Informática

para físicos e ingenieros electrónicos

José M. Gómez

Agustí Gutierrez

Manuel López

Xavier Luri

Correo electrónico: jm.gomez@ub.edu

Índice

1 El Ordenador

1.1 Introducción

Informática es una palabra de origen francés formada por la contracción de los vocablos INFORmación y autoMÁTICA. La Real Academia Española define la informática como *el conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores.*

Siguiendo con las definiciones, podemos decir que Computador, computadora u ordenador *es una máquina capaz de aceptar unos datos de entrada, efectuar con ellos operaciones lógicas y aritméticas, y proporcionar la información resultante a través de un medio de salida: todo ello sin la intervención de un operador humano y bajo el control de una serie de instrucciones, llamado programa, previamente almacenado en el ordenador*.

El primer ordenador fue diseñado en el siglo XIX por Charles Babbage con el propósito de automatizar los cálculos de tablas de logaritmos, aunque nunca llegó a funcionar en la práctica. Sin embargo, la primera computadora electrónica de propósito general fue ENIAC (*Electronic Numerical Integrator And Computer*), presentada en 1946. Fue desarrollada y construida por el ejército Americano para su laboratorio de investigación balística. Tenía como objetivo calcular tablas balísticas de proyectiles. Fue concebida y diseñada por J. Presper Eckert y John William Mauchly de la universidad de Pensilvania. Ocupaba una superficie de 167 m² y operaba con un total de 17.468 válvulas electrónicas o tubos de vacío. Físicamente, la ENIAC tenía 17.468 tubos de vacío, 7.200 diodos de cristal, 1.500 relés, 70.000 resistencias, 10.000 condensadores y 5 millones de soldaduras. Pesaba 27 Tm, medía 2,4 m x 0,9 m x 30 m; utilizaba 1.500 conmutadores electromagnéticos y relés; requería la operación manual de unos 6.000 interruptores, y su programa o software, cuando requería modificaciones, tardaba semanas de instalación manual. Su potencia de cálculo le permitía realizar unas 5000 sumas o restas por segundo.

ENIAC fue programada fundamentalmente por Kay McNulty, Betty Jennings, Betty Snyder, Marlyn Wescoff, Fran Bilas y Ruth Lichterman. Todas ellas fueron las encargadas de realizar las primeras librerías de rutinas, y de elaborar los primeros programas para ENIAC. Podríamos decir que fueron las primeras informáticas.

Figura 1. Ordenador ENIAC

Una de las características de ENIAC era su gran consumo, unos 160kW. A raíz de ello corría el mito de que la ciudad de Filadelfia, donde se encontraba instalada, sufría apagones cuando la ENIAC entraba en funcionamiento. Este consumo elevaba la temperatura del local a 50ºC.

Para efectuar las diferentes operaciones era preciso cambiar, conectar y reconectar los cables como se hacía, en esa época, en las centrales telefónicas. Este trabajo podía demorar varios días dependiendo del cálculo a realizar. Sin olvidar que la vida media entre fallos era de una hora.

ENIAC fue la pionera de las actuales computadoras, y permitió abrir un campo que todavía está evolucionando, aunque muchos de los aspectos originales de ENIAC se mantienen.

Un aspecto muy importante de los ordenadores, desde ENIAC hasta los actuales, es que utilizan código binario para almacenar y procesar la información. Los número binarios como su nombre indica, están representados en base a dos valores, ‘0’ y ‘1’, y la unidad mínima de información es el bit (BInary digIT). El motivo se debe a que establecer los estados totalmente abierto o totalmente cerrado es más sencillo que definir estados intermedios.

La situación vendría a ser semejante a cuando un niño cuenta con los dedos. Un dedo levantado indica un 1, dos dedos levantado indica un 2 y así sucesivamente (Figura 2):

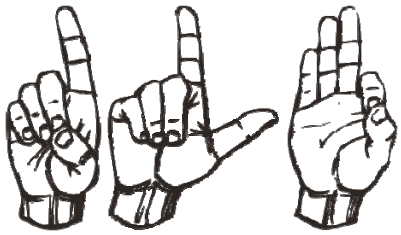


Figura 2. Signos de las cifras 1, 2 y 3 con los dedos.

Ahora bien, cuando intentamos mostrar un cuatro, si lo hacemos escondiendo el pulgar, no hay problema, pero si lo hacemos con el meñique, es muy probable que se nos quede a medias el anular, y en este caso, que es un tres o un cuatro. Para que el dedo anular se pueda extender completamente, necesitamos ayudar al meñique con el pulgar, lo que nos vuelve a mostrar un tres (Figura 3).

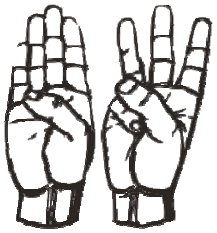


Figura 3. Signo de la cifra 4 y de 3 con los dedos.

De la misma forma, cuando trabajamos con un ordenador es mejor trabajar con encendido/apagado (extendido/doblado). Si trabajamos con estados intermedios, nos puede pasar como cuando se intenta doblar meñique para mostrar un cuatro, ya que el anular quedará en un estado que no sabremos que número se esta mostrando. En el caso de un ordenador, los dedos serían transistores, y extendido/doblado pasaría a ser abierto/cerrado (una posibilidad sería totalmente abierto es un '1' y totalmente cerrado un '0').

Por este motivo, cualquier información que queramos introducir en el ordenador ha de ser traducida con anterioridad a código binario (digitalizada) para que este pueda ser guardada y/o procesada. En los tiempos del ENIAC esto era relativamente sencillo, ya que se utilizaban interruptores cuyo estado representaba directamente un valor de un dígito binario. Esto tenía un pequeño inconveniente, las programadoras no podían cometer errores, ya que identificar en un panel de interruptores cuál de ellos estaba mal podía ser una tarea muy tediosa. Hoy en día, la situación ha mejorado notablemente, y podemos llegar incluso a hablarle al ordenador para que realice operaciones básicas.

Como el lenguaje natural de los ordenadores es el código binario es importante familiarizarse con algunos términos propios de este código. Como hemos dicho, un *bit* es un dígito en código binario y se representa como unidad con una "b" minúscula. Un *byte* son 8 bits y se representa con una "B" mayúscula. Para referirse a un mayor número de bits se suelen utilizar los prefijos habituales definidos por el Sistema Internacional de Medidas (Kilo, Mega, Giga, Tera, etc.) aunque debe tenerse en cuenta que, debido a motivos históricos[1], hay una cierta confusión en el uso de los mismos y coexisten (al menos) dos interpretaciones de estos prefijos, como se recoge en la Tabla 1.

Tabla 1. Prefijos establecidos por el SI

Prefijo	Símbolo	Factor decimal (SI)	Factor binario
Kilo-	k-	10^3	1024^1
Mega-	M-	10^6	1024^2
Giga-	G-	10^9	1024^3
Tera-	T-	10^{12}	1024^4
Peta-	P-	10^{15}	1024^5
Exa-	E-	10^{18}	1024^6
Zetta-	Z-	10^{21}	1024^7
Yotta-	Y-	10^{24}	1024^8

[1] Véase http://physics.nist.gov/cuu/Units/binary.html para una discusión en detalle del problema

Es importante remarcar la diferencia entre el factor decimal y el binario, ya que éste último es siempre superior al primero. Cuando trabajamos con un ordenador, normalmente se trabaja con factores binarios. Sin embargo, al comprar ciertos productos (como en el caso de los discos duros) es posible que nos indiquen la capacidad en factor decimal, y por lo tanto el ordenador indique un tamaño inferior al que aparece en la caja del producto.

1.2 Componentes de un ordenador

La estructura básica de los ordenadores actuales se basa en la arquitectura propuesta por John von Neumann en 1945. Esta arquitectura tiene tres componentes básicos que interactúan entre ellos. Estos componentes son una unidad de procesamiento, una memoria donde se almacenan tanto instrucciones como datos y los dispositivos de entrada salida. Si nos fijamos en una calculadora, los datos serían los números que introducimos, y las instrucciones, las operaciones a realizar con dichos datos.

La Figura 4 muestra un diagrama un poco más detallado de la estructura de los ordenadores actuales. Estos esta formados por los siguientes elementos o unidades funcionales: unidades de entrada, unidades de salida, memoria externa, memoria interna, unidad aritmético-lógica y unidad de control.

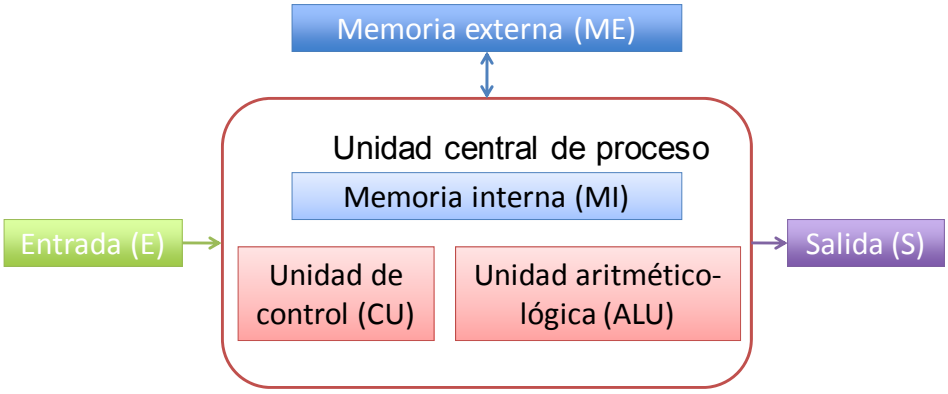


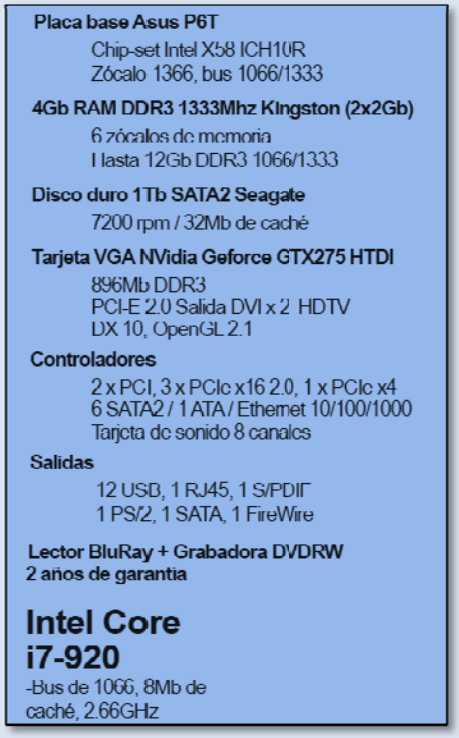
Figura 4. Estructura funcional de un ordenador

Cada uno de estos elementos acostumbra a equivaler a un chip físico o un dispositivo. Así, la unidad central de proceso, es el chip central de un ordenador, mientras que la entrada, salida y memoria externa suelen ser dispositivos que se pueden añadir o quitar a voluntad.

- **Unidad central de proceso (CPU, Central Processing Unit):** Es el centro neurálgico del ordenador, y gestiona todos los elementos del mismo. Se compone de tres elementos básicos:
  - **Unidad de control (CU, *Control Unit*):** Forma parte de la CPU y se encarga de interpretar las instrucciones de los programas y generar las señales de control dirigidas a todas las unidades del ordenador con el fin de ejecutar la instrucción. La unidad de control también recibe señales de los demás dispositivos para comunicarle su estado (disponibles, ocupados, etc.). Estas señales se conocen como señales de estado. Es la responsable de que el ordenador funcione.
  - **Unidad aritmético-lógica (ALU, Arithmetic-Logic Unit):** Realiza operaciones aritméticas (sumas, restas, etc.) y operaciones lógicas (comparación, AND, OR, XOR). Este modulo recibe los datos para ser procesados.
  - **Memoria interna (MI):** Inicialmente guarda la información básica para poder realizar las operaciones, en lo que se denominan registros. Serían equivalentes a las memorias de una calculadora. Con el tiempo, y al aumentar la capacidad de integración, ha aparecido la memoria Cache, que puede llegar a almacenar megabytes de información para acelerar el trabajo de la CPU. En ambos casos, la memoria es volátil, también conocida como memoria de acceso aleatorio (RAM, Random Access Memory), y por lo tanto al apagar el ordenador, se pierde la información. Utilizan un tipo de memoria que se denomina estática (SRAM, Static RAM), y que requiere 6 transistores para guardar cada unidad de información.

- **Memoria externa (ME):** La memoria interna es muy rápida (decenas de millones de palabras transmitidas por segundo), pero no tiene gran capacidad para almacenar datos. Para guardar una mayor cantidad de información se utilizan dos tipos de memorias más:
  - **Memoria principal:** Es la memoria donde se guarda la información que utiliza el ordenador durante la ejecución de procesos y en la actualidad suele tener tamaños entorno al gigabyte. Su principal característica es que utiliza memoria dinámica (DRAM, Dynamic RAM), la cual tiene la ventaja de ocupar menos espacio que su homóloga estática, ya que solo precisa de un transistor por cada bit, pero a cambio, requiere irla refrescando cada cierto tiempo para que no pierda los datos.
  - **Dispositivos de almacenamiento:** Son memorias basadas en otras tecnologías como discos magnéticos, discos ópticos, cintas magnéticas, memoria flash… que aunque son más lentos permiten almacenar más información (del orden de un millón de veces más lentos pero con cien o mil veces más capacidad).
- **Unidad de entrada (E):** Es un dispositivo por el que se introducen en el ordenador los datos e instrucciones. En estas unidades se transforman la información de entrada en señales binarias de naturaleza eléctrica. Un mismo ordenador puede tener varias unidades de entrada: teclado, ratón, escáner de imágenes, lector de tarjetas.
- **Unidad de salida (S):** Permite obtener los resultados de los programas ejecutados en el ordenador. La mayor parte de estas unidades transforman las señales binarias en señales perceptibles por el usuario. Los dispositivos de salida incluyen pantallas, impresoras y altavoces.

Ejercicio 1: En el cuadro inferior podéis ver las características de un ordenador de sobremesa. Decir a qué tipo de unidad funcional corresponde cada una de estas características.



Hoy en día, las CPUs suelen incluir varios núcleos, lo que significa que dentro de un mismo chip hay varias CPUs interconectadas, aunque, desde el punto de vista externo, es como si solo hubiera una. Esto permite ejecutar varios programas en paralelo, lo cual incrementa las prestaciones del ordenador.

1.3 Sistemas operativos

El sistema operativo es el programa principal del ordenador y tiene por objetivo facilitar su uso y conseguir que éste se utilice eficientemente. Es un programa de control que se encarga de gestionar, asignar y acceder a los recursos hardware que requieren los demás programas. Estos recursos hardware son: el procesador, la memoria principal, los discos, y los periféricos. Si varios usuarios están utilizando el mismo ordenador al mismo tiempo, el sistema operativo se encarga de asignar recursos, evitar los conflictos que pueden surgir cuando dos programas quieren acceder a los mismos recursos simultáneamente (la unidad de disco o la impresora por ejemplo). Decimos que el sistema operativo hace transparente al usuario las características del hardware sobre el cual trabaja.

El sistema operativo también permite que el ordenador se utilice eficientemente. Así por ejemplo, un sistema operativo que permita multiprogramación se encarga de asignar los tiempos de ejecución de dos programas que están activos de manera que durante los tiempos de espera de uno, el otro programa se esté ejecutando. E incluso, si corremos sobre una plataforma multiprocesador (con varias CPUs y/o varios núcleos) puede asignar cada programa a uno de los procesadores.

Las principales funciones, aunque no las únicas, que realiza el sistema operativo son la gestión de los siguientes recursos del computador:

- Procesador
- Memoria principal
- Dispositivos
- Sistema de archivos

A continuación pasamos a describirlas.

1.3.1 Gestión del procesador

La gestión y administración del procesador consiste en determinar cuál es la utilización que se va a hacer de él en cada momento. Esto dependerá de que programas tengan que ejecutarse, de la disponibilidad de datos para ejecutar y de las peticiones prioritarias de ejecución (interrupciones) que reciba el sistema. La gestión del procesador por parte del sistema operativo se centra en el concepto de proceso. Un proceso es *un programa en el que se ha iniciado su ejecución, y que es esta formado no solo por el código (instrucciones) sino también por su estado de ejecución (valores de registro de CPU) y su memoria de trabajo.* Un programa por sí sólo es un ente pasivo mientras que un proceso es un ente activo. Por lo tanto, el sistema operativo se encarga de planificar el tiempo del procesador determinando que proceso se ejecuta en cada instante.

Los primeros sistemas operativos sólo permitían un proceso activo a la vez, de manera que hasta que el proceso no acababa de ejecutarse completamente no se empezaba el siguiente. Esto se conoce como monoprogramación. El orden de la ejecución de los procesos se determinaba por el orden de llegada de las peticiones de ejecución y por la prioridad de los procesos. Procesos de más prioridad se ejecutaban antes. La monoprogramación tiene el inconveniente de desperdiciar tiempo de procesador siempre que los procesos están esperando que las operaciones de Entrada/Salida (lectura/escritura en memoria, disco) se ejecuten. Este problema se subsanó con la multiprogramación.

Los sistemas operativos más recientes permiten tener varios procesos activos a la vez (multiprogramación) de manera que la ejecución de los procesos se va intercalando en el procesador sin necesidad de que acabe completamente un proceso antes de dar paso al siguiente. Esto permite que durante los tiempos de espera de un proceso se cambie de proceso en ejecución y no se desperdicie tiempo de procesador. La multiprogramación permite dar la sensación al usuario de que varios programas se están ejecutando simultáneamente aunque el ordenador solo tenga un procesador, con un único núcleo, y por lo tanto sólo se puede estar ejecutando un proceso a la vez.

1.3.2 Gestión de la memoria

La memoria principal del ordenador se utiliza para guardar las instrucciones y datos de los programas que se han de ejecutar. El contenido de esta memoria varia a medida que los programas que hay en ejecución acaban, liberando el espacio de memoria que ocupan, y nuevos programas se cargan en memoria para ser ejecutados. La gestión de la memoria consiste en hacer un seguimiento de la ocupación de las diferentes posiciones y el nada trivial problema de determinar en qué posiciones de memoria se cargan los programas nuevos sin reducir la eficiencia de los accesos a memoria.

Con este fin, se suelen utilizar diferentes estrategias para la asignación de memoria principal: particiones estáticas, particiones dinámicas, segmentación. En las particiones estáticas se divide la memoria en particiones de tamaños variables y se asigna un proceso a cada una de ellas. Las particiones no se cambian y por eso se llama segmentación estática. Como es muy difícil que los programas tengan exactamente la misma longitud que alguna de las particiones de memoria y se les asigna una partición de tamaño mayor al programa, la parte vacía de la partición queda inutilizada. Este es el principal inconveniente de este método. La partición dinámica evita este problema asignando exactamente la misma cantidad de memoria que cada programa requiere. Se genera una partición del tamaño del programa que hay que cargar en memoria. Sin embargo este método de gestión de memoria tiene el inconveniente de la fragmentación de la memoria. Se llena la memoria con diferentes procesos. En un momento determinado se vacía espacio de dos procesos no contiguos. Si un nuevo proceso quiere entrar, si este es más grande que cualquiera de los procesos antiguos, no podrá entrar, aunque realmente hay memoria suficiente en el sistema. El método de segmentación es parecido al de partición dinámica solo que el programa en vez de guardarse en un solo bloque se separa en los segmentos de datos, código y pila, que se guardan en particiones separadas.

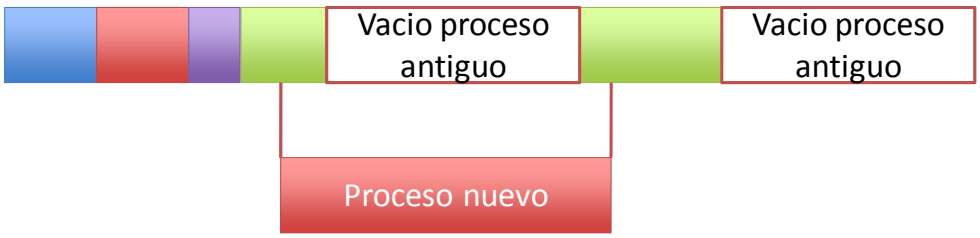


Figura 5. Fragmentación de la memoria.

El método de paginación se basa en dividir los programas en segmentos de longitud fija llamados páginas y dividir la memoria en marcos de página de igual tamaño. De manera que cada página se podrá posicionar en un marco de página. El tamaño de las páginas suele ser pequeño (512 bytes) de manera que el espacio inutilizado que puede quedar por al posicionar el último segmento del programa es pequeño. Además, la fragmentación de la memoria se evita al tener particiones fijas. Como contrapartida, este método exige una carga elevada de actividad de procesador para gestionar las páginas, los marcos de página y la asignación de páginas a marcos. Este es uno de los método más utilizado por los sistemas operativos, por ejemplo las últimas versiones de Windows o Unix.

1.3.3 Controladores

Tal como hemos dicho, el sistema operativo debe hacer lo más transparente posible el hardware de la máquina donde trabaja al programa. De esta forma se consigue que un mismo programa funcione con dispositivos completamente diferentes, dando resultados equivalentes. Ejemplos de esta situación van desde los teclados y ratones, a las pantallas y tarjetas gráficas, o las impresoras.

Para conseguir este objetivo, el sistema operativo define un dispositivo virtual con unas características genéricas. Evidentemente, no todos los dispositivos cumplen de la misma forma con dichas características, por lo que es necesario un traductor. Este traductor se denomina controlador.

El controlador realiza tres tareas básicas:

- Gestiona el dispositivo, lo inicializa cuando es necesario y mira el estado en que se encuentra.
- Indica que características concretas del dispositivo virtual cumple el dispositivo real.
- Traduce la información del dispositivo real para que cumpla con las especificaciones del dispositivo virtual.

Con todo ello se consigue que un programa siempre vea de la misma forma un dispositivo, lo cual simplifica mucho la tarea de la persona que desarrolla. Un ejemplo sería escribir en un fichero, que independientemente de que estemos utilizando (un disco duro interno, una memoria USB, un disco duro externo…) siempre se siguen los mismos pasos.

Otro ejemplo serían las impresoras. Todas ellas suelen trabajar en un pseudo lenguaje de composición de páginas que es diferente para cada fabricante. Para solucionarlo, el sistema operativo suele utilizar una impresora genérica con un lenguaje de paginación. Cuando un programa desea enviar algo a imprimir, utiliza este lenguaje, y posteriormente, es el controlador el que realiza la traducción, tal como se muestra en la Figura 6.

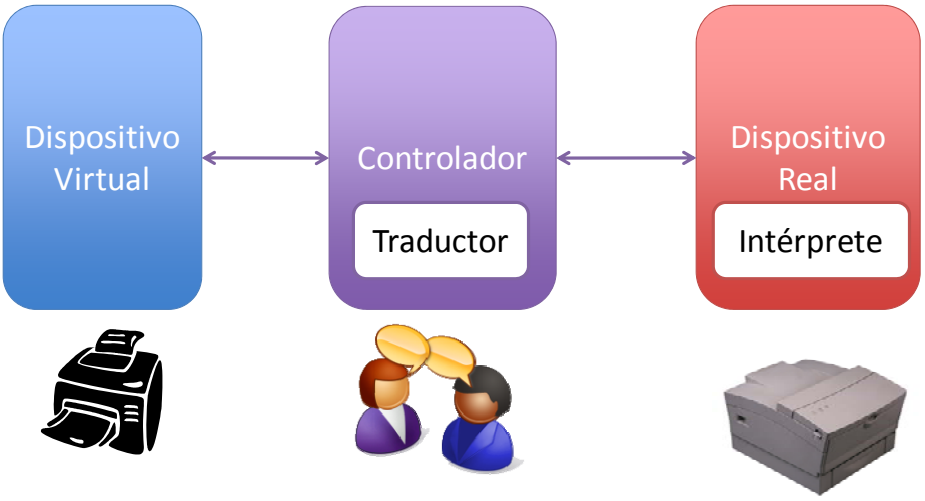


Figura 6. Diagrama de bloques de un controlador de impresora.

Por último, tenemos el caso de las tarjetas gráficas. Hoy en día se suelen utilizar dos pseudo lenguajes para gestionarlas, el OpenGL y las Direct X. En este caso, la propia tarjeta suele incorporar el traductor de este lenguaje, y el controlador se limita a transferir la información del sistema operativo al dispositivo.

Como vemos, las opciones son múltiples, pero el resultado final es siempre el mismo, el programa deja de ser dependiente de los periféricos, lo cual permite al programador dedicarse a optimizar su funcionamiento.

1.3.4 El sistema de ficheros

Los datos que se encuentran en la memoria externa suelen organizarse en archivos, también conocidos como ficheros. El concepto de archivo posibilita aislar al usuario de los problemas físicos de almacenamiento específicos de cada unidad de memoria externa. Cada archivo tiene asignado un nombre, atributos, descriptor de seguridad (quien puede acceder al archivo y con qué privilegios), dirección (donde se encuentran los datos). Los atributos incluyen información tal como fecha y hora de creación, fecha y hora de la última actualización, bits de protección (sólo lectura, o lectura y escritura), contraseña de acceso, número de bytes por registro, capacidad del archivo.

Para acceder al sistema de archivos y hacer operaciones como crear, leer o escribir el sistema operativo tiene una serie de llamadas al sistema como las recogidas en la Tabla 2.

Tabla 2. Llamadas al sistema para el acceso a archivos

Llamada al sistema	UNIX	Windows XP
Crear archivo	`open`	`CreateFile`
Borrar archivo	`unlink`	`DeleteFile`
Cerrar archivo	`close`	`CloseHandle`
Leer datos de un archivo	`read`	`ReadFile`
Escribir datos en un archivo	`write`	`WriteFile`
Obtener propiedades del archivo	`stat`	`GetFileAtributes`

La segunda abstracción que utiliza el sistema operativo para gestionar volúmenes de datos es la de directorio o carpeta. Los directorios o carpetas son conjuntos de archivos agrupados. La estructura global del sistema de archivos suele tener forma de árbol, en el que los nodos interiores son los directorios y los nodos exteriores son archivos. En la Tabla 3 se incluyen las llamadas al sistema más comunes relacionadas con la gestión de directorios o carpetas.

Tabla 3. Llamadas al sistema para el acceso a directorios

Llamada al sistema	UNIX	Windows XP
Crear un directorio	`mkdir`	`CreateDirectory`
Borrar un directorio vacío	`rmdir`	`RemoveDirectory`
Abrir un directorio para lectura	`opendir`	`FindFirstFile`
Leer el siguiente elemento del directorio	`readdir`	`FindNextFile`
Eliminar un archivo de un directorio	`unlink`	
Llevar un archivo de un directorio a otro		`MoveFile`

1.3.5 Sistemas operativos de Microsoft

Son los sistemas operativos dominantes en el mercado de microcomputadores. El primero de ellos fue el **MS-DOS** (Microsoft Disk Operating System) y debe su difusión a que fue adoptado por IBM al principio de la década de los 80 como el sistema operativo estándar para el IBM-PC. En la Figura 7 se muestra un esquema de la evolución de este sistema operativo. El MS-DOS inicial era un sistema operativo para microprocesadores de 16 bits (de Intel), monousuario y tenía una interfaz de usuario de línea de órdenes. A inicios de los 90 se comercializó el Microsoft *Windows 3.0* que disponía de una GUI (Graphic User Interface) que permitía mostrar ventanas con gestión de menús, y era capaz de cargar en memoria más de un programa a la vez. En 1995 Microsoft comercializó el *Windows 95*, que contiene una GUI basada en iconos, y es un sistema operativo de 16/32 bits con multiprogramación apropiativa (puede suspender temporalmente la ejecución de un trabajo para ejecutar otro) y con memoria virtual. Posteriormente se comercializaron las versiones *Windows 98, Windows ME (Millennium Edition)* en 1998 y 2000 respectivamente. Paralelamente al desarrollo de estas versiones de Windows, Microsoft comercializó el **Windows NT** (Windows New Technology), diseñado fundamentalmente para estaciones de trabajo potentes y

servidores de red con procesadores de 32 bits. En 2001 se disponía de tres alternativas: *Windows 2000 professional, Windows NT Server y Windows CE* (Consumer Electronics).

En el año 2001 se comercializó el Windows XP, que supuso la unificación de la línea clásica de los sistemas operativos de Microsoft (Windows 3.1/95/98/98SE/ME) con la de Windows NT (NT 3.1, 4.1 y Windows 2000), véase Figura 7. El Windows XP contiene el núcleo del Windows NT y se comercializó en dos versiones: Home y Professional.

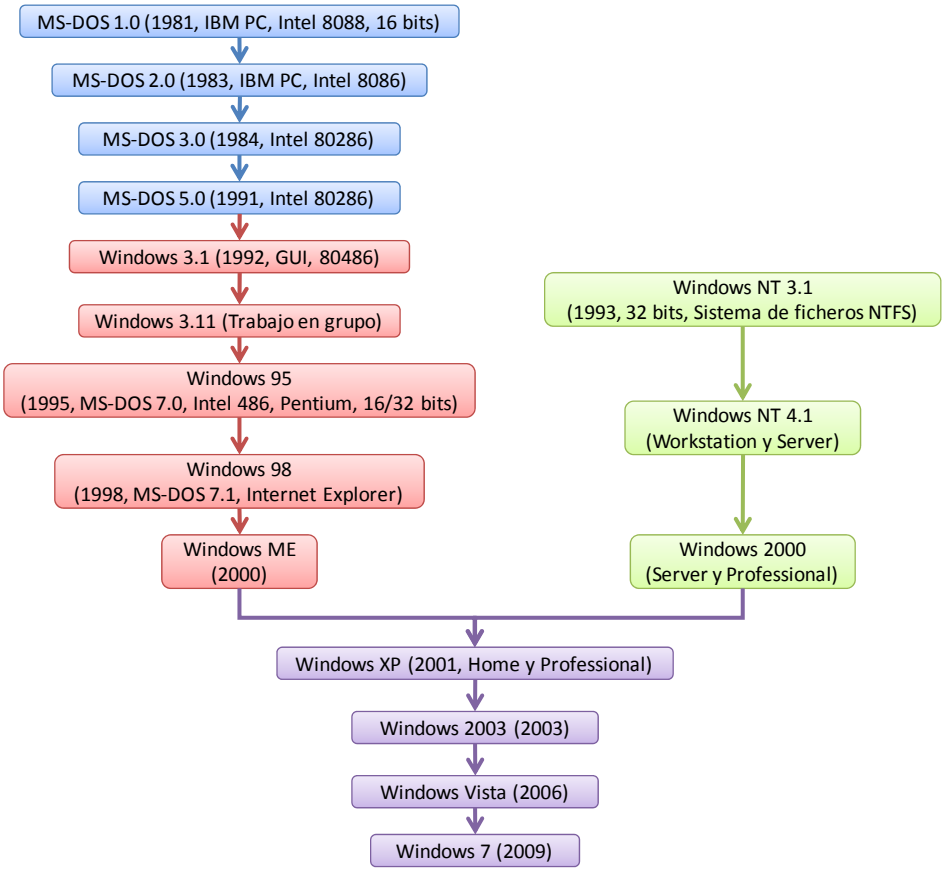


Figura 7. Evolución de los sistemas operativos de Microsoft

Posteriormente apareció el Windows 2003, destinado a servidores de bases de datos, aplicaciones, Internet… El núcleo era fundamentalmente el mismo que el de XP, pero optimizado para realizar las tareas anteriormente mencionadas, lo cual implica una alta tasa de transferencias de información por red, seguridad, multiusuario…

Más recientemente llego el Windows Vista, que tuvo muchos problemas iniciales, debidos fundamentalmente a algunas incompatibilidades con el Windows XP, al nuevo sistema de seguridad que era bastante engorroso y a la falta de controladores para algunos dispositivos. Esto llevó a muchas empresas a retrotraer al Windows XP sus nuevos equipos.

El último sistema operativo de Microsoft es el Windows 7. Éste se espera que de respuesta a todos los problemas encontrados en Vista.

1.3.6 Sistemas operativos UNIX

El sistema operativo UNIX inicialmente se desarrolló en los Bell Labs de la compañía AT&T en 1970 y fue comercializado en sus inicios por Honeywell. A principios de la década de los 90 Novell paso a ser el propietario. En la Figura 8 se incluye un esquema simplificado de la evolución de este sistema operativo. Una de las versiones más conocidas es la desarrollada en el Campus de Berkeley de Universidad de la Universidad de California.

Desde un punto de vista técnico, UNIX puede considerarse como un conjunto de familias de sistemas operativos que comparten ciertos criterios de diseño e interoperabilidad. Esta familia incluye más de 100 sistemas operativos desarrollados a lo largo de 35 años. Pueden funcionar en multitud de computadores, desde un supercomputador como el Mare Nostrum, pasando por un PC de sobremesa, y llegando a una agenda electrónica o un teléfono móvil de última generación. Debido a ello puede considerarse como un auténtico sistema operativo estándar y la columna vertebral de Internet. Permite multiprogramación, multiusuario y multiprocesamiento, pudiéndose utilizar también en entornos de tiempo real.

Un estudiante de informática de la Universidad de Helsinki (Finlandia) llamado Linus Torvalds concluyó en 1991 (con 23 años de edad) un sistema operativo que denominó Linux, versión 0.01. Este sistema operativo fue concebido como una versión PC compatible y sustancialmente mejorada del sistema operativo Minix (muy parecido al UNIX), descrito por el profesor Tanenbaum en sus primeros textos de sistemas operativos. Aunque Linux no es un programa de dominio público, se distribuye con una licencia GPL (General Public License) de GNU, de forma que los autores no han renunciado a sus derechos, pero la obtención de su licencia es gratuita, y cualquiera puede disponer de todos los programas fuente, modificarlos y desarrollar nuevas aplicaciones basadas en Linux.

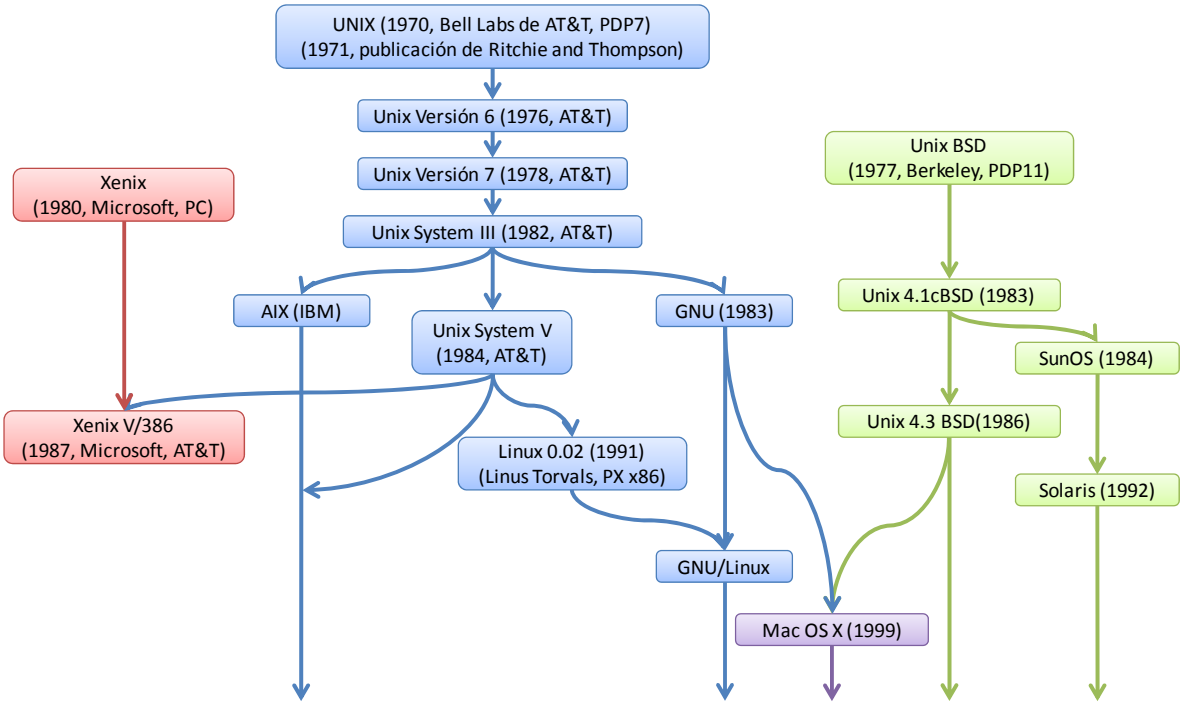


Figura 8. Evolución de los sistemas operativos UNIX

En 1999 Apple presento un sistema operativo basado en BSD, con un kernel Mach. Éste se distribuye bajo licencia de código abierto como Darwin. Por encima corre una interfaz gráfica muy elaborada, que es lo que le diferencia fundamentalmente del resto de Unix. En el resto de aspectos, Darwin es un Unix más y comparte un gran número de aplicaciones.

Finalmente, indicar que en la actualidad, las versiones de Unix más extendidas son Mac OS X, Linux y Solaris.

1.4 Periféricos

Cuando hablamos de periféricos, solemos referirnos a los que nos permiten interactuar con el ordenador:

- Pantalla
- Teclado
- Ratón

Sin embargo, existen todo un conjunto de dispositivos que suelen estar incluidos dentro del ordenador, y que generalmente damos por supuesto que funcionan. En esta sección vamos ha ver sus principales características para entender su funcionamiento.

Nos centraremos en dos periféricos básicos, como son el disco magnético y el disco óptico. También estudiaremos los principales buses de conexión cableados de periféricos (USB y Firewire) así como inalámbricos (Bluetooth).

1.4.1 Discos magnéticos

El disco duro es el medio más utilizado para el almacenamiento de grandes volúmenes de información, debido a su reducido coste, alta velocidad y gran capacidad. Éstos discos utilizan un apilamiento de platos de discos magnéticos montados en estructuras rígidas y aislados de la posible contaminación ambiente y atmosférica.

Estos discos, también conocidos como tecnología Winchester, debido al nombre que le dio IBM al primer modelo que tenia los discos y los cabezales montados en una estructura rígida para darle mayor velocidad, capacidad y fiabilidad. Este primer disco tenía una capacidad de 30MB.

Todos los sistemas basados en discos, guardan la información en superficies circulares divididas en pistas concéntricas, como se muestra en la Figura 9. Cada pista se divide a su vez en sectores. Para acceder a un sector determinado, la cabeza lectora/grabadora pivota sobre los discos en rotación hasta encontrar la pista correcta. Cuando se encuentra sobre ésta, espera hasta que el disco rote hasta el sector correcto. En este punto puede empezar a leer la orientación magnética de los puntos del sector, o bien graba una nueva orientación, en función de la necesidad de ese momento.

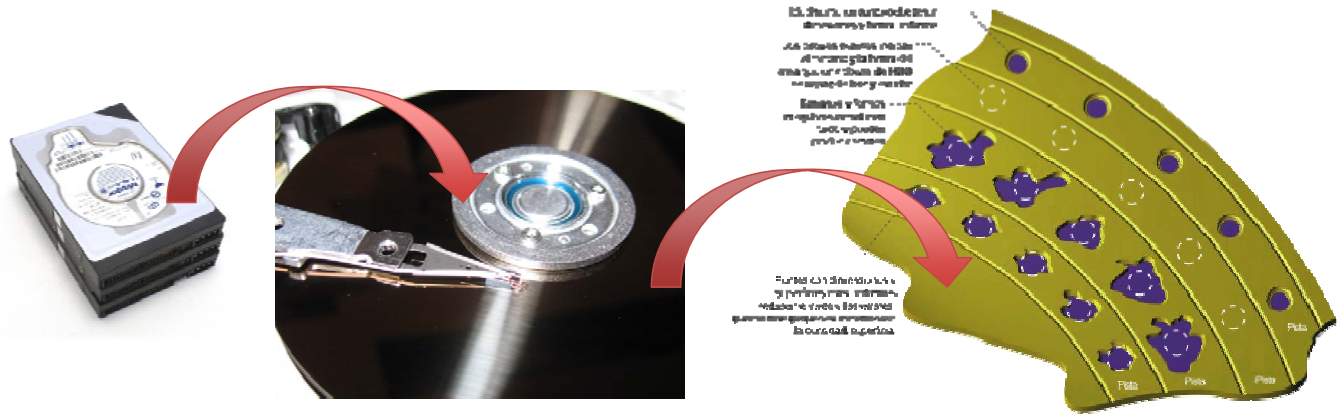


Figura 9. Del disco magnéticos a los puntos en el plato.

El propósito de montar todo el sistema electromecánico en un contenedor hermético es para evitar la contaminación de partículas en los platos. De esta forma las cabezas lectoras pueden volar a una distancia mínima respecto del disco sin posibilidad de chocar contra ningún objeto. Esta menor altura permite leer y escribir información con menores dimensiones, a la vez que permite un funcionamiento más rápido, al poder pasar más bits en una unidad de tiempo.

1.4.2 Discos ópticos

Otro posible método de almacenamiento es utilizando dispositivos ópticos. En este caso, en lugar de modificar propiedades magnéticas de un material, se modifica la forma en que interacciona con la luz. Éste es el caso de tecnologías como el CD o el DVD.

Existen varios tipos de almacenamiento en estos dispositivos. La primera son los CD-ROM o DVD-ROM. En este caso, la información se guarda en forma de microsurcos en un disco transparente. Los cambios de altura, obtenidos por zonas con agujeros (Pits) y zonas sin, permiten indicar la información guardada. Para crear estos microsurcos se utiliza un método de estampación sobre un sustrato de policarbonato (PC).

Para conseguir discos cuya superficie se puede escribir se utilizan materiales con cambio de fase. En la fase cristalina son perfectos reflectores de la luz, y en la fase amorfa son malos reflectores. Este material se introduce como una capa tinte en el disco, tal como se muestra en la Figura 10. Este cambio de fase es detectado un láser, cuya diferencia es leída como un 1 o un 0.

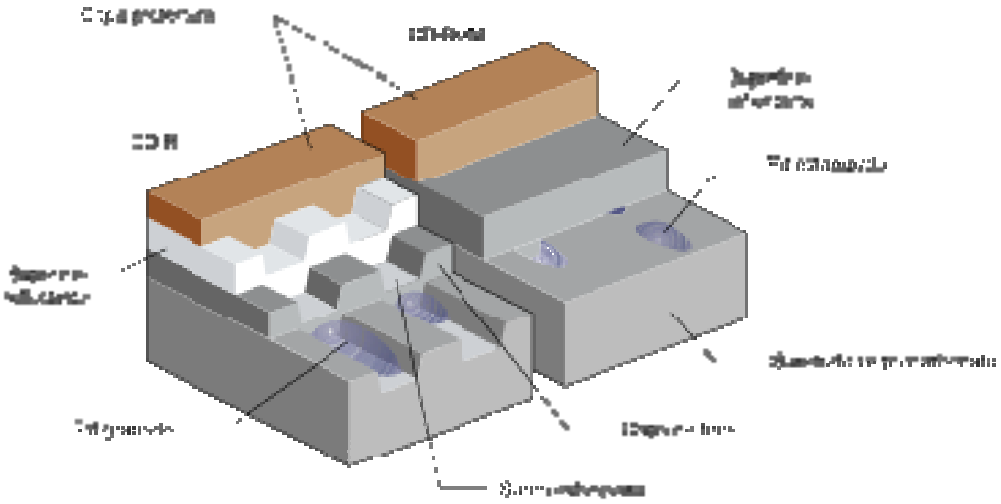


Figura 10. Capas de un CD-ROM y un CD-R

Los CDs grabables (CD-R), los DVD-R y los DVD+R utilizan un material que solo puede ser modificado una vez de estado cristalino a estado amorfo. En el caso de los discos regrabables (CD-RW, DVD-RW y DWD+RW) se utilizan materiales que pueden modificar su estado miles de veces (los defensores de los sistemas magnéticos dicen que unas 100.000 veces).

En general, las tecnologías del CD y del DVD para su lectura y escritura son muy semejantes, aunque el DVD aprovecha los años de experiencia del CD e incluye algunas mejoras respecto a este.

En primer lugar, el DVD utiliza un lector láser rojo que tiene una longitud de onda inferior a la del infrarrojo con la que trabaja el CD. Esta reducción de la longitud de onda permite reducir el tamaño del pit. En la Figura 11 se muestra una comparación entre el tamaño del pit y la distancia entre pistas.

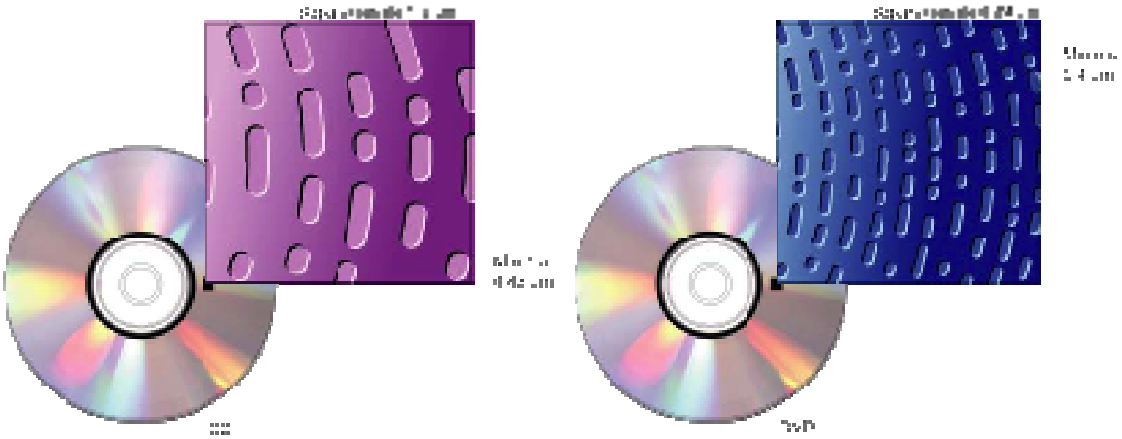


Figura 11. Comparación de tamaño de pit entre CD y DVD

Con esta reducción de dimensiones se consigue incrementar la capacidad del DVD en un 2,56. Esto unido a mejoras en las técnicas de codificación y en el sistema de ficheros han permitido incrementar la capacidad por capa en 6:1.

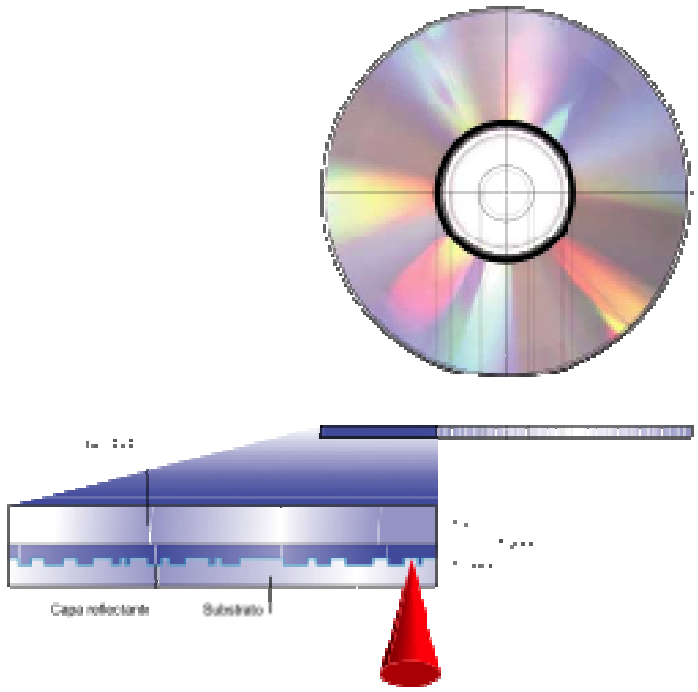


Figura 12. Estructura de un DVD simple cara una capa

En la Figura 12 se puede ver que el sustrato del DVD esta divido en dos capas. Esto permite reducir las deformaciones, y proporciona la posibilidad de escribir en ambos substratos. De esta forma se puede incrementar la capacidad del DVD por dos. Por último, indicar que cada substrato se puede dividir en dos, con lo que se consigue incrementar la capacidad por otros dos. Utilizando un DVD doble cara doble capa se puede llegar a obtener una capacidad de 18GB.

En la actualidad, está popularizándose el formato Blu-ray, basado en un láser azul, lo que permite incrementar la capacidad hasta los 25GB (una cara) y 50GB (doble cara).

1.4.3 USB

El estándar USB (Universal Serial Bus) fue definido por Intel con la intención de sustituir todo el conjunto de estándares para la conexión de dispositivos de baja velocidad al ordenador (serie, paralelo, teclado, ps/2...). Este protocolo trabaja a 12Mbps, y puede ser utilizado la conexión de dispositivos multimedia como cámaras de Web.

Debido a los requerimientos actuales de algunos periféricos, como las impresoras de alta calidad o sistemas de almacenamiento masivo como los DVD, Intel decidió ampliar el estándar USB para dar servicio a este tipo de aplicaciones. A este estándar le ha dado el nombre de USB 2.0 y permite trabajar a 480Mbps (40 veces más rápido que su predecesor), manteniendo la compatibilidad con su predecesor.

El comportamiento del USB es equivalente al de una red, con una gestión basada en el modelo maestro esclavo. Un dispositivo es la cabecera de la red, y el que establece que dispositivo debe acceder a dicha red en cada momento. Dicha cabecera suele ser un PC. Éste, haciendo uso de controladores, establece el comportamiento del dispositivo, y gestiona la utilización de la red. Esto permite reducir los costes asociados al hardware, ya que el dispositivo precisa de menor inteligencia.

Para interconectar varios dispositivos se utilizan concentradores, que en función de los requerimientos de velocidad pueden ser USB2.0 de altas prestaciones o USB1.1 cuando necesidades son inferiores. En la Figura 13 se muestra la topología de esta red, que tiene siempre un ordenador como raíz del árbol.

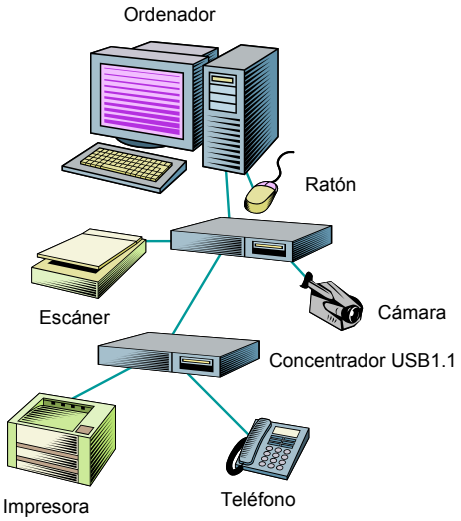


Figura 13. Topología en árbol del USB2.0

Incluye un modo de transferencia isócrono, pensado para transferencia de audio y video. Este envía información con una control temporal muy preciso, pero en la que no se realiza ninguna comprobación de errores. Para poderlo utilizar el PC debe dar su conformidad.

En la actualidad este bus está muy introducido en el mercado. Siendo uno de los más utilizados para la interconexión digital de dispositivos.

1.4.4 Firewire

El Firewire o IEEE 1394, es un estándar que diseño inicialmente Apple como red para la transmisión de información entre dispositivos multimedia. Fue estandarizado en 1995. Su principal característica es que permite trabajar de forma isócrona, asegurando que todos los nodos reciben la misma información a la cadencia deseada. Por este motivo es una red optima para aplicaciones de video y audio digital, aunque permite intercambiar todo tipo de información.

Para lograr el gran control preciso para el modo isócrono, utiliza una topología en árbol. En la Figura 14 se muestra una ejemplo de la topología, donde se puede ver que los diferentes dispositivos se interconectan a través de sus superiores, siendo el nodo raíz el STB, aunque en muchos es un ordenador.

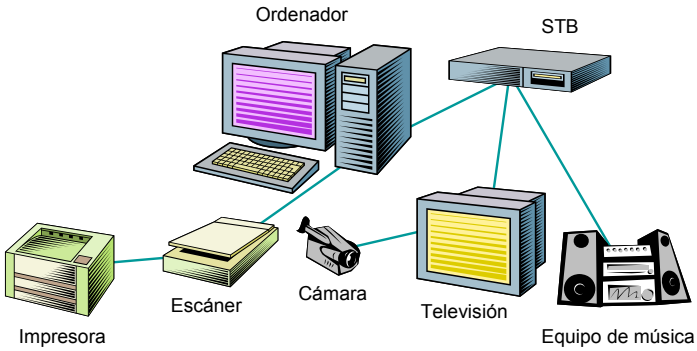


Figura 14. Topología en árbol del estándar IEEE 1394

Desde el punto de vista de un dispositivo conectado a la red, esta es como un gran espacio de memoria virtual al que puede acceder. Cada dispositivo ocupa una determinadas direcciones de esta memoria, con direcciones de 48bits de longitud (256 Terabytes). En cada tramo de red pueden haber hasta 64 dispositivos conectados, y la red puede contener un total de 1024 segmentos. Esto significa que se puede acceder a 16 Exabytes de memoria, muy superior a las necesidades de cualquier sistema actual.

