Introducción.

Una computadora digital es una máquina que puede resolver problemas ejecutando una secuencia de instrucciones dadas. Se llama programa a una secuencia de instrucciones que describe paso a paso como ejecutar cierta tarea. Los circuitos electrónicos de cada computadora pueden reconocer y ejecutar directamente un conjunto limitado de instrucciones simples. Todos los programas que se desean ejecutar en una computadora deben convertirse previamente en una secuencia de estas instrucciones simples. Estas instrucciones básicas pocas veces rebasan la complejidad de:

- Sumar dos números.
- Comprobar si un número es cero.
- Mover datos de una parte de la memoria a otra.

El conjunto de instrucciones primitivas de una computadora forma el lenguaje con el cual podemos comunicarnos con ella. Dicho lenguaje se llama leguaje de máquina. Normalmente intentan hacer las instrucciones primitivas lo más simple posible, siempre que estén de acuerdo con el uso para el que se ha proyectado la computadora y el rendimiento requerido, a fin de reducir la complejidad y el costo de la electrónica que se necesite. Debido a que la mayoría de los lenguajes de máquina son demasiado elementales, es difícil y tedioso utilizarlos.

Hay dos formas de atacar este problema; ambas incluyen el diseño de un nuevo conjunto de instrucciones, más convenientes para las personas que el conjunto de instrucciones propias de la máquina. Estas instrucciones, en conjunto forman un nuevo lenguaje que llamaremos L2, de manera semejante al que forman las nuevas instrucciones propias de la máquina, que llamaremos L1. Las dos aproximaciones difieren en el modo en que los programas escritos en L2 son ejecutados por la computadora, ya que, después de todo, sólo puede ejecutar programas escritos en su lenguaje de máquina L1.

Un método para ejecutar un programa escrito en L2 consiste en sustituir primero cada instrucción por una secuencia equivalente de instrucciones L1. El resultado es un nuevo programa escrito totalmente con instrucciones en L1. La computadora ejecutará entonces el nuevo programa en L1 y no el anterior en L2. Esta técnica se denomina traducción o compilación.

La otra técnica es escribir un programa en L1 que tome programas escritos en L2 como datos de entrada y los lleve a cabo examinando una instrucción a la vez y ejecutando directamente la secuencia equivalente de instrucciones en L1. Esta técnica, que no requiere la generación previa de un nuevo programa en L1 se llama interpretación y el programa que la lleva a cabo, interprete.

La traducción y la interpretación son bastantes similares. En ambos métodos, las instrucciones L2 se llevan a cabo al ejecutar secuencias equivalentes de instrucciones en L1. La diferencia radica en que, en la traducción todo programa en L2 se convierte en un programa en L1 (código objeto), el programa en L2 se desecha y entonces se ejecuta el

programa generado en L1. En la interpretación se ejecuta cada instrucción en L2 inmediatamente después de examinarla y decodificarla. No se genera ningún programa traducido. Ambos métodos se usan ampliamente.

En vez de pensar en términos de traducción o interpretación, a menudo conviene imaginar la existencia de una computadora hipotética o máquina virtual cuyo lenguaje sea L2.

La invención de toda una serie de lenguajes, cada uno más conveniente que sus predecesores, puede continuar indefinidamente hasta que se consiga una adecuado. Cada lenguaje usa a su predecesor como base, de manera que una computadora que usa esta técnica puede considerarse como una serie de capas o niveles, uno por encima del otro. El lenguaje de alto niveles el más simple, y el de más bajo nivel el más complejo.

Lenguajes, Niveles y Máquinas virtuales.

Existe una relación importante entre un lenguaje y una máquina virtual. Cada máquina tiene algún lenguaje de máquina, que consiste en todas las instrucciones que puede ejecutar. De hecho, una máquina define un lenguaje. En forma similar, un lenguaje define una máquina: la que puede ejecutar todos los programas escritos en ese lenguaje.

Una máquina con n niveles puede verse como n máquinas virtuales diferentes, cada una de las cuales tiene un lenguaje de máquina especial. Solo los programas escritos en L1 pueden ser ejecutados directamente por los circuitos electrónicos sin que se tenga que utilizar la traducción ni la interpretación. Los programas escritos en L2, L3, ..., Ln deben interpretarse por un intérprete en un nivel inferior o traducidos a otro lenguaje correspondiente a un nivel inferior.

Una persona cuyo trabajo sea escribir programas para la máquina virtual de nivel n no necesita conocer los interpretes ni los traductores subyacentes.

Máquinas Multinivel actuales.

La mayoría de las computadoras modernas constan de dos o más niveles. En el nivel 0, el más inferior, es realmente el hardware de la máquina. Sus circuitos ejecutan los programas en el lenguaje de máquina de nivel 1.

En el nivel más bajo, el nivel de lógica digital, los objetos que nos interesan se denominan compuertas. Si bien se construyen a partir de componentes analógicos, tales como transistores, las compuertas pueden modelarse con precisión como dispositivos digitales. Cada compuerta tiene una o más entradas digitales (señales que se representan con 0 o 1) y calcula alguna función simple de estas entradas, como las funciones lógicas AND, OR y EXOR. La tabla de verdad de estas compuertas es.

AND			OR			EXOR		
x	y	f	x	y	f	x	y	f
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

El siguiente nivel por encima es el nivel 1, que conforma el verdadero nivel de lenguaje de máquina. En contraste con el nivel 0, donde no existe el concepto de programa como conjunto de instrucciones a realizar, en el nivel 1 existe ya un programa llamado microprograma, cuya función es interpretar las instrucciones del nivel 2. Llamaremos al nivel 1 el nivel de microprogramación.

Es preciso señalar que algunas computadoras no tienen un nivel de micro computación. En estas máquinas las instrucciones del nivel de máquina convencional son realizadas directamente por los circuitos electrónicos (nivel 0), sin ningún intérprete que intervenga. En consecuencia, el nivel de máquina convencional es el 1 no el 2. De todos modos, continuaremos llamando al nivel de máquina convencional "nivel 2".

El tercer nivel normalmente es un nivel híbrido. La mayoría de las instrucciones de su lenguaje están también en el lenguaje 2. Además existe un nuevo conjunto de instrucciones, una diferente organización de la memoria y la posibilidad de ejecutar dos o más programas en paralelo, entre otras cosas.

Las nuevas posibilidades que se añaden al nivel tres las lleva a cabo un intérprete que actúa en el nivel 2 al que tradicionalmente se llama sistema operativo. Las instrucciones idénticas a las del nivel 2 las lleva directamente a cabo el microprograma en lugar de ejecutarlas el sistema operativo. En otras palabras, algunas de las instrucciones del nivel 3 las interpreta el sistema operativo y otras las interpreta el microprograma, de ahí lo híbrido.

En el nivel 4 se encuentra los leguajes de nivel intermedio como el C y en el nivel 5 los lenguajes de alto nivel como C++ y Java.

Hardware, Software y máquinas multinivel.

El hardware esta constituido por los circuitos electrónicos, junto con la memoria y los dispositivos de entrada y salida.

El Software consta de las instrucciones detalladas que dicen como hacer algo, es decir los programas. Los programas pueden representarse en tarjetas perforadas, cinta magnética, película fotográfica y otros medios.

Cualquier operación realizada por el software puede ser implementada con el hardware y cualquier instrucción ejecutada por hardware puede ser simulada por software.

Cronología histórica de la arquitectura de computadoras.
La generación 0, computadoras mecánicas (1642 – 1945)

La primera generación, bulbos (1945 – 1955)
La segunda generación, transistores (1955 – 1965)
La tercer generación, circuitos integrados (1965 – 1980)
La cuarta generación, computadoras personales e integración a alta escala (1980 – 20??)

Organización de computadoras.

En la figura 1 se muestra la organización de una computadora con un solo bus. La unidad central de procesamiento (CPU) es el cerebro de la computadora. Su función es ejecutar programas almacenados en la memoria central tomando sus instrucciones, examinándolas y luego ejecutándolas una tras otra. La CPU se compone de varias partes. La unidad de control se encarga de traer las instrucciones de la memoria principal y de determinar su tipo. La unidad aritmética y lógica realiza operaciones como la suma o la función booleana AND, necesarias para llevar a cabo las instrucciones.

La CPU también contiene una pequeña memoria de alta velocidad utilizada para almacenar resultados intermedios y cierta información de control. Esta memoria consta de varios registros, cada uno de los cuales tiene cierta función. El registro más importante es el Contador de programa (CP), que indica la próxima instrucción que debe ejecutarse. También es importante el registro de instrucción (RI), que contiene la instrucción que se está ejecutando.

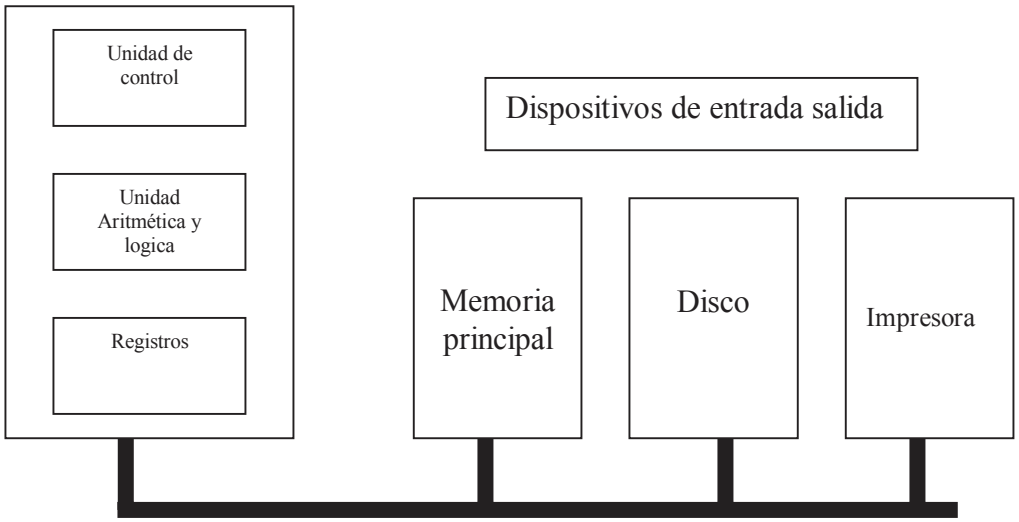


Figura 1

Ejecución de las instrucciones.

La CPU ejecuta instrucciones en una serie de pequeños pasos:

1.- Extrae de la memoria la siguiente instrucción y la lleva al registro de instrucción.
2.- Cambia el contador de programa de modo que señale la siguiente instrucción.
3.- Determina el tipo de instrucción que acaba de extraer.
4.- Verifica si la instrucción requiere datos de la memoria y, si así es, determina donde están situados.
5.- Extrae los datos, si los hay, y los carga en los registros internos del CPU.
6.- Ejecuta la instrucción.
7.- Almacena los datos en el lugar apropiado.
8.- Va al paso 1 para empezar la ejecución de la siguiente instrucción.

Códigos.

Binario.

El código binario es una representación de los números decimales en binario. Es muy utilizado por la computadoras ya que es el lenguaje que entiende. En binario tendremos la siguiente ponderación dependiendo de la posición de los dígitos.

wn....	w_7	w_6	w_5	w_4	w_3	w_2	w_1	w_0	w_{-1}	w_{-2}	w_{-3}	w_{-4}
2^n ...	128	64	32	16	8	4	2	1	½	¼	1/8	1/16

y el número en decimal lo calculamos como

$$D = \sum_{i=0}^{n} b_i 2^i$$

La ecuación anterior nos sirve para hacer la conversión de binario a decimal, y para hacer la conversión de decimal a binario hacemos:

Para hacer

$$\frac{D}{2} = \sum_{i=1}^{n} b_i 2^{i-1}$$

y el residuo será $b_{0,}$ si repetimos esta operación, es decir, dividimos nuevamente entre dos tendremos el bit 1. Repitiendo esta operación hasta que el divisor sea cero, los residuos que encontramos son el número binario que buscamos.

Ejemplo.

Hacer la conversión de los siguientes números de decimal a binario o binario a decimal según sea el caso.

a) $(25)_{10}$ =

```
25 /  2
---|----
12 | 1
 6 | 0
 3 | 0
 1 | 1
 0 | 1
```

$(25)_{10} = (11001)_2$

b) $(23.862)_{10}$

```
23 / 2     0.862 * 2
---|----  -------|---
11 | 1     0.724 | 1
 5 | 1     0.448 | 1
 2 | 1     0.896 | 0
 1 | 0     0.792 | 1
 0 | 1     0.584 | 1
           0.168 | 1
           0.336 | 0
           0.672 | 0
           0.344 | 1
```

$(23.862)_{10} = (10111.110111001 \ldots)_2$

c) $(135)_{10}$ =

```
135 / 2
---|----
67 | 1
33 | 1
16 | 1
 8 | 0
 4 | 0
 2 | 0
 1 | 0
 0 | 1
```

$(135)_{10} = (10000111)_2$

d) $(1100)_2$ =

$1101 * \ 2^n$

```
----|----------
  0 *   1 =   0
  0 *   2 =   0
  1 *   4 =   4
  1 *   8 =   8
----------------
          12
```

$(1100)_2 = (12)_{10}$

e) $(1000101101)_2$ =

$1000101101 * \ 2^n$

```
-----------|----------
        1 *   1 =   1
        0 *   2 =   0
        1 *   4 =   4
        1 *   8 =   8
        0 *  16 =   0
        1 *  32 =  32
        0 *  64 =   0
        0 * 128 =   0
        0 * 256 =   0
        1 * 512 = 512
----------------------
              557
```

$(1000101101)_2 = (557)_{10}$

f) $(101.101)_2$ =

$101.101 * \ 2^n$

```
--------|-------------
     1 * 1/16 =   0.0625
     0 * 1/4  =   0
     1 * 1/2  =   0.5
     1 *   1  =   1
     0 *   2  =   0
     1 *   4  =   4
-------------------------
              5.5625
```

$(101.101)_2 = (5.5625)_{10}$

Hexadecimal.

