma:

```
struct persona{
  char nombre[25];
  char apellidos[50];
  char dni[10];
int edad;
};
typedef struct persona Persona;
```

Con esa instrucción acabamos de definir el tipo Persona (a los nombres de tipo se les suele poner la primera letra en mayúsculas). Ese tipo se puede utilizar igual que los propios de C. Es decir:

```
Persona pepe;
```

pepe es una variable de tipo Persona. Se puede hacer todo lo anterior de esta forma (definiendo la estructura y creando el tipo a la vez).

Ejemplo:

```
typedef struct{
  char nombre[25];
  char apellidos[50];
  char dni[10];
  int edad;
}Persona;
```

Así se define directamente el tipo ***Persona***. Otra vez podríamos usar:

```
Persona pepe;
```

Es mucho más aconsejable este método que el anterior.

(6.5.3) acceso a los miembros de una estructura

Los datos de las estructuras a veces se llaman **campos** (campo nombre, campo dni, etc...). Para poder cambiar o leer los datos de un campo de una estructura, se debe poner el nombre de la variable de estructura, un punto y el nombre del dato que queremos ver. Ejemplo:

```
strcpy(pepe.nombre,”Pepe”);
strcpy(pepe.apellidos,"Villegas Varas”);
pepe.edad=12;
```

(6.5.4) estructuras dentro de estructuras

Los miembros de una estructura pueden ser variables de estructura.

Por ejemplo:

```
typedef struct{
  char tipoVia;
  char nombre[50];
  int numero;
  int piso;
  char puerta[10];
} Direccion;

typedef struct{
  char nombre[25];
  char apellidos[50];
  int edad;
  Direccion direccion;
} Alumno;
```

En la definición del tipo de estructura ***Alumno*** se utiliza la variable de estructura direccion. Eso se podría hacer también como:

```
typedef struct{
  char nombre[25];
  char apellidos[50];
  int edad;
  struct{
       char tipoVia;
       char nombre[50];
       int numero;
       int piso;
       char puerta[10];
  } direccion;
} Alumno;
```

Pero es menos recomendable, ya que en el primer modo, se pueden crear variables de tipo ***Direccion*** independientemente de la estructura ***Alumno***.

Si declaramos una variable de tipo Alumno como:

```
Alumno pedro;
```

El acceso por ejemplo al número de la dirección de ***pedro*** se haría:

```
pedro.direccion.numero=7;
```

(6.5.5) operaciones con estructuras

asignación

Las estructuras pueden utilizar la operación de asignación para que dos variables de estructura copien sus valores. Por ejemplo:

```
Persona pepe,jaime;
strcpy(pepe.nombre,”Pepe”);
strcpy(pepe.apellidos,"Villegas Varas”);
pepe.edad=12;
jaime=pepe;
```

En el ejemplo, la variable ***jaime*** contendrá los mismos valores que pepe.

comparaciones

Las variables de estructura, no se pueden comparar con los operadores ==, >=, ... En su lugar habrá que comparar cada dato miembro.

operador sizeof

Este operador permite obtener el tamaño en bytes de una estructura. Por ejemplo:

```
printf("\n%d", sizeof pepe);
```

En el ejemplo anterior se devuelven los bytes que ocupa en memoria la variable **pepe.**

(6.5.6) arrays y punteros a estructuras

arrays

Se pueden crear arrays de estructuras. Antes hemos definido el siguiente tipo de estructura:

```
typedef struct{
  char nombre[25];
  char apellidos[50];
  char dni[10];
  int edad;
}Persona;
```

Tras crear dicho tipo, podemos crear un array de **Personas** de esta forma:

```
Persona amigos[15];
```

Para utilizar dicho array:

```
  Persona amigos[10];
  amigos[0].nombre="Alfonso";
  amigos[0].edad=25;
  amigos[1].nombre="Ana";
  amigos[1].edad=31;
puts(amigos[1].nombre);
```

punteros a estructuras

Ejemplo:

```
Persona ana;
Persona *ptr1=&ana;
```

ptr1 es un puntero que señala a variables de tipo **Persona.** En este ejemplo, ese puntero señala a **ana** ya que toma su dirección. Para acceder a los valores a los que señala el puntero, no se utiliza la expresión ***ptr1** a la que estábamos acostumbrados, sino que se utiliza el operador **->**. De esta forma:

```
  strcpy(ptr1->nombre,"Ana");
  strcpy(ptr1->apellidos,"Fernandez Castro");
  ptr1->edad=19;
printf("%s %s",ptr1->nombre,ptr1->apellidos);
```

En el ejemplo la expresión ***ptr1->nombre***, es equivalente a ***ana.nombre***.

(6.5.7) estructuras y funciones

Una estructura puede pasar a una función un elemento de la estructura (el nombre de una persona por ejemplo), la estructura completa o un puntero a una estructura. Imaginemos que declaramos la siguiente persona:

```
Persona pedro;
strcpy(pedro.nombre,"Pedro");
strcpy(pedro.apellidos,"Herrera Quijano");
strcpy(pedro.dni,"12345678W");
pedro.edad=23;
```

En el código se observa que Pedro tiene 23 años. Supongamos que creamos una función para subir la edad de una persona a un año.

La función sería:

```
void  subirEdad1(Persona pers){
  pers.edad++;
}
```

Pero en la llamada:

```
subirEdad1(pedro);
printf("%d",pedro.edad);
```

La edad de pedro no cambia tras la llamada. La razón es que la estructura se ha pasado por valor. Con lo que el contenido se copia en la función a la variable **pers**, y esa es la variable que se modifica. No se ha cambiado la estructura original.

La forma de pasar estructuras por referencia es utilizar punteros (al igual que en el caso de cualquier otra variable). Así el formato de la función sería:

```
void  subirEdad2(Persona *ptrPers){
  ptrPers->edad++;
}
```

Y la llamada se haría:

```
subirEdad2(&pedro);
printf("%d",pedro.edad); /* Sale 24, un año más */
```

(6.6) uniones

Es un tipo de dato muy parecido a las estructuras. En este caso los miembros comparten el mismo espacio de almacenamiento. El gasto en bytes de las uniones será el correspondiente al miembro que gasta más espacio.

El caso típico de unión sería:

```
union numero{
  int x;
  double y;
};
```

Lo que ganamos es que podemos asignar a esa variable valores tanto enteros como de tipo **double**. Ejemplo:

```
union numero1 valor = {10}; /*El 10 se almacena como
                              entero*/
union numero2 valor = {1.03}; /*Se almacena como double*/
```

(6.7) campos de bits

Los campos de bits son estructuras que sirven para almacenar bits. Los miembros de esta estructura son valores **unsigned** o **int**. En cada miembro se indica los bits que ocupa por ejemplo:

```
struct color{
  unsigned rojo:2;
  unsigned verde:2;
  unsigned azul:2;
  unsigned transpar:3;
};
```

Como en las estructuras, se puede definir el tipo con **typedef:**

```
typedef{
  unsigned rojo:2;
  unsigned verde:2;
  unsigned azul:2;
  unsigned transpar:3;
} Color;
```

En el caso de asignar valores, se obra como si fuera una estructura:

```
Color c1;
c1.rojo=3;
c1.verde=0;
c1.azul=0;
c1.transpar=5;
```

Se utiliza para empaquetar bits que tienen un significado especial.

(6.8) enumeraciones

Las enumeraciones son otro tipo definido por el usuario que sirve para definir un conjunto entero de **constantes de enumeración.** Las constantes de enumeración son identificadores (que se suelen poner en mayúsculas) asociados a números enteros.

Gracias a las enumeraciones se pueden utilizar tipos de datos que aportan una mayor claridad en el código.

Ejemplo:

```
enum meses{
  ENE, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV,
  DIC
};
```

Las constantes de enumeración (de ENE a DIC) tomarán valores de 0 a 11. Si deseáramos que tomarán valores de 1 a 12:

```
enum meses{
  ENE=1, FEB, MAR, ABR, MAY, JUN, JUL, AGO,SEP, OCT,
  NOV, DIC
};
```

Para declarar una variable de tipo enum se haría:

```
enum meses m1;
```

También se puede utilizar **typedef**, para definir un tipo propio. De hecho se suele utilizar más:

```
typedef enum {
  ENE=1, FEB, MAR, ABR, MAY, JUN, JUL, AGO,SEP, OCT,
  NOV, DIC
}Meses;
...
Meses m1;
```

Después se pueden utilizar las constantes definidas para asignar valores a la variable de enumeración:

```
m1=JUL; /* Es lo mismo que m1=7 */
```

Ejemplo completo de uso de enum:

```
typedef enum{
  ESO1, ESO2, ESO3, ESO4, FP1, FP2, BACH1, BACH2, PREU
}Clases;


int main(){
  Clases c1;
  char *nombreClases[]={   "Primero de ESO",
                           "Segundo de ESO",
                           "Tercero de ESO",
                           "Cuarto de ESO",
                           "Formación profesional 1",
                           "Formación profesional 2",
                           "Bachiller 1",
                           "Bachiller 2",
                           "Preuniversitario"
                       };
  int i;
for(i=ESO1;i<=PREU;i++){
    puts(nombreClases[i]);
  }
}
```

Ese bucle, muestra los nombres de todas las clases. Ese listado se podía hacer sin enumeraciones, pero el código es así mucho más claro.

(Unidad 7) Estructuras Externas. Archivos

(7.1) introducción

(7.1.1) archivos

El problema de los datos utilizados por un programa, es qué todos los datos se eliminan cuando el programa termina. En la mayoría de los casos se desean utilizar datos que no desaparezcan cuando el programa finaliza.

De cara a la programación de aplicaciones, un archivo no es más que una corriente (también llamada **stream**) de bits o bytes que posee un final (generalmente indicado por una **marca de fin de archivo**).

Para poder leer un archivo, se asocia a éste un **flujo** (también llamado secuencia) que es el elemento que permite leer los datos del archivo.

En C un archivo puede ser cualquier cosa, desde un archivo de una unidad de disco a un terminal o una impresora. Se puede asociar un flujo a un archivo mediante la operación de apertura del archivo.

(7.1.2) jerarquía de los datos

La realidad física de los datos es que éstos son números binarios. Como es prácticamente imposible trabajar utilizando el código binario, los datos deben de ser reinterpretados como enteros, caracteres, cadenas, estructuras, etc.

Al leer un archivo los datos de éste pueden ser leídos como si fueran binarios, o utilizando otra estructura más apropiada para su lectura por parte del programador. A esas estructuras se les llama **registros** y equivalen a las estructuras (**structs**) del lenguaje C. Un archivo así entendido es una colección de registros que poseen la misma estructura interna.

Cada registro se compone de una serie de **campos** que pueden ser de tipos distintos (incluso un campo podría ser una estructura o un array). En cada campo los datos se pueden leer según el tipo de datos que almacenen (enteros, caracteres,...), pero en realidad son unos y ceros.

En la **Ilustración 10** se intenta representar la realidad de los datos de un fichero. En el ejemplo, el fichero guarda datos de trabajadores. Desde el punto de vista humano hay salarios, nombres, departamentos, etc. Desde el punto de vista de la programación hay una estructura de datos compuesta por un campo de tipo String, un entero, un double y una subestructura que representa fechas.

El hecho de que se nos muestre la información de forma comprensible depende de cómo hagamos interpretar esa información, ya que desde el punto de vista de la máquina todos son unos y ceros.

El programador maneja la información desde un punto de vista lógico; es decir los datos los maneja de forma distinta al usuario (que conoce los datos desde un punto de vista humano, desde un **nivel conceptual**) y a la máquina (que sólo reconoce números binarios **nivel físico**).

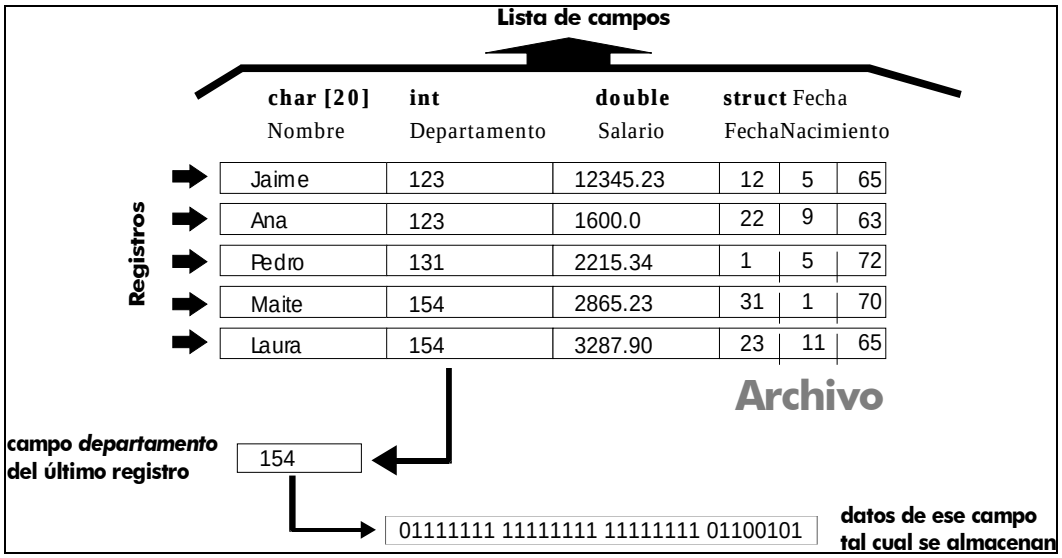


Ilustración 10, Ejemplo de la jerarquía de los datos de un archivo

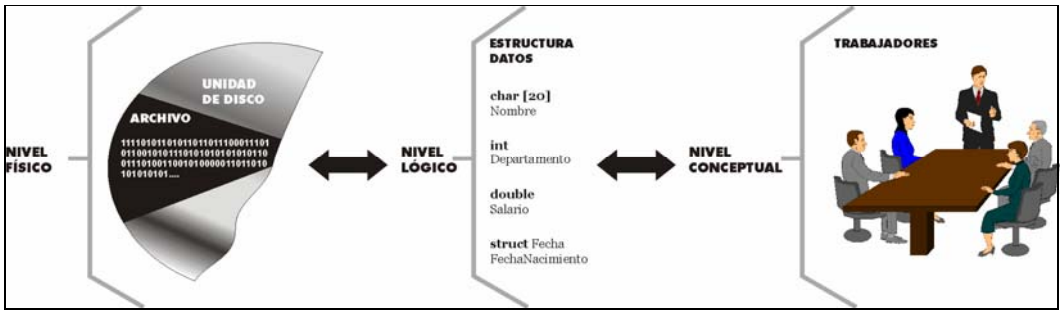


Ilustración 11, Distintos niveles de interpretación de los mismos datos, dependiendo de si les lee el sistema (nivel físico), el programador (nivel lógico) o el usuario (nivel conceptual)

(7.1.3) clasificación de los archivos

por el tipo de contenido

- **Archivos de texto.** Contienen información en forma de caracteres. Normalmente se organizan los caracteres en forma de líneas al final de cada cual se coloca un carácter de fin de línea (normalmente la secuencia "\r\n"). Al leer hay que tener en cuenta la que la codificación de caracteres puede variar (la 'ñ' se puede codificar muy distinto según qué sistema utilicemos). Los códigos más usados son:

- **ASCII**. Código de 7 bits que permite incluir 128 caracteres. En ellos no están los caracteres nacionales por ejemplo la 'ñ' del español) ni símbolos de uso frecuente (matemáticos, letras griegas,…). Por ello se uso el octavo bit para producir códigos de 8 bits, llamado ASCII extendido (lo malo es que los ASCII de 8 bits son diferentes en cada país).

- **ISO 8859-1**. El más usado en occidente. Se la llama codificación de Europa Occidental. Son 8 bits con el código ASCII más los símbolos frecuentes del inglés, español, francés, italiano o alemán entre otras lenguas de Europa Occidental.

- **Windows 1252**. Windows llama ANSI a esta codificación. En realidad se trata de un superconjunto de ISO 8859-1 que es utilizado en el almacenamiento de texto por parte de Windows.

- **Unicode**. La norma de codificación que intenta unificar criterios para hacer compatible la lectura de caracteres en cualquier idioma. Unicode asigna un código distinto a cada símbolo, de modo que los que van de 0 a 255 son los códigos de **ISO 8859-1.** De ahí en adelante se asignan más valores a los símbolos del chino, cirílico, japonés,…

  Hay varias posibilidades de implementación de Unicode, pero la más utilizada en la actualidad es la **UTF-8** que es totalmente compatible con **ASCII**. EN esta codificación se usa un solo byte para los códigos Unicode que van de 0 a 127 (que se corresponden al código ASCII). Para los siguientes códigos se pueden utilizar 2,3 ó 4 bytes dependiendo de su posición en ña tabla Unicode. De ese modo los códigos ASCII (los más utilizados en teoría) ocupan menos que el resto.

  UTF-16 codifica Unicode utilizando 16 bits (almacena hasta 65535 caracteres). Otra forma es **UTF-32** que utiliza 32 bits fijos para almacenar la información (caben hasta 4 billones de caracteres). Ocupan mucho más en disco que la codificación UTF-8, pero para los alfabetos chino o japonés ocupa menos el UTF-16 ya que cada símbolo ocupa 2 bytes, mientras que en UTF-8 los ideogramas son códigos de 3 bytes.

- **Archivos binarios.** Almacenan datos que no son interpretables como texto (números, imágenes, etc.).

por la forma de acceso

Según la forma en la que accedamos a los archivos disponemos de dos tipos de archivo:

- **Archivos secuenciales.** Se trata de archivos en los que el contenido se lee o escribe de forma continua. No se accede a un punto concreto del archivo, para leer cualquier información necesitamos leer todos los datos hasta llegar a dicha información. En general son los archivos de texto los que se suelen utilizar de forma secuencial.
- **Archivos de acceso directo.** Se puede acceder a cualquier dato del archivo conociendo su posición en el mismo. Dicha posición se suele indicar en bytes. En general los archivos binarios se utilizan mediante acceso directo.

Cualquier archivo en C puede ser accedido de forma secuencial o usando acceso directo.

(7.1.4) estructura FILE y punteros a archivos

En el archivo de cabecera **stdio.h** se define una estructura llamada FILE. Esa estructura representa la cabecera de los archivos. La secuencia de acceso a un archivo debe poseer esta estructura. Un programa requiere tener un puntero de tipo *FILE a cada archivo que se desee leer o escribir. A este puntero se le llama **puntero de archivos.** Esa estructura depende del compilador, por ejemplo en Dev-C++ es:

```
typedef struct _iobuf
{
   char*   _ptr;
   int     _cnt;
   char*   _base;
   int     _flag;
   int     _file;
   int     _charbuf;
   int     _bufsiz;
   char*   _tmpfname;
} FILE;
```

(7.2) apertura y cierre de archivos

(7.2.1) apertura

La apertura de los archivos se realiza con la función **fopen**. Esta función devuelve un puntero de tipo FILE al archivo que se desea abrir. El prototipo de la función es:

```
FILE *fopen(const char *nombreArchivo, const char *modo)
```

nombreArchivo es una cadena que contiene la ruta hacia el archivo que se desea abrir. ***modo*** es otra cadena cuyo contenido puede ser:

modo	significado
"r"	Abre un archivo para lectura de archivo de textos (el archivo tiene que existir)
"w"	Crea un archivo de escritura de archivo de textos. Si el archivo ya existe se borra el contenido que posee.
"a"	Abre un archivo para adición de datos de archivo de textos
"rb"	Abre un archivo para lectura de archivos binarios (el archivo tiene que existir)
"wb"	Crea un archivo para escritura de archivos binarios (si ya existe, se descarta el contenido)
"ab"	Abre un archivo para añadir datos en archivos binarios
"r+"	Abre un archivo de texto para lectura/escritura en archivos de texto. El archivo tiene que existir
"w+"	Crea un archivo de texto para lectura/escritura en archivos de texto. Si el archivo tenía datos, estos se descartan en la apertura.
"a+"	Crea o abre un archivo de texto para lectura/escritura. Los datos se escriben al final.
"r+b"	Abre un archivo binario para lectura/escritura en archivos de texto
"w+b"	Crea un archivo binario para lectura/escritura en archivos de texto. Si el archivo tiene datos, éstos se pierden.
"a+b"	Crea o abre un archivo binario para lectura/escritura. La escritura se hace al final de el archivo.

Un archivo se puede abrir en **modo texto** o en **modo binario**. En modo texto se leen o escriben caracteres, en modo binario se leen y escriben cualquier otro tipo de datos.

La función **fopen** devuelve un puntero de tipo FILE al archivo que se está abriendo. En caso de que esta apertura falle, devuelve el valor NULL (puntero nulo). El fallo se puede producir porque el archivo no exista (sólo en los modos ***r***), porque la ruta al archivo no sea correcta, porque no haya permisos suficientes para la apertura, porque haya un problema en el sistema,….

(7.2.2) cierre de archivos

La función **fclose** es la encargada de cerrar un archivo previamente abierto. Su prototipo es:

```
int fclose(FILE *pArchivo);
```

pArchivo es el puntero que señala al archivo que se desea cerrar. Si devuelve el valor cero, significa que el cierre ha sido correcto, en otro caso se devuelve un número distinto de cero.

(7.3) procesamiento de archivos de texto

(7.3.1) leer y escribir caracteres

función fgetc

Esta función sirve para leer caracteres de un archivo de texto. Los caracteres se van leyendo secuencialmente hasta llegar al final. Su prototipo es:

```
int fgetc(FILE *pArchivo);
```

Esta función devuelve una constante numérica llamada EOF (definida también en el archivo **stdio.h**) cuando ya se ha alcanzado el final del archivo. En otro caso devuelve el siguiente carácter del archivo.

Ejemplo:

```
#include <stdio.h>
#include <conio.h>
int main(){
    FILE *archivo;
    char c=0;
    archivo=fopen("c:\\prueba.txt","r+");
    if(archivo!=NULL) { /* Apertura correcta */
        while(c!=EOF){ /* Se lee hasta llegar al final */
            c=fgetc(archivo);
            putchar(c);
        }
       fclose(archivo);
    }
    else  printf("Error");
}
```

función fputc

Es la función que permite escribir caracteres en un archivo de texto. Prototipo:

```
int putc(int carácter, FILE *pArchivo);
```

Escribe el carácter indicado en el archivo asociado al puntero que se indique. Si esta función tiene éxito (es decir, realmente escribe el carácter) devuelve el carácter escrito, en otro caso devuelve EOF.

(7.3.2) comprobar final de archivo

Anteriormente se ha visto como la función **fgetc** devuelve el valor EOF si se ha llegado al final del archivo. Otra forma de hacer dicha comprobación, es utilizar la función **feof** que devuelve verdadero si se ha llegado al final del archivo. Esta función es muy útil para ser utilizada en archivos binarios (donde la constante EOF no tiene el mismo significado) aunque se puede utilizar en cualquier tipo de archivo. Sintaxis:

```
int feof(FILE *pArchivo)
```

Así el código de lectura de un archivo para mostrar los caracteres de un texto, quedaría:

```
#include <stdio.h>
#include <conio.h>

int main(){
    FILE *archivo;
    char c=0;
    archivo=fopen("c:\\prueba.txt","r+");
    if(archivo!=NULL) {
        while(!feof(archivo)){
            c=fgetc(archivo);
            putchar(c);
        }
    }
    else{
        printf("Error");
    }
    fclose(archivo);
    return 0;
}
```

(7.3.3) leer y escribir strings

función fgets

Se trata de una función que permite leer textos de un archivo de texto. Su prototipo es:

```
char *fgets(char *texto, int longitud, FILE *pArchivo)
```

Esta función lee una cadena de caracteres del archivo asociado al puntero de archivos ***pArchivo*** y la almacena en el puntero ***texto.*** Lee la cadena hasta que llegue un salto de línea, o hasta que se supere la longitud indicada. La función devuelve un puntero señalando al texto leído o un puntero nulo (NULL) si la operación provoca un error.

Ejemplo (lectura de un archivo de texto):

```
#include <stdio.h>
#include <conio.h>

int main(){
    FILE *archivo;
    char texto[2000];
    archivo=fopen("c:\\prueba2.txt","r");
    if(archivo!=NULL) {
        fgets(texto,2000,archivo);
        while(!feof(archivo)){
           puts(texto);
           fgets(texto,2000,archivo);
        }
        fclose(archivo);
    }
    else{
        printf("Error en la apertura");
    }
    return 0;
}
```

En el listado el valor 2000 dentro de la funcón ***fgets*** tiene como único sentido, asegurar que se llega al final de línea cada vez que lee el texto (ya que ninguna línea del archivo tendrá más de 2000 caracteres).

función fputs

Es equivalente a la anterior, sólo que ahora sirve para escribir strings dentro del un archivo de texto. Prototipo:

```
int fputs(const char texto, FILE *pArchivo)
```

Escribe el texto en el archivo indicado. Además al final del texto colocará el carácter del salto de línea (al igual que hace la función **puts**). En el caso de que ocurra un error, devuelve EOF. Ejemplo (escritura en un archivo del texto introducido por el usuario en pantalla):

función fprintf

Se trata de la función equivalente a la función **printf** sólo que esta permite la escritura en archivos de texto. El formato es el mismo que el de la función **printf**, sólo que se añade un parámetro al principio que es el puntero al archivo en el que se desea escribir.

La ventaja de esta instrucción es que aporta una gran versatilidad a la hora de escribir en un archivo de texto.

Ejemplo. Imaginemos que deseamos almacenar en un archivo los datos de nuestros empleados, por ejemplo su número de empleado (un entero), su nombre (una cadena de texto) y su sueldo (un valor decimal). Entonces habrá que leer esos tres datos por separado, pero al escribir lo haremos en el mismo archivo separándolos por una marca de tabulación. El código sería:

```
#include <stdio.h>
int main(){
    int n=1; /*Número del empleado*/
    char nombre[80];
    double salario;
    FILE *pArchivo;

    pArchivo=fopen("c:\\prueba3.txt","w");
    if(pArchivo!=NULL){
        do{
            printf("Introduzca el número de empleado: ");
            scanf("%d",&n);
       /*Solo seguimos si n es positivo, en otro caso
       entenderemos que la lista ha terminado */
            if(n>0){
              printf("Introduzca el nombre del empleado: ");
              scanf("%s",nombre);
              printf("Introduzca el salario del empleado: ");
              scanf("%lf",&salario);
```

```
                fprintf(pArchivo,"%d\t%s\t%lf\n",
                                    n,nombre,salario);
            }
        }while(n>0);
        fclose(pArchivo);
    }
}
```

función fscanf

Se trata de la equivalente al **scanf** de lectura de datos por teclado. Funciona igual sólo que requiere un primer parámetro que sirve para asociar la lectura a un puntero de archivo. El resto de parámetros se manejan igual que en el caso de **scanf.**

Ejemplo (lectura de los datos almacenados con **fprintf**, los datos están separados por un tabulador).

```
#include <stdio.h>
#include <conio.h>

int main(){
    int n=1;
    char nombre[80];
    double salario;
    FILE *pArchivo;
    pArchivo=fopen("c:\\prueba3.dat","r");
    if(pArchivo!=NULL){
        while(!feof(pArchivo)){

fscanf(pArchivo,"%d\t%s\t%lf\n",&n,nombre,&salario);
          printf("%d\t%s\t%lf\n",n,nombre,salario);
        }
        fclose(pArchivo);
        return 0;
    }
}
```

función fflush

La sintaxis de esta función es:

```
int fflush(FILE *pArchivo)
```

Esta función vacía el buffer sobre el archivo indicado. Si no se pasa ningún puntero se vacían los búferes de todos los archivos abiertos. Se puede pasar también la corriente estándar de entrada ***stdin*** para vaciar el búfer de teclado (necesario si se leen caracteres desde el teclado, de otro modo algunas lecturas fallarían).

Esta función devuelve 0 si todo ha ido bien y la constante EOF en caso de que ocurriera un problema al realizar la acción.

(7.3.4) volver al principio de un archivo

La función **rewind** tiene este prototipo:

```
void rewind(FILE *pArchivo);
```

Esta función inicializa el indicador de posición, de modo que lo siguiente que se lea o escriba será lo que esté al principio del archivo. En el caso de la escritura hay que utilizarle con mucha cautela (sólo suele ser útil en archivos binarios).