Como en el caso del USB, existe un modo de transmisión isócrono permite enviar información de un nodo a otro o a todos. En este modo, no hay corrección de errores o retransmisión de la información, y puede llegar a ocupar un 80% de la capacidad del bus, siendo esta de 800Mbps. Un nodo puede solicitar en cualquier momento la utilización del modo isócrono, pero solo se le concede cuando la red tiene disponible capacidad de bus para hacerlo. En caso contrario, el nodo se tiene que esperar.

En modo asíncrono, un nodo puede enviar información cuando el canal no está ocupado. Este modo servicios de control de errores y de repetición de la trama, y esta especialmente indicado para aplicaciones que no permiten errores, como el acceso a un disco duro.

En la actualidad los dispositivos que utilizan este bus son cámaras digitales, videos digitales y grabadores de DVD. Además, también se utilizan con PCs. Permite que cualquier dispositivo se conecte con cualquier otro, ya que los considera iguales. A modo de ejemplo, una cámara puede conectarse con un grabador de DVD y ser controlada desde un televisor, todo ello sin necesidad de un ordenador. Sin embargo, ha ido perdiendo fuerza frente al USB, y algunos ordenadores de Apple ya no lo incorporan.

1.4.5 Bluetooth

La red Bluetooth o IEEE 802.15.1 es el equivalente inalámbrico del USB 1.1. Nació con el objetivo de permitir a pequeños dispositivos se interconecten entre sí en lo que se ha denominado una Red de Área Personal (*Personal Area Network*, PAN). Posibles aplicaciones de esta red es la intercomunicación de un teléfono móvil con unos auriculares, una PDA con un móvil o un teclado con un PC. La distancia máxima entre nodos es de 10 metros.

Transmite en la banda ISM de 2,4GHz, haciendo uso del esquema de transmisión de Espectro Disperso por Salto de Frecuencia (*Frequency Hopping Spread Spectrum*, FHSS). Este esquema divide el espectro de frecuencias en sub-bandas de ancho equivalente a la tasa de bits a transmitir. Durante la transmisión de la señal, se va saltando de una sub-banda a otra, de forma que se utiliza todo el espectro, y se disminuye la posibilidad de interferencias.

Existen dos modalidades de FHSS, la rápida, que realiza varios saltos por bit transmitido, y la lenta que transmite varios bits por salto. En el caso del Bluetooth se utiliza el método rápido para incrementar la seguridad.

Un dispositivo Bluetooth puede comunicarse con otros ocho dispositivos. Cuando se crean los enlaces entre los diferentes dispositivos, se crea una piconet o PAN, la cual es supervisada por un nodo maestro. Todos ellos hacen

uso de la misma secuencia de salto. Un nodo esclavo puede pertenecer a varias piconets, tal como se muestra en la Figura 15.

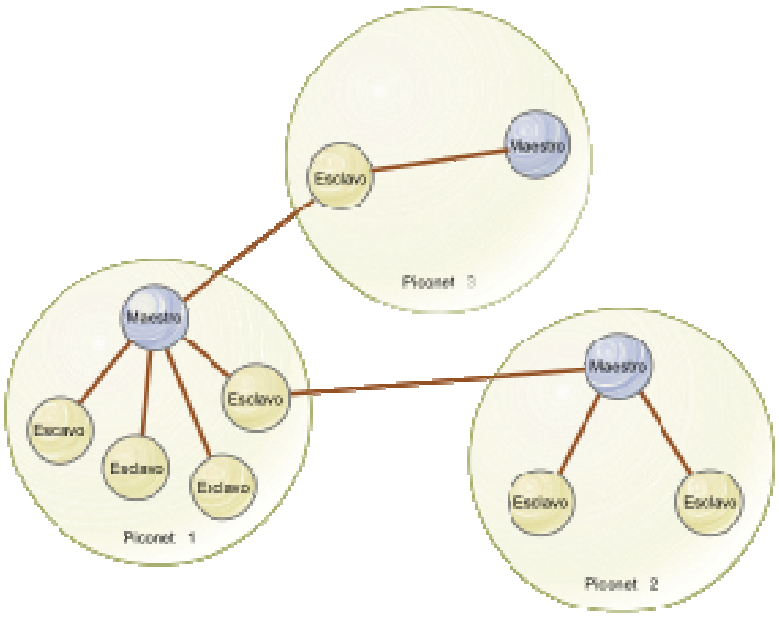


Figura 15. Arquitectura de una red Bluetooth

Cada piconet se conecta a una velocidad de hasta 721Kbps, y es el maestro quien determina como se distribuye el ancho de banda. Pueden existir hasta 10 piconets de 8 dispositivos, con lo que el ancho de banda alcanzando transferencia útil de unos 6Mbps. Para incrementar la seguridad, además de realizar saltos 1600 veces por segundo, utiliza encriptación 64 bits y autentificación por dispositivo.

En la actualidad ya existen multitud de dispositivos preparados para conectarse a través de Bluetooth, como cámaras fotográficas, PDAs, teléfonos móviles, manos libres, teclados, ratones, PCs…

1.5 Redes

Hoy en día, la mayoría de ordenadores se conectan entre sí a través de una red. De esta forma pueden intercambiar información, ver el estado de un determinado proceso de forma remota o enviar mensajes. La mayoría de las redes están interconectadas, formando una red mundial conocida como Internet. Internet está constituida por equipos de muy diversa capacidad (desde supercomputadores hasta equipos móviles), y proporciona a decena de millones de personas la utilización de múltiples servicios como el acceso a recursos hardware compartidos, correo electrónico, transferencia de archivos, noticias, acceso a documentos multimedia, transacciones bancarias y comercio electrónico. Podemos definir una red de ordenadores como un conjunto de equipos autónomos interconectados a través de un medio por el que se intercambian información. Entendemos por medio el sistema (material o no) que nos sirve para transmitir la información. Entre los medios de comunicación más destacados tenemos el hilo o conjunto hilos de cobre, el cable coaxial, el aire y la fibra óptica.

Todo ordenador o equipo que pretenda conectarse a una red necesita de un hardware específico, módem o tarjeta de interfaz de red (NIC) y un programa especial de comunicaciones. Ambos elementos deben adecuarse a las característica y protocolos de la red.

Las redes de ordenadores aumentan notablemente el cociente prestaciones/precio de sistemas informáticos. Las principales ventajas que se obtienen son:

- Posibilitan un servicio remoto de utilización de aplicaciones, sin necesidad de que el usuario disponga realmente de ellas.
- Pueden establecerse procesos en distintas máquinas, actuando de esta forma en paralelo y por consiguiente, mejorando las prestaciones al colaborar para la realización de una determinada tarea.
- Permite el acceso a documentos de tipo texto o multimedia en general. Con estos servicios se puede acceder a información de bases de datos y archivos desde equipos localizados a grandes distancias.
- Admiten la gestión de bases de datos distribuidas, de forma que un único sistema no necesite tener la capacidad suficiente para albergar la base de datos completa.
- Aumentan la seguridad del sistema a través de la redundancia.
- Si el equipo local al que se tiene acceso directo no dispone de las prestaciones adecuadas (poca memoria, o bien está muy cargado de trabajo, no dispone de impresoras rápidas o de suficiente calidad de impresión, etc.), el usuario puede conectarse a otro equipo de la red que reúna las características pertinentes.
- Permiten la utilización de la red de ordenadores como medio de comunicación: correo electrónico, radio y televisión a través de Internet, etc.

Los ordenadores se interconectan a través de una red de comunicaciones que puede ser una red de área amplia (WAN), cuyo ámbito geográfico es un entorno regional, nacional o internacional, una red de área metropolitana (MAN) que interconecta equipos de una ciudad, o una red de área local (LAN) cuyo ámbito es el de un departamento, un edificio o un campus. En la actualidad también están adquiriendo gran relevancia un tipo de redes inalámbricas de muy corto alcance. Este tipo de red se suele denominar red de área personal (PAN).

1.5.1 El Módem

Durante los últimos años la transmisión analógica ha dominado las comunicaciones. Hoy en día sin embargo, desde la implantación de la RDSI, y más recientemente el xDSL y FTTH, las comunicaciones se basan en transmisiones digitales, que nos permiten velocidades de transferencia muy superiores a aquellas que se podían conseguir mediante los sistemas basados en transmisión analógica. En particular, el sistema telefónico se basó inicialmente en la señalización analógica. En la actualidad las líneas troncales metropolitanas y de larga distancia son digitales, aunque existen aun lazos locales que son analógicos y probablemente seguirán así durante un tiempo por el coste

de convertirlos al sistema digital. En consecuencia, cuando tenemos un equipo conectado a Internet desde casa, el sistema envía datos digitales por una línea analógica. Los datos se MOdulan y DEModulan para poderse transmitir por el lazo analógico. Posteriormente, cuando los datos llegan al proveedor de servicios, se vuelven a convertir en señales digitales que se dirigen al destino correspondiente. El módem convencional permitía velocidades de comunicación relativamente pequeñas, unos 56Kbits/s. En la actualidad los modems ADSL 2+ permiten velocidades muy superiores: 25 Mbits/s de bajada y hasta 3,5 Mbit/s de subida.

1.5.2 La Tarjeta de Interfaz de Red (NIC)

Una Tarjeta de Interfaz de Red es una placa de circuito impreso cuya función es la conexión del PC a una red local de comunicaciones (LAN). Es por esta razón que también se le conoce por el nombre de adaptador LAN.

La conexión de la NIC con la placa base del PC se realiza a través del puerto de expansión, aunque en la actualidad se puede encontrar integrada en la placa base del ordenador. Aunque existen diversos tipos de NIC para los diferentes protocolos de comunicaciones, las mas utilizadas son las NIC Ethernet.

Usualmente, una NIC se comunica con la red de área local a través de una conexión serie(es decir, los datos se transmiten bit a bit). Para ello suele utilizarse un cableado estándar, definido por la organización EIA-TIA como el UTP Categoría 5 ó 6. La conexión entre el cableado y el PC se realiza a través de un conector estándar: el RJ-45. Sin embargo, cada vez más las conexiones entre PC's y LAN's se realizan a través del medio inalámbrico, es lo que se conoce como el *Wireless Fidelity* o WiFi. Una LAN cableada puede llegar a transmitir datos a velocidades de 1000Mbits/s. Por lo que respecta a las comunicaciones inalámbricas, las velocidades llegan hasta los 54Mbits/s.

Los datos que nos llegan desde la LAN (conexión serie) son transmitidos hacia el PC a través de una conexión paralela. De la misma forma, los datos transmitidos desde el PC hacia la LAN se reciben a través de una conexión paralela y se envían a través de una conexión serie.

Las tarjetas NIC llevan un número de identificación conocido como dirección Hardware o dirección MAC. Este número es de 48 bits y es diferente para cada tarjeta de red. Los primeros 24 bits identifican al fabricante de la NIC. Los restantes 24 bits son la identificación del producto. Cuando un ordenador transmite una serie de datos (un e-mail por ejemplo) hacia un determinado destino, el equipo origen debe conocer la dirección MAC del destino e introducirla en el mensaje juntamente con su propia MAC. Cuando la NIC del ordenador destino recibe los datos, primero hará una verificación de la dirección MAC, posteriormente determinará si los datos recibidos son correctos o ha habido algún problema durante la transmisión de estos (datos erróneos). Una vez verificado todo esto, los datos se transmitirán al PC para su procesado posterior.

Ejercicio 2: Identificando la dirección MAC de nuestro equipo

- Abrimos una consola
- Escribimos `ipconfig` (Windows) o `ifconfig` (Linux-MAC)
- Recordar que la dirección MAC es de 48 bits. ¿Cuál es vuestra MAC?

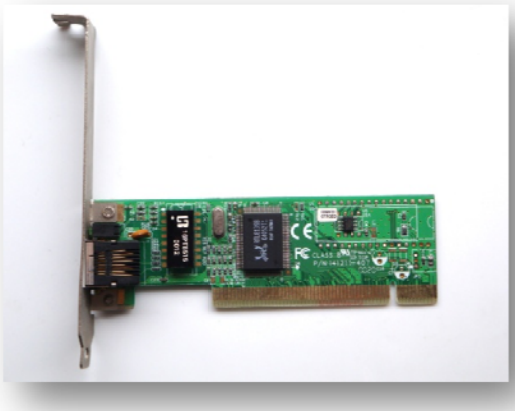


Figura 16. Tarjeta NIC de conexión para conexión de PC's a redes de área local.

En el caso de las conexiones inalámbricas (generalmente WiFi), la velocidad que se logra es muy inferior al caso de una LAN cableado. Por el contrario, se gana en movilidad, ya que no necesario conectar ningún cable para tener acceso a la red. Este hecho a permitido llevar Internet a sitios como un parque o una playa, para que las personas puedan trabajar y distraerse en un ambiente más relajado.

Ejercicio 3: Identificando la dirección MAC de otros equipos en nuestra tabla local.

- Abrimos la consola y ejecutamos el comando `ifconfig`
- Apuntamos la dirección IP de nuestro ordenador y se la pasamos a varios compañeros.
- Accedemos a los ordenadores de nuestros compañeros para ver si están conectados. Ejecutamos el comando Ping + IP del compañero
- Lo hacemos para varios compañeros…
- Finalmente ejecutamos el comando `arp –a`. Esto nos mostrará las direcciones MAC de las diferentes máquinas a las que hemos contactado.

La tabla donde se guardan las diferentes direcciones MAC está en una memoria volátil que se va actualizando a medida que se necesita. Es posible que se quiera enviar información a un determinado PC del que conocemos la IP pero del que se desconoce la dirección MAC. Cuando se produce este caso (más habitual de lo que se pueda pensar), se ejecuta un protocolo conocido como ARP. El ordenador origen envía un mensaje indicando sus direcciones MAC e IP e indicando que quiere conocer la dirección MAC de un equipo cuya IP es conocida. Cuando el ordenador destino recibe este mensaje, completa el campo de su dirección MAC para que el origen pueda actualizar su tabla ARP. Este formato se utiliza para redes locales. Si la comunicación se establece con un PC que pertenece a otra red, el funcionamiento es diferente.

1.5.3 Un modelo para las comunicaciones de datos: La pila TCP/IP

El departamento de defensa de los USA creó el modelo de referencia TCP/IP puesto que necesitaba una red robusta, capaz de sobreponerse a cualquier problema y permitiese la transmisión de datos entre diferentes ordenadores. TCP/IP es el acrónimo de Transmission Control Protocol / Internet Protocol. Este protocolo está formado por 4 niveles o capas, que cubren desde la parte más física (Hardware y cableado) hasta la parte de software que se encarga de la comunicación (http, ftp,…)

El TCP/IP es la base de Internet, y sirve para enlazar computadoras que utilizan diferentes sistemas operativos, incluyendo PC, minicomputadoras y computadoras centrales sobre redes de área local (LAN) y área extensa (WAN). Los cuatro niveles que forman el protocolo TCP/IP son:

- Capa de aplicación: Es la capa de más alto nivel en el protocolo. Se encarga de la presentación, la codificación de los datos y de herramientas de control. Algunos de los protocolos que conforman esta capa son SMTP (correo electrónico), FTP (transferencia de archivos), TELNET (conexiones remotas) y http (Hypertext Transfer Protocol).
- Capa de transporte: La capa de transporte típicamente realiza tareas de proporcionar la fiabilidad, necesaria para el transporte de datos, control de flujo de datos y retransmisiones en caso de errores. Tenemos dos protocolos dentro de esta capa: TCP y UDP. TCP es un protocolo orientado a conexión mientras que UDP es un protocolo no orientado a conexión. TCP es el protocolo más comúnmente usado. Encapsula http, FTP, y SMTP entre otros.
- Capa de red: El propósito de esta capa es el envío de paquetes desde cualquier ordenador de cualquier red conectada a Internet con cualquier destino que se encuentre conectado. El protocolo específico que gobierna esta capa es el Internet Protocol (IP). Para lograr su cometido, el protocolo IP se ayuda de protocolos de enrutamiento, que le permiten encontrar el camino para llegar de un punto a otro (estos protocolos serían como el navegador para un conductor).
- Capa de acceso al medio: El nombre de esta capa puede dar lugar a confusión. También se le conoce como capa de interconexión con el medio físico. Normalmente se compone de dos capas, la capa de enlace, encargada de montar la trama de datos y asegurar el control de errores y flujo en cada salto y la capa física, en donde se definen las señales utilizadas para codificar los bits. Es en esta capa donde intervienen la NIC o el módem. También encontramos los concentradores y conmutadores, que permiten interconectar segmentos de red.

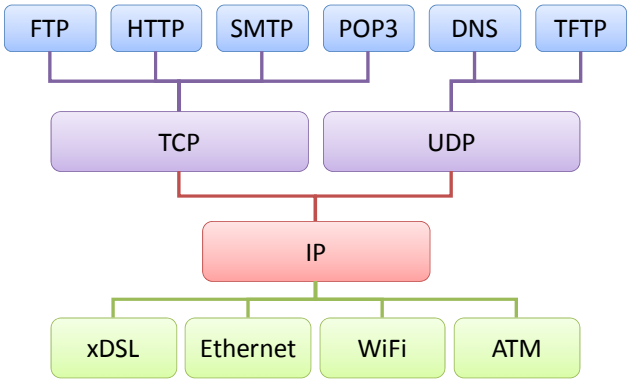


Figura 17. Representación gráfica de la pila de comunicaciones TCP/IP

En la Figura 17 se nuestra un diagrama donde se muestran los diferentes protocolos que se tienen en TCP/IP. En la capa de aplicación se encuentran algunos de los diferentes protocolos más utilizados en Internet. Es probable que el lector los use varias veces cada día.

En Internet un ordenador individual se identifica mediante una dirección IP que está formada por dos partes: el código de identificación de la red y el código identificador del equipo. La dirección IP está formada por 4 Bytes, que se suelen dar en decimal y separados por puntos.

Ejercicio 4: Determinar la dirección IP de nuestro ordenador. Utiliza para ello la instrucción `ipconfig` (Windows) o `ifconfig` (Linux-MAC). Una vez descubierta nuestra IP, determinar una de las IP de Google ejecutando el comando `traceroute` www.google.com

1.5.4 Aplicaciones de Internet

En este apartado se explican de forma general algunas de las aplicaciones de Internet más utilizadas y que ya han sido comentadas en el apartado anterior.

1.5.4.1 Conexión de Terminal Remoto (TELNET)

Posibilita a un usuario conectarse a cualquier equipo remoto en cualquiera de las redes interconectadas utilizando el conjunto de protocolos TCP/IP. La conexión se hace emulando, en el equipo del usuario un terminal virtual. De esta forma puede utilizar el terminal como si estuviese conectado directamente a él.

En la actualidad, este tipo de conexión ha evolucionado con el objetivo de conseguir conexiones seguras, a través del protocolo SSH.

1.5.4.2 Transferencia de archivos

La transferencia de archivos a través de Internet utiliza un protocolo para la transferencia de ficheros conocido como FTP (File Transfer Protocol). Este protocolo permite transferir ficheros desde un determinado servidor hacia aquella persona que haya realizado la petición de transferencia (cliente). Existen programas específicos que implementan el protocolo FTP, aunque también puede ejecutarse desde la consola de cualquier ordenador. La instrucción a implementar es:

- ftp dirección IP

a partir de aquí la captura de ficheros se realiza mediante la instrucción `get`. La subida de ficheros al servidor se realiza mediante la instrucción `put`.

La conexión entre un cliente y un servidor a través de FTP es una conexión no segura ya que no lleva ningún tipo de cifrado. Para transferir ficheros de forma segura se utilizan aplicaciones tales como SFTP (Secure FTP) que añaden las técnicas necesarias para establecer la conexión de forma segura.

1.5.4.3 Sistema de ficheros en la red (NFS)

NFS es un protocolo que permite acceder al cliente a un archivo remoto, como si estuviese en la máquina del servidor, de forma que puede editarlo, o incluso transferirlo a una tercera máquina si fuese necesario.

1.5.4.4 Correo electrónico

El correo electrónico tal vez sea el servicio más utilizado en Internet, ya que con él un usuario puede enviar mensajes o archivos a otro ordenador. El sistema se fundamenta en la utilización del concepto de buzón y de dirección de buzón. El buzón está ligado con una persona o ente determinado. Una dirección de buzón se compone de dos partes diferenciadas y separadas por el carácter @. La primera parte corresponde con la dirección del buzón dentro del dominio mientras que la segunda corresponde al nombre del dominio.

El correo electrónico en Internet se implementa con el protocolo SMTP (Simple Mail Transfer Protocol).

1.5.4.5 Tertulias en Internet o chats (IRC)

El IRC o chat es una aplicación mediante la cual distintos usuarios de Internet pueden comunicarse entre sí en tiempo real. La diferencia fundamental con el correo electrónico reside en que la comunicación es instantánea.

1.5.4.6 Aplicaciones multimedia

En este grupo se incluyen aplicaciones tales como videoconferencia, transmisión de audio o vídeo bajo demanda realizada utilizando el protocolo IP. Este conjunto de aplicaciones se caracteriza por la necesidad de utilizar un gran ancho de banda y realizar ejecuciones en tiempo real, con retardos muy bajos y una alta calidad. Suelen utilizar técnicas de compresión de datos con el fin de reducir el ancho de banda.

1.5.5 La web

La red mundial World Wide Web (abreviadamente www o simplemente web) es simplemente un conjunto de documentos (archivos) distribuidos a lo largo de todo el mundo y con enlaces entre ellos.

Los documentos se visualizan con palabras, frases o imágenes resaltadas(en negrita o en un color determinado) debajo de las cuales se ocultan punteros a otros documentos ubicados en el propio servidor u otro. Seleccionando y activando con el ratón un elemento con puntero, automáticamente se hace una llamada al documento apuntado.

Un puntero a otro documento se denomina hiperenlace y un texto con hiperenlaces se denomina hipertexto. Un hipertexto disponible en la www se denomina página

La primera página que aparece cuando abrimos el navegador se denomina página principal.

1.5.5.1 Direccionamiento en la web (URL)

Uno de los conceptos fundamentales de la web es la forma de localizar documentos. Esto se realiza utilizando un localizador uniforme de recursos (URL), que puede referenciar cualquier tipo de documento en la red. Un localizador URL se compone de tres partes, separadas con rayas inclinadas:

- Protocolo-de-recuperación://computador/ruta-y-nombre-del-archivo

Existen distintos protocolos de recuperación de documentos. El más conocido es http, pero también destacamos FTP, SMTP y TELNET. Computador es el nombre DNS del equipo, y finalmente el nombre completo del archivo que contiene la página buscada.

Ejemplo 1: Una dirección de una página web es:

http://www.ub.edu/homeub

que significa lo siguiente:

- http el protocolo de transferencia usado es http
- www la página forma parte de la World wide web
- ub.edu la página pertenece a la Universitat de Barcelona
- homeub documento solicitado. Página principal de la UB

Los hiperenlaces son sencillamente localizadores URL ya que estos determinan exactamente el nombre del archivo y el computador donde se encuentra la página así como el protocolo utilizado.

Ejemplo 2: La dirección del *Departament d'Electrònica* de la *Universitat de Barcelona* es:

http://el.ub.es

Se utiliza de nuevo el protocolo http. El nombre del equipo es el.ub.es, y como no indicamos ningún nombre de archivo, se accederá a la página principal del *Departament*.

Por otro lado, la dirección URL del servidor web del *Departament d'Electrònica* es

http://www.el.ub.es

por lo que el computador donde se encuentra es www.el.ub.es (www suele ser un alias del nombre del ordenador).

1.5.5.2 Protocolo http

Uno de los puntos clave en la expansión de la www es el protocolo http (protocolo de transferencia de hipertextos o Hipertext Transfer Protocol). http sigue el modelo cliente servidor, usado por lo general entre un navegador web y un servidor web. http establece como recuperar hiperdocumentos distribuidos y enlazados a través de la web.

El protocolo establece para el cliente tres posibles paquetes:

- Solicitar datos del servidor: El objetivo fundamental
- Solicitar la recepción de información sobre características del servidor
- Enviar información al servidor.

El servidor tiene definido un único paquete de respuesta que se limita a enviar la información de cabeceras o el archivo solicitado.

Cuando un usuario hace una petición de acceso a un determinado documento de una dirección dada, el http solicita una conexión TCP, con la que se transfiere al servidor la petición http, en la que se incluye la URL, información sobre el cliente y la orden concreta con sus parámetros asociados. El servidor responde realizando la operación requerida y enviando una respuesta http, que contiene información sobre el servidor y de control, y la respuesta propiamente dicha a la solicitud. Una vez dada la respuesta se cierra la conexión TCP.

1.5.5.3 Navegadores web

Un navegador (browser en inglés) es un programa que permite captar, interpretar y visualizar documentos web. La captación de dichos documentos se realiza con ayuda de uno de los protocolos vistos con anterioridad y que permiten la obtención de archivos a través de Internet: http, FTP, Telnet, etc.

Los documentos web pueden clasificarse en tres tipos: estáticos, dinámicos y activos.

- Documentos estáticos: Son documentos con información fija. El servidor web contiene el documento en lenguaje HTML y se lo envía al cliente a petición de éste. El cliente procesa la información que genera el documento que será visualizado.
- Documentos dinámicos: Son documentos cuya información cambia constantemente de un momento a otro. Un ejemplo es la página web que muestra los resultados de la jornada de futbol en tiempo real, o la hora y la temperatura de una determinada localidad. Cada vez que un usuario solicita una de estas páginas, se lanza en el servidor la ejecución de un programa que genera el documento a transmitir o actualiza sus partes dinámicas. Una vez generada la página se transmite al cliente. Para ello se utiliza un estándar denominado CGI o Interfaz Común de Pasarela. Los programas CGI generadores de documentos dinámicos pueden ser escritos en lenguajes orientados a objetos, tales como Perl, Phyton, C++ u otros.
- Documentos activos: Son documentos que también se generan en el momento de visualizarse, pero en lugar de hacerlo en el servidor, se crean en el propio cliente. En este caso, el servidor remite el programa y los datos, el cliente ejecuta el programa que genera la página y los cambios se van actualizando por el propio programa. Los documentos activos se suelen programar en Java. Los compiladores de Java generan código máquina para un procesador virtual conocido como JVM, de forma que ejecutar este código en una máquina concreta simplemente requiere que se disponga de un emulador de la JVM. La mayoría de los navegadores disponen de un emulador de JVM.

2 Programación

2.1 Introducción

El objetivo de este curso es una introducción a los diferentes aspectos de la programación a través del lenguaje Java. En este sentido, Java para nosotros es una herramienta que nos facilitará el proceso de aprendizaje, pero nuestro objetivo es aprender a programar correctamente.

La pregunta ¿qué es programar correctamente? No tiene una respuesta única, pero quizás un ejemplo ayude. Si resolvemos un problema en un sobre de correspondencia, con cuatro números, es muy probable que cuando vea alguien ese sobre otra persona dentro de un mes no sepa lo que hay escrito y lo tire a la basura.

Lo que se pretende en este asignatura es que el programa escrito hoy lo lea otra persona dentro de un mes y entienda qué hace el código con una lectura rápida. Además ese código debe ser eficiente, y fácilmente reutilizable.

La elección de Java se debe a que es un lenguaje multiplataforma, y que por lo tanto se puede utilizar en un ordenador, en un gran servidor o en un dispositivo embebido. En la Figura 18 se muestran los tres casos.

Figura 18. Plataformas donde corre Java. De izquierda a derecha, ordenador, servidor y dispositivo embebido.

A continuación vamos a analizar qué es un programa, una plataforma, cómo es el proceso de programación y por último que es un entorno de desarrollo.

2.1.1 ¿Qué es un programa?

Descrito de forma simple, un programa es un algoritmo que da una salida en función de unas entradas. Todo programa se puede reducir a tres elementos:

- Interfaces (entradas, salidas)
- Lógica
- Memoria

En la Figura 19 se muestra un diagrama genérico de una aplicación.

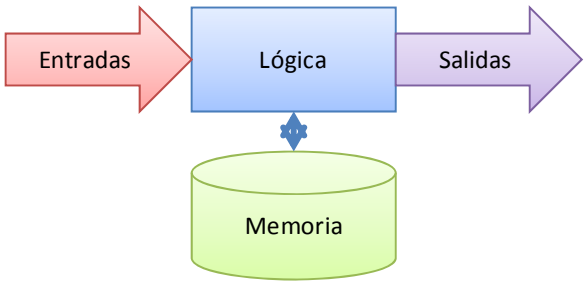


Figura 19. Diagrama de bloques de una aplicación.

Vamos a ver más en detalle estos elementos.

2.1.1.1 Interfaces

Si un programa no interacciona con el exterior, no sirve para nada. Es como tener la máquina más potente, metida en un edificio cerrado, puede ser muy interesante lo que hace, pero como no podemos ver los resultados, ¿de que nos sirve?

Las principales interfaces con el exterior son:

- Teclado: Es un conjunto de botones que simulan las teclas de una máquina de escribir
- Pantalla: Es una matriz de puntos luminosos que permiten crear una imagen
- Ratón: Es un periférico que permite seguir el movimiento de la mano sobre una superficies
- Indicadores luminosos: Son luces que tiene el ordenador para indicar estados (ej. encendido)

En la Figura 20 se muestran fotografías de algunas de ellas.

Figura 20. Principales interfaces de entrada y salida (ratón, teclado y pantalla).

Estas interfaces las suele gestionar el sistema operativo, por lo que generalmente su uso suele ser sencillo, y las diferencias entre ellos trasparentes al usuario.

2.1.1.2 Lógica

La lógica establece el conjunto de reglas que debe seguir el programa. Si tomamos el caso de una calculadora, nuestras acciones son:

- Introducir primer operando
- Introducir operación
- Introducir segundo operando
- Introducir tecla de igual (=)
- Esperar resultado

Desde el punto de vista del programa el proceso sería distinto. El diagrama de estados se muestra en la Figura 21.

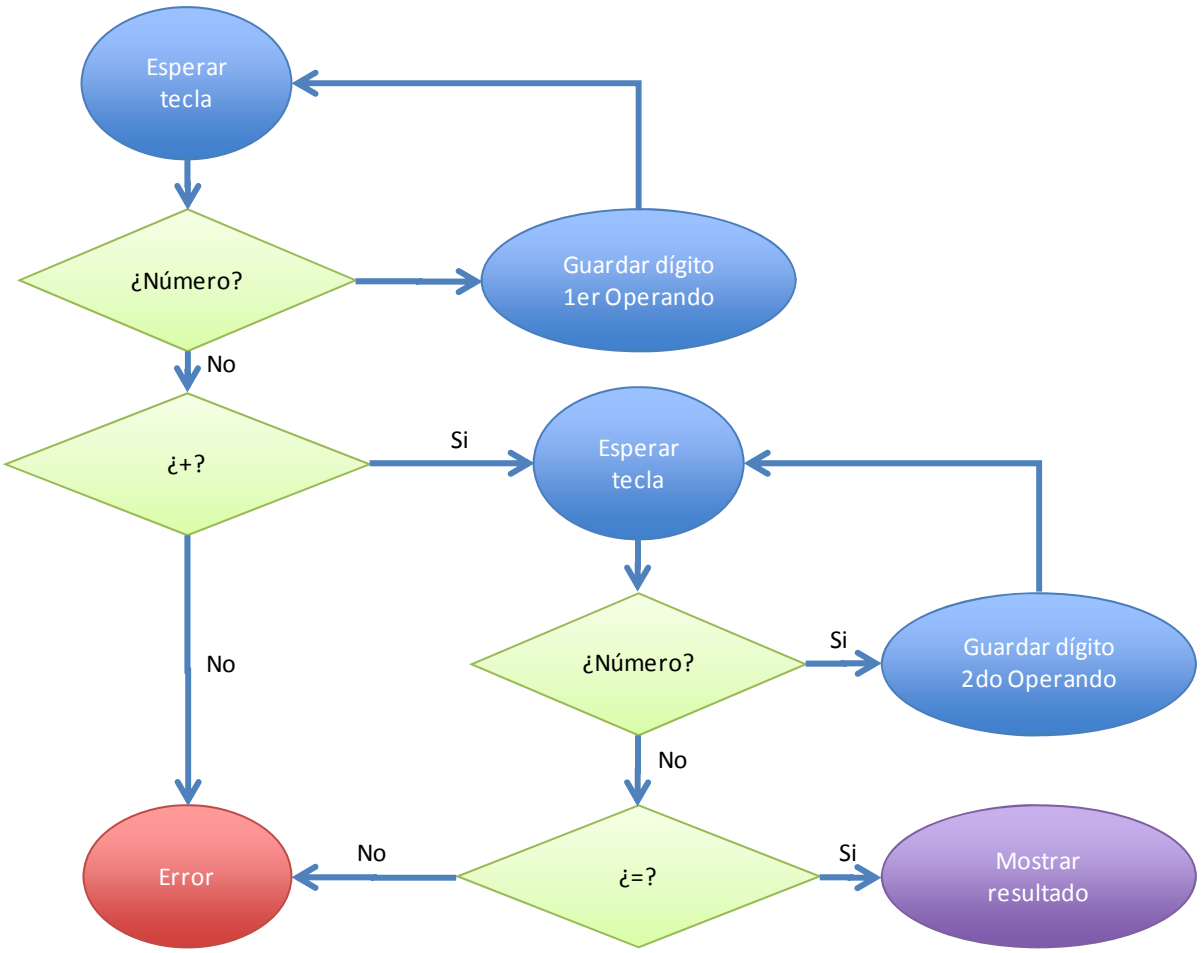


Figura 21. Diagrama de estados de una calculadora para una operación suma.

La diferencia fundamental es que mientras para el usuario la introducción de un dato es un proceso activo; para el programa es una espera (espera de tecla) y viceversa, cuando el usuario espera un resultado es porque el programa está realizando una acción (mostrar resultado).

También es importante indicar que la calculadora sabe que se acaba el primer operando porque introducimos el operador (+) y el segundo al introducir el igual (=). Si no se sigue esta secuencia, aparece un estado de error, que no existía desde el punto de vista del usuario.

Por último mencionar los estados de guardar, que se relaciona con el último elemento de un programa, la memoria.

2.1.1.3 Memoria

La memoria es el elemento que permite tener la información almacenada durante un periodo de tiempo. Para un programa la memoria se puede dividir en dos tipos:

- Volátil
- No-volátil

Dentro de la primera consideraremos aquella información que se utiliza para el funcionamiento del programa y que podemos perder cuando éste deja de funcionar. La no-volátil es aquella que deseamos tener disponible cuando volvamos a poner en marcha el programa.

Un ejemplo de la primera son los operandos de nuestra calculadora, que en principio no será necesario guardarlos. Sin embargo, si nuestro programa es para hacer gráficas, es muy probable que nos interese guardar tanto los datos como las gráficas.

La memoria no-volátil suelen ser ficheros en el disco duro, aunque también pueden ser datos en una base de datos. Este es el caso de sitios como la Wikipedia.

2.1.2 El proceso de programación

El proceso de programación es el camino que lleva de una idea a un programa que funciona de forma correcta y eficiente. Para ello el primer paso es escribir unos requerimientos que ayudan a concretar esa idea.

A modo de ejemplo yo puedo tener la idea de hacer un programa que sea una calculadora. Pero la calculadora puede ser muy simple, que solo realice las cuatro reglas y con números enteros, o yo puedo querer una calculadora científica que permita incluso integrar. Aunque la idea de base es la misma, los requerimientos son mucho más exigentes en el segundo caso que en el primero.

El siguiente paso es hacer el programa que cumpla estos requerimientos. El proceso de programación ha ido evolucionando de la mano de las mejoras en las plataformas.

Inicialmente, la programación se hacía a base de papel, lápiz y paciencia. La información se introducía en el lenguaje propio del microprocesador, el código máquina. Si había un error, en muchos casos se tenía que volver a escribir todo el código de nuevo.

Sin embargo, a medida que los equipos informáticos evolucionaban, se fueron creando programas que ayudaban a la creación de programas. Los primeros fueron los ensambladores, que permitían escribir los programas con letras y números, de forma que ya no había que saberse el identificador de cada instrucción, si no que se ponía una palabra clave (nemónico). Después el ensamblador convertía directamente las palabras clave a código máquina. En este caso, los programas escritos para una determinada plataforma no se podían utilizar en otra.

En paralelo aparecieron herramientas denominada debugadores que permitían ejecutar el código instrucción a instrucción para poder corregir posibles errores o ineficiencias. La unión de los ensambladores y debugadores simplificó mucho el proceso de la programación.

Del ensamblador se paso a los lenguajes de alto nivel como el C, el Pascal, que permitían escribir software sin tener que saber ni código máquina, ni ensamblador. Esto separaba la programación de la máquina que tenía debajo. Para poder ejecutar estos programas el primer paso era compilar, lo cual convertía el código escrito en el lenguaje de alto nivel a instrucciones del propio microprocesador. Una vez hecho, el siguiente paso era unir nuestro código con unas librerías que nos permitían independizar algunos aspectos de la plataforma que utilizáramos (lectura del teclado, escritura de texto en pantalla, lectura y escritura de ficheros).

Sin embargo, había aspectos en que las diferencias se mantenían, los principales eran el acceso a periféricos (ratón, cámaras, impresora) y la programación gráfica. La aparición de estos lenguajes requirió también de herramientas de desarrollo más elaboradas y de debugadores más inteligentes, al tener que trabajar en base al lenguaje de alto nivel, y no a la instrucción que realmente ejecutaba el programa.

En este punto aparece Java, que es un lenguaje independiente de plataforma. Para ello hace uso de la máquina virtual de Java (Java Virtual Machine, JVM). Esta máquina es un microprocesador virtual, ya que es un programa que corre sobre el microprocesador real de nuestra plataforma. Con ello se consiguen tres objetivos:

- El código funcionará igual si tenemos una máquina virtual de la misma versión en la nueva plataforma
- Si hay un error en el código, la máquina virtual puede evitar que afecte al resto de programas corriendo sobre la plataforma
- No hace falta reescribir el código cada vez que se mejoran las plataformas, simplemente es cuestión de modificar la máquina virtual, y todo el resto de código seguirá funcionando correctamente.

Evidentemente esto conlleva ciertos sobrecostes debido a la necesidad de correr la máquina virtual, pero si el programa se ejecuta de forma continuada, al final éste no es tan alto.

Por último indicar un aspecto curioso, y es que las herramientas creadas en una generación permiten crear las de la generación siguiente. Dicho de otra forma, los primeros ensambladores se escribieron en código máquina y los primeros compiladores en ensamblador. Pasado este punto, pasan a escribirse en la herramienta más cómoda, a modo de ejemplo, hoy en día los ensambladores se escriben en C. Cuando un lenguaje permite crear sus propias herramientas se dice que ya es maduro.

2.1.3 ¿Qué es una Plataforma?

En pocas palabras, es la electrónica que permite correr un programa. Esta electrónica funciona en base a impulsos eléctricos. Para hacer la electrónica más sencilla y robusta, esta utiliza dos niveles de voltaje. Usualmente el bajo equivale a un 0 y el alto a un 1. Este tipo de codificación se denomina binaria, al utilizar sólo dos valores. Los dígitos binarios se denominan bits (BInary digIT). Para alcanzar valores más altos simplemente tenemos que poner un dígito binario al lado de otro de la misma forma que cuando escribimos un número decimal.

Si tenemos dos dígitos decimales podemos codificar cien símbolos (00-99). En el caso de una codificación binaria esta se reduce bastante:

- Símbolo cero: 00
- Símbolo uno: 01
- Símbolo dos: 10
- Símbolo tres: 11

Podemos ver de forma sencilla que el número de símbolos que podemos codificar (S) se relaciona con el número de valores que codifica cada dígito (M) y el número total de dígitos a partir de la fórmula:

$$S = M^N$$

En el caso de dos dígitos decimales, el número de símbolos sería:

$$S = 10^2 = 100$$

Para el caso de dos dígitos binarios, pasa a ser:

$$S = 2^2 = 4$$

De esta forma, podemos codificar de forma binaria cualquier conjunto de símbolos, solo es necesario agrupar el número necesario de dígitos. La agrupación mínima suelen ser 8, conocida como Byte. Estos símbolos se reciben a través de las interfaces de entrada, como es el caso de las letras del teclado, o la posición del ratón. Se pueden mostrar al usuario a través de la pantalla, iluminando de forma adecuada los píxeles de la misma. O bien se pueden transferir a través de un cable, como en el caso del ADSL.

Haciendo uso de la lógica de Boole es posible asociar una algorítmica a los diferentes símbolos. Para ello se utilizan los operadores básicos de esta lógica:

- Y ($A \mid B$)
- O ($A \cup B$)
- No ($\bar{A}$)

Su comportamiento se puede describir en base a tablas de verdad:

$A \mid B$	0	1
0	0	0
1	0	1

$A \cup B$	0	1
0	0	1
1	1	1

A	$\bar{A}$
0	1
1	0

Tabla 4. Tabla de verdad de los operadores booleanos.

Cuando estos operadores se implementan en una electrónica, se denominan puertas lógica. Estas puertas, adecuadamente organizadas, permiten realizar procesos simples (comparar dos bits), o complejos (sumar millones de números por segundo).

La memoria, permite guardar la información binaria. Esta puede ser desde un chip, un discos duro, un CD, un DVD, Blu-Ray… En general, todas las plataformas tienen como mínimo dos tipos de memoria, una volátil, que permite guardar los estados del programa, y otra no-volátil, donde se guardan los programas.

Estos programas corren sobre los procesadores, que son puertas lógicas organizadas de forma que pueden correr programas. Estos tienen una serie de instrucciones que permiten comparar, sumar, acceder a la memoria… Estos procesadores suelen pertenecer a una familia, las cuales se diferencian en qué instrucciones disponen, y como las codifican en binario. Las familias se denominan arquitecturas, y algunas comunes son:

- Intel x86: Es la que se utiliza primordialmente en los ordenadores personales, aunque se utiliza en muchos dispositivos electrónicos y servidores.
- PowerPC: La utilizan las PlayStation 3, y algunos servidores.
- Sparc: Se utiliza primordialmente en servidores.
- Arm: Es la plataforma más utilizada en teléfonos móviles y PDAs.

En resumen, las plataformas se pueden dividir en tres elementos que son los homólogos a nivel electrónico de los presentes en un programa:

- Interfaces (entradas, salidas)
- Procesador
- Memoria

Si las plataformas son sencillas, el código corre directamente sobre la CPU, y todo el control de las interfaces de entrada y salida, así como la gestión de la memoria la realiza directamente nuestro programa. O en el caso de utilizar una máquina virtual, la propia máquina. En este caso se dice que el programa es embebido.

Si la plataforma es más compleja, como un ordenador, la gran cantidad de periféricos e interfaces disponibles hace que sea muy difícil que un solo programador pueda gestionarlos todos. Por este motivo, entran en juego los sistemas operativos que introducen un nivel de abstracción, separando al programador de la electrónica. Los diferentes periféricos se abstraen, de forma que todos los teclados se comportan de la misma forma desde el punto de vista del programador, lo mismo las pantallas, los ratones o los discos.

De esta forma, dos ordenadores con la misma arquitectura y sistema operativo se comportan de forma equivalente, independientemente de los periféricos que utilicen. Evidentemente, las prestaciones pueden variar, pero para el programador son iguales. Los Sistemas Operativos más comunes son:

- Windows: Desarrollado por Microsoft, el código es propietario.
- Linux: Proyecto de código abierto, desarrollado por personas particulares y empresas.
- MacOS: Desarrollado por Apple, tiene elementos de código abierto y otros propietario.

Sin embargo, un programa generalmente sólo sirve para una arquitectura y un sistema operativo concreto. Hay soluciones para emular arquitecturas y sistemas operativos, pero no suelen ser óptimas.

En el caso del Java, este nivel de abstracción se incrementa al trabajar sobre un procesador virtual, el cual está adaptado a la arquitectura correspondiente. De esta forma el programador trabaja de forma independiente del hardware. Si la máquina corre sobre una plataforma con sistema operativo, ésta se comporta como un programa más. Y por lo tanto aprovecha la abstracción de hardware que el SO proporciona. Además, la máquina virtual adapta el código a la plataforma, de forma que se mejoran sus prestaciones, habiendo casos donde corre más rápido el código en Java que el específico para la plataforma.

2.1.4 Entorno de desarrollo

Un entorno de desarrollo es un conjunto de herramientas de software que permiten realizar un programa. Suele estar compuesto por:

- Compilador: permite pasar el código de un lenguaje a código máquina.
- Librerías: proporciona elementos básicos de programación que facilitan la creación de software, y permiten al usuario abstraerse de la plataforma.
- Debugador: programa que permite ir viendo la evolución de un programa paso a paso, y determinar en que puntos hay errores (Bugs) e ineficiencias.

Todo ello puede estar integrado dentro de un Entorno de Desarrollo Integrado (Integrated Development Environment, IDE). En nuestro caso, haremos uso de un entrono de desarrollo, el Java Development Kit versión 1.6, y de un IDE de Java que se llama NetBeans.

El JDK contiene un compilador de Java, la máquina virtual y las librerías. El IDE además, proporciona un entorno para escribir y debugar programas.

2.1.5 Conclusión

En esta introducción hemos visto como es un programa y qué elementos lo conforman así como la evolución del proceso de programación. Por último hemos visto los elementos que forman parte de un entorno de desarrollo y el que utilizaremos en nuestro caso.

2.2 Primeros programas

Vamos a hacer nuestro primer programa en Java sobre el entorno de desarrollo NetBeans. Para ello, el primer paso es bajar de Internet el programa NetBeans. Este se puede encontrar en http://www.netbeans.org. Es importante indicar que si no tenemos instalado el JDK en nuestro ordenador, existe una versión que integra ambos elementos, lo cual nos simplificará el proceso de instalación.

El siguiente paso es instalar el software en nuestro ordenador. Para ello seguiremos las instrucciones que nos indican en el propio sitio web. En el caso de Windows será ejecutar el programa bajado. Una vez instalado podemos ejecutarlo, y nos encontraremos una ventana semejante a la de la Figura 22.

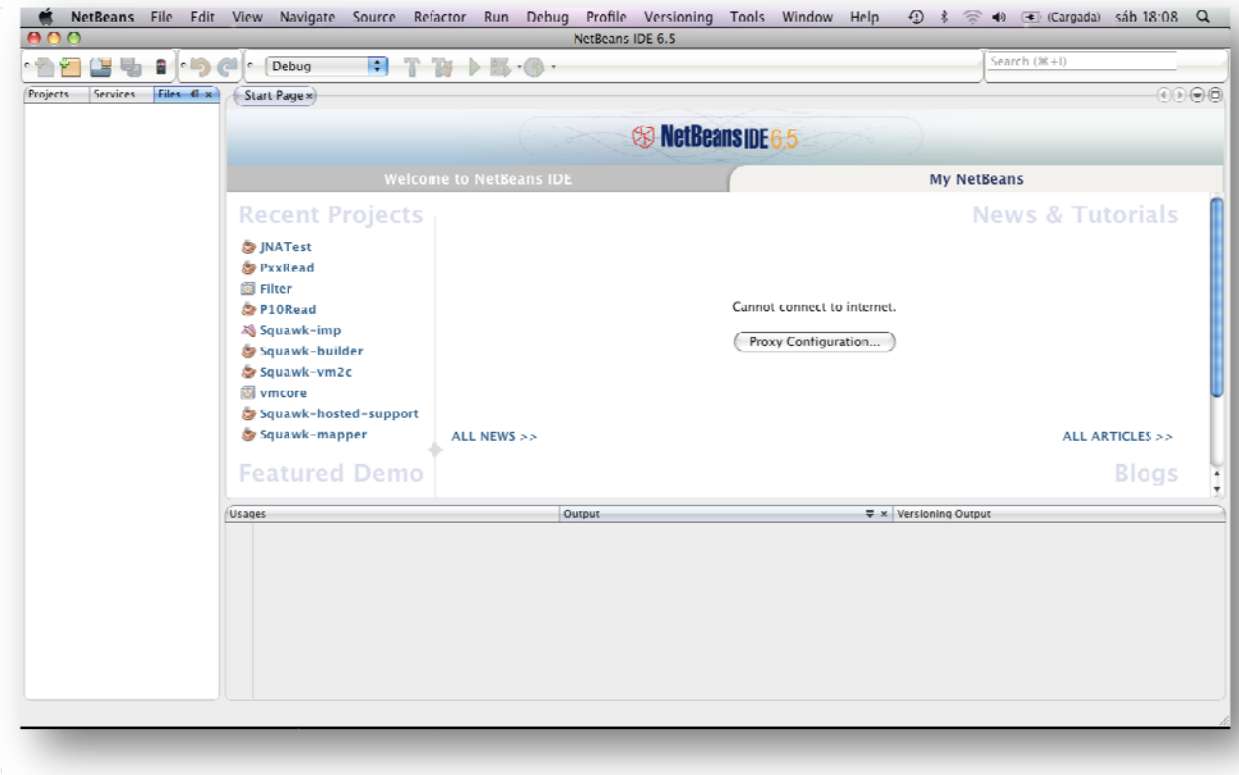


Figura 22. Entorno NetBeans.

Vemos que esta dividida en varios marcos. A lo largo de este apartado veremos su uso.

2.2.1 Proyecto Hola Java

El primer paso para poder realizar un programa es crear el proyecto asociado. Para ello iremos a la opción de menú:

File->New Project. Esto hará aparecer la ventana que se muestra en la Figura 23.

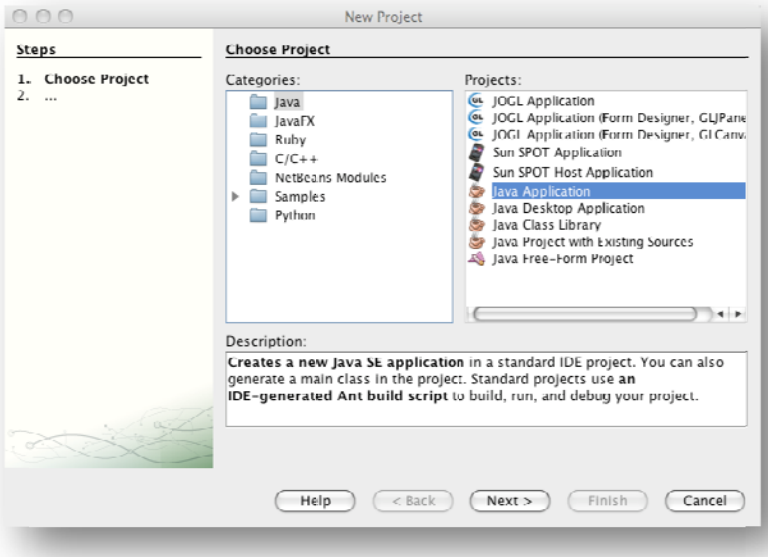


Figura 23. Ventana para escoger el tipo de proyecto.

En la ventana se pueden ver cuatro zonas diferenciadas, A la izquierda se muestra el paso en que nos encontramos, en este caso en el de escoger el tipo de proyecto (Choose Project). A la derecha arriba encontramos dos selectores, el primero son las categorías de proyectos, y el segundo el proyecto concreto. Los elementos de ambos pueden variar en función de la versión de NetBeans descargada. Debajo se encuentra un ventana donde se describe el tipo de proyecto que se va a crear. En nuestro caso escogeremos la categoría “Java” y el tipo de proyecto “Java Application”, y pulsar la tecla (next>).

Figura 24. Ventana para dar nombre y ubicación al proyecto.

Nos aparece la ventana donde hemos de decir el nombre del proyecto y su ubicación. En nuestro caso, en Project Name ponemos HolaJava, el resto lo podemos dejar por defecto, y apretamos (Finish).

Nos aparece en la ventana de edición la clase Main, con una plantilla de programa:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package holajava;

/**
 *
 * @author chema
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }
}
```

Ahora nos centraremos en realizar el TODO, introduciendo la lógica de la aplicación. En nuestro caso, substituiremos el TODO por:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package holajava;

/**
 *
 * @author chema
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Hola Java");
    }
}
```

Una vez hecho podemos correr la aplicación apretamos el botón  en la ventana de Output nos aparece el texto:

```
run:
Hola Java
BUILD SUCCESSFUL (total time: 0 seconds)
```

Vemos que aparece nuestro texto en la ventana. Podemos modificar el texto que hay entre las comillas y ver que al ejecutarlo nos aparece en la ventana.