El código Hexadecimal es un sistema de numeración que utiliza 16 dígitos, con los siguientes valores

dígito	valor binario	valor decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

La ponderación que se da a cada dígito hexadecimal es

... w_7 w_6 w_5 w_4 w_3 w_2 w_1 w_0 w_{-1} w_{-2} w_{-3} w_{-4} ...
... 16^7 16^6 16^5 16^4 16^3 16^2 16^1 16^0 16^{-1} 16^{-2} 16^{-3} 16^{-4} ...

y el número en decimal lo calculamos como

$$D = \sum_{i=0}^{n} h_i 16^i$$

Note que la representación hexadecimal, puede considerarse como una representación binaria simplificada.

Ejemplo.

Realizar la conversión de los siguientes números.

a) $(A95.FB)_{16}$ =

A95.FB * 16^n
----------|----------
B = 11 * 1/(16*16) = 0.04296875

F = 15 * 1/16 = 0.9375
5 = 5 * 1 = 5
9 = 9 * 16 = 144
A = 10 * 16*16 = 2560

= 2709.98046875

$(A95.FB)_{16} = (2709.98046875)_{10}$

b) $(1024)_{10}$ =

1024 / 16
-----|----
64 | 0
4 | 0
0 | 4

$(1024)_{10} = (400)_{16} = (100\ 0000\ 0000)_2 = 2^{10}$

c) $(23.862)_{10}$ =

23 / 16 0.862 * 16
---|---- -------|---
1 | 7 0.792 | 13 = D
0 | 1 0.672 | 12 = C
0.752 | 10 = A
0.032 | 12 = C

$(23.862)_{10}$ $=(17.DCAC)_{16} = (1\ 0111.1101\ 1100\ 1010\ 1100)_2$

ASCII.

El concepto de estandarizar internacionalmente los códigos de computadora, tiene una importancia extrema en las redes de comunicación mundiales de hoy en dia, ya que representa la intercomunicación de computadoras y terminales. Uno de los dos códigos estándar que se emplean en forma internacional, es el alfabeto Internacional Num. 5 de 7 bits. La versión nacional de este código en los estados Unidos se conoce como Código estándar estadounidense para el Intercambio de Información, reconocido comúnmente como ASCII.

HEX	DEC	CHR	CTRL		HEX	DEC	CHR		HEX	DEC	CHR		HEX	DEC	CHR
00	0	NUL	^@		20	32	SP		40	64	@		60	96	`
01	1	SOH	^A		21	33	!		41	65	A		61	97	a
02	2	STX	^B		22	34	"		42	66	B		62	98	b
03	3	ETX	^C		23	35	#		43	67	C		63	99	c

04	4	EOT	^D		24	36	$		44	68	D		64	100	d
05	5	ENQ	^E		25	37	%		45	69	E		65	101	e
06	6	ACK	^F		26	38	&		46	70	F		66	102	f
07	7	BEL	^G		27	39	'		47	71	G		67	103	g
08	8	BS	^H		28	40	(		48	72	H		68	104	h
09	9	HT	^I		29	41	)		49	73	I		69	105	i
0A	10	LF	^J		2A	42	*		4A	74	J		6A	106	j
0B	11	VT	^K		2B	43	+		4B	75	K		6B	107	k
0C	12	FF	^L		2C	44	,		4C	76	L		6C	108	l
0D	13	CR	^M		2D	45	-		4D	77	M		6D	109	m
0E	14	SO	^N		2E	46	.		4E	78	N		6E	100	n
0F	15	SI	^O		2F	47	/		4F	79	O		6F	111	o
10	16	DLE	^P		30	48	0		50	80	P		70	112	p
11	17	DC1	^Q		31	49	1		51	81	Q		71	113	q
12	18	DC2	^R		32	50	2		52	82	R		72	114	r
13	19	DC3	^S		33	51	3		53	83	S		73	115	s
14	20	DC4	^T		34	52	4		54	84	T		74	116	t
15	21	NAK	^U		35	53	5		55	85	U		75	117	u
16	22	SYN	^V		36	54	6		56	86	V		76	118	v
17	23	ETB	^W		37	55	7		57	87	W		77	119	w
18	24	CAN	^X		38	56	8		58	88	X		78	120	x
19	25	EM	^Y		39	57	9		59	89	Y		79	121	y
1A	26	SUB	^Z		3A	58	:		5A	90	Z		7A	122	z
1B	27	ESC			3B	59	;		5B	91	[		7B	123	{
1C	28	FS			3C	60	<		5C	92	\		7C	124	|
1D	29	GS			3D	61	=		5D	93	]		7D	125	}
1E	30	RS			3E	62	>		5E	94	^		7E	126	~
1F	31	US			3F	63	?		5F	95	_		7F	127	DEL

El estándar IEEE para aritmética de punto flotante.

El IEEE (Institute of Electrical and Electronics Engineers) ha creado un código estándar para representar números flotantes. Este estándar especifica como deben ser representados números en precisión simple de 32 bits y de doble precisión en 64 bits.

Los números en precisión simple se representan :

La representación estándar IEEE para números flotantes requiere de una palabra de 32, la cual debe ser numerada del 0 a 31, izquierda a derecha. El primer bit S, es un bit de signo, los siguientes 8 bits son el exponente del número 'E', y los 23 bits restantes son la mantisa 'F':

```
 S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
 0 1      8 9                   31
```

El valor V representado por la palabra debe determinarse como sigue:

- Si E = 255 y F es diferente de cero, entonces V = NaN ("Not a number", no es un número)
- Si E = 255 y F es cero y S es 1, entonces V = -Infinito
- Si E = 255 y F es cero y S es 0, entonces V = Infinito
- Si 0<E<255 entonces V = (-1)**S * 2 ** (E-127) * (1.F) donde "1.F" representa el número codificado en binario como numero fraccionario F con la perdida del 1 y del punto decimal.
- Si E = 0 y F es diferente de cero, entonces V = (-1)**S * 2 ** (-126) * (0.F)
- Si E = 0 y F es cero y S es 1, entonces V = -0
- Si E = 0 y F es cero y S es 0, entonces V = 0

En particular,

```
 0 00000000 00000000000000000000000 = 0
 1 00000000 00000000000000000000000 = -0

 0 11111111 00000000000000000000000 = Infinity
 1 11111111 00000000000000000000000 = -Infinity

 0 11111111 00000100000000000000000 = NaN
 1 11111111 00100010001001010101010 = NaN

 0 10000000 00000000000000000000000 = +1 * 2**(128-127) * 1.0 = 2
 0 10000001 10100000000000000000000 = +1 * 2**(129-127) * 1.101 = 6.5
 1 10000001 10100000000000000000000 = -1 * 2**(129-127) * 1.101 = -6.5

 0 00000001 00000000000000000000000 = +1 * 2**(1-127) * 1.0 = 2**(-126)
 0 00000000 10000000000000000000000 = +1 * 2**(-126) * 0.1 = 2**(-127)
 0 00000000 00000000000000000000001 = +1 * 2**(-126) *
                                      0.00000000000000000000001 =
                                      2**(-149)  (valor mas pequeño posible)
```

Regresar.

Introducción a Java.

¿Qué es Java?

Java es un lenguaje de programación de ordenadores, diseñado como una mejora de C++, y desarrollado por **Sun Microsystems**. *(**Nota**: Para más información sobre Sun, puede consultar su página Web: www.sun.com)*

Hay varias hipótesis sobre su **origen**, aunque la más difundida dice que se creó para ser utilizado en la programación de aparatos electrodomésticos (desde microondas hasta televisores interactivos). De ahí evolucionó (parece ser que porque el proyecto inicial no acabó de funcionar) hasta convertirse en un lenguaje orientado a Internet.

Pero su campo de **aplicación** no es exclusivamente Internet: una de las grandes ventajas de Java es que se procura que sea totalmente independiente del hardware: existe una "máquina virtual Java" para varios tipos de ordenadores. Un programa en Java podrá funcionar en cualquier ordenador para el que exista dicha "máquina virtual Java" (hoy en día es el caso de los ordenadores equipados con los sistemas operativos Windows, Linux, Solaris y algún otro). A cambio, la existencia de ese paso intermedio hace que los programas Java no sean tan rápidos como puede ser un programa realizado en C, C++ o Pascal y optimizado para una cierta máquina en concreto.

Tiene varias **características** que pueden sonar interesantes a quien ya es programador, y que ya irá conociendo poco a poco quien no lo sea: La sintaxis del lenguaje es muy parecida a la de C++ y, al igual que éste último, es un lenguaje orientado a objetos. Java soporta el manejo de threads, excepciones, está preparado para la realización de aplicaciones cliente/servidor, es independiente de la máquina, es más fiable y seguro que C++ (no existen los punteros) y cada vez incorpora más facilidades para la creación y manipulación de gráficos, para el acceso a bases de datos, etc. Buena parte de estas posibilidades las iremos tratando más adelante, cuando hayamos visto las más básicas.

¿Qué hace falta para usar un programa creado en Java?

Normalmente, la forma en que más frecuentemente encontramos programas creados en lenguaje Java será dentro de páginas Web.

Estas aplicaciones Java incluidas en una página Web reciben el nombre de **"Applets"**, y para utilizarlos normalmente no hará falta nada especial. Si nuestro Navegador Web reconoce el lenguaje Java (es lo habitual hoy en día, y es el caso de Internet Explorer de Microsoft y de Netscape Communicator, los más extendidos), estos Applets funcionarán perfectamente sin necesidad de ningún software adicional.

Las aplicaciones que deban funcionar "por sí solas" necesitarán que en el ordenador de destino exista algún intérprete de Java. Esto ya no es tan frecuente, pero tampoco es problemático: basta con instalar en el ordenador de destino el "Java Runtime Enviroment" (**JRE**), que se puede descargar desde la página web de Sun.

¿Qué hace falta para crear un programa en Java?

Existen diversas herramientas que nos permitirán crear programas en Java. La más habitual (y la más recomendable) es la propia que suministra Sun, y que se conoce como **JDK** (Java Development Kit). Es de libre distribución y se puede conseguir en la propia página Web de Sun.

El inconveniente del JDK es que puede resultar incómodo de manejar para quien esté acostumbrado a otros entornos integrados, como los que incorporan Delphi o Visual Basic.

En ese caso, es fácil encontrar por un precio casi simbólico editores que hagan más fácil nuestro trabajo, o incluso sistemas de desarrollo completos (como el JBuilder de Borland), si bien estos sistemas de desarrollo suelen tener un precio más elevado.

__Nota__: Para más información sobre Borland (ahora Inprise),
puede consultar su página Web:
www.borland.com www.inprise.com

Los programas que crearemos estarán probados con la versión 1.2.2 del JDK (que soporta la versión del lenguaje conocida como "Java 2"), salvo que se indique lo contrario.

Instalación del JDK bajo Windows.

Vamos a ver el caso más habitual: que trabajemos con Windows 95 (o superior) y que no dispongamos de mucho dinero para gastar. Entonces nos interesará conseguir gratuitamente el JDK e instalarlo en nuestro ordenador.

¿Dónde se puede conseguir el JDK?

El sitio más fiable es la propia página Web de Sun:

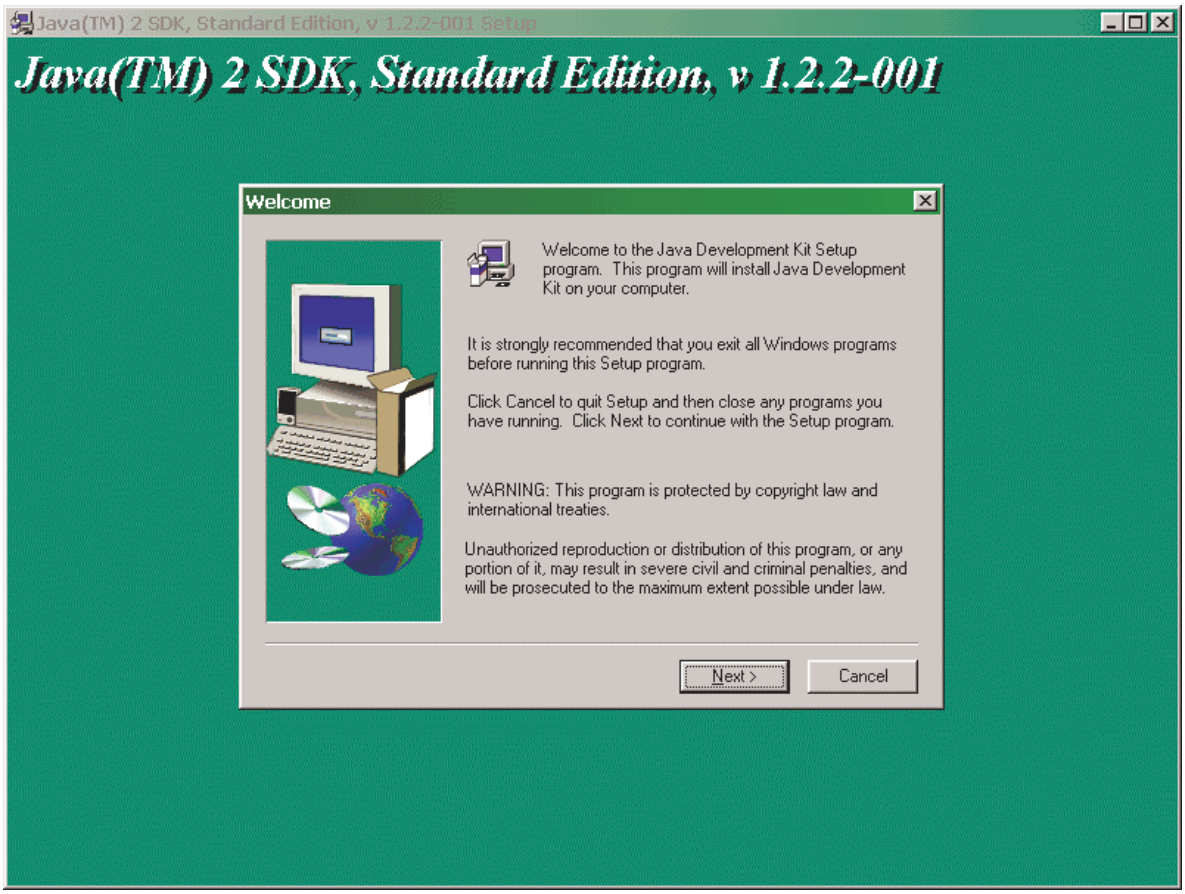
```
http://java.sun.com
```

Allí encontraremos enlaces para **descargar** (download) la última versión disponible. Bastará con que leamos y aceptemos las condiciones de la licencia, y entonces empezaremos a recibir un único fichero de gran tamaño (20 Mb para la versión 1.2.2 para Windows; también existe la opción de descargar varios ficheros de aproximadamente 1.4Mb de tamaño)