(7.4) operaciones para uso con archivos binarios

(7.4.1) función *fwrite*

Se trata de la función que permite escribir en un archivo datos binarios del tamaño que sea. El prototipo es:

```
size_t fwrite(void *búfer, size_t bytes, size_t cuenta,FILE *p)
```

En ese prototipo aparecen estas palabras:

- ***size_t.*** Tipo de datos definido en el archivo **stdio.h,** normalmente equivalente al tipo **unsigned**. Definido para representar tamaños de datos.
- ***búfer.*** Puntero a la posición de memoria que contiene el dato que se desea escribir.
- ***bytes.*** Tamaño de los datos que se desean escribir (suele ser calculado con el operador **sizeof**)
- ***cuenta.*** Indica cuantos elementos se escribirán en el archivo. Cada elemento tendrá el tamaño en bytes indicado y su posición será contigua a partir de la posición señalada por el argumento búfer
- ***p.*** Puntero al archivo en el que se desean escribir los datos.

La instrucción **fwrite** devuelve el número de elementos escritos, que debería coincidir con el parámetro ***cuenta***, de no ser así es que hubo un problema al escribir.

Ejemplo de escritura de archivo. En el ejemplo se escriben en el archivo ***datos.dat*** del disco duro C registros con una estructura (llamada ***Persona***) que posee un texto para almacenar el nombre y un entero para almacenar la edad. Se escriben registros hasta un máximo de 25 o hasta que al leer por teclado se deje el nombre vacío:

```
#include <conio.h>
#include <stdio.h>

typedef struct {
  char nombre[25];
  int edad;
}Persona;

int main(){
  Persona per[25];
  int i=0;
  FILE *pArchivo;

  pArchivo=fopen("C:\\datos.dat","wb");
  if(pArchivo!=NULL){
    do{
      fflush(stdin); /* Se vacía el búfer de teclado */
      printf("Introduzca el nombre de la persona: ");
      gets(per[i].nombre);
      if(strlen(per[i].nombre)>0){
        printf("Introduzca la edad");
        scanf("%d",&(per[i].edad));
        fwrite(&per[i],sizeof(Persona),1,pArchivo);
        i++;
      }
    }while(strlen(per[i].nombre)>0 && i<=24);
    fclose(pArchivo);
  }
  else{
    printf("Error en la apertura del archivo");
  }
}
```

La instrucción **fwrite** del ejemplo, escribe el siguiente elemento leído del cual recibe su dirección, tamaño (calculado con **sizeof**) se indica que sólo se escribe

un elemento y el archivo en el que se guarda (el archivo asociado al puntero ***pArchivo***).

(7.4.2) función *fread*

Se trata de una función absolutamente equivalente a la anterior, sólo que en este caso la función lee del archivo. Prototipo:

```
size_t fread(void *búfer, size_t bytes, size_t
                                        cuenta,FILE *p)
```

El uso de la función es el mismo que **fwrite**, sólo que en este caso lee del archivo. La lectura es secuencial se lee hasta llegar al final del archivo. La instrucción **fread** devuelve el número de elementos leídos, que debería coincidir con el parámetro ***cuenta***, de no ser así es que hubo un problema o es que se llegó al final del archivo (el final del archivo se puede controlar con la función **feof** descrita anteriormente).

Ejemplo (lectura del archivo **datos.dat** escrito anteriormente):

```
#include <conio.h>
#include <stdio.h>
typedef struct {
  char nombre[25];
  int edad;
}Persona;
int main(){
  Persona aux;
  FILE *pArchivo;
  pArchivo=fopen("C:\\datos.dat","rb");
  if(pArchivo!=NULL){
    /* Se usa lectura adelantada, de otro modo
    el último dato sale repetido */
    fread(&aux,sizeof(Persona),1,pArchivo);
    while(!feof(pArchivo)){
      printf("Nombre: %s, Edad: %d\n",aux.nombre,
aux.edad);
      fread(&aux,sizeof(Persona),1,pArchivo);
    }
    fclose(pArchivo);
  }
  else{
    printf("Error en la apertura del archivo");
  }}
```

(7.5) uso de archivos de acceso directo

(7.5.1) función *fseek*

Los archivos de acceso directo son aquellos en los que se puede acceder a cualquier parte del archivo sin pasar por las anteriores. Hasta ahora todas las funciones de proceso de archivos vistas han trabajado con los mismos de manera secuencial.

En los archivos de acceso directo se entiende que hay un indicador de posición en los archivos que señala el dato que se desea leer o escribir. Las funciones **fread** o **fwrite** vistas anteriormente (o las señalas para leer textos) mueven el indicador de posición cada vez que se usan.

El acceso directo se consigue si se modifica ese indicador de posición hacia la posición deseada. Eso lo realiza la función **fseek** cuyo prototipo es:

```
int fseek(FILE * pArchivo, long bytes, int origen)
```

Esta función coloca el cursor en la posición marcada por el origen desplazándose desde allí el número de bytes indicado por el segundo parámetro (que puede ser negativo). Para el parámetro origen se pueden utilizar estas constantes (definidas en **stdio.h**):

- **SEEK_SET. I**ndica el principio del archivo.
- **SEEK_CUR.** Posición actual.
- **SEEK_END.** Indica el final del archivo.

La función devuelve cero si no hubo problemas al recolocar el indicador de posición del archivo. En caso contrario devuelve un valor distinto de cero:

Por ejemplo:

```
typedef struct {
  char nombre[25];
  int edad;
}Persona;

int main(){
  Persona aux;
  FILE *pArchivo;

  pArchivo=fopen("C:\\datos.dat","rb");
  if(pArchivo!=NULL){
    fseek(pArchivo,3*sizeof(Persona),SEEK_SET);
    fread(&aux,sizeof(Persona),1,pArchivo);
    printf("%s, %d años",aux.nombre,aux.edad);
    fclose(pArchivo);
  }
}
```

El ejemplo anterior muestra por pantalla a la cuarta persona que se encuentre en el archivo (***3*sizeof(Persona)*** devuelve la cuarta persona, ya que la primera está en la posición cero).

(7.5.2) función *ftell*

Se trata de una función que obtiene el valor actual del indicador de posición del archivo (la posición en la que se comenzaría a leer con una instrucción del lectura). En un archivo binario es el número de byte en el que está situado el cursor desde el comienzo del archivo. Prototipo:

```
long ftell(FILE *pArchivo)
```

Por ejemplo para obtener el número de registros de un archivo habrá que dividir el tamaño del mismo entre el tamaño en bytes de cada registro. Ejemplo:

```
/* Estructura de los registros almacenados en
el archivo*/
typedef struct{
    int n;
    int nombre[80];
    double saldo;
}Movimiento;
```

```
int main(){
    FILE *f=fopen("movimientos3.bin","rb");
    int nReg;/*Guarda el número de registros*/

    if(f!=NULL){
        fseek(f,0,SEEK_END);
        nReg=ftell(f)/sizeof(Movimiento);
        printf("N° de registros en el archivo =
%d",nReg);
        getch();
    }
    else{
        printf("Error en la apertura del archivo");
    }
}
```

En el caso de que la función **ftell** falle, da como resultado el valor -1.

(7.5.3) funciones fgetpos y fsetpos

Ambas funciones permiten utilizar marcadores para facilitar el trabajo en el archivo. Sus prototipos son:

```
int fgetpos(FILE *pArchivo, fpos_t *posicion);
int fsetpos(FILE *pArchivo, fpos_t *posicion);
```

La primera almacena en el puntero ***posición*** la posición actual del cursor del archivo (el indicador de posición), para ser utilizado más adelante pos **fsetpos** para obligar al programa a que se coloque en la posición marcada.

En el caso de que todo vaya bien ambas funciones devuelven cero. El tipo de datos **fpos_t** está declarado en la librería ***stdio.h*** y normalmente se corresponde a un número **long**. Es decir normalmente su declaración es:

```
typedef long fpos_t;
```

Aunque eso depende del compilador y sistema en el que trabajemos. Ejemplo de uso de estas funciones:

```
FILE *f=fopen("prueba.bin","rb+");
fpos_t posicion;
if(f!=NULL){
  ..... /*operaciones de lectura o de manipulación*/
  fgetpos(f,&posicion); /*Captura de la posición
                                          actual*/
  ... /*operaciones de lectura o manipulación */
  fsetpos(f,&posicion); /*El cursor vuelve a la posición
                          capturada */
```

(7.6) ejemplos de archivos de uso frecuente

(7.6.1) archivos de texto delimitado

Se utilizan muy habitualmente. En ellos se entiende que cada registro es una línea del archivo. Cada campo del registro se separa mediante un carácter especial, como por ejemplo un tabulador.

Ejemplo de contenido para un archivo delimitado por tabuladores:

```
1 Pedro     234.430000
2 Pedro     45678.234000
3 Pedro     123848.321000
5 Javier    34120.210000
6 Alvaro    324.100000
7 Alvaro    135.600000
8 Javier    123.000000
9 Pedro     -5.000000
```

La escritura de los datos se puede realizar con la función **fprintf**, por ejemplo:

```
fprintf("%d\t%s\t\%lf",numMov,nombre,saldo);
```

La lectura se realizaría con **fscanf:**

```
fscanf("%d\t%s\t\%lf",&numMov,nombre,&saldo);
```

Sin embargo si el delimitador no es el tabulador, podríamos tener problemas. Por ejemplo en este archivo:

```
AR,6,0.01
AR,7,0.01
AR,8,0.01
AR,9,0.01
AR,12,0.02
AR,15,0.03
AR,20,0.04
AR,21,0.05
BI,10,0.16
BI,20,0.34
BI,38,0.52
BI,57,0.77
```

La escritura de cada campo se haría con:

```
fprintf("%s,%d,%lf",tipo,modelo,precio);
```

Pero la instrucción:

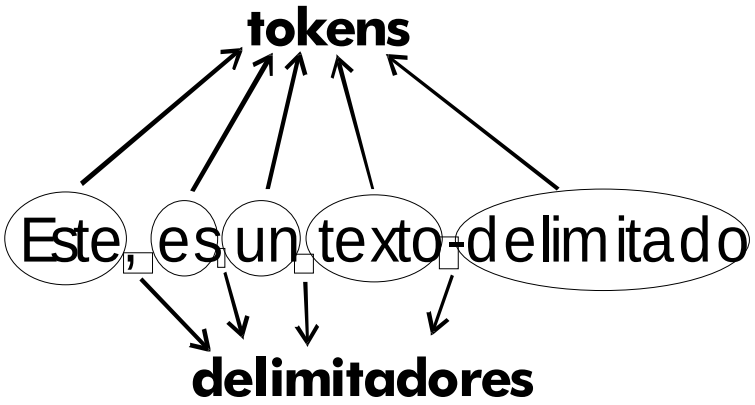
```
fscanf("%s,%d,%lf",tipo,&modelo,&precio);
```

da error, al leer ***tipo*** se lee toda la línea, no sólo hasta la coma.

La solución más recomendable para leer de texto delimitado es leer con la instrucción **fgets** y guardar la lectura en una sola línea. Después se debería utilizar la función **strtok**.

Esta función permite extraer el texto delimitado por caracteres especiales. A cada texto delimitado se le llama **token.** Para ello utiliza la cadena sobre la que se desean extraer las subcadenas y una cadena que contiene los caracteres delimitadores. En la primera llamada devuelve el primer texto delimitado.

El resto de llamadas a **strtok** se deben hacer usando NULL en el primer parámetro. En cada llamada se devuelve el siguiente texto delimitado. Cuando ya no hay más texto delimitado devuelve **NULL**.



En el diagrama anterior la frase "Este, es un texto-delimitado" se muestra la posible composición de ***tokens*** y delimitadores del texto. Los delimitadores se deciden a voluntad y son los caracteres que permiten separar cada ***token.***

Ejemplo de uso:

```
int main(){
   char texto[] = "Texto de ejemplo. Utiliza, varios delimitadores\n\n";
   char delim[] = " ,.";
   char *token;

   printf("Texto inicial: %s\n", texto);

   /* En res se guarda el primer texto delimitado (token) */
   token = strtok( texto, delim);
   /* Obtención del resto de tokens (se debe usar NULL
   en el primer parámetro)*/
   do{
      printf("%s\n", token);
      token=strtok(NULL,delim);
   }while(token != NULL );
}
```

Cada palabra de la frase sale en una lína separada.

Para leer del archivo anterior (el delimitado con comas, el código sería):

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

/* Se encarga de transformar una línea del archivo de
texto
en los datos correspondientes. Para ello extrae los
tokens
y los convierte al tipo adecuado*/
void extraeDatos(char *linea, char *tipo, int *modelo,
            double *precio)  {
    char *cadModelo, *cadPrecio;
    strcpy(tipo,strtok(linea,","));
    cadModelo=strtok(NULL,",");
    *modelo=atoi(cadModelo);
    cadPrecio=strtok(NULL,",");
    *precio=atof(cadPrecio);
}

int main(){
    FILE *pArchivo=fopen("piezas.txt","r");
    char tipo[3];
    char linea[2000];
    int modelo;
    double precio;

    if(pArchivo!=NULL){
        fgets(linea,2000,pArchivo);
        while(!feof(pArchivo)){
            extraeDatos(linea,&tipo,&modelo,&precio);
            printf("%s %d %lf\n",tipo,modelo,precio);
            fgets(linea,2000,pArchivo);
        }
        fclose(pArchivo);
    }
    getch();
}
```

(7.6.2) archivos de texto con campos de anchura fija

Es otra forma de grabar registros de datos utilizando ficheros de datos. En este caso el tamaño en texto de cada fila es el mismo, pero cada campo ocupa un tamaño fijo de caracteres.

Ejemplo de archivo:

```
AR6  0.01
AR7  0.01
AR8  0.01
AR9  0.01
AR12 0.02
AR15 0.03
AR20 0.04
AR21 0.05
BI10 0.16
BI20 0.34
BI38 0.52
```

En el ejemplo (con los mismos datos vistos en el ejemplo de texto delimitado), los dos primeros caracteres indican el tipo de pieza, los tres siguientes (que son números) el modelo y los seis últimos el precio (en formato decimal):

En todos los archivos de texto de tamaño fijo hay que leer las líneas de texto con la función **fgets**. Y sobre ese texto hay que ir extrayendo los datos de cada campo (para lo cual necesitamos, como es lógico, saber la anchura que tiene cada campo en el archivo).

Ejemplo (lectura de un archivo de texto de anchura fija llamado **Piezas2.txt** la estructura del archivo es la comentada en el ejemplo de archivo anterior):

```
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <conio.h>

/*Extrae de la cadena pasada como primer parámetro los c
caracteres que van de la posición inicio a la posición
fin
(ambos incluidos) y almacena el resultado en el puntero
subcad*/
```

```
void subcadena(const char *cad, char *subcad,
                int inicio, int fin){
    int i,j;
    for(i=inicio,j=0;i<=fin;i++,j++){
        subcad[j]=cad[i];
    }
    subcad[j]=0;/* Para finalizar el String */
}

/*Se encarga de extraer adecuadamente los campos de cada
línea del archivo */
void extraeDatos(char *linea, char *tipo,
                int *modelo, double *precio)  {
    char cadModelo[3], cadPrecio[6];
    /* Obtención del tipo */
    subcadena(linea, tipo,0,1);
    /*Obtención del modelo */
    subcadena(linea, cadModelo,2,5);
    *modelo=atoi(cadModelo);
    /* Obtención del precio */
    subcadena(linea,cadPrecio,6,11);
    *precio=atof(cadPrecio);
}
int main(){
    FILE *pArchivo=fopen("piezas.txt","r");
    char tipo[3];
    char linea[2000];
    int modelo;
    double precio;

    if(pArchivo!=NULL){
        fgets(linea,2000,pArchivo);
        while(!feof(pArchivo)){
            extraeDatos(linea,&tipo,&modelo,&precio);
            printf("%s %d %lf\n",tipo,modelo,precio);
            fgets(linea,2000,pArchivo);
        }
        fclose(pArchivo);
    }
}/*fin del main */
```

(7.6.3) ficheros maestro/detalle

…

(7.6.4) formatos binarios con registros de tamaño desigual

Los ficheros binarios explicados hasta ahora son aquellos que poseen el mismo tamaño de registro. De tal manera que para leer el quinto registros habría que posicionar el cursor de registros con **fseek** indicando la posición como **4*sizeof(Registro)** donde ***Registro*** es el tipo de estructura del registro que se almacena en el archivo.

Hay casos en los que compensa otro tipo de organización en la que los registros no tienen el mismo tamaño.

Imaginemos que quisiéramos almacenar este tipo de registro:

```
typedef struct{
  char nombre[60];
  int edad;
  int curso;
} Alumno;
```

Un alumno puede tener como nombre un texto que no llegue a 60 caracteres, sin embargo al almacenarlo en el archivo siempre ocupará 60 caracteres. Eso provoca que el tamaño del archivo se dispare. En lugar de almacenarlo de esta forma, el formato podría ser así:

Tamaño del texto **(int)**	Nombre **(char [])**	Edad **(int)**	Curso **(int)**

El campo nombre ahora es de longitud variable, por eso hay un primer campo que nos dice qué tamaño tiene este campo en cada registro. Como ejemplo de datos:

7	Alberto	14	1
11	María Luisa	14	1
12	Juan Antonio	15	1

Los registros son de tamaño variable, como se observa con lo que el tamaño del archivo se optimiza. Pero lo malo es que la manipulación es más compleja y hace imposible el acceso directo a los registros (es imposible saber cuando comienza el quinto registro).

Para evitar el problema del acceso directo, a estos archivos se les acompaña de un **archivo de índices**, el cual contiene en cada fila dónde comienza cada registro.

Ejemplo:

Nº reg	Pos en bytes
1	0
2	20
3	46

En ese archivo cada registro se compone de dos campos (aunque el primer campo se podría quitar), el primero indica el número de registro y el segundo la posición en la que comienza el registro.

En los ficheros con índice lo malo es que se tiene que tener el índice permanentemente organizado.

(7.6.5) ficheros indexados

Una de los puntos más complicados de la manipulación de archivos es la posibilidad de mantener ordenados los ficheros. Para ello se utiliza un fichero auxiliar conocido como índice.

Cada registro tiene que poseer una clave que es la que permite ordenar el fichero. En base de esa clave se genera el orden de los registros. Si no dispusiéramos de índice, cada vez que se añade un registro más al archivo habría que regenerar el archivo entero (con el tiempo de proceso que consume esta operación).

Por ello se prepara un archivo separado donde aparece cada clave y la posición que ocupan en el archivo. Al añadir un registro se añade al final del archivo; el que sí habrá que reorganizar es el fichero de índices para que se actualicen, pero cuesta menos organizar dicho archivo ya que es más corto. Ejemplo de fichero de datos:

Fecha movimiento (Clave)	Importe	Id Cuenta	Concepto	Posición en el archivo
12/4/2005 17:12:23	1234.21	23483484389	Nómina	0
11/4/2005 9:01:34	-234.98	43653434663	Factura del gas	230
12/4/2005 17:45:21	-12.34	23483484389	Compras	460
12/4/2005 12:23:21	987.90	43653434663	Nómina	690
11/4/2005 17:23:21	213.45	34734734734	Devolución hacienda	920

Fichero Índice:

Fecha movimiento (Clave)	Inicio en bytes
11/4/2005 9:01:34	230
11/4/2005 17:23:21	920
12/4/2005 17:12:23	0
12/4/2005 12:23:21	690
12/4/2005 17:45:21	460

En el índice las claves aparecen ordenadas, hay un segundo campo que permite indicar en qué posición del archivo de datos se encuentra el registro con esa clave. El problema es la reorganización del archivo de índices, que se tendría que hacer cada vez que se añade un registro para que aparezca ordenado.

(7.6.6) otros formatos binarios de archivo

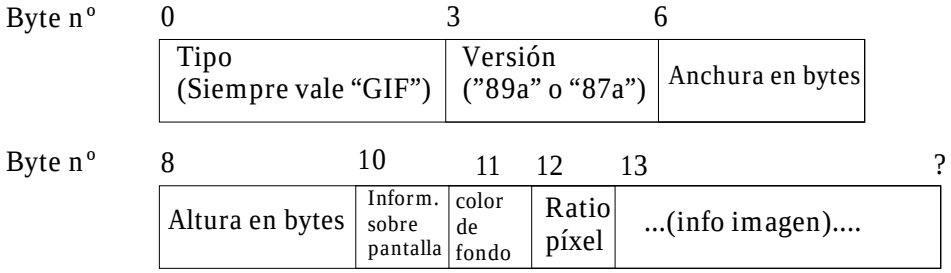
No siempre en un archivo binario se almacenan datos en forma de registros. En muchas ocasiones se almacena otra información. Es el caso de archivos que almacenan música, vídeo, animaciones o documentos de una aplicación.

Para que nosotros desde un programa en C podamos sacar información de esos archivos, necesitamos conocer el **formato de ese archivo.** Es decir como hay que interpretar la información (siempre binaria) de ese archivo. Eso sólo es posible si conocemos dicho formato o si se trata de un formato de archivo que hemos diseñado nosotros.

Por ejemplo en el caso de los archivos **GIF**, estos sirven para guardar imágenes. Por lo que lo que guardan son píxeles de colores. Pero necesitan guardar otra información al inicio del archivo conocida como cabecera.

Esta información tiene este formato:

Cabecera de los archivos GIF



Así para leer la anchura y la altura de un determinado archivo GIF desde un programa C y mostrar esa información por pantalla, habría que:

```
int main(){
    FILE *pArchivo;
    int ancho=0, alto=0; /* Aquí se almacena el resultado
    pArchivo=fopen("archivo.gif","rb");
    if(pArchivo!=NULL) {
    /* Nos colocamos en el sexto byte, porque ahí está la
    información sobre la anchura y después la altura*/
        fseek(pArchivo,6,SEEK_SET);
        fread(&ancho,2,1,pArchivo);
        fread(&alto,2,1,pArchivo);
        printf("Dimensiones: Horizontal %d, Vertical
%d\n",
                                ancho,alto);
    }
}
```

En definitiva para extraer información de un archivo binario, necesitamos conocer exactamente su formato.

(Unidad 8) Programación en Java

(8.1) introducción

(8.1.1) historia de Java y relación con el lenguaje C

A pesar de la potencia del lenguaje C, que permite realizar cualquier tipo de aplicación, lo cierto es que según han ido apareciendo lenguajes nuevos, se ha ido demostrando que este lenguaje no era el apropiado para crear muchos tipos de aplicaciones.

En especial el lenguaje C siempre ha tenido el problema de no dar excesivas facilidades para organizar código compuesto de miles y miles de líneas. La aparición de la programación orientada a objetos supuso la asimilación de un nuevo tipo de análisis de programas que facilitaba en gran medida el mantenimiento, modificación y creación de aplicaciones.

En este sentido el lenguaje **C++** se sigue considerando como uno de los más potentes. Éste lenguaje (que desbancó al anterior) aportó las siguientes ventajas:

- Añadir soporte para objetos (**POO**)
- Los creadores de compiladores crearon librerías de clases de objetos (como **MFC**[3] por ejemplo) que facilitaban el uso de código ya creado para las nuevas aplicaciones.
- Mejorar la sintaxis del lenguaje C
- Incluir todo lo bueno del lenguaje C.

C++ pasó a ser el lenguaje de programación más popular a principios de los 90 (sigue siendo uno de los lenguajes más utilizados y sigue siendo considerado por muchos programadores el más potente).

[3] **Microsoft Foundation Classes,** librería creada por Microsoft para facilitar la creación de programas para el sistema Windows.

En los años 90 la aparición de Internet propició el éxito del lenguaje Java ya que era más interesante para crear aplicaciones que utilizarán la red. La idea partió de la empresa **Sun Microsystems** que en 1991 crea el lenguaje **Oak (**de la mano del llamado **proyecto Green**), que en breve (y por que ya había otro lenguaje con ese nombre) se le llamó Java**.**

Inicialmente se pretendía crear un sistema de televisión interactiva y sólo se llegó a utilizar de forma interna por Sun. Viendo las posibilidades que el lenguaje ofrecía como sustituto a C++ en muchos aspectos, y en especial como lenguaje de creación de elementos para la web, se lanzó al mercado y se estandarizó su sintaxis. Su éxito fue fulgurante.

(8.1.2) ventajas de Java

Cuando se creo Java se pretendían lograr estos objetivos:

(1) Debería usar una metodología de programación orientada a objetos.

(2) Debería permitir la ejecución de un mismo programa en múltiples plataformas

(3) Debería incluir facilidades para trabajo en red.

(4) Debería crear aplicaciones para ejecutar código en sistemas remotos de forma segura.

(5) Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, en especial C++.

Los cinco objetivos fueron logrados, además como características del lenguaje se puede añadir:

- Su sintaxis es similar a C y C++
- No hay punteros (lo que le hace más seguro y más estructurado al leer)
- Lenguaje totalmente orientado a objetos
- Muy preparado para crear aplicaciones que se ejecuten en redes TCP/IP
- Implementa excepciones de forma nativa
- Elimina el espacio gastado en memoria que el programa ha dejado (la basura digital).
- Es interpretado (lo que acelera su ejecución remota, aunque provoca que las aplicaciones Java se ejecuten más lentamente que las C++ en un ordenador local).
- Permite multihilos.
- Admite firmas digitales. Es más seguro.
- Tipos de datos y control de sintaxis más rigurosa
- Es independiente de la plataforma

- Es distribuido. Un programa Java puede estar ejecutando rutinas en un servidor remoto del cliente sin que este lo perciba.

La última ventaja (quizá la más importante) se consigue ya que el código Java no se compila, sino que se **precompila,** de tal forma que se crea un código intermedio que no es ejecutable. Para ejecutarle hace falta pasarle por un intérprete que va ejecutando cada línea. Ese intérprete suele ser la **máquina virtual** de Java. La idea es programar para la máquina virtual y ésta es un programa que se puede ejecutar en cualquier plataforma (Linux, Windows, Mac,…) lo que permite que Java sea un lenguaje mucho más portable.

(8.1.3) características de Java

bytecodes

Un programa C o C++ es totalmente ejecutable y eso hace que no sea independiente de la plataforma y que su tamaño normalmente se dispare ya que dentro del código final hay que incluir las librearías de la plataforma

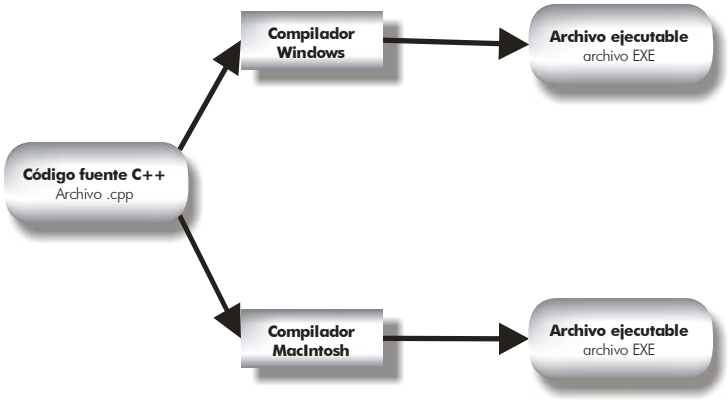


Ilustración 12, Proceso de compilación de un programa C++

Los programas Java no son ejecutables, no se compilan como los programas en C o C++. En su lugar son interpretados por una aplicación conocida como la **máquina virtual de Java** (JVM). Gracias a ello no tienen porque incluir todo el código y librerías propias de cada sistema.

Previamente el código fuente en Java se tiene que compilar generando un código (que no es directamente ejecutable) previo conocido como **bytecode** o **J-code**. Ese código (generado normalmente en archivos con extensión **class**) es el que es ejecutado por la máquina virtual de Java que interpreta las instrucciones generando el código ejecutable de la aplicación

La máquina virtual de Java, además es un programa muy pequeño y que se distribuye gratuitamente para prácticamente todos los sistemas operativos.

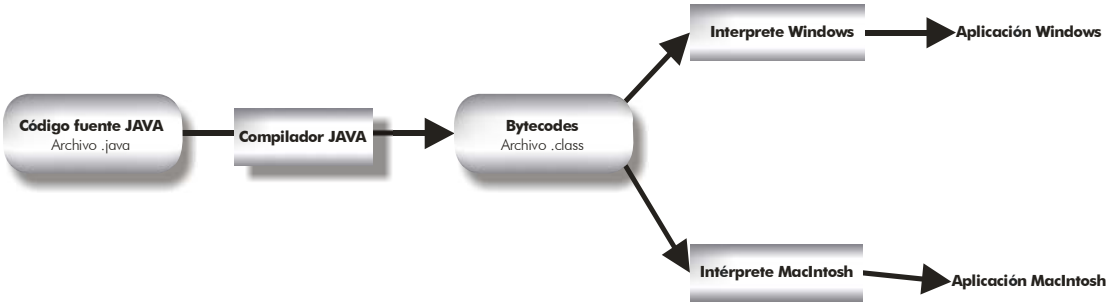


Ilustración 13, Proceso de compilación de un programa Java

En Java la unidad fundamental del código es la **clase**. Son las clases las que se distribuyen en el formato ***bytecode*** de Java. Estas clases se cargan dinámicamente durante la ejecución del programa Java.

A este método de ejecución de programas en tiempo real se le llama ***Just in Time*** (**JIT**).

seguridad

Al interpretar el código, la JVM puede delimitar las operaciones peligrosas, con lo cual la seguridad es fácilmente controlable. Además, Java elimina las instrucciones dependientes de la máquina y los **punteros** que generaban terribles errores e inseguridades en C y C++. Tampoco se permite el acceso directo a memoria.

La primera línea de seguridad de Java es un **verificador del bytecode** que permite comprobar que el comportamiento del código es correcto y que sigue las reglas de seguridad de Java. Normalmente los compiladores de Java no pueden generar código que se salte las reglas de seguridad de Java. Pero un programador ***malévolo*** podría generar artificialmente código ***bytecode*** que se salte las reglas. El verificador intenta eliminar esta posibilidad.

Hay un segundo paso que verifica la seguridad del código que es el **verificador de clase** que es el programa que proporciona las clases necesarias al código. Lo que hace es asegurarse que las clases que se cargan son realmente las del sistema original de Java y no clases creadas reemplazadas artificialmente.

Finalmente hay un **administrador de seguridad** que es un programa configurable que permite al usuario indicar niveles de seguridad a su sistema para todos los programas de Java.

Hay también una forma de seguridad relacionada con la confianza. Esto se basa es saber que el código Java procede de un sitio de confianza y no de una fuente no identificada. En Java se permite añadir firmas digitales al código para verificar al autor del mismo.

tipos de aplicaciones Java

applet

Son programas Java pensados para ser colocados dentro de una página web. Pueden ser interpretados por cualquier navegador con capacidades Java. Estos programas se insertan en las páginas usando una etiqueta especial (como también se insertan vídeos, animaciones flash u otros objetos).

Los applets son programas independientes, pero al estar incluidos dentro de una página web las reglas de éstas le afectan. Normalmente un applet sólo puede actuar sobre el navegador.

Hoy día mediante applets se pueden integrar en las páginas web aplicaciones multimedia avanzadas (incluso con imágenes 3D o sonido y vídeo de alta calidad)

aplicaciones de consola

Son programas independientes al igual que los creados con los lenguajes tradicionales. Utilizan la consola del sistema (la entrada y salida de datos estándar) para mostrar o recoger la información.

aplicaciones gráficas

Aquellas que utilizan las clases con capacidades gráficas (como **awt** o **swing** por ejemplo) a fin de crear programas con ventanas.

servlets

Son aplicaciones que se ejecutan en un servidor de aplicaciones web y que como resultado de su ejecución resulta una página web. La diferencia con las applets reside en que en este caso el cliente no tiene que instalar nada, es el servidor el que tendrá instalado el software que traduce Java a forma de página web

JavaBean

Componentes que son fácilmente reutilizables en otras aplicaciones

aplicaciones para dispositivos móviles

Cada vez más populares gracias al uso de librerías construidas con ese fin. Se las llama también ***midlets***

(8.1.4) empezar a trabajar con Java

JRE

El **Java Runtime Environment** o **JRE** es el entorno de ejecución de programas Java. Se trata del paquete completo de software que permite traducir cualquier programa Java.

Un usuario que quiera ejecutar aplicaciones Java requiere este software.

el kit de desarrollo Java (SDK)

Para escribir en Java hacen falta los programas que realizan el precompilado y la interpretación del código. El kit de desarrollo originalmente conocido como ***Java Development Kit*** o **JDK** y ahora llamado **SDK** (***Standard Development Kit***,

SDK) son los programas que permiten crear aplicaciones para la plataforma estándar de Java (llamada **J2SE**).

El SDK está formado por aplicaciones de línea de comandos que permiten generan y ejecutar el código precompilado Java. Hay programas de todo tipo en el kit (generadores de documentación, tratamiento de errores, compresores,….).

Este kit es imprescindible para programar en Java y está disponible en la dirección http://java.sun.com

plataformas

Actualmente hay tres ediciones de la plataforma Java 2

J2SE

Se denomina así al entorno de Sun relacionado con la creación de aplicaciones y applets en lenguaje Java. la última versión del kit de desarrollo de este entorno es el J2SE 1.5.6 (llamado J2SE 5.0)

J2EE

Pensada para la creación de aplicaciones Java empresariales y del lado del servidor. Su última versión es la 1.4

J2ME

Pensada para la creación de aplicaciones Java para dispositivos móviles.

versiones de Java

Desde su aparición en 1995, Java ha experimentado varias versiones del lenguaje (y de las herramientas que le dan soporte)

Java 1.0 (JDK 1.0)

Fue la primera versión de Java y propuso el marco general en el que se desenvuelve Java. Está oficialmente obsoleto, pero hay todavía muchos dispositivos que utilizan software de esta versión

Java 1.1 (JDK 1.1)

Mejoró la versión anterior incorporando como mejoras:

- **JDBC**, paquete de conexión con bases de datos.
- Mejoras en la librería gráfica **AWT**
- **RMI**, librería para llamadas remotas a través de la web
- **JavaBeans,** componentes reutilizables en distintas aplicaciones Java.

Java 1.2 o Java 2 (J2SE 1.2)

Apareció en Diciembre de 1998 al aparecer el JDK 1.2. Incorporó tantas mejoras que desde entonces se llamó **Java 2** al lenguaje compatible con esta versión. El JDK ahora es llamado SDK. Ese mismo año se desarrolla **J2EE** y por ello al Java estándar se le llamará **J2SE** (***Java 2 Standard Edition***).

Entre sus mejoras:

- **JFC** (***Java Foundation classes***) que incluye los paquetes:
  - **Swing**
  - **Java Media** y otras librerías avanzadas.
- **Servlets**
- **Java cards**
- compilador **JIT**
- **Java IDI** para la compatibilidad con **CORBA,** interfaz para crear aplicaciones distribuidas por red
- Clases para colecciones, implementan estructuras dinámicas avanzadas

Java 1.3 (J2SE 1.3)

Solucionó errores y problemas de la versión previa. Incorporó el compilador **JIT (*Just In Time)*** de **HotSpot** (más potente). Otras mejoras:

- **RMI** compatible con **CORBA**
- **Java Sound**
- **JNDI**, ***Java Naming and Directory Interface***, librería para utilizar nombres en Java

Java 1.4 (J2SE 1.4)

Llamado ***Merlin*** incorporó:

- Soporte de aserciones (**asserts**)
- Soporte de expresiones regulares
- **Java NIO** (***new Input Output***) la API de entrada/salida
- **JAXP** para soporte **XML** en Java.
- Mejoras en la seguridad.

Java 1.5 (J2SE 1.5, Java 5.0)

Llamado ***Tiger***. Se le cambió la denominación y ahora se llama **Java 5.0**. Añadió numerosas mejoras:

- Metadatos (datos que describen otros datos)
- Tipos genéricos
- Mejoras en las conversiones entre tipos
- Plantillas.
- Bucle **for** para recorrido de listas
- Enumeraciones
- Parámetros variables para las funciones.
- Método ***printf*** (como el del lenguaje C)

Java 1.6 (Java SE 6.0)

Llamado ***Mustang***. Ya no llama J2SE al lenguaje, sino Java SE (***Java Standard Edition***). Incorpora:

- Mejor comunicación con otros lenguajes **PHP, Python, Ruby**,…
- Mejoras en rendimiento
- Mejoras en creación de servicios web (**JAX-WS 2.0, JAXB 2.0, STAX y JAXP**)
- Uso de **JavaScript** dentro de Java **(Rhino).**

Java 1.7 (Java SE 7.0)

Llamado ***Dolphin***, verá la luz en 2008

entornos de trabajo

El código en Java se puede escribir en cualquier editor de texto. Y para compilar el código en bytecodes, sólo hace falta descargar la versión del JDK deseada. Sin embargo, la escritura y compilación de programas así utilizada es un poco incomoda. Por ello numerosas empresas fabrican sus propios entornos de edición, algunos incluyen el compilador y otras utilizan el propio JDK de Sun.

- **NetBeans**. Entorno gratuito de código abierto fabricado por Sun para la generación de código en diversos lenguajes (especialmente pensado para Java). Contiene prácticamente todo lo que se suele pedir a un IDE, editor avanzado de código, depurador, diversos lenguajes, extensiones de todo tipo (CORBA, Servlets,...). Incluye además un servidor de aplicaciones **Tomcat** para probar aplicaciones de servidor. Se descarga en www.netbeans.org.
- **Eclipse**. Es un entorno completo de código abierto que admite numerosas extensiones (incluido un módulo para J2EE) y posibilidades. Es uno de los más utilizados por su compatibilidad con todo tipo de aplicaciones Java y sus interesantes opciones de ayuda al escribir código. Es el entorno que más crece ya que existen decenas de proyectos pensados para mejorar la funcionalidad de Eclipse.
- **Sun ONE Studio**. Entorno para la creación de aplicaciones Java creado por la propia empresa Sun a partir de **NetBeans** (casi es clavado a éste). la versión ***Community Edition*** es gratuita (es más que suficiente), el resto son de pago (incorporan herramientas de productividad mejorada). Está basado en el anterior. Antes se le conocía con el nombre **Forte for Java**. Está relacionado con los servidores ONE de Java.
- **Visual Studio y Visual Studio.NET**. El entorno de desarrollo de Microsoft. Es uno de los mejores, pero sólo es interesante si se desea programar en la versión Java de Microsoft (no estándar)
- **Visual Cafe**. Otro entorno veterano completo de edición y compilado. Bastante utilizado. Es un producto comercial de la empresa Symantec.

- **JBuilder**. Entorno completo creado por la empresa Borland (famosa por su lenguaje Delphi) para la creación de todo tipo de aplicaciones Java, incluidas aplicaciones para móviles. Es uno de los favoritos (es de pago).
- **JDeveloper**. De Oracle. Entorno completo para la construcción de aplicaciones Java y XML. Ideal para programadores de Oracle.
- **Visual Age**. Entorno de programación en Java desarrollado por IBM. Es de las herramientas más veteranas. Actualmente en desuso.

(8.2) escritura de programas Java

(8.2.1) codificación del texto

Todos el código fuente Java se escriben en documentos de texto con extensión **.java**. Al ser un lenguaje para Internet, la codificación de texto debía permitir a todos los programadores de cualquier idioma escribir ese código. Eso significa que Java es compatible con la codificación **Unicode**[4].

En la práctica significa que los programadores que usen lenguajes distintos del inglés no tendrán problemas para escribir símbolos de su idioma (como por ejemplo la ***ñ*** o la ***á*** en el caso del castellano). En definitiva cualquier identificador dentro de un programa Java puede llevar esos símbolos. También el texto que utilice símbolos nacionales será perfectamente visible en pantalla.

(8.2.2) notas previas

Los archivos con código fuente en Java deben guardarse con la extensión .java. Como se ha comentado cualquier editor de texto basta para crear código Java. Algunos detalles importantes son:

- En Java (como en C) hay diferencia entre mayúsculas y minúsculas.
- Cada línea de código debe terminar con **;**
- Los comentarios; si son de una línea debe comenzar con **//** y si ocupan más de una línea deben comenzar con **/*** y terminar con ***/**

```
/* Comentario
de varias líneas */
//Comentario de una línea
```

- A veces se marcan bloques de código, los cuales comienza con **{** y terminan con **}** (al igual que en C). Los bloques sirven para agrupar varias líneas de código.

[4] Para más información acudir a http://www.unicode.org

(8.2.3) el primer programa en Java

El código más sencillo en Java podría ser aquel que escribe en pantalla un texto (como el famoso ***Hola mundo***). Por ejemplo:

```
public class App
{
  public static void main(String[] args)
  {
       System.out.println("¡Mi primer programa!");
  }
}
```

Este código escribe "¡Mi primer programa!" en la pantalla. El archivo debería llamarse **App.java** ya que esa es la clase pública. El resto define el método **main** que es el que se ejecutará al lanzarse la aplicación. Ese método utiliza la instrucción que escribe en pantalla.

Aunque no es obligatorio, en Java los nombres de las clases (por ahora entenderemos que un programa es una clase, más adelante se matizará esta idea de forma más apropiada) deben tener la primera inicial en mayúsculas. Esto es una norma de ***buenas maneras*** al escribir código en Java (como dejar espacio a la izquierda cuando un código está dentro de una llave, por ejemplo) para facilitar su legibilidad.

(8.2.4) instrucción **import**

Hay código que se puede utilizar en los programas que realicemos en Java. Se importar clases de objetos que están contenidas, a su vez, en paquetes estándares.

Por ejemplo la clase **Date** es una de las más utilizadas, sirve para manipular fechas. Si alguien quisiera utilizar en su código objetos de esta clase, necesita incluir una instrucción que permita utilizar esta clase. La sintaxis de esta instrucción es:

```
import paquete.subpaquete.subsubapquete....clase
```

Esta instrucción se coloca arriba del todo en el código. Para la clase **Date** sería:

```
import java.util.Date
```

Lo que significa, importar en el código la clase **Date** que se encuentra dentro del paquete **util** que, a su vez, está dentro del gran paquete llamado **java**.

También se puede utilizar el asterisco en esta forma:

```
import java.util.*
```

Esto significa que se va a incluir en el código todas las clases que están dentro del paquete **util** de **java.**

(8.2.5) instrucción package

La primera instrucción de un programa Java habitualmente es la instrucción **package**. La sintaxis es la siguiente:

```
package nombrePaquete;
```

Ejemplo:

```
package misclases.utiles;
```

Con esto lo que estamos indicando es que la clase que estamos creando pertenece al paquete ***utiles*** que, a su vez, está dentro del paquete ***misclases***.

Un paquete se puede entender como una carpeta que contiene clases y/o más paquetes. La carpeta a la que se refiere el paquete puede tener cualquier ruta dentro del ordenador en el que se ejecuta el programa Java, pero para que esa ruta se considere raíz de paquetes, debe estar incluida en la variable de sistema **classpath**. Así suponiendo que ***misclases*** sea una carpeta dentro de la ruta ***C:\paquetes***. Entonces a la variable classpath (de manejo similar al path del sistema) hay que añadirla la ruta ***c:\paquetes***.

(8.3) compilación y ejecución de programas Java

(8.3.1) descargar del kit de desarrollo (JDK)

El kit de desarrollo de Java está disponible en la página web http://java.sun.com/jse/downloads

Al tiempo de escribir estos apuntes la última versión es la JDK 6u1 (***Java SE update 1***). Tras descargar el programa, basta ejecutarle y el software necesario para ejecutar Java estará disponible.

Normalmente (salvo que dispongamos otra cosa) el software se habrá instalado en la carpeta **Archivos de programa** del disco duro en el que está instalado Windows (en el caso de que sea Windows el Sistema Operativo que estemos utilizando). Dentro de la carpeta Java estará la carpeta del kit (en su última versión, se llama **jdk1.6.0_01**), y en ella estará la carpeta **bin** que contiene todos los programas necesarios para compilar código java.

Es muy interesante modificar la variable **path** del sistema para hacer que se busquen los programas de desarrollo java en la carpeta correspondiente. De esta forma podremos invocar a los compiladores y lanzadores del kit de desarrollo Java aunque nuestro programa esté en otra carpeta.

La modificación de la variable ***path*** se hace de esta forma (en Windows):

(1) Pulsar el botón derecho sobre Mi PC y elegir **Propiedades**

(2) Ir al apartado **Opciones avanzadas**

(3) Hacer clic sobre el botón **Variables de entorno**

(4) Añadir a la lista de la variable **Path** la ruta a la carpeta con los programas del JDK.

Para comprobar la variable path, podemos ir a la consola del sistema y escribir **path**. Ejemplo de contenido de la variable path:

```
PATH=C:\WINNT\SYSTEM32;C:\WINNT;C:\WINNT\SYSTEM32\WBEM;C:
\Archivos de programa\Microsoft Visual
Studio\Common\Tools\WinNT;C:\Archivos de
programa\Microsoft Visual
Studio\Common\MSDev98\Bin;C:\Archivos de
programa\Microsoft Visual Studio\Common\Tools;C:\Archivos
de programa\Microsoft Visual Studio\VC98\bin; C:\Archivos
de programa\Java\jdk1.6.0_01
```

(8.3.2) proceso de compilación y ejecución desde la línea de comandos

La compilación del código Java se realiza mediante el programa **javac** incluido en el software de desarrollo de Java. La forma de compilar es (desde la línea de comandos):

```
javac archivo.java
```

El resultado de esto es un archivo con el mismo nombre que el archivo java pero con la extensión **class**. Esto ya es el archivo con el código en forma de **bytecodes**. Es decir con el código precompilado.

Si es un código de consola se puede probar usando el programa **java** del kit de desarrollo. Sintaxis:

```
java archivoClass
```

Estos comandos hay que escribirlos desde la línea de comandos de en la carpeta en la que se encuentre el programa. Pero antes hay que asegurarse de que los programas del kit de desarrollo son accesibles desde cualquier carpeta del sistema. Para ello hay que comprobar que la carpeta con los ejecutables del kit de desarrollo está incluida en la variable de entorno **path** (como ya se explicó anteriormente).

(8.3.3) creación de javadocs desde la línea de comandos

Javadoc es una herramienta muy interesante del kit de desarrollo de Java para generar automáticamente documentación Java. genera documentación para paquetes completos o para archivos java. Su sintaxis básica es:

```
javadoc archivo.java o paquete
```

En el caso de indicar un archivo con código Java, se genera la documentación de ese archivo, si se indica un paquete se genera la documentación para todos los archivos (clases) del paquete).

Para ello necesitamos, previamente al uso del herramienta javadoc, crear comentarios especiales (***comentarios javadoc***) en el código. El funcionamiento de estos comentarios es el siguiente: Los comentarios que comienzan con los códigos **/**** se llaman comentarios de documento y serán utilizados por los programas de generación de documentación javadoc. En esos comentarios se pueden utilizar códigos HTML.

Ejemplo:

```
/** Esto es un comentario para probar el javadoc
* este texto aparecerá en el archivo HTML generado.
* <strong>Realizado en agosto 2003</strong>
*  @author Jorge Sánchez
* @version 1.0
*/
public class Prueba1 {
//Este comentario no aparecerá en el javadoc
  public static void main(String args[]){
        System.out.println("¡Mi segundo programa! ");
  }
}
```

El resultado de ese código tras usar javadoc es la página web:

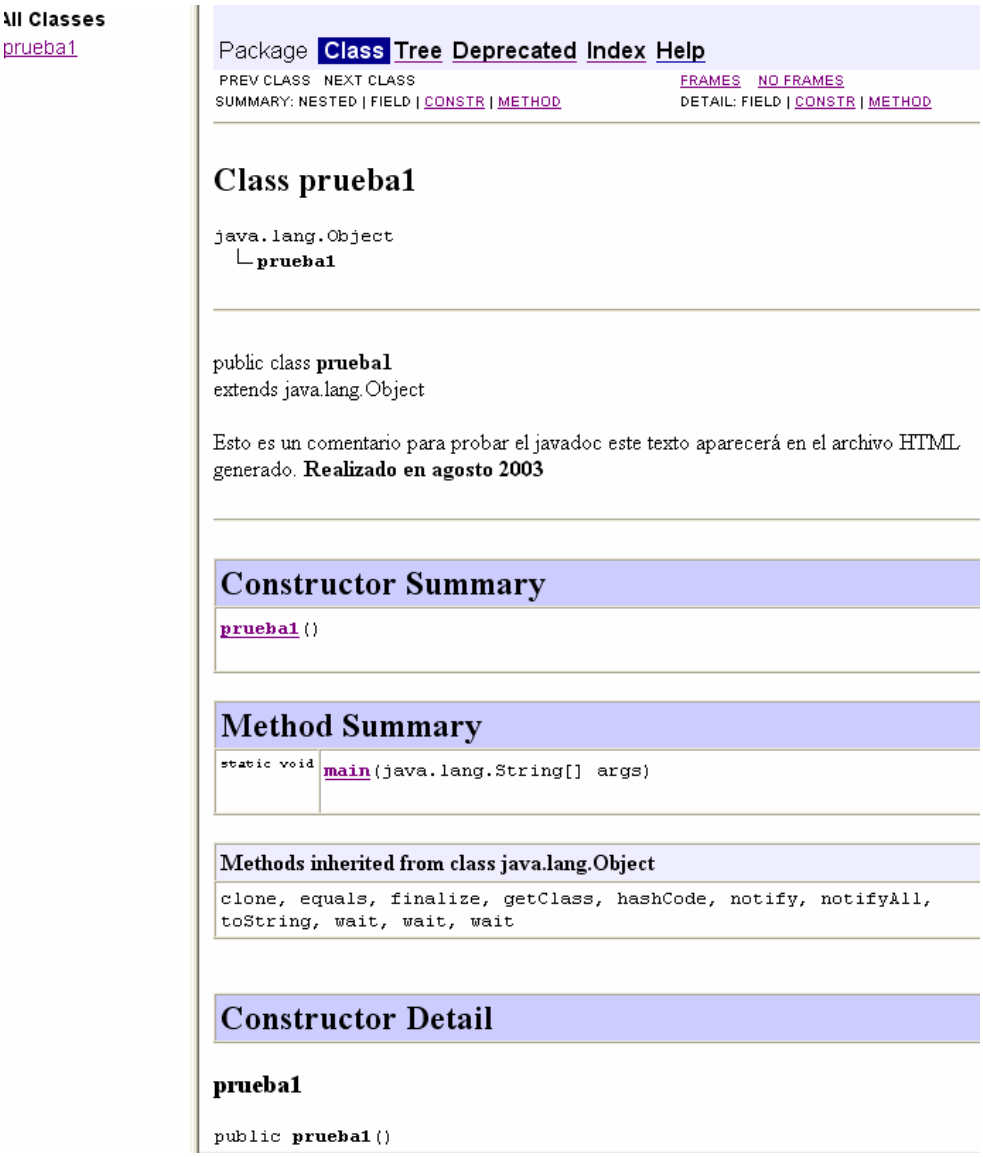


Ilustración 14, Página de documentación de un programa Java

Al inicio del programa se debe poner un comentario ***javadoc*** que contendrá lo que hace el programa y además puede tener las siguientes palabras clave:

- **@author**. Marca el autor del documento
- **@version.** Versión del programa

- **@see.** Para referenciar una dirección. Por ejemplo ***@see*** www.jorgesanchez.net
- **@throw.** Permite indicar las excepciones (errores) que puede producir el código
- **@param.** Permite documentar cómo se deben utilizar los parámetros de una función
- **@return.** Para documentar el resultado de una función.

(8.3.4) uso del entorno de desarrollo integrado Eclipse

introducción

Se trata de un software pensado para crear entornos de desarrollo (**IDEs**). Inicialmente concebido por **IBM** para suceder a su herramienta **Visual Age**, se trata de un software de código abierto regido por la llamada **Fundación Eclipse**; organismo sin ánimo de lucro.

Al ser mejorado y utilizado por una creciente y dinámica comunidad de usuarios, se ha convertido en la herramienta más utilizada para programar en Java.

La descarga del software Eclipse se hace desde la dirección www.eclipse.org desde el apartado **download**. La última versión en el momento de escribir estas líneas es el **SDK 3.2.2**. Más adelante se pueden instalar los complementos que permiten mejorar la utilidad de Eclipse. Por otro lado para Eclipse funcione correctamente debe de estar instalado el kit de desarrollo de Java (al menos en la versión 1.4).

Al descargar tendremos un archivo **zip**, descomprimiéndole donde queramos utilizar el programa ya tendremos Eclipse instalado (no se instala como una aplicación de Windows, no estará en el **Panel de Control** ni en el apartado **Programas**).

Por otro lado podremos descargar el kit de traducción al español de la dirección:

http://download.eclipse.org/eclipse/downloads/drops/L-3.2.1_Language_Packs-200609210945/index.php

o bien buscando la descarga de los ***Language Packs*** en la página de Eclipse. Tras descargar el pack de lenguaje, bastará con descomprimir el archivo en la misma carpeta en la que se instaló Eclipse.

uso básico de Eclipse

Cuando se arranca Eclipse por primera vez se nos pregunta la carpeta en la que se guardarán los proyectos. La sugerencia de Eclipse la podemos cambiar para elegir la ruta que deseemos.

Después se nos permite elegir si queremos ayuda o bien empezar a trabajar con el entorno de trabajo (en inglés ***workbench***).

Normalmente al ir hacia el entorno de trabajo, Eclipse aparece preparado en la perspectiva de Java. Podemos asegurarnos si elegimos **Ventana-Abrir Perspectiva-Java**.

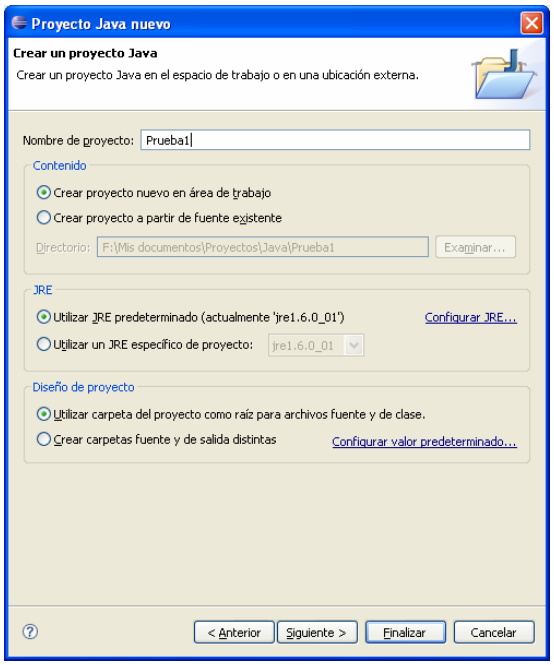
crear nuevos proyectos

Para trabajar en Java desde Eclipse, necesitamos crear un proyecto nuevo para poder empezar a trabajar. Eso se realiza de esta forma:

(1) Elegir **Archivo-Nuevo Proyecto**

(2) Elegir **Proyecto Java**

(3) En el cuadro siguiente elegir el nombre del proyecto y las casillas de abajo (nos permite elegir la versión de Java que utilizaremos, la carpeta de proyecto y la ruta que utilizaremos para nuestras clases)



(4) Pulsar siguiente y finalizar en la siguiente pantalla (salvo que se la quiera modificar

crear un paquete

Conviene que nuestras clases (nuestros archivos) estén dentro de un paquete, de otro modo se entenderá que están en el paquete por defecto. Para ello basta con crear un paquete haciendo **Archivo-Nuevo-Paquete** y después eligiendo la carpeta en la que queremos guardar el paquete (normalmente un proyecto tiene una sola carpeta fuente) y el nombre que queramos dar al paquete (se recomienda usar sólo caracteres en minúsculas para el nombre del paquete) si espacios ni símbolos.

crear una nueva clase (un nuevo archivo)

Por ahora diremos que una clase y un archivo es lo mismo (aunque no lo son). En java cada archivo que queramos ejecutar será una clase. Si además esa clase es ejecutable, contendrá el método **main**.

Para crear una nueva clase en Eclipse (en el proyecto en el que estemos trabajando), se hace:

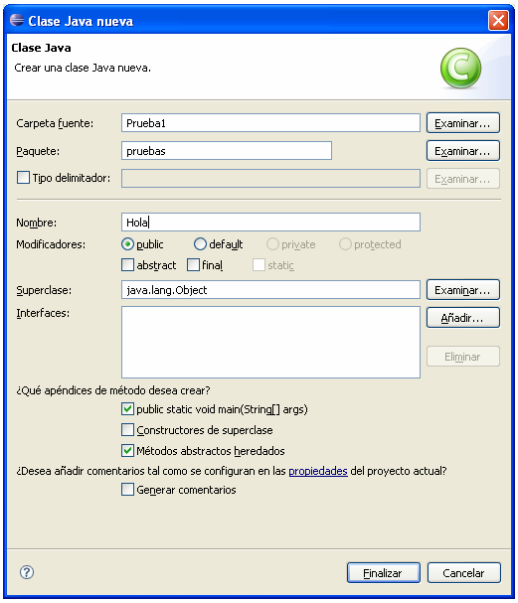
(1) Elegir **Archivo-Nuevo-Clase**

(2) Elegir el nombre de la carpeta fuente (normalmente un proyecto tiene una sola carpeta fuente).

(3) Elegir el paquete en el que se guardará nuestra clase.

(4) Poner nombre a la clase (la primera letra del nombre debe ir en mayúsculas), por supuesto sin espacios en blanco ni símbolos.

(5) Activar las casillas que nos interesen para nuestros archivos. Inicialmente sólo nos interesará activar la casilla con el método ***public static void main***, si queremos que la clase sea ejecutable.



ejecutar un programa

Basta con pulsar el botón derecho sobre la clase que queremos ejecutar y elegir **Ejecutar como-Aplicación Java.** En la consola aparecerá el resultado

modificación del funcionamiento del editor

Seguramente Eclipse es uno de los mejores entornos de desarrollo que existen y eso también se nota en las espectaculares posibilidades que otorga a su editor.

La forma de funcionar el editor puede ser distinta en cada proyecto. Por ello podemos tocar las preferencias desde **Ventana-Preferencias**.

Señalamos algunas de las posibilidades que ofrece Eclipse para mejorar el trabajo con el editor.

coloreado del código

- El **formato de la letra** en el editor se configura desde **General-Aspecto-Colores y fonts.** Después en el cuadro hay que abrir la carpeta Java y seleccionar ***Font del texto del editor Java***. Pulsando en cambiar podremos modificar la letra del editor.
- Los distintos colores con los que Eclipse marca el código, se pueden modificar desde el apartado **Java-Editor-Coloreado de sintaxis** (en el mismo cuadro de Preferencias). Después podremos elegir el color y formato deseado para la sintaxis de nuestro código Java.

plantillas

Las plantillas permiten colocar trozos de código predefinidos asociados a una determinada palabras clave. La tecla **ctrl.+Barra espaciadora** es la que permite invocar a las plantillas. Por ejemplo si escribimos **switch** y después pulsamos ctrl.+barra, se nos permitirá rellenar de golpe la sentencia **switch** (que consta de varias líneas.

Podemos modificar o crear nuevas plantillas yendo al apartado **Plantillas** en el apartado **Editor** de **Java** en el cuadro de preferencias.

opciones de tecleo

En Eclipse hay opciones que permiten automatizar la escritura de código (por ejemplo nos cierra automáticamente las llaves y paréntesis que abrimos), todas ellas se configuran en el apartado **Tecleo** del cuadro anterior.

formateador

Eclipse va formateando automáticamente el código. La forma en que el código queda sangrado, dónde aparecen las llaves, etc. Se puede configurar en el cuadro **Preferencias** del menú **Ventana**, eligiendo **Java-Estilo de código-Formateador.** Desde esa opción podemos elegir un perfil y editarle para modificar la forma en la que formatea el código. Hay varios perfiles y podemos asignar diferentes perfiles a los proyectos; cada perfil contiene todas las opciones sobre como dar formato al código,

código por defecto

El apartado **Plantillas de código** permite indicar el código que colocará Eclipse al hacer una nueva clase o una nueva función, etc.

(8.4) variables

(8.4.1) declaración de variables

Antes de poder utilizar una variable, ésta se debe declarar. Lo cual se hacede esta forma (igual que en C):