2.2.2 Debugar

También podemos ejecutar el programa paso a paso. Para ello introducimos un punto de corte (breakpoint) en el programa, esto lo hacemos pulsando dos veces encima de la tira gris que hay justo al lado del texto. Nos aparecerá un cuadrado rojo, y la línea de programa se pondrá del mismo color tal como se muestra en la Figura 25.

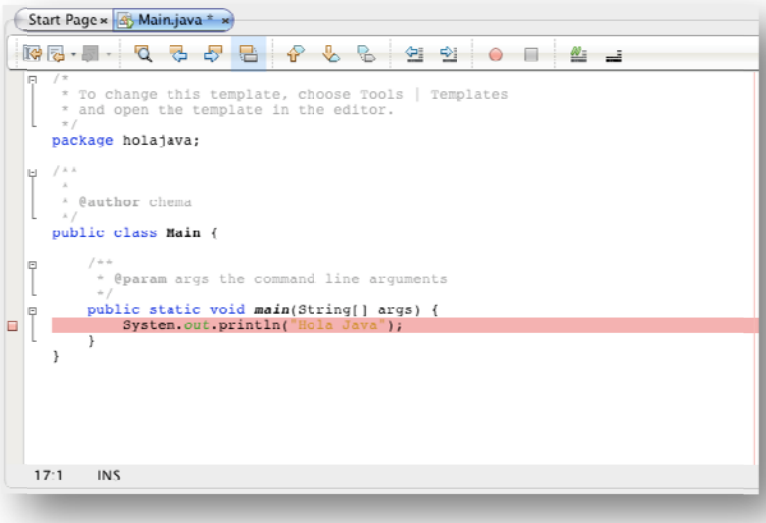


Figura 25. Pantalla de debugado con punto de corte.

Apretando el botón  se inicia el programa, quedando parado en el punto de corte. La línea donde ha parado se muestra en color verde. Apretando el botón  salta a la siguiente línea de programa, y en la pestaña "HolaJava (debug)" dentro de la pestaña Output aparece el texto.

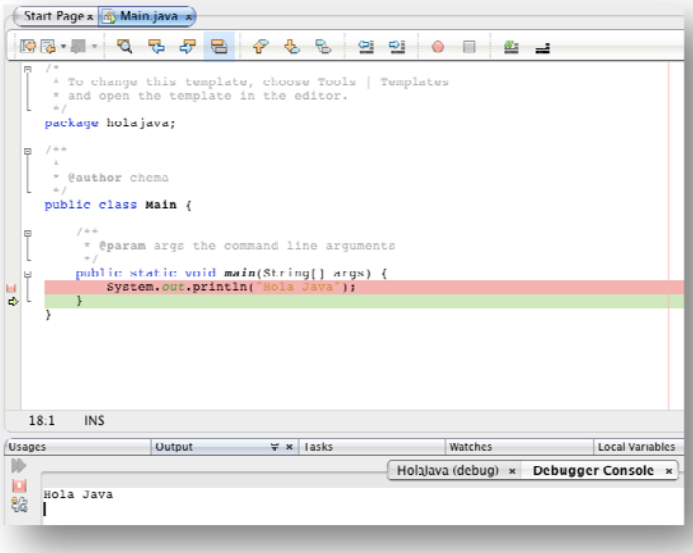


Figura 26. En la parte superior, ventana de debugado con el punto de corte y la línea actual. En la parte inferior, salida por consola.

Si volvemos a dar al botón de salto, el programa acaba, cerrándose la sesión de debugado.

2.2.3 Proyecto Hola Java GUI

Vamos a hacer la aplicación, pero de forma más gráfica. Para ello crearemos un nuevo proyecto y escogeremos el tipo “Java Desktop Application”. Le pondremos por nombre HolaJavaGUI. El entorno se modifica y aparece el editor gráfico:

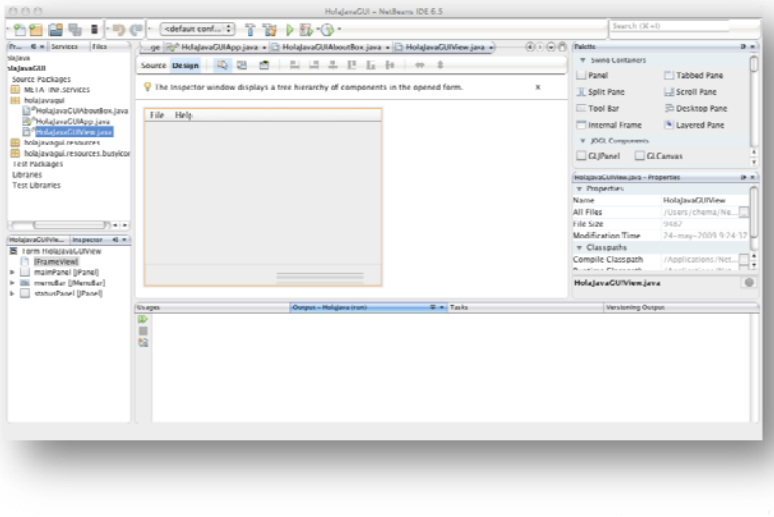


Figura 27. Entorno de programación gráfica.

En el centro podemos ver una plantilla de la ventana de la aplicación. Incluye un menú, y debajo una barra de estado. A la derecha se encuentra la paleta de componentes. Buscamos el grupo Swing Controls y pulsamos sobre Label. Al desplazarnos sobre la ventana de la aplicación, nos aparece el control, con el nombre “jLabel1”. Lo soltamos en el lugar que mejor nos parezca.

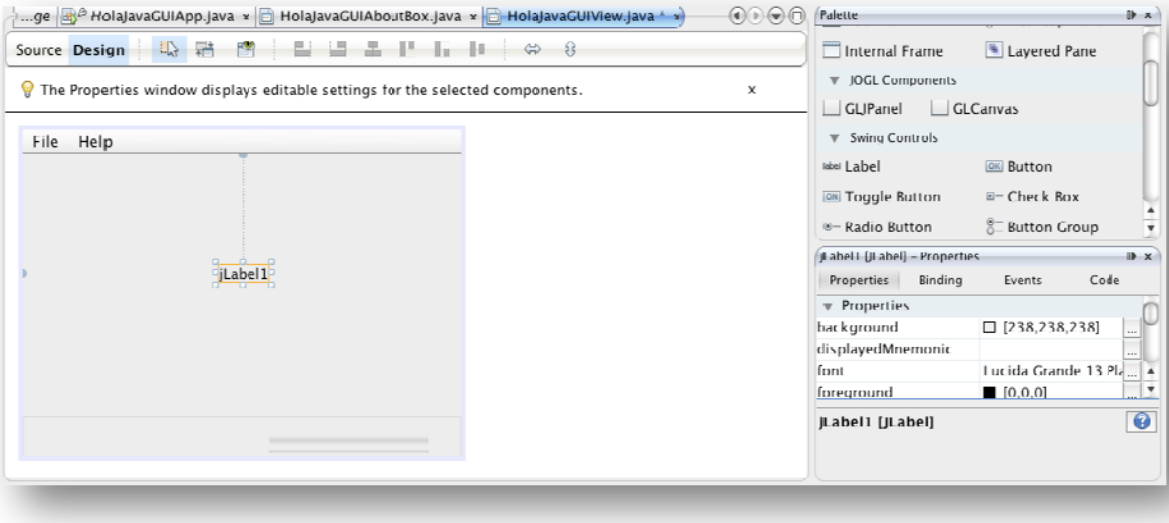


Figura 28. Diseño con el jLabel1 incluido.

Si pulsamos dos veces sobre jLabel1, podemos editar su contenido y poner el texto que deseemos. En nuestro caso es “Hola Java GUI”. Si pulsamos la tecla de ejecutar, nos aparece la ventana de la Figura 29.

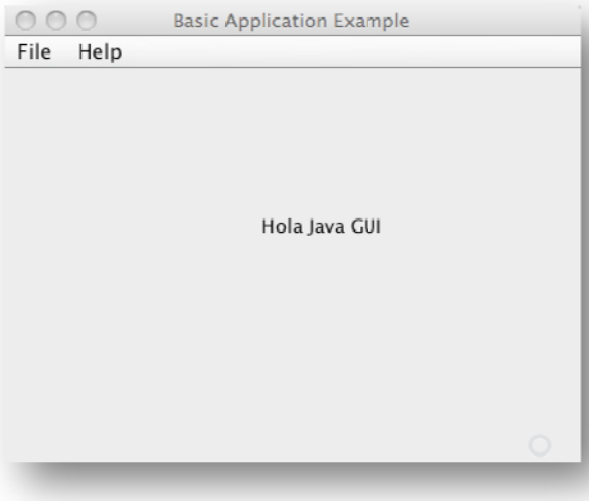


Figura 29. Ventana de Hola Java GUI.

En este caso no hemos escrito ni una línea de código. Aunque para hacer un programa interesante tendremos que darle una lógica.

2.2.4 Conclusiones

En este capítulo hemos aprendido a instalar el entorno de desarrollo NetBeans. Hemos creado dos proyectos, el primero para consola, y el segundo para ventana.

2.3 Estructuras básicas

Hemos realizado nuestros primeros programas, pero para poder continuar necesitamos conocer como se implementan los conceptos básicos de programación en Java, como los tipos de datos, los cambios de flujo o los bucles. Los programas de este apartado no están basado en GUI, ya que se precisan aún algunos conocimientos más para poder hacerlo.

2.3.1 Un programa básico

Vamos a volver sobre el proyecto HolaJava. Este programa presenta un mensaje en la consola. Contiene todas las estructuras básicas de Java, que se repetirán en todos los programas. Es importante tener en cuenta que Java tiene en cuenta las mayúsculas y minúsculas.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package holajava;

/**
 *
 * @author chema
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Hola Java");
    }
}
```

Encontramos los siguientes elementos:

/* texto */	El compilador ignora cualquier texto que se encuentre entre /* y */
package holajava	Los paquetes permiten organizar las clases de forma jerárquica.
/ texto */**	El compilador ignora este texto, pero el javadoc lo utiliza para generar la documentación del programa.
@author	Indica quien es el autor del programa
public	Es un modificador de acceso, en este caso indica que es accesible publico, o dicho de otra forma, que no hay restricciones.
class {… }	Todos los elementos de Java deben encontrarse definidos dentro una clase. En este caso dentro de la clase Main.
@param	En este caso, la documentación indica los parámetros del método siguiente.

`public static void main()`	Todos los programas Java empiezan con una llamada al método main, y este tiene que estar definido de esta forma.
`String args[]`	Argumentos para el programa en este caso no se utilizan.
`{ … }`	Las llaves marcan el principio y el final del cuerpo del método.
`System.out.println();`	Utilizamos el objeto System.out para imprimir el mensaje con el método println(). Todas las sentencias en Java deben acabar en ";".
`"Hola Java"`	Cadena a presentar en la consola.

Vamos a ir analizando todos los conceptos a lo largo de este documento.

2.3.2 Comentarios

Los permiten describir el comportamiento del programa. Y en el caso de Java, se pueden aprovechar para elaborar de forma automática la documentación. Existen tres tipos de comentarios:

`/* texto */`	El compilador ignora el contenido entre /* y */
`/** texto */`	Es parte de la documentación de la clase. El compilador ignora de la misma forma el contenido, pero la aplicación javadoc lo tiene en cuenta a la hora de generar la documentación asociada.
`// texto`	El compilador ignora el texto entre // y el final de línea.

Es importante indicar que los comentarios no ocupan espacio en el programa compilado, por lo que pueden, y deben, ser tan extensos como sea necesario para que el código sea fácil de entender.

2.3.3 Objetos

2.3.3.1 Programación orientada a objetos

Java, entre otros lenguajes modernos, es un lenguaje orientado a objetos; todos los elementos de Java están integrados en algún tipo de objeto. La programación orientada a objetos es un paradigma de programación que, como su nombre indica, hace uso de los objetos. Éstos son tipos abstractos de datos que encapsulan (ocultan) los datos y funciones necesarias para su funcionamiento, ofreciendo al "mundo exterior" sólo acceso controlado a los mismos. En otras palabras, la programación orientada a objetos tiene como base la realización de código modular y encapsulado con el objetivo de hacer que la tarea de desarrollar grandes sistemas de *software* sea más efectiva y menos propensa a errores. En este curso introductorio el nivel de programación que realizaremos no requerirá apenas el uso de los principios y posibilidades de la programación orientada a objetos, aunque proporcionaremos una introducción a sus conceptos básicos.

2.3.3.2 Objetos

Usualmente un objeto en un lenguaje de programación se diseña para modelar un objeto o proceso real. A modo de ejemplo, un jarrón es un objeto (real), que tiene atributos como el color, la forma y la posición. También puede estar en diferentes estados (lleno o vacio), y para ello se tiene que seguir un método de llenado y otro de vaciado. Si tuviésemos la necesidad de realizar un programa que manejase datos sobre jarrones podríamos definir un objeto (virtual) que los modelase incluyendo datos y funciones que representasen las características del jarrón real (color, forma, posición) sus estados (lleno, vacio) y las operaciones que pueden realizarse con él (llenado y vaciado).

El diseño de los objetos para construir un sistema de software es parecido al bricolaje; debemos definir las piezas que necesitamos, sus características y funciones y como se relacionan entre ellas. Imaginemos por ejemplo que queremos montar una mesa. Lo básico es tener un tablero y cuatro patas. El tablero y las patas pueden tener

atributos como el material y las dimensiones. También tienen que unirse el tablero con las patas. Esto se puede hacer con una rosca directa de las patas con el tablero, para lo cual el tablero tendrá que tener la hembra de una rosca y las patas el macho. Además tendremos que llevar a cabo la acción de enroscar. Esta acción introduce el concepto de estado, enroscado o no enroscado. La robustez es función del estado.

O también podemos poner tornillos, para lo cual hemos de tener agujeros en el tablero para ponerlos así como un destornillador que nos permita realizar la acción de atornillar. Los propios tornillos y el destornillador pueden tener atributos en función del material y el tipo de cabeza.

Los diferentes elementos que hemos mencionado corresponderían a los objetos, mientras que los atributos y estados corresponderían a los campos que contiene el objeto. Además se crea una jerarquía de objetos basada en las relaciones entre ellos: la mesa contiene (está compuesta de) el tablero y las patas. Por último el cajón puede contener otros objetos, pero que se limita a almacenar. Este sería el caso de un objeto contenedor, que guarda información, pero que dicha información no influye en el comportamiento del objeto.

2.3.3.3 Clases

Siguiendo con el ejemplo del mueble, los diferentes cajones de la cajonera se basan en un diseño común que se describe en unos planos. En nuestro caso, este diseño común de todos los objetos del mismo tipo se define construyendo una **clase** en Java. Cuando a partir de una clase se crea un objeto del tipo dado diremos que se ha creado una instancia de la clase. Por ejemplo, un cajón dado es una instancia de las clase de cajón.

Una clase contiene una conjunto de campos y métodos que permiten actuar sobre los mismos. Una vez creado un objeto basado en una determinada clase, este puede ir cambiando de atributos, estado y contenido en función de las necesidades de la aplicación. Un ejemplo de clase cajón sería:

```
public class Cajon {
    boolean abierto = false;

    void abrir() {
      abierto = true;
    }

    void cerrar() {
      abierto = false;
    }

    boolean isAbierto() {
      return abierto;
    }
}
```

Esta clase no contiene el método *main* ya que no es una aplicación, es sólo una definición para su uso posterior. Una aplicación que hiciera uso de esta clase podría ser:

```
public class CajonDemo {
    public static void main(String[] args) {
      Cajon cajonSuperior = new Cajon();
      Cajon cajonInferior = new Cajon();

      System.out.println("Cajon Superior abierto: " + cajonSuperior.isAbierto());
      System.out.println("Cajon Inferior abierto: " + cajonInferior.isAbierto());

      cajonSuperior.abrir();

      System.out.println("Cajon Superior abierto: " + cajonSuperior.isAbierto());
    }
}
```

Un aspecto importante de las clases es que el código fuente se escribe en un fichero independiente para cada clase de tipo .java. Este fichero se compila, creándose un fichero de tipo .class. A modo de ejemplo, la clase Cajon se guardaría en Cajon.java y su código compilado en Cajon.class.

2.3.3.4 Herencia

De la misma forma que en la vida real el concepto “cajón” corresponde a una gran variedad de tipos específicos, al construir una aplicación podemos tener la necesidad de modelar diferentes tipos de cajones: básicos, con separadores, con archivador para carpetas... En lugar de definir una única clase con todas las características posibles (que serían innecesarias en la mayoría de casos) partimos de la clase cajón general y definimos nuevas clases (más específicas) a partir de ella añadiendo nuevas funcionalidades según nuestras necesidades a través de la denominada **herencia**.

Una clase madre tiene una propiedades básicas (como el cajón), luego se puede crear a partir de ella una clase hija que “hereda” las propiedades de la madre y le añade nuevas, como la posibilidad de poner carpetas o los separadores. La clase madres pasa a ser la superclase, y la hija la subclase.

La subclase hereda los campos y métodos asociados, lo cual permite reutilizar el código. Además puede redefinir los métodos, de forma que estos adquieren nuevas características. En Java las clases solo pueden tener una superclase.

Para definir una clase hija utilizamos la sintaxis:

```
public class Archivador extends Cajon {
    // Los campos y métodos necesarios irían aquí
}
```

En este caso archivador adquiere las propiedades de cajón, y por lo tanto sólo nos tenemos que preocupar de aquellos aspectos que hacen diferente a archivador respecto de cajón.

2.3.3.5 Interfaz

Una interfaz define la forma en que se ve un determinado dispositivo desde fuera o, en otras palabras, define un patrón de como una clase debe interactuar con el mundo exterior. A modo de ejemplo, los equipos reproductores de video y de audio suelen tener una interfaz mínima basada en cinco acciones:

- Reproducir
- Avance rápido
- Retroceso rápido

- Capítulo/Tema siguiente
- Capítulo/Tema anterior

Esto permite que una persona pueda acceder de forma sencilla a la utilización de estos dispositivos. El usuario no tiene que preocuparse de cómo está hecho por dentro el dispositivo ya que todos funcionan de forma equivalente.

En Java, una interfaz contiene la declaración de diferentes métodos. En el caso del cajón, podríamos definir una interfaz guía con los diferentes métodos asociados:

```
interface Guia {
    void abrir();
    void cerrar();
    boolean isAbierto();
}
```

Observemos que en este caso sólo se definen los métodos, y no su contenido, por lo que en este caso no es posible aprovechar la funcionalidad como pasaba cuando se heredaba una clase. Sencillamente las clases que implementen este interfaz deben obligatoriamente implementar los métodos descritos.

La ventaja es que todas las clases que implementan una interfaz aseguran en el momento de la compilación que cumplen con unos determinados requisitos. Ello permite substituir de forma sencilla una clase por otra cuando se necesita una funcionalidad semejante, como por ejemplo escuchar un canción, leyéndola de un CD, o bien de un disco duro en formato mp3.

Para indicar que un clase codifica un determinada interfaz se utiliza implements.

```
class Zapatero implements Guia {
    void abrir() {...}
    void cerrar() {...}
    boolean isAbierto() {...}
}
```

Si una clase implementa una interfaz, esta tiene que definir todos los métodos definidos en la misma.

2.3.3.6 Paquete

Un programa de Java puede hacer uso de unas pocas clases o de miles de estas. Para simplificar su localización y uso, estas se organizan en paquetes. Estos paquetes se estructuran de forma jerárquica y guardan las clases e interfaces que tienen características similares. Los paquetes son desde el punto de vista físico los directorios donde se encuentra las clases e interfaces.

2.3.4 Tipos de datos

Los tipo de datos de un lenguaje de programación son las diferentes formas disponibles para la representación de datos que el lenguaje permite definir i manipular. Distinguiremos entre los llamados **tipos básicos de datos,** que son aquellos definidos por defecto en la sintaxis del lenguaje, y los **tipos abstractos de datos,** aquellos que se definen a posteriori (en las librerías del lenguaje o por el programador) a partir de los primeros.

En Java existen 8 tipos básicos de datos. Cuatro tipos son enteros, dos en coma flotante, uno tipo carácter y otro booleano para valores lógicos. A diferencia de otros lenguajes, todos ellos están unívocamente definidos y son iguales en todas las plataformas. Todos ellos además tienen tamaños múltiplos de bytes, a excepción del '`boolean`'. Por otra parte, en java los tipos abstractos de datos son las clases (discutidas en la sección 2.3.3.3).

2.3.4.1 Enteros

Los tipos enteros como su nombre indica, carecen de parte fraccionaria. Todos ellos son agrupaciones múltiplos de bytes, y codifican el número en formato binario.

$$n=10011_2$$

Donde el subindice 2, nos indica que esta codificado en binario. Para realizar su conversión a decimal, utilizamos la fórmula:

$$N = \sum_{i=0}^{n-1} 2^i a_i$$

Para utilizarla, por ejemplo en el caso del número 10011_2, lo primero que tenemos que hacer es contar el número de bits, para obtener el valor de n, que en este caso es 5. Y después tener en cuenta que el bit menos significativo ocupa la posición 0, y que el bit más significativo se encuentra en la n-1, que en nuestro caso es 4. Por lo tanto, el número se puede cambiar a notación decimal de la siguiente forma:

$$N = 10011_2 = \sum_{i=0}^{5-1} 2^i a_i = 2^4 \cdot 1 + 2^3 \cdot 0 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 16 + 2 + 1 = 19$$

A diferencia de otros lenguajes, en Java los números son siempre con signo, y con la codificación que hemos mostrado, no es posible codificar el signo. Para solucionarlo se utiliza el complementos a dos. Supongamos un número codificado en 8 bits (1 byte). Se utilizan la mitad de los códigos para números positivos y la otra mitad para negativos.

$$S = M^N = 2^8 = 256 \rightarrow \begin{cases} 128 \text{ positivos} \\ 128 \text{ negativos} \end{cases}$$

El 0 se toma como positivo y se codifica con todos los dígitos a 0 (00000000), mientras que el 127 es máximo valor codificable, y se codifica con 0 en el dígito más significativo y 1 en el resto (01111111). Los números negativos se codifican invirtiendo los bits del número y sumándoles 1 de forma binaria. A modo de ejemplo, el -127 y el 0 sería:

$$-y = \bar{y} + 1 \rightarrow \begin{cases} -127 = \overline{127} + 1 = \overline{01111111} + 1 = 10000000 + 1 = 10000001 \\ -0 = \bar{0} + 1 = \overline{00000000} + 1 = 11111111 + 1 = 1'00000000 \end{cases}$$

Podemos observar que el -0 y el 0 se codifican igual, por motivos evidentes., aunque se precisa de un bit más para poder codificar el resultado. Teniendo en cuenta que solo se disponen de los 8 bits para guardar el resultado, el resultado acaba teniendo los 8 bits a 0. Indicar también que en los negativos se codifica un símbolo más que en los positivos al no tener el 0. Dicho de otra forma con 8 bits se codifica en complemento a dos del -128 a 127.

Por último indicar que las sumas y las restas se realizan de la misma forma. Vamos a ver un caso concreto como se calcula la diferencia ente 23 y 45.

$$23 - 45 = 23 + (-45) = 00010111 + (\overline{00101101} + 1) = 00010111 + 11010011 = 11101010$$

Podemos validar el resultado convirtiendo el valor obtenido a positivo y de ahí a decimal.

$$11101010 = -(\overline{11101010} + 1) = -00010110 = -22$$

En el proceso hemos realizado el paso a negativo de un número positivo y viceversa. Se observa que el resultado es correcto. La gran ventaja de esta notación es que no hay que hacer procesos especiales para realizar las sumas de números negativos o las restas. Indicar también que en esta notación, el signo del número lo indica el bit más significativo. Si este bit es 1 nos dice que el número es negativo, mientras que un 0 significa que su valor es positivo.

En la tabla siguiente se muestran los diferentes tipos enteros definidos en Java, así como su tamaño y el rango de números que permite codificar.

Tipo	Espacio	Rango (inclusive)
int	32 bits (4 bytes)	Desde -2.147.483.648 a 2.147.483.647
short	16 bits (2 bytes)	Desde -32.768 a 32.767
long	64 bits (8 bytes)	Desde -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
byte	8 bits (1 byte)	Desde -128 a 127

Tabla 5. Tipos de datos enteros.

Los límites de los diferentes formatos están recogidos en constantes dentro de sus clases asociadas.

Tipo	Clases	Constante valor máximo	Constante valor mínimo
int	Integer	Integer.MAX_VALUE	Integer.MIN_VALUE
short	Short	Short.MAX_VALUE	Short.MIN_VALUE
long	Long	Long.MAX_VALUE	Long.MIN_VALUE
byte	Byte	Byte.MAX_VALUE	Byte.MIN_VALUE

Tabla 6. Clases asociadas a los enteros y constantes de valor máximo y mínimo.

El tipo más utilizado es el int, ya que es el básico de Java. Los formatos byte y short se suelen utilizar en tres casos:

- Cuando se trabaja con información a bajo nivel donde la codificación se realiza bit a bit.
- Si es conveniente indicar el valor máximo del campo para entender el código.
- Para reducir el espacio ocupado en memoria si se trabaja vectores, matrices…

Se definen además los siguientes formatos de literales para asignar los valores a las variables:

- 123 implica notación decimal, o en base 10. Dicho de otra forma, como escribimos normalmente.
- El sufijo L para indicar que el valor es un entero largo, o fuera del rango de int.
- El prefijo 0x para indicar que se trata de un valor en hexadecimal, o escrito en base 16, esto implica que utilizamos los dígitos de 0 a 9 y de A a F (A es 10 y F es 15). Esta notación se utiliza por que permite hacer representaciones de 4 bits en 4 bits (53 es 00110101_2 que agrupándolo de cuatro en cuatro sería 0011 0101 y traducido a hexadecimal 0x35).
- El prefijo 0 indica formato octal, o escrito en base 8, por lo que sólo se utilizan los dígitos de 0 a 7. En este caso las agrupaciones son de 3bit en 3 bits (53 es 00 110 101 que traducido es 065).

En la Tabla 7 podemos ver las conversiones entre las diferentes notaciones.

Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bin	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
Oct	00	01	02	03	04	05	06	07	010	011	012	013	014	015	016	017

Tabla 7. Conversión entre las diferentes notaciones enteras

En general, no se recomienda utilizar la notación octal ya que puede llevar a confusiones.

Ejercicio 5: Escribir el número 23 en binario. Convertirlo a hexadecimal y octal. Ver si el resultado coincide con el del programa:

```
/*
 * Clase para ver diferentes notaciones en Java
 */

package testnotaciones;

/**
 *
 * @author Jose M.
 */
public class Notaciones {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
                int valor = 23;

                System.out.println("Decimal: " + valor);
                System.out.println("Binario: " +
                                   Integer.toBinaryString(valor));
                System.out.println("Hexadecimal: 0x" +
                                   Integer.toHexString(valor));
                System.out.println("Octal: 0" +
                                   Integer.toOctalString(valor));
    }
}
```

Escribir el número -23 en binario. Convertirlo a hexadecimal y octal. ¿Se parece el valor 0x17 obtenido anteriormente?

2.3.4.2 *Coma flotante*

Los tipos en coma flotante se utilizan para valores fraccionarios o reales. En este caso, la notación científica:

$$x = (-3{,}27 \cdot 10^{18})_{10}$$

En este caso, el signo es negativo, la mantisa 3,27 y el exponente +18.

En primer lugar vamos a ver como guardar un número fraccionario en binario. En este caso, también podemos dividir los bits en enteros y fraccionarios (anteriores y posteriores a la coma):

$$x = 1011{,}111001_2$$

Para transformarlo a decimal, se utiliza la misma fórmula que para los enteros, pero teniendo en cuenta que el índice de los dígitos posteriores a la coma es negativo:

$$x = \sum_{i=-m}^{n-1} 2^i a_i$$

Don de m es el número de bits fraccionarios. En el caso del número $(1011{,}111001)_2$, el número de bits fraccionarios m es 6. Por lo tanto podemos hacer la conversión:

$$x = 1011{,}111001_2 = \textstyle\sum_{i=-6}^{4-1} 2^i a_i =$$
$$2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 + 2^{-1} \cdot 1 + 2^{-1} \cdot 1 + 2^{-2} \cdot 1 + 2^{-3} \cdot 1 + 2^{-4} \cdot 0 + 2^{-5} \cdot 0 + 2^{-6} \cdot 1 =$$
$$= 8 + 2 + 1 + \tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + \tfrac{1}{64} = 11{,}890625_{10}$$

Podemos reescribir el mismo número con notación exponencial:

$$x = 1011{,}111001 = 1011{,}111001 \cdot \frac{2^3}{2^3} = 1{,}011111001 \cdot 2^3$$

Fijémonos que en este caso hemos reducido la parte entera a su bit más significativo, lo cual ha implicado desplazar 3bits hacia la izquierda la coma, por lo que hemos de multiplicar por 2^3 el número para que sea el mismo número. Si el número es menor que uno el proceso sería semejante:

$$x = 0{,}000111011 = 0{,}000111011 \cdot \frac{2^4}{2^4} = 1{,}11011 \cdot \frac{1}{2^4} = 1{,}11011 \cdot 2^{-4}$$

Hemos vuelto a normalizar para que sólo haya un bit entero., y este debe ser 1.

Para codificar los números en Java se utiliza la notación IEEE 754. Esta divide los números en signo, exponente y mantisa:

Número	Signo	Exponente	Mantisa
1,11011	+	0	1,11011
-1,11101	-	0	1,11101
$1{,}1101 \cdot 2^3$	+	3	1,1101
$1{,}1101 \cdot 2^{-3}$	+	-3	1,1101
$-1{,}111001 \cdot 2^4$	-	4	1,111001
$-1{,}10001 \cdot 2^{-5}$	-	-5	1,10001

Tabla 8. Elementos de un número en coma flotante.

Se observa que a diferencia de los número enteros, en este caso guardamos por un lado el valor de la mantisa, y por otro el signo, este tipo de codificación se denomina signo-módulo. Para guardar el signo del exponente, en lugar de codificar el signo, lo que se hace es añadirle un desplazamiento. En otras palabras, no se guarda el exponente en si mismo sino la diferencia respecto al máximo exponente posible y de este forma el valor que se guarda es siempre positivo.

Además, también se codifican cuatro valores especiales:

- Infinito positivo (+∞)
- Infinito negativo (-∞)

- No es un número (Not a Number, NaN)

El valor NaN se utiliza en el caso de indeterminaciones y/o errores. A modo de ejemplo, el resultado de la indeterminación 0/0 es NaN. Otro caso es la raíz cuadrada de un número negativo, ya que esta no se puede codificar con un real, por lo que el resultado vuelve a ser NaN.

La codificación de los diferentes número en forma binaria viene establecida por la propia normativa. Para un valor en coma flotante de precisión simple (float), la organización de los bits es:

31	**30**	**…**	**23**	**22**	**…**	**0**
Signo(s)	**Exponente desplazado (e)**			**Mantisa(f)**		

Figura 30. Campos del tipo en coma flotante de precisión simple (`float`).

Donde S es el signo de la mantisa, el exponente es de 8 bits, lo cual permite codificar 256 exponentes. La norma establece que estos van de -126 a 127 con un desplazamiento de 127. Y por último la mantisa, que se guardan los dígitos fraccionarios, ya que se considera que los números están normalizados para que el bit entero sea siempre 1.

Los valores del exponente y la mantisa definen los posibles valores:

- Si el exponente es 255 y la mantisa es 0, el valor será ±∞ en función del valor del signo
- Si el exponente es 255 y la mantisa es distinta de 0, el valor del número es indeterminado (Not a Number, NaN)
- Si el exponente está entre 1 y 254 ambos inclusive, el número es $(-1)^s \cdot 2^{e-127} \cdot (1,f)$
- Si el exponente es 0 y la mantisa es distinta de 0, el valor del número es $(-1)^s \cdot 2^{-126} \cdot (0,f)$ (números no normalizados)
- Si el exponente es 0 y la mantisa es 0, el valor es $(-1)^s \cdot 0$ (cero)

Para el tipo `double` la organización pasa a ser:

63	**62**	**…**	**52**	**51**	**…**	**0**
Signo(s)	**Exponente desplazado (e)**			**Mantisa(f)**		

Figura 31. Campos del tipo en coma flotante de precisión doble (`double`).

En este caso, la codificación pasa a ser:

- Si el exponente es 2047 y la mantisa es 0, el valor será ±∞ en función del valor del signo
- Si el exponente es 2047 y la mantisa es distinta de 0, el valor del número es indeterminado (Not a Number, NaN)
- Si el exponente está entre 1 y 2046 ambos inclusive, el número es $(-1)^s \cdot 2^{e-1023} \cdot (1,f)$
- Si el exponente es 0 y la mantisa es distinta de 0, el valor del número es $(-1)^s \cdot 2^{-1022} \cdot (0,f)$ (números no normalizados)
- Si el exponente es 0 y la mantisa es 0, el valor es $(-1)^s \cdot 0$ (cero)

En la tabla se muestran los valores más pequeños y más grandes que se pueden representar.

Tipo	Espacio	Rango (inclusive)
float	32bits (4 bytes)	Desde $\pm 1{,}4 \cdot 10^{-45}$ a $\pm 3{,}4028235 \cdot 10^{38}$
double	64 bits (8 bytes)	Desde $\pm 4{,}9 \cdot 10^{-324}$ a $\pm 1{,}7976931348623157 \cdot 10^{308}$

Tabla 9. Tipos de datos en coma flotante.

Como en el caso de los enteros, existe un objeto asociado que contiene constantes que indican el valor máximo y mínimo posible.

Tipo	Clase	Constante valor máximo	Constante valor mínimo
float	Float	Float.MAX_VALUE	Float.MIN_VALUE
double	Double	Double.MAX_VALUE	Double.MIN_VALUE

Tabla 10. Objetos asociados a los números en coma flotante y constantes de valor máximo y mínimo.

Se observa que el exponente del número menor en valor absoluto es mayor que el del número máximo. El motivo son los números no normalizados, que permiten codificar números más pequeños.

Se recomienda utilizar valores de tipo '`double`', al asegurar una mayor precisión. El motivo fundamental de utilizar '`float`' es para reducir espacio en el caso de vectores y matrices. Aunque a la hora de realizar cálculos es recomendables convertirlo a '`double`'.

También existen literales para estos tipos:

- Se pueden escribir en formato decimal, utilizando la E para separar el exponente. La separación entre la parte entera y la fraccionaria sigue la notación inglesa utilizando un punto '.' ($3{,}23 \cdot 10^{-15}$ se escribe 3.23E-15).
- El sufijo F indica que es de tamaño '`float`'.
- En las nuevas versiones de Java (a partir de la 5.0) se puede especificar en hexadecimal, donde p sirve para separar el exponente (0,125 es 0x1.0p-3)

En programas para ciencia e ingeniería, los tipos en coma flotante se utilizan de forma intensiva.

Ejercicio 6: Escribir el número -1.0 en binario. Convertirlo a hexadecimal. Ver si el resultado coincide con el del programa:

```
/*
 * Clase para ver diferentes notaciones en Java
 */
package testnotaciones;

/**
 *
 * @author Jose M.
 */
public class Notaciones {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
                float valor = -1.F;

                System.out.println("Decimal: " + valor);
                System.out.println("Binario: " +
                                   Integer.toBinaryString(
                                   Float.floatToIntBits(valor)));
                System.out.println("Hexadecimal: " +
                                   Float.toHexString(valor));
    }
}
```

Hacer lo mismo con 0.F, -1.F/0.F y 0.F/0.F.

2.3.4.3 Caracteres

El tipo para caracteres es el **char**. Inicialmente se codificaban los datos con 8 bits en los diferentes formatos de EEUU (ASCII), Europa (ISO), Rusos (KOI), Chinos (GB y BIG)… En este caso un mismo código puede implicar diferentes letras en las diferentes codificaciones (el valor 196 es д en KOI-8 y Ä en ISO 8859-1). En 1991 se publicó el Unicode 1.0, donde se utiliza un formato de 16 bits, esto solventaba los problemas. Java nació poco después (1995) y utilizó el tamaño de 16 bits (2 bytes). En la actualidad ya se han superado los 100.000 caracteres al incorporar los ideogramas del chino, japonés y coreano. Por este motivo el Unicode ha pasado de 16 bits a 32 bits (4 bytes). Sin embargo Java mantiene el tamaño de 16 bits. Para solventar la diferencia, se utiliza la codificación UTF-16 (ver RFC2781 o Unicode).

Existen diferentes secuencias de escape aparte de los caracteres estándar:

Tabla 11. Secuencias de escape.

Secuencia	Nombre	Valor Unicode
\uXXXX	Carácter unicode	U+XXXX
\b	Retroceso	U+0008
\t	Tabulador	U+0009
\n	Salto de línea	U+000a
\f	Salto de página	U+000c
\r	Retorno de carro	U+000d
\”	Comillas dobles	U+0022
\’	Comillas	U+0027
****	Barra atrás	U+005c

La secuencia de escape Unicode se puede utilizar para el nombre de un método o sus parámetros.

Los literales de carácter pueden ser su valor unicode escrito como un entero, o bien el carácter o la secuencia de escape entre dos comillas:

- ‘A’, 65
- ‘\n’, 10
- ‘\u2122’, 2122

Ambos valores son idénticos a la hora de definir un carácter.

2.3.4.4 Booleano

El tipo ‘`boolean`’ sólo puede tener dos valores, false y true. Se utiliza para evaluar condiciones lógicas. No se pueden hacer conversiones del resto de tipos a ‘`boolean`’.

2.3.4.5 Tipos abstractos de datos: las clases

Como ya hemos comentado, los tipos abstractos de datos son aquellos que se definen a posteriori (en las librerías del lenguaje o por el programador) a partir de los tipos básicos. En Java los tipos básicos corresponden a las clases; una clase es una forma de empaquetar conjuntamente datos y los métodos que los gestionan.

Aunque los tipos abstractos de datos se usan a través de variables como los tipos básicos (ver la siguiente sección), su manipulación es mucho más rica y compleja y viene definida en general (encapsulada) por los métodos que ofrece la clase para ello, en lugar de las operaciones básicas que se usan para manipular los tipos básicos (suma, resta, multiplicación, etc. Ver sección 0).

2.3.5 Variables

Una variable es una referencia a un dato (o conjunto de datos) almacenado en memoria. Mediante las variables podemos almacenar datos en memoria, consultarlos o modificarlos. Los lenguajes de programación de alto nivel (y Java entre ellos) permiten gestionar estas referencias usando "nombres" definidos por el programador. Cada variable tiene, por tanto, un nombre asignado por el programador que permite trabajar con los datos asociados a ella.

El lenguaje Java utiliza una comprobación estricta de los tipos de datos. Esto significa que todas las variables utilizadas tienen que tener el tipo de dato declarado:: antes de usar una variable hay que añadir una sentencia donde se especifica el nombre de la variable y su tipo. A modo de ejemplo, podemos definir:

```
int indice;
double distancia;
boolean abierto;
```

Nótese que al final de la declaración debemos poner siempre un ';', ya que desde el punto de vista de Java es una sentencia completa. El tipo de la variable puede ser cualquiera de los tipos básicos discutidos en la sección anterior o bien una clase Java. Por ejemplo, la definición de una variable de la clase *String* que usaremos más adelante es:

```
String texto;
```

Las reglas para nombrar una variable se pueden resumir en:

- Los nombres de las variables distinguen entre mayúsculas y minúsculas y no pueden ser una palabra reservada (pag. 169).
- Un nombre de variable puede ser cualquier identificador legal (una secuencia de letras y dígitos Unicode que empiezan por una letra, el signo del dólar '$' o el carácter de subrayado '_'). De todas formas, se suele tomar la convención de utilizar una letra, evitando el dólar y el subrayado. Estos últimos sólo suelen utilizados por programas de generación automática de código. Y también procuran evitarlos. El carácter de espacio ' ' esta prohibido.
- Se recomienda utilizar palabras completas (distancia), evitando abreviaturas crípticas (d). De esta forma, el código puede ser autodocumentado, ya que los diferentes pasos se entiende de forma sencilla. También es importante tener en cuenta que no se pueden utilizar palabras clave o reservadas.
- Aunque respetando las normas anteriores Java permite escoger libremente los nombres de variables, suelen adoptarse unas convenciones básicas para escoger los mismos. Si el nombre se forma a partir de una sola palabra, se recomienda escribirlo en minúsculas. Si el nombre se forma a partir de dos o más palabras unidas (por ejemplo "distancia" y "recorrida"), se escribe en mayúscula la primera letra de la segunda y siguientes palabras (*distanciaRecorrida*). Si es una constante (ver la siguiente sección), por convención se escribe en mayúsculas y separando las palabras con subrayado (K_BOLTZMAN).

Se pueden realizar múltiples declaraciones en una línea, separándolas con una coma:

```
int i, j;
```

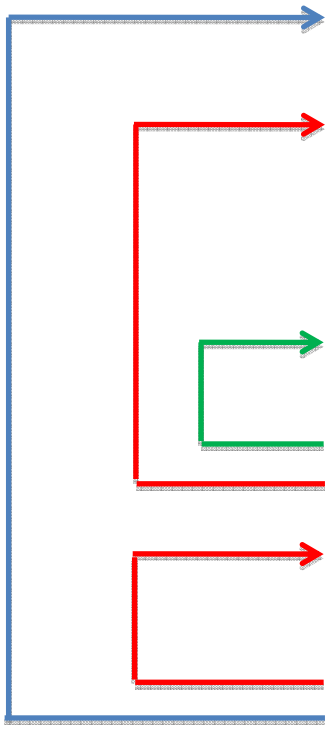
Aunque no se recomienda, ya que dificulta la lectura del programa.

2.3.5.1 *Ámbito de las variables (dentro de una clase)*

Cuando se declara una variable, la posición en el código de la clase donde se realiza la declaración determina en que partes de la clase la variable es accesible, es decir, en que partes puede usarse (su ámbito). En función de esta posición, el lenguaje Java define los siguientes tipos de variables.

- Variables definidas en el cuerpo de la clase (fuera de cualquier método). Estas variables son accesibles (se pueden usar) en cualquier lugar de la clase.
- Variables locales: son las variables definidas dentro de algún método.
    - Si se definen al principio del método son accesibles en todo el método.
    - Si se definen dentro del bloque de alguna estructura (if, while, for, …) sólo son accesibles dentro del bloque.
- Parámetros de un método: los datos necesarios para la ejecución de un método se pasan al mismo como parámetros, variables accesibles en todo el método.

En general, hablaremos de campos dentro del contexto de la clase; de parámetros en la llamada a un método; y de variables en un contexto genérico que incluye todas las anteriores.



```
/**
 * Clase main
 */
public class Main {

   /* Variables de ámbito global de la clase */
   double variableGlobal;
   /* Método con variables locales y parámetros*/
   public void metode1(double parametro1) {
        double variableLocal1;
       /* Bloque con variables locales */
       if( parametro1 > 0 ){
            double variableDeBloque;

            Operaciones ….
        }
   }
   public void metodo2(double parametro2) {
        double variableLocal2;

       Operaciones ….
    }
}
```

2.3.5.2 *Variables de clase y variables de instancia*

Cuando se declara una variable puede añadirse el modificador `static` a la declaración:

```
static double distanciaAlSol;
```

Dependiendo de si se incluye o no este modificador, tendremos dos tipos de variables:

- **Variables de instancia** (campos no estáticos): la declaración del campo no incluye el modificador `static`**.** Este tipo de variables son únicas para cada instancia, de forma que la modificación de un campo en una instancia no afecta al resto de objetos de la misma clase. Los objetos guardan sus estados individuales en campos no estáticos.
- **Variables de clase** (campos estáticos): una variable de clase es cualquier campo declarado con el modificador '`static`'. Esto indica que sólo hay una copia de esta variable para todas las instancias de la misma clase. Por ejemplo, en una clase que necesite usar la hora actual este valor deberá ser común a todas las instancias de la clase en cada momento. Queremos que el valor inicial y cualquier modificación sean compartidos por todas las instancias. En este caso podemos definir '`static long timeNanoseconds = now();`' y todas las instancias de la clase compartirán la misma hora a través de esta variable.

2.3.5.3 Constantes

Si además del modificador `static` se le añade el modificador `final`, la variable pasa a ser una **constante**. La variable sólo puede inicializarse y no puede modificarse posteriormente, y su valor es compartido por todas las instancias de la clase. Un ejemplo podría ser el número π '`static final double PI=3.141592653;`'.

Ejercicio 7: Comprobar la accesibilidad de las diferentes variables en función de su tipología.

```
class Alumne {
    private int id = 0;
    public static int proves;
    public double notes;
}

public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
     Alumne.proves = 5;
     //alumne.notes = 6.5; // No accesible
     System.out.println("Proves alumnes: " + Alumne.proves);

     Alumne pepito = new Alumne();

     pepito.notes = 6.5;
     //System.out.println("Id Pepito:" + pepito.id); // No accesible
     System.out.println("Notes Pepito: " + pepito.notes);
     System.out.println("Proves Pepito:" + pepito.proves);

     pepito.proves = 8;
     System.out.println("Proves Pepito:" + pepito.proves);
     System.out.println("Proves alumnes:" + Alumne.proves);

     testAccess(34);
    }

    static void testAccess(int dada){
     //System.out.println(pepito.proves); //No accesible
     System.out.println("Test: " + Alumne.proves);
     System.out.println("Parametre:" + dada);
    }
}
```

Quitar las barras "//" delante de las líneas de código y ver que dan error. ¿A qué es debido en cada caso?

Modificar el código anterior para crear un nuevo objeto de la clase Alumno (p.ej. "juanito") y poner de manifiesto las diferencias entre variable de instancia y variable de clase.

Analizar el efecto de cambiar la línea "testAccess(34);" por "testAccess(52);". En base a las conclusiones que

saquéis, modificar el método “testAccess()” para que tenga acceso a los datos del objeto “pepito”.

2.3.5.4 *Inicialización de variables*

Tras declarar una variable es necesario asignarle un valor de forma explícita; no es posible utilizar dicha variable si no se ha inicializado, ya que su valor es indeterminado.

En al caso de variables de uno de los **tipos básicos** la inicialización implica sólo la asignación de un valor inicial a la variable; esto se hace mediante el operador de asignación “=“, escribiendo el nombre de la variable, seguidamente el signo igual ‘=’ y por último el valor a asignar seguido de un punto y coma ‘;’.

```
int dato;
dato = 3;
```

También es posible declarar y asignar un valor a la variable en una sola sentencia:

```
double masa = 3.25;
```

El formato del valor de inicialización dependerá del tipo de dato (ver sección 2.3.4). Hay que tener cuidado con no confundirse al usar el operador de asignación, que para un programador novel se presta a confusiones por el uso del símbolo de igualdad “=”. La sentencia debe leerse como:

assigna el valor 3.25 a la variable masa: `masa` ← `3.25`

Este caso es bastante claro, pero en otros casos las asignaciones se pueden prestar a confusiones. Por ejemplo las sentencias

```
masa = 9.;
masa = masa +1.;
```

Resultan en una asignación del valor 10. a la variable masa.

Ejercicio 8: Crear un proyecto nuevo (TrabajoVariables). Crear los campos con el formato y tamaño mínimo que permitan guardar el valor indicado:

- Pequeño: 23
- Mediano: 1025
- Grande: 80000
- Extra Grande: 10^{100}

Los valores se deberán mostrar en la consola. Los nombres de la variables deberán tener en cuenta las reglas indicadas.

En la misma clase, crear una variable local al inicio del método ‘`main`’ que se llame Grande y tenga por valor 10000. Ver si afecta en la salida por consola.

Mantener la declaración de los campos, pero sin inicializarlos. Ver el error que da el compilador.

Crear un variable de tipo ‘`char`’ y asignarle un valor. Imprimir en la consola el valor asignado.

En al caso de variables de un **tipo abstracto** (clases) la inicialización implica crear una nueva instancia (un nuevo objeto) del tipo dado. Esto se hace usando la palabra clave “new”. Por ejemplo, por crear una instancia de la clase Random (generador de números aleatorios) haremos:

```
Random generador = new Random();
```

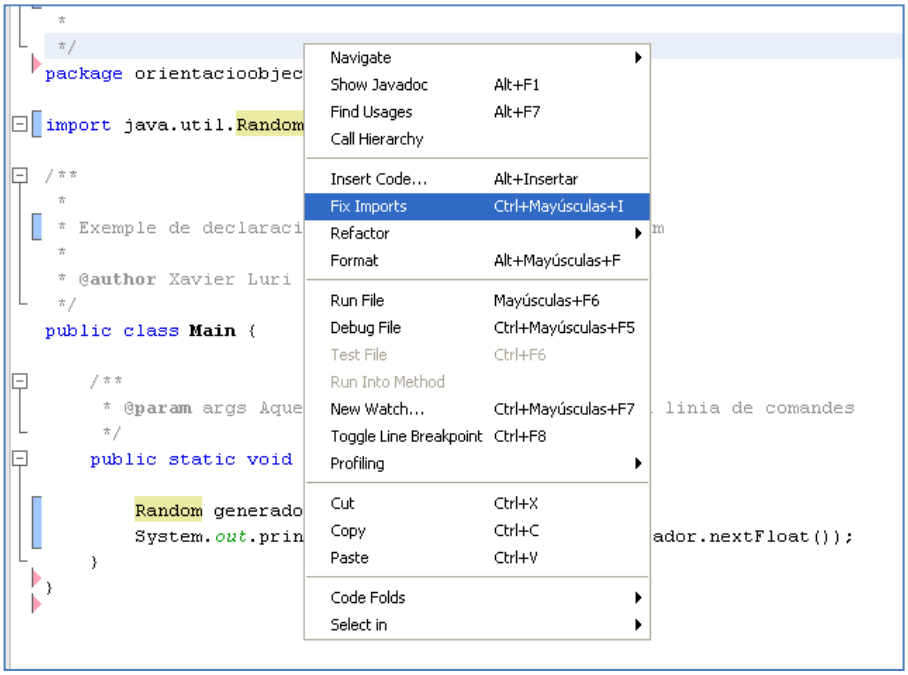
En estos casos hay que tener en cuenta que antes de poder declarar una variable de una clase dada hace falta añadir una sentencia `import` al principio del programa. En nuestro caso:

```
importe java.util.Random;
```

Esta sentencia indica a Java que el programa usará la clase `Random` de la librería básica de Java, localizada en el paquete `java.util`. El programa completo quedaría como sigue:

```
/*
 * 2.1.5 - Variables
 *
 */
package orientacioobjectes;

import java.util.Random;

/**
 *
 * Ejemplo de declaración de una variable de tipo Random
 *
 * @author Xavier Luri xluri@am.ub.es
 */
public class Main {

    /**
     * @param args Este programa no usa argumentos en línea de comandos
     */
    public static void main(String[] args) {

        Random generador = new Random();
        System.out.println("Número aleatorio: " + generador.nextFloat());
    }
}
```

NetBeans ofrece una herramienta para incluir automáticamente las declaraciones `import` en el código para las variables a las que les falte: pulsar el botón derecho del ratón sobre el código y escoger la opción “Fix imports”:



Hay algunas excepciones a este mecanismo de instanciación de clases; algunas clases de **core java** (el núcleo básico de Java) se instancian de forma diferente, como la clase `String`:

```
String texto = “Cadena de texto”;
```

Por otra parte, algunas clases que tienen métodos estáticos se pueden usar sin instanciarlas, como la clase System:

```
System.out.println(“Texto a imprimir”);
```

de hecho, esta clase no se puede instanciar.

Finalmente, si una variable de tipo abstracto (clase) no ha sido instanciada, no se puede usar pero toma un valor denominado `null` (null pointer) de forma que en el código se puede verificar si ha sido instanciada.

2.3.6 Matrices

Una matriz (array) es un contenedor que permite guardar un conjunto de datos del mismo tipo. El tamaño de una matriz se define en el momento de su creación. Una vez creada, su tamaño queda fijado. En la Figura 32 se muestra un ejemplo de matriz.

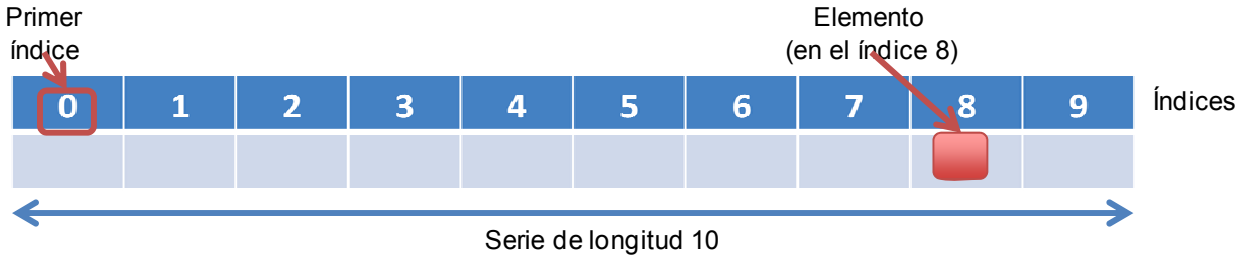


Figura 32. Matriz de 10 elementos.

Cada dato individual de la matriz se denomina elemento. Cada uno de estos elementos se accede mediante el índice o índices asociados. La numeración de cada índice comienza en el número 0, y acaba en el número de elementos de la serie menos uno, en este caso 9. El acceso a cada uno de los elementos se hace utilizando los corchetes. A modo de ejemplo, si tenemos una matriz de enteros y queremos guardar un 3 en el elemento con índice 8 utilizamos la notación:

```
datos[8] = 3;
```

Desde este punto de vista, un elemento es equivalente a una variable.

2.3.6.1 *Declarando una variable de tipo matriz*

De la misma forma que la declaración de una variable, esta también consta de dos componentes, el tipo de la matriz y su nombre. El tipo de una matriz se escribe como tipo[], donde tipo es el tipo de dato de los elementos que contiene y los corchetes indican que estamos declarando una matriz. El tamaño de la variable no se define en la declaración, sólo se indica el tipo.

Ejemplos de definiciones de matrices pueden ser:

```
byte[] matrizDeBytes;
short[] matrizDeShorts;
int[] matrizDeInts;
long[] matrizDeLongs;
float[] matrizDeFloats;
double[] matrizDeDoubles;
boolean[] matrizDeBooleans;
char[] matrizDeChars;
String[] matrizDeStrings;
```

También se pueden poner los corchetes detrás del nombre de la variable:

```
float matrizDeFloats[];
```

Aunque por convención, se recomienda evitar este formato. El poner los corchetes en el tipo simplifica la lectura.

2.3.6.2 *Creación, inicialización y acceso*

Como en el caso de las variables simples, no es suficiente con declarar una matriz para poder usarla, debe también crearse la matriz. Existen diferentes formas de crear una matriz. La primera opción es utilizar el operador 'new'. Por ejemplo, la siguiente sentencia crea una matriz de 15 dobles:

```
double[] matrizDeDoubles = new double[15];
```

Si no se crea, el compilador generará un mensaje de error indicando este hecho. Podemos corroborar el tamaño mediante la variable `length`.