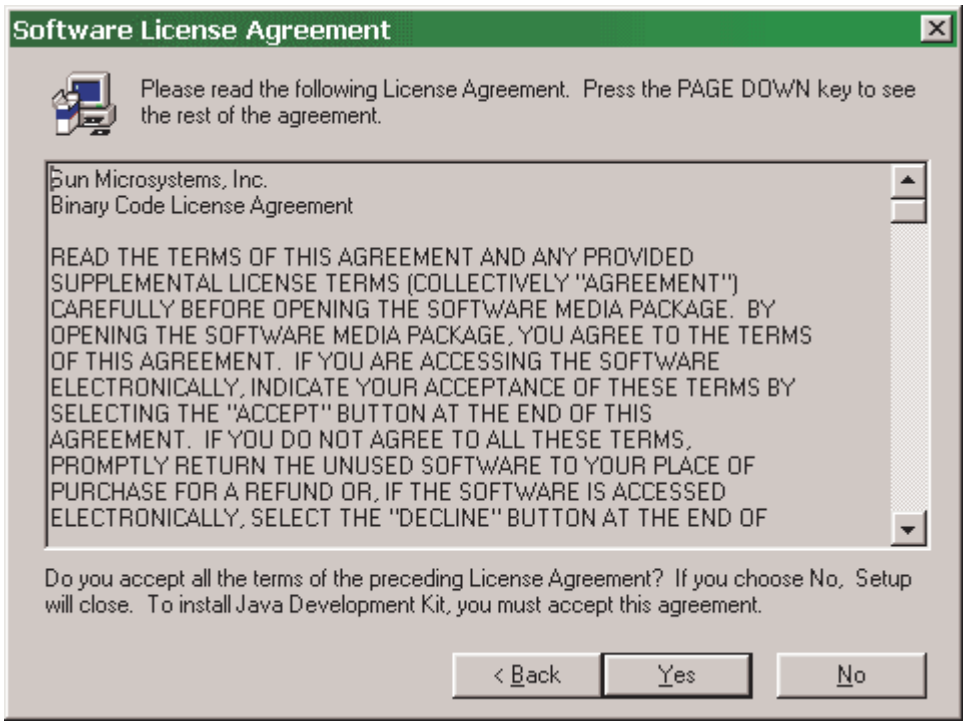
¿Cómo instalo el JDK?

Bastará hacer **doble clic** sobre el fichero recibido. Por ejemplo, para la versión 1.2.2 del JDK para Windows (95 o superiores y NT o superiores), este fichero tenía el nombre `jdk1_2_2-001-win.exe`

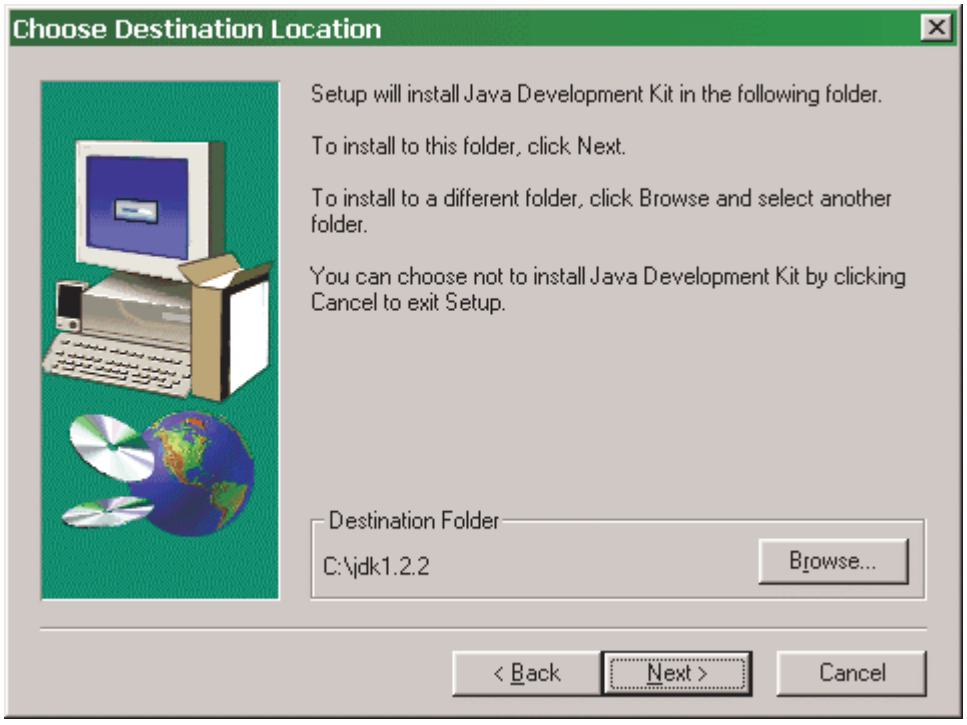
Al hacer doble clic sobre este fichero, aparecerán varios mensajes de que está comenzando a "descomprimir" datos temporales, y finalmente una pantalla de **bienvenida**:



Si pulsamos el botón “Next” para proseguir con la instalación, se nos mostrará la **licencia de uso**.

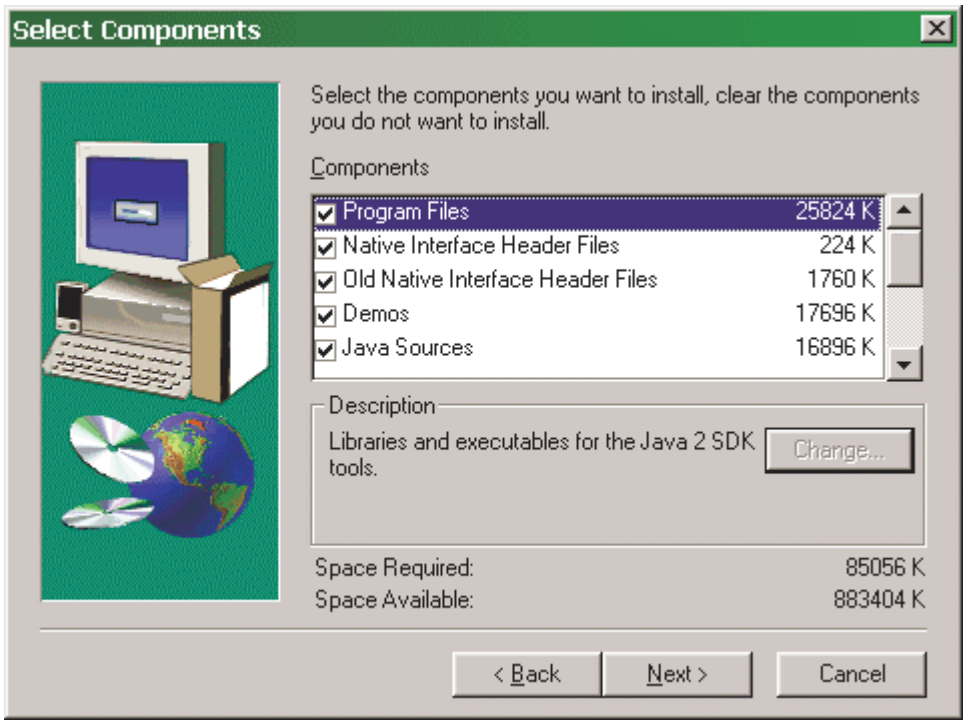


Si aceptamos estas condiciones (pulsando “Yes”), aparecerá otra ventana, en la que se nos pregunta en qué carpeta (o directorio) queremos instalar el JDK:



Podríamos cambiar el directorio de destino paseando por nuestro disco duro con el botón “Browse” o bien aceptar el directorio que se nos propone y pulsar el botón “Next” para continuar.

A continuación se nos pide que escojamos las **partes** del JDK que deseamos instalar. Ante la duda, es preferible dejar todas las opciones marcadas:



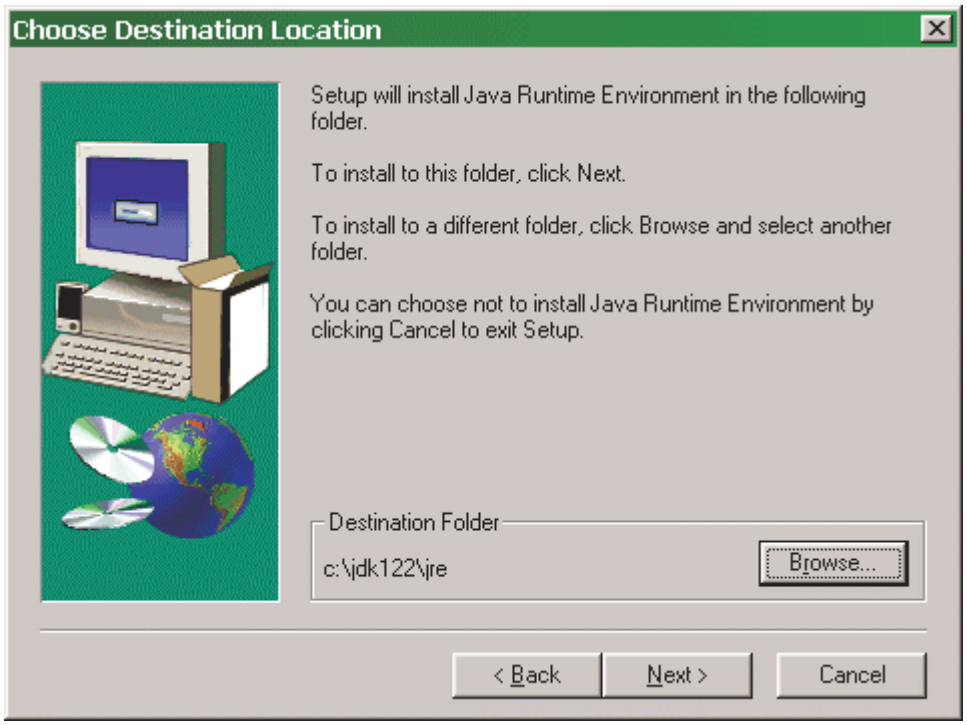
Después pasamos a la instalación del **Java Runtime Environment** (JRE), que es la "máquina virtual Java" que permitirá que podamos utilizar en un cierto ordenador los programas creados en Java.

Si se tratase de otro ordenador distinto del nuestro, en el que quisiéramos instalar nuestros programas creados en Java, pero no necesitásemos desarrollar programas desde dicho ordenador, no necesitaríamos instalar el JDK completo, sino sólo el JRE, de modo que podríamos descargar sólo éste desde el sitio Web de Sun.

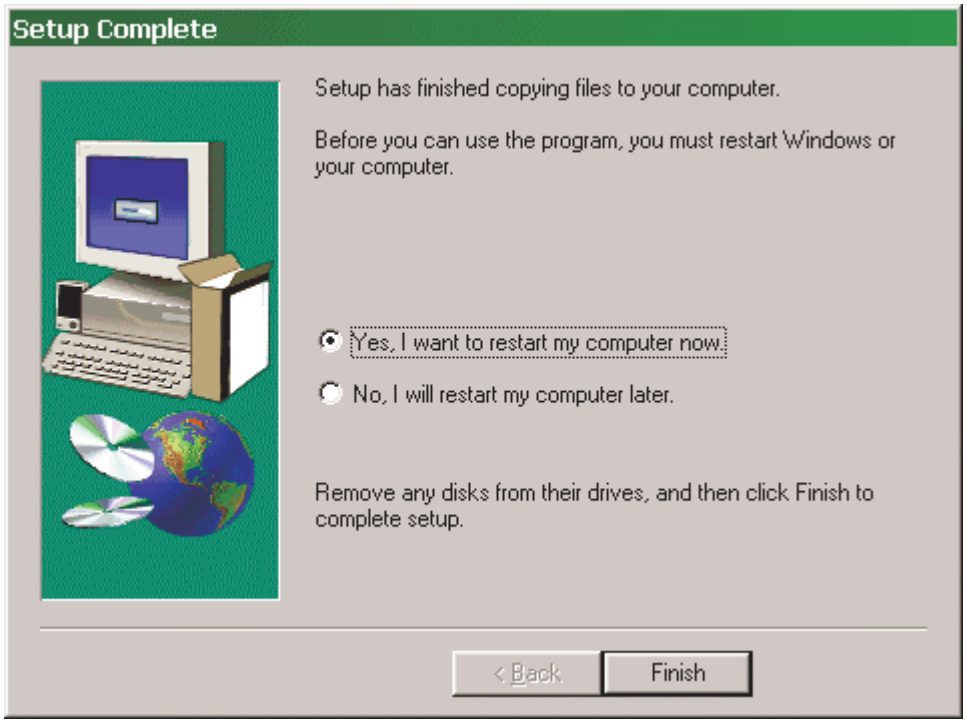
En cualquiera de ambos casos, deberemos comenzar por aceptar las condiciones de **licencia**.

Después, al igual que ocurría con el JDK, se nos preguntará en qué directorio (**carpeta**) deseamos hacer la instalación.

Se nos propone un directorio dentro de “Archivos de programa”. En mi caso, al instalarlo en mi propio ordenador, prefiero cambiarlo para que quede dentro del directorio en el que ha quedado el JDK:



Al final de la instalación se nos recomienda **reiniciar** nuestro ordenador, y el proceso queda terminado:



¿Hay que hacer algo más para que funcione el JDK?