```
tipo nombrevariable;
```

Donde **tipo** es el tipo de datos que almacenará la variable (texto, números enteros,...) y ***nombrevariable*** es el nombre con el que se conocerá la variable. Ejemplos:

```
int dias;
boolean decision;
```

También se puede hacer que la variable tome un valor inicial al declarar:

```
int dias=365;
```

Y también se puede declarar más de una variable a la vez:

```
int dias=365, anio=23, semanas;
```

Al declarar una variable se puede incluso utilizar una expresión (por ejemplo una suma):

```
int a=13, b=18;
int c=a+b;
```

(8.4.2) alcance o ámbito

Esas dos palabras sinónimas, hacen referencia a la duración de una variable. En el ejemplo:

```
{
  int x=12;
}
System.out.println(x); //Error
```

Java dará error, porque la variable se usa fuera del bloque en el que se creo. Eso no es posible, porque una variable tiene como ámbito el bloque de código en el que fue creada (salvo que sea una propiedad de un objeto).

La vida de una variable comienza desde su declaración y termina cuando se cierra la llave correspondiente al bloque de código en el que se declaró.

(8.4.3) tipos de datos primitivos

Tipo de variable	Bytes que ocupa	Rango de valores
boolean	2	**true, false**
byte	1	-128 a 127
short	2	-32.768 a 32.767
int	4	-2.147.483.648 a 2.147.483.649
long	8	$-9 \cdot 10^{18}$ a $9 \cdot 10^{18}$
double	8	$-1{,}79 \cdot 10^{308}$ a $1{,}79 \cdot 10^{308}$
float	4	$-3{,}4 \cdot 10^{38}$ a $3{,}4 \cdot 10^{38}$
char	2	Caracteres (en Unicode)

enteros

Los tipos **byte, short, int** y **long** sirven para almacenar datos enteros. Los enteros son números sin decimales. Se pueden asignar enteros normales o enteros octales y hexadecimales. Los octales se indican anteponiendo un cero al número, los hexadecimales anteponiendo 0x.

```
int numero=16; //16 decimal
numero=020; //20 octal=16 decimal
numero=0x14; //10 hexadecimal=16 decimal
```

Normalmente un número literal se entiende que es entero salvo si al final se le coloca la letra **L** (por ejemplo ***673L***).

No se acepta en general asignar variables de distinto tipo. Sí se pueden asignar valores de variables enteras a variables enteras de un tipo superior (por ejemplo asignar un valor **int** a una variable **long**). Pero al revés no se puede:

```
int i=12;
byte b=i;   //error de compilación
```

La solución es hacer un **cast**. Esta operación permite convertir valores de un tipo a otro. Se usa así:

```
int i=12;
byte b=(byte) i; //No hay problema por el (cast)
```

números en coma flotante

Los decimales se almacenan en los tipos **float** y **double**. Se les llama de coma flotante por como son almacenados por el ordenador. Los decimales no son almacenados de forma exacta por eso siempre hay un posible error. En los decimales de coma flotante se habla, por tanto de precisión. Es mucho más preciso el tipo **double** que el tipo **float.**

A un valor literal (como 1.5 por ejemplo), se le puede indicar con una **f** al final del número que es float (1.5F por ejemplo) o una **D** para indicar que es **double.** Si no se indica nada, un número literal siempre se entiende que es double, por lo que al usar tipos float hay que convertir los literales.

Las valores decimales se pueden representar en notación decimal: 1.345E+3 significaría $1.345 \cdot 10^3$ o lo que es lo mismo 1345.

booleanos

Los valores booleanos (o lógicos) sirven para indicar si algo es verdadero (**true**) o falso (**false**).

En C y C++ se puede utilizar cualquier valor lógico como si fuera un número; así verdadero es el valor -1 y falso el 0**. Eso no es posible en Java.**

caracteres

Los valores de tipo carácter sirven para almacenar símbolos de escritura (en Java se puede almacenar cualquier código Unicode). Los valores Unicode son los que Java utiliza para los caracteres. Ejemplo:

```
char letra;
letra='C'; //Los caracteres van entre comillas
letra=67; //El código Unicode de la C es el 67. Esta línea
            //hace lo mismo que la anterior
```

conversión entre tipos (casting)

Hay veces en las que se deseará realizar algo como:

```
int a;
byte b=12;
a=b;
```

La duda está en si esto se puede realizar. La respuesta es que sí. Sí porque un dato de tipo ***byte*** es más pequeño que uno de tipo ***int*** y Java realizará la conversión de forma implícita. Sin embargo en:

```
int a=1;
byte b;
b=a;
```

El compilador de Java informará de un error, aunque el número 1 sea válido para un dato byte. La razón es que el tipo ***int*** es de mayor tamaño (en bytes). Si deseamos realizar esa operación necesitamos convertir el entero al tipo byte, esto se conoce como ***casting***.

Un casting consiste en poner el tipo al que deseamos convertir entre paréntesis, la expresión a la derecha del casting se convertirá automáticamente:

```
int a=1;
byte b;
b= (byte) a; //No da error. Ahora funciona bien
```

En el siguiente ejemplo:

```
byte n1=100, n2=100, n3;
n3= n1 * n2 /100;
```

Aunque el resultado es 100, y ese resultado es válido para un tipo byte; lo que ocurrirá en realidad es que ocurrirá un error. Eso es debido a que primero multiplica 100 * 100 y como eso da 10000, no tiene más remedio el compilador que pasarlo a entero y así quedará aunque se vuelva a dividir. La solución correcta sería:

```
n3 = (byte) (n1 * n2 / 100);
```

(8.4.4) modificador final. Constantes

En Java no existe el modificador **const** como ocurre en **C++** para declarar constantes. En su lugar disponemos de un modificador llamado **final** que antepuesto al tipo en una declaración de variable, hace que dicha variable no pueda modificar su valor en el código del programa.

```
final int x=7;
x=9; //Error: No puede haber asignación hacia una variable
       //de tipo final
```

(8.4.5) operadores

introducción

Los datos se manipulan muchas veces utilizando operaciones con ellos. Los datos se suman, se restan, ... y a veces se realizan operaciones más complejas.

operadores aritméticos

Son:

operador	significado
+	Suma
-	Resta
*	Producto
/	División

operador	significado
%	Módulo (resto)

Hay que tener en cuenta que el resultado de estos operadores varía notablemente si usamos enteros o si usamos números de coma flotante.

Por ejemplo:

```
double resultado1, d1=14, d2=5;
int resultado2, i1=14, i2=5;

resultado1= d1 / d2;
resultado2= i1 / i2;
```

resultado1 valdrá 2.8 mientras que ***resultado2*** valdrá 2.

El operador del módulo (%) para calcular el resto de una división entera. Ejemplo:

```
int resultado, i1=14, i2=5;

resultado = i1 % i2; //El resultado será 4
```

operadores condicionales

Sirven para comparar valores. Siempre devuelven valores booleanos. Son:

operador	significado
<	Menor
>	Mayor
>=	Mayor o igual
<=	Menor o igual
==	Igual
!=	Distinto
!	No lógico (NOT)
&&	"Y" lógico (AND)
||	"O" lógico (OR)

Los operadores lógicos (AND, OR y NOT), sirven para evaluar condiciones complejas. NOT sirve para negar una condición. Ejemplo:

```
boolean mayorDeEdad, menorDeEdad;
int edad = 21;
mayorDeEdad = edad >= 18; //mayorDeEdad será true
menorDeEdad = !mayorDeEdad; //menorDeEdad será false
```

El operador && (AND) sirve para evaluar dos expresiones de modo que si ambas son ciertas, el resultado será **true** sino el resultado será **false**.

Ejemplo:

```
boolean carnetConducir=true;
int edad=20;
boolean puedeConducir= (edad>=18) && carnetConducir;
/*Si la edad es de al menos 18 años y carnetConducir es
//true, puedeConducir es true*/
```

El operador **||** (**OR**) sirve también para evaluar dos expresiones. El resultado será **true** si al menos uno de las expresiones es **true**.

Ejemplo:

```
boolean nieva = true, llueve= false, graniza = false;
boolean malTiempo= nieva || llueve || graniza;
//Mal tiempo valdrá true
```

operadores de BIT

Manipulan los bits de los números. Son:

operador	significado
&	AND
|	OR
~	NOT
^	XOR
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda
>>>	Desplazamiento derecha con relleno de ceros
<<<	Desplazamiento izquierda con relleno de ceros

operadores de asignación

Permiten asignar valores a una variable. El fundamental es "=". Pero sin embargo se pueden usar expresiones más complejas como:

```
x += 3;
```

En el ejemplo anterior lo que se hace es sumar 3 a la x (es lo mismo x+=3, que x=x+3). Eso se puede hacer también con todos estos operadores:

+=	-=	*=	/=
&=	|=	^=	%=
>>=	<<=		

También se pueden concatenar asignaciones:

```
x1 = x2 = x3 = 5;
```

Otros operadores de asignación son "++" (incremento) y "- -" (decremento). Ejemplo:

```
x++; //esto es x=x+1;
x--; //esto es x=x-1;
```

Pero hay dos formas de utilizar el incremento y el decremento. Se puede usar por ejemplo x++ o ++x

La diferencia estriba en el modo en el que se comporta la asignación. Ejemplo:

```
int x=5, y=5, z;
z=x++; //z vale 5, x vale 6
z=++y; //z vale 6, y vale 6
```

operador ?

Este operador (conocido como **if** de una línea) permite ejecutar una instrucción u otra según el valor de la expresión. Sintaxis:

```
expresionlogica?instruccionSiVerdadera:instruccionSiFalsa;
```

Ejemplo:

```
System.out.println(nota>5 ? "Aprobado": "Suspenso");
```

En el ejemplo, se escribe en pantalla la palabra ***Aprobado*** si la nota supera el 5, sino se escribe suspenso.

instanceof

Se usa para determinar el tipo de un objeto durante la ejecución del programa. Devuelve **true** en el caso de que el objeto posea el tipo indicado. Ejemplo:

```
boolean b=(numero instanceof int);
//Si numero es int, b valdrá verdadero
```

precedencia

A veces hay expresiones con operadores que resultan confusas. Por ejemplo en:

```
resultado = 8 + 4 / 2;
```

Es difícil saber el resultado. ¿Cuál es? ¿seis o diez? La respuesta es 10 y la razón es que el operador de división siempre precede en el orden de ejecución al

de la suma. Es decir, siempre se ejecuta antes la división que la suma. Siempre se pueden usar paréntesis para forzar el orden deseado:

```
resultado = (8 + 4) / 2;
```

Ahora no hay duda, el resultado es seis. No obstante el orden de precedencia de los operadores Java es:

1	( )	[ ]	.	
2	++	--	~	!
3	*	/	%	
4	+	-		
5	>>	>>>	<<	<<<
6	>	>=	<	<=
7	==	!=		
8	&			
9	^			
10	|			
11	&&			
12	||			
13	?:			
14	=	+=, -=, *=,...		

En la tabla anterior los operadores con mayor precedencia está en la parte superior, los de menor precedencia en la parte inferior. De izquierda a derecha la precedencia es la misma. Es decir, tiene la misma precedencia el operador de suma que el de resta.

Esto último provoca conflictos, por ejemplo en:

```
resultado = 9 / 3 * 3;
```

El resultado podría ser uno ó nueve. En este caso el resultado es nueve, porque la división y el producto tienen la misma precedencia; por ello el compilador de Java realiza primero la operación que este más a la izquierda, que en este caso es la división.

Una vez más los paréntesis podrían evitar estos conflictos.

la clase Math

Se echan de menos operadores matemáticos más potentes en Java. Por ello se ha incluido una clase especial llamada **Math** dentro del paquete **java.lang.** Para poder utilizar esta clase, se debe incluir esta instrucción (a veces no es necesaria porque los editores de Java importan automáticamente el contenido de **java.lang**):

```
import java.lang.Math;
```

Esta clase posee métodos muy interesantes para realizar cálculos matemáticos complejos. Por ejemplo:

```
double x= Math.pow(3,3); //x es 33
```

Math posee dos constantes, que son:

constante	significado
final static double **E**	El número **e** (2, 7182818245...)
final static double **PI**	El número ∏ (3,14159265...)

Por otro lado posee numerosos métodos que son:

operador	significado
double **ceil**(double x)	Redondea ***x*** al entero mayor siguiente: ♦ **Math.ceil**(2.8) vale 3 ♦ **Math.ceil**(2.4) vale 3 ♦ **Math.ceil**(-2.8) vale -2
double **floor**(double x)	Redondea ***x*** al entero menor siguiente: ♦ **Math.floor**(2.8) vale 2 ♦ **Math. floor** (2.4) vale 2 ♦ **Math. floor** (-2.8) vale -3
int **round**(double x)	Redondea ***x*** de forma clásica: ♦ **Math.round**(2.8) vale 3 ♦ **Math. round** (2.4) vale 2 ♦ **Math. round** (-2.8) vale -3
double **rint**(double x)	Idéntico al anterior, sólo que éste método da como resultado un número **double** mientras que **round** da como resultado un entero tipo **int**
double **random**()	Número aleatorio de 0 a 1
tiponúmero **abs**(*tiponúmero* x)	Devuelve el valor absoluto de ***x.***
tiponúmero **min**(*tiponúmero* x, *tiponúmero* y)	Devuelve el menor valor de ***x*** o ***y***

operador	significado
tiponúmero **max(** *tiponúmero* x**,** *tiponúmero* y**)**	Devuelve el mayor valor de ***x*** o ***y***
double sqrt(double x**)**	Calcula la raíz cuadrada de ***x***
double pow(double x**, double** y***)***	Calcula x^y
double exp(double x**)**	Calcula e^x
double log(double x**)**	Calcula el logaritmo neperiano de ***x***
double acos(double x**)**	Calcula el arco coseno de ***x***
double asin(double x**)**	Calcula el arco seno de ***x***
double atan(double x**)**	Calcula el arco tangente de ***x***
double sin(double x**)**	Calcula el seno de ***x***
double cos(double x**)**	Calcula el coseno de ***x***
double tan(double x**)**	Calcula la tangente de ***x***
double toDegrees(double anguloEnRadianes**)**	Convierte de radianes a grados
double toRadians(double anguloEnGrados**)**	Convierte de grados a radianes

(8.5) control del flujo

(8.5.1) if

Permite crear estructuras condicionales simples; en las que al cumplirse una condición se ejecutan una serie de instrucciones. Se puede hacer que otro conjunto de instrucciones se ejecute si la condición es falsa. La condición es cualquier expresión que devuelva un resultado de **true** o **false.** La sintaxis de la instrucción **if** es:

```
if (condición) {
  instrucciones que se ejecutan si la condición es true
}
else {
  instrucciones que se ejecutan si la condición es false
}
```

La parte **else** es opcional. Ejemplo:

```
if ((diasemana>=1) && (diasemana<=5)){
  trabajar = true;
}
else {
```

```
  trabajar = false;
}
```

Se pueden anidar varios if a la vez. De modo que se comprueban varios valores. Ejemplo:

```
if (diasemana==1) dia="Lunes";
else if (diasemana==2) dia="Martes";
else if (diasemana==3) dia="Miércoles";
else if (diasemana==4) dia="Jueves";
else if (diasemana==5) dia="Viernes";
else if (diasemana==2) dia="Sábado";
else if (diasemana==2) dia="Domingo";
else dia="?";
```

(8.5.2) switch

Es la estructura condicional compleja porque permite evaluar varios valores a la vez. Sintaxis:

```
switch (expresión) {
  case valor1:
       sentencias si la expresiona es igual al valor1;
       [break]
  case valor2:
       sentencias si la expresiona es igual al valor2;
       [break]
       .
       .
       .
  default:
       sentencias que se ejecutan si no se cumple
       ninguna de las anteriores
}
```

Esta instrucción evalúa una expresión (que debe ser de tipo **short, int, byte** o **char**), si toma el primera valor ejecuta las sentencias correspondientes al **case** de ese valor. Y así con las demás excepto con el grupo **default** que se ejecuta si la expresión no tomó ningún valor de la lista.

La sentencia opcional **break** se ejecuta para hacer que el flujo del programa salte al final de la sentencia **switch,** de otro modo se ejecutarían todas las sentencias restantes, sean o no correspondientes al valor que tomo la expresión.

Ejemplo 1:

```
switch (diasemana) {
  case 1:
       dia=”Lunes”;
       break;
  case 2:
       dia=”Martes”;
       break;
  case 3:
       dia=”Miércoles”;
       break;
  case 4:
       dia=”Jueves”;
       break;
  case 5:
       dia=”Viernes”;
       break;
  case 6:
       dia=”Sábado”;
       break;
  case 7:
       dia=”Domingo”;
       break;
  default:
       dia=”?”;
}
```

Ejemplo 2:

```
switch (diasemana) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:
       laborable=true;
       break;
  case 6:
  case 7:
       laborable=false;}
```

(8.5.3) while

La instrucción **while** permite crear bucles. Un bucle es un conjunto de sentencias que se repiten si se cumple una determinada condición. En el caso de que la condición pase a ser falsa, el bucle deja de ejecutarse. Sintaxis:

```
while (condición) {
  sentencias que se ejecutan si la condición es true
}
```

Ejemplo (cálculo del factorial de un número, el factorial de 4 sería: 4*3*2*1):

```
//factorial de 4
int n=4, factorial=1, temporal=n;
while (temporal>0) {
  factorial*=temporal--;
}
```

(8.5.4) do while

Crea también un bucle, sólo que en este tipo de bucle la condición se evalúa después de ejecutar las instrucciones; lo cual significa que al menos el bucle se ejecuta una vez. Sintaxis:

```
do {
  instrucciones
} while (condición);
```

(8.5.5) for

Es un bucle más complejo especialmente pensado para rellenar arrays. Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición. Un contador determina las veces que se ejecuta el bucle. Sintaxis:

```
for (expresiónInicial; condición; expresiónEncadavuelta)
{
  instrucciones;
}
```

La ***expresión inicial*** es una instrucción que se ejecuta una sola vez, al entrar en el **for** (normalmente esa expresión lo que hace es dar valor inicial al contador del bucle). La condición es una expresión que devuelve un valor lógico. En el caso de que esa expresión sea verdadera se ejecutan las instrucciones.

Después de ejecutarse las instrucciones interiores del bucle, se realiza la expresión que tiene lugar en cada vuelta (que, generalmente, incrementa o decrementa el contador). Luego se vuelve a evaluar la condición y así sucesivamente hasta que la condición sea falsa.

Ejemplo (factorial):

```
//factorial de 4
int n=4, factorial=1, temporal=n;

for (temporal=n;temporal>0;temporal--){
  factorial *=temporal;
}
```

Ese código es equivalente a este otro (utilizando **while**):

```
//factorial de 4
int n=4, factorial=1, temporal=n;
temporal=n;
while (temporal>0){
  factorial *=temporal;
  temporal--;
}
```

(8.5.6) sentencias de salida de un bucle

break

Es una sentencia que permite salir del bucle en el que se encuentra inmediatamente. Hay que intentar evitar su uso ya que produce malos hábitos al programar.

continue

Instrucción que siempre va colocada dentro de un bucle y que hace que el flujo del programa ignore el resto de instrucciones del bucle; dicho de otra forma, va hasta la siguiente iteración del bucle. Al igual que ocurría con **break**, hay que intentar evitar su uso.

(8.6) arrays y cadenas

(8.6.1) arrays

unidimensionales

Un array es una colección de valores de un mismo tipo engrosados en la misma variable. De forma que se puede acceder a cada valor independientemente. Para Java además un array es un objeto que tiene propiedades que se pueden manipular.

Los arrays solucionan problemas concernientes al manejo de muchas variables que se refieren a datos similares. Por ejemplo si tuviéramos la necesidad de almacenar las notas de una clase con 18 alumnos, necesitaríamos 18 variables, con la tremenda lentitud de manejo que supone eso. Solamente calcular la nota media requeriría una tremenda línea de código. Almacenar las notas supondría al menos 18 líneas de código.

Gracias a los arrays se puede crear un conjunto de variables con el mismo nombre. La diferencia será que un número (índice del array) distinguirá a cada variable.

En el caso de las notas, se puede crear un array llamado notas, que representa a todas las notas de la clase. Para poner la nota del primer alumno se usaría notas[0], el segundo sería notas[1], etc. (los corchetes permiten especificar el índice en concreto del array).

La declaración de un array unidimensional se hace con esta sintaxis.

```
tipo nombre[];
```

Ejemplo:

```
double cuentas[]; //Declara un array que almacenará valores
                  // doubles
```

Declara un array de tipo **double**. Esta declaración indica para qué servirá el array, pero no reserva espacio en la RAM al no saberse todavía el tamaño del mismo. Por eso hay que utilizar el operador **new**. Con él se indica ya el tamaño y se reserva el espacio necesario en memoria. Un array no inicializado es un array **null.** Ejemplo:

```
int notas[]; //sería válido también int[] notas;
notas = new int[3]; //indica que el array constará de
tres                        //valores de tipo int

//También se puede hacer todo a la vez
//int notas[]=new int[3];
```

Los valores del array se asignan utilizando el índice del mismo entre corchetes:

```
notas[2]=8;
```

También se pueden asignar valores al array en la propia declaración:

```
int notas[] = {8, 7, 9};
```

Esto declara e inicializa un array de tres elementos.

En Java (como en otros lenguajes) el primer elemento de un array es el cero. El primer elemento del array notas, es notas[0]. Se pueden declarar arrays a cualquier tipo de datos (enteros, booleanos, doubles, ... e incluso objetos).

La ventaja de usar arrays (volviendo al caso de las notas) es que gracias a un simple bucle **for** se puede rellenar o leer fácilmente todos los elementos de un array:

```
//Calcular la media de las 18 notas
int i;
suma=0;
for (i=0;i<=17;i++){
  suma+=nota[i];
}
media=suma/18;
```

A un array se le puede inicializar las veces que haga falta:

```
int notas[]=new int[16];
...
notas=new int[25];
```

En el ejemplo anterior cuando se vuelve a definir notas, hay que tener en cuenta que se pierde el contenido que tuviera el array anteriormente. Es decir, no se amplía el array, se crea otro pero que utiliza el mismo nombre. El array primero (que crea 16 números enteros en memoria, se perderá, el recolector de basura de Java lo acabará eliminando).

Un array se puede asignar a otro array (si son del mismo tipo):

```
int notas[];
int ejemplo[]=new int[18];
notas=ejemplo;
```

En el último punto, ***notas*** equivale a ***ejemplo***. Esta asignación provoca que cualquier cambio en ***notas*** también cambie el array ***ejemplos*** (de hecho se trata del mismo array). Es decir ***notas[2]=8*** es lo mismo que ***ejemplo[2]=8.***

Esto nos permite decir que el array en sí son los valores que realmente están almacenándose en memoria, y que tanto ***notas*** como ***ejemplo*** son dos referencias al mismo array. Esta idea de ***referencia*** será pulida en los siguientes apartados.

arrays multidimensionales

Los arrays además pueden tener varias dimensiones. Entonces se habla de arrays de arrays (arrays que contienen arrays) Ejemplo:

```
int notas[][];
```

notas es un array que contiene arrays de enteros

```
notas = new int[3][12];//notas está compuesto por 3 arrays
                       //de 12 enteros cada uno
notas[0][0]=9;//el primer valor es 0
```

Puede haber más dimensiones, incluso ***notas[3][2][7]***. Los arrays multidimensionales se pueden inicializar de forma más creativa incluso. Ejemplo:

```
int notas[][]=new int[5][];//Hay 5 arrays de enteros
notas[0]=new int[100]; //El primer array es de 100 enteros
notas[1]=new int[230]; //El segundo de 230
notas[2]=new int[400];
notas[3]=new int[100];
notas[4]=new int[200];
```

Hay que tener en cuenta que en el ejemplo anterior, notas[0] es un array de 100 enteros. Mientras que notas, es un array de 5 arrays de enteros. Es decir, desde el punto de vista de Java, un array de dos dimensiones es un array de arrays.

obtener propiedades de un array

longitud de un array

Los arrays poseen un método que permite determinar cuánto mide un array. Se trata de **length.** Ejemplo (continuando del anterior):

```
int datos[]={21,34,56,23,12,8];
System.out.println(notas.length); //Sale 6
```

En el caso de arrays de dos dimensiones hay que tener en cuenta que la propiedad varía según donde la utilicemos. Por ejemplo si utilizamos el array ***notas*** definido en el apartado anterior:

```
System.out.println(notas.length); //Sale 5
System.out.println(notas[2].length); //Sale 400
```

la clase Arrays

En el paquete **java.utils** se encuentra una clase estática llamada **Arrays**. Una clase estática permite ser utilizada como si fuera un objeto, es decir se accede a sus funciones usando el nombre de la clase (como ocurre con **Math**). Esta clase posee métodos muy interesantes para utilizar sobre arrays.

Su uso es

```
Arrays.método(argumentos);
```

toString

Convierte un array en un texto que contiene los valores del array. Ejemplo:

```
int a[]={3,4,5,6};
System.out.println(Arrays.toString(a)); //Sale [3 4 5 6]
```

fill

Permite rellenar todo un array unidimensional con un determinado valor. Sus argumentos son el array a rellenar y el valor deseado:

```
int valores[]=new int[23];
Arrays.fill(valores,-1);//Todos los elementos del array valen -1
```

También permite decidir desde que índice hasta qué índice rellenamos:

```
Arrays.fill(valores,5,8,-1);//Del elemento 5 al 7 valdrán -1
```

equals

Compara dos arrays y devuelve true si son iguales. Se consideran iguales si son del mismo tipo, tamaño y contienen los mismos valores.

sort

Permite ordenar un array en orden ascendente. Se pueden ordenar sólo una serie de elementos desde un determinado punto hasta un determinado punto.