```
System.out.println("Tamaño de dobles: " + matrizDeDoubles.length);
```

Al contrario que las variables simples, una vez se ha creado la matriz sus elementos son automáticamente inicializados con valor cero. Los valores de los elementos se asignan de la misma forma que hemos visto anteriormente:

```
MatrizDeDoubles[8] = 3.14;
```

Podemos presentar por consola el elemento:

```
System.out.println("Valor del elemento 8: " + matrizDeDoubles[8]);
```

También es posible crear e inicializar una matriz en un solo paso:

```
double[] matrizDeDoubles = {1.3, 72.0, 3.45, 4.23, -13.0};
```

En este caso la longitud de la matriz viene establecida por el número de valores que hay entre las llaves.

En java también existen las matrices multidimensionales, que es una matriz de matrices, para lo cual solo es necesario añadir más corchetes. Cada elemento puede ser accedido con la combinación correspondiente de índices.

```
double[][] matrizRectangular = new double[15][10];
double[][][] matrizCubica = new double[15][10] [7];
double[][] matrizTriangular = {{11.0, 12.0, 13.0}, {21.0, 22.0}};
System.out.println("Matriz triangular [0][2]=" + matrizTriangular[0][2]);
```

En el caso de '`matrizTriangular`', cada elemento de la matriz de matrices tiene un tamaño diferente. Sin embargo, hay que tener cuidado para no superar el número de elementos de la matriz. Para ello podemos volver a utilizar la propiedad '`length`', que permite saber el tamaño de la matriz. El código

```
System.out.println(matrizTriangular.length); //Primera dimension: 2
System.out.println(matrizTriangular[0].length); //Dimension matriz del indice 0: 3
System.out.println(matrizTriangular[1].length); //Dimension matriz del indice 1: 2
```

muestra la longitud de la primera dimensión de la matriz.

A partir de los ejemplos anteriores podemos ver que las matrices (o arrays) en Java (contenedores de un conjunto de datos dispuestos jerárquicamente mediante uno o más índices) son una generalización del concepto matemático de matriz (por lo general, un conjunto de números dispuestos bidimensionalmente -2 índices- en filas y columnas de longitud regular).

2.3.6.3 Copia de matrices

La clase 'System' tiene el método 'arraycopy' que permite copiar de forma eficiente matrices unidimensionales:

```
public static void arraycopy(Object src,
                             int srcPos,
                             Object dest,
                             int destPos,
                             int length)
```

Los dos argumentos objeto son la matriz origen y destino. Los tres argumentos enteros indican la posición origen de la matriz, la posición destino y el número de elementos a copiar.

Ejercicio 9: Crea una matriz de 10 elementos de tipo entero. Asígnale a cada elemento el valor de su índice. Muestra en pantalla el valor de cada elemento.

Utilizando el mismo código, intenta imprimir el elemento con índice 10. Observa lo que ocurre.

Modifica ahora el código apara asignar los 5 últimos elementos de la matriz anterior a una nueva matriz mediante el método 'System.arraycopy'

Crea una matriz de 3x2 de tipo 'double'. Llenadla con diferentes valores y mostrarlos en pantalla.

2.3.7 Cadenas

Desde el punto de vista de Java, las cadenas son secuencias de caracteres Unicode. Para definir una cadena se escribe un conjunto de caracteres y/o secuencias de escape entre comillas dobles. Un ejemplo es:

```
String cadenaHola = "Hola Java\n";
```

Las cadenas son también objetos Java, y contienen métodos que simplifican su utilización. Es importante indicar que el contenido de las cadenas no es mutable, no se puede modificar. Por lo que una modificación de cualquier cadena implica la creación previa de una nueva donde se guarda el nuevo contenido.

2.3.7.1 Longitud de una cadena

El primero, y más utilizado método de la clase 'String' es el que sirve para obtener su longitud 'length()'. Este nos proporciona la longitud en caracteres Unicode de la cadena. Es importante resaltar que esta longitud puede ser diferente del número de caracteres escritos en su inicialización. En el caso de la cadena hola, el número de caracteres escritos es 11, pero debido a que '\n' es una secuencia de escape que representa un único carácter, el método 'length()' dará un resultado de 10.

```
String cadenaHola = "Hola Java\n";
System.out.println("Longitud de :" + cadenaHola + cadenaHola.length());
```

Resultado:

```
Longitud de :Hola Java
10
```

Podemos necesitar acceder al carácter en una determinada posición de la cadena. Para ello podemos utilizar el método 'charAt':

```
System.out.println(cadenaHola.charAt(5));
```

Nos mostrara el carácter con índice 5. Como en las matrices, los índices en las cadenas empiezan en 0, el carácter que obtendremos será el 'J', de la palabra 'Java'.

2.3.7.2 Obtención de subcadenas

También es posible obtener una parte de una cadena a partir de otra. Un ejemplo:

```
String hola = "Hola Java\n";
String s = hola.substring(2, 6);
System.out.println(s);
```

La salida por consola será:

```
la J
```

El primer parámetro de 'substring' indica el índice de inicio de las subcadena, y el segundo, el índice del último carácter a coger más 1. De esta forma, la longitud de la subcadena será el último índice menos el índice inicial.

2.3.7.3 Concatenación

Podemos también concatenar cadenas. Esto se puede hacer de dos formas, con el método 'concat':

```
String saludo = "Hola";
String despedida = "Adios";
String concatenacion = saludo.concat(despedida);
System.out.println(concatenacion);
```

La salida será:

```
HolaAdios
```

Vemos que ha realizado la concatenación directa, por lo que ambas palabras están juntas. La concatenación sólo pone una cadena detrás de otra, sin ningún tipo de procesado. Veremos que el operador '+' permite realizar la misma operación.

2.3.7.4 Igualdad

Por último podemos querer ver la igualdad entre cadenas, para ello se tiene que utilizar siempre el método '`equals`'. Este método compara el contenido de la cadena, por lo que nos asegura que el resultado sólo dependerá de dicho contenido.

```
String h = "Hello";
System.out.println("Hola".equals(h));
System.out.println(h.equals("Hola"));
```

El resultado es el mismo:

```
false
false
```

Vemos que el literal de una cadena es a efectos de Java como el objeto que representaría la cadena de caracteres.

2.3.8 Enumeraciones

En determinados casos queremos clasificar una información expresada como un conjunto de valores posibles, por ejemplo las notas de una asignatura. Tendríamos en este caso seis valores posibles: Suspenso, Aprobado, Notable, Sobresaliente, Matrícula de Honor y No presentado. Podemos asociarles arbitrariamente un valor numérico y usar una variable entera para representar estos posibles valores. Sin embargo, este procedimiento lleva fácilmente a errores en la programación y Java ofrece una alternativa mejor para la representación de conjuntos de valores posibles: las enumeraciones. Como en el caso de las cadenas (*String*) las enumeraciones son **clases** que reciben un tratamiento especial. Por ejemplo, para representar las notas definiríamos la clase Notas como el siguiente enumerador:

```
enum Notas { SUSPENDIDO, APROBADO, NOTABLE, SOBRESALIENTE, MATRICULA_DE_HONOR,
             NO_PRESENTADO };
```

Una vez definida la clase, los elementos que forman la enumeración son cada uno de ellos un objeto único (en el sentido doble de que no hay otro objeto con el mismo nombre y que sólo existe una instancia del mismo) que corresponde a un valor entero específico. Podemos entonces usar cada uno de los objetos para representar el valor correspondiente en cualquier lugar del programa: almacenar el valor en una variable, comparar valores, imprimir valores:

```
Notas notaAlumno1= Notas.APROBADO;
Notas notaAlumno2= Notas.SUSPENDIDO;
…
if( notaAlumne1 == Nota.APROBADO ) {
   …
}
```

Es importante indicar que en el caso de las enumeraciones el operador '==' si funciona, ya que como hemos mencionado cada objeto de la enumeración es único y sólo se crea una vez.

Por otra parte, como clases que son los enumeradores ofrecen herramientas para facilitar su uso. Por ejemplo, podemos dar los valores de las notas de dos formas

```
System.out.println(Notas.APROBADO);
System.out.println(Notas.valueOf("NOTABLE"));
```

En la primera hacemos uso directamente del objeto para representar una nota e imprimirla como una cadena; en el segundo imprimimos el valor entero asociado (de forma unívoca) a la nota.

2.3.9 Operadores

Los operadores son elementos de la sintaxis de Java que permiten realizar diferentes acciones con las variables (modificarlas y compararlas fundamentalmente). En general son símbolos especiales que permiten realizar operaciones específicas con uno, dos o tres *operandos*, y devolver el resultado asociado. En la Tabla 12 se muestran los diferentes operadores, en función de su tipología.

Como los diferentes operadores se pueden encadenar, es importante conocer qué orden se sigue para aplicarlos cuando se usa más de uno a la vez (lo que se conoce como la **precedencia** de los operadores). En la misma Tabla 12 se indica la precedencia de los operadores: cuanto más arriba está un operador en la tabla, mayor es su precedencia. Los operadores que se encuentran en la misma línea tiene la misma precedencia y en este caso, se aplica la regla de evaluarlos de izquierda a derecha. Sólo en el caso de los operadores de asignación se invierte esta regla, evaluando de derecha a izquierda.

Se puede modificar la precedencia utilizando paréntesis '(a + b) * c' de esta forma nos aseguramos que la expresión que está dentro del paréntesis se evalúa antes. Si la expresión dentro del paréntesis tiene varios operadores, se siguen la reglas de precedencia.

Tabla 12. Operadores y su precedencia.

Precedencia	Operadores
Sufijo	*expr++ expr--*
Unitario	*++expr --expr +expr -expr* ~ !
Multiplicativo	* / %
Aditivo	+ -
Desplazamiento	<< >> >>>
Relacional	< > <= >= instanceof
Igualdad	== !=
Y binario	&
O exclusiva binaria	^
O inclusiva binaria	|
Y lógica	&&
O lógica	||
Ternario	? :
Asignación	= += -= *= /= %= &= ^= |= <<= >>= >>>=

A continuación se van a estudiar los diferentes operadores en función de su probabilidad de aparición en un programa.

2.3.9.1 *Operador de asignación simple*

El operador '=' es el más utilizado de los operadores. Permite asignar un contenido a una determinada variable.

```
dato = 3;
matrizDobles[8] = 3.14;
matrizEnteros = new int[15];
```

Como ya hemos discutido en la sección 2.3.5.4 es importante remarcar la diferencia entre una variable de tipo básico y una variable de referencia a un objeto. Las primeras guardan valores mientras que las segundas guardan la referencia a un objeto. Una referencia es como el número de serie de un objeto, permite diferenciarlos de forma

unívoca, pero no es el objeto en sí. Es como un documento en un ordenador, si sabemos su nombre completo (nombre del fichero y directorio) podemos localizarlo en el disco duro, pero el nombre no es el fichero. Sin embargo conociendo su nombre podremos acceder a él y leerlo.

Por lo tanto si tenemos una referencia podremos acceder al objeto y modificar su contenido, si está permitido. Pero si modificamos el contenido de la variable de referencia, lo que estamos haciendo es cambiar el objeto al que se refiere, no el contenido del objeto.

Existen más operadores de asignación, pero están asociaciones a otros operadores, por lo que los veremos una vez vistos el resto.

2.3.9.2 Operadores aritméticos

El lenguaje Java proporciona diferentes operadores para poder realizar sumas, restas, multiplicaciones y divisiones. Estos operadores utilizan los mismos símbolos que se utilizan en matemáticas:

- Suma (también utilizado para concatenar cadenas): '+'
- Resta: '-'
- Multiplicación: '*'
- División: '/'

También está el operador de módulo, que permite calcular el resto de una división entera:

- Módulo: '%'

Y por último quedan los operadores unitarios de indicación de signo, o cambio de signo:

- Positivo: '+'
- Negativo: '-'

A continuación se muestran ejemplos de operaciones:

```
suma = 1 + 3; // 4
resta = 4 - 5; // -1
multiplicacion = -3 * 2; // -6
division = 5 / 3; // 1
modulo = 5 % 3; // 2
sumad = 1.0 + 3.0; // 4.0
restad = 4.0 - 5.0; // -1.0
multiplicaciond = -3.0 * 2.0; // -6.0
divisiond = 5.0 / 3.0; // 1.6666666666666667
modulod = 5.0 % 3.0; // 2.0
```

Es importante indicar que los operadores aritméticos no trabajan de la misma forma con números con enteros que en coma flotante. Una primera diferencia la podemos observar en la división, que en el caso de enteros no tiene decimales.

Otros casos son:

```
suma = Integer.MAX_VALUE + Integer.MAX_VALUE; // -2
suma = Integer.MIN_VALUE + Integer.MIN_VALUE; // 0
multiplicacion = Integer.MAX_VALUE * 2; // -2
division = 1 / 0; // Exception in thread "main" java.lang.ArithmeticException: /
by zero
sumad = Double.MAX_VALUE + Double.MAX_VALUE; // Infinity
sumad = Double.MIN_VALUE + Double.MIN_VALUE; // 1.0E-323
multiplicaciond = Double.MAX_VALUE * 2.0; // Infinity
divisiond = 1.0 / 0.0; // Infinity
divisiond = 0.0 / 0.0; // NaN
```

Se observa que la suma de los enteros de valor máximo da un número negativo. Si realizamos la operación de forma binaria podemos ver por que aparece este resultado:

$$2.147.483.647_{10} + 2.147.483.647_{10} =$$

$$01111111111111111111111111111111_2 + 01111111111111111111111111111111_2 =$$

$$11111111111111111111111111111110_2 = -(11111111111111111111111111111110_2 + 1) =$$

$$-(00000000000000000000000000000001_2 + 1) = -00000000000000000000000000000010 = -2_{10}$$

Vemos que al realizar la suma, el bit de acarreo final ocupa el bit de signo, lo cual hace que el número pase de ser positivo a negativo, y que por lo tanto, el resultado sea distinto al esperado. Para realizar de forma correcta esta suma, deberíamos haber utilizado una variable de tipo largo, ya que entonces hubiéramos tenido bits suficientes. El resultado de la multiplicación es coherente con el resultado de la suma, como esperaríamos. Este comportamiento cíclico de la aritmética con enteros se utiliza en algunas ocasiones para optimizar la velocidad de algunos cálculos.

Otra situación equivalente se produce al sumar los valores más pequeños, ya que su resultado es 0, en lugar de un valor negativo. Queda como ejercicio al lector resolver el cálculo de forma binaria.

En el caso de la división entera por 0, Java genera una excepción cuando ejecuta el programa, ya que el resultado no está definido. Es importante tener en cuenta estas posibles excepciones, ya que en caso contrario el programa termina de forma inesperada.

En el caso de variables en coma flotante, la situación es muy diferente. Cuando se supera el límite inferior, el resultado pase a ser *Infinito*. Sucede de forma equivalente con la división de un número por 0, si el número es 0 el resultado es indefinido, en caso contrario devuelve un infinito.

Por último indicar que Java permite dos aritméticas en coma flotante:

- Predeterminada: La fija la plataforma
- Estricta: Como su nombre indica, sigue de forma estricta la normativa

La predeterminada suele proporcionar resultados más rápidos, al estar optimizada. La estricta nos asegura que el resultado será igual en todas las plataformas. En general, las diferencias serán mínimas, pero si se quieren comparar resultados, es recomendable utilizar el formato estricto. Para ello se pondrá la palabra clave '`strictfp`' delante del método que tenga que realizar los cálculos de esta forma:

```
public strictfp void calculo(double x)
```

Una vez comparados los resultados se puede trabajar en formato predeterminado para acelerar los cálculos.

Ejercicio 10: Crear un programa que realice las operaciones indicadas anteriormente y corroborar los resultados.

Substituir los enteros por largos por un número en coma flotante y ver el cambio en el resultado.

2.3.9.3 Operadores de incremento y decremento

En Java se incluyen operadores de incremento y decremento de variables numéricas. El operador '++' incrementa en 1 la variable, mientras que '--' decrementa 1. A modo de ejemplo:

```
int n = 23;
n++;
--n;
```

La variable n al principio tiene un valor 23, en el siguiente paso se incrementa a 24, y posteriormente se decrementa a 23 de nuevo. Teniendo en cuenta que el valor resultante se guarda en la variable, no es posible aplicar estos operadores a números directamente (3++ o 5--).

Estos operadores pueden aparecer como prefijo o sufijo. En ambos casos se realiza el incremento, pero si esta dentro de una expresión, si se utiliza el prefijo, primero se calcula el incremento y luego se evalúan el resto de operaciones de la expresión. Por el contrario, si esta como sufijo, la variable se evalúa primero, y después se le aplica el incremento. A modo de ejemplo:

```
int m;
int n;

m = 3;
n = 5;
resultado = m * ++n; // El resultado es 18 y n vale 6

m = 3;
n = 5;
resultado = m * n++; // El resultado sigue siendo 15 y n vale 6
```

En general, no es recomendable utilizar estos operadores dentro de otras expresiones ya que suelen llevar a errores y confusiones. Si se puede evitar, es mejor separar esta operación del resto de la expresión.

Ejercicio 11: Crear un programa que vaya mostrando el resultado de diferentes operaciones de incremento y decremento. Cambiar de posición en prefijo a sufijo y viceversa para ver el efecto en el resultado de las expresiones.

2.3.9.4 Operadores relacionales y booleanos

Los operadores de igualdad y relacionales permiten comparar los operadores. La mayor parte de los operadores son semejantes a los utilizados en matemáticas:

- Igual a '=='
- Distinto de '!='
- Mayor que '>'
- Menor o igual que '>='
- Menor que '<'
- Menor o igual que '<='

El resultado de estos operadores es siempre de tipo 'bool'. Un ejemplo de operación con estos operadores podría ser:

```
System.out.println("1 > 2 : " + (1 > 2));
```

El resultado será:

```
1 > 2 : false
```

Es importante recordar que el operador '==' en el caso de que se comparen objetos sólo indica si un objeto es el mismo que otro. Si, como en el caso de las cadenas, la igualdad de un objeto depende únicamente de que tengan el mismo contenido, el resultado en pocos casos será correcto. A modo de ejemplo:

```
String saludo = "Hola";
String hola = "Hola Java\n";
String subs = hola.substring(0, 4);
String saludo2 = saludo;
System.out.println(saludo + "==" + saludo2 + "? " + (saludo == saludo2));
System.out.println(saludo + "==" + subs + "? " + (saludo == subs));
System.out.println(saludo + ".equals(" + subs + ")? " + saludo.equals(subs));
```

La salida obtenida es:

```
Hola==Hola? true
Hola==Hola? false
Hola.equals(Hola)? true
```

Podemos ver que si las cadenas son el mismo objeto, el resultado es correcto. Pero que si estas tienen el mismo contenido, pero no son el mismo objetos, el resultado es erróneo para el operador '==', mientras que con 'equals' el resultado es correcto. Es importante tener en cuenta este hecho a la hora de comparar objetos.

También tenemos los operadores booleanos:

- Y lógica: '&&'
- O lógica: '||'
- Negación: '!'

Estos siguen las reglas de la Tabla 4. En muchos casos haremos caso de su traducción inglesa (And, Or y Not) por abuso del lenguaje. Es importante tener en cuenta que los operadores booleanos sólo actúan sobre variables de tipo 'boolean', que en muchos casos serán resultado de una comparación.

El operador ternario permite traducir un booleano a otro valor de forma sencilla:

- Ternario: ?:

Un ejemplo de uso podría ser:

```
System.out.println("1 > 2 : " + ((1 > 2)?"cierto":"falso"));
```

Cuyo resultado sería:

```
1 > 2 : falso
```

Finalmente se encuentra el operador de comparación de tipo:

- Comparación de tipo: 'instanceof'

En este caso nos indica si una variable o campo es de un tipo u otro. En esta caso creamos un objeto `'Integer'` y lo asignamos a uno de tipo `'Object'`:

```
Object n = new Integer(5);
System.out.println("¿n es Integer? " + ((n instanceof Integer)?"cierto":"falso"));
```

Donde preguntamos si un objeto es de tipo '`Integer`'. El resultado es:

```
¿n es entero? cierto
```

Como la clase '`Integer`' es hija de '`Number`', si hacemos la misma pregunta pero con '`Number`', el resultado también será cierto:

```
Object n = new Integer(5);
System.out.println("¿n es number? " + ((n instanceof Number)?"cierto":"falso"));
```

Donde preguntamos si un objeto es de tipo '`Number`'. El resultado es:

```
¿n es number? cierto
```

Lo mismo podríamos hacer respecto a una interfaz.

Ejercicio 12: Realizar un programa que incluya las diferentes operaciones de igualdad y booleanas y ver los resultados que aparecen en pantalla.

2.3.9.5 Operadores binarios y desplazamiento

Los operadores binarios permiten modificar bit a bit los tipos enteros. En general, dentro de Java son poco utilizados, únicamente cuando es necesario leer o escribir información de forma binaria. Los operadores son:

- Y binario: '&'
- O inclusiva binaria: '|'
- O exclusiva binaria: '^'
- Complemento a 1: '~'

Las tablas de verdad serían:

Tabla 13. Tabla de verdad de los operadores binarios

A&B	0	1
0	0	0
1	0	1

A|B	0	1
0	0	1
1	1	1

A^B	0	1
0	0	1
1	1	0

A	~A
0	1
1	0

Un ejemplo de estos operadores sería:

```
System.out.print(Integer.toBinaryString(0x00000040) + " | ");
  System.out.print(Integer.toBinaryString(0x00000008) + " = ");
  System.out.println(Integer.toBinaryString(0x40 | 0x08));
System.out.print(Integer.toBinaryString(0x00000048) + " & ");
  System.out.print(Integer.toBinaryString(0x00000008) + " = ");
  System.out.println(Integer.toBinaryString(0x48 & 0x08));
```

El resultado sería:

```
1000000 | 1000 = 1001000
1001000 & 1000 = 1000
```

Vemos que el operador | pone el bit correspondiente a 1, en este caso el 3, mientras que el & sólo mantiene a 1 cuando los dos bits son 1, de nuevo el 3, el 6 pasa a ser 0. De esta forma podemos crear máscaras. Por ejemplo, si queremos saber la mantisa del número 3.14159, tenemos que leer los bits que se muestran en rojo, o lo que es equivalente los últimos 23 bits:

$01000000100100100000111111010000_2$

Esta mascara sería el número $00000000011111111111111111111111_2$ que es equivalente a $007FFFFF_{16}$. Traducido a Java sería:

```
System.out.print("Mantisa de ");
System.out.print(3.14159F);
System.out.print("=>");
System.out.println(Integer.toBinaryString(Float.floatToIntBits(3.14159F) &
                   0x007FFFFF));
```

Cuyo resultado sería:

```
Mantisa de 3.14159=>10010010000111111010000
```

Que coincide con el resultado esperado.

Por otro lado se encuentran los operadores de desplazamiento, que permiten mover los bits a la derecha o a la izquierda.

- Desplazamiento a la izquierda: '<<'
- Desplazamiento a la derecha con signo: '>>'
- Desplazamiento a la derecha sin signo: '>>>'

Por ejemplo, si queremos obtener el exponente del mismo número tenemos que leer los bits que están en azul:

$01000000100100100000111111010000_2$

Para ello podemos aplicar la máscara $01111111100000000000000000000000_2$ o $7F800000_{16}$. Sin embargo, esto nos dejaría el valor:

$01000000000000000000000000000000_2$

El cual no deja de ser el exponente desplazado 23 posiciones a la izquierda. Por lo tanto si realizamos un desplazamiento de 23 posiciones a la derecha obtendremos el exponente del número en coma flotante. Si lo hacemos en java:

```
System.out.print("Exponente de ");
System.out.print(3.14159F);
System.out.print("=");
System.out.println(Integer.toBinaryString((Float.floatToIntBits(3.14159F) &
                        0x7F800000) >> 23));
```

El resultado obtenido es:

```
Exponente de 3.14159=10000000
```

El exponente 10000000_2 es 128_{10}, y si le restamos el desplazamiento de 127, nos queda que el exponente es 1. Teniendo en cuenta que en base a la mantisa obtenida, el número en notación binaria sería:

$$1,10010010000111111010000 \cdot 2^1 = 11,00100100000111111010000$$

La parte entera sería 11_2 o 3_{10} que es la esperada.

Fijémonos que un desplazamiento a la izquierda es equivalente a multiplicar el número por $2^{desplazamiento}$, lo cual nos puede ser muy útil para optimizar estas multiplicaciones. En este caso hemos de tener en cuenta no superar el número de bits disponibles en el tipo entero, ya que en caso contrario el comportamiento ya no sería como una multiplicación.

Por otro lado un desplazamiento a la derecha con signo es equivalente a dividir por $2^{desplazamiento}$, ya que mantiene el signo. En algunos casos nos puede interesar no mantener el signo en un desplazamiento a la derecha. En este caso utilizaremos el desplazamiento sin signo.

Con el desplazamiento podemos crear máscaras de bit sencillas. Si queremos convertir en negativo el número 3.14159, podemos aplicar una 'or' inclusiva al bit de signo. La máscara sería un bit a uno en la posición 31 o lo que es lo mismo:

```
1<<31
```

Si lo codificamos:

```
System.out.print("Cambio de signo de ");
System.out.print(3.14159F);
System.out.print("=");
System.out.println(Float.intBitsToFloat(Float.floatToIntBits(3.14159F) ^
                (1 << 31)));
```

El resultado sería:

```
Cambio de signo de 3.14159=-3.14159
```

Es importante indicar que este cambio de signo sólo es posible en coma flotante por que trabaja en signo módulo. Si hacemos la misma operación en un entero no funcionaría:

```
System.out.print("Cambio de signo de ");
System.out.print(1);
System.out.print("=");
System.out.println(1 | (1 << 31));
```

En este caso, el resultado sería:

```
Cambio de signo de 1=-2147483647
```

Esto se debe a que los enteros utilizan notación en complemento a 2.

Ejercicio 13: Obtener el bit de signo de un número en coma flotante negativo. Ver el efecto de utilizar el desplazamiento a la derecha con signo y sin signo

Realiza un programa que cambie el signo a 1,5 y comparar el número en binario. Decodificar el signo, el exponente y la mantisa y comprobar que coincide con el valor esperado.

2.3.9.6 Operadores de asignación

Existen ocasiones en que deseamos asignar a la misma variable con la que estamos haciendo un cálculo el resultado del mismo, como podría ser el caso del operador de incremento. Por ejemplo:

```
a = a + 2;
```

Es equivalente a:

```
a += 2;
```

Estos operadores de asignación permiten reducir el código en este tipo de expresiones. A continuación se detallan todos ellos:

Tabla 14. Secuencias de escape.

Operador	Utilización	Expresión equivalente
+=	`Op1 += Op2`	`Op1 = Op1 + Op2`
-=	`Op1 -= Op2`	`Op1 = Op1 - Op2`
*=	`Op1 *= Op2`	`Op1 = Op1 * Op2`
/=	`Op1 /= Op2`	`Op1 = Op1 / Op2`
%=	`Op1 %= Op2`	`Op1 = Op1 % Op2`
&=	`Op1 &= Op2`	`Op1 = Op1 & Op2`
|=	`Op1 |= Op2`	`Op1 = Op1 | Op2`
<<=	`Op1 <<= Op2`	`Op1 = Op1 << Op2`
>>=	`Op1 >>= Op2`	`Op1 = Op1 >> Op2`
>>>=	`Op1 >>>= Op2`	`Op1 = Op1 >>> Op2`

Se puede observar que son simplificaciones de sus expresiones equivalentes. Por este motivo, la expresión '+=' también puede utilizarse con cadenas para su concatenación.

Ejercicio 14: Haciendo uso únicamente de operadores de asignación, calcular la media de los números: 10, 20, 30, 40.

2.3.9.7 Conversión entre tipos numéricos

En Java se permite la conversión directa entre tipos, si es seguro que no habrá desbordamiento.

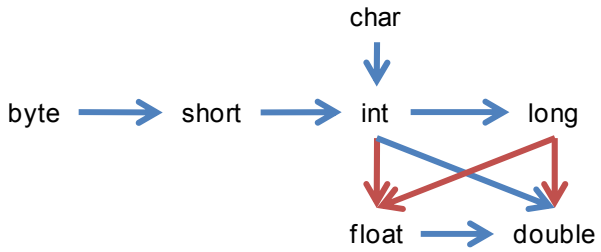


Figura 33. Conversiones válidas entre tipos numéricos.

En la Figura 33 se muestran las posibles conversiones. Las flechas en rojo indican conversiones donde puede haber pérdida de precisión, mientras que las azules aseguran que no la habrá. A modo de ejemplo si convertimos un entero de 32 bits a float, la mantisa solo nos permite 23 dígitos, lo que son 24 reales si tenemos el digito de la normalización y si añadimos el signo serán 25, 7 dígitos menos que los de un int. Dicho de otra forma, los número 2147483457 y 2147483583 se redondean al mismo número float 2147483520.

Este valor está relacionado con la precisión de la máquina, que es el valor más pequeño que sumado a 1,0 da un valor distinto de 1,0. Este valor se suele denotar con ε y su valor para float es $1{,}19 \cdot 10^{-7}$, mientras que para double es $2{,}22 \cdot 10^{-16}$. Si miramos el caso del float en binario, vemos que $1{,}19 \cdot 10^{-7}$ es 2^{-23}. Teniendo en cuenta que la mantisa tiene 23 dígitos binarios, esta suma significa cambiar el valor del bit menos significativo de la mantisa del número 1,0.

Cuando se realizan operaciones entre valores de diferente tipo numérico, se realiza la conversión a un tipo que asegure que no se produce desbordamiento, aunque se pueda perder precisión:

- Si un operando es double se convierte a double.
- En caso contrario, si un operando es float se convierte a float.
- Si no se cumple alguna de la anteriores, y un operando es long, se realiza la conversión a long
- En el resto de casos, los operandos se traducen a int.

Si queremos realizar alguna conversión entre tipos que no es directa, podemos aplicar moldes. Estos son el tipo al que deseamos convertir entre paréntesis '`(tipo)`'. Un ejemplo de molde seria:

```
double x = 3.9;
int nx = (int) x; // nx es 3
```

En este caso, al realizar la conversión se descarta la parte fraccionaria. Si lo que deseamos es redondear el número en coma flotante, podemos aplicar el método Math.round:

```
double x = 3.9;
int nx = (int) Math.round(x); // nx será 4
```

Si realizamos una conversión de un tipo a otro, y el número original supera el valor máximo permitido, pueden obtenerse valores extraños:

```
int n = Integer.MAX_VALUE; // n = 2147483647
short s = (short) n; // s = -1
```

A simple vista, el valor obtenido no tiene que ver nada con el original, sin embargo, se puede ver que el error se debe al truncamiento.

Por último indicar que existen dos clase en Java que permiten mantener la precisión si se trabaja con números enteros y racionales, estas son BigInteger para los enteros, y BigDecimal para los racionales. Sin embargo su uso se debe realizar con mucha precaución ya que incrementa de forma notable los recursos necesarios para realizar los cálculos.

Ejercicio 15: Codifica el cálculo de las suma de ε a un '`float`' y obtén el resultado en forma binaria. Mira el resultado de la suma en forma binaria.

Analiza porque al cambiar el '`int`' más alto a '`short`' este pasa a ser -1.

Obtener con precisión absoluta el valor máximo que se puede representar con un '`float`' (0x1,FFFFFEp127).

2.3.9.8 *Errores de redondeo*

Hemos visto que en la conversión de números se puede producir una reducción de la precisión debido al número de dígitos disponibles para la mantisa. También hemos visto que si sumamos un número muy pequeño a otro, puede que este no se vea modificado por que es mucho más pequeño de lo que la mantisa del resultado permite representar. Un ejemplo podría ser el cálculo de una de las soluciones de la ecuación cuadrática:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Si b es positivo y |ac| <<b^2 el resultado puede ser completamente erróneo. Un ejemplo:

```
float fb = 1.0F;
float fa = 1.0F;
float fc = 1e-9F;
System.out.println("a = " + fa);
System.out.println("b = " + fb);
System.out.println("c = " + fc);
System.out.println("Solucion erronea: " + (-fb+(float)Math.sqrt(fb*fb-
       4*fa*fc))/(2.0*fa));
System.out.println("Solucion correcta: " +
       (float)((-fb+Math.sqrt(fb*fb-((double)4*fa*fc)))/(2.0*fa)));
```

El resultado es:

```
a = 1.0
b = 1.0
c = 1.0E-9
Solucion erronea: 0.0
Solucion correcta: -1.0E-9
```

Podemos comprobar que el resultado es completamente diferente en el primero y segundo caso.

En general estos problemas nos los encontraremos cuando se calcula una resta entre números muy semejantes y a uno de ellos se le suma o resta previamente uno mucho más pequeño, como en el caso anterior. Podemos utilizar números de mayor precisión, como hemos hecho en este caso, o bien reorganizar las fórmulas de forma adecuada, para que el efecto no sea tan dramático.

Ejercicio 16: Crear un programa que ejecute las sentencias anteriores. Extraer la información en binario y analizar el efecto en la mantisa de las dos soluciones.

2.3.10 Métodos

Un método es un bloque de código separado del resto e identificado con un nombre (el nombre del método). Este bloque de código puede ejecutarse desde otras partes de la clase (o como una llamada externa) mediante el nombre asignado (llamada al método) y con la posibilidad de pasar al mismo algunas variables (parámetros del método) que se pueden usar en el bloque de código. Al final de la ejecución del bloque de código el método puede devolver un resultado en forma de cualquiera de los tipos de datos de Java.

Los métodos permiten organizar el código de una clase, agrupando fragmentos de código que se usan repetidamente, simplificando así su reutilización y minimizando los errores. Por otra parte, los métodos proporcionan también los interfaces con que una clase devuelve resultados.

La definición de un método consta de seis elementos:

- **Modificadores:** modifican la accesibilidad o tipo del método; los modificadores pueden ser '`public`', '`private`', '`static`',...
- **Tipo de retorno:** si el método retorna un valor, indica el tipo; en caso contrario se indica con '`void`'.
- **Nombre del método:** se aplican convenios similares a los nombres de variables, aunque con recomendaciones suplementarias para seleccionarlos dependiendo de la función que el método realice.
- **Parámetros:** lista entre paréntesis de los parámetros. Se separan por coma y van precedidos por el tipo de datos asociado. Si no tienen parámetros se ponen unos paréntesis vacios.
- **Lista de excepciones:** lo veremos más adelante (sección 0)
- **Cuerpo del método:** el bloque de código del método (delimitado por las llaves), es decir, el conjunto de sentencias y variables que implementan el algoritmo del método. El cuerpo se debe escribir entre llaves.

Un ejemplo de método podría ser:

```
public static double sinh(double x) {...}
```

En terminología de Java, se denomina **firma** (*signature* en inglés) de un método al nombre y los parámetros asociados al mismo. En nuestro ejemplo:

```
sinh(double)
```

En general, **los nombres de los métodos deben empezar por el verbo de la acción que realizan**. Sólo en casos excepcionales, como las funciones matemáticas, no se cumple esta regla. En general sólo suele haber un método en un clase con el mismo nombre. Sin embargo, Java permite que hayan dos métodos con el mismo nombre (se denomina sobrecarga), pero no pueden tener la misma firma. A modo de ejemplo, podríamos definir estos métodos en la misma clase:

```
public static double sinh(double x);
public static float sinh(float x);
```

Sin embargo estos serían incompatibles al tener la misma firma:

```
public static double sinh(double x);
public static float sinh(double x);
```

Es importante tenerlo en cuenta a la hora de definir los métodos. También es importante tener en cuenta en el uso de los métodos que dos métodos con diferente firma son en principio distintos en su comportamiento. Se debe procurar minimizar el número de métodos sobrecargados, especialmente si no tienen comportamientos equivalentes sobre los diferentes tipos que tienen como parámetros.

2.3.10.1 Retorno de un método

Cuando un método termina la ejecución de su código devuelve en general como resultado de la misma unos ciertos datos. El tipo de estos datos viene determinado por el tipo de retorno con que se haya definido el método. Por ejemplo el método:

```
double radio(double x, double y)
```

devuelve un *double* como resultado de la ejecución. Un método puede devolver como resultado cualquier tipo de datos de Java, sea un tipo básico o un tipo abstracto (un objeto o una matriz). También puede no devolver ningún dato y en este caso debe declararse como de tipo 'void'.

Los datos de retorno de un método pueden usarse en cualquier punto de un programa donde pueda usarse un valor del tipo dado. Por ejemplo, podemos usar el retorno del método *radio* definido anteriormente de la forma siguiente:

```
public static void main(String[] args) {
  double r = radio(1.0, 1.0);
  System.out.println(r);
}
```

En este caso usamos el dato de retorno para asignarlo a una variable del mismo tipo *double*.

Un método puede acabar su ejecución de tres formas distintas:

- Se llevan a cabo todas las sentencias del método y se llega al final del bloque
- Se ejecuta una sentencia de 'return' en el código del bloque
- Lanza una excepción

El primer caso sólo es posible en los métodos declarados como 'void'; en estos, no hace falta incluir una sentencia 'return' sino que simplemente con llegar al final del bloque de código el método terminará. En este tipo de métodos también puede usarse la sentencia 'return' para terminar el método en cualquier punto del código y en este caso no llevará ningún parámetro:

```
  return;
```

Por ejemplo:

```
public static void metodoVoid(int valor) {
  if( valor > 0 ){
        System.out.println(”Valor positivo”);
        return;
  }

  System.out.println(”Valor no positivo”);
}
```

En cualquier otro tipo de método para finalizar normalmente debe incluirse (en cualquier punto del código) una sentencia 'return' con el valor a retornar:

```
  return valor;
```

dicho valor deberá ser del mismo tipo que el especificado en la definición del método. Por ejemplo:

```
double radio(double x, double y) {
  return Math.sqrt(x*x + y*y);
}
```

En caso de devolver una clase o interfaz, el objeto ha de cumplir los requerimientos para que 'instanceof' sea cierto.

2.3.10.2 Parámetros

La declaración del método establece el número y tipos de los parámetros que se pasaran al mismo para su uso. A través de ellos se pasan los datos que permiten al método realizar sus operaciones. Por ejemplo el método que hemos definido anteriormente para calcular el radio (distancia al origen) a partir de dos coordenadas tiene como parámetros dos variables 'double':

```
double radio(double x, double y) {
  return Math.sqrt(x*x + y*y);
}
```

Para realizar una llamada al método deben simplemente proporcionarse los valores de los parámetros, en forma de valores literales o de variables del tipo adecuado. Por ejemplo, podemos llamar al método *radio()* de la formas siguientes:

```
public static void main(String[] args) {
  double x = 2.;
  double y = 2.;
  double r1 = radio(1.0, 1.0);
  double r2 = radio(x, y);

  System.out.println(r1 + " " + r2);
}
```

Los métodos pueden tener como parámetros tipos primitivos (int, byte, double, float…) o referencias, como objetos, matrices y cadenas. En el ejemplo siguiente el método tiene un parámetro de tipo matriz de objetos *Cajon*:

```
public void abrirCajones(Cajon[] cajones) {...};
```

Nótese que, en lo que respecta al método, los parámetros pueden usarse como variables normales dentro del mismo. Sin embargo, debe tenerse en cuenta que el comportamiento en lo que respecta a la llamada al método difiere según el tipo de los parámetros. **Si los parámetros son tipos primitivos, cualquier modificación que se realice en el método de su valor no tiene efecto fuera del método**. Por ejemplo:

```
class Test {
  public static void main(String[] args) {
        int x = 10;

        System.out.println(x);
        prueba(x);
        System.out.println(x);
  }

  public static void prueba(int p) {
        p = 5;
  }
}
```

El resultado en la consola es:

```
10
10
```

El valor de la variable x que hemos usado para la llamada al método *prueba* no se ve modificado en el método *main*. Lo que llega a *prueba* es únicamente el valor de x, a través de **una copia de este valor** en la variable p.

Por el contrario, en el caso de parámetros de tipos abstractos (clases) o de tipo matriz la situación es distinta. En este caso no se pasa una copia del valor almacenado en la variable, sino **una referencia al objeto asociado a la variable**. En el caso de paso de referencias se accede directamente a los datos originales, no a una copia de los valores, por lo que **cualquier modificación afecta también a los objetos o matrices** usados en el código desde donde se haya llamado el método. A continuación se muestra el mismo ejemplo anterior pero pasando los argumentos como matrices.

```
class Test {
  public static void main(String[] args) {
        int x[] = {10};

        System.out.println(x[0]);
        prueba(x);
        System.out.println(x[0]);
  }

  public static void prueba(int[] p) {
        p[0] = 5;
  }
}
```

En este caso, el resultado pasa a ser:

```
10
5
```

Como puede verse, en este caso si que se modifica el contenido de la variable x (una matriz) en el método *main* desde donde se llama al método *prueba*.

2.3.10.3 Modificadores

La declaración de un método puede incluir modificadores de acceso. Estos modificadores determinan desde donde puede accederse (usarse) el método tal como se describe en la tabla siguiente:

Tabla 15. Modificadores de accesibilidad de un método

Modificador	Clase	Paquete	Subclase	Exterior
	Si	Si	No	No
public	Si	Si	Si	Si
protected	Si	Si	Si	No
private	Si	No	No	No

Por ejemplo, un método declarado '`private`' puede usarse desde cualquier punto de la clase donde esté definido pero no podrá usarse desde clases que hereden de la misma ni podrán usarlo objetos de dicha clase instanciados en otras partes mismo paquete de la clase o en otros paquetes.

Finalmente, un método puede declararse estático. En este caso todas las variables del método serán estáticas (variables de clase, véase la sección 2.3.5.2) y el método podrá usarse sin instanciar la clase.

2.3.11 Control de flujo

Los programas que hemos hecho hasta ahora se ejecutaban en una secuencia lineal, siguiendo las sentencias en orden de la primera a la última. Sin embargo, suele ser necesario modificar el flujo de ejecución en base a sentencias condicionales basadas en el estado y valores de los datos de programa. Para esta fin Java proporciona sentencias para realizar bifurcaciones, bucles y conmutaciones que discutiremos en esta sección.

2.3.11.1 Ámbito de bloque

Un bloque es un conjunto de sentencias situadas entre dos llaves { }. Hemos visto diversos ejemplos de bloques en los programas listados en secciones anteriores, como es el caso del bloque de un método **main**:

```
public static void main(String[] args) {
  int n;
  ...
}
```

Como ya hemos discutido en la sección 2.3.5.1 las variables creadas dentro de un bloque sólo son accesibles dentro del mismo, a partir de la sentencia donde se inicializan, así como en los bloques anidados posteriores a la creación. A este respecto debe tenerse en cuenta que, no es posible crear una variable dentro un bloque anidado que tenga el mismo identificador que otra de un bloque superior. Por ejemplo, el siguiente programa nos dará un error:

```
public static void main(String[] args) {
  int n;
  ...
  {
        int n;
  }
}
```

En este ejemplo hemos definido un bloque autónomo, pero en general los bloques se usan como parte de las sentencias de control de flujo (que veremos a continuación) o para delimitar los métodos.

2.3.11.2 Bifurcaciones

La sentencia de bifurcación básica de Java tiene la forma:

```
if (condicion) sentencia;
```

La condición tiene que estar entre paréntesis. Si es necesario ejecutar múltiples sentencias, se puede hacer uso de un bloque (de hecho, se recomienda usar el bloque aunque sólo haya una sentencia para hacer más legible el código):

```
if (condicion) {
  ...
}
```

Por ejemplo, si queremos realizar el valor absoluto de un número entero:

```
if (n < 0) n = -n;
```

En este caso, la sentencia de asignación se ejecutará cuando n sea negativo.

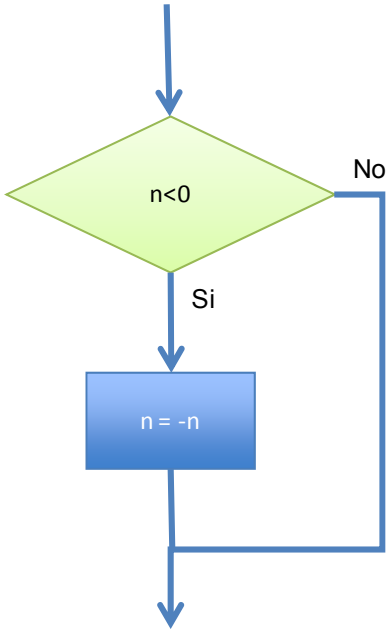


Figura 34. Diagrama de flujo de la sentencia if.

Tal como se muestra en la Figura 34, una vez ejecutadas las sentencia if, el flujo del programa vuelve a ser común.

La sentencia condicional más general de Java tiene este aspecto:

```
if (condicion) sentencia1;
else sentencia2;
```

O bien:

```
if (condicion) { … }
else {…}
```

En el ejemplo siguiente se muestra como obtener el signo de un entero.

```
if (n < 0) n = -1;
else n = 1;
```

El diagrama de flujo de la sentencia if-else se muestra en la Figura 35.

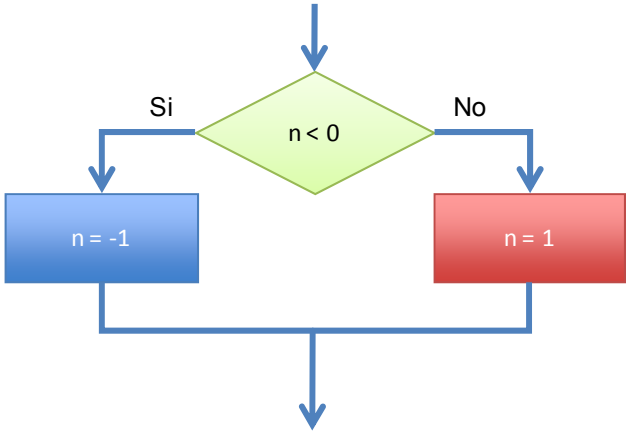


Figura 35. Diagrama de flujo de la sentencia if-else.

Se pueden anidar sentencias if-else. De esta forma se pueden establecer rangos.

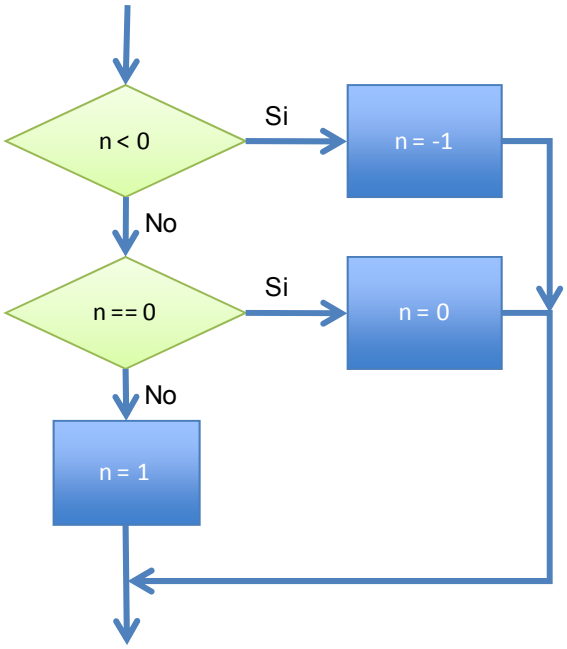


Figura 36. Diagrama de flujo de una bifurcación anidada.

En este caso la codificación en Java sería:

```
if (n < 0) n = -1;
else if (n == 0) n = 0;
else n = 1;
```

En todos los casos se podría hacer uso de bloques en lugar de sentencias, sin embargo por simplicidad no se han mostrado.

Otro ejemplo sería obtener la calificación a partir de una nota:

```
enum Notas { SUSPENDIDO, APROBADO, NOTABLE, SOBRESALIENTE, MATRICULA_DE_HONOR,
             NO_PRESENTADO };
public static Notas clasificacionNotas(float nota) {
    if (nota < 5.0) {
      calificacion = Notas.SUSPENDIDO;
    } else if (nota < 6.5) {
      calificacion = Notas.APROBADO;
    } else if (nota < 8.5) {
      calificacion = Notas.NOTABLE;
    } else if (nota < 10.0) {
      calificacion = Notas.SOBRESALIENTE;
    } else if (nota == 10.0) {
      calificacion = Notas.MATRICULA_DE_HONOR;
    } else {
      calificacion = Notas.NO_PRESENTADO;
    }

    return calificacion;

}
```

En este caso realizamos la conversión de un rango a una categoría.

Ejercicio 17: Realizar un programa con los diferentes ejemplos presentados en esta sección, mostrando por consola el resultado de las comparaciones.

2.3.11.3 Bucles

Los bucles permiten realizar de forma repetitiva un conjunto de sentencias. Existen cuatro bucles básicos:

- Mientras se cumpla la condición ejecuta repetidamente un bloque de sentencias
- Ejecuta el bloque de sentencias y repite la ejecución mientras se cumpla la condición
- Ejecuta un bloque de sentencias un número de veces controlado por un contador
- Ejecuta las sentencias para todos los elementos de una colección

El primer bucle utiliza la palabra ‘while’ (mientras), y ejecuta una sentencia o un bloque.