Sí. Ya está la información dentro de nuestro ordenador, pero tenemos que acceder a ella. Hay dos opciones:

- La opción "mala" es usar el JDK **tal cual** ha quedado instalado. Esto supone que los fuentes (nuestros programas Java "originales") deberán estar en el directorio "BIN" de nuestra instalación "JDK". Es decir, deberíamos entrar al directorio C:\JDK122\BIN (o el que corresponda en nuestro caso, si hemos empleado otro distinto), teclear allí nuestros fuentes, y ponerlos en funcionamiento desde allí.

***Nota sobre Windows**: Se puede modificar el Autoexec.bat de varias formas. Una puede ser pulsar Inicio/Ejecutar y teclear: notepad c:\autoexec.bat*

- La opción "buena" es que nuestros fuentes puedan estar en **cualquier sitio**. Para eso, debemos modificar el "AUTOEXEC.BAT" de nuestro ordenador, añadiendo al final del mismo la línea

```
path=%path%;c:\jdk122\bin
```

Creando una primera aplicación.

Escribiendo "Hola Mundo"

Comenzaremos por crear una aplicación (programa) en **modo texto**. Esta aplicación hará lo que se suele usar como primer ejemplo en todos los lenguaje de programación: escribir un texto en la pantalla.

Quien ya conozca otros lenguajes, verá que conseguirlo en Java parece más complicado. Por ejemplo, en Basic bastaría con escribir PRINT "HOLA MUNDO!". Pero esta mayor complejidad inicial es debida al "cambio de mentalidad" que tendremos que hacer, y dará lugar a otras ventajas más adelante.

A quien no conozca ningún lenguaje de programación todo le sonará extraño, pero dentro de muy poco (apenas veamos que nuestro programa funciona) volveremos atrás para ver paso a paso qué hace cada una de las líneas que habremos tecleado.

Vamos allá...

- Lo primero que haremos será **crear una carpeta** en la que guardaremos nuestros fuentes (o al menos éste).

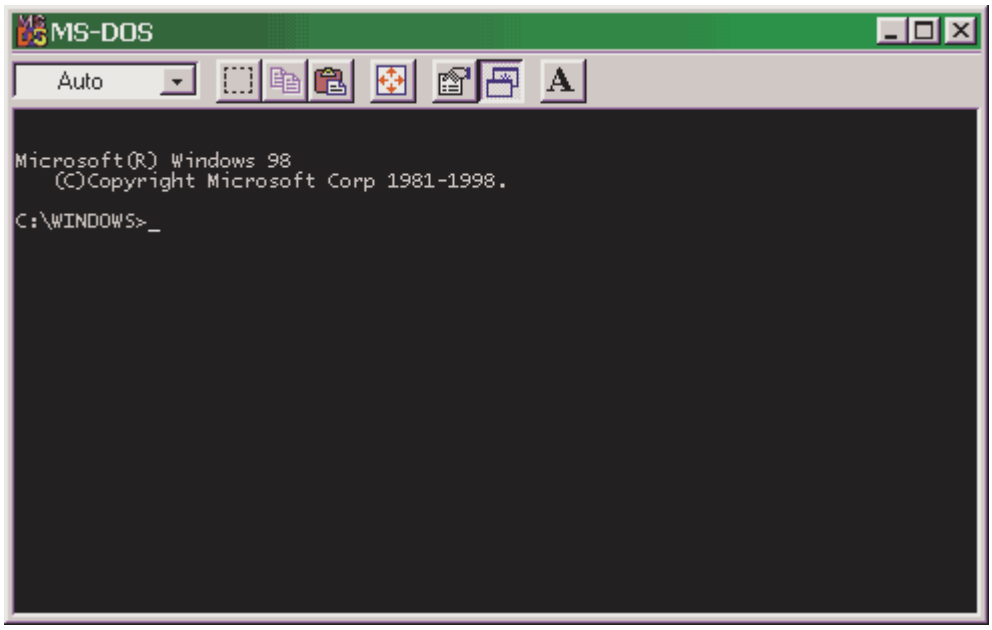
***Nota sobre Windows**: Se puede crear una carpeta*

desde el Explorador de Windows: entramos a la carpeta dentro de la que queremos situarla, pulsamos el botón derecho del ratón y escogemos la opción Nuevo / Carpeta

- Una vez que tenemos un "sitio" donde guardar nuestros programas, tenemos que **teclearlos**. Lo haremos con cualquier editor de texto. En un principio, puede servirnos el Bloc de Notas que incorpora Windows. *(**Nota**: No deberemos emplear Word ni ningún otro procesador de textos que permita distintos tipos de letra, cursivas, negritas y otros cambios de apariencia, salvo que seamos usuarios avanzados y sepamos cómo guardar como "Sólo texto").*

- En nuestro caso, no usaremos el Bloc de Notas, porque tiene una característica peligrosa: suele añadir las letras "txt" al final de los archivos, y eso haría que Java no los reconociera como programas. Haremos todo desde el "**modo MsDos**" de Windows 95 y superiores.

- Para entrar al "modo MsDos" de Windows, entramos al menú "Inicio", submenú "Programas", hacemos clic en la opción "MS-DOS". Aparecerá la ventana negra típica de MsDos y del "interfaz de comandos" de Windows NT:



***Nota sobre Windows**: Puede que la ventana negra anterior realmente esté ocupando toda la pantalla; podemos cambiar entre la visualización en pantalla completa y en ventana pulsando simultáneamente Alt y Return (o Enter, o la tecla de avance de línea)*

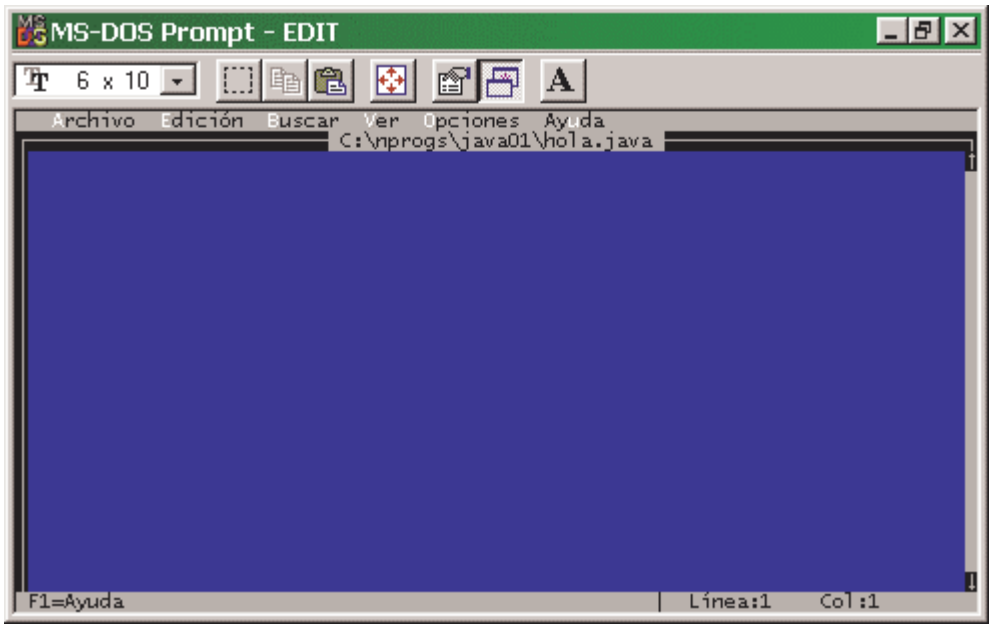
- Ahora deberemos **desplazarnos** hasta la carpeta que hemos creado, o bien crearla en este momento. Por ejemplo, en mi caso, he creado una carpeta JAVA01 dentro

de la carpeta NPROGS, que es parte de mi disco C, de modo que teclearía CD \NPROGS\JAVA01 para entrar a esa carpeta.

- Una vez dentro de esta carpeta, podríamos crear un fichero llamado "hola.java". que será nuestro primer fuente en Java. Lo haremos empleando el Editor de texto de MsDos, llamado **EDIT**. De modo que teclearemos

```
edit hola.java
```

Debería aparecer la pantalla vacía del editor de MsDos:



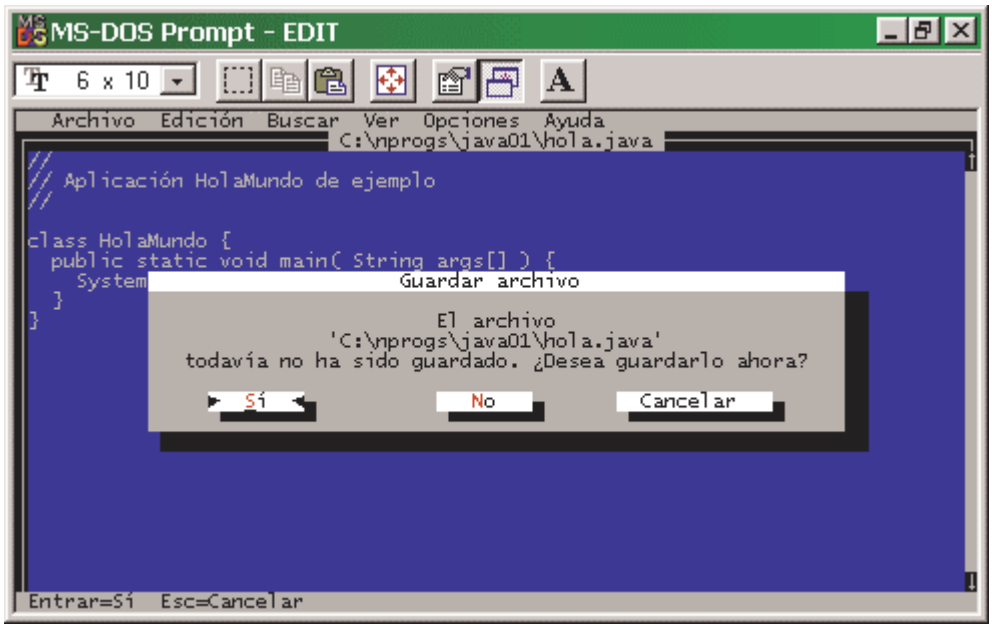
***Nota**: El uso de mayúsculas y minúsculas es irrelevante para MsDos y para Wndows, pero no para Java (ni para otros sistemas, como Unix). Por eso, deberemos respetar las mayúsculas y minúsculas tal y como aparezcan en los ejemplos.*

Ya dentro de este editor, teclearemos lo siguiente:

```
//
// Aplicación HolaMundo de ejemplo
//

class HolaMundo {
    public static void main( String args[] ) {
        System.out.println( "Hola Mundo!" );
```

```
        }
    }
```

- Una vez que hemos tecleado todo esto, salimos del Editor, entrando al menú Archivo (con el ratón o pulsando Alt+A) y escogiendo la opción Salir. El editor se dará cuenta de que hemos escrito algo y nos preguntará si queremos guardar los cambios (deberemos decirle que Sí):

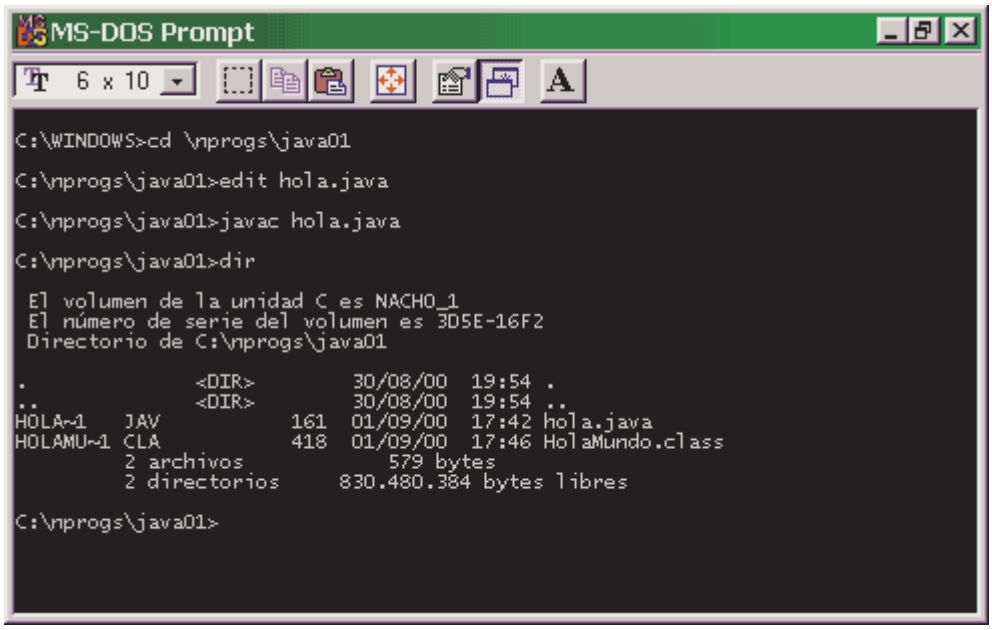


- Deberíamos volver la ventana negra de MsDos, pero ahora ya hemos avanzado un poco: tenemos nuestro programa escrito.