```
int x[]={4,5,2,3,7,8,2,3,9,5};
Arrays.sort(x);//Estará ordenado
Arrays.sort(x,2,5);//Ordena del 2º al 4º elemento
```

binarySearch

Permite buscar un elemento de forma ultrarrápida en un array ordenado (en un array desordenado sus resultados son impredecibles). Devuelve el índice en el que está colocado el elemento (si no lo encuentra devuelve un número negativo). Ejemplo:

```
int x[]={1,2,3,4,5,6,7,8,9,10,11,12};
System.out.println(Arrays.binarySearch(x,8));//Da  7
```

Si el array no está ordenado, esta función no la podemos utilizar con él. Aunque cualquier array se puede ordenar con el método **sort** de la clase **Arrays**

el método System.arraycopy

La clase System también posee un método relacionado con los arrays, dicho método permite copiar un array en otro. Recibe cinco argumentos: el array que se copia, el índice desde que se empieza a copia en el origen, el array destino de la copia, el índice desde el que se copia en el destino, y el tamaño de la copia (número de elementos de la copia).

Ejemplo:

```
int uno[]={1,1,2};
int dos[]={3,3,3,3,3,3,3,3,3};
System.arraycopy(uno, 0, dos, 0, uno.length);
for (int i=0;i<=8;i++){
    System.out.print(dos[i]+" ");
} //Sale 112333333
```

(8.6.2) clase String

introducción

Para Java las cadenas de texto son objetos especiales. Los textos deben manejarse creando objetos de tipo String. Ejemplo:

```
String texto1 = "¡Prueba de texto!";
```

Las cadenas pueden ocupar varias líneas utilizando el operador de concatenación "+".

```
String texto2 ="Este es un texto que ocupa " +
            "varias líneas, no obstante se puede "+
            "perfectamente encadenar";
```

También se pueden crear objetos String sin utilizar constantes entrecomilladas, usando otros constructores (otras formas de definir las variables String):

```
char[] palabra = {'P','a','l','a','b','r','a'};//Array de char
String cadena = new String(palabra);
byte[] datos = {97,98,99};
String codificada = new String (datos, "8859_1");
```

En el último ejemplo la cadena ***codificada*** se crea desde un array de tipo byte que contiene números que serán interpretados como códigos Unicode. Al asignar, el valor ***8859_1*** indica la tabla de códigos a utilizar.

comparación entre objetos String

Los objetos **String** no pueden compararse directamente con los operadores de comparación (==, >=,…). En su lugar se debe usar alguna de estas expresiones:

(1) ***s1*.equals(*s2*).** Da **true** si ***s1*** es igual a ***s2***

(2) ***s1*.equalsIgnoreCase(*s2*).** Da **true** si ***s1*** es igual a ***s2*** (ignorando mayúsculas y minúsculas)

(3) ***s1*.compareTo(*s2*).** Si ***s1 < s2*** devuelve un número menor que 0, si ***s1*** es igual a ***s2*** devuelve cero y si ***s1>s2*** devuelve un número mayor que 0

(4) ***s1*.compareToIgnoreCase(*s2*).** Igual que la anterior, sólo que además ignora las mayúsculas (disponible desde Java 1.2)

métodos de String

En realidad las variables String no son variables, sino objetos. Y como se verá más adelante, sobre los objetos se puede acceder a sus propiedades utilizando el nombre del objeto, un punto y el nombre de la propiedad.

valueOf

Este método pertenece no sólo a la clase String, sino a otras y siempre es un método que convierte valores de una clase a otra. En el caso de los objetos String, permite convertir valores que no son de cadena a forma de cadena.

Ejemplos:

```
String numero = String.valueOf(1234);
String fecha = String.valueOf(new Date());
```

En el ejemplo se observa que este método pertenece a la clase String directamente, en el resto se utilizaría el nombre del objeto.

length

Permite devolver la longitud de una cadena:

```
String texto1="Prueba";
System.out.println(texto1.length());//Escribe 6
```

concatenar cadenas

Se puede hacer de dos formas, utilizando el método **concat** o con el operador +. Ejemplo:

```
String s1="Buenos ", s2="días", s3, s4;
s3 = s1 + s2;
s4 = s1.concat(s2);
```

charAt

Devuelve un carácter de la cadena. El carácter a devolver se indica por su posición (el primer carácter es la posición 0). Ejemplo;

```
String s1="Prueba";
char c1=s1.charAt(2); //c1 valdrá 'u'
```

substring

Da como resultado una porción del texto de la cadena. La porción se toma desde una posición inicial hasta una posición final (sin incluir esa posición final). Ejemplo:

```
String s1="Buenos días";
String s2=s1.substring(7,10); //s2 = día
```

indexOf

Devuelve la primera posición en la que aparece un determinado texto en la cadena. Ejemplo:

```
String  s1="Quería decirte que quiero que te vayas";
System.out.println(s1.indexOf("que")); //Da 15
```

Se puede buscar desde una determinada posición. En el ejemplo anterior:

```
System.out.println(s1.indexOf("que",16)); //Ahora da 26
```

lastIndexOf

Devuelve la última posición en la que aparece un determinado texto en la cadena. Es casi idéntica a la anterior, sólo que busca desde el final. Ejemplo:

```
String  s1="Quería decirte que quiero que te vayas";
System.out.println(s1.lastIndexOf("que"); //Da 26
```

También permite comenzar a buscar desde una determinada posición.

endsWith

Devuelve **true** si la cadena termina con un determinado texto, que se indica como parámetro.

replace

Cambia todas las apariciones de un carácter por otro en el texto que se indique:

```
String s1="Mariposa";
System.out.println(s1.replace('a','e'));//Da Meripose
```

toUpperCase

Devuelve la versión mayúsculas de la cadena.

toLowerCase

Devuelve la versión minúsculas de la cadena.

lista completa de métodos de String

método	descripción
char charAt(int index**)**	Proporciona el carácter que está en la posición dada por el entero ***index.***
int compareTo(String s**)**	Compara las dos cadenas. Devuelve un valor menor que cero si la cadena ***s*** es mayor que la original, devuelve 0 si son iguales y devuelve un valor mayor que cero si ***s*** es menor que la original.
int compareToIgnoreCase(String s**)**	Compara dos cadenas, pero no tiene e cuenta si el texto es mayúsculas o no.
String concat(String s**)**	Añade la cadena ***s*** a la cadena original.
String copyValueOf(char[] data**)**	Produce un objeto **String** que es igual al array de caracteres ***data***.
boolean endsWith(String s**)**	Devuelve **true** si la cadena termina con el texto ***s***
boolean equals(String s**)**	Compara ambas cadenas, devuelve **true** si son iguales
boolean equalsIgnoreCase(String s**)**	Compara ambas cadenas sin tener en cuenta las mayúsculas y las minúsculas.
byte[] getBytes()	Devuelve un array de caracteres que toma a partir de la cadena de texto
void getBytes(int srcBegin, **int** srcEnd, **char[]** dest, **int** dstBegin);	Almacena el contenido de la cadena en el array de caracteres ***dest***. Toma los caracteres desde la posición ***srcBegin*** hasta la posición ***srcEnd*** y les copia en el array desde la posición ***dstBegin***
int indexOf(String s**)**	Devuelve la posición en la cadena del texto s
int indexOf(String s, **int** primeraPos)	Devuelve la posición en la cadena del texto s, empezando a buscar desde la posición ***PrimeraPos***
int lastIndexOf(String s**)**	Devuelve la última posición en la cadena del texto s
int lastIndexOf(String s, **int** primeraPos)	Devuelve la última posición en la cadena del texto s, empezando a buscar desde la posición ***PrimeraPos***
int length()	Devuelve la longitud de la cadena

método	descripción
String replace(char carAnterior, char ncarNuevo)	Devuelve una cadena idéntica al original pero que ha cambiando los caracteres iguales a *carAnterior* por *carNuevo*
String replaceFirst(String str1, String str2)	Cambia la primera aparición de la cadena *str1* por la cadena *str2*
String replaceFirst(String str1, String str2)	Cambia la primera aparición de la cadena uno por la cadena dos
String replaceAll(String str1, String str2)	Cambia la todas las apariciones de la cadena uno por la cadena dos
String startsWith(String s)	Devuelve true si la cadena comienza con el texto *s.*
String substring(int primeraPos, int segundaPos)	Devuelve el texto que va desde *primeraPos* a *segunaPos*.
char[] toCharArray()	Devuelve un array de caracteres a partir de la cadena dada
String toLowerCase()	Convierte la cadena a minúsculas
String toLowerCase(Locale local)	Lo mismo pero siguiendo las instrucciones del argumento *local*
String toUpperCase()	Convierte la cadena a mayúsculas
String toUpperCase(Locale local)	Lo mismo pero siguiendo las instrucciones del argumento *local*
String trim()	Elimina los blancos que tenga la cadena tanto por delante como por detrás
static String valueOf(tipo elemento)	Devuelve la cadena que representa el valor *elemento*. Si elemento es booleano, por ejemplo devolvería una cadena con el valor true o false

(8.7) objetos y clases

(8.7.1) programación orientada a objetos

Se ha comentado anteriormente en este manual que Java es un lenguaje totalmente orientado a objetos. De hecho siempre hemos definido una clase pública con un método **main** que permite que se pueda visualizar en la pantalla el programa Java.

La gracia de la POO es que se hace que los problemas sean más sencillos, al permitir dividir el problema. Está división se hace en objetos, de forma que cada objeto funcione de forma totalmente independiente. Un objeto es un elemento del programa que posee sus propios datos y su propio funcionamiento.

Es decir un objeto está formado por datos (**propiedades**) y funciones que es capaz de realizar el objeto (**métodos**).

Antes de poder utilizar un objeto, se debe definir su **clase.** La clase es la definición de un tipo de objeto. Al definir una clase lo que se hace es indicar como funciona un determinado tipo de objetos. Luego, a partir de la clase, podremos crear objetos de esa clase.

Por ejemplo, si quisiéramos crear el juego del parchís en Java, una clase sería la casilla, otra las fichas, otra el dado, etc., etc. En el caso de la casilla, se definiría la clase para indicar su funcionamiento y sus propiedades, y luego se crearía tantos objetos casilla como casillas tenga el juego.

Lo mismo ocurriría con las fichas, la clase **ficha** definiría las propiedades de la ficha (color y posición por ejemplo) y su funcionamiento mediante sus métodos (por ejemplo un método sería mover, otro llegar a la meta, etc., etc.,), luego se crearían tantos objetos ficha, como fichas tenga el juego.

Por ejemplo la clase coche representa (simboliza) a todos los coches. Esa clase sirve para abstraernos y pensar que todos los coches tienen propiedades y métodos comunes (cuatro ruedas, corren , frenan,…). Sin embargo si por la ventana de casa veo un coche, eso ya es un objeto, puesto que es un coche concreto y no la idea de coche.

(8.7.2) propiedades de la POO

- **Encapsulamiento**. Una clase se compone tanto de variables (propiedades) como de funciones y procedimientos (métodos). De hecho no se pueden definir variables (ni funciones) fuera de una clase (es decir no hay variables ***globales***).
- **Ocultación**. Hay una zona oculta al definir la clases (zona privada) que sólo es utilizada por esa clases y por alguna clase relacionada. Hay una zona pública (llamada también **interfaz** de la clase) que puede ser utilizada por cualquier parte del código.
- **Polimorfismo**. Cada método de una clase puede tener varias definiciones distintas. En el caso del parchís: partida.empezar(4) empieza una partida para cuatro jugadores, partida.empezar(rojo, azul) empieza una partida de dos jugadores para los colores rojo y azul; estas son dos formas distintas de emplear el método empezar, que es polimórfico.
- **Herencia**. Una clase puede heredar propiedades de otra.

(8.7.3) clases

Las clases son las plantillas para hacer objetos. En una clase se define los comportamientos y propiedades que poseerán los objetos. Hay que pensar en una clase como un molde. A través de las clases se obtienen los objetos en sí.

Es decir antes de poder utilizar un objeto se debe definir la clase a la que pertenece, esa definición incluye:

(1) **Sus propiedades.** Es decir, los datos miembros de esa clase. Los datos pueden ser públicos (accesibles desde otra clase) o privados (sólo accesibles por código de su propia clase. También se las llama **campos**.

(2) **Sus métodos.** Las funciones miembro de la clase. Son las acciones (u operaciones) que puede realizar la clase.

(3) **Código de inicialización.** Para crear una clase normalmente hace falta realizar operaciones previas (es lo que se conoce como el **constructor** de la clase).

(4) **Otras clases.** Dentro de una clase se pueden definir otras clases (clases internas).

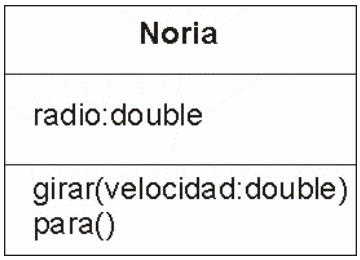
El formato general para crear una clase es:

```
[acceso] class nombreDeClase {
  [acceso] [static] tipo propiedad1;
  [acceso] [static] tipo propiedad2;
  [acceso] [static] tipo propiedad3;
  ...
  [acceso] [static] tipo método1(listaDeArgumentos) {
       ...código del método...
  }
  ...
}
```

La palabra opcional **static** sirve para hacer que el método o la propiedad a la que precede se pueda utilizar de manera genérica (más adelante se hablará de clases genéricas), los métodos o propiedades así definidos se llaman **propiedades de clase** y **métodos de clase** respectivamente. Su uso se verá más adelante. Ejemplo;

```
class Noria {
  double radio;
  void girar(int velocidad){
       ...//definición del método
  }
  void parar(){...
```

En notación UML la clase Noria se simboliza de esta manera:



Es muy conveniente (podríamos tomarlo como obligatorio) que el nombre de las clases se ponga haciendo que la primera letra del mismo se coloque en mayúsculas.

(8.7.4) objetos

Se les llama **instancias o ejemplares de clase** (del verbo inglés ***instance***, de difícil traducción). Son un elemento en sí de la clase (en el ejemplo del parchís, una ficha en concreto), por lo que poseerá todas las propiedades y métodos definidos para la clase.

datos miembro (propiedades)

Para poder acceder a las propiedades de un objeto, se utiliza esta sintaxis:

```
objeto.propiedad
```

Por ejemplo:

```
noria5.radio;
```

métodos

Los métodos se utilizan de la misma forma que las propiedades:

```
objeto.método(argumentosDelMétodo)
```

Los métodos siempre tienen paréntesis (es la diferencia con las propiedades) y dentro de los paréntesis se colocan los argumentos del método. Que son los datos que necesita el método para funcionar.

Por ejemplo:

```
Noria.gira(5);
```

Lo cual podría hacer que la Noria avance a 5 Km/h.

herencia

En la POO tiene mucha importancia este concepto, la herencia es el mecanismo que permite crear clases basadas en otras existentes. Se dice que esas clases ***descienden*** de las primeras. Así por ejemplo, se podría crear una clase llamada **vehículo** cuyos métodos serían mover**, *parar, acelerar*** y ***frenar***. Y después se

podría crear una clase **coche** basada en la anterior que tendría esos mismos métodos (les heredaría) y además añadiría algunos propios, por ejemplo ***abrirCapó*** o ***cambiarRueda.***

creación de objetos de la clase

Una vez definida la clase, se pueden utilizar objetos de la clase. Normalmente consta de dos pasos. Su declaración, y su creación. La declaración consiste en indicar que se va a utilizar un objeto de una clase determinada. Y se hace igual que cuando se declara una variable simple. Por ejemplo:

```
Noria noriaDePalencia;
```

Eso declara el objeto ***noriaDePalencia*** como objeto de tipo ***Noria;*** se supone que previamente se ha definido la clase ***Noria.***

Para poder utilizar un objeto, hay que crearle de verdad. Eso consiste en utilizar el operador **new.** Por ejemplo:

```
noriaDePalencia = new Noria();
```

Al hacer esta operación el objeto reserva la memoria que necesita y se inicializa el objeto mediante su **constructor**. Más adelante veremos como definir el constructor.

(8.7.5) especificadores de acceso

Se trata de una palabra que antecede a la declaración de una clase, método o propiedad de clase. Hay tres posibilidades: **public, protected** y **private.** Una cuarta posibilidad es no utilizar ninguna de estas tres palabras; entonces se dice que se ha utilizado el modificador por defecto (**friendly**).

Los especificadores determinan el alcance de la visibilidad del elemento al que se refieren. Referidos por ejemplo a un método, pueden hacer que el método sea visible sólo para la clase que lo utiliza (**private**), para éstas y las heredadas (**protected**), para todas las clases del mismo paquete (**friendly**) o para cualquier clase del tipo que sea (**public**).

En la siguiente tabla se puede observar la visibilidad de cada especificador:

zona	private (privado)	sin modificador (friendly)	protected (protegido)	public (público)
Misma clase	X	X	X	X
Subclase en el mismo paquete		X	X	X
Clase (no subclase) en el mismo paquete		X		X
Subclase en otro paquete			X	X
No subclase en otro paquete				X

En los diagramas de clases UML (la notación más popular para representar clases), las propiedades y métodos privados anteponen al nombre el signo -, los públicos el signo **+**, los protegidos el signo **#** y los *friendly* no colocan ningún signo.

(8.7.6) creación de clases

definir propiedades de la clase (variables o datos de la clases)

Cuando se definen los datos de una determinada clase, se debe indicar el tipo de propiedad que es (String, int, double, int[][],...) y el **especificador de acceso** (public, private,...). El especificador indica en qué partes del código ese dato será visible.

Ejemplo:

```
class Persona {
  public String nombre;//Se puede acceder desde
cualquier clase
  private int contraseña;//Sólo se puede acceder desde
la
                        //clase Persona
protected String dirección;//Acceden a esta propiedad
                  //esta clase y sus descendientes
```

Por lo general las propiedades de una clase suelen ser privadas o protegidas, a no ser que se trate de un valor constante, en cuyo caso se declararán como públicos.

Las propiedades de una clase pueden ser inicializadas.

```
class Auto{
  public nRuedas=4;
```

definir métodos de clase (operaciones o funciones de clase)

Un método es una llamada a una operación de un determinado objeto. Al realizar esta llamada (también se le llama enviar un mensaje), el control del programa pasa a ese método y lo mantendrá hasta que el método finalice o se haga uso de **return**.

Para que un método pueda trabajar, normalmente hay que pasarle unos datos en forma de argumentos o parámetros, cada uno de los cuales se separa por comas. Ejemplos de llamadas:

```
balón.botar(); //sin argumentos
miCoche.acelerar(10);
ficha.comer(posición15);//posición 15 es una variable que se
                        //pasa como argumento
partida.empezarPartida("18:15",colores);
```

Los métodos de la clase se definen dentro de ésta. Hay que indicar un modificador de acceso (**public, private, protected** o ninguno (***friendly***), al igual que ocurre con las variables y con la propia clase) y un tipo de datos, que indica qué tipo de valores devuelve el método.

Esto último se debe a que los métodos son funciones que pueden devolver un determinado valor (un entero, un texto, un valor lógico,...) mediante el comando **return**. Si el método no devuelve ningún valor, entonces se utiliza el tipo **void** que significa que no devuelve valores (en ese caso el método no tendrá instrucción **return**).

El último detalle a tener en cuenta es que los métodos casi siempre necesitan datos para realizar la operación, estos datos van entre paréntesis y se les llama argumentos. Al definir el método hay que indicar que argumentos se necesitan y de qué tipo son.

Ejemplo:

```
public class Vehiculo {
    /** Función principal */
    int ruedas;
    private double velocidad=0;
    String nombre;
  /**
   * aumenta la velocidad
   * @param cantidad - Cuanto se incrementa la velocidad
   */
    public void acelerar(double cantidad) {
        velocidad += cantidad;
    }

 /** Disminuye la velocidad
   * @param cantidad - Cuanto se incrementa la velocidad
   */
     public void  frenar(double cantidad) {
        velocidad -= cantidad;
    }
    /** Devuelve la velocidad
   * @return Velocidad actual
   */
    public double obtenerVelocidad(){
         return velocidad;
     }

public static void main(String args[]){
      Vehiculo miCoche = new Vehiculo();
      miCoche.acelerar(12);
      miCoche.frenar(5);
      System.out.println(miCoche.obtenerVelocidad());
    } // Da 7.0
```

En la clase anterior, los métodos **acelerar** y **frenar** son de tipo **void** por eso no tienen sentencia **return**. Sin embargo el método ***obtenerVelocidad*** es de tipo double por lo que su resultado es devuelto por la sentencia **return** y puede ser escrito en pantalla.

Observar los comentarios **javadoc** (se suelen poner en todos los métodos) como comentan los parámetros mediante la clave **@param** y el valor retornado por la función mediante **@return**

argumentos por valor y por referencia

En todos los lenguajes éste es un tema muy importante. Los argumentos son los datos que recibe un método y que necesita para funcionar. Ejemplo:

```
public class Matemáticas {
  public double factorial(int n){
        double resultado;
        for (resultado=n;n>1;n--) resultado*=n;
        return resultado;
  }
  ...
public static void main(String args[]){
  Matemáticas m1=new Matemáticas();
  double x=m1.factorial(25);//Llamada al método
  }
}
```

En el ejemplo anterior, el valor 25 es un argumento requerido por el método **factorial** para que éste devuelva el resultado (que será el factorial de 25). En el código del método factorial, este valor 25 es copiado a la variable **n,** que es la encargada de almacenar y utilizar este valor.

Se dice que los argumentos son por valor, si la función recibe una copia de esos datos, es decir la variable que se pasa como argumento no estará afectada por el código. Ejemplo:

```
class prueba {
  public void metodo1(int entero){
        entero=18;
...
  }
...
public static void main(String args[]){
       int x=24;
       prueba miPrueba = new prueba();
       miPrueba.metodo1(x);
       System.out.println(x); //Escribe 24, no 18
}
```

Este es un ejemplo de paso de parámetros por valor. La variable x se pasa como argumento o parámetro para el método ***metodo1***, allí la variable ***entero*** recibe una **copia** del **valor** de x en la variable **entero**, y a esa copia se le asigna el valor 18. Sin embargo la variable x no está afectada por esta asignación.

Sin embargo en este otro caso:

```
class prueba {
  public void metodo1(int[] entero){
        entero[0]=18;
...
  }
...
public static void main(String args[]){
       int x[]={24,24};
       prueba miPrueba = new prueba();
       miPrueba.metodo1(x);
       System.out.println(x[0]); //Escribe 18, no 24
```

Aquí sí que la variable x está afectada por la asignación entero[0]=18. La razón es porque en este caso el método no recibe el valor de esta variable, sino la **referencia**, es decir la dirección física de esta variable. ***entero*** no es una replica de ***x***, es la propia x llamada de otra forma.

Los tipos básicos (**int, double, char, boolean, float, short** y **byte**) se pasan por valor. También se pasan por valor las variables **String.** Los objetos y arrays se pasan por referencia.

devolución de valores

Los métodos pueden devolver valores básicos (int, short, double, etc.), Strings, arrays e incluso objetos.

En todos los casos es el comando **return** el que realiza esta labor. En el caso de arrays y objetos, devuelve una referencia a ese array u objeto. Ejemplo:

```
class FabricaArrays {
    /**
    * Devuelve un array de valores 1,2,3,4,5
    * @return Un array entero con valores 1,2,3,4,5
        */
    public int[] obtenArray(){
        int array[]= {1,2,3,4,5};
        return array;
    }
}

public class ReturnArray {
    public static void main(String[] args) {
        FabricaArrays fab=new FabricaArrays();
        int nuevoArray[]=fab.obtenArray();
    }
}
```

El ***nuevoArray*** ahora es el array {1,2,3,4,5} creado con el método ***obtenArray***

la referencia this

La palabra **this** es una referencia al propio objeto en el que estamos. Ejemplo:

```
class Punto {
  int posX, posY;//posición del punto
  Punto(posX, posY){
        this.posX=posX;
        this.posY=posY;
  }
```

En el ejemplo hace falta la referencia **this** para clarificar cuando se usan las propiedades ***posX y posY,*** y cuando los argumentos con el mismo nombre. Otro ejemplo:

```
class Punto {
  int posX, posY;
  ...
        /**Suma las coordenadas de otro punto*/
        public void suma(Punto punto2){
              posX = punto2.posX;
              posY = punto2.posY;
        }
        /** Dobla el valor de las coordenadas del punto*/
        public void dobla(){
              suma(this);
        }
```

En el ejemplo anterior, la función dobla, dobla el valor de las coordenadas pasando el propio punto como referencia para la función suma (un punto sumado a sí mismo, daría el doble).

sobrecarga de métodos

Una propiedad de la POO es el polimorfismo. Java posee esa propiedad ya que admite sobrecargar los métodos. Esto significa crear distintas variantes del mismo método. Ejemplo:

```
class Matemáticas{
  public double suma(double x, double y) {
        return x+y;
  }
  public double suma(double x, double y, double z){
```

```
        return x+y+z;
    }
    public double suma(double[] array){
        double total =0;
        for(int i=0; i<array.length;i++){
            total+=array[i];
        }
        return total;
    }
```

La clase matemáticas posee tres versiones del método suma. una versión que suma dos números double, otra que suma tres y la última que suma todos los miembros de un array de ***doubles***. Desde el código se puede utilizar cualquiera de las tres versiones según convenga.

creación de constructores

Un constructor es un método que es llamado automáticamente al crear un objeto de una clase, es decir al usar la instrucción **new**. Sin embargo en ninguno de los ejemplos anteriores se ha definido constructor alguno, por eso no se ha utilizado ningún constructor al crear el objeto.

Un constructor no es más que un método que tiene el mismo nombre que la clase. Con lo cual para crear un constructor basta definir un método en el código de la clase que tenga el mismo nombre que la clase. Ejemplo:

```
class Ficha {
    private int casilla;

    Ficha() { //constructor
        casilla = 1;
    }

    public void avanzar(int n) {
        casilla += n;
    }
    public int casillaActual(){
        return casilla;
    }
}
```

```
public class App {
  public static void main(String[] args) {
        Ficha ficha1 = new Ficha();
        ficha1.avanzar(3);
        System.out.println(ficha1.casillaActual());//Da 4
```

En la línea ***Ficha ficha1 = new Ficha();*** es cuando se llama al constructor, que es el que coloca inicialmente la casilla a 1. Pero el constructor puede tener parámetros:

```
class Ficha {
  private int casilla;
  Ficha() { //constructor por defecto
        casilla = 1;
  }
  Ficha(int n) { //constructor con un parámetro
        casilla = n;
  }
  public void avanzar(int n) {
        casilla += n;
  }
  public int casillaActual(){
        return casilla;
  }
}

public class App {
  public static void main(String[] args) {
        Ficha ficha1 = new Ficha(6);
        ficha1.avanzar(3);
        System.out.println(ficha1.casillaActual());//Da 9
  }
}
```

En este otro ejemplo, al crear el objeto ***ficha1***, se le da un valor a la casilla, por lo que la casilla vale al principio **6**.

Hay que tener en cuenta que puede haber más de un constructor para la misma clase. Al igual que ocurría con los métodos, los constructores se pueden sobrecargar.

De este modo en el código anterior de la clase ***Ficha*** se podrían haber colocado los dos constructores que hemos visto, y sería entonces posible este código:

```
Ficha ficha1= new Ficha(); //La propiedad casilla de la
                           //ficha valdrá 1
Ficha ficha1= new Ficha(6); //La propiedad casilla de la
                           //ficha valdrá 6
```

(8.7.7) métodos y propiedades genéricos (*static*)

Hemos visto que hay que crear objetos para poder utilizar los métodos y propiedades de una determinada clase. Sin embargo esto no es necesario si la propiedad o el método se definen precedidos de la palabra clave **static**. De esta forma se podrá utilizar el método sin definir objeto alguno. Así funciona la clase **Math** (véase

la clase Math, página 185). Ejemplo:

```
class Calculadora {
  static public int factorial(int n) {
        int fact=1;
        while (n>0) {
             fact *=n--;
        }
        return fact;
  }
}
public class App {
  public static void main(String[] args) {
        System.out.println(Calculadora.factorial(5));
  }
}
```

En este ejemplo no ha hecho falta crear objeto alguno para poder calcular el factorial. Una clase puede tener métodos y propiedades genéricos (**static**) y métodos y propiedades dinámicas (normales).

Cada vez que se crea un objeto con **new**, se almacena éste en memoria. Los métodos y propiedades normales, gastan memoria por cada objeto que se cree, sin embargo los métodos estáticos no gastan memoria por cada objeto creado, gastan memoria al definir la clase sólo.

Hay que crear métodos y propiedades genéricos cuando ese método o propiedad vale o da el mismo resultado en todos los objetos. Pero hay que utilizar métodos normales (dinámicos) cuando el método da resultados distintos según el objeto.

(8.7.8) destrucción de objetos

En C y C++ todos los programadores saben que los objetos se crean con **new** y para eliminarles de la memoria y así ahorrarla, se deben eliminar con la instrucción **delete.** Es decir, es responsabilidad del programador eliminar la memoria que gastaban los objetos que se van a dejar de usar. La instrucción **delete** del C++ llama al destructor de la clase, que es una función que se encarga de eliminar adecuadamente el objeto.

La sorpresa de los programadores C++ que empiezan a trabajar en Java es que **no hay instrucción delete en Java.** La duda está entonces, en cuándo se elimina la memoria que ocupa un objeto.

En Java hay una recolección de basura (**garbage**=basura) la que se encarga de gestionar los objetos que se dejan de usar. Este proceso es automático e impredecible y trabajo en un hilo (**thread**) de baja prioridad.

Por lo general ese proceso de recolección de basura, trabaja cuando detecta que un objeto hace demasiado tiempo que no se utiliza en un programa. Esta

eliminación depende de la máquina virtual, en casi todas la recolección se realiza periódicamente en un determinado lapso de tiempo. La implantación de máquina virtual conocida como HotSpot[5] suele hacer la recolección mucho más a menudo

Se puede forzar la eliminación de un objeto asignándole el valor **null**, pero eso no es lo mismo que el famoso **delete** del lenguaje C++; no se libera inmediatamente la memoria, sino que pasará un cierto tiempo (impredecible, por otro lado). Se puede invocar al recolector de basura invocando al método estático **System.gc().** Esto hace que el recolector de basura trabaje en cuanto se lea esa invocación.

Sin embargo puede haber problemas al crear referencias circulares.

Como:

```
class Uno {
  Dos d;
  Uno() { //constructor
       d = new Dos();
  }
}

class Dos {
  Uno u;
  Dos() {
       u = new Uno();
  }
}

public class App {
  public static void main(String[] args)  {
       Uno prueba = new Uno();//referencia circular
       prueba = null; //no se liberará bien la memoria
  }
}
```

Al crear un objeto de clase uno, automáticamente se crea uno de la clase dos, que al crearse creará otro de la clase uno. Eso es un error que provocará que no se libere bien la memoria salvo que se eliminen previamente los objetos referenciados.

[5] Para saber más sobre HotSpot acudir a java.sun.com/products/hotspot/index.html.

el método finalize

Es equivalente a los destructores del C++. Es un método que es llamado antes de eliminar definitivamente al objeto para hacer limpieza final. Un uso puede ser eliminar los objetos creados en la clase para eliminar referencias circulares. Ejemplo:

```
class Uno {
  Dos d;
  Uno() {
        d = new Dos();
  }

protected void finalize(){
        d = null;//d hace referencia a NULL para que eso
                        //quede marcado como basura
  }
}
```

finalize es un método de tipo **protected** heredado por todas las clases ya que está definido en la clase raíz **Object.**

(8.8) reutilización de clases

(8.8.1) herencia

introducción

Es una de las armas fundamentales de la programación orientada a objetos. Permite crear nuevas clases que heredan características presentas en clases anteriores. Esto facilita enormemente el trabajo porque ha permitido crear clases estándar para todos los programadores y a partir de ellas crear nuestras propias clases personales. Esto es más cómodo que tener que crear nuestras clases desde cero.

Para que una clase herede las características de otra hay que utilizar la palabra clave **extends** tras el nombre de la clase. A esta palabra le sigue el nombre de la clase cuyas características se heredarán. Sólo se puede tener herencia de una clase (a la clase de la que se hereda se la llama **superclase** y a la clase heredada se la llama **subclase**). Ejemplo:

```
class Coche extends Vehiculo {
...
} //La clase coche parte de la definición de vehículo
```

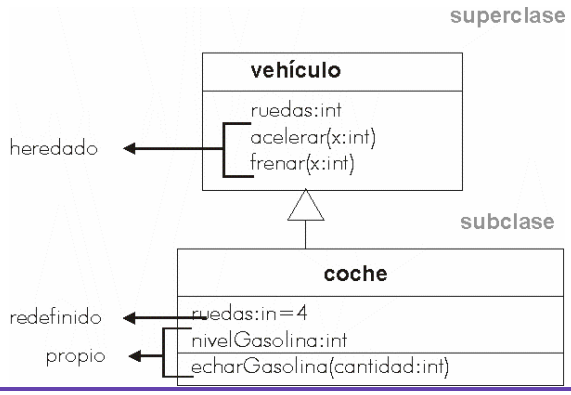


Ilustración 15, Diagrama UML de las clases vehículo y coche

métodos y propiedades no heredados

Por defecto se heredan todos los métodos y propiedades ***friendly, protected*** y ***public*** (no se heredan los ***private***). Además si se define un método o propiedad en la subclase con el mismo nombre que en la superclase, entonces se dice que se está redefiniendo el método, con lo cual no se hereda éste, sino que se reemplaza por el nuevo.

Ejemplo:

```
class Vehiculo {
  public int velocidad;
  public int ruedas;
  public void parar() {
       velocidad = 0;
  }
  public void acelerar(int kmh) {
       velocidad += kmh;
  }

class Coche extends Vehiculo{
  public int ruedas=4;
  public int gasolina;
  public void repostar(int litros) {
       gasolina+=litros;
  }
}
.....................................................
public class App {
  public static void main(String[] args) {
       coche coche1=new coche();
```

```
        coche.acelerar(80);//Método heredado
        coche.repostar(12);
    }
}
```

anulación de métodos

Como se ha visto, las subclases heredan los métodos de las superclases. Pero es más, también los pueden sobrecargar para proporcionar una versión de un determinado método.

Por último, si una subclase define un método con el mismo nombre, tipo y argumentos que un método de la superclase, se dice entonces que se sobrescribe o anula el método de la superclase.

Ejemplo:

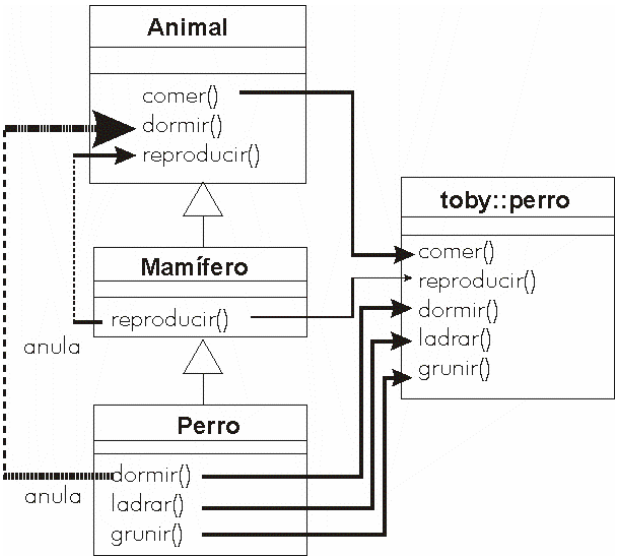


Ilustración 16, anulación de métodos

super

A veces se requiere llamar a un método de la superclase. Eso se realiza con la palabra reservada **super**. Si **this** hace referencia a la clase actual, **super** hace referencia a la superclase respecto a la clase actual, con lo que es un método imprescindible para poder acceder a métodos anulados por herencia. Ejemplo

```
public class Vehiculo{
  double velocidad;
  ...
  public void acelerar(double cantidad){
       velocidad+=cantidad;
```

```
  }
}

public class Coche extends Vehiculo{
  double gasolina;
  public void acelerar(double cantidad){
        super.acelerar(cantidad); gasolina*=0.9;
  }
}
```

En el ejemplo anterior, la llamada ***super.acelerar(cantidad)*** llama al método acelerar de la clase vehículo (el cual acelerará la marcha). Es necesario redefinir el método acelerar en la clase coche ya que aunque la velocidad varía igual que en la superclase, hay que tener en cuenta el consumo de gasolina

Se puede incluso llamar a un **constructor** de una superclase, usando la sentencia **super()**. Ejemplo:

```
public class vehiculo{
  double velocidad;
  public vehiculo(double v){
        velocidad=v;
  }
}

public class coche extends vehiculo{
  double gasolina;
  public coche(double v, double g){
        super(v);//Llama al constructor de la clase Vehiculo
        gasolina=g
  }
}
```

Por defecto Java realiza estas acciones:

Si la primera instrucción de un constructor de una subclase es una sentencia que no es ni **super** ni **this**, Java añade de forma invisible e implícita una llamada **super()** al constructor por defecto de la superclase, luego inicia las variables de la subclase y luego sigue con la ejecución normal.

Si se usa **super(..)** en la primera instrucción, entonces se llama al constructor seleccionado de la superclase, luego inicia las propiedades de la subclase y luego sigue con el resto de sentencias del constructor.

Finalmente, si esa primera instrucción es **this(..)**, entonces se llama al constructor seleccionado por medio de **this**, y después continúa con las

sentencias del constructor. La inicialización de variables la habrá realizado el constructor al que se llamó mediante **this**.

casting de clases

Como ocurre con los tipos básicos (ver conversión entre tipos **(casting)**, página 179, es posible realizar un casting de objetos para convertir entre clases distintas. Lo que ocurre es que sólo se puede realizar este ***casting*** entre subclases. Es decir se realiza un casting para especificar más una referencia de clase (se realiza sobre una superclase para convertirla a una referencia de una subclase suya).

En cualquier otro caso no se puede asignar un objeto de un determinado tipo a otro.

Ejemplo:

```
Vehiculo vehiculo5=new Vehiculo();
Coche cocheDePepe = new Coche("BMW");
vehiculo5=cocheDePepe //Esto sí se permite
cocheDePepe=vehiculo5;//Tipos incompatibles
cocheDepepe=(Coche)vehiculo5;//Ahora sí se permite
```

Hay que tener en cuenta que los objetos nunca cambian de tipo, se les prepara para su asignación pero no pueden acceder a propiedades o métodos que no les sean propios. Por ejemplo, si ***repostar()*** es un método de la clase ***coche*** y no de ***vehículo***:

```
Vehiculo v1=new Vehiculo();
Coche c=new Coche();
v1=c;//No hace falta casting
v1.repostar(5);//¡¡¡Error!!!
```

Cuando se fuerza a realizar un casting entre objetos, en caso de que no se pueda realizar ocurrirá una excepción del tipo **ClassCastingException**

instanceof

Permite comprobar si un determinado objeto pertenece a una clase concreta. Se utiliza de esta forma:

```
objeto instanceof clase
```

Comprueba si el objeto pertenece a una determinada clase y devuelve un valor **true** si es así. Ejemplo:

```
Coche miMercedes=new Coche();
if (miMercedes instanceof Coche)
```

```
  System.out.println("ES un coche");
if (miMercedes instanceof Vehículo)
  System.out.println("ES un coche");
if (miMercedes instanceof Camión)
  System.out.println("ES un camión");
```

En el ejemplo anterior aparecerá en pantalla:

```
ES un coche
ES un vehiculo
```

(8.8.2) clases abstractas

A veces resulta que en las superclases se desean incluir métodos teóricos, métodos que no se desea implementar del todo, sino que sencillamente se indican en la clase para que el desarrollador que desee crear una subclase heredada de la clase abstracta, esté obligado a sobrescribir el método.

A las clases que poseen métodos de este tipo (métodos abstractos) se las llama **clases abstractas.** Son clases creadas para ser heredadas por nuevas clases creadas por el programador. Son clases base para herencia. Las clases abstractas no pueden ser instanciadas (no se pueden crear objetos de las clases abstractas).

Una clase abstracta debe ser marcada con la palabra clave **abstract.** Cada método abstracto de la clase, también llevará el ***abstract***. Ejemplo:

```
abstract class Vehiculo {
  public int velocidad=0;
  abstract public void acelera(); //no la define ésta clase
                                  //deberán definirla las
                                  //clases herederas
  public void para() {velocidad=0;}
}

class Coche extends Vehiculo {
  public void acelera() {
       velocidad+=5;
  }
}

public class Prueba {
  public static void main(String[] args) {
       coche c1=new coche();
       c1.acelera();
       System.out.println(c1.velocidad);
```

```
        c1.para();
        System.out.println(c1.velocidad);
    }
}
```

(8.8.3) final

Se trata de una palabra que se coloca antecediendo a un método, variable o clase. Delante de un método en la definición de clase sirve para indicar que ese método no puede ser sobrescrito por las subclases. Si una subclase intentar sobrescribir el método, el compilador de Java avisará del error.

Si esa misma palabra se coloca delante de una clase, significará que esa clase no puede tener descendencia.

Por último si se usa la palabra **final** delante de la definición de una propiedad de clase, entonces esa propiedad pasará a ser una constante, es decir no se le podrá cambiar el valor en ninguna parte del código.

(8.8.4) relaciones entre clases

relaciones es-a y relaciones tiene-a

Se trata de una diferencia sutil, pero hay dos maneras de que una subclase herede de otra clase.

Siempre que aquí se ha hablado de herencia, en realidad nos referíamos a relaciones del tipo es-a. Veamos un ejemplo:

```
class Uno{
  public void escribe() {
        System.out.println("la clase uno, escribe");
  }
}
class Dos extends Uno{
  Dos() {
        escribe();
  }
}

public class Prueba{
  public static void main(String[] args){
        Dos d= new Dos();//El método escribe
                         //"La clase uno, escribe"
  }
}
```

La clase ***dos*** tiene una relación es-a con la clase ***uno*** ya que es una subclase de uno. Utiliza el método escribe por herencia directa (relación es-a). Este mismo ejemplo en la forma de una relación tiene-a sería:

```
class Uno{
  public void escribe() {
        System.out.println("la clase uno, escribe");
  }
}
class Dos {
  Uno u;
  Dos() {
        u = new Uno();
```

```
        u.escribe();
  }
} //fin class dos

public class Prueba{
  public static void main(String[] args){
        Dos d= new Dos();//El método escribe
                         //"La clase uni escribe
  }
}
```

El resultado es el mismo, pero el método ***escribe*** se ha utilizado, no por herencia, sino por el hecho de que la clase dos **tiene** un objeto de clase ***uno***.

Se habla de relaciones **tiene-a** cuando una clase está contenida en otra.

clases internas

Se llaman clases internas a las clases que se definen dentro de otra clase. Esto permite simplificar aun más el problema de crear programas. Ya que un objeto complejo se puede descomponer en clases más sencillas. Pero requiere esta técnica una mayor pericia por parte del programador.

(8.8.5) interfaces

La limitación de que sólo se puede heredar de una clase, hace que haya problemas ya que muchas veces se deseará heredar de varias clases. Aunque ésta no es la finalidad directa de las interfaces, sí que tiene cierta relación

Mediante interfaces se definen una serie de comportamientos de objeto. Estos comportamientos puede ser "implementados" en una determinada clase. No definen el tipo de objeto que es, sino lo que puede hacer (sus capacidades). Por ello lo normal es que las interfaces terminen con el texto "**able**" (***configurable***, ***modificable, cargable***).

Por ejemplo en el caso de la clase Coche, esta deriva de la superclase Vehículo, pero además puesto que es un vehículo a motor, puede implementar métodos de una interfaz llamado por ejemplo **arrancable.**

utilizar interfaces

Para hacer que una clase utilice una interfaz, se añade detrás del nombre de la clase la palabra **implements** seguida del nombre del interfaz. Se pueden poner varios nombres de interfaces separados por comas (solucionando, en cierto modo, el problema de la herencia múltiple).

```
class Coche extends Vehiculo implements Arrancable {
  public void arrancar (){
       ....
  }
  public void detenerMotor(){         ....      }
```

Hay que tener en cuenta que la interfaz ***arrancable*** no tiene porque tener ninguna relación con la clase vehículo, es más se podría implementar el interfaz ***arrancable*** a una bomba de agua.

creación de interfaces

Una interfaz en realidad es una serie de **constantes y métodos abstractos**. Cuando una clase implementa un determinado interfaz puede anular los métodos abstractos de éste, redefiniéndolos en la propia clase.

Una interfaz se crea exactamente igual que una clase (se crean en archivos propios también), la diferencia es que la palabra **interface** sustituye a la palabra **class** y que sólo se pueden definir en un interfaz constantes y métodos abstractos.

Todas las interfaces son abstractas y sus métodos también son todos abstractos y públicos. Las variables se tienen obligatoriamente que inicializar. Ejemplo:

```
interface Arrancable(){
  boolean motorArrancado=false;
  void arrancar();
  void detenerMotor();
}
```

Los métodos son simples prototipos y la variable se considera una constante (a no ser que se redefina en una clase que implemente esta interfaz)

subinterfaces

Una interfaz puede heredarse de otra interfaz, como por ejemplo en:

```
interface Dibujable extends Escribible, Pintable {…
```

dibujable es subinterfaz de ***escribible*** y ***pintable.*** Es curioso, pero los interfaces sí admiten herencia múltiple.

variables de interfaz

Al definir una interfaz, se pueden crear después variables de interfaz. Se puede interpretar esto como si el interfaz fuera un tipo especial de datos (que no de clase). La ventaja que proporciona esto es que pueden asignarse variables interfaz a cualquier objeto que tenga en su clase implementada la interfaz. Esto permite cosas como:

```
Arrancable motorcito; //motorcito es una variable de tipo
                      // arrancable
Coche c=new Coche(); //Objeto de tipo coche
BombaAgua ba=new BombaAgua(); //Objeto de tipo BombaAgua
motorcito=c; //Motorcito apunta a c
motorcito.arrancar() //Se arrancará c
motorcito=ba; //Motorcito apunta a ba
motorcito=arrancar; //Se arranca la bomba de agua
```

El juego que dan estas variables es impresionante, debido a que fuerzan acciones sobre objetos de todo tipo, y sin importar este tipo; siempre y cuando estos objetos tengan implementados los métodos del interfaz de la variable.

interfaces como funciones de retroinvocación

En C++ una función de retroinvocación es un puntero que señala a un método o a un objeto. Se usan para controlar eventos. En Java se usan interfaces para este fin. Ejemplo:

```
interface Escribible {
  void escribe(String texto);
}

class Texto implements Escribible {
  ...
  public void escribe(texto){
       System.out.println(texto);
  }
}
class Prueba {
  Escribible escritor;
  public Prueba(Escribible e){
       escritor=e;
  }
  public void enviaTexto(String s){
       escritor.escribe(s);
  }}
```

En el ejemplo ***escritor*** es una variable de la interfaz ***Escribible***, cuando se llama a su método ***escribe***, entonces se usa la implementación de la clase ***texto***.

(8.8.6) creación de paquetes

Un paquete es una colección de clases e interfaces relacionadas. El compilador de Java usa los paquetes para organizar la compilación y ejecución. Es decir, un paquete es una biblioteca. Mediante el comando **import** (visto anteriormente), se permite utilizar una determinada clase en un programa. Esta sentencia se coloca arriba del código de la clase.

```
import ejemplos.tema5.vehiculo;
import ejemplos.tema8.*
//Usa todas las clase del paquete  tema8
```

Cuando desde un programa se hace referencia a una determinada clase se busca ésta en los paquetes que se han importado al programa. Si ese nombre de clase se ha definido en un solo paquete, se usa. Si no es así podría haber ambigüedad por ello se debe usar un prefijo delante de la clase con el nombre del paquete. Es decir:

```
paquete.clase
```

O incluso:

```
paquete1.paquete2......clase
```

En el caso de que el paquete sea subpaquete de otro más grande.

Las clases son visibles en el mismo paquete a no ser que se las haya declarado con el modificador **private.** Para que sean visible para cualquier clase de cualquier paquete, deben declararse con **public.**

organización de los paquetes

Los paquetes en realidad son subdirectorios cuyo raíz debe ser absolutamente accesible por el sistema operativo. Para ello a veces es necesario usar la variable de entorno **CLASSPATH** de la línea de comandos. Esta variable se suele definir en el archivo **autoexec.bat** o en MI PC en el caso de las últimas versiones de Windows

Así para el paquete **prueba.reloj** tiene que haber una carpeta prueba, dentro de la cual habrá una carpeta reloj.

Una clase se declara perteneciente aun determinado paquete usando la instrucción **package** al principio del código:

```
//Clase perteneciente al paquete tema5 que está en
ejemplos
package ejemplos.tema5;
```

(8.9) excepciones

(8.9.1) introducción a las excepciones

Uno de los problemas más importantes al escribir aplicaciones es el tratamiento de los errores. Errores no previstos que distorsionan la ejecución del programa. Las **excepciones** de Java hacen referencia e este hecho. Se denomina excepción a una situación que no se puede resolver y que provoca la detención del programa; es decir una condición de error en tiempo de ejecución (es decir cuando el programa ya ha sido compilado y se está ejecutando). Ejemplos:

- El archivo que queremos abrir no existe
- Falla la conexión a una red
- La clase que se desea utilizar no se encuentra en ninguno de los paquetes reseñados con **import**

Los errores de sintaxis son detectados durante la compilación. Pero las excepciones pueden provocar situaciones irreversibles, su control debe hacerse en tiempo de ejecución y eso presenta un gran problema. En Java se puede preparar el código susceptible a provocar errores de ejecución de modo que si ocurre una excepción, el código es ***lanzado (throw)*** a una determinada rutina previamente preparada por el programador, que permite manipular esa excepción. Si la excepción no fuera capturada, la ejecución del programa se detendría irremediablemente.

Con formato: Español (España - alfab. internacional)

En Java hay muchos tipos de excepciones (de operaciones de entrada y salida, de operaciones irreales. El paquete **java.lang.Exception** y sus subpaquetes contienen todos los tipos de excepciones.

Cuando se produce un error se genera un objeto asociado a esa excepción. Este objeto es de la clase **Exception** o de alguna de sus herederas. Este objeto se pasa al código que se ha definido para manejar la excepción. Dicho código puede manipular las propiedades del objeto Exception.

Con formato: Inglés (Estados Unidos)

Hay una clase, la **java.lang.Error** y sus subclases que sirven para definir los errores irrecuperables más serios. Esos errores causan parada en el programa, por lo que el programador no hace falta que los manipule. Estos errores les produce el sistema y son incontrolables para el programador. Las excepciones son fallos más leves, y más manipulables.

(8.9.2) try y catch

Las sentencias que tratan las excepciones son **try** y **catch.** La sintaxis es:

```
try {
        instrucciones que se ejecutan salvo que haya un
error
}
catch (ClaseExcepción objetoQueCapturaLaExcepción) {
        instrucciones que se ejecutan si hay un error
}
```

Con formato: Sintaxis

Puede haber más de una sentencia **catch** para un mismo bloque **try.** Ejemplo:

```
try {
        readFromFile("arch");
        ...
}
catch(FileNotFoundException e) {
        //archivo no encontrado
...
}
catch (IOException e) {
...
}
```

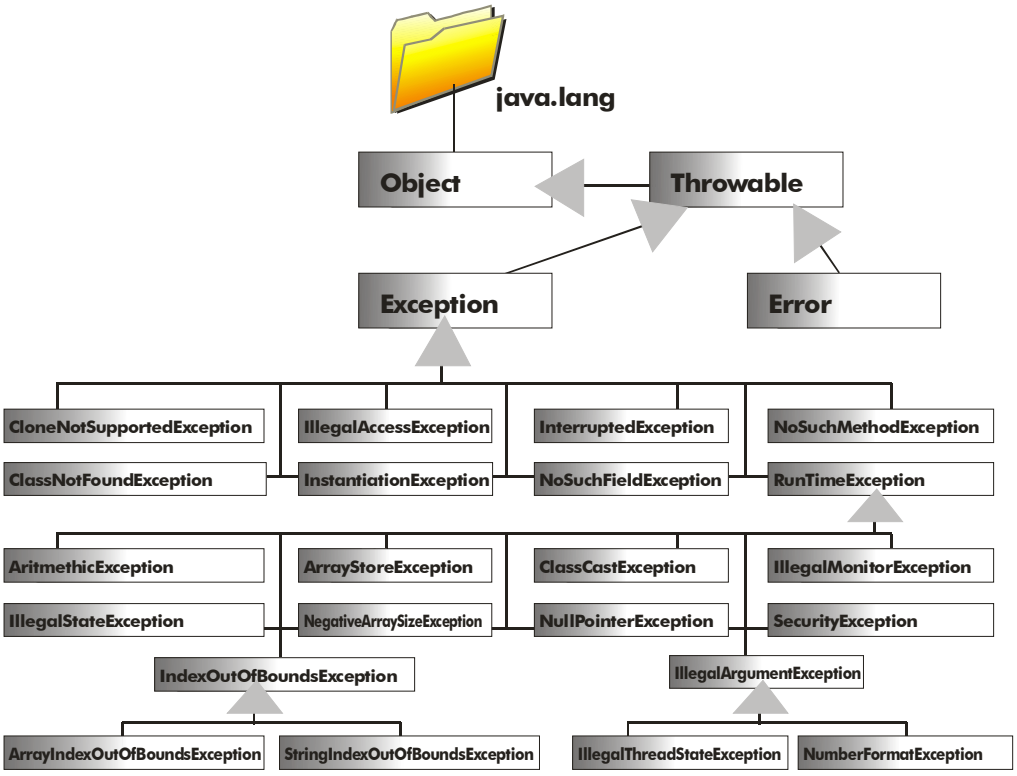


Ilustración 17, Jerarquía de las clases de manejo de excepciones

}

Dentro del bloque **try** se colocan las instrucciones susceptibles de provocar una excepción, el bloque **catch** sirve para capturar esa excepción y evitar el fin de la ejecución del programa. Desde el bloque catch se maneja, en definitiva, la excepción.

Cada catch maneja un tipo de excepción. Cuando se produce una excepción, se busca el catch que posea el manejador de excepción adecuado, será el que utilice el mismo tipo de excepción que se ha producido. Esto puede causar problemas si no se tiene cuidado, ya que la clase **Exception** es la superclase de todas las demás. Por lo que si se produjo, por ejemplo, una excepción de tipo **AritmethicException** y el primer **catch** captura el tipo genérico **Exception**, será ese catch el que se ejecute y no los demás.

Por eso el último catch debe ser el que capture excepciones genéricas y los primeros deben ser los más específicos. Lógicamente si vamos a tratar a todas las excepciones (sean del tipo que sean) igual, entonces basta con un solo **catch** que capture objetos Exception.

(8.9.3) manejo de excepciones

Siempre se debe controlar una excepción, de otra forma nuestro software está a merced de los fallos. En la programación siempre ha habido dos formas de manejar la excepción:

- **Interrupción.** En este caso se asume que el programa ha encontrado un error irrecuperable. La operación que dio lugar a la excepción se anula y se entiende que no hay manera de regresar al código que provocó la excepción. Es decir, la operación que dio pies al error, se anula.
- **Reanudación.** Se puede manejar el error y regresar de nuevo al código que provocó el error.

La filosofía de Java es del tipo **interrupción**, pero se puede intentar emular la reanudación encerrando el bloque **try** en un **while** que se repetirá hasta que el error deje de existir. Ejemplo:

```
boolean indiceNoValido=true;
int i; //Entero que tomará nos aleatorios de 0 a 9
String texto[]={"Uno","Dos","Tres","Cuatro","Cinco"};
while(indiceNoValido){
  try{
       i=Math.round(Math.random()*9);
       System.out.println(texto[i];
       indiceNoValido=false;
  }catch(ArrayIndexOutOfBoundsException exc){
       System.out.println("Fallo en el índice");
  }
}
```

En el código anterior, el índice ***i*** calcula un número del 0 al 9 y con ese número el código accede al array ***texto*** que sólo contiene 5 elementos. Esto producirá muy a menudo una excepción del tipo ***ArrayIndexOutOfBoundsException*** que es manejada por el ***catch*** correspondiente. Normalmente no se continuaría intentando. Pero como tras el bloque ***catch*** está dentro del **while**, se hará otro intento y así hasta que no haya excepción, lo que provocará que ***indiceNovalido*** valga ***true*** y la salida, al fin, del while.

Como se observa en la **¡Error! No se encuentra el origen de la referencia.**, la clase **Exception** es la superclase de todos los tipos de excepciones.

Esto permite utilizar una serie de métodos comunes a todas las clases de excepciones:

- **String getMessage().** Obtiene el mensaje descriptivo de la excepción o una indicación específica del error ocurrido:

```
try{
   ....
} catch (IOException ioe){
   System.out.println(ioe.getMessage());
}
```

- **String toString().** Escribe una cadena sobre la situación de la excepción. Suele indicar la clase de excepción y el texto de **getMessage().**
- **void printStackTrace()**. Escribe el método y mensaje de la excepción (la llamada información de pila). El resultado es el mismo mensaje que muestra el ejecutor (la máquina virtual de Java) cuando no se controla la excepción.

(8.9.4) throws

Al llamar a métodos, ocurre un problema con las excepciones. El problema es, si el método da lugar a una excepción, ¿quién la maneja? ¿El propio método?¿O el código que hizo la llamada al método?

Con lo visto hasta ahora, sería el propio método quien se encargara de sus excepciones, pero esto complica el código. Por eso otra posibilidad es hacer que la excepción la maneje el código que hizo la llamada.

Esto se hace añadiendo la palabra **throws** tras la primera línea de un método. Tras esa palabra se indica qué excepciones puede provocar el código del método. Si ocurre una excepción en el método, el código abandona ese método y regresa al código desde el que se llamó al método. Allí se posará en el catch apropiado para esa excepción. Ejemplo:

```
void usarArchivo (String archivo) throws IOException,
InterruptedException {...
```

En este caso se está indicando que el método ***usarArchivo*** puede provocar excepciones del tipo ***IOException*** y ***InterruptedException***. Esto significará, además, que el que utilice este método debe preparar el ***catch*** correspondiente para manejar los posibles errores.

Ejemplo:

```
try{
   ...
   objeto.usarArchivo("C:\texto.txt");//puede haber
excepción
   ..
}
catch(IOException ioe){...
}
catch(InterruptedException ie){...
}
...//otros catch para otras posibles excepciones
```

(8.9.5) throw

Esta instrucción nos permite lanzar a nosotros nuestras propias excepciones (o lo que es lo mismo, crear artificialmente nosotros las excepciones). Ante:

```
throw new Exception();
```

El flujo del programa se dirigirá a la instrucción ***try/catch*** más cercana. Se pueden utilizar constructores en esta llamada (el formato de los constructores depende de la clase que se utilice):

```
throw new Exception("Error grave, grave");
```

Eso construye una excepción con el mensaje indicado.

throw permite también ***relanzar*** excepciones. Esto significa que dentro de un catch podemos colocar una instrucción **throw** para lanzar la nueva excepción que será capturada por el catch correspondiente:

```
try{
   ...
} catch(ArrayIndexOutOfBoundsException exc){
   throw new IOException();
} catch(IOException ioe){
   ...
}
```

El segundo catch capturará también las excepciones del primer tipo

(8.9.6) finally

La cláusula ***finally*** está pensada para limpiar el código en caso de excepción. Su uso es:

```
try{
  ...
}catch (FileNotFoundException fnfe){
  ...
}catch(IOException ioe){
  ...
}catch(Exception e){
  ...
}finally{
  ...//Instrucciones de limpieza
}
```

Las sentencias finally se ejecutan tras haberse ejecutado el catch correspondiente. Si ningún catch capturó la excepción, entonces se ejecutarán esas sentencias antes de devolver el control al siguiente nivel o antes de romperse la ejecución.

Hay que tener muy en cuenta que **las sentencias finally se ejecutan independientemente de si hubo o no excepción**. Es decir esas sentencias se ejecutan siempre, haya o no excepción. Son sentencias a ejecutarse en todo momento. Por ello se coloca en el bloque ***finally*** código común para todas las excepciones (y también para cuando no hay excepciones.

(8.10) clases fundamentales (I)

(8.10.1) la clase Object

Todas las clases de Java poseen una superclase común, esa es la clase **Object**. Por eso los métodos de la clase Object son fundamentales ya que todas las clases los heredan. Esos métodos están pensados para todas las clases, pero hay que redefinirlos para que funcionen adecuadamente.

Es decir, Object proporciona métodos que son heredados por todas las clase. La idea es que todas las clases utilicen el mismo nombre y prototipo de método para hacer operaciones comunes como comprobar igualdad, clonar, ... y para ello habrá que redefinir esos métodos a fin de que se ajusten adecuadamente a cada clase.

comparar objetos

La clase Object proporciona un método para comprobar si dos objetos son iguales. Este método es **equals.** Este método recibe como parámetro un objeto con quien comparar y devuelve **true** si los dos objetos son iguales.

No es lo mismo ***equals*** que usar la comparación de igualdad. Ejemplos:

```
Coche uno=new Coche("Renault","Megane","P4324K");
Coche dos=uno;
boolean resultado=(uno.equals(dos)); //Resultado valdrá
true
resultado=(uno==dos); //Resultado también valdrá true
dos=new Coche("Renault","Megane","P4324K");
resultado=(uno.equals(dos)); //Resultado valdrá true
resultado=(uno==dos); //Resultado ahora valdrá false
```

En el ejemplo anterior **equals** devuelve **true** si los dos coches tienen el mismo modelo, marca y matrícula . El operador "==" devuelve **true** si los dos objetos se refieren a la misma cosa (las dos referencias apuntan al mismo objeto).

Realmente en el ejemplo anterior la respuesta del método equals sólo será válida si en la clase que se está comparando (***Coche*** en el ejemplo) se ha redefinido el método equals. Esto no es opcional sino **obligatorio si se quiere usar este método.** El resultado de equals depende de cuándo consideremos nosotros que devolver verdadero o falso. En el ejemplo anterior el método equals sería:

```
public class Coche extends Vehículo{
  public boolean equals (Object o){
       if ((o!=null) && (o instanceof Coche)){
            if (((Coche)o).matricula==matricula &&
             ((Coche)o).marca==marca
                 && ((Coche)o).modelo==modelo))
             return true
       }
       return false; //Si no se cumple todo lo anterior
  }
}
```

Es necesario el uso de **instanceOf** ya que equals puede recoger cualquier objeto ***Object***. Para que la comparación sea válida primero hay que verificar que el objeto es un coche. El argumento **o** siempre hay que convertirlo al tipo Coche para utilizar sus propiedades de Coche.

código hash

El método **hashCode()** permite obtener un número entero llamado **código hash.** Este código es un entero único para cada objeto que se genera aleatoriamente según su contenido. No se suele redefinir salvo que se quiera anularle para modificar su función y generar códigos hash según se desee.

clonar objetos

El método **clone** está pensado para conseguir una copia de un objeto. Es un método **protected** por lo que sólo podrá ser usado por la propia clase y sus descendientes, salvo que se le redefina con **public**.

Además si una determinada clase desea poder clonar sus objetos de esta forma, debe implementar la interfaz **Cloneable** (perteneciendo al paquete java.lang), que no contiene ningún método pero sin ser incluída al usar clone ocurriría una excepción del tipo **CloneNotSupportedException.** Esta interfaz es la que permite que el objeto sea clonable.

Ejemplo:

```
public class Coche extends Vehiculo implements
Arrancable, Cloneable{
  public Object clone(){
       try{
            return (super.clone());
       }catch(CloneNotSupportedException cnse){
         System.out.println("Error inesperado en clone");
         return null;
       }
}
....
//Clonación
        Coche uno=new Coche();
        Coche dos=(Coche)uno.clone();
```

En la última línea del código anterior, el cast “(Coche)” es obligatorio ya que clone devuelve forzosamente un objeto tipo Object. Aunque este código generaría dos objetos distintos, el código hash sería el mismo.

método toString

Este es un método de la clase Object que da como resultado un texto que describe al objeto. la utiliza, por ejemplo el método **println** para poder escribir un método por pantalla. Normalmente en cualquier clase habría que definir el método **toString**. Sin redefinirlo el resultado podría ser:

```
Coche uno=new Coche();
System.out.println(uno);//Escribe: Coche@26e431
```

Si redefinimos este método en la clase Coche:

```
public String toString(){
  return("Velocidad :"+velocidad+"\nGasolina: "+gasolina);
}
```

Ahora en el primer ejemplo se escribiría la velocidad y la gasolina del coche.

lista completa de métodos de la clase Object

método	significado
protected Object clone()	Devuelve como resultado una copia del objeto.
boolean equals(Object obj)	Compara el objeto con un segundo objeto que es pasado como referencia (el objeto *obj*). Devuelve true si son iguales.
protected void finalize()	Destructor del objeto
Class getClass()	Proporciona la clase del objeto
int hashCode()	Devuelve un valor *hashCode* para el objeto
void notify()	Activa un hilo (thread) sencillo en espera.
void notifyAll()	Activa todos los hilos en espera.
String toString()	Devuelve una cadena de texto que representa al objeto
void wait()	Hace que el hilo actual espere hasta la siguiente notificación
void wait(long tiempo)	Hace que el hilo actual espere hasta la siguiente notificación, o hasta que pase un determinado tiempo
void wait(long tiempo, int nanos)	Hace que el hilo actual espere hasta la siguiente notificación, o hasta que pase un determinado tiempo o hasta que otro hilo interrumpa al actual

(8.10.2) clases para tipos básicos

En Java se dice que todo es considerado un objeto. Para hacer que esta filosofía sea más real se han diseñado una serie de clases relacionadas con los tipos básicos. El nombre de estas clases es:

clase	representa al tipo básico..
java.lang.Void	void
java.lang.Boolean	boolean
java.lang.Character	char
java.lang.Byte	byte
java.lang.Short	short
java.lang.Integer	int
java.lang.Long	long
java.lang.Float	float
java.lang.Double	double

Hay que tener en cuenta que no son equivalentes a los tipos básicos. La creación de estos tipos lógicamente requiere usar constructores, ya que son objetos y no tipos básicos.

```
Double n=new Double(18.3);
Double o=new Double("18.5");
```

El constructor admite valores del tipo básico relacionado e incluso valores String que contengan texto convertible a ese tipo básico. Si ese texto no es convertible, ocurre una excepción del tipo **NumberFormatException**.

La conversión de un String a un tipo básico es una de las utilidades básicas de estas clases, por ello estas clases poseen el método estático **valueOf** entre otros para convertir un String en uno de esos tipos. Ejemplos:

```
String s="2500";
Integer a=Integer.valueOf(s);
Short b=Short.valueOf(s);
Double c=Short.valueOf(s);
Byte d=Byte.valueOf(s);//Excepción!!!
```

Hay otro método en cada una de esas clases que se llama **parse**. La diferencia estriba en que en los métodos **parse** la conversión se realiza hacia tipos básicos (int, double, float, boolean,...) y no hacia las clase anteriores.

Ejemplo:

```
String s="2500";
int y=Integer.parseInt(s);
short z=Short.parseShort(s);
double c=Short.parseDouble(s);
byte x=Byte.parseByte(s);
```

Estos métodos son todos estáticos. Todas las clases además poseen métodos dinámicos para convertir a otros tipos (intValue, longValue,... o el conocido toString).

Todos estos métodos lanzan excepciones del tipo **NumberFormatException,** que habrá que capturar con el try y el catch pertinentes.

Además han redefinido el método **equals** para comparar objetos de este tipo. Además poseen el método **compareTo** que permite comparar dos elementos de este tipo (este método se maneja igual que el **compareTo** de la clase **String**,

(8.10.3) números aleatorios

La clase **java.util.Random** está pensada para la producción de elementos aleatorios. Los números aleatorios producen dicha aleatoriedad usando una fórmula matemática muy compleja que se basa en, a partir de un determinado número obtener aleatoriamente el siguiente. Ese primer número es la semilla.

El constructor por defecto de esta clase crea un número aleatorio utilizando una semilla obtenida a partir de la fecha y la hora. Pero si se desea repetir continuamente la misma semilla, se puede iniciar usando un determinado número **long**:

```
Random r1=Random();//Semilla obtenida de la fecha y hora
Random r2=Random(182728L);//Semilla obtenida de un long
```

métodos de Random

método	devuelve
boolean **nextBoolean()**	**true** o **false** aleatoriamente
int **nextInt()**	Un **int**
int **nextInt**(int n)	Un número entero de 0 a ***n-1***
long **nextLong()**	Un **long**
float **nextFloat()**	Número decimal de -1,0 a 1.0
double **nextDouble()**	Número doble de -1,0 a 1.0
void **setSeed**(long semilla)	Permite cambiar la semilla.

(8.10.4) fechas

Sin duda alguna el control de fechas y horas es uno de los temas más pesados de la programación. Por ello desde Java hay varias clase dedicadas a su control.

La clase **java.util.Calendar** permite usar datos en forma de día mes y año, su descendiente **java.util.GregorianCalendar** añade compatibilidad con el calendario Gregoriano, la clase **java.util.Date** permite trabajar con datos que representan un determinado instante en el tiempo y la clase **java.text.DateFormat** está encargada de generar distintas representaciones de datos de fecha y hora.

clase Calendar

Se trata de una clase abstracta (no se pueden por tanto crear objetos Calendar) que define la funcionalidad de las fechas de calendario y define una serie de atributos estáticos muy importante para trabajar con las fechas. Entre ellos (se usan siempre con el nombre Calendar, por ejemplo Calendar.DAY_OF_WEEK):

- **Día de la semana: DAY_OF_WEEK** número del día de la semana (del 1 al 7). Se pueden usar las constantes MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY. Hay que tener en cuenta que usa el calendario inglés, con lo que el día 1 de la semana es el domingo (SUNDAY).
- **Mes: MONTH** es el mes del año (del 0, enero, al 11, diciembre). Se pueden usar las constantes: JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER.
- **Día del mes: DAY_OF_MONTH** número del día del mes (empezando por 1).
- **Semana del año: WEEK_OF_YEAR** indica o ajusta el número de semana del año.
- **Semana del mes: WEEK_OF_MONTH** indica o ajusta el número de semana del mes.
- **Día del año: DAY_OF_YEAR** número del día del año (de 1 a 365).
- **Hora: HOUR**, hora en formato de 12 horas. **HOUR_OF_DAY** hora en formato de 24 horas.
- **AM_PM.** Propiedad que sirve para indicar en qué parte del día estamos, AM o PM.
- **Minutos.** MINUTE
- **Segundos.** SECOND también se puede usar MILLISECOND para los milisegundos.

Esta clase también define una serie de métodos abstractos y estáticos.

clase GregorianCalendar

Es subclase de la anterior (por lo que hereda todos sus atributos). Permite crear datos de calendario gregoriano. Tiene numerosos constructores, algunos de ellos son:

```
GregorianCalendar fecha1=new GregorianCalendar();
  //Crea fecha1 con la fecha actual
GregorianCalendar fecha2=new GregorianCalendar(2003,7,2);
  //Crea fecha2 con fecha 2 de agosto de 2003
GregorianCalendar fecha3=new
GregorianCalendar(2003,Calendar.AUGUST,2);
  //Igual que la anterior
GregorianCalendar fecha4=new
GregorianCalendar(2003,7,2,12,30);
  //2 de Agosto de 2003 a las 12:30
GregorianCalendar fecha5=new
GregorianCalendar(2003,7,2,12,30,15);
  //2 de Agosto de 2003 a las 12:30:15
```

método get

El método **get** heredado de la clase **Calendar** sirve para poder obtener un detalle de una fecha. A este método se le pasa el atributo a obtener (véase lista de campos en la clase **Calendar**). Ejemplos:

```
GregorianCalendar fecha=new
  GregorianCalendar(2003,7,2,12,30,23);
System.out.println(fecha.get(Calendar.MONTH));
System.out.println(fecha.get(Calendar.DAY_OF_YEAR));
System.out.println(fecha.get(Calendar.SECOND));
System.out.println(fecha.get(Calendar.MILLISECOND));
/* La salida es
  7
  214
  23
  0
*/
```

método set

Es el contrario del anterior, sirve para modificar un campo del objeto de calendario. Tiene dos parámetros: el campo a cambiar (MONTH, YEAR,...) y el valor que valdrá ese campo:

```
fecha.set(Calendar.MONTH, Calendar.MAY);
fecha.set(Calendar.DAY_OF_MONTH, 12)
```

Otro uso de set consiste en cambiar la fecha indicando, año, mes y día y, opcionalmente, hora y minutos.

```
fecha.set(2003,17,9);
```

método getTime

Obtiene el objeto **Date** equivalente a al representado por el GregorianCalendar. En Java los objetos Date son fundamentales para poder dar formato a las fechas.

método setTime

Hace que el objeto de calendario tome como fecha la representada por un objeto date. Es el inverso del anterior

```
Date d=new Date()
GregorianCalendar gc=new GregorianCalendar()
g.setTime(d);
```

método getTimeInMillis

Devuelve el número de milisegundos que representa esa fecha.

clase Date

Representa una fecha en forma de milisegundos transcurridos, su idea es representar un instante. Cuenta fechas desde el 1900. Normalmente se utiliza conjuntamente con la clase **GregorianCalendar.** Pero tiene algunos métodos interesantes.

construcción

Hay varias formas de crear objetos Date:

```
Date fecha1=new Date();//Creado con la fecha actual
Date fecha2=(new GregorianCalendar(2004,7,6)).getTime();
```

método after

Se le pasa como parámetro otro objeto Date. Devuelve **true** en el caso de que la segunda fecha no sea más moderna. Ejemplo:

```
GregorianCalendar gc1=new GregorianCalendar(2004,3,1);
GregorianCalendar gc2=new GregorianCalendar(2004,3,10);
Date fecha1=gc1.getTime();
Date fecha2=gc2.getTime();
System.out.println(fecha1.after(fecha2));
//Escribe false porque la segunda fecha es más reciente
```

método before

Inverso al anterior. Devuelve **true** si la fecha que recibe como parámetro no es más reciente.

métodos **equals** y **compareTo**

Funcionan igual que en otros muchos objetos. **equals** devuelve **true** si la fecha con la que se compara es igual que la primera (incluidos los milisegundos). **compareTo** devuelve -1 si la segunda fecha es más reciente, 0 si son iguales y 1 si es más antigua.

clase DateFormat

A pesar de la potencia de las clases relacionadas con las fechas vistas anteriormente, sigue siendo complicado y pesado el hecho de hacer que esas fechas aparezcan con un formato más legible por un usuario normal.

La clase DateFormat nos da la posibilidad de formatear las fechas. Se encuentra en el paquete **java.text**. Hay que tener en cuenta que no representa fechas, sino maneras de dar formato a las fechas. Es decir un objeto DateFormat representa un formato de fecha (formato de fecha larga, formato de fecha corta,...).

creación básica

Por defecto un objeto DateFormat con opciones básicas se crea con:

```
DateFormat sencillo=DateFormat.getInstance();
```

Eso crea un objeto DateFormat con formato básico. **getInstance()** es un método estático de la clase **DateFormat** que devuelve un objeto DateFormat con formato sencillo.

el método **format**

Todos los objetos DateFormat poseen un método llamado **format** que da como resultado una cadena String y que posee como parámetro un objeto de tipo Date. El texto devuelto representa la fecha de una determinada forma. El formato es el indicado durante la creación del objeto DateFormat. Ejemplo:

```
Date fecha=new Date();//fecha actual
DateFormat df=DateFormat.getInstance();//Formato básico
System.out.println(df.format(fecha);
//Ejemplo de resultado: 14/04/04 10:37
```

creaciones de formato sofisticadas

El formato de fecha se puede configurar al gusto del programador. Esto es posible ya que hay otras formas de crear formatos de fecha. Todas las opciones consisten en utilizar los siguientes métodos estáticos (todos ellos devuelven un objeto DateFormat):

- **DateFormat.getDateInstance.** Crea un formato de fecha válido para escribir sólo la fecha; sin la hora.
- **DateFormat.getTimeInstance.** Crea un formato de fecha válido para escribir sólo la hora; sin la fecha.

- **DateFormat.getDateTimeInstance**. Crea un formato de fecha en el que aparecerán la fecha y la hora.

Todos los métodos anteriores reciben un parámetro para indicar el formato de fecha y de hora (el último método recibe dos: el primer parámetro se refiere a la fecha y el segundo a la hora). Ese parámetro es un número, pero es mejor utilizar las siguientes constantes estáticas:

- **DateFormat.SHORT.** Formato corto.
- **DateFormat.MEDIUM**. Formato medio
- **DateFormat.LONG .** Formato largo.
- **DateFormat.FULL**. Formato completo

Ejemplo:

```
DateFormat
df=DateFormat.getDateInstance(DateFormat.LONG);
System.out.println(df.format(new Date()));
//14 de abril de 2004
DateFormat
  df2=DateFormat.getDateTimeInstance(DateFormat.LONG);
System.out.println(df2.format(new Date()));
// 14/04/04 00H52' CEST
```

La fecha sale con el formato por defecto del sistema (por eso sale en español si el sistema Windows está en español).

método parse

Inverso al método format. Devuelve un objeto Date a partir de un String que es pasado como parámetro. Este método lanza excepciones del tipo **ParseException** (clase que se encuentra en el paquete **java.text**), que estamos obligados a capturar. Ejemplo:

```
DateFormat df=DateFormat.getDateTimeInstance(
  DateFormat.SHORT,DateFormat.FULL);
try{
  fecha=df2.parse("14/3/2004 00H23' CEST");
}
catch(ParseException pe){
  System.out.println("cadena no válida");
}
```

Obsérvese que el contenido de la cadena debe ser idéntica al formato de salida del objeto DateFormat de otra forma se generaría la excepción.

Es un método muy poco usado.

(8.10.5) cadenas delimitadas. StringTokenizer

introducción

Se denomina cadena delimitada a aquellas que contienen texto que está dividido en partes (**tokens**) y esas partes se dividen mediante un carácter (o una cadena) especial. Por ejemplo la cadena 7647-34-123223-1-234 está delimitada por el guión y forma 5 tokens.

Es muy común querer obtener cada zona delimitada, cada **token**, de la cadena. Se puede hacer con las clases que ya hemos visto, pero en el paquete **java.util** disponemos de la clase más apropiada para hacerlo, **StringTokenizer.**

Esa clase representa a una cadena delimitada de modo además que en cada momento hay un puntero interno que señala al siguiente ***token*** de la cadena. Con los métodos apropiados podremos avanzar por la cadena.

construcción

La forma común de construcción es usar dos parámetros: el texto delimitado y la cadena delimitadora. Ejemplo:

```
StringTokenizer st=
  new StringTokenizer(“1234-5-678-9-00”,”-“);
```

Se puede construir también el ***tokenizer*** sólo con la cadena, sin el delimitador. En ese caso se toma como delimitador el carácter de nueva línea (\n), el retorno de carro (\r), el tabulador (\t) o el espacio. Los ***tokens*** son considerados sin el delimitador (en el ejemplo sería 1234, 5, 678, 9 y 00, el guión no cuenta).

uso

Para obtener las distintas partes de la cadena se usan estos métodos:

- **String nextToken().** Devuelve el siguiente token. La primera vez devuelve el primer texto de la cadena hasta la llegada del delimitador. Luego devuelve el siguiente texto delimitado y así sucesivamente. Si no hubiera más tokens devuelve la excepción **NoSuchElementException**. Por lo que conviene comprobar si hay más tokens.
- **boolean hasMoreTokens().** Devuelve **true** si hay más tokens en el objeto **StringTokenizer.**
- **int countTokens().** Indica el número de tokens que quedan por obtener. El puntero de tokens no se mueve.

Ejemplo:

```
String tokenizada="10034-23-43423-1-3445";
StringTokenizer st=new StringTokenizer(tokenizada,"-");
while (st.hasMoreTokens()){
  System.out.println(st.nextToken());
}// Obtiene:10034 23 43423 1 y 3445
```

(8.11) entrada y salida en Java

El paquete **java.io** contiene todas las clases relacionadas con las funciones de entrada (**input**) y salida (**output**). Se habla de E/S (o de I/O) refiriéndose a la entrada y salida. En términos de programación se denomina **entrada** a la posibilidad de introducir datos hacia un programa; **salida** sería la capacidad de un programa de mostrar información al usuario.

(8.11.1) clases para la entrada y la salida

Java se basa en las secuencias para dar facilidades de entrada y salida. Cada secuencia es una corriente de datos con un emisor y un receptor de datos en cada extremo. Todas las clases relacionadas con la entrada y salida de datos están en el paquete **java.io.**

Los datos fluyen en serie, byte a byte. Se habla entonces de un **stream** (corriente de datos, o mejor dicho, corriente de bytes). Hay otro stream que lanza caracteres (tipo ***char*** Unicode, de dos bytes), se habla entonces de un ***reader*** (si es de lectura) o un ***writer*** (escritura).

Los problemas de entrada / salida suelen causar excepciones de tipo **IOException** o de sus derivadas. Con lo que la mayoría de operaciones deben ir inmersas en un **try.**

InputStream/ OutputStream

Clases **abstractas** que definen las funciones básicas de lectura y escritura de una secuencia de bytes pura (sin estructurar). Esas son corrientes de bits, no representan ni textos ni objetos. Poseen numerosas subclases, de hecho casi todas las clases preparadas para la lectura y la escritura, derivan de estas.

Aquí se definen los métodos **read()** (leer) y **write()** (escribir). Ambos son métodos que trabajan con los datos, byte a byte.

Reader/Writer

Clases **abstractas** que definen las funciones básicas de escritura y lectura basada en caracteres Unicode. Se dice que estas clases pertenecen a la jerarquía de lectura/escritura orientada a caracteres, mientras que las anteriores pertenecen a la jerarquía orientada a bytes.

Aparecieron en la versión 1.1 y no substituyen a las anteriores. Siempre que se pueda es más recomendable usar clases que deriven de estas.

Posee métodos **read** y **write** adaptados para leer arrays de caracteres.

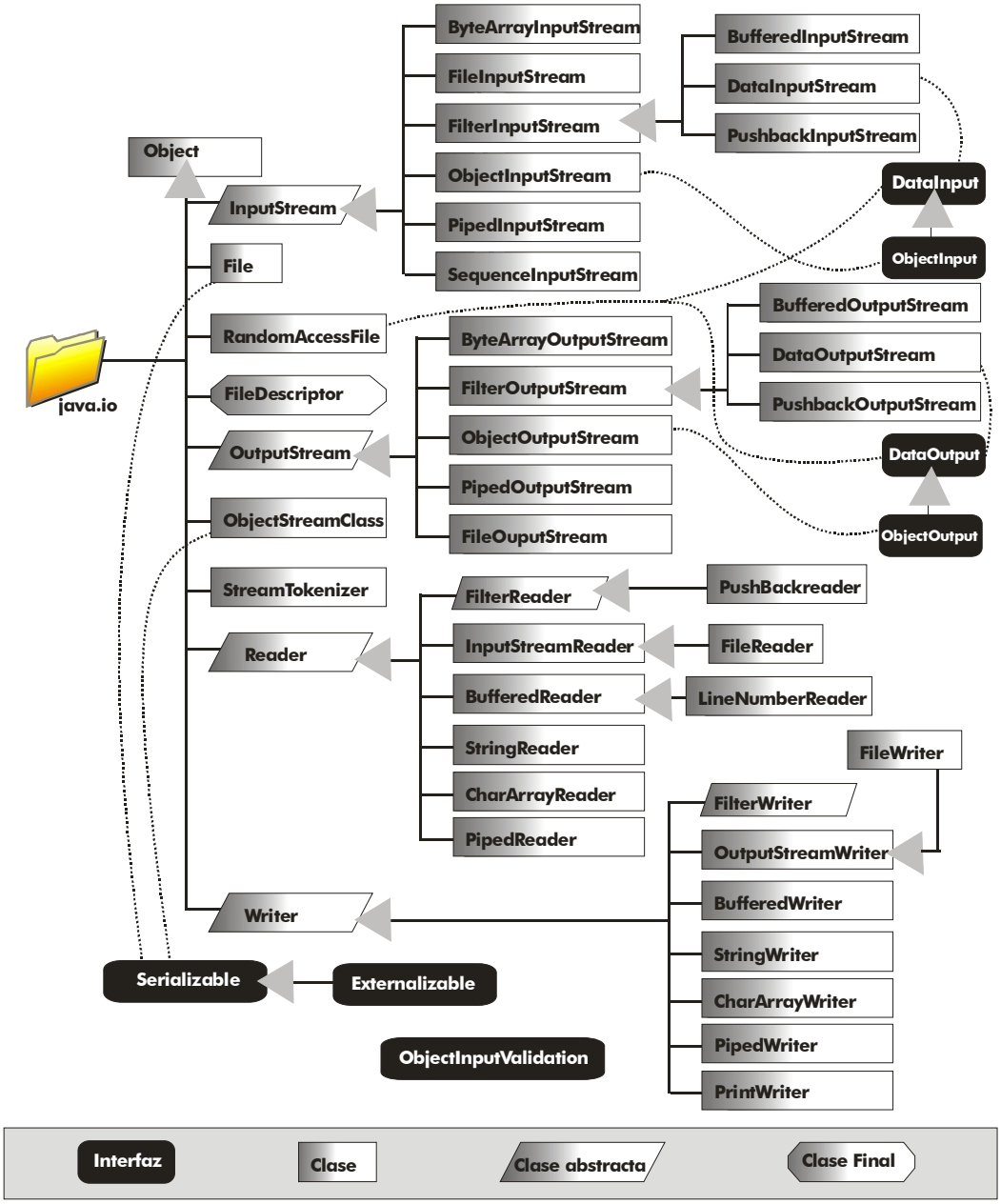


Ilustración 18, Clases e interfaces del paquete java.io

InputStreamReader/ OutputStreamWriter

Son clases que sirven para adaptar la entrada y la salida. El problema está en que las clases anteriores trabajan de forma muy distinta y ambas son necesarias. Por

ello InputStreamReader convierte una corriente de datos de tipo InputStream a forma de Reader.

DataInputStream/DataOutputStream

Leen corrientes de datos de entrada en forma de byte, pero adaptándola a los tipos simples de datos (**int, short, byte,..., String**). Tienen varios métodos read y write para leer y escribir datos de todo tipo. En el caso de DataInputStream son:

- **readBoolean().** Lee un valor booleano de la corriente de entrada. Puede provocar excepciones de tipo **IOException** o excepciones de tipo **EOFException**, esta última se produce cuando se ha alcanzado el final del archivo y es una excepción derivada de la anterior, por lo que si se capturan ambas, ésta debe ir en un **catch** anterior (de otro modo, el flujo del programa entraría siempre en la **IOException**).
- **readByte().** Idéntica a la anterior, pero obtiene un byte. Las excepciones que produce son las mismas
- **readChar, readShort, readInt, readLong, readFloat, readDouble**. Como las anteriores, pero leen los datos indicados.
- **readUTF()**. Lee un String en formato UTF (codificación norteamericana). Además de las excepciones comentadas antes, puede ocurrir una excepción del tipo **UTFDataFormatException (**derivada de **IOException**) si el formato del texto no está en UTF.

Por su parte, los métodos de DataOutputStream son:

- **writeBoolean, writeByte, writeDouble, writeFloat, writeShort, writeUTF, writeInt, writeLong.** Todos poseen un argumento que son los datos a escribir (cuyo tipo debe coincidir con la función).

ObjectInputStream/ObjectOutputStream

Filtros de secuencia que permiten leer y escribir objetos de una corriente de datos orientada a bytes. Sólo tiene sentido si los datos almacenados son objetos. Aporta un nuevo método de lectura:

- **readObject.** Devuelve un objeto Object de los datos de la entrada. En caso de que no haya un objeto o no sea serializable, da lugar a excepciones. Las excepciones pueden ser: **ClassNotFoundExcepcion, InvalidClassExcepcion, StreamCorruptedException, OptionalDataException** o **IOException** a secas.

La clase ObjectOutputStream posee el método de escritura de objetos **writeObject** al que se le pasa el objeto a escribir. Este método podría dar lugar en caso de fallo a excepciones **IOException, NotSerializableException** o **InvalidClassException.**.

BufferedInputStream/BufferedOutputStream/BufferedReader/BufferedWriter

La palabra **buffered** hace referencia a la capacidad de almacenamiento temporal en la lectura y escritura. Los datos se almacenan en una memoria temporal antes

de ser realmente leídos o escritos. Se trata de cuatro clase que trabajan con métodos distintos pero que suelen trabajar con las mismas corrientes de entrada que podrán ser de bytes (Input/OutputStream) o de caracteres (Reader/Writer).

La clase **BufferedReader** aporta el método **readLine** que permite leer caracteres hasta la presencia de **null** o del salto de línea.

PrintWriter

Secuencia pensada para impresión de texto. Es una clase escritora de caracteres en flujos de salida, que posee los métodos **print y println** ya comentados anteriormente, que otorgan gran potencia a la escritura.

FileInputStream/FileOutputStream/FileReader/FileWriter

Leen y escriben en archivos (File=Archivo).

PipedInputStream/PipedOutputStream

Permiten realizar canalizaciones entre la entrada y la salida; es decir lo que se lee se utiliza para una secuencia de escritura o al revés.

(8.11.2) entrada y salida estándar

las clases in y out

java.lang.System es una clase que poseen multitud de pequeñas clases relacionadas con la configuración del sistema. Entre ellas están la clase **in** que es un **InputStream** que representa la entrada estándar (normalmente el teclado) y **out** que es un **OutputStream** que representa a la salida estándar (normalmente la pantalla). Hay también una clase **err** que representa a la salida estándar para errores. El uso podría ser:

```
InputStream stdin =System.in;
OutputStream stdout=System.out;
```

El método **read()** permite leer un byte. Este método puede lanzar excepciones del tipo **IOException** por lo que debe ser capturada dicha excepción.

```
int valor=0;
try{
  valor=System.in.read();
}
catch(IOException e){
  ...
}
System.out.println(valor);
```

No tiene sentido el listado anterior, ya que **read()** lee un byte de la entrada estándar, y en esta entrada se suelen enviar caracteres, por lo que el método read no es el apropiado. El método **read** puede poseer un argumento que es un array de bytes que almacenará cada carácter leído y devolverá el número de caracteres leído

```
InputStream stdin=System.in;
int n=0;
byte[] caracter=new byte[1024];
try{
  n=System.in.read(caracter);
}
catch(IOException e){
  System.out.println("Error en la lectura");
}
for (int i=0;i<=n;i++)
  System.out.print((char)caracter[i]);
```

El lista anterior lee una serie de bytes y luego los escribe. La lectura almacena el código del carácter leído, por eso hay que hacer una conversión a **char.**

Para saber que tamaño dar al array de bytes, se puede usar el método **available()** de la clase **InputStream** la tercera línea del código anterior sería:

```
byte[] carácter=new byte[System.in.available];
```

Conversión a forma de Reader

El hecho de que las clases **InputStream** y **OutputStream** usen el tipo byte para la lectura, complica mucho su uso. Desde que se impuso Unicode y con él las clases **Reader** y **Writer**, hubo que resolver el problema de tener que usar las dos anteriores.

La solución fueron dos clases: **InputStreamReader** y **OutputStreamWriter.** Se utilizan para convertir secuencias de byte en secuencias de caracteres según

una determinada configuración regional. Permiten construir objetos de este tipo a partir de objetos **InputStream** u **OutputStream.** Puesto que son clases derivadas de **Reader** y **Writer** el problema está solucionado.

El constructor de la clase **InputStreamReader** requiere un objeto **InputStream** y, opcionalmente, una cadena que indique el código que se utilizará para mostrar caracteres (por ejemplo "ISO-8914-1" es el código Latín 1, el utilizado en la configuración regional). Sin usar este segundo parámetro se construye según la codificación actual (es lo normal).

Lo que hemos creado de esa forma es un objeto ***convertidor.*** De esa forma podemos utilizar la función read orientada a caracteres Unicode que permite leer caracteres extendidos. Está función posee una versión que acepta arrays de caracteres, con lo que la versión ***writer*** del código anterior sería:

```
InputStreamReader stdin=new InputStreamReader(System.in);
char caracter[]=new char[1024];
int numero=-1;
try{
     numero=stdin.read(caracter);
}
catch(IOException e){
     System.out.println("Error en la lectura");
}
for(int i=0;i<numero;i++)
     System.out.print(caracter[i]);
```

Lectura con readLine

El uso del método ***read*** con un array de caracteres sigue siendo un poco enrevesado. Por ello para leer cadenas de caracteres se suele utilizar la clase **BufferedReader**. La razón es que esta clase posee el método **ReadLine()** que permite leer una línea de texto en forma de String, que es más fácil de manipular. Esta clase usa un constructor que acepta objetos **Reader** (y por lo tanto **InputStreamReader**¸ ya que desciende de ésta) y, opcionalmente, el número de caracteres a leer.

Hay que tener en cuenta que el método ***ReadLine*** (como todos los métodos de lectura) puede provocar excepciones de tipo ***IOException*** por lo que, como ocurría con las otras lecturas, habrá que capturar dicha lectura.

```
String texto="";
try{
  //Obtención del objeto Reader
  InputStreamReader conv=new
InputStreamReader(System.in);
  //Obtención del BufferedReader
  BufferedReader entrada=new BufferedReader(conv);
  texto=entrada.readLine();
}
catch(IOException e){
  System.out.println("Error");
}
System.out.println(texto);
```

(8.12) Ficheros

Una aplicación Java puede escribir en un archivo, salvo que se haya restringido su acceso al disco mediante políticas de seguridad. La dificultad de este tipo de operaciones está en que los sistemas de ficheros son distintos en cada sistema y aunque Java intentar aislar la configuración específica de un sistema, no consigue evitarlo del todo.

(8.12.1) clase File

En el paquete **java.io** se encuentra la clase **File** pensada para poder realizar operaciones de información sobre archivos. No proporciona métodos de acceso a los archivos, sino operaciones a nivel de sistema de archivos (listado de archivos, crear carpetas, borrar ficheros, cambiar nombre,...).

construcción de objetos de archivo

Utiliza como único argumento una cadena que representa una ruta en el sistema de archivo. También puede recibir, opcionalmente, un segundo parámetro con una ruta segunda que se define a partir de la posición de la primera.

```
File archivo1=new File("/datos/bd.txt");
File carpeta=new File("datos");
```

El primer formato utiliza una ruta absoluta y el segundo una ruta relativa. La ruta absoluta se realiza desde la raíz de la unidad de disco en la que se está trabajando y la relativa cuenta desde la carpeta actual de trabajo.

Otra posibilidad de construcción es utilizar como primer parámetro un objeto File ya hecho. A esto se añade un segundo parámetro que es una ruta que cuenta desde la posición actual.

```
File carpeta1=new File("c:/datos");//ó c\\datos
File archivo1=new File(carpeta1,"bd.txt");
```

Si el archivo o carpeta que se intenta examinar no existe, la clase ***File*** no devuelve una excepción. Habrá que utilizar el método **exists**. Este método recibe **true** si la carpeta o archivo es válido (puede provocar excepciones ***SecurityException***).

También se puede construir un objeto **File** a partir de un objeto **URL**.

el problema de las rutas

Cuando se crean programas en Java hay que tener muy presente que no siempre sabremos qué sistema operativo utilizará el usuario del programa. Esto provoca que la realización de rutas sea problemática porque la forma de denominar y recorrer rutas es distinta en cada sistema operativo.

Por ejemplo en Windows se puede utilizar la barra **/** o la doble barra invertida **** como separador de carpetas, en muchos sistemas Unix sólo es posible la primera opción. En general es mejor usar las clases **Swing** (como **JFileDialog**) para especificar rutas, ya que son clases en las que la ruta de elige desde un cuadro y, sobre todo, son independientes de la plataforma.

También se pueden utilizar las **variables estáticas** que posee File. Estas son:

propiedad	uso
char **separatorChar**	El carácter separador de nombres de archivo y carpetas. En Linux/Unix es "/" y en Windows es "\", que se debe escribir como \\, ya que el carácter \ permite colocar caracteres de control, de ahí que haya que usar la doble barra.
String **separador**	Como el anterior pero en forma de String
char **pathSeparatorChar**	El carácter separador de rutas de archivo que permite poner más de un archivo en una ruta. En Linux/Unix suele ser ":", en Windows es ";"
String **pathSeparator**	Como el anterior, pero en forma de String

Para poder garantizar que el separador usado es el del sistema en uso:

```
String ruta="documentos/manuales/2003/java.doc";
ruta=ruta.replace('/',File.separatorChar);
```

Normalmente no es necesaria esta comprobación ya que Windows acepta también el carácter / como separador.

métodos generales

método	uso
String **toString()**	Para obtener la cadena descriptiva del objeto
boolean **exists()**	Devuelve **true** si existe la carpeta o archivo.
boolean **canRead()**	Devuelve **true** si el archivo se puede leer
boolean **canWrite()**	Devuelve **true** si el archivo se puede escribir
boolean **isHidden()**	Devuelve **true** si el objeto File es oculto
boolean **isAbsolute()**	Devuelve **true** si la ruta indicada en el objeto File es absoluta
boolean **equals(File** f2**)**	Compara f2 con el objeto **File** y devuelve verdadero si son iguales.
String **getAbsolutePath()**	Devuelve una cadena con la ruta absoluta al objeto File.
File **getAbsoluteFile()**	Como la anterior pero el resultado es un objeto File
String **getName()**	Devuelve el nombre del objeto File.
String **getParent()**	Devuelve el nombre de su carpeta superior si la hay y si no **null**
File **getParentFile()**	Como la anterior pero la respuesta se obtiene en forma de objeto File.
boolean **setReadOnly()**	Activa el atributo de sólo lectura en la carpeta o archivo.
URL **toURL()** throws MalformedURLException	Convierte el archivo a su notación URL correspondiente
URI **toURI()**	Convierte el archivo a su notación URI correspondiente

métodos de carpetas

método	uso
boolean **isDirectory()**	Devuelve **true** si el objeto File es una carpeta y **false** si es un archivo o si no existe.
boolean **mkdir()**	Intenta crear una carpeta y devuelve **true** si fue posible hacerlo
boolean **mkdirs()**	Usa el objeto para crear una carpeta con la ruta creada para el objeto y si hace falta crea toda la estructura de carpetas necesaria para crearla.
boolean **delete()**	Borra la carpeta y devuelve **true** si puedo hacerlo
String[] **list()**	Devuelve la lista de archivos de la carpeta representada en el objeto File.
static File[] **listRoots()**	Devuelve un array de objetos File, donde cada objeto del array representa la carpeta raíz de una unidad de disco.
File[] **listfiles()**	Igual que la anterior, pero el resultado es un array de objetos File.

métodos de archivos

método	uso
boolean isFile()	Devuelve **true** si el objeto File es un archivo y **false** si es carpeta o si no existe.
boolean renameTo(File f2**)**	Cambia el nombre del archivo por el que posee el archivo pasado como argumento. Devuelve **true** si se pudo completar la operación.
boolean delete()	Borra el archivo y devuelve **true** si puedo hacerlo
long length()	Devuelve el tamaño del archivo en bytes
boolean createNewFile() **Throws** IOException	Crea un nuevo archivo basado en la ruta dada al objeto File. Hay que capturar la excepción ***IOException*** que ocurriría si hubo error crítico al crear el archivo. Devuelve **true** si se hizo la creación del archivo vacío y **false** si ya había otro archivo con ese nombre.
static File createTempFile(String prefijo, **String** sufijo**)**	Crea un objeto File de tipo archivo temporal con el prefijo y sufijo indicados. Se creará en la carpeta de archivos temporales por defecto del sistema. El prefijo y el sufijo deben de tener al menos tres caracteres (el sufijo suele ser la extensión), de otro modo se produce una excepción del tipo **IllegalArgumentsException** Requiere capturar la excepción **IOException** que se produce ante cualquier fallo en la creación del archivo
static File createTempFile(String prefijo, **String** sufijo, **File** directorio**)**	Igual que el anterior, pero utiliza el directorio indicado.
void deleteOnExit()	Borra el archivo cuando finaliza la ejecución del programa

(8.12.2) secuencias de archivo

lectura y escritura byte a byte

Para leer y escribir datos a archivos, Java utiliza dos clases especializadas que leen y escriben orientando a byte (Véase tema anterior); son **FileInputStream** (para la lectura) y **FileOutputStream** (para la escritura).

Se crean objetos de este tipo construyendo con un parámetro que puede ser una ruta o un objeto File:

```
FileInputStream fis=new FileInputStream(objetoFile);
FileInputStream fos=new
                FileInputStream("/textos/texto25.txt");
```

La construcción de objetos **FileOutputStream** se hace igual, pero además se puede indicar un segundo parámetro booleano que con valor **true** permite añadir más datos al archivo (normalmente al escribir se borra el contenido del archivo, valor **false**).

Estos constructores intentan abrir el archivo, generando una excepción del tipo **FileNotFoundException** si el archivo no existiera u ocurriera un error en la apertura. Los métodos de lectura y escritura de estas clases son los heredados de las clases InputStream y OutputStream. Los métodos **read** y **write** son los que permiten leer y escribir. El método **read** devuelve -1 en caso de llegar al final del archivo.

Otra posibilidad, más interesante, es utilizar las clases DataInputStream y DataOutputStream. Estas clases está mucho más preparadas para escribir datos de todo tipo.

escritura

El proceso sería:

(1) Crear un objeto **FileOutputStream** a partir de un objeto **File** que posee la ruta al archivo que se desea escribir.

(2) Crear un objeto **DataOutputStream** asociado al objeto anterior. Esto se realiza en la construcción de este objeto.

(3) Usar el objeto del punto 2 para escribir los datos mediante los métodos **writeTipo** donde ***tipo*** es el tipo de datos a escribir (Int, Double, ...). A este método se le pasa como único argumento los datos a escribir.

(4) Se cierra el archivo mediante el método **close** del objeto DataOutputStream.

Ejemplo:

```
File f=new File("D:/prueba.out");
Random r=new Random();
double d=18.76353;
try{
    FileOutputStream fis=new FileOutputStream(f);
    DataOutputStream dos=new DataOutputStream(fis);
    for (int i=0;i<234;i++){ //Se repite 233 veces
        dos.writeDouble(r.nextDouble());//Nº aleatorio
    }
    dos.close();
}
catch(FileNotFoundException e){
    System.out.println("No se encontro el archivo");
}
catch(IOException e){
    System.out.println("Error al escribir");
}
```

lectura

El proceso es análogo. Sólo que hay que tener en cuenta que al leer se puede alcanzar el final del archivo. Al llegar al final del archivo, se produce una excepción del tipo **EOFException** (que es subclase de **IOException**), por lo que habrá que controlarla.

Ejemplo, leer los números del ejemplo anterior :

```
boolean finArchivo=false;//Para bucle infinito
try{
    FileInputStream fis=new FileInputStream(f);
    DataInputStream dis=new DataInputStream(fis);
    while (!finArchivo){
        d=dis.readDouble();
        System.out.println(d);
    }

    dis.close();
}
catch(EOFException e){
    finArchivo=true;
}
```

```
catch(FileNotFoundException e){
    System.out.println("No se encontro el archivo");
}
catch(IOException e){
    System.out.println("Error al leer");
}
```

En este listado, obsérvese como el bucle **while** que da lugar a la lectura se ejecuta indefinidamente (no se pone como condición a secas **true** porque casi ningún compilador lo acepta), se saldrá de ese bucle cuando ocurra la excepción EOFException que indicará el fin de archivo.

Las clases DataStream son muy adecuadas para colocar datos binarios en los archivos.

lectura y escritura mediante caracteres

Como ocurría con la entrada estándar, se puede convertir un objeto **FileInputStream** o **FileOutputStream** a forma de **Reader** o **Writer** mediante las clases **InputStreamReader** y **OutputStreamWriter**.

Existen además dos clases que manejan caracteres en lugar de bytes (lo que hace más cómodo su manejo), son **FileWriter** y **FileReader.**

La construcción de objetos del tipo FileReader se hace con un parámetro que puede ser un objeto File o un String que representarán a un determinado archivo.

La construcción de objetos FileWriter se hace igual sólo que se puede añadir un segundo parámetro booleano que, en caso de valer **true**, indica que se abre el archivo para añadir datos; en caso contrario se abriría para grabar desde cero (se borraría su contenido).

Para escribir se utiliza **write** que es un método void que recibe como parámetro lo que se desea escribir en formato int, String o array de caracteres. Para leer se utiliza el método read que devuelve un int y que puede recibir un array de caracteres en el que se almacenaría lo que se desea leer. Ambos métodos pueden provocar excepciones de tipo IOException.

Ejemplo:

```
File f=new File("D:/archivo.txt");
int x=34;
try{
    FileWriter fw=new FileWriter(f);
    fw.write(x);
    fw.close();

}
```

```
catch(IOException e){
    System.out.println("error");
    return;
}
//Lectura de los datos
try{
    FileReader fr=new FileReader(f);
    x=fr.read();
    fr.close();
}
catch(FileNotFoundException e){
    System.out.println("Error al abrir el archivo");
}
catch(IOException e){
    System.out.println("Error al leer");
}
System.out.println(x);
```

En el ejemplo anterior, primero se utiliza un ***FileWrite*** llamado ***fw*** que escribe un valor entero (aunque realmente sólo se escribe el valor carácter, es decir sólo valdrían valores hasta 32767). La función ***close*** se encarga de cerrar el archivo tras haber leído. La lectura se realiza de forma análoga.

Otra forma de escribir datos (imprescindible en el caso de escribir texto) es utilizar las clases **BufferedReader** y **BufferedWriter** vistas en el tema anterior. Su uso sería:

```
File f=new File("D:/texto.txt);
int x=105;
try{
    FileReader fr=new FileReader(f);
    BufferedReader br=new BufferedReader(fr);
    String s;
    do{
       s=br.readLine();
            System.out.println(s);
    }while(s!=null);
}
catch(FileNotFoundException e){
    System.out.println("Error al abrir el archivo");
}
catch(IOException e){
    System.out.println("Error al leer");}
```

En este caso el listado permite leer un archivo de texto llamado **texto.txt**. El fin de archivo con la clase BufferedReader se detecta comparando con **null**, ya que en caso de que lo leído sea null, significará que hemos alcanzado el final del archivo. La gracia de usar esta clase está en el método **readLine** que agiliza enormemente la lectura.

(8.12.3) RandomAccessFile

Esta clase permite leer archivos en forma aleatoria. Es decir, se permite leer cualquier posición del archivo en cualquier momento. Los archivos anteriores son llamados secuenciales, se leen desde el primer byte hasta el último.

Esta es una clase primitiva que implementa los interfaces **DataInput** y **DataOutput** y sirve para leer y escribir datos.

La construcción requiere de una cadena que contenga una ruta válida a un archivo o de un archivo File. Hay un segundo parámetro obligatorio que se llama **modo**. El modo es una cadena que puede contener una **r** (lectura), **w** (escritura) o ambas, **rw**.

Como ocurría en las clases anteriores, hay que capturar la excepción **FileNotFound** cuando se ejecuta el constructor.

```
File f=new File("D:/prueba.out");
RandomAccessFile archivo = new RandomAccessFile( f,
"rw");
```

Los métodos fundamentales son:

- **void seek(long pos).** Permite colocarse en una posición concreta, contada en bytes, en el archivo. Lo que se coloca es el puntero de acceso que es la señal que marca la posición a leer o escribir.
- **long getFilePointer().** Posición actual del puntero de acceso
- **long length().** Devuelve el tamaño del archivo
- **readBoolean, readByte, readChar, readInt, readDouble, readFloat, readUTF, readLine.** Funciones de lectura. Leen un dato del tipo indicado. En el caso de ***readUTF*** lee una cadena en formato Unicode.
- **writeBoolean, writeByte, writeBytes, writeChar, writeChars writeInt, writeDouble, writeFloat, writeUTF, writeLine.** Funciones de escritura. Todas reciben como parámetro, el dato a escribir. Escribe encima de lo ya escrito. Para escribir al final hay que colocar el puntero de acceso al final del archivo.

(8.12.4) el administrador de seguridad

Llamado **Security manager**, es el encargado de prohibir que subprogramas y aplicaciones escriban en cualquier lugar del sistema. Por eso numerosas acciones podrían dar lugar a excepciones del tipo **SecurityException** cuando no se permite escribir o leer en un determinado sitio.

(8.12.5) serialización

Es una forma automática de guardar y cargar el estado de un objeto. Se basa en la interfaz **serializable** que es la que permite esta operación. Si un objeto ejecuta esta interfaz puede ser guardado y restaurado mediante una secuencia.

Cuando se desea utilizar un objeto para ser almacenado con esta técnica, debe ser incluida la instrucción **implements Serializable** (además de importar la clase **java.io.Serializable**) en la cabecera de clase. Esta interfaz no posee métodos, pero es un requisito obligatorio para hacer que el objeto sea serializable.

La clase **ObjectInputStream** y la clase **ObjectOutputStream** se encargan de realizar este procesos. Son las encargadas de escribir o leer el objeto de un archivo. Son herederas de **InputStream** y **OutputStream**, de hecho son casi iguales a **DataInput/OutputStream** sólo que incorporan los métodos **readObject** y **writeObject** que son muy poderosos. Ejemplo:

```
try{
  FileInputStream fos=new
            FileInputStream("d:/nuevo.out");
  ObjectInputStream os=new ObjectInputStream(fos);
  Coche c;
  boolean finalArchivo=false;

  while(!finalArchivo){
       c=(Coche) readObject();
        System.out.println(c);
   }
}
catch(EOFException e){
  System.out.println("Se alcanzó el final");
}
catch(ClassNotFoundException e){
  System.out.println("Error el tipo de objeto no es compatible");
}
catch(FileNotFoundException e){
  System.out.println("No se encontró el archivo");
}
catch(IOException e){
  System.out.println("Error  "+e.getMessage());
  e.printStackTrace();
}
```

El listado anterior podría ser el código de lectura de un archivo que guarda coches. Los métodos **readObject** y **writeObject** usan objetos de tipo Object,

readObject les devuelve y writeObject les recibe como parámetro. Ambos métodos lanzan excepciones del tipo **IOException** y **readObject** además lanza excepciones del tipo **ClassNotFoundException**.

Obsérvese en el ejemplo como la excepción **EOFException** ocurre cuando se alcanzó el final del archivo.