```
while (condicion) {
  ...
}
```

Por ejemplo, para hacer un división entera por “fuerza bruta”, el algoritmos sería:

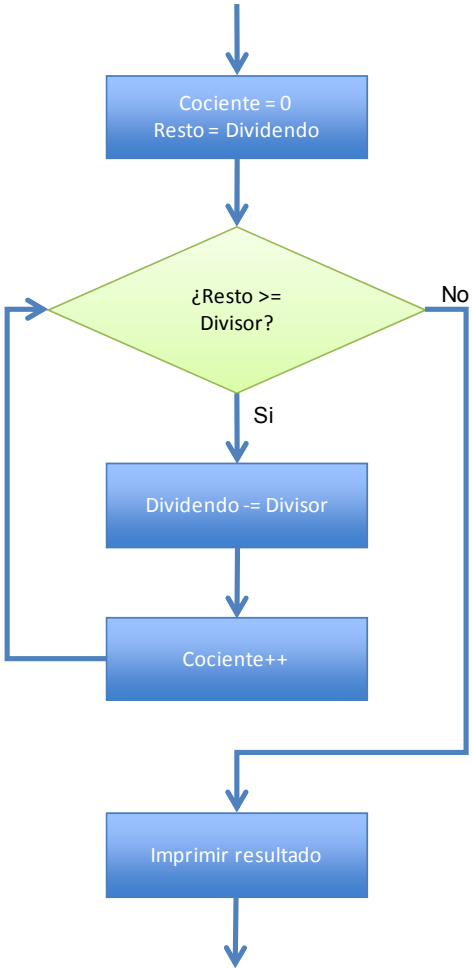


Figura 37. Diagrama de flujo de una división básica.

Si lo codificamos en Java, quedaría de la siguiente forma:

```
byte dividendo = 59;
byte divisor = 3;
byte resto = dividendo;
byte cociente = 0;

while (resto >= divisor) {
  resto -= divisor; // Equivalente a resto = resto - divisor
  cociente++;
}

System.out.print(dividendo + "/" + divisor + " = ");
System.out.println(cociente + " + " + resto + "/" + divisor);
```

El resultado sería:

```
59/3 = 19 + 2/3
```

Evidentemente esta forma de dividir no es muy optima, sobre todo para números grandes, pero nos sirve como ejemplo.

Otra opción es que se tengan que realizar las sentencias antes de poder realizar la comprobación. Este sería el caso de bucle 'do-while' (haz-mientras).

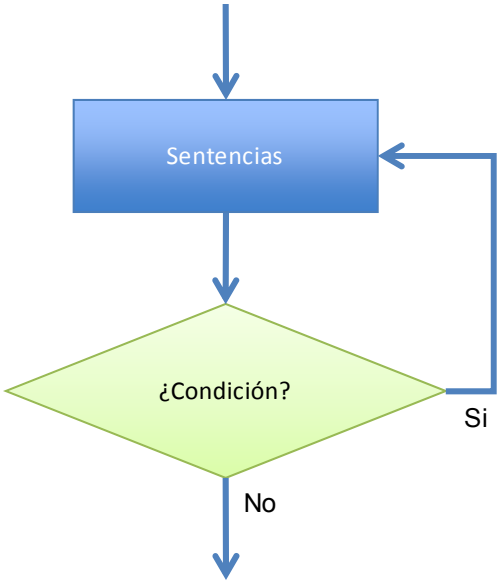


Figura 38. Diagrama de bloques de un bucle do-while.

La sintaxis sería:

```
do{
  ...
}while(condicion);
```

Este tipo de bucle suele ser útil por ejemplo en la gestión de entrada-salida, como veremos más adelante.

Los bucles 'for' permiten ejecutar un bloque de sentencias un número de veces controlado por un contador. La sintaxis del bucle 'for' sería:

```
for (inicializacion; condicion; incremento) {
  ...
}
```

Donde los diferentes elementos serían:

- Inicialización: Se crean las variables de contador necesarias y se inicializan
  - int i = 0
  - int i = 1
  - float a = Math.PI
  - int i = 1, j = 3
- Condición: es la condición que se tiene que cumplir para seguir en el bucle
  - i < matriz.length
- Incremento: sentencia de modificación del contador
  - I++
  - a += Math.PI/2.0;

Por ejemplo, si queremos imprimir el valor de todos los elementos de una matriz podemos hacerlo:

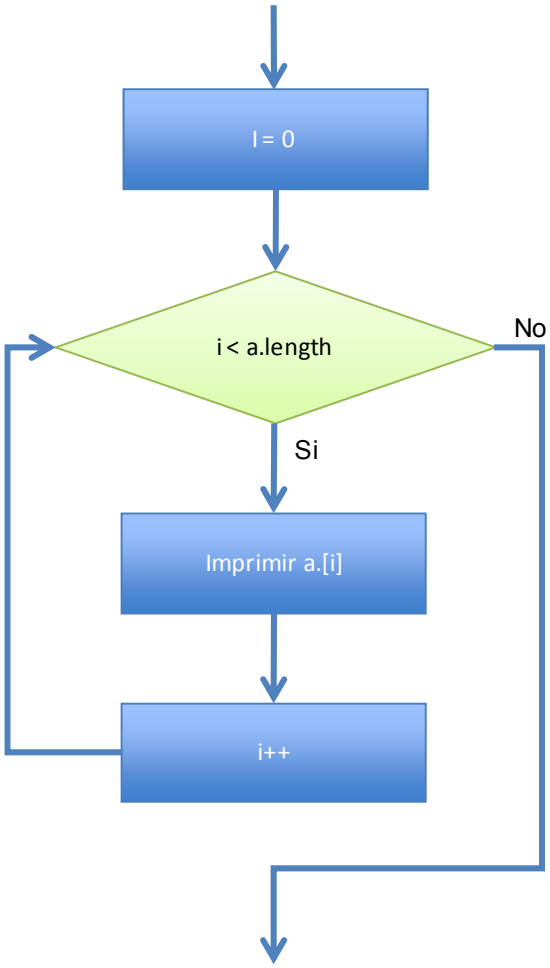


Figura 39. Diagrama de bloques de un bucle for.

La sintaxis en Java sería:

```
int[] a= {2, 3, 9, 23, 15};

for (int i = 0; i < a.length; i++){
  System.out.println("a[" + i + "] = " + a[i]);
}
```

Vemos que primero inicializamos el bucle creando una variable índice (i) y dándole su valor inicial. Esta variable se compara con la condición, y si se cumple imprimimos el valor. Cuando se han ejecutado las sentencias, la variable índice se incrementa. El resultado sería:

```
a[0] = 2
a[1] = 3
a[2] = 9
a[3] = 23
a[4] = 15
```

El bucle 'for' permite una sintaxis bastante compleja. Por ejemplo, supongamos que guardas un conjunto de números complejos en una matriz de dobles, siendo los números pares la parte real y los impares la imaginaria. Para simplificar la impresión de la matriz podríamos hacerlo de esta forma:

```
double[] c = {2.0, 3.3, 7.0, 1.0, 3.5, 2.8};

for (int i = 0, j = 0, k = 1; j < c.length; i++, j+=2, k+= 2){
  System.out.println("c[" + i + "] = " + c[j] + "+i" + c[k]);
}
```

En este caso, inicializamos tres variables enteras, la primera (i) será el índice del número complejo, la segunda (j) es el índice de los valores reales, y la última variable (k), el índice de los números imaginarios. La comparación la hacemos respecto al índice de los números reales, en caso contrario podríamos tener un desbordamiento. Los incrementos se realizan de 1 en 1 en el caso de la i, mientras que para los reales e imaginarios es de 2 en 2.

La salida por pantalla es:

```
c[0] = 2.0+i3.3
c[1] = 7.0+i1.0
c[2] = 3.5+i2.8
```

Es importante indicar que el bucle for permite cualquier tipo de condición (no es necesario que sea basada en los índices), y que permite cualquier tipo de incremento. Sin embargo, es recomendable ceñirse al formato anterior, ya que sino el programa se puede hacer bastante difícil de leer.

Por último existe una opción para recorrer colecciones, entre ellas las matrices, denominado bucle for avanzado. En este caso la sintaxis pasa a ser:

```
for (tipo elemento: coleccion) {
  ...
}
```

En el caso de una matriz sería:

```
for (double elemento: c) {
  ...
}
```

Que traducido al formato del for básico sería:

```
for (int i = 0; i < c.length; i++) {
  double elemento = c[i];

  ...
}
```

Vemos que lo que hace el for avanzado es copiar el valor de la colección, en esta caso la matriz c al elemento. Por lo tanto cualquier asignación que realicemos a elemento no afectará al contenido de c. Si queremos llenar una matriz y posteriormente mostrar los datos tenemos que utilizar esta alternativa:

```
long[] a = new long[10];

for (int i = 0; i < a.length; i++) {
  a[i] = System.currentTimeMillis();
}

for (long l : a) {
  System.out.println(l);
}
```

Es importante remarcar que la variable (en este caso l) sólo recibe el valor contenido en la colección, pero no se puede modificar. Por eso hemos hecho uso del for básico para la inicialización.

Ejercicio 18: Hacer una multiplicación de forma semejante a la división mediante un bucle while.

Realizar los ejercicios de matrices (pag. 67) utilizando el bucle *for* básico y *for* avanzado.

2.3.11.4 Conmutador

La sentencia de conmutación es una de las más potentes para realizar máquinas de estado, y una de las más susceptibles de error por su sintaxis.

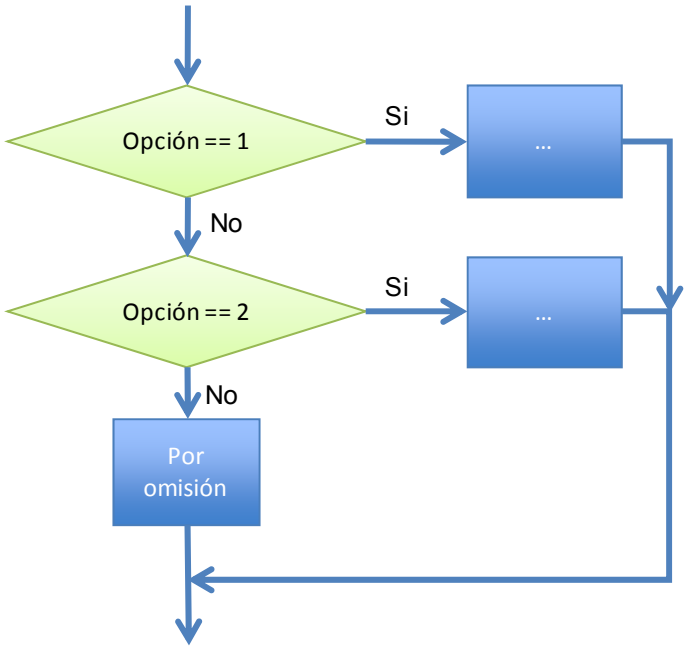


Figura 40. Diagrama de flujo de un conmutador.

Su utilización conjuntamente con las enumeraciones permite gestionar estados:

```
enum Estados { INICIO, … ,FIN };
…
Estados estado = Estados.INICIO;
switch (estado){
      case INICIO:
            System.out.println("Inicializando");
            estado = Estados.POSICION;
            break;
      case POSICION:
            System.out.println("Leyendo la posicion");
            estado = Estados.DATO;
            break;
      case DATO:
            System.out.println("Leyendo el dato");
            estado = Estados.FIN;
            break;
      case FIN:
            System.out.println("Finalizando");
            estado = null;
            break;
      default:
            System.out.println("Error");
}
return estado;
```

Este podría ser parte de una máquina para ir leyendo los datos de un fichero. Importante tener en cuenta que hay que finalizar cada grupo de sentencias de cada caso con una sentencia 'break'. En caso contrario, se continúan ejecutan las sentencias hasta encontrar un 'break' o llegar al final del conmutador.

Ejercicio 19: Hacer con una enumeración y un conmutador un sistema para determinar el número de días de un mes.

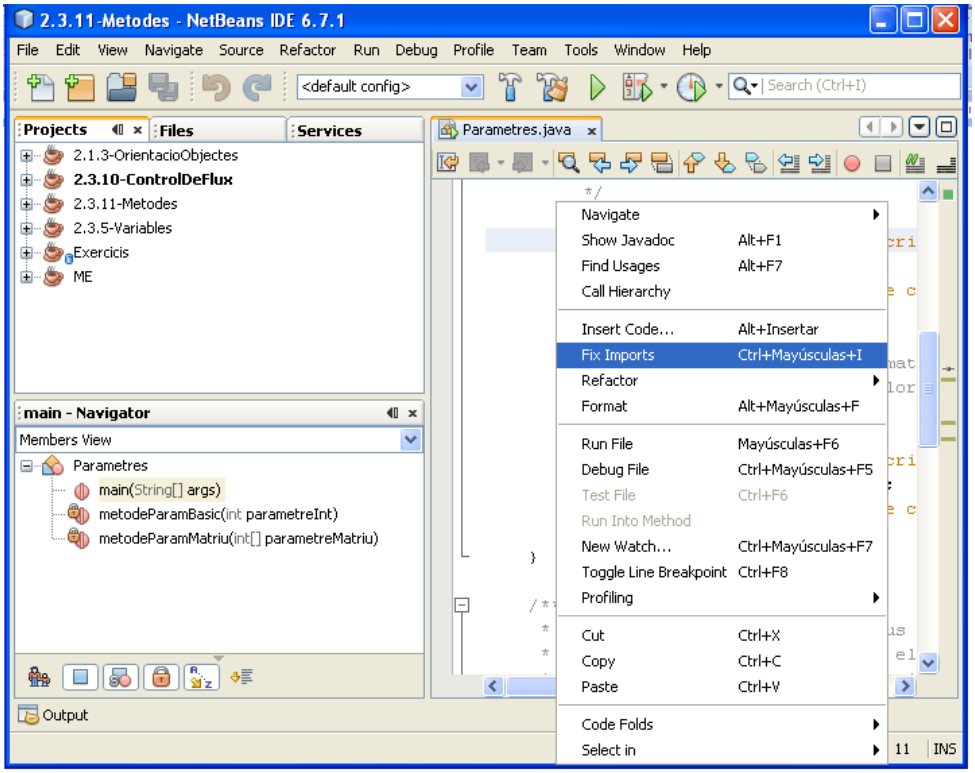
2.3.12 Importar paquetes

Tal como indicamos en la sección 2.3.3.6, Java organiza las clases en paquetes. El paquete genérico java.lang esta siempre accesible, pero el resto de paquetes tiene que ser importado para poderse utilizar. Para ello se tiene que utilizar la sentencia `'import'`. Podemos importar de tres formas diferentes:

- `'import java.util.*'`: Importa toda las clases del paquete java.util
- `'import java.util.Scanner'`: Importa exclusivamente la clase Scanner
- `'import java.lang.Math.*'`: Importa todos los métodos y variables de tipo `'static'` de la clase (*Math* en este caso), y se pueden utilizar como si estuvieran definidos en la clase actual.

En general ser recomienda evitar el uso de las opciones primera y tercera, ya que facilitan la aparición de errores..

La sentencia `'import'` se ubica al principio del fichero, antes de empezar la definición de la clase. *NetBeans* incluye una herramienta para incluir automáticamente las sentencias `'import'` necesarias en un programa. Para ello simplemente debe pulsarse el botón derecho del ratón sobre la ventana del código fuente; con ello aparece el menú contextual siguiente:



Al seleccionar la opción "Fix imports" se añaden automáticamente las sentencias `'import'` necesarias; en algunos casos puede haber ambigüedades (dos clases con el mismo nombre en dos paquetes distintos) y *NetBeans* puede pedirnos que seleccionemos el paquete apropiado.

2.3.13 Entrada y salida estándar

Hasta ahora hemos estudiado como implementar la lógica de un programa y como guardar la información en memoria, pero no hemos visto como introducir datos en el programa desde periféricos o como guardar los resultados. Existen diferentes opciones para ello:

- Argumentos
- Flujos estándar
- Ficheros
- Interfaz gráfica

En esta sección analizaremos las dos primeras opciones y en la sección 2.3.15 estudiaremos el uso de ficheros. Por otra parte, el uso de la interfaz gráfica cae fuera del ámbito de este curso y no lo estudiaremos.

2.3.13.1 Argumentos

Los argumentos permiten proporcionar datos a un programa a través de la línea de comandos (la sentencia que se escribe en una terminal para ejecutar un programa). Por ejemplo, el siguiente comando ejecuta el programa contenido en el fichero miPrograma.jar:

```
java –jar miPrograma.jar Estos son mis 5 argumentos
```

Esta llamada pasa cinco argumentos al programa (las cinco palabras separadas por espacios). En Java estos argumentos se reciben mediante los parámetros del método 'main':

```
public static void main(String[] args)
```

La matriz de cadenas 'args' contiene los parámetros pasados en línea de comandos. En el ejemplo anterior los parámetros serían la matriz:

```
args = { "Estos", "son", "mis", "5", "argumentos" };
```

Los elementos de esta matriz pueden usarse en el programa. Por ejemplo, podemos convertir la cadena "5" a su valor entero usando el método de la clase Integer valueOf(String), que devuelve el valor correspondiente:

```
int n = Integer.valueOf(args[3]);
```

Es importante que un programa que trabaje con argumentos especifique claramente como deben ordenarse los parámetros para que el usuario no cometa errores. También debe controlarse si el usuario ha incluido el número y tipo correcto de parámetros, mostrando un mensaje de error por consola si el usuario comete un error.

Para poder pasar parámetros en *Netbeans*, deberemos ponernos en la jerarquía de proyectos encima del nuestro proyecto y pulsar el botón derecho del ratón. Seleccionaremos la opción "Properties" que aparece en el menú y nos aparecerá la ventana que se muestra en la Figura 41.

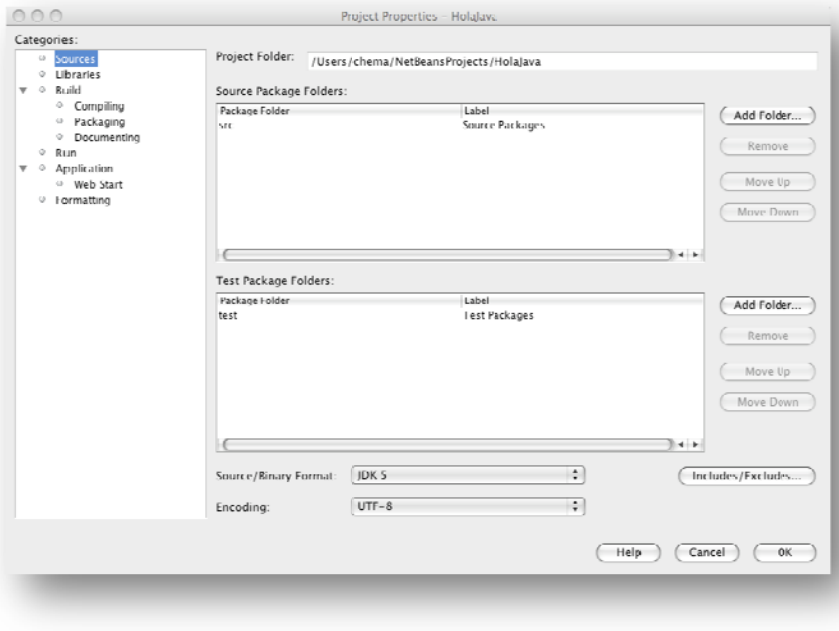


Figura 41. Ventana de propiedades de un proyecto.

En la jerarquía que aparece a la derecha seleccionaremos la opción “Run”, que nos pasará a la ventana de la Figura 42.

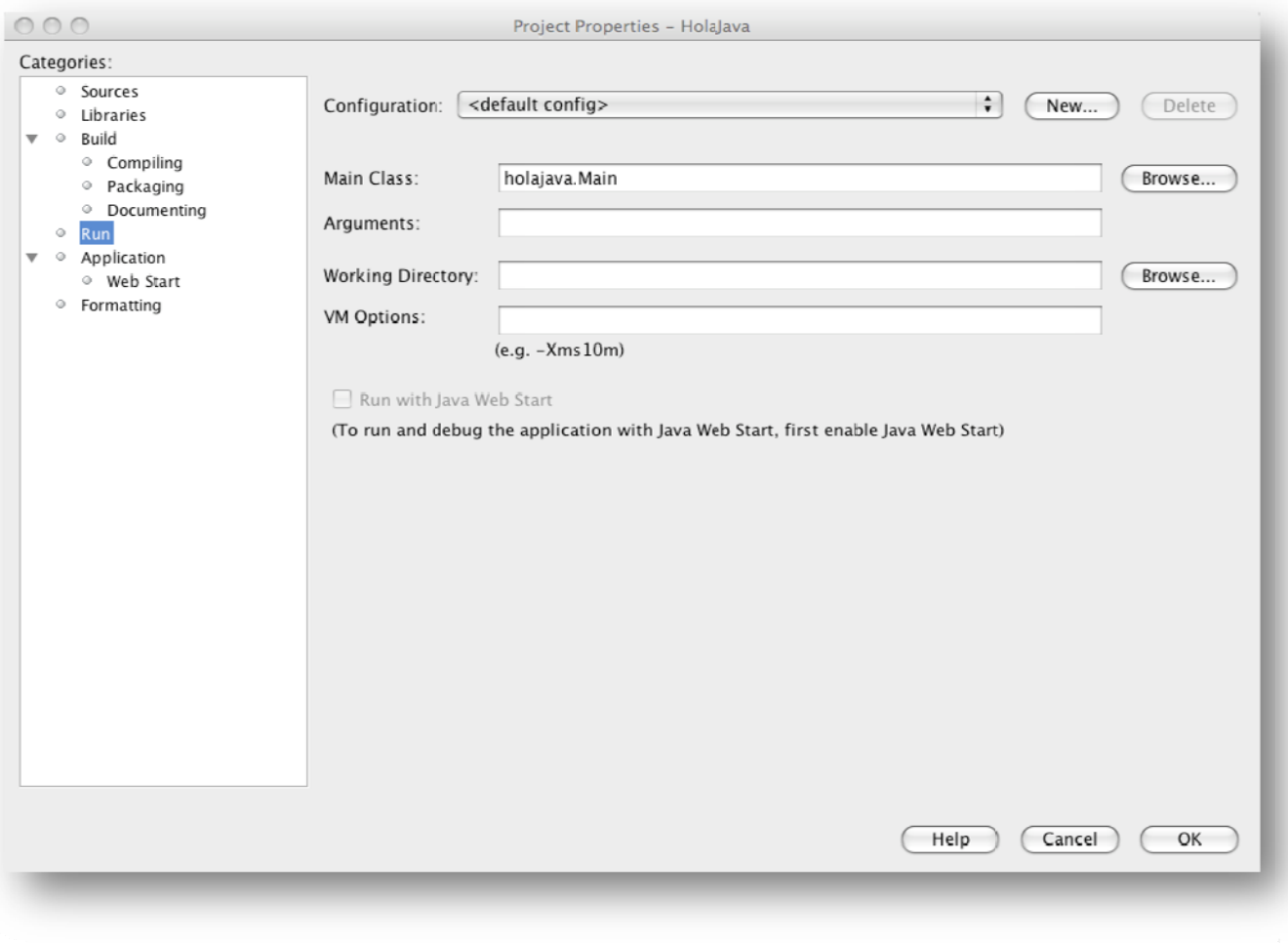


Figura 42. Ventana de opciones para ejecutar el programa.

Vemos que aparece una línea donde podemos poner los argumentos (Arguments). Escribiremos en ellos los que deseemos, tal como se muestra en la Figura 43.

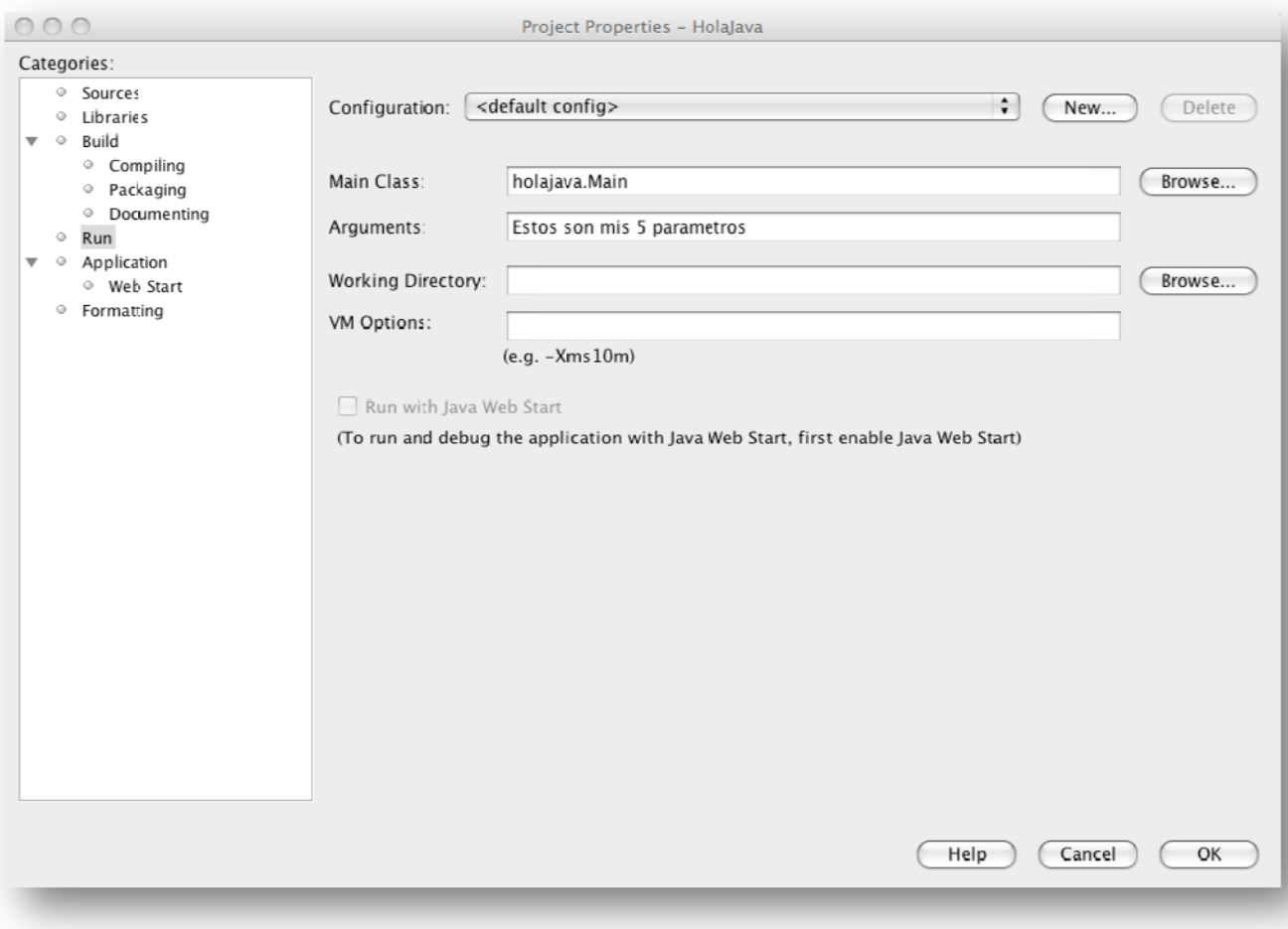


Figura 43. Ventana de propiedades de ejecución con nuestros parámetros.

Es importante tener en cuenta que estos parámetros sólo se pueden modificar en esta ventana, y no olvidarse de que están.

Ejercicio 20: Hacer un programa que pida un determinado número de parámetros. Estos deben ser de diferentes tipos numéricos. Hacer la prueba de cambiar un parámetro que debería ser numérico por una cadena cualquiera. Ver la excepción que se produce y corregir el programa para que la tenga en cuenta.

2.3.13.2 *Flujos estándar*

En los sistemas operativos modernos, unos de los elementos fundamentales son los flujos de datos (entrada y salida de datos a los distintos elementos del ordenador). Un caso particular de flujos de datos son los que permiten recibir entradas a través de la consola (teclado y pantalla del ordenador) y escribir datos en la misma; son los llamados **flujos estándar,** que estudiaremos en esta sección. Trataremos en las secciones siguientes otros flujos de datos que permiten leer y escribir en ficheros. Por último mencionar que, en muchos casos, las transmisiones de datos por Internet también se tratan como flujos de datos.

En Java, como en la mayoría de lenguajes, existen tres flujos estándar de datos:

- **System.in:** permite recibir entradas por consola (teclado en general).
- **System.out:** permite escribir en la consola (terminal o pantalla en general), y ya lo hemos utilizado de forma intensiva, por ejemplo con el método System.out.println().
- **System.err:** similar a System.in pero utilizado para la presentación de mensajes de error; en *NetBeans* los datos de System.err se ven como mensajes en rojo en pantalla.

La separación entre la salida normal (System.out) y la salida de errores (System.err) permite gestionarlas separadamente siempre que sea necesario, evitando que se mezcle la información normal con los mensajes de error.

La lectura directa del flujo System.in para recibir información es tediosa. Por este motivo, a partir de la versión 5.0 Java proporciona la clase *Scanner*, del paquete java.util, que simplifica la lectura de datos. Para ello crearemos un objeto de este tipo:

```
java.util.Scanner scan = new java.util.Scanner(System.in);
```

Una vez instanciado un objeto como este podemos usar los diferentes métodos del mismo para gestionar la entrada de datos. En general estos métodos están diseñados para tratar la entrada de datos dividiéndola en bloques separados por caracteres espaciadores, de forma similar a como hemos visto que se tratan los argumentos (sección anterior). Los caracteres espaciadores son los siguientes:

- Espacio: ‘ ‘
- Tabulación: ‘\t’
- Final de línea: '\u000A'
- Tabulación vertical: '\u000B'
- Final de formulario: '\u000C'
- Retorno de carro: '\u000D'
- Separador de fichero: '\u001C'
- Separador de grupo: '\u001D'
- Separador de campo: '\u001E'
- Separador de unidad: '\u001F'

El ejemplo siguiente usa la clase *Scanner* para implementar una calculadora que permite sumar dos números que se proporcionen por consola.:

```
public static void main(String[] args) {
  // Nombre del programa
  System.out.println("Calculadora que suma enteros");

  // Creamos una clase de scanner para simplificar la lectura de la consola
  java.util.Scanner scan = new java.util.Scanner(System.in);

  // Leemos el primer operando
  int primerOperando = scan.nextInt();
  // Leemos el operador
  String operador = scan.next();
  // Leemos el segundo operando
  int segundoOperando = scan.nextInt();

  // Comparamos el operador con el caracter de suma
  // Si coincide calculamos la suma
  if (operador.equals("+")) {
        int resultado = primerOperando + segundoOperando;
        // Presentamos el resultado
        System.out.println("Resultado: " + resultado);
  }
  // En caso contrario presentamos un mensaje de error
  else {
        System.err.println("Solo se sumar :-(");
  }
}
```

Nótese que para leer los operandos (valores enteros) hacemos uso del método `nexInt()`, mientras que para leer las cadenas usamos el método `next()`.

Ejercicio 21: Probar el programa mostrado anteriormente.

Debugar el programa y ver por dónde va pasando en función de los operandos y que tengan un formato correcto.

Cambiar el programa para que pueda trabajar con operandos en coma flotante.

Separar el programa en métodos, una dedicado a la gestión de la entrada y salida y otra a las operaciones. Utilizar enumeraciones para los operandos.

2.3.14 Gestión de errores

En la ejecución de un programa pueden producirse situaciones que alteran su flujo normal. Normalmente estas situaciones excepcionales se denominan “errores” aun cuando a veces pueden no corresponder con el concepto habitual de error (puede no haber nada erróneo en el programa).

Durante la compilación se corrigen los errores sintácticos y podemos intentar corregir los errores algorítmicos mediante el depurado (“debugging”). Sin embargo, la experiencia demuestra que casi siempre aparecen errores inesperados durante la ejecución de un programa. Java incorpora en el propio lenguaje la gestión de errores, proporcionando un marco unificado para la gestión de los mismos; para ello Java proporciona una jerarquía de clases y unos bloques de control que estudiaremos a continuación.

2.3.14.1 Jerarquía Throwable

Cuando se produce un error (una situación excepcional) en un programa Java, la máquina virtual asocia a esta situación una clase especial que permite su gestión y altera el flujo del programa (como discutiremos en la sección siguiente). Las clases proporcionadas por Java para la gestión de errores forman una jerarquía (árbol de herencia) que nace de la clase *Throwable*, tal como se muestra en la Figura 44.

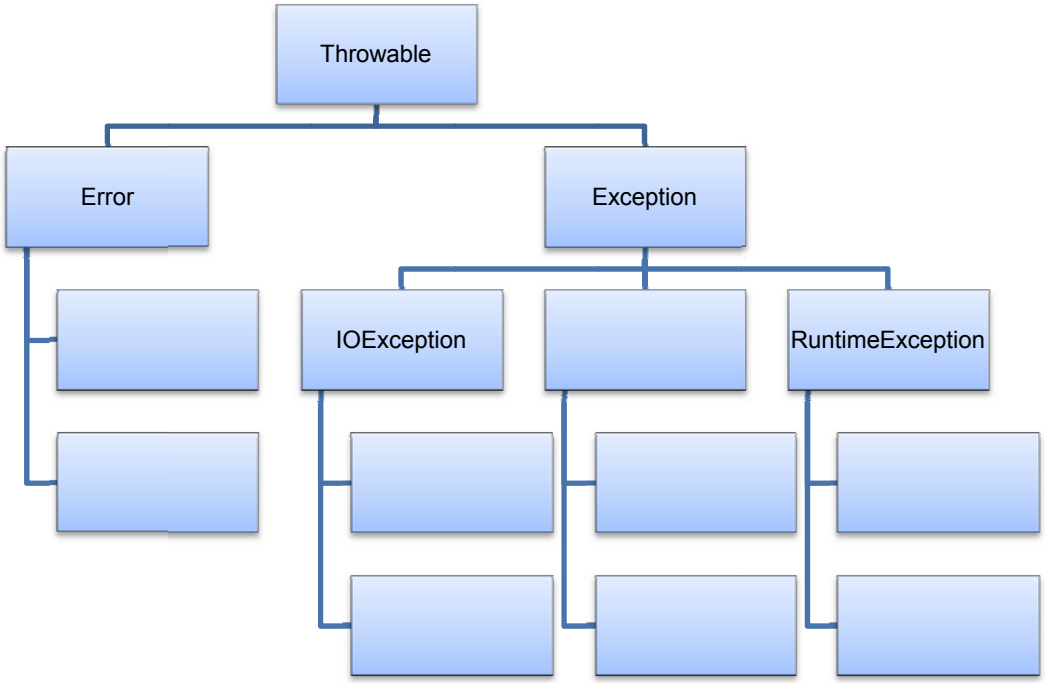


Figura 44. Jerarquía de clases de Error y Excepción.

Las dos clases hijas de *Throwable* son:

- **Error:** clase que indica un problema en la máquina virtual y obliga al programa a cerrarse
- **Exception**: es un problema del programa y el programa debe determinar si se puede solucionar y actuar en consecuencia.

Un ejemplo de error puede ser el intento de creación de una matriz que no cabe en la memoria del ordenador. En este caso, se le debe dar un mensaje al usuario para informarle, e intentar cerrar el programa procurando que no se pierda información. Los errores sólo los debe generar la máquina virtual.

Por otra parte, hay dos tipos de excepciones:

- **Excepciones en tiempo de ejecución:** son aquellas que se producen porque el programa tiene errores, por ejemplo, que no se haya comprobado un división por 0 entera, que nos hayamos salido del índice de una matriz, que estemos intentando acceder a una referencia no inicializada.
- **Excepciones comprobadas:** son aquellas que se producen por aspectos ajenos al programa. Por ejemplo que el usuario haya indicado el nombre de un fichero que no existe, o que el formato de una cadena que debería ser un número no es correcta.

En general, las excepciones en tiempo de ejecución son un problema del programador que debe corregir en el código para evitarlas, mientras que las excepciones comprobadas se producen debido a factores externos. Por este **motivo las excepciones comprobadas deben gestionarse de forma explícita,** es decir, Java requiere la inclusión en el programa de mecanismos de gestión explícitos cuando estas excepciones pueden producirse.

2.3.14.2 Captura de excepciones

Como ya hemos comentado, cuando Java detecta una situación excepcional en la ejecución de un programa, altera el flujo normal del mismo asociando una clase de la jerarquía *Throwable* al problema. Dependiendo del caso, Java puede simplemente terminar el programa en el punto donde se ha producido el problema mostrando información sobre el mismo en la consola o bien ejecutar las acciones que el programador haya especificado y continuar con el programa.

Este proceso de gestionar desde el programa (mediante sentencias `try-catch`, discutidas a continuación) una situación excepcional detectada por Java y asociada a un objeto de la clase *Throwable* se denomina **captura de excepciones**. Cuando Java detecta dicha situación excepcional en la ejecución de alguna clase, se "lanza" un objeto *Throwable*, la ejecución del programa se para, y el control se devuelve al método que ha usado la clase. Si este método no la captura (no incluye una gestión explícita de la misma mediante `try-catch`), se pasa el control al método superior que haya llamado al mismo y así sucesivamente (en términos técnicos, se sube por la pila de métodos) hasta que algún método lo captura, o se llega a la misma máquina virtual. En este último caso, ésta indica que se ha generado el error o excepción y termina la ejecución.

Para evitar un final tan drástico del programa, Java permite "capturar "estos objetos en cualquiera de los métodos de la cadena y actuar en consecuencia. Esta captura se especifica mediante las sentencias '`try`' y '`catch`' (y en determinadas situaciones puede ser necesario usar además la sentencia '`finally`'). La sintaxis más sencilla seria:

```
try {
  ...
} catch (Throwable t) {
  ...
}
```

La función de cada bloque es:

- `try`: las sentencias dentro del bloque '`try`' se intentan ejecutar por la máquina virtual. Si se produce una excepción la ejecución salta hasta el bloque '`catch`'. Si no se produce excepción dentro del bloque '`try`' se sigue con el programa después del bloque '`catch`'.
- `catch`: se encarga de gestionar la excepción que se haya lanzado. Esta excepción tiene que ser del tipo esperado por '`catch`' o hija del mismo (debe ser cierto el operador '`instanceof`'). Si es así, se ejecuta el código dentro del bloque. En caso contrario se pasa la excepción al método superior.

Puede ser que el código pueda generar diferentes tipos de excepciones, y cada una de ellas se deba gestionar de una manera diferente. En este caso, es posible anidar bloques '`catch`':

```
try {
  ...
} catch (NumberFormatException nfe) {
  ...
} catch (RuntimeException rte) {
  ...
}
```

La gran ventaja de esta opción es que podemos agrupar código dentro del bloque 'try', aunque es importante gestionar correctamente cada caso distinto para no tener problemas.

A modo de ejemplo podemos modificar el programa de la calculadora para poder tener en cuenta las excepciones cuando se introduce un parámetro con formato erróneo.

```
public static void main(String[] args) {
  // Nombre del programa
  System.out.println("Calculadora");

  try {
        // Creamos una clase de scanner para simplificar la lectura de la consola
        java.util.Scanner scan = new java.util.Scanner(System.in);
        // Leemos el primer operando
        int primerOperando = scan.nextInt();
        // Leemos el operador
        String operador = scan.next();
        // Leemos el segundo operando
        int segundoOperando = scan.nextInt();
        // Leemos el resto de la línea
        // Comparamos el operador con el caracter de suma
        if (operador.equals("+")) {
              // Si coincide calculamos la suma
              int resultado = primerOperando + segundoOperando;
              // Presentamos el resultado
              System.out.println("Resultado: " + resultado);
        } else {
              // Como solo sabe sumar mostramos un mensaje de error
              System.out.println("Solo se sumar :-(");
        }
  } catch (java.util.InputMismatchException ime) {
        System.err.println("Error de formato");
        System.err.println("Operador + Operador");
  }
}
```

De esta forma podemos avisar al usuario sin tener que parar el programa de forma anómala.

Ejercicio 22: Probar el nuevo programa de la calculadora y ver qué ocurre cuando se produce la excepción.

Depurar el programa paso a paso.

2.3.14.3 Lanzamiento de excepciones

Las excepciones no son sólo algo que puede gestionarse cuando aparecen, sino que pueden usarse como herramienta para controlar la ejecución de nuestras propias clases. Si se produce una situación inesperada en una clase escrita por nosotros, podemos generar una excepción y lanzarla dejando su gestión para el usuario de la clase.

Por ejemplo, estamos realizando un conjunto de cálculos matemáticos y nos encontramos que hay una indeterminación 0/0 al procesar los datos de un fichero. Según nuestra definición del fichero esto no se debería producir nunca, por lo que generamos una excepción para indicar que hay un problema en el fichero, dejando la gestión del problema al método que nos haya pasado el fichero. Para ello podemos generar por ejemplo una excepción del tipo '`ArithmeticException`'.

Para lanzar dicha excepción se utiliza la cláusula '`throw`':

```
throw new ArithmeticException("Parametro produce indeterminación 0/0");
```

Generar una excepción es relativamente sencillo si ya se dispone de una que se adecua a nuestras necesidades:

- Se busca la clase de excepción más adecuada (ver http://java.sun.com/javase/6/docs/api/)
- Se crea un objeto de esta clase, idealmente con una descripción de la misma
- Se lanza la excepción como se ha indicado

Si se genera una excepción, se para la ejecución del código dentro del método en que se ha producido y se pasa el control al método que lo ha llamado. Esto implica que no hay que devolver ningún valor especial cuando se ha producido una excepción.

Si la excepción a lanzar es comprobada (no es hija de RuntimeException), el método que la lance deberá incluir en su definición el tipo de excepción que genera mediante la cláusula '`throws`'.

```
public void leerDatos() throws Exception {
  ...
  throw new Exception("Parametro en fichero produce indeterminación 0/0");
  ...
}
```

En este caso será necesario capturar la excepción desde cualquier punto donde se use este método.

Ejercicio 23: Crear un método que genere una excepción no comprobada (por ejemplo haciendo una división de un entero por 0) y depurar para ver el efecto.

2.3.14.4 Terminación de capturas (finally)

En determinados casos al ejecutar un método es necesario asegurarse de que aunque se produzcan excepciones se realicen siempre algunas acciones de "limpieza" (*cleanup*), como guardar datos importantes, actualizar el estado del método o desbloquear un recurso que se haya estado usando. Para ello puede añadirse un bloque '`finally`' que se ejecuta siempre, se hayan producido excepciones o no en el bloque '`try`'.

```
try {
  //1
  Puede lanzarse una excepción
  //2
} catch (Exception e) {
  //3
  Presentamos el error en consola
  //4
} finally {
  //5 realizamos acciones de "limpieza" (cleanup)
  ...
}
//6
```

En este caso, se pueden producir tres situaciones distintas:

- El código no lanza ninguna excepción. En este caso, se ejecuta primero el bloque '`try`', seguido del bloque '`finally`' y se continua la ejecución del programa de forma normal. El programa pasaría por los puntos 1, 2, 5 y 6.
- El código lanza una excepción reconocida por '`catch`'. En este caso se ejecuta el código dentro del bloque '`try`' hasta que salta la excepción y seguidamente se salta a la cláusula '`catch`'. Ésta puede lanzar una excepción, en cuyo caso se ejecutaría directamente el bloque '`finally`' siguiendo la secuencia 1, 3, 5 y devolviendo la nueva excepción al método que haya llamado al actual; o bien puede no lanzarla realizando la secuencia 1, 3, 4, 5 y 6.
- Si el código lanza una excepción no reconocida por 'catch', se ejecuta de todas formas el bloque '`finally`', y se devuelve la excepción al método que haya llamado al actual. En este caso solo pasa por los puntos 1, 3 y 5.

También es posible utilizar la cláusula '`finally`' sin la '`catch`':

```
Capturamos un recurso
try {
  Puede lanzarse una excepción
} finally {
  Realizamos acciones de "limpieza"
}
```

Llegados a este punto puede aparecer la pregunta de ¿que es más optimo en algunos casos, hacer comprobaciones de posibles errores antes de que se produzcan o gestionar excepciones no comprobadas? Teniendo en cuenta que una comprobación puede ser un simple '`if`', y una excepción implica la creación de un objeto y romper dramáticamente el flujo de programa, la primera opción es la más acertada siempre que sea posible.

2.3.15 Ficheros

Existen infinidad de casos en que nos puede interesar no tener que introducir la información directamente a través de un teclado, sino por medio de un fichero que esté guardado en el disco duro. Esta forma de trabajar nos será especialmente interesante cuando tenemos que introducir una gran cantidad de información. Por ejemplo si tenemos que introducir una matriz de datos.

El acceso a los datos de un fichero se trata desde Java como un flujo de datos, similar a los flujos estándar que ya hemos estudiado en la sección 2.3.13 pero que en lugar de recibir información o generarla de forma interactiva lo hace desde un soporte de almacenamiento (un disco duro, un CD, un disco USB, etc.). De esta forma, disponemos de las mismas herramientas para trabajar con un fichero que con un flujo estándar.

2.3.15.1 Escritura en ficheros de texto

Para guardar información de tipo texto en un fichero puede usarse la clase '`FileWriter`'. El primer paso es crear el acceso al fichero y la forma más simple es a través de esta clase:

```
FileWriter fw = new FileWriter("MiPrimerFichero.txt");
```

El primer inconveniente que encontramos es que un '`FileWriter`' sólo proporciona métodos muy básicos (de bajo nivel) para la escritura de datos en el fichero (no incluye métodos del estilo '`print`' o '`println`' que simplifiquen la escritura). Para disponer de métodos más prácticos podemos un flujo de datos más fácil de usar mediante la clase '`PrintWriter`' que proporciona dichos métodos. Basándose en el acceso al fichero proporcionado por '`fw`' podemos crear el flujo de datos como sigue:

```
PrintWriter pw = new PrintWriter(fw);
```

A partir de este punto podemos trabajar sobre el flujo '`pw`' como hacíamos con el flujo estándar '`System.out`'. A modo de ejemplo, para escribir una línea usaremos:

```
pw.println("Escribo en el fichero.");
```

Una vez hayamos terminado de escribir toda la información en el fichero debemos "cerrar" el flujo de datos con el mismo usando:

```
fw.close();
```

Si no realizamos este cierre puede perderse parte de la información que hayamos escrito en el fichero.

Es importante recalcar que la información que se genere se guardará en un fichero de texto, con lo que esto supone respecto a la codificación de caracteres (ver sección 2.3.4.3) y que por lo tanto hay que tener cuidado con temas como los acentos y la codificación de caracteres no ASCII en general. **No deben usarse nunca estas clases para guardar datos binarios en un fichero (los datos son modificados según el mapa de caracteres).**

Ejercicio 24: Escribir un programa que cree un fichero y escriba en él un texto.

Modificar el programa para que reciba el nombre del fichero a través de los argumentos.

Modificar el programa para que en lugar de escribir un texto cualquiera, escriba en el fichero la entrada por consola.

2.3.15.2 Lectura de ficheros de texto

Ya sabemos cómo guardar datos de texto en un fichero y ahora discutiremos cómo leerlos. De momentos seguiremos el mismo método que con la entrada a través de consola (System.in). En primer lugar, como en el caso de la escritura, crearemos un flujo de entrada de tipo fichero '`FileReader`', que abra el fichero creado anteriormente.