- El siguiente paso es **"compilarlo"**, es decir, convertirlo en algo que pueda entender cualquier ordenador para el que exista la "máquina virtual Java". Para conseguirlo, escribimos:

```
javac hola.java
```

- Si no aparece ningún **mensaje de error**, quiere decir que todo ha ido bien. Para comprobarlo, podemos teclear la orden "dir" de MsDos (sin las comillas), que nos muestra el contenido de ese directorio en el que estamos trabajando. Debería aparecer algo parecido a:

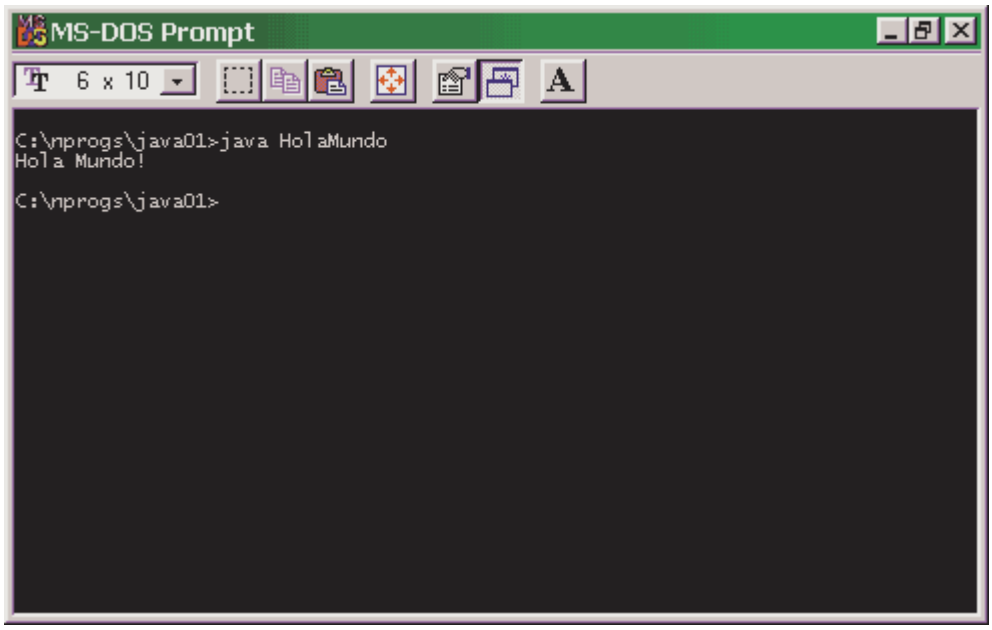


- Vemos que hay algo que "no hemos puesto nosotros": hay un fichero llamado "**HolaMundo.class**". En efecto, es la "clase HolaMundo" que habíamos definido en nuestro fuente. Un poco más adelante hablaremos de clases y veremos detalles sobre qué hemos escrito en nuestro fuente y por qué, pero de momento ya notamos que "algo ha ocurrido" a pesar de que nuestro ordenador no hubiera respondido con ningún mensaje.

- Si hubiésemos **tecleado algo mal**, sí que obtendríamos algún mensaje de error como respuesta. Por ejemplo, si hubiéramos escrito "clas" en vez de "class", en la línea 5 de nuestro programa, obtendríamos como respuesta:

```
hola.java:5: Class or interface declaration expected.
clas HolaMundo {
^
1 error
```

- Pues vamos a ver **funcionar** nuestro programa: tecleamos "java HolaMundo" (sin comillas, como siempre) para que nuestro ordenador siga los pasos que le hemos detallado en "HolaMundo.class", y debería aparecer escrito en pantalla "Hola Mundo!":

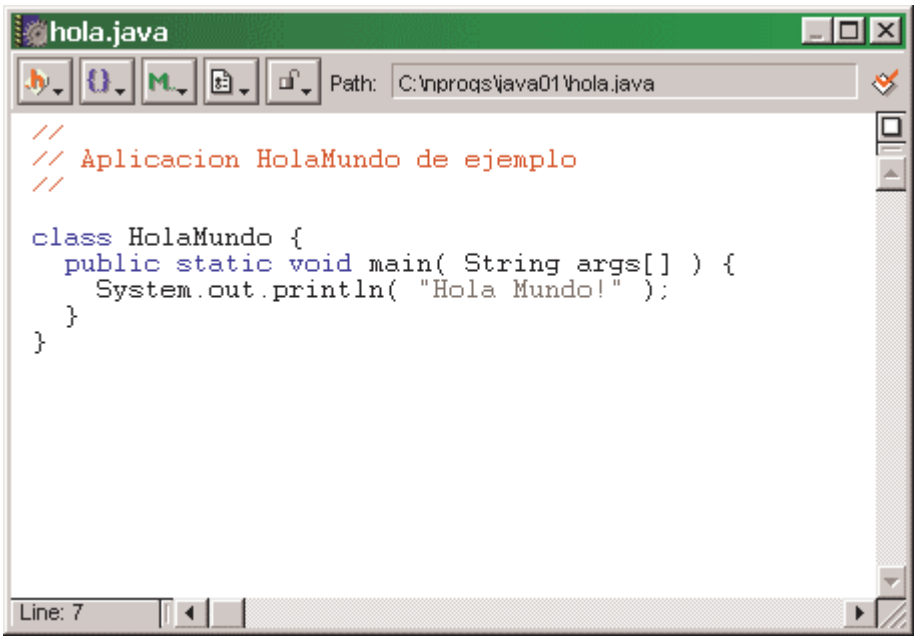


Este es **nuestro primer programa**. Es poco útil, pero al menos funciona. Ahora veremos qué hemos hecho en este programa, y comentaremos algunas de las órdenes que permite Java, de modo que podamos pronto crear algo "más práctico".

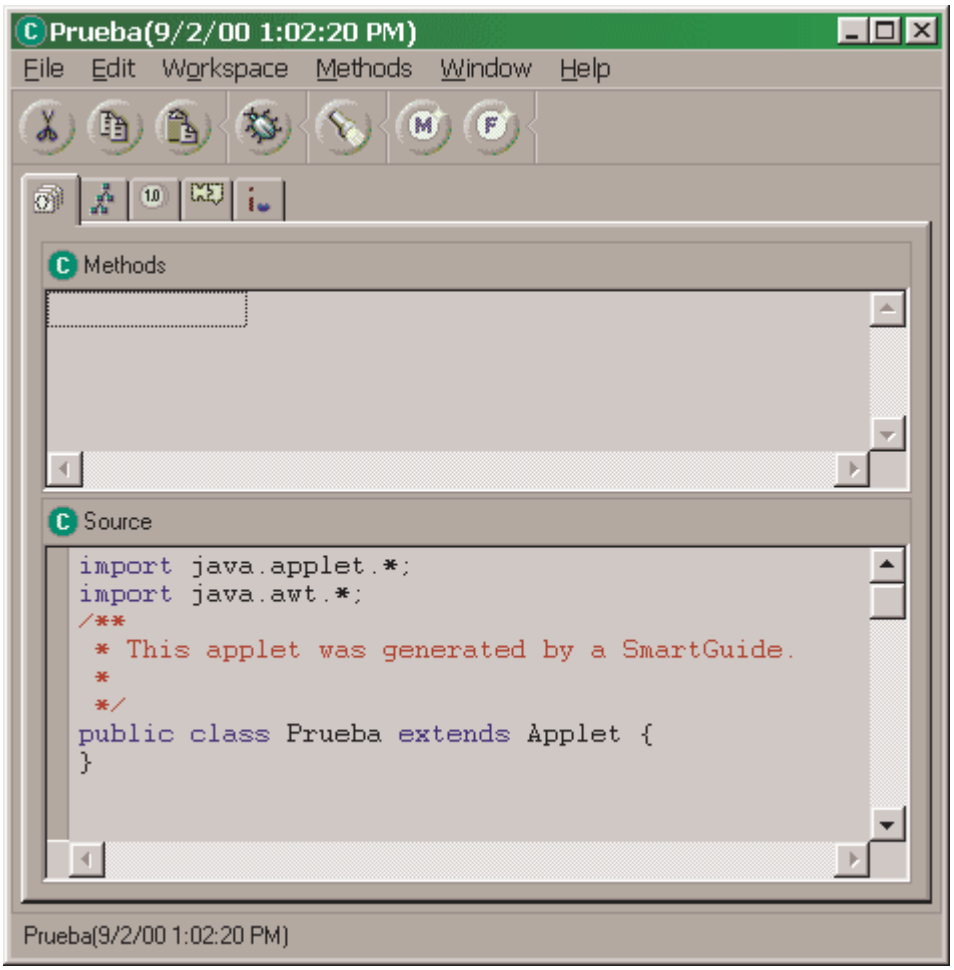
¿Siempre es tan incómodo programar en Java?

No necesariamente:

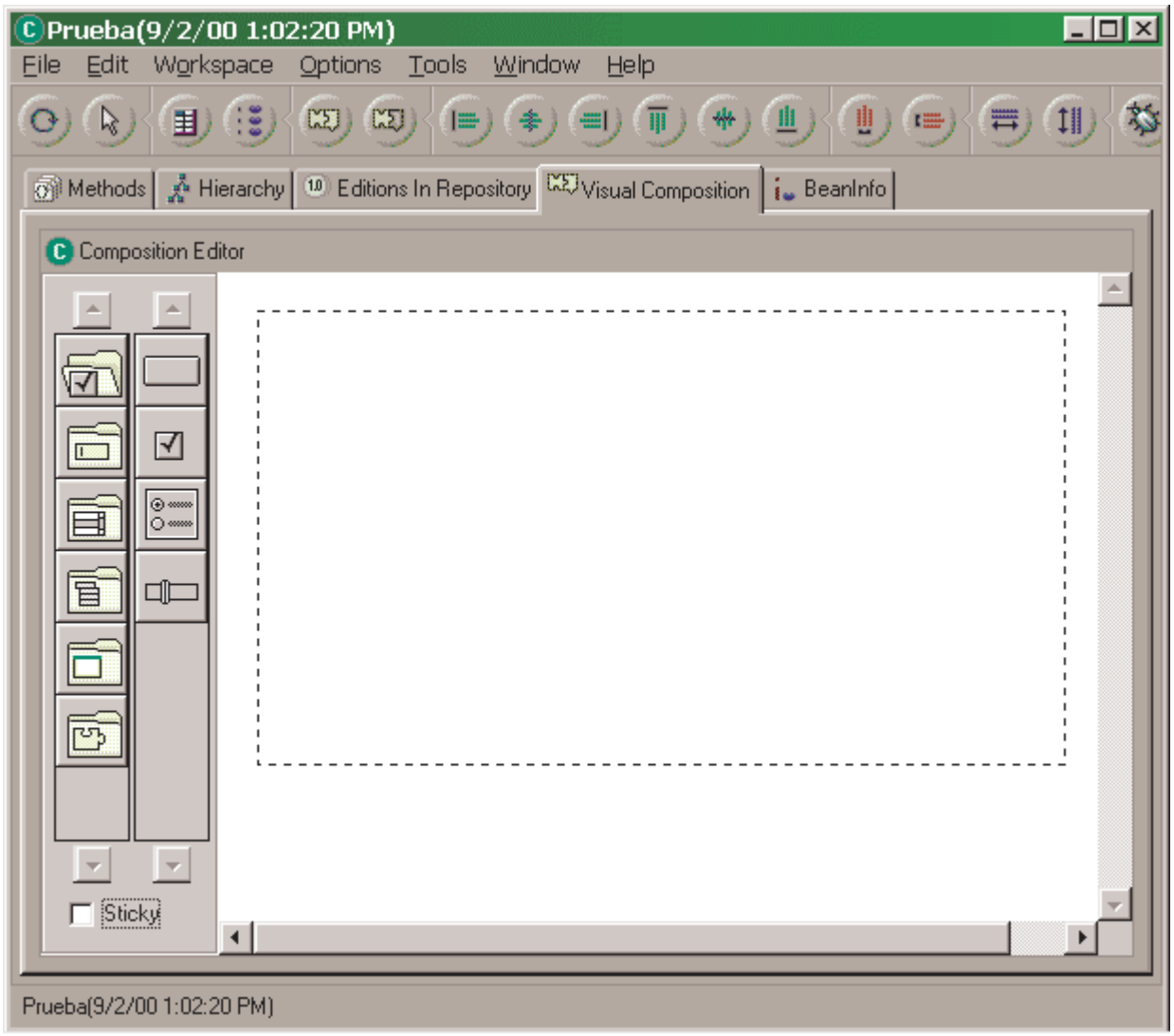
Hay otros editores más cómodos, que realzan en colores la sintaxis de lo que hemos escrito, para ayudarnos a detectar errores (*por ejemplo, la siguiente imagen muestra el editor de "CodeWarrior"*).



Incluso hay **entornos integrados** que nos ayudan a teclear nuestro programa y probarlo desde un mismo sitio, sin tener que conocer órdenes de MsDos (*es el caso de Ibm Visual Age for Java, al que corresponden las dos próximas pantallas, y también es el caso de Borland JBuilder*).



Muchos de estos entornos integrados nos ayudarán a diseñar de forma **"visual"** las pantallas que conformarán la parte visible de nuestros programas, de un modo similar al que emplear otras herramientas como Delphi (para lenguaje Pascal), Visual Basic (para Basic) o C++ Builder (para C++).



Pero, como ya habíamos anticipado, no comenzaremos por lo más fácil, sino por lo más **"barato"**, que no suponga ningún desembolso de dinero adicional. Por ello, de momento continuaremos tecleando los programas con el Editor de MsDos, y compilándolos y probándolos con las herramientas del JDK. Más adelante veremos el manejo básico de alguno de los entornos integrados que se distribuyen gratuitamente.

Entendiendo esta primera aplicación.

Vayamos línea por línea para ver qué hemos tecleado y por qué:

El comienzo del programa: un comentario.

Las tres primeras líneas son:

```
//
// Aplicación HolaMundo de ejemplo
//
```

Estas líneas, que comienzan con una doble barra inclinada (//) son **comentarios**. Nos sirven a nosotros de aclaración, pero nuestro ordenador las ignora, como si no hubiésemos escrito nada.

Continuamos definiendo una clase.

Después aparece un bloque de varias órdenes; vamos a eliminar las líneas del centro y conservar solamente las que nos interesarán en primer lugar:

```
class HolaMundo {
    [... aquí faltan más cosas ...]
}
```

Esto es la definición de la "clase" llamada "HolaMundo", que nos indica que Java es un lenguaje **orientado a objetos**...

Nociones básicas de programación orientada a objetos.

En Java no debemos pensar simplemente en líneas de programa, sino que antes debemos plantearnos el diseño de nuestro programa como algo mucho más general: deberemos haber descompuesto el problema en una serie de **objetos que se relacionan entre sí**.

Para entender esta idea, vamos llevarlo a una **situación cotidiana**:

Imaginemos una persona (a la que llamaremos Eva), que todas las mañanas va a trabajar con su coche (por ejemplo, un Seat Ibiza). En algo tan habitual como esto, aparecen muchas de los conceptos que nosotros emplearemos. Tenemos dos "**objetos**" (Eva y su Seat), pero no son objetos "aislados": Eva es una persona, y su Seat es un coche, es decir cada uno pertenece a un grupo de objetos que comparten una serie de características comunes (lo que llamaremos una "**clase**"). Existe una interacción entre ellos: Eva puede avisar a su Seat de que desea circular más deprisa, pisando el pedal del acelerador (es lo que llamaremos "**mandar un mensaje**"), pero también el coche puede avisar a Eva, por ejemplo encendiendo una luz cuando le queda poco combustible (también le estaría enviando un mensaje).

Estos objetos interaccionando se pueden expresar gráficamente, en forma de diagrama (lo que nos ayuda a diseñar nuestros "sistemas"), y se pueden simular con una cierta facilidad mediante un ordenador si se emplean lenguajes o herramientas orientados a objetos. De hecho, existen utilidades que nos permiten "dibujar" en nuestro ordenador los diagramas que muestran los objetos y las relaciones existentes entre ellos, y son capaces de crear el "esqueleto" de nuestro programa, en el que nosotros deberemos ir rellenando "los huecos" con los detalles necesarios.

En nuestro caso, apenas tenemos un par de objetos: nuestra aplicación y la pantalla de nuestro ordenador. Y también se "mandarán mensajes" entre ellos: nuestra aplicación pedirá a la pantalla que muestre un cierto mensaje.

Pero en la Programación Orientada a Objetos existen algunos **otros conceptos** importantes, y que emplearemos más adelante. Vamos a comentar ya alguno de ellos:

- Hablaremos de **atributos** (o variables miembro), que son las "características" de un determinado objeto (o de toda su clase), como el peso de una persona, su estatura o su color de ojos.

- También hablaremos de **métodos** (o funciones miembro), que son las cosas que un objeto (o su clase) es capaz de hacer. Por ejemplo, un coche puede acelerar, frenar, cambiar de dirección, etc.

- Se habla mucho de **herencia**. La herencia es el hecho de que no siempre haga falta definir las cosas "desde cero", sino que podremos apoyarnos en lo que ya conocemos. Por ejemplo, el día que Eva necesita conducir otro coche distinto del suyo, no es una tarea difícil, porque ambos son coches y apenas existirán unas pocas diferencias entre ambos, de manera que tendrá pocas cosas nuevas que aprender. De igual modo, nosotros podremos definir unos objetos a partir de otros ya existentes (porque ya estén creados en Java, o porque los hayamos creado nosotros mismos -u otra persona-), lo que nos puede ahorrar mucho trabajo.

- También puede aparecer la palabra **"polimorfismo"**, que se refiere a que una misma palabra se puede aplicar en sitios relativamente distintos. Por ejemplo, igual que hablamos de "conducir" un coche y de "conducir" una motocicleta, aunque ambas formas de conducir se parezcan poco, también en nuestros programas podremos hablar de "dibujar" figuras geométricas muy distintas, cada una de las cuales requerirá una serie de pasos específicos.

- Otra de las cararacterísticas de la programación orientada a objetos es la **ocultación de los detalles**. Por ejemplo: Eva no necesita saber casi nada de mecánica para conducir su Seat Ibiza, y si el día de mañana la casa Seat decide cambiar los motores en los nuevos Seat Ibiza, Eva podría conducir otro Seat nuevo sin necesidad de aprender nada nuevo. De igual modo, nuestros programas deberían ser lo suficientemente flexibles como para que una parte del programa no necesite saber detalles internos de otra, y podamos permitirnos mejorar internamente una parte del programa sin que ello suponga modificar el resto.

Todos estos conceptos los iremos aplicando a medida que vayamos avanzando.

¿Qué hace nuestra clase?

Como ya hemos comentado, hará poco más que pedir a la pantalla que muestre un cierto mensaje. Los detalles concretos de lo que debe hacer nuestra clase los indicamos entre **llaves** ({ y }), así:

```
class HolaMundo {
    [... aquí faltan más cosas ...]
}
```

Estas llaves serán frecuentes en Java, porque las usaremos para delimitar cualquier **bloque** de programa (y normalmente habrá bastantes...)

Java es un lenguaje de **formato libre**, de modo que podemos dejar más o menos espacios en blanco entre los distintas palabras y símbolos, así que la primera línea de la definición de nuestra clase se podría haber escrito más espaciada

```
class    HolaMundo     {
```

o bien así (formato que prefieren algunos autores, para que las llaves de principio queden justo encima de las de final)

```
class HolaMundo
{
```

o incluso cualquier otra "menos legible":

```
  class
HolaMundo   {
```

Nuestra clase de objetos "Hola Mundo" sólo hace una cosa: englobar al cuerpo del programa, de modo que en su interior sólo existe una función, llamada "**main**", que representa al cuerpo de la aplicación.

```
public static void main( String args[] ) {
    [... aquí faltan más cosas ...]
}
```

***Nota para C y C++**: La función "main" es, al igual que en C y C++, la que*

indicará el cuerpo de un programa, pero en Java el programa en sí debe ser también un objeto, de modo que "main" debe estar dentro de la declaración de una clase.

Pero vemos que hay muchas "cosas" rodeando a "main". Estas "cosas" deberán estar siempre, y más adelante volveremos a ellas según las vayamos necesitando, pero vamos a comentar brevemente cada una de ellas:

- **public**: indica que esta función estará accesible para cualquier otro objeto (ya veremos más adelante los distintos especificadores de acceso que permite Java).
- **static**: indica que esta función en un "método estático" o "método de clase", al que se podrá acceder sin necesidad de haber declarado ningún objeto concreto perteneciente a esa clase.
- **void**: indica que esta función no va a devolver ningún valor (por ejemplo, una función que calculase la raíz cuadrada de un número sí debería devolver el valor que ha obtenido).
- **String args[]** nos permitiría leer los "argumentos" que se indicasen en nuestro programa (por ejemplo, cuando escribíamos "edit hola.java", la aplicación que estamos utilizando es "edit", y el argumento -los detalles concretos sobre qué queremos editar- es "hola.java"). También veremos más adelante cómo utilizarlos.

Al igual que decíamos de la clase, todo el bloque de cosas que va a hacer esta función "main" se engloba entre **llaves**:

```
public static void main( String args[] ) {
    [... aquí faltan más cosas ...]
}
```

En concreto, nuestra clase de objetos "Hola Mundo" interacciona únicamente con la **salida en pantalla** de nuestro sistema (System.out), para enviarle el mensaje de que escriba un cierto texto en pantalla (**println**):

```
System.out.println( "Hola Mundo!" );
```

Como se ve en el ejemplo, el texto que queremos escribir en pantalla se debe indicar entre **comillas**.

También es importante el **punto y coma** que aparece al final de esa línea: cada orden en Java deberá terminar con punto y coma (nuestro programa ocupa varias líneas pero sólo tiene una órden, que es “println”).

Ahora ya podemos volver a leer **todo nuestro programa**, con la esperanza de entenderlo un poco más...

```
//
// Aplicación HolaMundo de ejemplo
//

class HolaMundo {
    public static void main( String args[] ) {
        System.out.println( "Hola Mundo!" );
    }
```

Regresar.

Tipos de datos y variable.

Variables.

En nuestro primer ejemplo escribíamos un texto en pantalla, pero este texto estaba prefijado dentro de nuestro programa.

Esto no es lo habitual: normalmente los datos que maneje nuestro programa serán el resultado de alguna operación matemática o de cualquier otro tipo. Por eso, necesitaremos un espacio en el que ir almacenando valores temporales y resultados.

En casi cualquier lenguaje de programación podremos reservar esos "espacios", y asignarles un nombre con el que acceder a ellos. Esto es lo que se conoce como "**variables**".

Por ejemplo, si queremos que el usuario introduzca dos números y el ordenador calcule la suma de ambos números y la muestre en pantalla, necesitaríamos el espacio para almacenar al menos esos dos números iniciales (en principio, para la suma no sería imprescindible, porque podemos mostrarla en pantalla nada más calcularla, sin almacenarla previamente en ningún sitio). Los pasos a dar serían los siguientes:

- Pedir al usuario que introduzca un número.
- Almacenar ese valor que introduzca el usuario (por ejemplo, en un espacio de memoria al que podríamos asignar el nombre "primerNumero").
- Pedir al usuario que introduzca otro número.
- Almacenar el nuevo valor (por ejemplo, en otro espacio de memoria llamado "segundoNumero").
- Mostrar en pantalla el resultado de sumar "primerNumero" y "segundoNumero".

Pues bien, en este programa estaríamos empleando dos variables llamadas "primerNumero" y "segundoNumero". Cada una de ellas sería un espacio de memoria capaz de almacenar un número.

El espacio de memoria que hace falta "reservar" será distinto según la precisión que necesitemos para ese número, o según lo grande que pueda llegar a ser dicho número. Por eso, tenemos disponibles diferentes "**tipos de variables**".

Por ejemplo, si vamos a manejar números sin decimales ("números enteros") de como máximo 9 cifras, nos interesaría el tipo llamado "int" (abreviatura de "integer", que es "entero" en inglés). Este tipo de datos consume un espacio de memoria de 4 bytes (1 Mb es cerca de 1 millón de bytes). Si no necesitamos guardar datos tan grandes (por ejemplo, si nuestros datos va a ser números inferiores a 1.000), podemos emplear el tipo de datos llamado "short" (entero "corto").

Los **tipos de datos númericos** disponibles en Java son los siguientes:

Nombre	¿Admite decimales?	Valor mín.	Valor máx.	Precisión	Ocupa
byte	no	-128	127	-	1 byte
short	no	-32.768	32.767	-	2 bytes
int	no	-2.147.483.648	2.147.483.647	-	4 bytes
long	no	-9.223.372.036.854.775.808	9..223.372.036.854.775.807	-	8 bytes
float	sí	1.401298E-45	3.402823E38	6-7 cifras	4 bytes
double	sí	4.94065645841247E-324	1.79769313486232E308	14-15 cifras	8 bytes

La forma de "**declarar**" variables de estos tipos es la siguiente:

```
int numeroEntero;    // La variable numeroEntero será un número de tipo
"int"
short distancia;     // La variable distancia guardará números "short"
long gastos;         // La variable gastos es de tipo "long"
byte edad;           // Un entero de valores "pequeños"
float porcentaje;    // Con decimales, unas 6 cifras de precisión
double numPrecision; // Con decimales y precisión de unas 14 cifras
```

(es decir, primero se indica el tipo de datos que guardará la variable y después el nombre que queremos dar a la variable).

Se pueden declarar varias variables del mismo tipo "a la vez":

```
int primerNumero, segundoNumero;    // Dos enteros
```

Y también se puede **dar un valor** a las variables a la vez que se declaran:

```
int a = 5;            // "a" es un entero, e inicialmente vale 5
short b=-1, c, d=4;   // "b" vale -1, "c" vale 0, "d" vale 4
```

Los nombres de variables pueden contener letras y números (y algún otro símbolo, como el de subrayado, pero no podrán contener otros muchos símbolos, como los de las distintas operaciones matemáticas posibles, ni podrán tener vocales acentuadas o eñes).

Vamos a aplicarlo con un ejemplo sencillo en Java que sume dos números:

```
    /* ------------------------------*
     *  Introducción a Java - Ejemplo *
```

```
     * -------------------------------*
     *  Fichero: suma.java            *
     *  (Suma dos números enteros)    *
     * -------------------------------*/


class Suma2 {

  public static void main( String args[] ) {

     int primerNumero = 56;    // Dos enteros con valores prefijados
     int segundoNumero = 23;

     System.out.println( "La suma es:" );  // Muestro un mensaje de aviso
     System.out.println( primerNumero+segundoNumero );

       // y el resultado de la operación
   }
}
```

Hay una importante **diferencia** entre las dos órdenes "println": la primera contiene comillas, para indicar que ese texto debe aparecer "tal cual", mientras que la segunda no contiene comillas, por lo que no se escribe el texto "primerNumero+segundoNumero", sino que se intenta calcular el valor de esa expresión (en este caso, la suma de los valores que en ese momento almacenan las variables primerNumero y segundoNumero, 56+23 = 89).

Lo correcto habría sido que los datos no los hubiéramos escrito nosotros en el programa, sino que fuera el usuario quien los hubiera introducido, pero esto es algo relativamente complejo cuando creamos programas "en modo texto" en Java (al menos, comparado con otros lenguajes anteriores), y todavía no tenemos los conocimientos suficientes para crear programas en "modo gráfico", así que más adelante veremos cómo podría ser el usuario de nuestro programa el que introdujese los valores que desea sumar.

Tenemos **otros dos tipos** básicos de variables, que no son para datos numéricos:

- **char**. Será una letra del alfabeto, o un dígito numérico, o un símbolo de puntuación. Ocupa 2 bytes. Sigue un estándar llamado Unicode (que a su vez engloba a otro estándar anterior llamado ASCII).
- **boolean**. se usa para evaluar condiciones, y puede tener el valor "verdadero" (true) o "false" (false). Ocupa 1 byte.

Estos ocho tipos de datos son lo que se conoce como "tipos de datos **primitivos**" (porque forman parte del lenguaje estándar, y a partir de ellos podremos crear otros más complejos).
Nota para C y C++*: estos tipos de datos son muy similares a los que se emplean en C y en C++, apenas con alguna "pequeña" diferencia, como es el hecho de que el tamaño de cada tipo de datos (y los valores que puede almacenar) son independientes en Java del sistema que empleemos, cosa que no ocurre en C y C++, lenguajes en los que un valor de 40.000*

puede ser válido para un "int" en unos sistemas, pero "desbordar" el máximo valor permitido en otros. Otra diferencia es que en Java no existen enteros sin signo ("unsigned"). Y otra más es que el tipo "boolean" ya está incorporado al lenguaje, en vez de corresponder a un entero disinto de cero

Regresar.