```
FileReader fr = new FileReader("MiPrimerFichero.txt");
```

Seguidamente, tenemos que procesar el fichero para ver que contiene. En nuestro caso, la información está escrita en formato texto, y cada dato está separado por espacios. Así pues, podemos utilizar la clase 'Scanner' para procesar el fichero:

```
Scanner scn = new Scanner(fr);
```

Una vez creada, es posible leer los diferentes datos escritos, de forma semejante a como hacíamos con el flujo estándar de entrada:

```
while(scn.hasNext()) {
  System.out.println(scn.next());
}
```

Finalmente hay que cerrar el fichero:

```
fr.close();
```

En este caso, la salida por consola será:

```
Escribo
en
el
fichero.
```

Es importante indicar que la clase Scanner no "sabe" qué contiene el fichero, se limita a dividir la información en base a los caracteres separadores que hayamos definido, como discutiremos más adelante.

Ejercicio 25: Escribir un programa que escriba un vector de número del 1 al 10, cada uno en una línea.

Escribir un segundo programa que lea los datos del fichero y los guarde en un vector.

Modificar el programa de escritura para que ponga en la primera línea el número de datos que tiene el vector.

Modificar el programa de lectura para lea la primera línea y en función de ésta, lea el resto de datos que contiene el fichero y los guarde en un vector.

Debugar los diferentes programas.

2.3.15.3 Modificar el separador de datos

Si usamos la clase *Scanner* para la lectura de datos de un fichero de texto, ya hemos visto que muchos de sus métodos están diseñados para tratar la entrada de datos dividiéndola en bloques separados por caracteres espaciadores. En según qué casos, nos puede interesar utilizar otro tipo de separador, por ejemplo para usar ficheros con valores separados por coma (*Comma Separated Values*, CSV), que son leídos de forma sencilla por OpenOffice.org Calc o Excel.

En estos ficheros los valores están separados por ";" o por "," y un salto de línea (se escribe como "\n") indica el fin de una fila de datos. Por ejemplo, para guardar en un fichero CSV la matriz:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

escribiremos en el fichero la siguiente cadena de caracteres:

```
pw.print("  1;  2;  3\n  4;  5;  6\n  7;  8;  9\n");
```

Si visualizamos el fichero así creado con un editor de texto veremos lo siguiente:

```
1;  2;  3
4;  5;  6
7;  8;  9
```

Para leer la matriz desde el fichero CSV con un objeto Scanner tenemos que cambiar su comportamiento por defecto, puesto que tiene establecidos como separadores por defecto los caracteres espaciadores, tal como se indicaba en la sección 2.3.13. Sin embargo, en nuestro caso nos interesa utilizar como separador de datos el ';'. Para ello deberemos modificar el llamado "delimitador" (el carácter o caracteres que Scanner usa como separadores de datos).

Empezaremos por establecer el punto y coma `';'` como separador. Para ello usaremos las siguientes sentencias; creamos un patrón (objeto `Pattern` que describe como es el separador) y se le indicamos a la clase Scanner que use este patrón como delimitador mediante el método '`useDelimiter`':

```
Pattern csvDelimiter = Pattern.compile(";");
scn.useDelimiter(csvDelimiter);
```

Un vez creado, leemos el fichero utilizando el bucle:

```
for (int i = 0; scn.hasNext(); i++) {
 System.out.print(i + ":\t\t");

 String str = scn.next();
 System.out.println(str);
}
```

Ejercicio 26: usar el código indicado para hacer un programa que lea el fichero creado en formato CSV con la matriz indicada más arriba.

Si ejecutamos el programa indicado en el ejercicio, la salida obtenida es:

```
0:              1
1:              2
2:              3
  4
3:              5
4:              6
  7
5:              8
6:              9
```

El programa no está leyendo correctamente los elementos individuales de la matriz. Vemos que el "4" y el "7" no se escriben como uno podría esperar en la segunda columna, la columna de la derecha no tiene números correlativos y hay espacios vacíos. El problema en este caso es que después del "3" y el "6" no hay un ";" sino un salto de línea "\n"; como hemos indicado que el separador de datos sólo es ";" al leer el tercer elemento obtenemos "3\n4" que al imprimirlo por pantalla produce el resultado indicado (escribe el 3, salta línea y escribe los dos espacios y el 4 que hemos puesto en el fichero).

Para leer correctamente los elementos de la matriz debemos especificar que los separadores de datos son dos: ";" y "\n". Para ello escribiremos:

```
Pattern csvDelimiter = Pattern.compile("[;\n]");
scn.useDelimiter(csvDelimiter);
```

Si volvemos a ejecutar el programa después de este cambio obtenemos:

```
0:                  1
1:                  2
2:                  3
3:                  4
4:                  5
5:                  6
6:                  7
7:                  8
8:                  9
```

Que nos separa individualmente los elementos de la matriz.

Más en general, para definir separadores de datos para la clase Scanner podemos crear patrones usando las llamadas **expresiones regulares**[2]. Las expresiones regulares son una forma muy general de especificar patrones de caracteres, y constituyen casi un lenguaje de programación en sí mismas. Ya hemos visto un par de ejemplos:

- ";" → específicamente el carácter de punto y coma
- "[;\n]" → el carácter de punto y coma o el carácter de fin de línea (las llaves [ ] indican una lista de posibilidades).

Otro ejemplo sencillo es el siguiente:

- [\t \n\f\r\x0B;] → cualquiera de los caracteres de espaciado estándar (ver sección 2.3.13.2). Esta expresión puede sustituirse por la abreviatura estándar "\\s".

Ejercicio 27: Modificar el programa de lectura para utilizar el delimitador '[\\s;]' .

En este caso el resultado es:

```
0:
1:                  1
2:
3:
4:                  2
5:
6:
7:                  3
8:
9:
10:                  4
11:
12:
13:                  5
14:
15:
```

[2] http://es.wikipedia.org/wiki/Expresi%C3%B3n_regular

```
16:             6
17:
18:
19:             7
20:
21:
22:             8
23:
24:
25:             9
```

Vemos que el número de elementos se multiplica, debido a que si encuentra dos espacios juntos o un espacio y un punto y coma, considera que tiene que haber un elemento intermedio.

En este caso el programa tampoco hace lo que deseamos, pues querríamos que si encuentra un salto de línea, o un '`;`' entre cualquier número de espacios, que considere que es un único delimitador. Para ello podemos utilizar el llamado **numerador** de expresiones regulares:

- "`\\s*[\n;]\\s*`" → la expresión '\\s*' indica "cualquier número de caracteres espaciadores". La expresión global indica que si encontramos un '`\n`' o un '`;`' entre cualquier número de caracteres espaciadores, es un delimitador.

Ejercicio 28: Modificar el programa de lectura para utilizar el delimitador "`\\s*[\n;]\\s*`".

En este caso, el resultado pasa a ser:

```
0:              1
1:            2
2:            3
3:            4
4:            5
5:            6
6:            7
7:            8
8:            9
```

En este caso, el resultado es casi correcto, el único "problema" son los espacios delante del 1 de la primera línea. En este caso tenemos dos opciones. La primera es modificar el delimitador para tener en cuenta que el primer carácter de la línea puede tener un espacio. La segunda es simplemente no poner el espacio al crear el fichero. En nuestro caso, optamos por la primera solución:

- "(\\s*[\n;]\\s*)|(\\A\\s*)" → en este caso, especificamos que podemos tener dos delimitadores. El anterior, basado en el salto de línea y el punto y coma "\\s*[\n;]\\s*" y uno nuevo, que es, si estamos al principio del todo del fichero "\\A", un número indeterminado de espacios "\\A\\s*". Para ello agrupamos ambos delimitadores entre paréntesis '()' y los separamos con un operador de unión '|'.

Ejercicio 29: Modificar el programa de lectura para utilizar el delimitador "(\\s*[\n;]\\s*)|(\\A\\s*)".

Volvemos a obtener una separación de los elementos individuales de la matriz:

```
0:              1
1:              2
2:              3
3:              4
4:              5
5:              6
6:              7
7:              8
8:              9
```

Hemos vuelto a recuperar la separación inicial obtenida con "[\n;]" pero con la ventaja de que podemos refinar aun más esta expresión para diferenciar entre filas y columnas, como veremos en la siguiente sección.

2.3.15.4 Utilizar varios separadores de datos

Para poder identificar separadamente filas y columnas en la lectura del fichero dividiremos el problema en dos partes: el primero es identificar las filas, y el segundo las columnas dentro de cada fila. Como entrada para este ejemplo utilizaremos la matriz irregular:

$$\begin{pmatrix} 11 & 12 & 13 & & \\ & & & & \\ 31 & & 33 & 34 & 35 \\ 41 & 42 & 43 & & \end{pmatrix}$$

Que escribiremos en un fichero CSV mediante la siguiente cadena de caracteres:

```
"11; 12; 13\n\n31; ;33;34;35\n41;42;43"
```

En este caso, lo que hacemos es trabajar con la clase Scanner para separar la filas, y la clase Pattern para separar las celdas dentro de una fila. Para ello partiremos de uno de los delimitadores definidos en la sección anterior:

- (\\s*\n\\s*)

Así definido este delimitador no es apropiado para nuestro caso, ya que el grupo espacio '\s' también incluye el salto de línea, por lo que cualquier agrupación de saltos de línea será considerada como un delimitador. Así pues, tenemos que adaptar el delimitador para sólo se tenga en cuenta un salto de línea. Esto lo conseguimos con la expresión regular:

- (\\s&^\n)*\n(\\s&^\n)*: En este caso el delimitador salto de línea puede tener como prefijo o sufijo un carácter de espacio y que no sea un salto de línea (*espacio* '\\s' *y* '&' que *no* '^' sea un *salto de línea* '\n').

Ejercicio 30: Modificar el programa de lectura para utilizar el delimitador '(\\s&^\n)*\n(\\s&^\n)*'.

Con ello se han separado las diferentes líneas:

```
11;         12;         13

31;         ;33;34;35
41;42;43
```

El siguiente paso es separar las diferentes celdas de la fila. Para ello creamos un 'Pattern' semejante al último del apartado anterior, pero que sólo tiene en cuenta el ';':

- (\s*;\s*)|(\A\s*)

Y aprovechamos el método `splits` para obtener las diferentes cadenas. La rutina para leer el fichero se muestra a continuación:

```
/**
 *
 * @param nombreFichero nombre del fichero de entrada
 */
static double[][] leerFicheroCSV(String nombreFichero) {
    // Definimos la variable FileReader y la inicializamos a null
    FileReader fr = null;
    // Definimos la variable filas que es un objeto de tipo ArrayList que
    // contiene vectores de tipo double[] que serán cada una de las filas.
    // trabajamos con un ArrayList y no con un vector por que no sabemos
    // el número de filas del fichero.
    ArrayList<double[]> filas = new ArrayList<double[]>();

    // Abrimos un bloque try para intentar leer la información del fichero
    try {
          // Abrimos el fichero con el objeto fr
          fr = new FileReader(nombreFichero);

          // Creamos un Scanner tomado como base fr
          Scanner scn = new Scanner(fr);

          // Definimos el delimitador de fila, que es cualquier número de
          // espacios, un ; y cualquier número de espacios o bien, al
          // inicio de la fila un espacio
          Pattern delimitadorColumna =
                Pattern.compile("(\\s*;\\s*)|(\\A\\s)");

          // Establecemos como delimitador de línea el \n y admitimos que
          // puedan haber varios \n.
          Pattern delimitadorLinea =
                Pattern.compile("(\\s&(^\\n))*\\n(\\s&(^\\n))*");

          // Definimos que el Scanner sólo se utilizará para leer las
          // líneas
          scn.useDelimiter(delimitadorLinea);

          for (int i = 0; scn.hasNext(); i++) {
                // Vamos leyendo las líneas una a una
                String line = scn.next();

                // Si la línea no esta vacia
                if (line != null) {
                      // Buscamos los diferentes elementos de las columnas
                      String[] cadenaFila = delimitadorColumna.split(line);

                      // Creamos un vector para esta fila con el tamaño de
                      // cadenaFila
                      double[] fila = new double[cadenaFila.length];

                      // Vamos convirtiendo cada uno de las columnas a formato
                      // double
                      for (int j = 0; j < cadenaFila.length; j++) {
                            // El bloque try-catch es necesario por si hay un
                            // error al realizar la conversión de texto a
                            // número
                            try {
                                  fila[j] =
                                        Double.parseDouble(cadenaFila[j]);
                            } catch (ParseException ex) {
                                  // Si hay algún problema al realizar la
                                  // conversión, se informa y se guarda en el
                                  // vector fila un NaN.
```

```
                                    Logger.getLogger(
                                          LeerFichero.class.getName()).log(
                                          Level.SEVERE, null, ex);
                                    fila[j] = Double.NaN;
                              }
                        }

                        // Introducimos la fila en la lista de filas
                        filas.add(i, fila);
                  }
            }
      } catch (FileNotFoundException ex) {
            // Si no se ha encontrado el fichero podemos realizar algún tipo de
            // operación antes de salir del catch
            Logger.getLogger(
                  LeerFichero.class.getName()).log(
                  Level.SEVERE, null, ex);
      } catch (IOException ex) {
            // Si hay cualquier tipo de excepción lo informamos
            Logger.getLogger(
                  LeerFichero.class.getName()).log(
                  Level.SEVERE, null, ex);
      } finally {
            // Chequeamos primero que se haya creado el FileReader y en ese
            // caso, procedemos a cerrarlo
            if (fr != null) {
                  try {
                        fr.close();
                  } catch (IOException ex) {
                        Logger.getLogger(
                              LeerFichero.class.getName()).log(
                              Level.SEVERE, null, ex);
                  }
            }
      }

      // Una vez tenemos todas las filas, creamos una vector de vectores con
      // el número de filas que contiene el fichero.
      double[][] A = new double[filas.size()][];

      // Traspasamos el contenido de la lista a la matriz.
      for (int i = 0; i < A.length; i++) {
            A[i] = filas.get(i);
      }

      // Se devuelve la matriz
      return A;
}
```

Ejercicio 31: modificar el método para que en lugar de mostrar la matriz por pantalla, la guarde en un fichero.

Depurar los programas.

2.3.15.5 *Escritura de datos con formato*

En los ejemplos discutidos en las secciones anteriores hemos tratado los datos numéricos sin formato, es decir, al tratar con un valor numérico (entero, coma flotante) Java decide como se escribe o lee. Frecuentemente, sin embargo, es necesario tener control sobre cómo se escriben o leen los datos numéricos; por ejemplo, podemos querer controlar cuantos decimales se incluyen al escribir una cifra en coma flotante o escribir los enteros con un número fijo de cifras, añadiendo ceros al principio, para encolumnarlos correctamente.

En estos casos puede usarse la clase `java.text.NumberFormat` que proporciona herramientas para este tipo de operaciones. Veamos un ejemplo de su uso:

```
NumberFormat formater = NumberFormat.getInstance();
formater.setMinimumFractionDigits(5); // Fijamos el número mínimo de decimales en
                                      // los números de coma flotante
formater.setMaximumFractionDigits(5); // Fijamos el número máximo de decimales en
                                      // los números de coma flotante
formater.setMinimumIntegerDigits(5); // Fijamos el número mínimo de cifras en
                                     // los números enteros
formater.setMaximumIntegerDigits(5); // Fijamos el número máximo de cifras en
                                     // los números enteros

System.out.println("double sin formato: " + 1./3. );
System.out.println("double con formato: " + formatter.format(1./3.) );
System.out.println("int sin formato: " + 3 );
System.out.println("int con formato: " + formatter.format(3) );
```

El resultado de este fragmento de código es:

```
double sin formato: 0.3333333333333333

double con formato: 00.000,33333

int sin formato: 3

int con formato: 00.003,00000
```

Nótese que las cifras están formateadas según la especificación local del lenguaje ("locale"), en este caso las especificaciones del castellano donde la coma decimal es "," i los millares se separan con un punto. Puede cambiarse la especificación local al crear la instancia del objeto:

```
NumberFormat formater = NumberFormat.getInstance(Locale.ENGLISH);
```

En este caso el resultado es

```
double sin formato: 0.3333333333333333

double con formato: 00,000.33333

int sin formato: 3

int con formato: 00,003.00000
```

Tambien puede eliminarse la agrupación de cifras por múltiplos de millar, que no suele usarse en un contexto científico:

```
formatador.setGroupingUsed(false);
```

y en este caso el resultado es

```
double sin formato: 0.3333333333333333

double con formato: 00000.33333

int sin formato: 3

int con formato: 00003.00000
```

Finalmente, la clase `NumberFormat` también puede usarse para leer cifras con formato. El siguiente ejemplo ilustra su uso básico:

```
 // Definimos un formateador con
NumberFormat formateador = NumberFormat.getInstance();

// Scanner de lectura
Scanner scn = new Scanner(System.in);

// Pedimos y leemos un valor de coma flotante por consola
System.out.print("Dame un número en coma flotante " );
String input = scn.next();
double valor;

try{

  valor = formateador.parse(input).doubleValue();
  System.out.println("El valor es: " +  valor);

}catch(ParseException pe){

  System.err.println("No puedo leer el valor");

}
```

3 Métodos numéricos

3.1 Introducción

El término *métodos numéricos* (o también cálculo numérico, o análisis numérico) hace referencia al conjunto de métodos que permiten obtener el valor o valores numéricos correspondientes a la formulación matemática de la solución de un problema dado, todo ello en un número finito de pasos y con una precisión arbitraria[3].

Ejemplo 3: dada una variable aleatoria con distribución normal N(0,1), ¿cuál es la probabilidad a priori de obtener un valor en el intervalo [0,1]? La respuesta puede formularse analíticamente como:

$$P(t \in [0,1]) = \frac{1}{\sqrt{2\pi}} \int_0^1 e^{-\frac{1}{2}x^2} dx$$

Sin embargo, esta integral definida no tiene solución analítica y por lo tanto para evaluar esta probabilidad debemos recurrir a métodos de integración numérica como los presentados en la Sección 3.7.

Un **algoritmo** es una secuencia finita de pasos basados en las operaciones disponibles (las operaciones aritméticas básicas u otros más complejas) que permiten la resolución de un problema.

La definición de algoritmos para la solución de problemas numéricos se remonta a la antigüedad y han sido ampliamente usados antes del advenimiento del ordenador o las calculadoras[4]. Por ejemplo, el papiro de Ahmes[5], fechado entre el 2000 y el 1800 a. C., contiene entre otros algoritmos un método para la resolución de una ecuación lineal sencilla. Sin embargo, en la actualidad, el uso de ordenadores ha permitido su aplicación masiva a todo tipo de problemas complejos, de forma que actualmente su aplicación es muy amplia en casi todos los campos de la ciencia y la ingeniería.

[3] La definición formal del análisis numérico es el estudio de métodos constructivos en análisis matemático

[4] http://www.britannica.com/EBchecked/topic/422388/numerical-analysis/235497/Historical-background

[5] http://es.wikipedia.org/wiki/Papiro_de_Ahmes

La lista siguiente resume los tipos de métodos más usuales que se encuentran en los tratados de análisis numérico:

- Solución de sistemas de ecuaciones lineales y no lineales
- Interpolación y extrapolación
- Integración y derivación de funciones
- Determinación de raíces (ceros) de funciones
- Maximización y minimización de funciones (uni y multidimensionales)
- Determinación de valores y vectores propios
- Cálculo de transformadas de Fourier
- Aproximación de funciones
- Resolución de ecuaciones diferenciales ordinarias y en derivadas parciales

Cabe remarcar, sin embargo, que más allá de estas categorías de “uso general”, existen diversos tipos más especializados de métodos numéricos. Como ejemplos destacados cabe citar los métodos de elementos finitos usados en cálculo estructural y dinámica de fluidos o los métodos de Monte Carlo usados en la simulación de procesos físicos.

Dado el papel de herramienta fundamental que juegan actualmente los métodos numéricos, esta parte de la asignatura de informática pretende proporcionar al alumno unos conocimientos básicos sobre su uso en física e ingeniería, familiarizándolo con los conceptos fundamentales y su aplicación práctica. Con este fin, los contenidos de esta sección se centran en la presentación simplificada de un número limitado de métodos (sistemas de ecuaciones lineales, interpolación, integración y aproximación de funciones) con el objetivo de proporcionar al alumno la oportunidad de analizar algunos algoritmos sencillos, adecuados al nivel de un primer curso de carrera, y realizar implementaciones propias de los mismos. Al mismo tiempo, se propone también el uso de librerías numéricas que implementan algoritmos más sofisticados para familiarizar al alumno con el uso de las mismas, permitiéndoles además resolver problemas más complejos que los resolubles con las implementaciones sencillas realizables al nivel de este curso.

Finalmente, nos gustaría insistir vehementemente en que la correcta aplicación de los métodos numéricos no es nunca una cuestión puramente informática. La comprensión de los fundamentos matemáticos de los métodos usados, y por lo tanto de sus limitaciones y rendimiento, es esencial para su correcto uso. Hemos intentado incluir en estos apuntes una descripción matemática sencilla pero al mismo tiempo suficiente de los métodos presentados para que el alumno pueda llegar a esta correcta comprensión antes de lanzarse a implementarlos.

3.2 Conceptos previos

3.2.1 Precisión de la representación en coma flotante

En la Sección 2.3.4 se han descrito los tipos de datos que el lenguaje Java incluye, en particular la representación de números en tipos de coma flotante. Este tipo de representación es la que se usa mayoritariamente en los cálculos científicos y de ingeniería, y es importante tener en cuenta sus características y limitaciones. Específicamente, debe tenerse en cuenta que la representación en coma flotante trabaja con un número limitado de dígitos y que los errores de representación que este hecho conlleva pueden tener consecuencias indeseadas en los cálculos. Veamos algunos ejemplos[6].

3.2.1.1 Pérdida de precisión en las operaciones

Si realizamos una operación aritmética (suma, resta, multiplicación o división) entre dos números representados en coma flotante el resultado no será en general exacto sino que contendrá un error de redondeo. Por ejemplo, al realizar una suma de dos números, debido a la representación en coma flotante el resultado final se redondeará alrededor del número de dígitos significativos del mayor de ellos, perdiéndose algunos decimales en el proceso.

Veamos un caso práctico. Supongamos que queremos sumar 123456.7 y 101.7654. En representación de coma flotante separaremos exponente y mantisa de la forma:

123456.7 → e=5; m=1.234567

101.76547 → e=2; m=1.017654

Desplazando los dígitos para poder sumar con el mismo exponente obtenemos

123456.7 → e=5; m=1.234567

101.76547 → e=5; m=0.001017654

El resultado exacto de la suma será entonces

123558.4654 → e=5; s=1.235584654

Sin embargo, el resultado final debe almacenarse en una variable de coma flotante, por lo que supongamos que se redondeará la mantisa al número de dígitos correspondiente:

123558.4654 → e=5; s=1.2355847

Vemos, por lo tanto, que el hecho de usar la representación de coma flotante introduce errores en las operaciones.

Ejercicio 32: compruébese este resultado ejecutando el siguiente fragmento de código

```
float a= 123456.7F;
float b= 101.7654F;
System.out.println("a= " + a);
System.out.println("b= " + b);
System.out.println("a+b= " + (a+b));
```

¿Qué ocurre si se cambia la declaración de las variables *a* y *b* de *float* a *double*? ¿Qué ocurre si el orden de

[6] http://en.wikipedia.org/wiki/Floating_point

magnitud de las dos variables muy diferente? Probad por ejemplo variables de tipo *double* con a=*1e10* y *b=1e-1* y luego a=*1e10* y *b=1e-10.*

En general, para un ordenador ideal que usa variables de coma flotante con mantisas de *t* dígitos puede decirse que los errores de las operaciones están acotados según las siguientes expresiones (véase Bonet (1996)):

$$\left.\begin{array}{rcl} Err(a \pm b) & = & (a \pm b)(1 + \Phi \cdot 10^{-t}) \\ Err(ab) & = & ab(1 + \Phi \cdot 10^{-t}) \\ Err\left(\frac{a}{b}\right) & = & \frac{a}{b}(1 + \Phi \cdot 10^{-t}) \end{array}\right\} donde\ 0 \leq \Phi \leq 5$$

3.2.1.2 *No representabilidad exacta de valores expresados en formato decimal*

Además de los errores introducidos por las operaciones aritméticas, hay que tener en cuenta que aunque en nuestros programas expresemos los valores de trabajo en formato decimal (por ejemplo 0.1234) estos valores se representan internamente en el ordenador en formato binario. El simple hecho de traducir un número en formato decimal a formato binario de coma flotante puede introducir ya un error, puesto que el número puede no ser exactamente representable en este formato.

Por ejemplo, tomemos el valor 0.1f; su representación en coma flotante (*float*) binaria será

0.1 → e = −4; s = 11001100110011001100110

Sin embargo, esta representación traducida a decimal corresponde exactamente a 0.100000001490116119384765625. De la misma forma, cuando el número 0.01 se representa en coma flotante (*float*) la traducción decimal exacta de su representación es: 0.009999999776482582092285156250. En ambos casos se ha traduce a coma flotante el número decimal con la representación binaria más próxima, pero debido que esta representación no es exacta, resultará que el resultado de la operación 0.1*0.1 no coincidirá con la representación de 0.01.

Ejercicio 33: compruébese lo anterior ejecutando el siguiente fragmento de código. Nótese que el resultado de la operación a*a-b no es cero.

```
float a= 0.1F;
float b= 0.01F;
System.out.println("a= " + a);
System.out.println("b= " + b);
System.out.println("a*a-b= " + (a*a-b));
```

3.2.1.3 *Comparación de valores en sentencias "if"*

Debido a los problemas descritos en las secciones anteriores (errores de redondeo de las operaciones y no representatividad de valores expresados en decimal) deben evitarse las sentencias *if* donde se comparan variables de coma flotante de la forma:

```
if( x == y) {
   Codigo a ejecutar
}
```

Debido a los efectos mencionados es muy probable que comparaciones de este tipo no produzcan los resultados deseados; las pequeñas diferencias introducidas por el redondeo o la no representatividad son suficientes para que la comparación de cómo resultado *false* y no se ejecute el código del cuerpo de la sentencia (por ejemplo, usando las variables *a* y *b* del ejemplo anterior constrúyase una sentencia *if(a*a==b)* y compruébese que el cuerpo de la

sentencia no se ejecuta). Si es realmente necesario usar sentencias *if* que comparen variables de coma flotante de esta forma, debe implementarse como sigue:

```
if( Math.abs(x-y) < tolerancia ) {
      Codigo a ejecutar
}
```

donde *tolerancia* es el valor prefijado del margen de error aceptable en la comparación. Nótese también que en el caso de variables enteras estos problemas no se dan y su comparación no es problemática.

3.2.2 Estabilidad numérica de un algoritmo

La ejecución de un algoritmo implica normalmente el uso de muchas variables de coma flotante y la realización de muchas operaciones con las mismas. Los efectos acumulativos de los errores de representación y de redondeo pueden tener un efecto no despreciable en el resultado del algoritmo. El término *estabilidad numérica* de un algoritmo hace referencia a la sensibilidad del mismo en frente a estos errores; los buenos algoritmos son numéricamente estables, es decir, los errores de representación y redondeo sobre los mismos tienen un efecto reducido sobre el resultado final. Por el contrario, hay algoritmos numéricamente inestables en los cuales estos errores se acumulan produciendo resultados finales donde el error es grande.

Veamos un ejemplo[7]: supongamos que queremos calcular el valor de la constante π. Para ello podemos usar el método de Arquímedes[8] basado en aproximar el área y perímetro de una circunferencia de radio 1 mediante polígonos con un número creciente de lados que la circunscriben. Cuando el número de lados del polígono tiende a infinito su área tiende a la de la circunferencia, y por lo tanto a π. Este cálculo puede plantearse de forma recursiva (el resultado para un número de lados N se expresa en función del resultado para N-1) de dos formas distintas, de estabilidad numérica diferente.

Partimos de un hexágono que circunscribe la circunferencia. Podemos descomponer este hexágono en seis triángulos cuya altura coincide con el radio de la circunferencia:

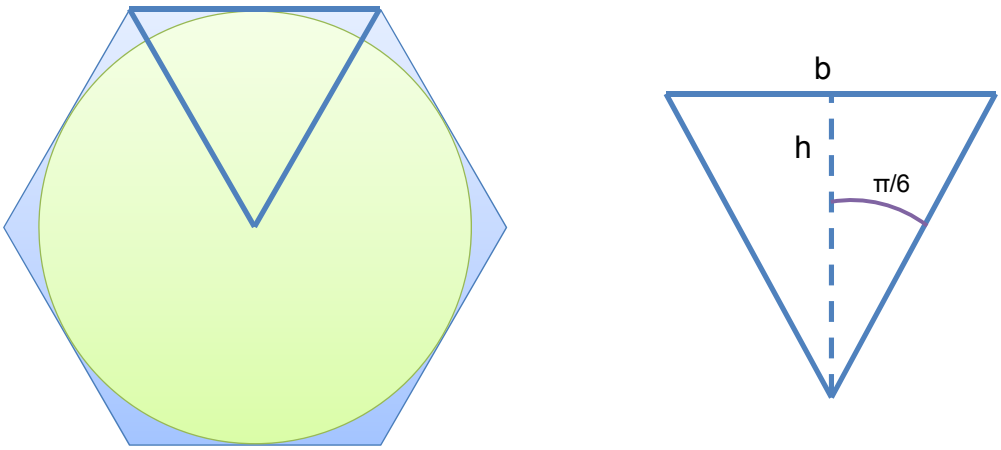


Figura 45. Método de Arquímedes para el cálculo de π

Los triángulos tienen altura h=R=1. Por lo tanto su base será $b = 2h \tan\left(\frac{\pi}{6}\right)$ y su área:

[7] http://en.wikipedia.org/wiki/Floating_point#Minimizing_the_effect_of_accuracy_problems

[8] http://www.math.psu.edu/courses/maserick/circle/circleapplet.html

$$T_0 = \frac{bh}{2} = h^2 tan\left(\frac{\pi}{6}\right) = \frac{1}{\sqrt{3}}$$

A continuación multiplicaremos el número de lados del polígono por dos sucesivamente (12, 24, 48, …), de forma que las áreas de los triángulos sucesivos serán:

$$T_1 = tan\left(\frac{\pi}{12}\right)$$
$$T_2 = tan\left(\frac{\pi}{24}\right)$$
$$\vdots$$

O lo que es lo mismo:

$$T_i = tan(\alpha_i)$$
$$\alpha_i = \frac{\pi}{6 \cdot 2^i}$$

Podemos ver que hay una relación recursiva entre los ángulos:

$$\alpha_i = \frac{\pi}{6 \cdot 2^i} \Rightarrow \alpha_{i+1} = 2\,\alpha_i$$

Si tenemos en cuenta la relación del ángulo doble para la tangente y la cotangente[9]:

$$tan(2\alpha) = \frac{2tan(\alpha)}{1 - \tan^2(\alpha)} \Rightarrow tan(2\alpha)\tan^2(\alpha) + 2\,tan(\alpha) - tan(2\alpha) = 0$$

$$cot(2\alpha) = \frac{1 - cot^2(\alpha)}{2\cot(\alpha)} \Rightarrow \cot^2(\alpha) - 2cot(2\alpha)cot(\alpha) - 1 = 0$$

podemos relacionar las tangentes de esto dos ángulos de dos formas distintas, respectivamente:

$$tan(x) = \frac{\sqrt{1 + tan^2(2x)} - 1}{tan(2x)} \Rightarrow T_{i+1} = \frac{\sqrt{1 + T_i^2} - 1}{T_i}$$
$$tan(x) = \frac{tan(2x)}{1 + \sqrt{1 + tan^2(2x)}} \Rightarrow T_{i+1} = \frac{T_i}{\sqrt{1 + T_i^2} + 1}$$

Usando cualquiera de las dos relaciones podemos calcular partiendo de T_0 una secuencia de áreas de triángulos y a partir de ellas las áreas de los polígonos correspondientes, que convergen hacia π cuando $i \to \infty$:

$$A_i = 6 \cdot 2^i T_i \Rightarrow \lim_{i \to \infty}(A_i) = \pi$$

Matemáticamente las dos relaciones son estrictamente equivalentes; a priori cualquiera de ellas puede ser usada para calcular una secuencia A_i que tienda a π. Sin embargo, la estabilidad numérica de las dos expresiones no es la misma, como puede verse a partir de los resultados en la tabla siguiente obtenidos con un programa en Java que implementa el cálculo recursivo descrito:

[9] http://www.mathpages.com/HOME/kmath425/kmath425.htm

	$T_{i+1} = \frac{\sqrt{1+T_i^2}-1}{T_i}$	$T_{i+1} = \frac{T_i}{\sqrt{1+T_i^2}+1}$
A_0	3.4641016151377553	3.4641016151377553
A_1	3.2153903091734750	3.2153903091734730
A_2	3.1596599420975098	3.1596599420975013
A_3	3.1460862151314670	3.1460862151314357
A_4	3.1427145996455734	3.1427145996453696
A_5	3.1418730499798660	3.1418730499798250
A_6	3.1416627470550680	3.1416627470568503
A_7	3.1416101765995217	3.1416101766046913
A_8	3.1415970343233370	3.1415970343215283
A_9	3.1415937488168560	3.1415937487713546
A_{10}	3.1415929278736330	3.1415929273850987
A_{11}	3.1415927256225915	3.1415927220386157
A_{12}	3.1415926717415450	3.1415926707020000
A_{13}	3.1415926189008863	3.1415926578678460
A_{14}	3.1415926717415450	3.1415926546593083
A_{15}	3.1415919358819730	3.1415926538571740
A_{16}	3.1415926717415450	3.1415926536566400
A_{17}	3.1415810075793640	3.1415926536065070
A_{18}	3.1415926717415450	3.1415926535939738
A_{19}	3.1414061547376217	3.1415926535908403
A_{20}	3.1405434924010995	3.1415926535900570
A_{21}	3.1400068646909682	3.1415926535898615
A_{22}	3.1349453756588517	3.1415926535898127
A_{23}	3.1400068646909682	3.1415926535898000
A_{24}	3.2245152435348190	3.1415926535897976
A_{25}	2.7911172130586380	3.1415926535897967
A_{26}	0.0	3.1415926535897967
A_{27}	NaN	3.1415926535897967
A_{28}	NaN	3.1415926535897967
A_{29}	NaN	3.1415926535897967

El algoritmo basado en la primera expresión es numéricamente inestable. Los errores numéricos hacen que la secuencia A_i que debería converger al valor de π, no lo haga e incluso llegue a producir NaNs. Por el contrario, el algoritmo basado en la segunda expresión es numéricamente estable y converge hacia un valor cercano a π (aunque no exactamente al valor correcto, puesto que π = 3.14159265358979323846264338327...).

Ejercicio 34: comprobar los cálculos anteriores escribiendo un programa que realice los cálculos iterativos descritos usando las dos expresiones. Razonar cual puede ser la causa de la diferencia de comportamiento entre los dos casos.

3.2.3 Recursos necesarios para la ejecución de un algoritmo

Otra característica importante de un algoritmo es su uso de los recursos del ordenador: CPU, memoria RAM y espacio en disco. Un algoritmo puede ser matemáticamente óptimo y numéricamente estable, pero al mismo tiempo requerir tantos recursos del ordenador que sea inviable su ejecución. Por ejemplo, si un algoritmo requiere en algún momento de su ejecución una cantidad de memoria RAM que sobrepasa la disponibilidad de nuestro ordenador, no podrá ejecutarse.

Por todo ello, cuando se vaya a seleccionar un algoritmo para su implementación en la resolución de un problema es importante plantearse los recursos que requerirá su ejecución. Si requiere demasiados recursos tal vez deba seleccionarse (si ello es posible) un algoritmo con menos prestaciones pero ejecutable con el ordenador disponible. En general, la selección suele ser un compromiso entre estabilidad numérica, velocidad de cálculo y necesidad de memoria y disco.

3.2.4 Escalabilidad de un algoritmo

Ocurre frecuentemente que los problemas numéricos que se plantean en ciencia e ingeniería corresponden a casos en que existe un cierto parámetro que determina la "talla" del problema y por tanto la complejidad de los cálculos asociados. Por ejemplo, la complejidad del proceso de inversión de una matriz está relacionado con la talla de las matrices implicadas; la inversión de una matriz 1000x1000 requiere muchos más cálculos que la inversión de una matriz 10x10.

En estos casos es muy importante analizar como aumenta la complejidad de la ejecución de un algoritmo dado con la talla del problema, y por lo tanto como aumenta el tiempo de ejecución requerido para completar el cálculo con la talla. El comportamiento del algoritmo con la talla del problema se denomina *escalabilidad del algoritmo* y dependiendo de cómo sea este comportamiento los algoritmos pueden ser muy escalables o poco escalables.

Supongamos que tenemos un problema cuya talla está caracterizada por un parámetro N (en el ejemplo anterior de inversión de una matriz el parámetro seria el número de filas o columnas de la matriz). Para estudiar la escalabilidad de un algoritmo aplicado al problema debemos analizar como aumenta el tiempo de ejecución cuando aumenta N:

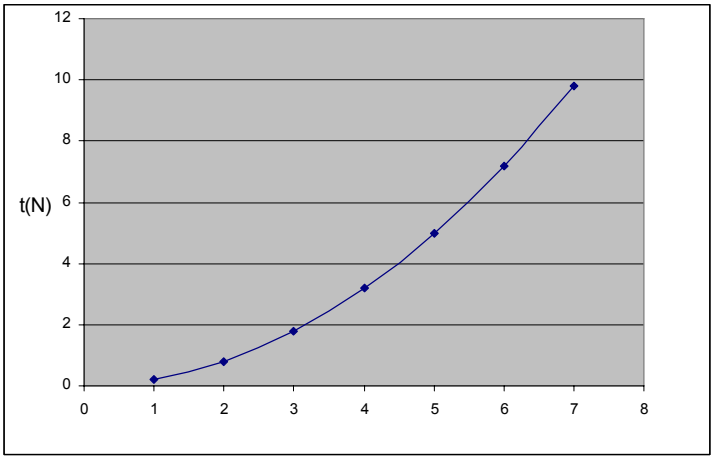


Figura 46. Escalabilidad de un algoritmo y tiempo de ejecución

Este resultado nos permitirá estimar cómo se comporta el algoritmo para valores de N grandes y así evaluar si es adecuado para nuestras necesidades. En un caso ideal de escalabilidad óptima t(N) crece linealmente con N. En otros casos de escalabilidad peor t(N) puede crecer por ejemplo como una potencia de N. La escalabilidad suele denotarse haciendo referencia al "orden de magnitud" del tiempo de ejecución del algoritmo en función de N. Un algoritmo donde t(N) es linear se denota como O(N); un algoritmo donde t(N) crece como N^2 se denota como $O(N^2)$.

Puede verse claramente la importancia de la escalabilidad tomando como ejemplo el problema de la solución de sistemas de ecuaciones lineales. En este caso N corresponde al tamaño del sistema (número de ecuaciones).

Supongamos que queremos resolver sistemas de 100 ecuaciones y tenemos dos implementaciones, una usando el método de Cramer y la otra usando el método de Gauss (ver sección). Supongamos también para sistemas de 10 ecuaciones el tiempo de ejecución es aproximadamente el mismo en los dos casos. ¿Cómo podemos esperar que se comporten las dos implementaciones para nuestros sistemas de 100 ecuaciones? El método de Cramer es O(N!) mientras que el método de Gauss es $O(2\ N^3\ /3)$ lo cual que implica que en el primer caso el tiempo de ejecución se incrementará en un factor $100!/10! \cong 2.6x10^{151}$ mientras que en el segundo caso el incremento será de $100^3/10^3 = 1000$. Vemos como la aplicación del método de Cramer es completamente inviable para sistemas de 100 ecuaciones mientras que el método de Gauss es aplicable.

3.2.5 Problemas mal condicionados

En algunos casos la resolución de un problema numérico puede ser difícil, o hasta incluso imposible, debido a factores intrínsecos al planteamiento matemático del problema. Esto se da en general cuando pequeñas variaciones en los parámetros del problema inducen grandes variaciones en la solución del mismo. En casos como éste el efecto de los errores numéricos es amplificado, dificultando en gran manera la aplicación de métodos numéricos y debe analizarse el problema de forma individualizada, siendo muy cuidadoso con la programación para minimizar los errores numéricos.

3.3 Librerías numéricas

Como ya hemos mencionado, desde el advenimiento de los ordenadores el cálculo numérico ha estado estrechamente ligado a la informática. A lo largo de los años se han desarrollado e implementado en software muchos métodos numéricos usando los diversos lenguajes disponibles en cada momento (FORTRAN, C, C++, Java, etc). En algunos casos las implementaciones desarrolladas se han organizado en paquetes de software que agrupan de forma organizada un amplio abanico de herramientas de cálculo numérico y están disponibles (gratuitamente o previo pago) para su utilización. Estos paquetes de software se denominan usualmente *librerías numéricas*. Puede encontrarse una lista bastante completa de librerías numéricas en la siguiente entrada de la wikipedia:

http://en.wikipedia.org/wiki/List_of_numerical_libraries

Estas librerías facilitan en gran medida el uso de métodos numéricos puesto que proporcionan buenas (en general) implementaciones de los algoritmos numéricos más habituales que pueden usarse fácilmente desde nuestro propio código. Destacamos algunas recomendaciones básicas para un uso correcto de estas librerías:

- Antes de usarlas, es muy recomendable leer la documentación que las acompaña para aprender a hacer un uso correcto de las mismas
- Cada librería suele definir (en forma de interfaces, clases propias o simplemente técnicas de programación recomendadas) la forma de trabajar con ellas. La familiarización con la filosofía de trabajo de una librería facilita su uso y permite una programación más eficiente.
- La tentación de usar las implementaciones de los algoritmos proporcionadas por las librerías como “cajas negras”, sin entender en detalle cómo funciona el algoritmo que implementan, es grande. Este tipo de uso es justificable si estamos realizando tareas sencillas que los algoritmos pueden resolver fácilmente o si estamos en una fase preliminar de desarrollo, probando el funcionamiento de los mismos. Sin embargo, antes de usar un algoritmo para una tarea importante y/o compleja debería tenerse una comprensión suficiente de sus fundamentos matemáticos como para hacer un uso correcto del mismo, teniendo en cuenta sus limitaciones y características.

En esta asignatura haremos uso de la librería numérica en *Apache Commons Math*, versión 2.0. Esta es una librería desarrollada en Java y libremente distribuible, con la ventaja añadida de que el código fuente está disponible lo que permite analizar cómo se ha realizado la implementación de un algoritmo dado (e incluso modificarla si es necesario). Presentamos a continuación diversos enlaces relevantes de *Apache Commons Math*:

Página principal	http://commons.apache.org/math/
Guía de usuario	http://commons.apache.org/math/userguide/index.html
JavaDoc de la version 2.0	http://commons.apache.org/math/api-2.0/index.html
Acceso al código fuente	http://svn.apache.org/viewvc/commons/proper/math/trunk/

Recomendamos al alumno visitar estos enlaces y familiarizarse mínimamente con sus contenidos antes de abordar las secciones siguientes.

3.3.1 Integración de librerías en Netbeans

Para poder utilizar una librería es necesario darla de alta en Netbeans. En primer lugar bajaremos los ficheros binarios, fuente y javadoc. Una vez hecho, crearemos la librería en el gestor de librerías (Tools->Libraries):

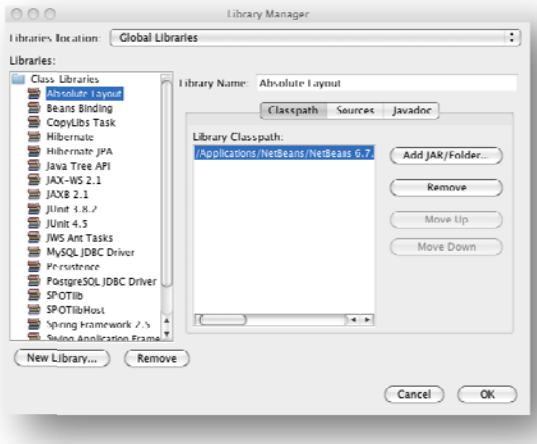


Figura 47. Gestor de librerías

En ésta ventana se puede crear una nueva librería, a la que se le pueden asignar las clases compiladas (.jar), el código fuente y la documentación. Para ello en primer lugar creamos la librería (New Library…):

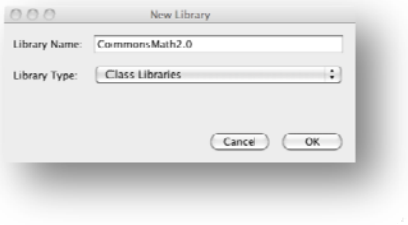


Figura 48. Ventana de Librería Nueva

A partir de este momento, la librería ya esta creada, ahora sólo es necesario indicar que ficheros forma la librería. Empezamos con los binarios (classpath). Añadimos el fichero de binarios con (Add jar/folder).

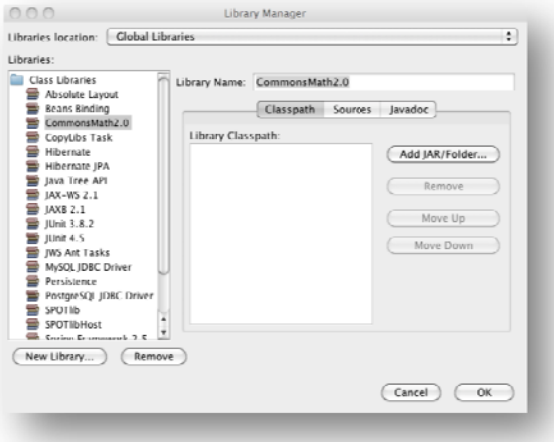


Figura 49. Gestión de la librería CommonsMath2.0

Escogemos el fichero jar que corresponde a los binarios, con lo que aparece en el listado de Library Classpath.

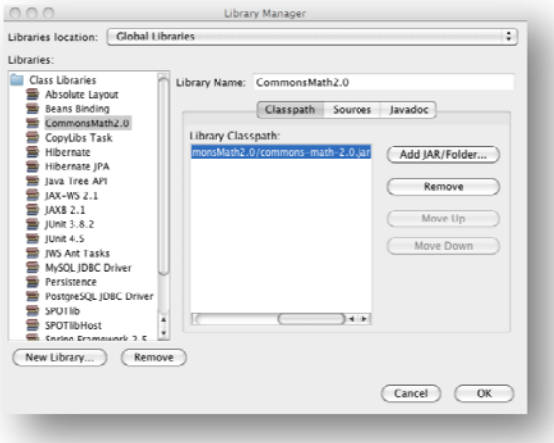


Figura 50. Fichero binario de librería escogido

De la misma forma podemos dar de alta los ficheros con las fuentes y la documentación.

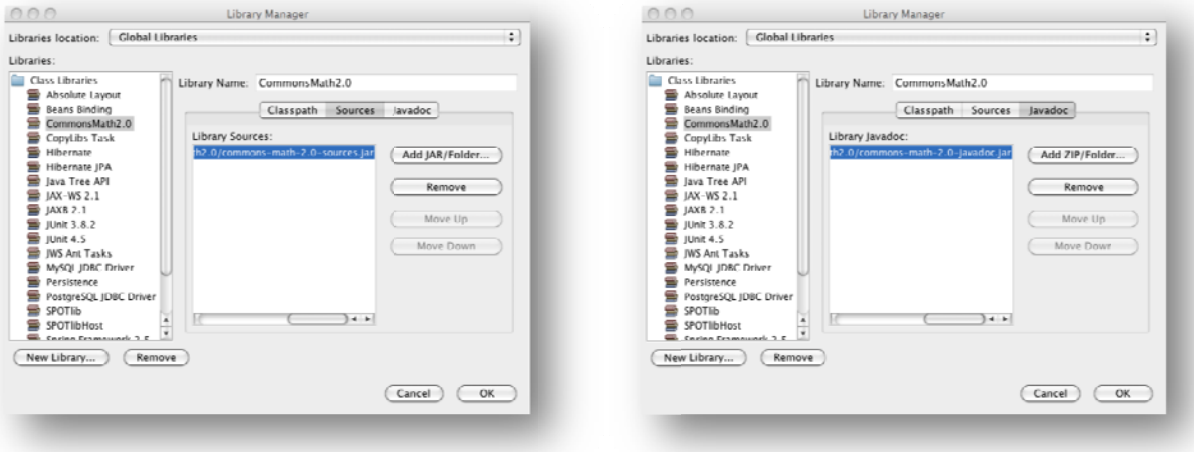


Figura 51. Fichero de fuentes y documentación escogidos

Una vez hecho esto, ya disponemos de la librería en Netbeans. Ahora es necesario incluirla en el proyecto para poderla utilizar. Esto lo hacemos poniéndonos dentro de la ventana de proyectos y encima del nodo librería de nuestro proyecto. En el menú contextual, escogemos “Add Library”.

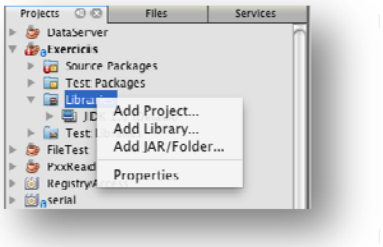


Figura 52. Añadir una librería a un proyecto

En la ventana que se muestra escogemos CommonsMath2.0, y a partir de ese momento está disponible en el proyecto.

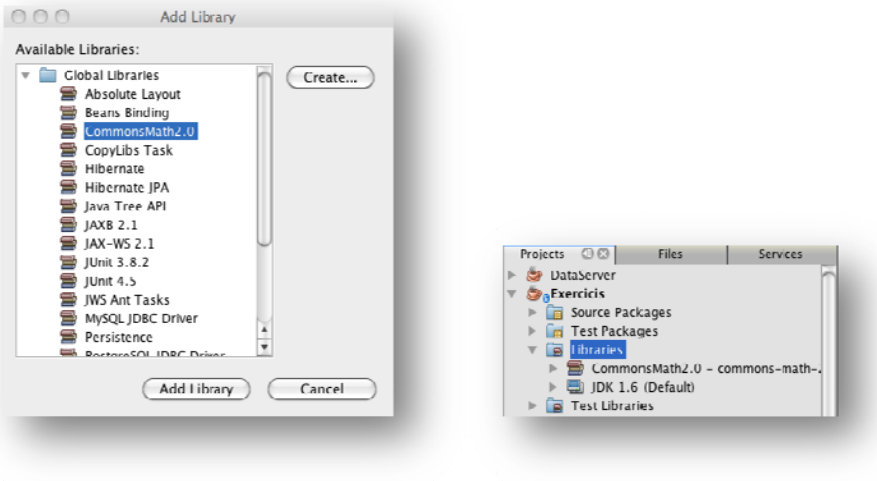


Figura 53. Proyecto con la librería dada de alta

De esta forma ya podemos trabajar con las diferentes clases de la librería Commons Math.

3.4 Sistemas de ecuaciones lineales

3.4.1 Planteamiento del problema

Un sistema de *N* ecuaciones lineales con *N* incógnitas es un conjunto de ecuaciones con la siguiente estructura:

$$\begin{array}{ccccccccc} a_{0,0}x_0 & + & a_{0,1}x_1 & + & \cdots & + & a_{0,N-1}x_{N-1} & = & b_0 \\ a_{1,0}x_0 & + & a_{1,1}x_1 & + & \cdots & + & a_{1,N-1}x_{N-1} & = & b_1 \\ \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{N-1,0}x_0 & + & a_{N-1,1}x_1 & + & \cdots & + & a_{N-1,N-1}x_{N-1} & = & b_{N-1} \end{array}$$

Aunque es posible trabajar con sistemas de *M* ecuaciones con *N* incógnitas donde *M≠N* (como se discute en la asignatura de álgebra), nos limitaremos aquí a los sistemas con *M=N*; sólo en este caso es posible la existencia de una solución única, cuya determinación es el objetivo de los métodos numéricos presentados.

Nótese que:

- Los coeficientes de las ecuaciones $a_{i,j}$ y los términos independientes b_i son números reales o complejos (aunque en esta asignatura sólo trataremos con sistemas de ecuaciones con valores reales) prefijados, que definen el sistema de ecuaciones.
- Las incógnitas x_j son números reales o complejos (aunque en esta asignatura sólo trataremos con sistemas de ecuaciones con valores reales) cuyos valores queremos determinar de forma que las ecuaciones anteriores se cumplan. Nótese que el sistema puede tener solución única, múltiples soluciones o no tener ninguna solución. Nos remitimos a la asignatura de álgebra para una discusión de la casuística de los sistemas de ecuaciones lineales y la existencia de soluciones.
- Hemos usado para la numeración de los índices de los coeficientes y términos independientes la convención de tomar el primer índice como cero, de forma que el último índice es *N-1*. De esta forma somos consistentes con el uso de los índices de las variables de tipo matriz en Java, a[0][0], a[0][1], etc. Es posible que al tratar con matrices en otras asignaturas los índices de sus coeficientes se numeren como 1, …, N por lo que hay que ir con cuidado para evitar confusiones en los índices al escribir código.

Usando notación matricial el sistema puede escribirse de forma más compacta como:

$$Ax = b$$

donde

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{0,N-1} \\ \vdots & \ddots & \vdots \\ a_{N-1,0} & \cdots & a_{N-1,N-1} \end{pmatrix} \quad x = \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} \quad b = \begin{pmatrix} b_0 \\ \vdots \\ b_{N-1} \end{pmatrix}$$

3.4.2 Interfaces

En primera instancia el trabajo con vectores y matrices en Java puede realizarse usando *arrays* de una y dos dimensiones respectivamente. Por ejemplo las declaraciones:

```
double[][] A = { {1.,2.,3.}, {4.,5.,6.},{7.,8.,9.} };
double[]   x = {1.,2.,3.};
```

definen dos *arrays* para representar una matriz *A* y un vector *x*. Podemos acceder a sus elementos usando los índices correspondientes, por ejemplo:

```
a[0][0]= 1.0;
a[0][1]= 2.0;
a[1][0]= 4.0;
a[2][1]= 8.0;
x[0]=    1.0;
```

Usando estos elementos podemos programar cualquier tipo de operación con matrices, vectores o entre vectores y matrices.

Ejercicio 35: usando las definiciones anteriores realizar un programa que calcule el producto *Ax*, lo guarde en un vector *b* y escriba este vector en pantalla.

Sin embargo, trabajar directamente con *arrays* se vuelve rápidamente tedioso si el problema a tratar es mínimamente complejo y por ello es conveniente definir interfaces que simplifiquen el trabajo con vectores y matrices. En lugar de definir nuestros propias interfaces recomendamos usar las definidas en el paquete de álgebra linear de *Apache Commons Math*[10]: *RealMatrix* y *RealVector*. Estos interfaces incluyen diversas operaciones básicas que simplificarán nuestro trabajo:

- Suma, resta y producto de matrices
- Producto y suma de un escalar
- Transposición
- Cálculo de la norma y la traza de una matriz
- Operaciones de un vector sobre una matriz

En este caso la matriz A y el vector x definidos anteriormente se instanciarán de la forma siguiente usando las clases *Array2DRowRealMatrix* y *ArrayRealVector* que implementan las interfaces:

```
double[][] auxA = { {1.,2.,3.}, {4.,5.,6.},{7.,8.,9.} };
Array2DRowRealMatrix A = new Array2DRowRealMatrix(auxA);

double[]   auxx = {1.,2.,3.};
ArrayRealRealVector x = new ArrayRealVector(auxx);
```

Usando los métodos provistos por estas clases podemos realizar fácilmente diversas operaciones.

Ejercicio 36: usando las definiciones anteriores realizar un programa que calcule el producto *Ax*, la traza de A y A·A usando los métodos de la clase *Array2DRowRealMatrix*, escribiendo los resultados en pantalla.

[10] http://commons.apache.org/math/userguide/linear.html

3.4.3 Métodos numéricos de resolución de sistemas de ecuaciones lineales

3.4.3.1 *Inversa y determinante de una matriz*

La solución de sistemas de ecuaciones lineales y el cálculo de la inversa de una matriz y su determinante son problemas estrechamente ligados. Obviamente, si se dispone de un método de inversión de matrices la resolución de un sistema de ecuaciones lineales es trivial:

$$Ax = b \Rightarrow x = A^{-1}b$$

Por su parte, la inversión de matrices requiere del cálculo de determinantes.

Ejercicio 37: la clase *LUDecompositionImpl* de Apache Common Math incluye un método para la inversión de matrices: *LUDecompositionImpl(RealMatrix A).getSolver().getInverse()*. Dados

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 4 & 4 \\ 5 & 6 & 6 \end{pmatrix} \; b = \begin{pmatrix} 14 \\ 24 \\ 35 \end{pmatrix}$$

realizar un programa que invierta A y calcule la solución del sistema *Ax=b* como $x = A^{-1}b$. Comprobad que la solución es x = {1; 2; 3}; comprobad también que la matriz inversa es

$$A^{-1} = \begin{pmatrix} 0 & 1{,}5 & -1 \\ -1 & -2{,}25 & 2 \\ 1 & 1 & -1 \end{pmatrix}$$

pero que se obtienen pequeñas diferencias respecto a este resultado debido a los errores numéricos.

No vamos, sin embargo, a discutir específicamente métodos de inversión de matrices o cálculo de determinantes, puesto que resulta que la forma más eficiente de realizar estos cálculos forma parte de los métodos de resolución de sistemas de ecuaciones lineales. Veamos por ejemplo el problema del cálculo de un determinante. El determinante de una matriz A se define como (véase la asignatura de álgebra):

$$det(A) = \sum_{\sigma} \varepsilon(\sigma) \prod_{i=1}^{N} a_{i,\sigma(i)}$$

Este cálculo está basado en las permutaciones de N elementos y por lo tanto es de O(N!). Abordarlo directamente a partir de esta definición es ineficiente y es mucho más eficiente usar las características de algunos de los métodos de solución de ecuaciones lineales que discutiremos en esta sección. Por ejemplo, el método de Gauss incluye la reducción de la matriz A del sistema a una matriz triangular A' equivalente (es decir, correspondiente a un sistema de ecuaciones con la misma solución). Resulta que en este caso

$$\det(A) = \sigma \det(A')$$

Donde σ = ±1 y el cálculo del determinante de A' es muy sencillo puesto que se trata de una matriz triangular:

$$det(A') = \prod_{i=1}^{N} a'_{i,i}$$

Nótese que este cálculo es O(N) y dado que la triangularización es $O(N^2)$ puede verse que globalmente este procedimiento es más eficiente para matrices grandes que el cálculo directo.

3.4.3.2 *Método de Cramer*

Dado un sistema de ecuaciones

$$Ax = b$$

la regla de Cramer o método de Cramer[11] es un teorema que proporciona una expresión formalmente muy simple para calcular la solución del sistema usando determinantes:

$$x_i = \frac{D_i}{\det(A)} \; i = 0, \dots, N-1 \quad D_i = \begin{vmatrix} a_{0,0} & \cdots & a_{0,i-1} & b_0 & a_{0,i+1} & \cdots & a_{0,N-1} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{N-1,0} & \cdots & a_{N-1,i-1} & b_{N-1} & a_{N-1,i+1} & \cdots & a_{N-1,N-1} \end{vmatrix}$$

Vemos como las componentes de la solución del sistema pueden calcularse directamente a partir del determinante de su matriz *det(A)* y del determinante D_i de la matriz que resulta de reemplazar la i-ésima columna de *A* por el vector *b*. Aunque formalmente muy simple, su aplicación práctica es limitada puesto que requiere el cálculo de determinantes; dado que, como ya hemos comentado, la forma más eficiente de calcular determinantes se basa en métodos de resolución de ecuaciones lineales es entonces preferible usar estos últimos directamente. Sin embargo, puede resultar útil para matrices muy pequeñas (N=2, 3, 4) en cuyo caso la implementación puede hacerse muy sencilla y eficiente.

Ejercicio 38: implementar el método de Cramer para la resolución de sistemas de ecuaciones con N=3 y aplicarlo al ejemplo del ejercicio anterior:

$$x_0 = \frac{\begin{vmatrix} b_0 & a_{0,1} & a_{0,2} \\ b_1 & a_{1,1} & a_{1,2} \\ b_2 & a_{2,1} & a_{2,2} \end{vmatrix}}{\det(A)} \qquad x_1 = \frac{\begin{vmatrix} a_{0,0} & b_0 & a_{0,2} \\ a_{1,0} & b_1 & a_{1,2} \\ a_{2,0} & b_2 & a_{2,2} \end{vmatrix}}{\det(A)} \qquad x_2 = \frac{\begin{vmatrix} a_{0,0} & a_{0,1} & b_0 \\ a_{1,0} & a_{1,1} & b_1 \\ a_{2,0} & a_{2,1} & b_2 \end{vmatrix}}{\det(A)}$$

Tener en cuenta que *det(A)* puede ser cero en cuyo caso el sistema es incompatible o no tiene solución única, y por lo tanto no puede calcularse una solución.

3.4.3.3 Método de Gauss

La idea básica del método de Gauss es la siguiente: nuestro sistema de ecuaciones está representado por una matriz *A* y un vector *b*

$$Ax = b$$

nos proponemos buscar una representación equivalente (en el sentido de que la solución *x* sea la misma) de nuestro sistema basado en otra matriz A' y otro vector b'

$$A'x = b'$$

con la particularidad de que A' sea una matriz triangular inferior:

$$A' = \begin{pmatrix} a'_{0,0} & a'_{0,1} & a'_{0,2} & \cdots & a'_{0,N-2} & a'_{0,N-1} \\ 0 & a'_{1,1} & a'_{1,2} & \cdots & a'_{1,N-2} & a'_{1,N-1} \\ 0 & 0 & a'_{2,2} & \cdots & a'_{2,N-2} & a'_{2,N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a'_{N-2,N-2} & a'_{N-2,N-1} \\ 0 & 0 & 0 & \cdots & 0 & a'_{N-1,N-1} \end{pmatrix}$$

Una vez conseguido esto, la solución puede obtenerse de forma trivial resolviendo el sistema alternativo resultante:

[11] Denominado así en honor del matemático Gabriel Cramer

$$\begin{array}{ccccccccccccc} a'_{0,0}x_0 & + & a'_{0,1}x_1 & + & a'_{0,2}x_2 & + & \cdots & + & a'_{0,N-2}x_{N-2} & + & a'_{0,N-1}x_{N-1} & = & b'_0 \\ & & a'_{1,1}x_1 & + & a'_{1,2}x_2 & + & \cdots & + & a'_{1,N-2}x_{N-2} & + & a'_{1,N-1}x_{n-1} & = & b'_1 \\ & & & & a'_{2,2}x_2 & + & \cdots & + & a'_{2,N-2}x_{N-2} & + & a'_{2,N-1}x_{N-1} & = & b'_2 \\ & & & & & & \ddots & & \vdots & & \vdots & & \vdots \\ & & & & & & & & a'_{N-2,N-2}x_{N-2} & + & a'_{N-2,N-1}x_{N-1} & = & b'_{N-2} \\ & & & & & & & & & & a'_{N-1,N-1}x_{N-1} & = & b'_{N-1} \end{array}$$

Empezando por la última fila obtenemos

$$x_{N-1} = \frac{b'_{N-1}}{a'_{N-1,N-1}}$$

y a partir de este primer valor el resto como:

$$x_i = b'_j - \frac{\sum_{j=i+1}^{N-1} a'_{i,j}x_j}{a'_{i,i}} \; i = N-2, \ldots, 0$$

La cuestión es, por lo tanto, como determinar este sistema de ecuaciones equivalente a partir del original *Ax=b*. La respuesta se encuentra en la teoría de aplicaciones lineales y su aplicación a la resolución de sistemas de ecuaciones lineales que se desarrolla en la asignatura de álgebra. Dejamos para esa asignatura la justificación teórica de los procedimientos aquí usados, limitándonos a exponerlos y usarlos.

En primer lugar, para facilitar la notación en nuestra discusión escribiremos nuestro sistema de una forma más compacta uniendo la matriz A y el vector b en una única matriz:

$$A^{(0)} = \left(\begin{array}{ccc|c} a_{0,0} & \cdots & a_{0,N-1} & b_0 \\ \vdots & \ddots & \vdots & \vdots \\ a_{N-1,0} & \cdots & a_{N-1,N-1} & b_{N-1} \end{array}\right)$$

3.4.3.3.1 Gauss sin pivotaje

La teoría de aplicaciones lineales nos dice que si sumamos a una fila de esta matriz una combinación lineal del resto de filas, la solución del sistema no cambia[12]. Suponiendo que $a_{0,0}$ no sea nulo podemos a realizar entonces las siguientes operaciones:

$$\begin{array}{ccc} fila(1) & = & fila(1) - \frac{a_{1,0}}{a_{0,0}} fila(0) \\ fila(2) & = & fila(2) - \frac{a_{2,0}}{a_{0,0}} fila(0) \\ \vdots & & \vdots \\ fila(N-1) & = & fila(N-1) - \frac{a_{N-1,0}}{a_{0,0}} fila(0) \end{array}$$

Con ello obtenemos una nueva matriz donde toda la primera columna salvo el primer coeficiente es cero:

[12] En términos algebraicos, dada una aplicación lineal $f : E \to F$ la matriz *A* es la representación de f fijada una base para el espacio de partida ***E*** y otra base para el espacio de llegada ***F***, y el vector *b* corresponde a las coordenadas de $f(x)$ en la base de F. El sumar a una fila de $A^{(0)}$ una combinación lineal del resto de filas es equivalente a realizar un cambio de base en el espacio de llegada F sin realizarlo en el espacio de partida E; con este cambio la matriz asociada a f cambia, así como las coordenadas de $f(x)$ y por lo tanto el vector *b*, pero al no haber cambiado la base en el espacio de partida E no cambian las coordenadas del vector *x*, y por lo tanto no cambia la solución.

$$A^{(1)} = \left(\begin{array}{ccccc|c} a_{0,0}^{(1)} & a_{0,1}^{(1)} & \cdots & a_{0,N-2}^{(1)} & a_{0,N-1}^{(1)} & b_0^{(1)} \\ 0 & a_{1,1}^{(1)} & \cdots & a_{1,N-2}^{(1)} & a_{1,N-1}^{(1)} & b_1^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & a_{N-2,1}^{(1)} & \cdots & a_{N-2,N-2}^{(1)} & a_{N-2,N-1}^{(1)} & b_{N-2}^{(1)} \\ 0 & a_{N-1,1}^{(1)} & \cdots & a_{N-1,N-2}^{(1)} & a_{N-1,N-1}^{(1)} & b_{N-1}^{(1)} \end{array}\right) \quad a_{0,i}^{(1)} = a_{0,i}\,, b_0^{(1)} = b_0$$

Suponiendo que $a_{1,1}^{(1)}$ no sea nulo, si repetimos la operación usando en este caso la fila 1 para combinarla con el resto de filas obtendremos una nueva matriz $A^{(2)}$ donde la segunda columna contendrá ceros salvo los dos primeros coeficientes. Repitiendo la operación sucesivamente llegaremos a la matriz $A^{(N-1)}$ que será triangular superior:

$$A^{(N-1)} = \left(\begin{array}{ccccc|c} a_{0,0}^{(N-1)} & a_{0,1}^{(N-1)} & \cdots & a_{0,N-2}^{(N-1)} & a_{0,N-1}^{(N-1)} & b_0^{(N-1)} \\ 0 & a_{1,1}^{(N-1)} & \cdots & a_{1,N-2}^{(N-1)} & a_{1,N-1}^{(N-1)} & b_1^{(N-1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{N-2,N-2}^{(N-1)} & a_{N-2,N-1}^{(N-1)} & b_{N-2}^{(N-1)} \\ 0 & 0 & \cdots & 0 & a_{N-1,N-1}^{(N-1)} & b_{N-1}^{(N-1)} \end{array}\right)$$

Dado que en todos los casos hemos obtenido las matrices sumando a una fila una combinación lineal del resto de filas, todas ellas (y en particular $A^{(N-1)}$) son equivalentes, los sistemas de ecuaciones correspondientes tienen la misma solución que $Ax=b$. Por lo tanto $A^{(N-1)}$ es la matriz triangular que nos permite resolver el sistema de ecuaciones. Además se verifica que $\det(A)=\det(A^{(1)})=\ldots=\det(A^{(N-1)})$ con lo que podemos calcular el determinante de A fácilmente a partir de la matriz triangular $A^{(N-1)}$.

3.4.3.3.2 Gauss con pivotaje parcial

El método de Gauss sin pivotaje que hemos descrito tiene un inconveniente. En cada etapa, para obtener la matriz $A^{(k+1)}$ hemos tenido que suponer que el coeficiente $a_{k,k}^{(k)}$ de la matriz anterior no era cero. Estos elementos $a_{k,k}^{(k)}$ se denominan pivotes, y puede ocurrir que este durante el proceso de triangularización nos encontremos con que alguno de ellos sea cero, con lo que el procedimiento anterior no será utilizable.

Para solucionar este problema podemos aplicar otro resultado de la teoría de aplicaciones lineales: si cambiamos el orden de las filas de una matriz $A^{(k)}$ la solución del sistema no cambia[13]. Podemos ver que esto soluciona el problema, puesto que si en la etapa k-ésima de la triangularización nos aparece un pivote con valor cero, es suficiente con intercambiar dos filas de la matriz $A^{(k)}$de forma que obtengamos una matriz $A'^{(k)}$ donde el pivote $a'^{(k)}_{k,k}$ sea distinto de cero (este intercambio se llama pivotaje). Por ejemplo:

$$A^{(1)} = \left(\begin{array}{ccccc|c} a_{0,0}^{(1)} & a_{0,1}^{(1)} & \cdots & a_{0,N-2}^{(1)} & a_{0,N-1}^{(1)} & b_0^{(1)} \\ 0 & 0 & \cdots & a_{1,N-2}^{(1)} & a_{1,N-1}^{(1)} & b_1^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & a_{N-2,1}^{(1)} & \cdots & a_{N-2,N-2}^{(1)} & a_{N-2,N-1}^{(1)} & b_{N-2}^{(1)} \\ 0 & a_{N-1,1}^{(1)} & \cdots & a_{N-1,N-2}^{(1)} & a_{N-1,N-1}^{(1)} & b_{N-1}^{(1)} \end{array}\right) \rightarrow A'^{(1)} = \left(\begin{array}{ccccc|c} a_{0,0}^{(1)} & a_{0,1}^{(1)} & \cdots & a_{0,N-2}^{(1)} & a_{0,N-1}^{(1)} & b_0^{(1)} \\ 0 & a_{N-2,1}^{(1)} & \cdots & a_{N-2,N-2}^{(1)} & a_{N-2,N-1}^{(1)} & b_{N-2}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{1,N-2}^{(1)} & a_{1,N-1}^{(1)} & b_1^{(1)} \\ 0 & a_{N-1,1}^{(1)} & \cdots & a_{N-1,N-2}^{(1)} & a_{N-1,N-1}^{(1)} & b_{N-1}^{(1)} \end{array}\right)$$

En este caso, cuando se intercambian dos filas el determinante cambia de signo por lo que al final del proceso $\det(A)= \pm\det(A^{(N-1)})$, donde el signo será positivo o negativo dependiendo de si han realizado un número par o impar de pivotajes.

[13] De hecho, este es un caso particular de la propiedad que hemos usado anteriormente. Un cambio de orden de las filas puede interpretarse como un cambio de base en el espacio de partida ***E***.

Cuando encontramos un pivote de valor cero debemos decidir con que otra fila se realiza el intercambio. La estrategia recomendada es seleccionar la fila que proporcione un nuevo pivote cuyo valor absoluto sea máximo, evitando así en la medida de lo posible el potencial problema de seleccionar nuevos pivotes con valores muy pequeños que puedan dar problemas numéricos. En este sentido, la estrategia de pivotaje suele expandirse para mejorar la estabilidad numérica del procedimiento: en cada etapa de triangularización se realiza sistemáticamente el pivotaje, aunque el pivote disponible no sea cero, para conseguir que el pivote finalmente usado tenga el máximo valor absoluto posible, minimizando así posibles problemas numéricos.

Ejercicio 39: en el sistema de ecuaciones

$$A = \begin{pmatrix} \delta & 1 \\ 1 & 1 \end{pmatrix} \; b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

cuando $\delta \to 0$ la solución tiende a x=(1,1). Hallar **manualmente** la solución mediante el método de Gauss con y sin pivotaje y aplicar ambos resultados para los siguientes valores de δ: 10^{-10}, 10^{-15}, 10^{-16}, $5x10^{-17}$, $3x10^{-17}$, 10^{-17}. Discutir las diferencias de los resultados con y sin pivotaje y la estabilidad numérica en cada caso.

3.4.3.3.3 Gauss con pivotaje total

En el método de Gauss con pivotaje parcial nos hemos limitado a intercambiar filas cuando es necesario cambiar un pivote. Esto es suficiente para resolver cualquier sistema y bastante eficiente en la mayoría de los casos, pero puede mejorarse su estabilidad numérica usando el denominado pivotaje total. En este caso, cuando es necesario realizar un pivotaje puede hacerse intercambiando una fila y/o una columna de la matriz, lo que amplia las posibilidades de escoger un pivote adecuado, minimizando la posibilidad de inestabilidades numérica.

El pivotaje por columnas tiene sin embargo el inconveniente de que implica un cambio en el orden de las componentes del vector *x*, por lo que se hace necesario mantener un registro de estos cambios de orden para poder reconstituir la solución al final del proceso; además, es más lento que el pivotaje parcial puesto que la búsqueda del pivote adecuado es más compleja. Como en el caso anterior $\det(A) = \pm\det(A^{(N-1)})$, donde el signo será positivo o negativo dependiendo de si han realizado un número par o impar de pivotajes (de filas o columnas).

3.4.3.3.4 Gauss-Jordan

El método de Gauss-Jordan es una variante del método de Gauss en la cual en lugar de transformar la matriz *A* en una matriz triangular superior se transforma en una matriz diagonal. La única diferencia respecto al procedimiento descrito anteriormente es que en cada paso se usa el pivote para hacer cero todos los demás coeficientes de la columna y no solo los inferiores. Por ejemplo el segundo paso del proceso sería:

$$A^{(2)} = \left(\begin{array}{ccccc|c} a_{0,0}^{(2)} & 0 & \cdots & a_{0,N-2}^{(2)} & a_{0,N-1}^{(2)} & b_0^{(2)} \\ 0 & a_{1,1}^{(2)} & \cdots & a_{1,N-2}^{(2)} & a_{1,N-1}^{(2)} & b_1^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{N-2,N-2}^{(2)} & a_{N-2,N-1}^{(2)} & b_{N-2}^{(2)} \\ 0 & 0 & \cdots & a_{N-1,N-2}^{(2)} & a_{N-1,N-1}^{(2)} & b_{N-1}^{(2)} \end{array}\right)$$

De esta forma, en la última etapa obtendremos una matriz diagonal:

$$A^{(N-1)} = \left(\begin{array}{ccccc|c} a_{0,0}^{(N-1)} & 0 & \cdots & 0 & 0 & b_0^{(N-1)} \\ 0 & a_{1,1}^{(N-1)} & \cdots & 0 & 0 & b_1^{(N-1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{N-2,N-2}^{(N-1)} & 0 & b_{N-2}^{(N-1)} \\ 0 & 0 & \cdots & 0 & a_{N-1,N-1}^{(N-1)} & b_{N-1}^{(N-1)} \end{array}\right)$$

Al proceder así la solución del sistema se obtiene de forma más sencilla que en la aplicación del método de Gauss:

$$x_i = \frac{b_i'^{(N-1)}}{a_{i,i}'^{(N-1)}}$$

Sin embargo el método de Gauss-Jordan es algo menos eficiente puesto que es $O(N^3)$ mientras que el de Gauss es $O(2N^3/3)$. La ventaja del método de Gauss-Jordan es que puede usarse para obtener la matriz inversa de *A*. Veámoslo: la inversa de *A* es una matriz *X* tal que *A·X=I*

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,N-2} & a_{0,N-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,N-2} & a_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{N-2,0} & a_{N-2,1} & \cdots & a_{N-2,N-2} & a_{N-2,N-1} \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,N-2} & a_{N-1,N-1} \end{pmatrix} \times \begin{pmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,N-2} & x_{0,N-1} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,N-2} & x_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{N-2,0} & x_{N-2,1} & \cdots & x_{N-2,N-2} & x_{N-2,N-1} \\ x_{N-1,0} & x_{N-1,1} & \cdots & x_{N-1,N-2} & x_{N-1,N-1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix}$$

Esta expresión puede interpretarse como N sistemas de ecuaciones, uno por cada columna de *X* y de *I*. Por ejemplo el primer sistema sería:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,N-2} & a_{0,N-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,N-2} & a_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{N-2,0} & a_{N-2,1} & \cdots & a_{N-2,N-2} & a_{N-2,N-1} \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,N-2} & a_{N-1,N-1} \end{pmatrix} \times \begin{pmatrix} x_{0,0} \\ x_{1,0} \\ \vdots \\ x_{N-2,0} \\ x_{N-1,0} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}$$

Si resolvemos los N sistemas de ecuaciones obtendremos los coeficientes $x_{i,j}$ y habremos calculado la inversa de A. Podemos hacerlo de forma compacta construyendo la siguiente matriz extendida

$$A^{(0)} = \left(\begin{array}{ccccc|ccccc} a_{0,0} & a_{0,1} & \cdots & a_{0,N-2} & a_{0,N-1} & 1 & 0 & \cdots & 0 & 0 \\ a_{1,0} & a_{1,1} & \cdots & a_{1,N-2} & a_{1,N-1} & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{N-2,0} & a_{N-2,1} & \cdots & a_{N-2,N-2} & a_{N-2,N-1} & 0 & 0 & \cdots & 1 & 0 \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,N-2} & a_{N-1,N-1} & 0 & 0 & \cdots & 0 & 1 \end{array}\right)$$

es decir, adjuntando la matriz identidad a la matriz A. Aplicando Gauss-Jordan a esta matriz extendida obtendremos al final del proceso:

$$A^{(N-1)} = \left(\begin{array}{ccccc|ccccc} a_{0,0}^{(N-1)} & 0 & \cdots & 0 & 0 & m_{0,0}^{(N-1)} & m_{0,1}^{(N-1)} & \cdots & m_{0,N-2}^{(N-1)} & m_{0,N-1}^{(N-1)} \\ 0 & a_{1,1}^{(N-1)} & \cdots & 0 & 0 & m_{1,0}^{(N-1)} & m_{1,1}^{(N-1)} & \cdots & m_{1,N-2}^{(N-1)} & m_{1,N-1}^{(N-1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{N-2,N-2}^{(N-1)} & 0 & m_{N-2,0}^{(N-1)} & m_{N-2,1}^{(N-1)} & \cdots & m_{N-2,N-2}^{(N-1)} & m_{N-2,N-1}^{(N-1)} \\ 0 & 0 & \cdots & 0 & a_{N-1,N-1}^{(N-1)} & m_{N-1,0}^{(N-1)} & m_{N-1,1}^{(N-1)} & \cdots & m_{N-1,N-2}^{(N-1)} & m_{N-1,N-1}^{(N-1)} \end{array}\right)$$

Lo que nos permite obtener las componentes de la matriz inversa resolviendo los N sistemas de ecuaciones (uno por cada columna de la parte adjunta) de forma diagonal:

$$x_{i,j} = \frac{m_{i,j}^{(N-1)}}{a_{i,i}'^{(N-1)}}$$

3.4.3.4 Métodos de factorización directa

Supongamos que hemos aplicado el método de Gauss (sin pivotaje) al sistema siguiente hasta llegar a una matriz triangular superior:

$$Ax = b \rightarrow Ux = \tilde{b} \; donde \; U = A^{(N-1)} \; \tilde{b} = b^{(N-1)}$$

Nótese que denotamos como *U* la matriz triangular superior que se obtiene al final del proceso (cuya notación viene de la expresión inglesa ***U**pper triangular*). Si a continuación queremos resolver un sistema similar a éste salvo en el término independiente

$$Ax = c$$

nos encontramos con que a pesar de que buena parte de los cálculos son idénticos (la mayor parte de la matriz extendida no cambia y la matriz triangular final es la misma matriz *U*) debemos repetir todo el proceso puesto que tenemos que calcular el nuevo término independiente:

$$Ax = c \rightarrow Ux = \tilde{c}\ donde\ U = A^{(N-1)}\ \tilde{c} = c^{(N-1)}$$

Esta duplicación de esfuerzo puede evitarse puesto que se demuestra que el conjunto de operaciones necesarias para obtener $\tilde{b}$ a partir de b (o $\tilde{c}$ a partir de c) pueden modelarse una cierta matriz *L* triangular inferior (cuya notación viene de la expresión inglesa ***L**ower triangular*):

$$L\tilde{b} = b\ \ L = \begin{pmatrix} 1 & 0 & \ddots & 0 & 0 \\ \ddots & \ddots & 0 & \ddots & 0 \\ l_{i,j} & \ddots & 1 & 0 & \ddots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ l_{N-1,0} & \cdots & \ddots & \ddots & 1 \end{pmatrix}$$

Es decir, esta matriz *L* está determinada por todas las combinaciones de filas del método de Gauss sin pivotaje. Vemos entonces que

$$Ux\ =\ \tilde{b}\ \rightarrow\ LUx = L\tilde{b} = b$$

Vemos, por tanto, que el sistema $LUx = b$ es equivalente al sistema $Ux = \tilde{b}$ y por lo tanto al sistema $Ax = b$. Se demuestra además que $A = LU$, lo que constituye la base de uno de los métodos de factorización directa, el **método de descomposición LU**: se descompone la matriz *A* en la forma *LU* y, una vez conseguido, se resuelve el sistema en dos etapas

$$Ax = LUx = b$$

- Resolver $Ly = b$
- Resolver $Ux = y$

Dado que se trata de dos matrices triangulares, los dos sistemas son sencillos de resolver y producen como resultado la solución buscada *x*. Además, una vez realizada la descomposición LU podemos cambiar el término independiente y resolver el sistema con un mínimo de operaciones adicionales (de orden $O(N^2)$ en lugar de $O(N^3)$).

El método puede refinarse de forma simple para incluir pivotaje, pero no expondremos aquí los detalles de su implementación sino que nos remitimos a Bonet (1996) para el alumno interesado en ellos. Por otra parte, la librería *Apache Math* incluye una implementación del método LU que puede usarse muy fácilmente de la forma siguiente:

- Crear la matriz y el vector de términos independientes

```
double[][] A = new double[N][N];
double[] b = new double[N];
…
```

- Crear un objeto de tipo *Array2DRealMatrix* a partir de la matriz A

```
Array2DRowRealMatrix rmA = new Array2DRowRealMatrix(A);
```

- Instanciar la clase *LUDecompositionImpl* usando este objeto

```
LUDecompositionImpl LU = new LUDecompositionImpl(rmA);
```

- La solución se obtiene usando el método getSolver() de la clase *LUDecompositionImpl* tomando el vector b como argumento

```
double[] solucion = LU.getSolver().solve(b);
```

Ejercicio 40: realizar un programa que genere sistemas de N ecuaciones aleatorias (rellenando los coeficientes de *A* y *b* con números aleatorios). Usar la implementación del método LU en *Apache Math* para resolver estos sistemas y estudiar como aumenta el tiempo de resolución con N (por ejemplo, generando y resolviendo diez sistemas para cada valor de N=100, 200,…,1000). Comprobar que el método LU es $O(N^3)$ para la factorización de A y de $O(N^2)$ para la resolución de sistemas una vez factorizada A.

Nota: aunque es improbable, debe tenerse en cuenta que alguno de los sistemas generados puede ser incompatible.

Sugerencia: usar *System.nanoTime()* para medir los tiempos de ejecución.

Además de la factorización LU existen muchos otros métodos de factorización directa. Algunos de estos métodos son de aplicación general (por ejemplo método de Doolittle o método de Crout) mientras que otros son especialmente eficaces para (o estan restringidos a) algunos tipos especiales de matrices A (por ejemplo el método de Cholesky para matrices simétricas). Nos remitimos a Bonet (1996) para más detalles.

3.4.3.5 Métodos iterativos

Los métodos expuestos hasta ahora están basados en algoritmos que resuelven el sistema de ecuaciones en un número finito de pasos. Por el contrario, los métodos iterativos parten de una aproximación inicial a la solución que se refina iterativamente hasta conseguir una solución “suficientemente aproximada”. Estos métodos son adecuados para matrices grandes y dispersas (con muchos elementos nulos) y remitimos al alumno interesado a Bonet (1996) para más detalles.

3.5 Interpolación de funciones

3.5.1 Planteamiento del problema

Frecuentemente se da en caso en problemas científicos o de ingeniería de que una función sólo se ha podido evaluar en un cierto número de puntos. En otras palabras, sólo se dispone de una tabla de valores de la función y no se puede (o es demasiado complicado) evaluar la función en otros puntos:

x	y
x_0	y_0
x_1	y_1
x_2	y_2
…	…
x_{N-1}	y_{N-1}

Los motivos de la dificultad (o imposibilidad) de evaluar la función en otros puntos pueden ser muy diversos. En algunos casos la tabla de valores corresponde a los resultados de un experimento, de forma que sólo puede obtenerse para un número finito de puntos. En otros casos la evaluación de la función es posible a priori para cualquier valor de x pero puede requerir recursos computacionales significativos, por lo que sólo resulta factible calcularla en un número limitado de puntos; por ejemplo, el cálculo de las efemérides (posiciones y velocidades) de los objetos del sistema solar requiere la integración de un complejo sistema de ecuaciones diferenciales, por lo que no es factible realizar dicha resolución cada vez que se requieren las efemérides para una fecha dada.

En estos casos puede recurrirse a **técnicas de interpolación** para obtener una aproximación al valor de la función entre los puntos disponibles. En esencia, estos métodos consisten en buscar una función *f** que coincida con *f* en los puntos disponibles y que constituya una aproximación razonable de la misma en los intervalos entre puntos, sustituyendo entonces la evaluación de *f* por la aproximación obtenida mediante la evaluación de *f**. Debe tenerse en cuenta entonces que cuando usemos técnicas de interpolación estaremos trabajando con *aproximaciones* y que, por lo tanto, hay que tener presente que en algunos casos estas pueden no ser buenas, como se muestra en el ejemplo de la figura:

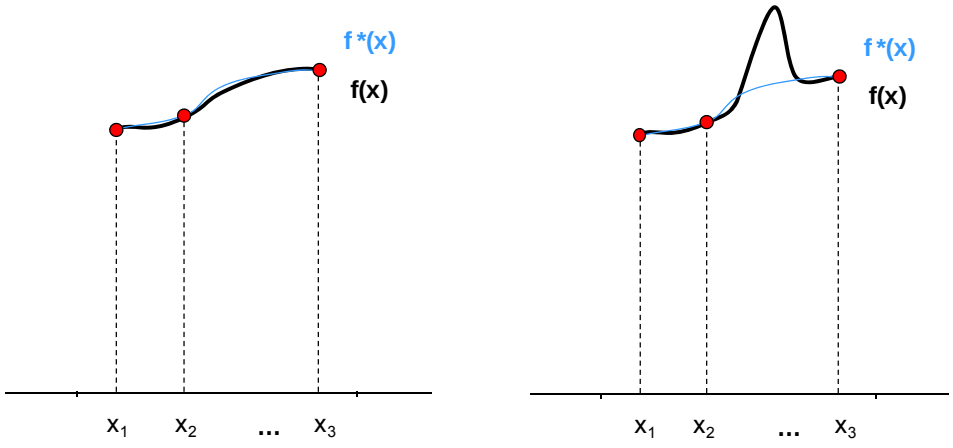


Figura 54. Aproximación por interpolación; caso favorable y caso desfavorable

Para construir la función *f** pueden usarse diferentes técnicas. Por ejemplo, en la interpolación polinómica *f** es un polinomio y en la interpolación trigonométrica *f** es una suma finita de senos y cosenos. Nos limitaremos aquí a estudiar dos tipos de interpolación, la interpolación polinómica y la interpolación por splines.

3.5.2 Interpolación polinómica

Como ya hemos mencionado, la interpolación polinómica consiste en hallar un polinomio *P(x)* que coincida en los puntos tabulados con la función problema *f(x)* para luego usar el polinomio como aproximación de la función entre los mismos. Nuestro punto de partida es por lo tanto el siguiente: dados N puntos donde tenemos evaluada la función *f(x)* buscamos el polinomio interpolador de grado N-1 que pase coincida con *f* en dichos puntos

$$\begin{cases} P(x_i) = f(x_i)\ i = 0, \dots, N-1 \\ P(x) = p_0 + p_1 x + p_2 x^2 + \dots + p_{N-1} x^{N-1} \end{cases}$$

Puede demostrarse (véase Bonet (1996)) que dadas estas condiciones siempre existe una solución y que esta solución es única. En otras palabras, el polinomio interpolador siempre existe y además es único.

3.5.2.1 *Evalución de polinomios*

Antes de abordar la discusión de los métodos para la determinación del polinomio interpolador realizaremos un inciso para discutir como implementar de forma óptima en un programa la evaluación de un polinomio. La solución más directa para esta implementación sería la translación literal de la forma canónica del polinomio usando la función que proporcione el lenguaje de programación para calcular potencias, en el caso de Java la función *Math.pow(), como se ilustra en el siguiente ejemplo*:

$$P(x) = p_0 + p_1 x + p_2 x^2 + \dots + p_{N-1} x^{N-1}$$
$$\downarrow$$

```
/**
* Método para la evaluación de un polinomio en su forma canónica
* usando la función Math.pow()
*
* @param p vector de coeficientes del polinomio, de grado menor a mayor
* @param x valor de x para la evaluación del polinomio
* @return P(x)
*/
 private static double evaluaPolinomio(double[] p, double x) {

        double valor=0;

        for( int i=0; i<p.length; ++i){
            valor = valor + p[i]*Math.pow(x,i);
        }

        return valor;
}
```

Sin embargo, es mejor evitar el uso de funciones de cálculo de potencias en el caso de exponentes enteros (como en el caso de polinomios) puesto que estas funciones están pensadas para evaluar potencias arbitrarias y están en general basadas en la evaluación de una serie truncada. En el caso de potencias enteras es mejor implementarlas

usando el producto, ganando así precisión y rapidez. En el caso de la evaluación de un polinomio esto puede realizarse reordenando los términos del polinomio de forma no canónica como sigue:

$$P(x) = p_0 + x\left(p_1 + x\left(p_2 + x(p_3 + \cdots)\right)\right)$$
$$\downarrow$$

```
/**
* Método para la evaluación de un polinomio usando sólo las operaciones
* producto y suma
*
* @param p vector de coeficientes del polinomio, de grado menor a mayor
* @param x valor de x para la evaluación del polinomio
* @return P(x)
*/
 private static double evaluaPolinomio(double[] p, double x) {

        double valor=0;

        for( int i=p.length-1; i>=0; --i){
          valor= valor*x + p[i];
        }

        return valor;
}
```

3.5.2.2 *Solución general*

La determinación de este polinomio consiste pues en hallar los coeficientes p_i dadas las condiciones anteriores. Puede verse fácilmente que este problema se reduce a un sistema de N ecuaciones lineales, una para cada punto tabulado, donde las incógnitas son los coeficientes p_i:

$$\begin{array}{ccccccccc} p_0 & + & p_1 x_0 & + & \cdots & + & p_{N-1} x_0^{N-1} & = & f(x_0) \\ p_0 & + & p_1 x_1 & + & \cdots & + & p_{N-1} x_1^{N-1} & = & f(x_1) \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ p_0 & + & p_1 x_{N-1} & + & \cdots & + & p_{N-1} x_{N-1}^{N-1} & = & f(x_{N-1}) \end{array}$$

Podemos reescribir el sistema en la notación matricial usada en la sección 3.4:

$$Ax = b$$

$$A = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \cdots & x_{N-1}^{N-1} \end{pmatrix} \quad x = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{N-1} \end{pmatrix} \quad b = \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{N-1}) \end{pmatrix}$$

Planteado en estos términos, el problema de hallar el polinomio interpolador se reduce al problema de la resolución del sistema de ecuaciones asociado a la matriz *A* y al vector de términos independientes *b* indicados, que podemos abordar usando los métodos expuestos en la sección 3.4: se construye la matriz *A* a partir de las abscisas de interpolación, el vector de términos independientes *b* a partir de los valores de la función y resolviendo el sistema se obtiene la solución *x* con los coeficientes del polinomio interpolador.

Ejercicio 41: escribir un programa que genere una tabla de interpolación de N puntos equiespaciados para la función $\sin(x)$ en el intervalo [0,4π]. A partir de esta tabla construir la matriz *A* y el vector *b* como se ha descrito y

resolver el sistema obtenido para hallar el polinomio interpolador de grado N-1. Comparar gráficamente el polinomio con la función $\sin(x)$ en el intervalo indicado para N=3, 4, … , 10.

La determinación de los coeficientes p_i a partir del sistema de ecuaciones asociado resuelve de forma general el problema de hallar el polinomio interpolador, siempre que se disponga de las rutinas de resolución de sistemas de ecuaciones lineales apropiadas. Sin embargo, para aquellos casos en que no se disponga de estas rutinas o bien sea necesario realizar una implementación propia compacta, presentamos a continuación métodos alternativos para hallar el polinomio interpolador. Téngase en cuenta que dada una tabla de interpolación el polinomio interpolador es único, por lo que todos los métodos proporcionan la misma solución.

3.5.2.3 Método de interpolación de Lagrange

Este es el método más simple de construir el polinomio interpolador, aunque no el más adecuado para su implementación en un programa. Sin embargo, nos será de gran utilidad en la sección dedicada a la integración numérica, por lo que lo presentamos aquí en detalle.

Dadas las N abscisas de interpolación $x_0, x_1, \dots, x_{N-1}$ definimos *N* polinomios, asociados respectivamente a cada una de las abscisas, de la forma siguiente:

$$L_i(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{i-1})(x-x_{x+1})\cdots(x-x_{N-1})}{(x_i-x_0)(x_i-x_1)\cdots(x_i-x_{i-1})(x_i-x_{x+1})\cdots(x_i-x_{N-1})} = \prod_{j\neq i}^{j=0\dots N-1} \frac{(x-x_j)}{(x_i-x_j)}$$

Nótese que son polinomios de grado N-1 i que tienen las siguientes propiedades:

$$L_i(x_j) = \begin{cases} 0 \; i \neq j \\ 1 \; i = j \end{cases} \; i,j = 0,\dots,N-1$$

Vemos que estos polinomios toman el valor 0 o 1 en los puntos de interpolación. Por lo tanto, podemos construir el polinomio interpolador a partir de ellos como:

$$P(x) = \sum_{i=0}^{N-1} f(x_i) L_i(x)$$

Puede comprobarse fácilmente que este polinomio verifica $P(x_i) = f(x_i)\,\forall i$, es decir, que es el polinomio interpolador que buscamos.

Ejercicio opcional: a partir de la tabla de interpolación calculada para la función $\sin(x)$ en el ejercicio anterior con N=4 construir (manualmente) el polinomio interpolador mediante el método de Lagrange y comprobar que los coeficientes coinciden con los obtenidos mediante la resolución del sistema de ecuaciones.

3.5.2.4 Método de interpolación de Newton

El método de Newton es algo más complicado que el de Lagrange, pero más adecuado para su implementación en un programa. Dado un valor de *N* prefijado permite obtener expresiones compactas para el cálculo de los coeficientes del polinomio interpolador que pueden implementarse directamente o, al contrario, permite implementar una rutina de interpolación de uso general (para valores de *N* arbitrarios) alternativa al método de resolución del sistema de ecuaciones asociado.

Dadas las *N* abscisas de interpolación $x_0, x_1, \dots, x_{N-1}$ el método de Newton parte de definir un polinomio de grado *N-1* de la forma siguiente:

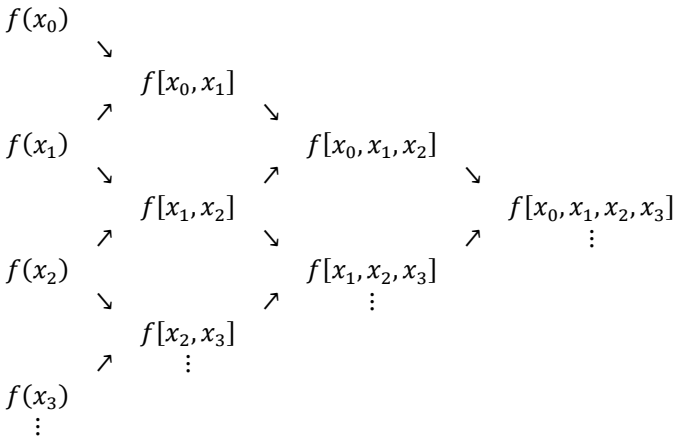
$$P(x) = C_0 + C_1(x-x_0) + C_2(x-x_0)(x-x_1) + \cdots + C_{N-1}(x-x_0)(x-x_1)\dots(x-x_{N-2})$$

A continuación, calcularemos los coeficientes C_i imponiendo que este polinomio sea el polinomio interpolador. Para ello tenemos que imponer las condiciones $P(x_i) = f(x_i)\ \forall i$. Empezando con la primera abscisa obtenemos:

$$P(x_0) = f(x_0) \rightarrow C_0 = f(x_0)$$
$$P(x_1) = f(x_1) \rightarrow C_1 = \frac{f(x_1) - C_0}{(x_1 - x_0)}$$
$$P(x_2) = f(x_2) \rightarrow C_2 = \frac{f(x_2) - C_0 - C_1(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)}$$
$$\vdots$$
$$P(x_i) = f(x_i) \rightarrow C_i = \frac{f(x_i) - C_0 - \sum_{j=1}^{j=i-1} C_j \prod_{k=0}^{k=j-1}(x_i - x_k)}{\prod_{j=0}^{j=i-1}(x_i - x_j)}$$
$$\vdots$$

Estas expresiones pueden implementarse fácilmente en un programa para calcular los coeficientes C_i. Nótese que no es necesario recalcular completamente el producto $\prod_{k=0}^{k=i-2}(x_i - x_k)$ para cada término del numerador de un coeficiente dado, sino que puede reutilizarse el resultado para el término anterior multiplicándolo por un término adicional $(x_i - x_k)$. Además, el resultado final puede usarse para el denominador. Una vez obtenidos todos los coeficientes C_i el polinomio interpolador queda determinado y no es necesario reordenarlo para expresarlo en la forma canónica $P(x) = p_0 + p_1x + p_2x^2 + \ldots + p_{N-1}x^{N-1}$: el polinomio puede evaluarse en el programa en la forma dada en función de los coeficientes C_i y las diferencias $(x - x_i)$ y el resultado será el mismo.

El cálculo de los coeficientes C_i puede realizarse de una forma alternativa mediante el llamado **método de las diferencias divididas**. Para ello construimos la siguiente tabla de valores (tabla de diferencias divididas)



Donde

$$f[x_i, x_j] = \frac{f(x_j) - f(x_i)}{x_j - x_i}$$
$$f[x_i, x_j, x_k] = \frac{f[x_j, x_k] - f[x_i, x_j]}{x_k - x_i}$$
$$f[x_i, x_j, x_k, x_l] = \frac{f[x_j, x_k, x_l] - f[x_i, x_j, x_k]}{x_l - x_i}$$
$$\vdots$$

Nótese que cada factor de la tabla se calcula a partir de los dos factores adyacentes en la columna a su izquierda ("diferencia dividida"). Puede comprobarse (Bonet (1996)) que con estas definiciones resulta que

$$C_0 = f(x_0)$$
$$C_1 = f[x_0, x_1]$$
$$C_2 = f[x_0, x_1, x_2]$$
$$C_3 = f[x_0, x_1, x_2, x_3]$$
$$\vdots$$

Por lo que el cálculo de la tabla de diferencias divididas es una alternativa para obtener los coeficientes C_i. Este procedimiento es muy cómodo para el cálculo manual de los coeficientes C_i, pero su implementación en un programa es algo más laboriosa que las fórmulas directas.

Ejercicio 42: escribir un programa que implemente el método de Newton, calculando los coeficientes C_i mediante uno de los dos métodos descritos. Aplicarlo a las mismas tablas de interpolación de N puntos equiespaciados en el intervalo [0,4π] para la función $\sin(x)$ usadas en ejercicios anteriores y comprobar que los resultados de la evaluación del polinomio coinciden con los obtenidos mediante el método de resolución del sistema de ecuaciones.

Nótese, para concluir, que si se añade un punto adicional a la tabla de interpolación no es necesario recalcular los coeficientes C_i ya obtenidos, sólo añadir uno adicional asociado al nuevo punto.

3.5.2.5 *Error de interpolación y abscisas de Chevichev*

Si la función *f(x)* a interpolar tiene derivadas continuas hasta orden *N* y *P(x)* es el polinomio interpolador de grado N-1 correspondiente a las abscisas $x_0, x_1, \dots, x_{N-1}$ puede demostrarse (Bonet (1996)) que el error de interpolación viene dado por la siguiente expresión:

$$f(x) - P(x) = \frac{f^{(N)}(\xi_x)}{N!}(x - x_0)(x - x_1)\dots(x - x_{N-1})$$

donde ξ es un cierto punto que depende de x y que se encuentra en el intervalo más pequeño que contiene $x_0, x_1, \dots, x_{N-1}$. Nótese que esta expresión sólo depende del valor de $f^{(N)}(\xi_x)$ y de las diferencias $(x - x_i)$. El primer término sólo depende de la función *f* y del punto de interpolación y por lo tanto no podemos controlarlo. Por el contrario, en caso de que podamos elegir las abscisas de interpolación libremente podemos escogerlas de forma que se minimice el producto de factores $(x - x_i)$ y reducir así el error de interpolación.

Nos planteamos por lo tanto el problema siguiente: **dada una función *f* que queremos interpolar en un intervalo [a,b] a partir de *N* abscisas de interpolación $x_0, x_1, \dots, x_{N-1}$, ¿cómo elegir estas abscisas de forma que $max_{x\in[a,b]}|(x - x_0)(x - x_1)\dots(x - x_{N-1})|$ sea lo menor posible y se minimice así el error de interpolación?**

Puede demostrarse (Bonet (1996)) que la selección óptima de abscisas de interpolación en [a,b] se obtiene a partir de las raíces de los polinomios de Chevichev. El polinomio de Chevichev de grado *M* se define de forma explícita como

$$T_M(x) = \cos(N\, arccos(x))$$

o de forma recurrente como:

$$T_M(x) = 2xT_{M-1}(x) - T_{M-2}(x)\ con\ T_0 = 1\ y\ T_1(x) = x$$

Estos polinomios tienen sus raíces en el intervalo [-1,1] y rescalándolas al intervalo [a,b] proporcionan las abscisas óptimas de interpolación de la forma siguiente:

$$x_k = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{2k+1}{2N}\pi\right)\ k = 0, \dots, N-1$$

Abscisas de interpolación de Chevichev

Usando estas expresiones para escoger las abscisas el error de interpolación se minimiza, ofreciendo los mejores resultados posibles.

Ejercicio 43: modificar el programa del ejercicio anterior que implementaba el método de Newton para usar abscisas de Chevichev en lugar de abscisas equiespaciadas para la interpolación de la función $\sin(x)$ en el intervalo [0,4π]. Comparar los resultados en ambos casos para N=3, 4, …, 10 y comprobar comparando numéricamente las diferencias respecto a la función (máxima, mínima y media), que los resultados son mejores usando las abscisas de Chevichev.

3.5.2.6 *Fenómeno de Runge*

En vista de los resultados de los ejercicios anteriores podría pensarse (erróneamente) que para conseguir mayor precisión en la interpolación de una función es suficiente aumentar el número de puntos de interpolación (es decir, el grado del polinomio interpolador). En general esto es falso puesto que puede demostrarse que normalmente los polinomios interpoladores $P_N(x)$ no convergen hacia la función $f(x)$ cuando N crece. Por el contrario, al aumentar N el polinomio puede "oscilar" de forma notable alejándose de la función, especialmente en los extremos del intervalo de interpolación, como muestra la figura:

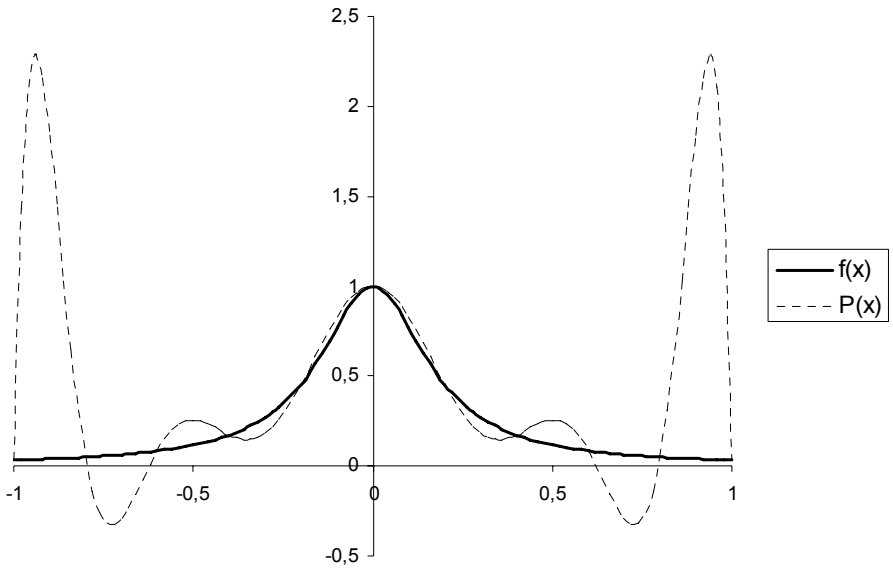


Figura 55. Fenómeno de Runge

Este fenómeno es típico de la interpolación con puntos equidistantes (se mitiga, aunque no desaparece, si se usan abscisas de Chevichev) de polinomios de grado elevado y se denomina **fenómeno de Runge**. Debido a este fenómeno hay que ser muy cuidadoso cuando se usan polinomios interpoladores de grado elevado, puesto que pueden producir resultados indeseados.

Ejercicio opcional: dada la función

$$f(x) = \frac{1}{1 + 30x^2}$$

determinar el polinomio interpolador con 11 puntos equiespaciados en el intervalo [-1,1], compararlo gráficamente con la función y comprobar que se reproduce la figura anterior. Comprobar que usando abscisas de Chevichev el resultado mejora, pero que aún se producen "oscilaciones" significativas en los bordes del intervalo.

3.5.3 Interpolación por "splines"

En caso de que estemos tratando un problema en el que los puntos de interpolación a usar sean muchos el riesgo de encontrarse con el fenómeno de Runge es elevado y, además, trabajar con polinomios de grado elevado es más complicado. En estos casos puede usarse una técnica alternativa de interpolación llamada **interpolación por splines**. El nombre "spline" proviene de un tipo de reglas flexibles que usaban los diseñadores gráficos (antes del advenimiento del diseño por ordenador) para trazar curvas suaves y continuas fijando algunos puntos de paso:

Figura 56. Regla flexible para el trazado de curvas "spline"

La interpolación por splines traduce este principio (curva continua con derivada continua que pasa por un cierto número de puntos dados) de forma matemática.

Una función spline de grado *p* que interpola una función *f* en un conjunto de puntos ordenados $x_0, x_1, \dots, x_{N-1}$ es una función *s(x)* tal que:

- $s(x_i) = f(x_i) \;\; i = 0, \dots, N-1$
- En cada intervalo $[x_i, x_{i+1}]$ la función $s(x)$ es un polinomio de grado p
- $s(x)$ y sus (p-1) primeras derivadas son continuas en $[x_0, x_{N-1}]$

Las splines más usadas y que trataremos aquí son las splines cúbicas. En este caso, en cada intervalo $[x_i, x_{i+1}]$ la función interpoladora $s(x)$ es un polinomio de grado 3 (distinto para cada intervalo):

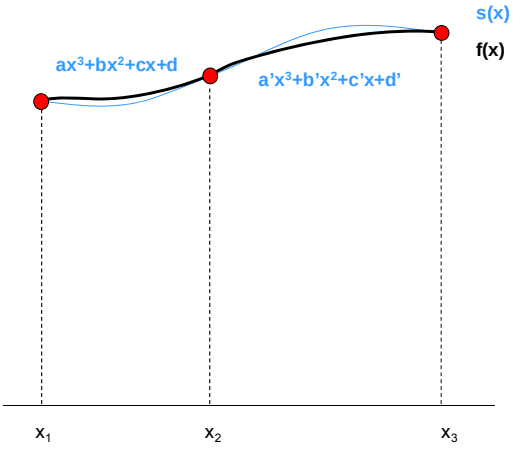


Figura 57. Interpolación por splines

Denominando $s_i(x)$ al polinomio de grado 3 que define la spline en el intervalo $[x_i, x_{i+1}]$ las condiciones para la determinación de s(x) son:

$$\left.\begin{array}{c} s_i(x_{i+1}) = s_{i+1}(x_{i+1}) = f(x_{i+1}) \\ s_i'(x_{i+1}) = s_{i+1}'(x_{i+1}) \\ s_i''(x_{i+1}) = s_{i+1}''(x_{i+1}) \end{array}\right\} i = 0, \dots, N-3$$

a las que hay que añadir dos condiciones suplementarias para que todos los coeficientes de los polinomios queden determinados. Usualmente se toma la condición de que las derivadas segundas se anulen en los extremos (splines naturales):

$$s''(x_0) = s''(s_{N-1}) = 0$$

Aunque no entraremos en detalles (referimos al alumno interesado a Bonet (1996) para más información) usando todas estas condiciones puede construirse un sistema de ecuaciones que determina los coeficientes de todos los polinomios $s_i(x)$ de la spline. A partir de ellos puede calcularse el valor de s(x) en cualquier punto del intervalo $[x_0, x_{N-1}]$ para usarlo como valor de interpolación.

La librería *Apache Math* proporciona una clase para la interpolación por splines de uso muy simple a partir de una tabla de interpolación expresada como dos vectores `x[],y[]` como puede verse en el ejemplo siguiente:

```
double x[] = { 0.0, 1.0, 2.0 };
double y[] = { 1.0, -1.0, 2.0);
UnivariateRealInterpolator interpolator = new SplineInterpolator();
UnivariateRealFunction function = interpolator.interpolate(x, y);
double interpolationX = 0.5;
double interpolatedY = function.value(interpolationX);
System.out println("f(" + interpolationX + ") = " + interpolatedY);
```

Ejercicio 44: usar Apache Math para construir un interpolador de spline con N=10 (puntos equiespaciados) de la función $\sin(x)$ en el intervalo [0,4π] y compararlo cuantitativamente con los resultados obtenidos en el caso de interpolación polinómica.

3.6 Aproximación de funciones: mínimos cuadrados lineales

3.6.1 Planteamiento del problema

El problema de la aproximación (o ajuste) de funciones está emparentado con el problema de la interpolación que ya hemos discutido en la sección 3.5.2. Como en dicho caso, el punto de partida es una tabla de valores de la función; sin embargo, en este caso no disponemos de los valores de la función en los puntos dados $f(x_i)$ sino de unos valores y_i que están sometidos a un cierto error

$$y_i = f(x_i) + \varepsilon_i$$

x	y
x_0	y_0
x_1	y_1
x_2	y_2
...	…
x_{N-1}	y_{N-1}

Este es el caso de, por ejemplo, valores obtenidos experimentalmente y que por tanto contienen un error (desconocido) de medida.

En estas situaciones no es razonable intentar ajustar una función *P(x)* imponiendo que pase por todos los *N* puntos de la tabla, puesto que trazaría no sólo la función *f(x)* sino también los errores, sino que el objetivo suele ser que *P(x)* "aproxime lo mejor posible" la función *f* subyacente. En términos de física experimental, el objetivo es buscar el "la función que mejor ajusta los datos disponibles" para así aproximar la función subyacente.

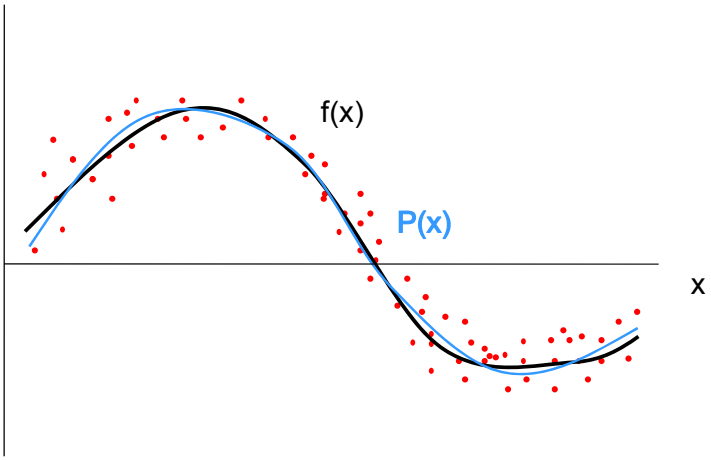


Figura 58. Aproximación de funciones

El primer paso en este proceso es definir la forma de la función *P(x)* con la que ajustaremos *f(x)*. El punto de partida es la forma funcional de *f(x)* que vendrá dada por un modelo físico que describa nuestro problema o una aproximación al mismo

$$y = f\left(x \middle| a_0, a_1, \ldots, a_{M-1}\right)$$

Usaremos entonces esta forma funcional para construir *P(x)*

$$P(x) = f(x|a_0^*, a_1^*, \dots, a_{M-1}^*)$$

donde los parámetros a_i^* son determinados de forma que ajusten lo mejor posible los datos experimentales. Por ejemplo, si fijamos el voltaje y medimos la intensidad en un circuito eléctrico sabemos que están relacionadas por la ley de Ohm. En este caso tenemos:

$$I = \frac{V}{R} \rightarrow f(x) = \frac{1}{R}x$$

y construimos

$$P(x) = a_1^* x$$

Los datos experimentales (V_i, I_i) incluirán errores de medida y tendrán un aspecto como el descrito en la figura siguiente

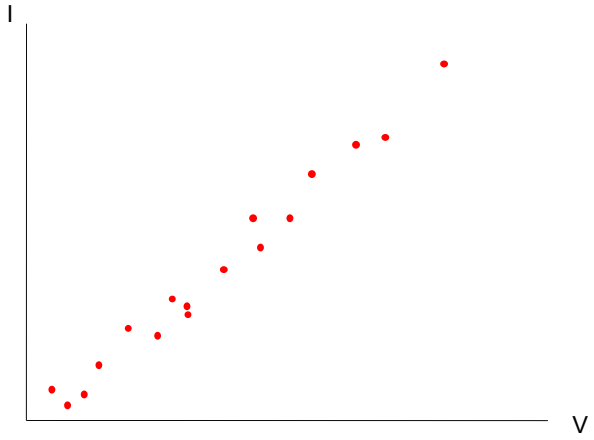


Figura 59. Ley de Ohm

Nuestro objetivo será entonces buscar el valor de a_1^* que "mejor ajusta estos datos" (la pendiente de la recta de ajuste) y tomaremos entonces $R \approx \frac{1}{a_1^*}$ como estimación de la resistencia.

En caso más general de problema de ajuste la dependencia de la función $f(x|a_0^*, a_1^*, \dots, a_{M-1}^*)$ en los parámetros a_i puede ser compleja. En estos casos debe recurrirse a métodos estadísticos de estimación (como por ejemplo el método de máxima verosimilitud) que escapan de los objetivos de esta asignatura.

Aquí nos centraremos en problemas de ajuste limitados a la siguiente condición: la función *P(x)* con la que realizaremos la aproximación tomará la forma de una combinación lineal de funciones

$$P(x) = a_0^*\phi_0(x) + a_1^*\phi_1(x) + \cdots + a_{M-1}^*\phi_{M-1}(x)$$

donde los parámetros a determinar en el ajuste son únicamente los coeficientes lineales a_i^*. Las funciones $\phi_i(x)$ pueden ser por ejemplo polinomios:

$$P(x) = a_0^* + a_1^*\mathrm{x} + a_2^*x^2 + \cdots + a_{M-1}^*x^{M-1}$$

o funciones trigonométricas

$$P(x) = a_0^* + a_1^*\sin(x) + a_2^*\sin(2x) + \cdots + a_{M-1}^* sin\big((M-1)x\big)$$

Una vez determinada la forma funcional de la función *P(x)* nos queda definir como realizaremos el ajuste, es decir, como determinaremos los valores de los parámetros a_i^* que "ajusten lo mejor posible los datos experimentales". Traduciremos matemáticamente esta condición imponiendo que la diferencia entre la función P(x) y los datos

experimentales y_i sea tan pequeña como sea posible. Nuestro punto de partida serán por lo tanto las diferencias entre *P(x)* y los datos:

$$d_i = y_i - P(x_i) \;\; i = 0, \dots, N-1$$

y buscaremos minimizar estas diferencias. Esta minimización puede imponerse de distintas formas: por ejemplo, imponiendo que las suma $\sum_i |d_i|$ sea mínima o bien que $\max_i |d_i|$ sea mínimo. Sin embargo, estos criterios llevan a ecuaciones difíciles de tratar para obtener los parámetros a_i^*, por lo que el criterio que se acostumbra a utilizar es la minimización de la suma de diferencias al cuadrado:

$$min \sum_{i=0}^{N-1} d_i^2 = min \sum_{i=0}^{N-1} \left[y_i - \left(a_0^* \phi_0(x) + a_1^* \phi_1(x) + \cdots + a_{M-1}^* \phi_{M-1}(x) \right) \right]$$

Ajuste por mínimos cuadrados

Este método de aproximación se denomina **mínimos cuadrados lineales** puesto que los parámetros a_i^* aparecen *únicamente* como términos lineales en *P(x)* y se obtienen por minimización de la suma anterior.

Finalmente, algunas consideraciones adicionales para concluir esta introducción:

- En los desarrollos presentados se ha tomado la hipótesis implícita de que en los datos experimentales disponibles la variable independiente *x* está libre de error (sólo se ha asumido un error en los valores y_i, no en los valores x_i). En la práctica esto suele ser una buena aproximación puesto que los errores en los valores x_i suelen ser significativamente menores que los errores en y_i, pero si no es el caso deben usarse otros métodos de ajuste.
- Como veremos más adelante, el problema de mínimos cuadrados lineales se resuelve a partir de un sistema de ecuaciones (ecuaciones normales). Puede ocurrir que el sistema esté mal condicionado (por ejemplo, cuando las medidas experimentales determinan sólo de forma marginal alguno de los parámetros) y que produzca resultados poco fiables. En estos casos debe recurrirse a técnicas de regularización del problema, en particular, a la selección de funciones $\phi_i(x)$ ortogonales (Bonet (1996)).
- La aproximación por mínimos cuadrados lineales es óptima si los errores experimentales no están correlacionados entre ellos, tienen media cero y una varianza constante. Si este no es el caso, deberían usarse otros métodos de estimación de los parámetros a_i^*.

3.6.2 Regresión lineal

El caso más sencillo, aunque de uso frecuente, de problema de ajuste por mínimos cuadrados lineales se da cuando tenemos una relación lineal:

$$P(x) = a_0^* + a_1^* X$$

En este caso la solución puede expresarse mediante fórmulas analíticas sencillas que permiten hallar la coordenada en el origen (a_0^*) y la pendiente (a_1^*) de esta relación lineal:

$$a_1^* = \frac{N \sum_{i=0}^{N-1} x_i y_i - \sum_{i=0}^{N-1} x_i \sum_{i=0}^{N-1} y_i}{N \sum_{i=0}^{N-1} x_i^2 - (\sum_{i=0}^{N-1} x_i)^2}$$

$$a_0^* = \frac{1}{N} \sum_{i=0}^{N-1} y_i - \frac{a_1^*}{N} \sum_{i=0}^{N-1} x_i$$

Regresión lineal

Además, los errores de estimación de estos parámetros pueden expresarse como sigue:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{N-1}(y_i - a_0^* - a_1^* x_i)^2}{N-2}}$$

$$\varepsilon(a_1^*) = \frac{\sigma\sqrt{N}}{\sqrt{N\sum_{i=0}^{N-1} x_i^2 - (\sum_{i=0}^{N-1} x_i)^2}}$$

$$\varepsilon(a_0^*) = \varepsilon(a_1^*)\sqrt{\frac{\sum_{i=0}^{N-1} x_i^2}{N}}$$

Ejercicio 45: dada la función *f(x) = 1 + 2x* escribir un programa que

1. Genere aleatoriamente N puntos x_i distribuidos uniformemente en el intervalo [0,1]
2. Genere aleatoriamente errores ε_i con una distribución gaussiana N(0,0.1) para cada uno de los puntos anteriores y calcule $y_i = f(x_i)+\varepsilon_i$ para cada uno de ellos.
3. Usando la tabla [x_i, y_i] realice una regresión lineal a partir de las fórmulas anteriores para N=10,100,1000.

Usando los resultados representar gráficamente los puntos *[x_i, y_i]* y el ajuste a los mismos para N=100 y comprobar que $a_0^* \to 1$ $a_1^* \to 2$ cuando N crece

Nota: para la generación de números aleatorios deberá usarse la clase *Random;* en particular, para la generación de números con una distribución uniforme deberá usarse *Random.nextDouble()* y para números con una distribución gausiana deberá usarse *Random.nextGaussian()*.

3.6.3 Método general de los mínimos cuadrados lineales: ecuaciones normales

Como ya hemos discutido, en el caso general del problema de mínimos cuadrados lineales la función *P(x)* toma la forma

$$P(x) = a_0^*\phi_0(x) + a_1^*\phi_1(x) + \cdots + a_{M-1}^*\phi_{M-1}(x)$$

y buscamos determinar los valores de los coeficientes de forma que P(x) ajuste los datos disponibles:

$$\begin{array}{ccccc} P(x_0) & = & a_0^*\phi_0(x_0) + a_1^*\phi_1(x_0) + \cdots + a_{M-1}^*\phi_{M-1}(x_0) & \approx & y_0 \\ P(x_1) & = & a_0^*\phi_0(x_1) + a_1^*\phi_1(x_1) + \cdots + a_{M-1}^*\phi_{M-1}(x_1) & \approx & y_1 \\ \vdots & & \vdots & & \vdots \\ P(x_{N-1}) & = & a_0^*\phi_0(x_{N-1}) + a_1^*\phi_1(x_{N-1}) + \cdots + a_{M-1}^*\phi_{M-1}(x_{N-1}) & \approx & y_{N-1} \end{array}$$

Podemos representar estas *N* expresiones en forma de un pseudo-sistema de ecuaciones de la forma siguiente:

$$\Phi a \approx y$$

$$\Phi = \begin{pmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_{M-1}(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_{M-1}(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_{N-1}) & \phi_1(x_{N-1}) & \cdots & \phi_{M-1}(x_{N-1}) \end{pmatrix} \quad a = \begin{pmatrix} a_0^* \\ a_1^* \\ \vdots \\ a_{M-1}^* \end{pmatrix} \quad y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix}$$

En este caso tratamos con un sistema de *N* ecuaciones y *M* incógnitas, con *N>M*. Por ejemplo, para una regresión lineal estas matrices tomarían la forma siguiente

$$\Phi = \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_{N-1} \end{pmatrix} \quad a = \begin{pmatrix} a_0^* \\ a_1^* \end{pmatrix} \quad y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix}$$

Es, por lo tanto, un sistema sobredeterminado y en general no tiene solución, es decir, no podemos conseguir que todas las ecuaciones se cumplan simultáneamente (no pueden ajustarse todos los puntos como en una interpolación). De todos modos, no es esto lo que pretendemos sino que queremos imponer la condición de mínimos cuadrados discutida anteriormente. Puede demostrarse (Bonet (1996)) que este problema tiene solución única y que esta se obtiene resolviendo el siguiente sistema de ecuaciones

$$(\Phi^T\Phi)a = \Phi^T y$$

Sistema de ecuaciones normales

Este es un sistema de *M* ecuaciones con *M* incógnitas cuya solución nos proporciona la estimación por mínimos cuadrados de los parámetros a_i^*. Para resolverlo podemos calcular el producto $\Phi^T\Phi$ y la matriz Φ^T y resolver el sistema de ecuaciones normales usando los procedimientos descritos en la sección 3.4, pero los métodos presentados en dicha sección no son los más adecuados para resolver el tipo de sistemas resultantes, por lo que recurriremos a la librería *Apache Math* que proporciona herramientas específicas para hacer ajustes por mínimos cuadrados que serán los que usaremos aquí. En particular usaremos la clase *QRDecompositionImpl* que implementa el método de descomposición QR para la resolución de sistemas de ecuaciones.

Si la clase *QRDecompositionImpl* se usa con un sistema de ecuaciones *NxN* proporciona la solución exacta al sistema (si existe), mientras que si se usa con un sistema sobredeterminado (como es nuestro caso) proporciona el resultado de un ajuste por mínimos cuadrados. El procedimiento de uso de esta clase es como sigue:

- Crear y rellenar dos vectores que contengan los datos experimentales

```
double[] x = new double[N];
double[] y = new double[N];
…
```

- Crear la matriz Φ y rellenarla a partir del vector *x*, por ejemplo en el caso de una regresión lineal

```
double[][] Phi = new double[N][2];
for( int i=0; i<N; ++i ){
    Phi[i][0] = 1.;
    Phi[i][1] = x[i];
}
```

- Instanciar la clase *QRDecompositionImpl* usando la matriz Φ; para ello hay que convertir la matriz en un objeto de tipo *Array2DRowRealMatrix*

```
Array2DRowRealMatrix rmPhi = new Array2DRowRealMatrix(Phi);
QRDecompositionImpl QR = new QRDecompositionImpl(rmPhi);
```

- La solución se obtiene usando el método getSolver() de la clase *QRDecompositionImpl* tomando el vector y como argumento

```
double[] solucion = QR.getSolver().solve(y);
```

Ejercicio 46: tomando los datos generados para el ejercicio anterior de regresión lineal, usar este método para realizar el ajuste por mínimos cuadrados y comprobar que se obtienen los mismos resultados.

Usando este procedimiento pueden realizarse fácilmente ajustes por mínimos cuadrados de cualquier combinación lineal de funciones de la forma

$$P(x) = a_0^*\phi_0(x) + a_1^*\phi_1(x) + \cdots + a_{M-1}^*\phi_{M-1}(x)$$

Sólo recordar, para concluir, que cuando se use este método deben tenerse en cuenta las limitaciones descritas en la introducción de esta sección.

Ejercicio 47: dada la función *f(x) = 1 + 2x + 3 sin(x)* escribir un programa que

1. Genere aleatoriamente N puntos x_i distribuidos uniformemente en el intervalo $[0,\pi]$
2. Genere aleatoriamente errores ε_i con una distribución gaussiana N(0,0.1) para cada uno de los puntos anteriores y calcule $y_i = f(x_i)+\varepsilon_i$ para cada uno de ellos.
3. Usando la tabla $[x_i, y_i]$ realice una regresión lineal con N=10,100,1000

Usando los resultados representar gráficamente los puntos $[x_i, y_i]$ y el ajuste a los mismos para N=100 y comprobar que $a_0^* \to 1$ $a_2^* \to 2$ $a_3^* \to 3$ cuando N crece

3.7 Integración

3.7.1 Planteamiento del problema

En esta sección trataremos el problema del cálculo de integrales definidas:

$$\int_a^b f(x)dx$$

Dada una función *f(x)* su integral definida se puede definir informalmente como el área en el plano *xy* delimitada por el gráfico de la función, teniendo en cuenta que las áreas en la región y>0 se toman como positivas y las áreas en la región y<0 se toman como negativas:

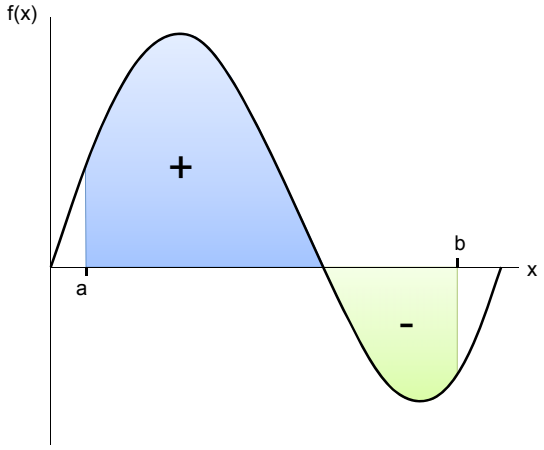


Figura 60. Integración como el cálculo de área

Sin embargo, para abordar el tratamiento numérico de este problema partiremos de la definición de la **integral de Riemann**[14] como base para nuestro desarrollo. Dejamos para la asignatura de *Cálculo de una variable* los detalles de esta definición y sólo presentaremos aquí una breve introducción recordatoria de la misma.

Dada una función continua *y=f(x)* en el intervalo [a,b], sean los siguientes puntos del eje x:

$$a = x_0 < x_1 < x_2 < \cdots < x_{N-1} < x_N = b \quad x_{i+1} - x_i = \Delta x$$

que dividen el intervalo *[a,b]* en *N* subintervalos iguales[15]. Seleccionamos ahora *N* puntos, uno dentro de cada subintervalo

$$c_0, c_1, \ldots, c_{N-1} \quad x_i < c_i < x_{i+1}$$

La integral de Riemann se define entonces como:

$$\int_a^b f(x)dx = \lim_{N\to\infty} \sum_{i=0}^{N-1} f(c_i)\Delta x$$

En términos más gráficos, podemos ver que esta definición es equivalente a aproximar el área de la función en un intervalo $[x_i, x_{i+1}]$ mediante un rectángulo, de forma que el sumatorio aproxima el área total de la función. Al hacer

[14] Existen definiciones más generales de integral, como la Integral de Lebesgue, pero para el tratamiento de integrales de funciones reales la definición de Riemann es suficiente

[15] Por simplicidad estamos suponiendo todos los intervalos iguales, de longitud *Δx*, pero esto no es necesario para la definición

tender *Δx* a cero, el error de la aproximación en el sumatorio se va reduciendo hasta que, en el límite, coincide con el área total de la función:

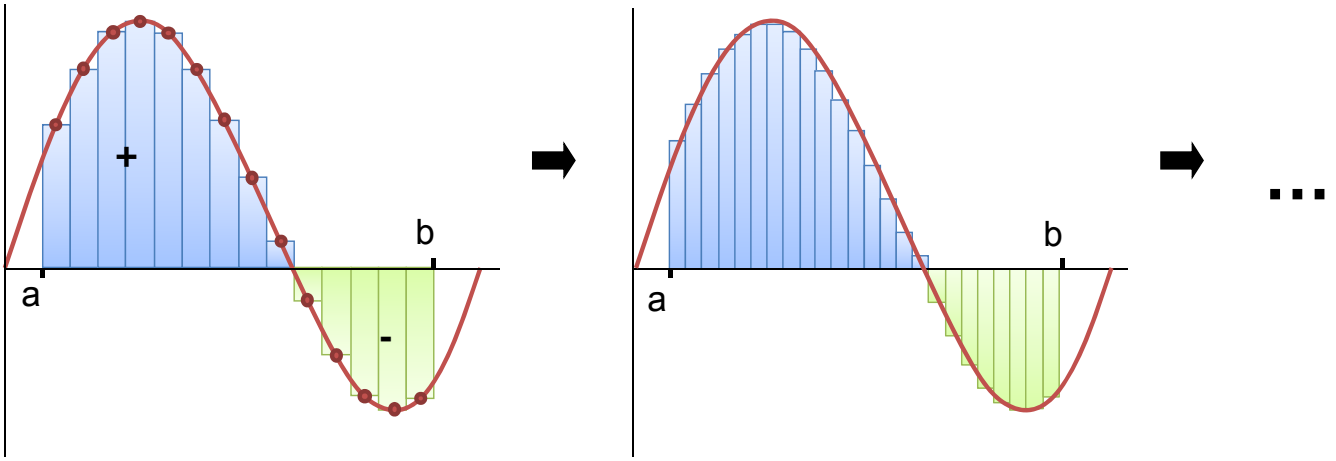


Figura 61. integral de Riemann

La propia definición de la integral de Riemann puede ya servirnos para definir un método de integración numérica; si tomamos uno de los pasos de la serie que tiene como límite la integral, nos proporcionará una aproximación a la misma:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{N-1} f(c_i)\Delta x$$

Si *N* es lo suficientemente grande (o de forma equivalente *Δx* lo suficientemente pequeño) el sumatorio tomará un valor arbitrariamente próximo a la integral. Podemos formularlo de forma más clara para su programación de la forma siguiente:

- Dado el intervalo [a,b] lo dividimos en N subintervalos iguales cuya longitud *Δx* será:

$$h = \frac{b-a}{N}$$

- Para cada subintervalo tomamos c_i como su punto medio, de forma que

$$c_i = a + ih + \frac{h}{2} \quad i = 0, \dots, N-1$$

- La aproximación con N intervalos de la integral será entonces

$$\int_a^b f(x)dx \approx h\sum_{i=0}^{N-1} f(c_i)$$

Este cálculo puede implementarse fácilmente como ejercicio de programación, pero no es un buen método numérico de integración; en su lugar deben usarse los métodos que se exponen a continuación.

Ejercicio 48: escribir un programa que implemente la fórmula anterior para la integral

$$\int_0^1 e^x dx$$

y comprobar que el resultado converge hacia *e - 1* a medida que se aumenta N.

3.7.2 Integración por trapecios

El método de integración por trapecios es un primer paso en el refinamiento de la (poco eficiente) fórmula anterior basada directamente en la definición de integral de Riemann. Manteniendo el concepto básico de dividir el intervalo *[a,b]* en *N* subintervalos y aproximar el área de la función en cada uno de ellos,

$$\int_a^b f(x)dx \approx \sum_{i=0}^{N-1} A_i$$

mejoramos esta aproximación sustituyendo los rectángulos usados anteriormente por trapecios que se ajustan a los valores de la función en los extremos del intervalo, como se describe en la siguiente figura:

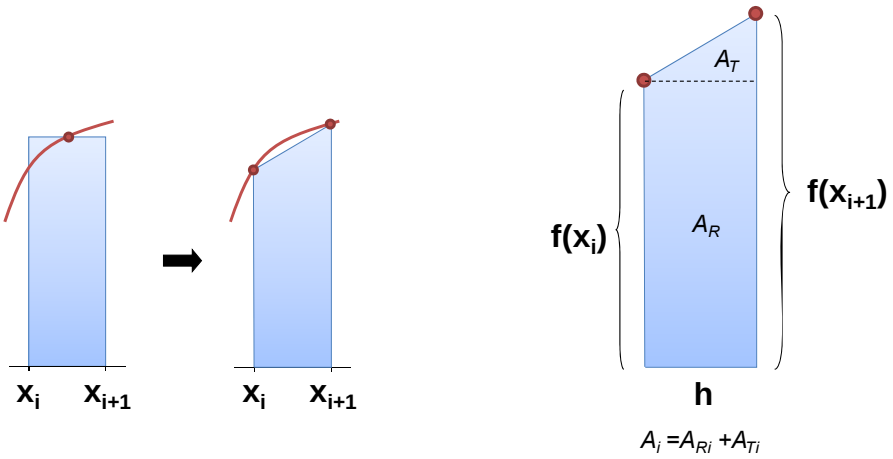


Figura 62. Integración por trapecios (I)

En este caso, el área A_i correspondiente al trapecio del i-ésimo intervalo se calcula como:

$$A_i = A_R + A_T = hf(x_i) + \frac{h}{2}\big(f(x_{i+1}) - f(x_i)\big) = \frac{h}{2}\big(f(x_i) + f(x_{i+1})\big)$$

Y a continuación sólo nos queda sumar las áreas de todos los trapecios para aproximar el área total, como se representa en la figura:

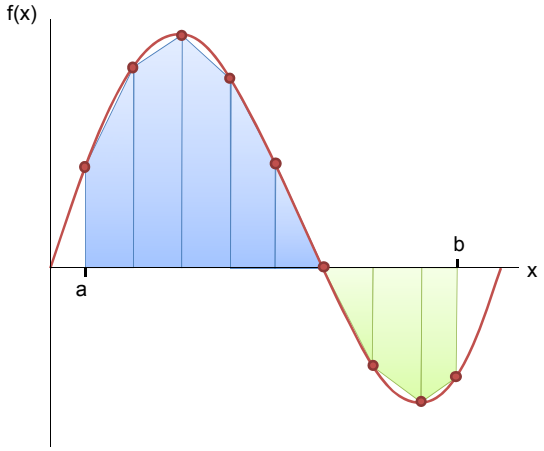


Figura 63. Integración por trapecios (II)

$$\int_a^b f(x)dx \approx \sum_{i=0}^{N-1} A_i = \frac{h}{2}\sum_{i=0}^{N-1}\big(f(x_i) + f(x_{i+1})\big)$$

Podemos formularlo de forma más conveniente para su programación de la forma siguiente:

- Dado el intervalo [a,b] lo dividimos en N subintervalos iguales cuya longitud *Δx* será:

$$h = \frac{b-a}{N}$$

- Podemos expresar los límites de los subintervalos como

$$x_i = a + ih \quad i = 0, \dots, N$$

- Reordenando los términos del sumatorio teniendo en cuenta la expresión anterior obtenemos la fórmula de integración por trapecios:

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{i=0}^{N-1}\big(f(a+ih) + f(a+(i+1)h)\big)$$

Esta fórmula normalmente se expresa reordenando los términos del sumatorio de la forma siguiente, que requiere menos evaluaciones de la función *f*:

$$\int_a^b f(x)dx \approx \frac{h}{2}\left(f(a) + 2\sum_{i=0}^{N-1} f(a+ih) + f(b)\right)$$

Fórmula de integración por trapecios

aunque en la literatura puede encontrarse frecuentemente la siguiente expresión, equivalente a la anterior pero menos apropiada para su programación:

$$\int_a^b f(x)dx \approx \frac{h}{2}\big(f(a) + 2f(a+h) + af(a+2h) + \cdots + 2f(b-h) + f(b)\big)$$

La fórmula de integración por trapecios es el método de integración numérica más sencillo, aunque normalmente sea conveniente recurrir a métodos más sofisticados como los que se describen más adelante.

Para concluir, puede demostrarse (Bonet (1996)) que el error en la estimación del área de la función de uno de los intervalos viene dado por:

$$\varepsilon(A_i) = \left|\frac{h^3}{12}f''(c_i)\right| \quad c_i \in (x_i, x_{i+1})$$

Donde el punto c_i es un punto (indeterminado) en el intervalo. A partir de esta expresión podemos calcular el error total de la integración por trapecios usando N subintervalos como:

$$\varepsilon(A) = \sum_{i=1}^{N}\varepsilon(A_i) = \frac{h^3}{12}\sum_{i=1}^{N}|f''(c_i)|$$

Finalmente, si tomamos un valor c^{max} tal que $f(c^{max})$ sea máxima en *(a,b)* tendremos

$$|f(c_i)| \leq |f(c^{max})| \quad \forall i$$

y por lo tanto podemos obtener la siguiente cota a partir de la expresión de $\varepsilon(A)$:

$$\varepsilon(A) = \left|\frac{Nh^3}{12}f''(c^{max})\right| \quad |f''(c^{max})| \text{ máximo en } (a,b)$$

Esta fórmula nos permite acotar el error de integración para un valor de N dado o, al contrario, determinar el mínimo valor de N necesario para obtener un resultado con un error máximo prefijado.

Ejercicio 49: escribir un programa que implemente la fórmula de integración por trapecios para la integral

$$\int_0^1 e^x dx$$

Calcular para cada valor de N la cota del error de integración y compararla con el error real.

3.7.3 Método de Simpson

El método de Simpson refina la integración por trapecios mediante una mejora de la fórmula de aproximación del área A_i en un intervalo. Para ello en lugar de aproximar la función mediante una recta la aproximamos mediante un polinomio de segundo grado *P(x)*, como se muestra en la figura:

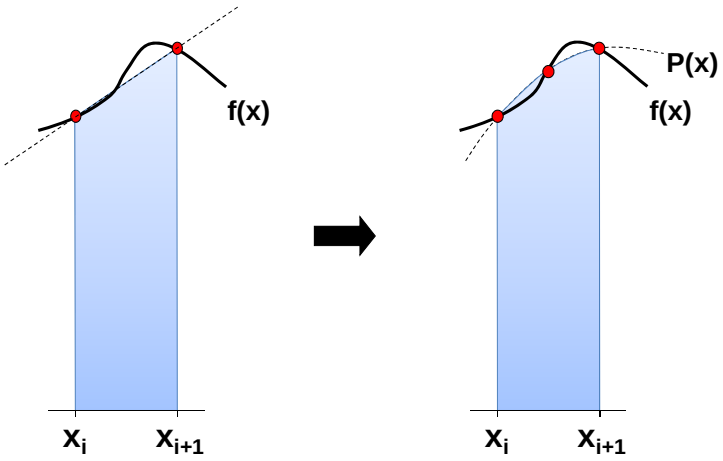


Figura 64. Método de integración de Simpson

El polinomio se ajusta a los valores de la función f en los extremos y el punto medio del intervalo. El área de la figura resultante se obtiene por integración del polinomio en $[x_i, x_{i+1}]$ y, como discutiremos en el apartado siguiente (sección 3.7.4) el resultado de esta integración es:

$$A_i = \frac{h}{6}\left(f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1})\right)$$

y sumando las áreas de todos los intervalos:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{N-1} A_i = \frac{h}{6}\sum_{i=0}^{N-1}\left(f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1})\right)$$

De nuevo, expresaremos esta fórmula de forma más conveniente para su programación:

- Dado el intervalo [a,b] lo dividimos en N subintervalos iguales cuya longitud Δx será:

$$h = \frac{b-a}{N}$$

- Podemos expresar los límites de los subintervalos como

$$x_i = a + ih \quad i = 0, \ldots, N$$

- Reordenando los términos del sumatorio teniendo en cuenta la expresión anterior obtenemos la fórmula de integración de Simpson:

$$\int_a^b f(x)dx \approx \frac{h}{6}\sum_{i=0}^{N-1}\left(f(a+ih)+4f\left(a+ih+\frac{h}{2}\right)+f(a+(i+1)h)\right)$$

Podemos expresar esta fórmula reordenando los términos del sumatorio de la forma siguiente, que requiere menos evaluaciones de la función *f*:

$$\int_a^b f(x)dx \approx \frac{h}{6}\left(f(a)+2\sum_{i=1}^{N-1}f(a+ih)+4\sum_{i=0}^{N-1}f\left(a+ih+\frac{h}{2}\right)+f(b)\right)$$

Fórmula de integración de Simpson

En este caso (Bonet (1996)) el error en la estimación del área de la función de uno de los intervalos viene dado por la siguiente expresión:

$$\varepsilon(A_i)=\left|\frac{h^5}{2880}f^{(4)}(c_i)\right| \quad c_i \in (x_i, x_{i+1})$$

Que de forma similar al caso de la integración por trapecios permite obtener una cota superior al error de integración de la fórmula de Simpson:

$$\varepsilon(A_i)=\left|\frac{Nh^5}{2880}f^{(4)}(c^{max})\right| \quad \left|f^{(4)}(c^{max})\right| \text{ máximo en } (a,b)$$

Nótese que el error decrece como $\frac{1}{N^4}$ mientras que en la integración por trapecios decrece como $\frac{1}{N^2}$, por lo que el método de Simpson es más eficiente al permitir conseguir el mismo error con un número menor de intervalos, aunque debe tenerse en cuenta también la dependencia del error en las derivadas de *f*.

Ejercicio 50: escribir un programa que implemente la fórmula de integración de Simpson para la integral

$$\int_0^1 e^x dx$$

Calcular para cada valor de N la cota del error de integración y compararla con el error real; comparar los resultados con los de la aplicación de la integración por trapecios y comprobar que la convergencia es más rápida.

3.7.4 Generalización: fórmulas de Newton-Cotes

En los apartados anteriores hemos discutido las fórmulas de integración de Riemann, de trapecios y de Simpson. Puede verse fácilmente que estas tres fórmulas pueden considerarse casos particulares de un método más general, pues en los tres métodos aproximamos la función en un intervalo $[x_i, x_{i+1}]$ mediante un polinomio: de grado cero en la fórmula de Riemann, de grado uno en la fórmula de trapecios y de grado dos en la fórmula de Simpson. La generalización de este procedimiento se denomina **fórmulas de Newton-Cotes**, y desarrollaremos a continuación su formulación general.

Como hemos dicho, el fundamento de las fórmulas de Newton-Cotes es aproximar la función a integrar en un intervalo $[x_i, x_{i+1}]$ mediante un polinomio de grado K. Para realizar esta aproximación usaremos un *polinomio interpolador de grado M* en dicho intervalo, es decir, un polinomio que coincide con el valor de la función a integrar

en M puntos dados del intervalo. En nuestro caso tomaremos puntos equiespaciados en $[x_i, x_{i+1}]$ como se describe en la figura.

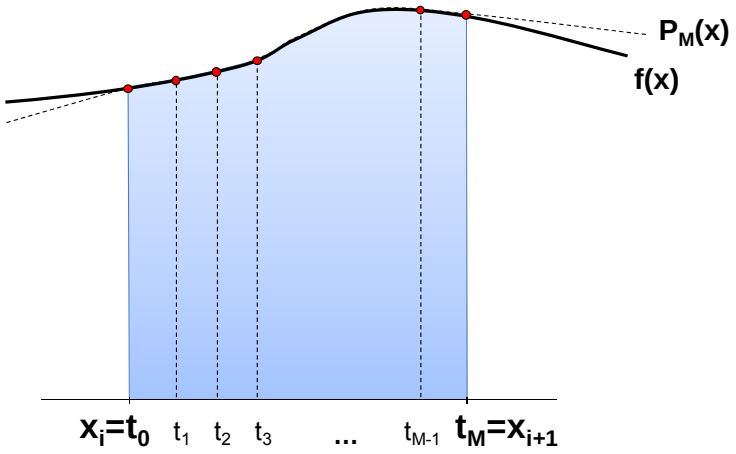


Figura 65. Fórmulas de integración de Newton-Cotes

donde

$$x_{i+1} - x_i = h = \frac{b-a}{N} \quad t_j = x_i + j\frac{h}{M}$$

Tomando el polinomio interpolador $P_M(x)$ como aproximación de la función en $[x_i, x_{i+1}]$ tenemos:

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx A_i = \int_{x_i}^{x_{i+1}} P_M(x)dx$$

donde $P_M(x)$ viene definido por las siguientes condiciones:

$$P_M(t_j) = f(t_j) \quad j = 0, \dots, M$$

y la fórmula de interpolación de Lagrange (ver sección 3.5.2.3) nos permite calcularlo como:

$$P_M(x) = \sum_{j=0}^{M} f(t_j)L_j(x) \quad con \quad L_j(x) = \prod_{k=0, k\neq j}^{k=M} \frac{x-t_k}{t_j-t_k}$$

Por lo tanto:

$$\int_{x_i}^{x_{i+1}} P_M(x) = \sum_{j=0}^{M} f(t_j) \int_{x_i}^{x_{i+1}} L_j(x)dx$$

Podemos ahora expresar la integral de los polinomios de Lagrange mediante un cambio de variable que tenga en cuenta que los puntos t_j están equiespaciados:

$$x = x_i + \frac{h}{M}\tau \rightarrow \left\{ \begin{matrix} \frac{x-t_k}{t_i-t_k} = \frac{\tau-k}{i-k} \\ dx = \frac{h}{M}d\tau \end{matrix} \right\} \rightarrow \int_{x_i}^{x_{i+1}} L_j(x)dx = \frac{h}{M}\int_0^M \prod_{k=0, k\neq j}^{k=M} \frac{\tau-k}{j-k}d\tau \equiv \frac{h}{M}a_j$$

Nótese que hemos definido unos coeficientes a_j que no dependen de x_i, x_{i+1} ni de la función f, sólo dependen de M. Además, resultan ser números racionales. A partir de ellos podemos expresar la aproximación resultante a la integral como

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \int_{x_i}^{x_{i+1}} P_M(x)dx = \frac{h}{M}\sum_{j=0}^{M} a_j f(t_j)$$

Es decir:

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{h}{M}\sum_{j=0}^{M} a_j f\left(x_i + j\frac{h}{M}\right)$$

Fórmulas de Newton-Cotes

Los coeficientes a_j solo tienen que calcularse una única vez, puesto que sólo dependen de M. Por ejemplo, para M=1 (integración por trapecios) tenemos:

$$a_0 = \int_0^1 \prod_{k=0,k\neq 0}^{k=1} \frac{\tau - k}{j - k} d\tau = \int_0^1 \frac{\tau - 1}{0 - 1} d\tau = \int_0^1 (1 - \tau) d\tau = \left[t - \frac{t^2}{2}\right]_0^1 = \frac{1}{2}$$

$$a_1 = \int_0^1 \prod_{k=0,k\neq 1}^{k=1} \frac{\tau - k}{j - k} d\tau = \int_0^1 \frac{\tau - 0}{1 - 0} d\tau = \int_0^1 \tau d\tau = \left[\frac{t^2}{2}\right]_0^1 = \frac{1}{2}$$

Y por lo tanto recuperamos la fórmula de integración por trapecios:

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{h}{2}\left(f(x_i) + f(x_{i+1})\right)$$

La tabla siguiente presenta los valores de los coeficientes de Newton-Cotes para diversos valores de M, así como el nombre con que se conoce la fórmula resultante y la cota de error del resultado:

Tabla 16. Coeficientes de las fórmulas de integración de Newton-Cotes

M	Nombre	a_0	a_1	a_2	a_3	a_4	a_5	a_6	Cota error
1	Trapecios	$\frac{1}{2}$	$\frac{1}{2}$						$\left\|\frac{h^3}{12} f^{(2)}(c^{max})\right\|$
2	Simpson	$\frac{1}{3}$	$\frac{4}{3}$	$\frac{1}{3}$					$\left\|\frac{h^5}{2880} f^{(4)}(c^{max})\right\|$
3	Regla 3/8	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{3}{8}$				$\left\|\frac{h^5}{6480} f^{(4)}(c^{max})\right\|$
4	Milne	$\frac{28}{90}$	$\frac{128}{90}$	$\frac{48}{90}$	$\frac{128}{90}$	$\frac{28}{90}$			$\left\|\frac{h^7}{1935360} f^{(6)}(c^{max})\right\|$
5	-	$\frac{95}{288}$	$\frac{375}{288}$	$\frac{250}{288}$	$\frac{250}{288}$	$\frac{375}{288}$	$\frac{95}{288}$		$\left\|\frac{257h^7}{945000000} f^{(6)}(c^{max})\right\|$
6	Weddle	$\frac{41}{140}$	$\frac{216}{140}$	$\frac{27}{140}$	$\frac{272}{140}$	$\frac{27}{140}$	$\frac{216}{140}$	$\frac{41}{140}$	$\left\|\frac{h^9}{1567641600} f^{(8)}(c^{max})\right\|$

Como hemos hecho en el caso de la integración por trapecios y por Simpson, aplicando estas fórmulas estas fórmulas a los *N* intervalos en que hemos dividido el intervalo y sumando los resultados obtendremos la integral en *[a,b]*; debe recordarse en este caso que la cota de error debe multiplicarse por *N* para obtener el error total.

En principio pueden calcularse los coeficientes a_j para valores de M mayores pero, aunque las cotas de error se reducirían, no es conveniente hacerlo puesto que algunos de los coeficientes a_j son negativos, por lo que no son adecuados para un método numérico al producirse cancelaciones que incrementan el error numérico.

3.7.5 Método de Romberg

Usando las fórmulas de cota de error presentadas en las secciones anteriores podemos fijar el error máximo deseado y deducir el número de intervalos N necesario para calcular el valor de la integral con un método dado. Sin embargo, para ello es necesario conocer el valor de $f^{(n)}(c^{max})$ que aparece en dichas fórmulas. Esto no siempre es fácil o posible, y en cualquier caso dificulta enormemente realizar un programa que calcule la integral de forma autónoma dada una función arbitraria.

Podemos, sin embargo, usar una aproximación alternativa para determinar cual es el valor de N que proporciona la integral fijado un error máximo. Efectivamente, las fórmulas que dan las cotas de error nos indican también como convergen los resultados hacia el valor de la integral en función de N. Por ejemplo, podemos ver que la integración por trapecios converge como $1/N^2$, lo que podemos comprobar con el ejemplo de la siguiente figura obtenida a partir del ejercicio propuesto en la sección 3.7.2:

$$\int_0^1 e^x dx$$

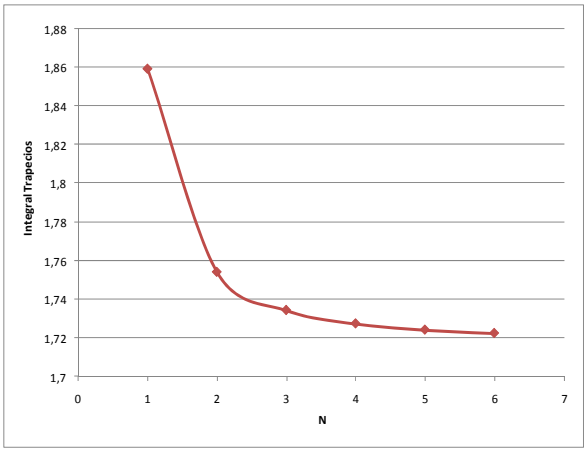


Figura 66. Método de integración de Romberg

El **método de Romberg** usa este conocimiento del comportamiento de la convergencia de los resultados de la integración por trapecios para optimizar el cálculo de la integral de dos formas:

4. Dados varios resultados para valores de N crecientes extrapola su tendencia para estimar el resultado correcto más rápidamente (extrapolación de Richardson):

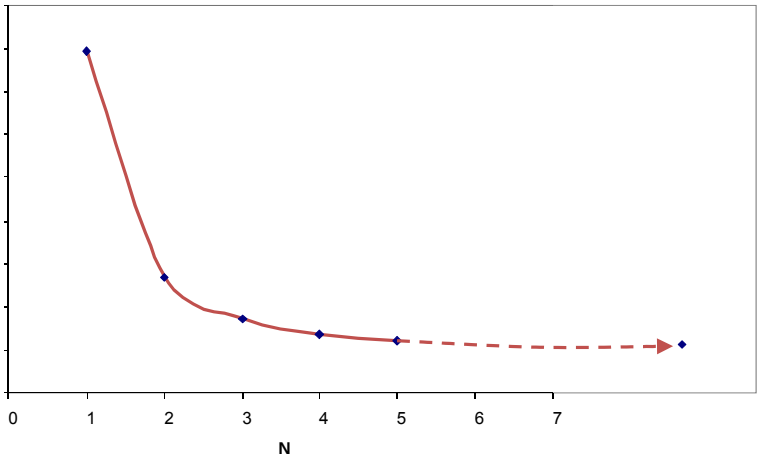


Figura 67. Extrapolación de Richardson

5. Construye las estimaciones sucesivas de forma que la diferencia entre dos de ellas proporciona una cota del error de la siguiente, lo cual permite determinar cuándo parar el proceso sin necesidad de conocer las características particulares de la función con que se trabaja (por ejemplo las derivadas).

No entraremos aquí en los detalles de los fundamentos matemáticos del método[16] y o su implementación[17] (bastante sencilla una vez se dispone de una implementación de la integración por trapecios) y nos limitaremos a usar la implementación del mismo que proporciona la librería *Apache Math*.

Esta implementación se basa en la clase *RombergIntegrator* cuyo método *integrate()* realiza el cálculo dada una función y un intervalo *[a,b]*:

```
double integrate(UnivariateRealFunction f, double a, double b)
```

Para usar este método debe definirse una clase que implemente el interfaz *UnivariateRealFunction*; este interfaz es muy sencillo y sólo requiere la implementación de un método *value()* que calcule el valor de la función para un *x* dado. El ejemplo siguiente implementa una la función $f(x) = e^x$:

```
/**
 * Función a integrar
 */
public static class Funcion implements UnivariateRealFunction {

    /**
     * Retorna el valor de la función
     *
     * @param x valor para la evaluación de la función
     * @return exp(x)
     */
    public double value(double x) {
        return Math.exp(x);
    }
}
```

[16] Remitimos al lector interesado en dichos fundamentos a http://www.daimi.au.dk/~oleby/NA2/romberg.pdf

[17] Puede encontrarse una breve descripción en la Wikipedia http://es.wikipedia.org/wiki/M%C3%A9todo_de_Romberg

Ejercicio 51: escribir un programa que use la implementación del método de Romberg en Apache Math para calcular la integral

$$\int_0^1 e^x dx$$

Ajustar los parámetros de *RombergIntegrator* para conseguir que el resultado sea correcto con el máximo de cifras decimales disponibles en una variable de tipo *double*. Para ello deberán usarse los métodos *setAbsoluteAccuracy()* y *setMinimalIterationCount()*.

3.7.6 Método de Legendre-Gauss

El método de Legendre-Gauss se basa en el mismo principio que el de Newton-Cotes: la integral de la función en un intervalo se aproxima como la suma ponderada de valores de la función en puntos especificados del intervalo:

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \sum_{j=0}^{M} \omega_j f(t_j)$$

donde los pesos ω_i se definen imponiendo que el resultado de la integral sea exacto para un polinomio de un cierto grado. Este tipo de formulas para la integración se denominan **cuadraturas** y la diferencia respecto al método de Newton-Cotes es que el método de Legendre-Gauss usa las denominadas cuadraturas de Gauss:

- Los *M* puntos t_i y pesos ω_i se escogen de forma que el resultado de la fórmula sea exacto para polinomios de grado 2M-1 o menor, en lugar de grado *M* como en el caso de Newton-Cotes. Esto hace que el método sea más eficiente que el de Newton-Cotes para un mismo valor de M.
- Como consecuencia, y al contrario que en el método de Newton-Cotes, los puntos t_i no están equiespaciados en $[x_i, x_{i+1}]$. Las posiciones de los puntos vienen definidas por las raíces de los polinomios de Legendre de grado *M*.

De nuevo, no entraremos en los detalles matemáticos o de implementación del método[18] de Legendre-Gauss y nos limitaremos a usar la implementación del mismo que proporciona la librería *Apache Math*.

Esta implementación se basa en la clase *LegendreGaussIntegrator* :

```
LegendreGaussIntegrator(int M, int maxN)
```

Nótese que al instanciar esta clase tenemos que especificar el número de puntos *M* a usar para la integración en cada subintervalo $[x_i, x_{i+1}]$ y el máximo número *N* de subintervalos a usar para descomponer el intervalo total de integración [a,b] (la implementación de *Apache Math* realiza integraciones con valores de *N* crecientes hasta conseguir la precisión requerida). Como en el caso del método de integración de Romberg, una vez instanciada la clase el método *integrate()* realiza el cálculo dada una función y un intervalo *[a,b]*:

```
double integrate(UnivariateRealFunction f, double a, double b)
```

[18] El lector interesado puede consultar por ejemplo la entrada de la Wikipedia http://en.wikipedia.org/wiki/Gaussian_quadrature

Ejercicio 52: escribir un programa que use la implementación del método de Legendre-Gauss en Apache Math para calcular la integral

$$\int_0^1 e^x dx$$

Ajustar los parámetros de instanciación *M* y *maxN* de *LegendreGaussIntegrator* y comprobar cómo varia el resultado.

Sugerencia: usar el método *getIterationCount()* para saber el número de intervalos *N* usados en la ejecución de *integrate()*.

4 Apéndices

4.1 Palabras reservadas

case	**enum******	**instanceof**	**return**	**transient**
catch	**extends**	**int**	**short**	**try**
char	**final**	**interface**	**static**	**void**
class	**finally**	**long**	**strictfp****	**volatile**
const*	**float**	**native**	**super**	**while**

* No se utiliza

** Añadida en 1.2

*** Añadida en 1.4

**** Añadida en 5.0

4.2 Expresiones regulares

Las expresiones regulares permiten crear patrones para buscarlos en cadenas de caracteres. No es el objetivo de este apéndice hacer una explicación completa y concisa del uso de las expresiones regulares, pero si dar algunas ideas de su uso.

Para ver de forma más sencilla el efecto de los expresiones regulares y las coincidencias encontradas, haremos uso del método '`mostrarCoincidencias`' que nos permite ver donde ha encontrado la coincidencia. El método hace uso de las clases '`Pattern`' y '`Matcher`' para localizar la coincidencia.

```
static public void mostrarCoincidencia(String entrada, String patron) {
      Matcher mtr = Pattern.compile(patron).matcher(entrada);
      //System.out.println(Pattern.quote(patron));
      if (mtr.find()) {
            System.out.print(entrada.substring(0, mtr.start()));
            System.out.print(" \"" + mtr.group() + "\" ");
            System.out.println(entrada.substring(mtr.end()));
      } else {
            System.out.println("No hay coincidencias.");
```

```
    }
}
```

A modo de ejemplo, si buscamos el patrón 'Regex' en la cadena 'Vamos a jugar con las Regex.', nos muestra el siguiente resultado:

```
mostrarCoincidencia("Vamos a jugar con las Regex.", "Regex");
```

```
Vamos a jugar con las  "Regex" .
```

Vemos que en este caso nos muestra entre comillas la expresión Regex, tal como esperábamos.

Podemos utilizar otras expresiones como:

```
mostrarCoincidencia("123,456", "3");      //  "1" 23,456
mostrarCoincidencia("123,456", "3,4");    // 12 "3,4" 56
mostrarCoincidencia("123,456", ";");      // No hay coincidencias.
```

Es importante indicar que esta rutina sólo nos muestra la primera coincidencia. Pero no sería difícil modificarla para que nos muestre todas las encontradas.

4.2.1 Patrones

Cada expresión regular contiene un patrón, el cual esta formado por diferentes caracteres. Los caracteres en general se representan a si mismos, excepto en el caso de los símbolos ., |, (,), [,], {, }, +, \, ^, $, *, y ?. Estos últimos sirven para agrupar y operar, dando forma a la expresión. Si se desea utilizar estos símbolos como expresión, se debe anteponer el la barra invertida para que se consideren de forma literal (\., \|, \(, \), …). Es importante tener en cuenta que en Java el carácter de barra invertida es a su vez el carácter de escape en las cadenas de caracteres, por lo que será necesario escaparlo también. En resumen, que si se desea utilizar el '(' como patrón, deberemos hacer uso de las siguiente secuencia:

```
mostrarCoincidencia("(123),456", "\\("); //  "(" 123),456
```

Existe también la opción en Java, si se quieren utilizar de forma literal más caracteres de introducir las secuencias '\Q', que indica el inicio de una cita literal (Quotation), y '\E', que cierra la cita. Cualquier carácter entre ambas secuencias es tomado de forma literal:

```
mostrarCoincidencia("(123),456()", "\\Q()\\E");      // (123),456 "()"
```

Esta solución es específica de Java, por lo que si queremos utilizar esta secuencia en otro lenguaje habremos de comprobar que está disponible.

4.3 Bibliografia y enlaces

- Tutoriales on-line:
  - http://java.sun.com/docs/books/tutorial/
  - http://www.programacion.com/java/tutorial/java_basico/
- Apuntes de java avanzados: Aprenda java como si estuviera en primero
  - http://ocw.uc3m.es/informatica/programacion/manuales/java2-U-Navarra.pdf/view/
- Bonet et al. "Càlcul numéric", Edicions UPC, 1996 (agotado, disponible en formato electrónico en el campus virtual)
- C.S. Horstmann, & G. Cornell, "Core Java 2" Vol. I - Fundamentos (Pearson Prentice Hall 2005)
- R.L. Burden, & J.D. Faires, "Numerical Analysis" (Thomson Brooks/cole, 2004).

- W.H. Press, S.A. Teukolsky, W.T. Vetterling, & B.P. Flannery, “Numerical Recipes: The Art of Scientific Computing, 3rd Edition” (Cambridge University Press, 2007).