Operadores.

Operaciones matemáticas básicas.

Hay varias operaciones matemáticas que son frecuentes. Veremos cómo expresarlas en Java, y también veremos otras operaciones "menos frecuentes".

Las que más encontramos normalmente son:

Operación matemática	Símbolo correspondiente
Suma	+
Resta	-
Multiplicación	*
División	/

Hemos visto un ejemplo de cómo calcular la suma de dos números; las otras operaciones se emplearían de forma similar.

Otras operaciones matemáticas menos habituales.

Esas son las operaciones que todo el mundo conoce, aunque no haya manejado ordenadores. Pero hay otras operaciones que son frecuentes en informática y no tanto para quien no ha manejado ordenadores, pero que aun así deberíamos ver:

Operación	Símbolo
Resto de la división	%
Desplazamiento de bits a la derecha	>>
Desplazamiento de bits a la derecha	<<
Desplazamiento rellenando con ceros	>>>
Producto lógico (and)	&
Suma lógica (or)	|
Suma exclusiva (xor)	^
Complemento	~

El **resto** de la división tambiés es una operación conocida. Por ejemplo, si dividimos 14 entre 3 obtenemos 4 como cociente y 2 como resto, de modo que el resultado de `10 % 3` sería 2.

Las operaciones a **nivel de bits** deberían ser habituales para los estudiantes de informática, pero quizás no tanto para quien no haya trabajado con el sistema binario. No entraré por ahora en más detalles, salvo poner un par de ejemplos para quien necesite emplear estas operaciones: para desplazar los bits de "a" dos posiciones hacia la izquierda, usaríamos `b << 2`; para invertir los bits de "c" (complemento) usaríamos `~c`; para hacer una suma lógica de los bits de "d" y "f" sería `d | f`.

Incremento y asignaciones abreviadas.

Hay varias operaciones muy habituales , que tienen una sintaxis abreviada en Java.

Por ejemplo, para sumar 2 a una variable "a", la forma "normal" de conseguirlo sería:

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

pero existe una forma abreviada en Java:

```
a += 2;
```

Al igual que tenemos el operador += para aumentar el valor de una variable, tenemos -= para disminuirlo, /= para dividirla entre un cierto número, *= para multiplicarla por un número, y así sucesivamente. Por ejemplo, para multiplicar por 10 el valor de la variable "b" haríamos

```
b *= 10;
```

También podemos aumentar o disminuir en una unidad el valor de una variable, empleando los operadores de "incremento" (++) y de "decremento" (--). Así, para sumar 1 al valor de "a", podemos emplear cualquiera de estas tres formas:

```
a = a +1;
a += 1;
a++;
```

Los operadores de incremento y de decremento se pueden escribir antes o después de la variable. Así, es lo mismo escribir estas dos líneas:

```
a++;
++a;
```

Pero hay una diferencia si ese valor se asigna a otra variable "al mismo tiempo" que se incrementa/decrementa:

```
int c = 5;
int b = c++;
```

da como resultado c = 6 y b = 5 (se asigna el valor a "b" antes de incrementar "c") mientras que

```
int c = 5;
int b = ++c;
```

da como resultado c = 6 y b = 6 (se asigna el valor a "b" después de incrementar "c").

Operadores relacionales.

También podremos comprobar si entre dos números (o entre dos variables) existe una cierta relación del tipo "¿es a mayor que b?" o "¿tiene a el mismo valor que b?". Los operadores que utilizaremos para ello son:

Operación	Símbolo
Mayor que	>
Mayor o igual que	>=
Menor que	<
Menor o igual que	<=
Igual que	== (dos símbolos de "igual")
Distinto de	!=

Así, por ejemplo, para ver si el valor de una variable "b" es distinto de 5, escribiríamos algo parecido (veremos la sintaxis correcta un poco más adelante) a

```
SI  b != 5 ENTONCES ...
```

o para ver si la variable "a" vale 70, sería algo como (nuevamente, veremos la sintaxis correcta un poco más adelante)

```
SI  a == 70 ENTONCES ...
```

Es **muy importante** recordar esta diferencia: para asignar un valor a una variable se emplea un único signo de igual, mientras que para comparar si una variable es igual a otra (o a un cierto valor) se emplean dos signos de igual.

Operadores lógicos.

Podremos enlazar varias condiciones, para indicar qué hacer cuando se cumplan ambas, o sólo una de ellas, o cuando una no se cumpla. Los operadores que nos permitirán enlazar condiciones son:

Operación	Símbolo
Y	&&

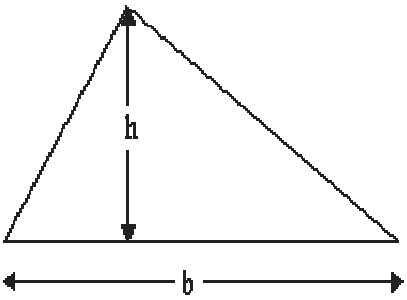
O	||
No	!

Por ejemplo, la forma de decir "si a vale 3 y b es mayor que 5, o bien a vale 7 y b no es menor que 4" en una sintaxis parecida a la de Java (aunque todavía no es la correcta) sería:

```
SI (a==3 && bb>5) || (a==7  && ! (b<4))
```

Regresar.

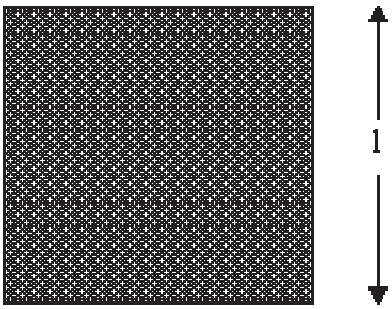
Cálculo de áreas y volúmenes de figuras geométricas.



Dada la siguiente información, hacer un programa que permita calcular el área de un triangulo, de un cuadrado, y de un rectángulo, así como los volúmenes de un cilindro y de un cono.

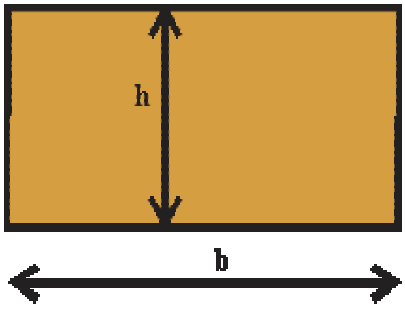
El triángulo es un polígono formado por tres lados y tres ángulos.
La suma de todos sus ángulos siempre es 180 grados.
Para calcular el área se emplea la siguiente fórmula:

Área del triángulo = (base * altura) / 2



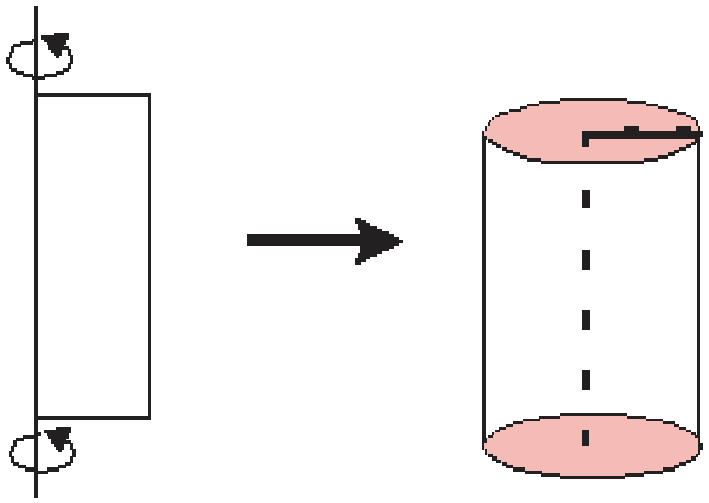
El cuadrado es un polígono de cuatro lados, con la particularidad de que todos ellos son iguales. Además sus cuatro ángulos son de 90 grados cada uno.
El área de esta figura se calcula mediante la fórmula:

Área del cuadrado = lado al cuadrado



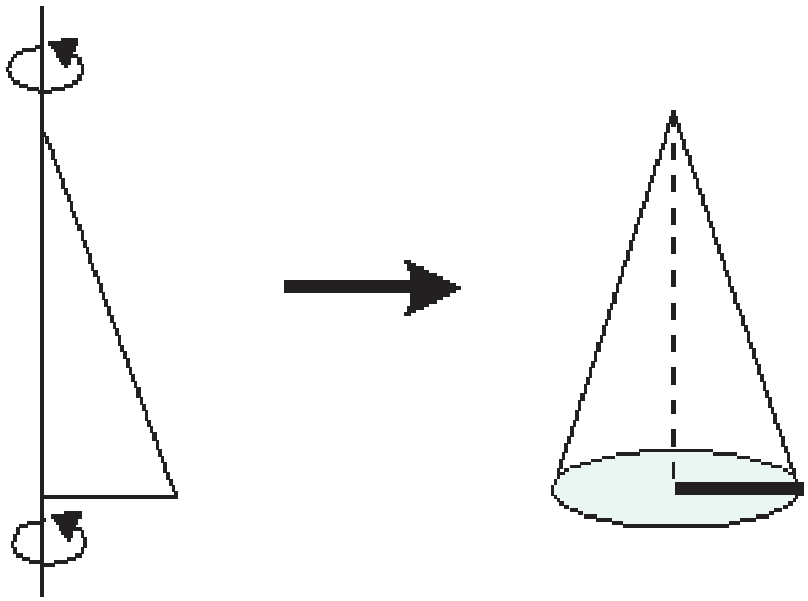
El rectángulo es un polígono de cuatro lados, iguales dos a dos. Sus cuatro ángulos son de 90 grados cada uno.
El área de esta figura se calcula mediante la fórmula:

Área del rectángulo = base * altura



El cilindro es el sólido engendrado por un rectángulo al girar en torno a uno de sus lados.
Para calcular su área lateral, su área total así como para ver su desarrollo pulsar sobre la figura anterior
Para calcular su volumen se emplea la siguiente fórmula:

Volumen del cilindro = área de la base * altura



El cono es el sólido engendrado por un triángulo rectángulo al girar en torno a uno de sus catetos.
Para calcular su área lateral, su área total así como para ver su desarrollo pulsar sobre la figura anterior
Para calcular su volumen se emplea la siguiente fórmula:

Volumen del cono = (área de la base * altura) / 3

La implementación en Java es:

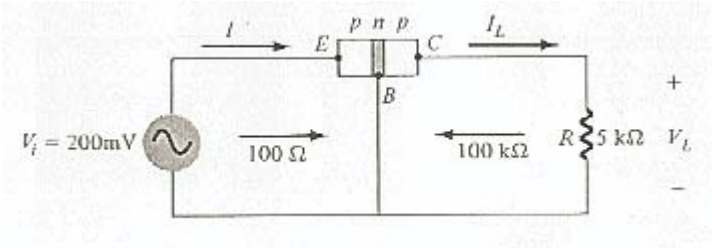
```
/**
 * Title: Calculo de areas y volumenes
 * Calcula utilizando simples sentencias las areas y volumenes de figuras
 * 2003
 * Company: UMSNH
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

public class ej021 {
  public static void main(String[] args)
  {
    // datos

    double base = 10, altura = 5, radio = 2;

    // areas

    double Atriangulo  = (base * altura) / 2.0;
    double Acuadrado   = base*base;
    double Arectangulo = base * altura;
```

```
    // volumenes

    double Abase       = 3.1416*radio*radio;
    double Vcilindro   = Abase * altura ;
    double Vcono       = Abase * altura / 3.0;

    System.out.println("Area de triangulo  = " + Atriangulo);
    System.out.println("Area de cuadrado   = " + Acuadrado);
    System.out.println("Area de rectangulo = " + Arectangulo);

    System.out.println("Volumen cilindro   = " + Vcilindro);
    System.out.println("Volumen cono       = " + Vcono);


  }

}
```

Regresar.

Ley de ohm. Análisis básico de transistores.

Para el circuito mostrado en la figura, determinar la ganancia de voltaje de un transistor en configuración de base común.



Hacer un programa en Java. Utilice los datos mostrados en la figura.

```
/**
 * Title: Calculo de la ganacia de voltaje para un transistor
 * Description: Programa para determinar ganancias de voltaje
 * Copyright: Copyright (c) 2003
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */


public class ej023 {
  public static void main(String[] args) {

    double Vi = 200e-3, Re = 100, Rc = 100e+3, R = 5e3;
    double I, Il, Vl, Av;

    // calculo de la corriente de emisor.

    I = Vi/Re;
    System.out.println("I = " + I);

    // Asumiendo una ganacia alpha = 1 y que Ic = Ie

    Il = I;
    System.out.println("Il = " + Il);

    // el voltaje en la resistencia R es

    Vl = Il*R;
    System.out.println("Vl = " + Vl);


    // finalmente la ganacia de voltaje es

    Av = Vl/Vi;
    System.out.println("Av = " + Av);
```

```
    }
}
```

Regresar.

Instrucciones condicionales.

Comprobación de condiciones.

En cualquier lenguaje de programación es habitual tener que comprobar si se cumple una cierta **condición**. La forma "normal" de conseguirlo es empleando una construcción que recuerda a:

```
SI   condición_a_comprobar   ENTONCES   pasos_a_dar
```

En el caso de Java, la forma exacta será empleando **if** (si, en inglés), seguido por la condición entre paréntesis, e indicando finalmente entre llaves los pasos que queremos dar si se cumple la condición, así :

```
if (condición)  { sentencias }
```

Por ejemplo,

```
if (x == 3)  {
  System.out.println( "El valor es correcto" );
  resultado = 5;
}
```

***Nota**: Si sólo queremos dar un paso en caso de que se cumpla la condición, no es necesario emplear llaves. Las llaves serán imprescindibles sólo cuando haya que hacer varias cosas:*

```
if (x == 3)
  System.out.println( "El valor es correcto" );
```

Una primera **mejora**, que también permiten muchos lenguajes de programación, es indicar qué hacer en caso de que no se cumpla la condición. Sería algo parecido a

```
SI  condición_a_comprobar  ENTONCES  pasos_a_dar  EN_CASO_CONTRARIO
pasos_alternativos
```

que en Java escribiríamos así:

```
if (condición)  { sentencias1 }   else   { sentencias2 }
```

Por ejemplo,

```
if (x == 3)  {
  System.out.println( "El valor es correcto" );
  resultado = 5;
}
else  {
```

```
      System.out.println( "El valor es incorrecto" );
      resultado = 27;
    }
```

Si queremos comprobar **varias condiciones**, podemos utilizar varios if - else - if - else - if encadenados, pero tenemos una forma más elegante en Java de elegir entre distintos valores posibles que tome una cierta expresión. Su formato es éste:

```
  switch (expresion) {
    case valor1: sentencias1; break;
    case valor2: sentencias2; break;
    case valor3: sentencias3; break;
    ...                                  // Puede haber más valores
}
```

Es decir, después de la orden **switch** indicamos entre paréntesis la expresión que queremos evaluar. Después, tenemos distintos apartados, uno para cada valor que queramos comprobar; cada uno de estos apartados se precede con la palabra **case**, indica los pasos a dar si es ese valor el que se da, y terminar con **break**.

Un ejemplo sería:

```
  switch ( x * 10) {
    case 30: System.out.println( "El valor de x era 3" ); break;
    case 50: System.out.println( "El valor de x era 5" ); break;
    case 60: System.out.println( "El valor de x era 6" ); break;
}
```

También podemos indicar qué queremos que ocurra en el caso de que el valor de la expresión no sea ninguno de los que hemos detallado:

```
  switch (expresion) {
    case valor1: sentencias1; break;
    case valor2: sentencias2; break;
    case valor3: sentencias3; break;
    ...                                  // Puede haber más valores
    default: sentencias;                 // Opcional: valor por defecto
}
```

Por ejemplo, así:

```
  switch ( x * 10) {
    case 30: System.out.println( "El valor de x era 3" ); break;
    case 50: System.out.println( "El valor de x era 5" ); break;
    case 60: System.out.println( "El valor de x era 6" ); break;
    default: System.out.println( "El valor de x no era 3, 5 ni 6" );
break;
}
```

Podemos conseguir que en varios casos se den los mismos pasos, simplemente eliminado la orden "break" de algunos de ellos. Así, un ejemplo algo más completo podría ser:

```
  switch ( x ) {
    case 1:
    case 2:
    case 3:
        System.out.println( "El valor de x estaba entre 1 y 3" );
        break;
    case 4:
    case 5:
        System.out.println( "El valor de x era 4 o 5" );
        break;
    case 6:
        System.out.println( "El valor de x era 6" );
        valorTemporal = 10;
        System.out.println( "Operaciones auxiliares completadas" );
        break;
    default:
        System.out.println( "El valor de x no estaba entre 1 y 6" );
        break;
}
```

Un poco más adelante propondremos ejercicios que permitan afianzar todos estos conceptos.

Existe una construcción adicional, que permite comprobar si se cumple una condición y devolver un cierto valor según si dicha condición se cumple o no. Es lo que se conoce como el "operador condicional (?)":

```
condicion ? resultado_si cierto : resultado_si_falso
```

Es decir, se indica la condición seguida por una interrogación, después el valor que hay que devolver si se cumple la condición, a continuación un símbolo de "dos puntos" y finalmente el resultado que hay que devolver si no se cumple la condición.

Es frecuente emplearlo en asignaciones (aunque algunos autores desaconsejan su uso porque puede resultar menos legible que un "if"), como en este ejemplo:

```
x = (a == 10) ? b*2 : a ;
```

En este caso, si a vale 10, la variable tomará el valor de b*2, y en caso contrario tomará el valor de a. Esto también se podría haber escrito de la siguiente forma, más larga pero más legible:

```
if (a == 10)  x = b*2; else x = a;
```

Ejemplo:

Hacer un programa, que dados tres números determine cual es el mayor y cual es el menor.

```
/**
 * Title: Mayor y menor
 * Description: Dados tres numeros dice cual es el mayor y cual el menor
```

```
 * Copyright: Copyright (c) 2003
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */

class ej003
{
  public static void main (String args[])
  {
    int a, b, c, d;

    a= 4; b = 2; c = 3;

    d = (a > b) ?  a : b;
    d = (d > c) ?  d : c;

    System.out.println("el Mayor es " + d);

    // otra forma

    if(a> b) d = a;
    else d = b;
    if(d< c) d = c;
    System.out.println("el Mayor es " + d);

  }
}
```

Regresar.

Solución de una ecuación cuadrática.

La formula general de una ecuación cuadrática es $f(x) = Ax^2 + Bx + C$, y resulta de interés determinar los puntos donde esta ecuación cruza el eje x, es decir $f(x)=0$. La solución de este problema esta dado por la formula general:

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

La programación de esta ecuación no tiene mayor problema, solamente debemos cuidar, cuando $B^2-4AC<0$, ya que en dicho caso, la raíz cuadrada será un número imaginario.

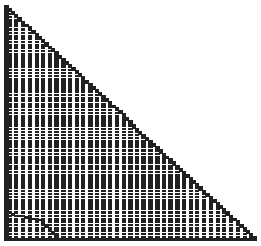
La implementación en Java queda:

```
/**
 * Title: Raices de un Polinomioo
 *  Description: este ejemplo calcula las raices de una ecuacion de
segundo grado
 * Copyright: Copyright (c) 2003
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */

class ej002
{
  public static void main(String arqs[])
  {
    double A=9, B=6, C=4;
    double X1=0, X2=0, G=0, I=0, D=0;

    D=(B*B-(4*A*C));

    if (D>0)
    {
      // si las reices son positivas

      X1=(-B+Math.sqrt(D))/(2*A);
      X2=(-B-Math.sqrt(D))/(2*A);
      System.out.println("X1="+X1);
      System.out.println("X2="+X2);
    }
    else
    {
      // si las raices son negativas
      G=-B/(2*A);
      I=Math.sqrt(-D)/(2*A);
      System.out.println("X1="+""+G+"+"+I+"j");
```

```
      System.out.println("X2="+""+G+"-"+I+"j");
    }
  }
}
```

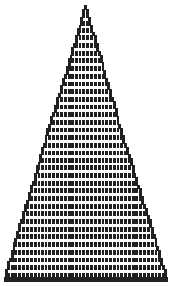
Regresar.

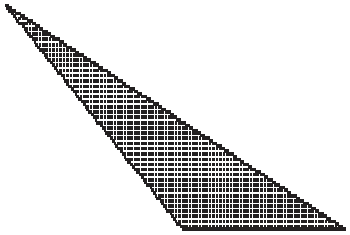
Tipos de triángulos.



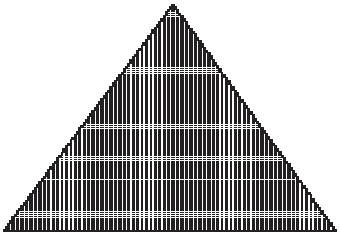
El triángulo rectángulo es aquél que tiene un ángulo de 90 grados

El triángulo isósceles El triángulo isósceles aquél que tiene dos lados iguales y uno desigual.





El triángulo escaleno es aquél que tiene los tres lados desiguales y por lo tanto sus ángulos.



El triángulo equilátero es aquél que tiene los tres lados iguales y por lo tanto sus ángulos, siendo cada uno de 60 grados.

Dada esta información hacer un programa en Java que permita determinar de que tipo de triangulo se trata, dados el tamaño de cada uno de sus tres lados.

```
/**
 * Title: Tipos de triangulos
 * Description: Programa para determinar el tipo de triangulo dados tres
lados
 * Copyright: Copyright (c) 2003
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */


public class ej022 {
  public static void main(String[] args)
  {
    double a = 1, b = 1.5, c = 2; // tamaño de los lados
    double A, B, C; // angulos
    double suma;

    A = Math.acos( (b * b + c * c - a * a) / (2 * b * c)) * 180 /
Math.PI;
    B = Math.acos( (a * a + c * c - b * b) / (2 * a * c)) * 180 /
Math.PI;
    C = Math.acos( (a * a + b * b - c * c) / (2 * a * b)) * 180 /
Math.PI;
    suma = A + B + C;

    System.out.println("Los lados son              = [ " + a + ", " + b +
", " +
                       c + "]");
    System.out.println("Los angulos son            = [ " + A + ", " + B +
", " +
                       C + "]");
    System.out.println("La suma de sus angulos es = " + suma);

    if (suma == 180) {
      if (a == b && b == c)
        System.out.println("Es un triangulo equilatero");
      else {
        if (A == 90 || B == 90 || C == 90)
          System.out.println("Es un triangulo rectangulo");
        else {
          if (a == b || b == c || a == c)
```

```
            System.out.println("Es un triangulo isoceles");
          else {
            if (A != B && B != C && C != A)
              System.out.println("Es un triangulo escaleno");
          }
        }
      }
    }
    else
      System.out.println("Estos valores no forman un triangulo ");
  }

}
```

Regresar.

Instrucciones de repetición.

Con frecuencia tendremos que hacer que una parte de nuestro programa se repita (algo a lo que con frecuencia llamaremos "**bucle**"), bien mientras se cumpla una condición o bien un cierto número prefijado de veces.

Java incorpora varias formas de conseguirlo. La primera que veremos es la orden "**while**", que hace que una parte del programa se repita mientras se cumpla una cierta condición. Su formato será:

```
while (condición)
  sentencia
```

Es decir, la sintaxis es similar a la de "if", con la diferencia de que aquella orden realizaba la sentencia indicada una vez como máximo (si se cumplía la condición), pero "while" puede repetir la sentencia más de una vez (mientras la condición sea cierta). Al igual que ocurría con "if", podemos realizar varias sentencias seguidas (dar "más de un paso") si las encerramos entre llaves:

```
x = 20;
while ( x > 10) {
  System.out.println("Aun no se ha alcanzado el valor limite");
  x --;
}
```

Existe una variante de este tipo de bucle. Es el conjunto **do..while**, cuyo formato es:

```
do {
  sentencias
} while (condicion)
```

En este caso, la condición se comprueba al final, lo que quiere decir que las "sentencias" intermedias se realizarán al menos una vez, cosa que no ocurría en la construcción anterior (un único "while" antes de las sentencias), porque si la condición era falsa desde un principio, los pasos que se indicaban a continuación de "while" no llegaban a darse ni una sola vez.

Un ejemplo típico de esta construcción "do..while" es cuando queremos que el usuario introduzca una contraseña que le permitirá acceder a una cierta información:

```
do {
  System.out.println("Introduzca su clave de acceso");
  claveIntentada = LeerDatosUsuario;  // LeerDatosUsuario realmente
no existe
} while (claveIntentada != claveCorrecta)
```

En este ejemplo hemos supuesto que existe algo llamado "LeerDatosUsuario", para que resulte legible. Pero realmente la situación no es tan sencilla: no existe ese "LeerDatosUsuario", ni sabemos todavía cómo leer información del teclado cuando trabajamos en modo texto (porque supondría hablar de excepciones y de otros conceptos que todavía son demasiado avanzados para nosotros), ni sabemos crear Applets en los que podamos utilizar una casilla de introducción de textos. Así que de momento nos creeremos que algo parecido a lo que hemos escrito podrá llegar a funcionar... pero todavía no lo hace.

Una tercera forma de conseguir que parte de nuestro programa se repita es la orden "**for**". La emplearemos sobre todo para conseguir un número concreto de repeticiones. Su formato es

```
for ( valor_inicial ; condicion_continuacion ; incremento ) {
  sentencias
}
```

(es decir, indicamos entre paréntesis, y separadas por puntos y coma, tres órdenes: la primera dará el valor inicial a una variable que sirva de control; la segunda orden será la condición que se debe cumplir para que se repitan las sentencias; la tercera orden será la que se encargue de aumentar -o disminuir- el valor de la variable, para que cada vez quede un paso menos por dar).

Esto se verá mejor con un ejemplo. Podríamos repetir 10 veces un bloque de órdenes haciendo:

```
for ( i=1 ; i<=10 ; i++ ) {
  ...
}
```

O bien podríamos contar descendiendo desde el 20 hasta el 2, con saltos de 2 unidades en 2 unidades, así:

```
for ( j = 20 ; j > 0 ; j -= 2 )
  System.out.println( j );
```

***Nota**: se puede observar una equivalencia casi inmediata entre la orden "for" y la orden "while". Así, el ejemplo anterior se podría reescribir empleando "while", de esta manera:*

```
j = 20;
while ( j > 0 )
  System.out.println( j );
  j -= 2;
}
```

Precauciones con los bucles: Casi siempre, nos interesará que una parte de nuestro programa se repita varias veces (o muchas veces), pero no indefinidamente. Si planteamos mal la condición de salida, nuestro programa se puede quedar "colgado", repitiendo sin fin los mismos pasos.

Se puede modificar un poco el comportamiento de estos bucles con las órdenes "break" y "continue".

La sentencia "**break**" hace que se salten las instrucciones del bucle que quedan por realizar, y se salga del bucle inmediatamente. Como ejemplo:

```
for ( i=1 ; i<=10 ; i++ ) {
  System.out.println( "Comenzada la vuelta" );
  System.out.println( i );
  if (i==8)  break;
  System.out.println( "Terminada esta vuelta );
}
System.out.println( "Terminado"  );
```

En este caso, no se mostraría el texto "Terminada esta vuelta" para la pasada con i=8, ni se darían las pasadas de i=9 e i=10, porque ya se ha abandonado el bucle.

La sentencia "**continue**" hace que se salten las instrucciones del bucle que quedan por realizar, pero no se sale del bucle sino que se pasa a la siguiente **iteración** (la siguiente "vuelta" o "pasada"). Como ejemplo:

```
for ( i=1 ; i<=10 ; i++ ) {
  System.out.println( "Comenzada la vuelta" );
  System.out.println( i );
  if (i==8)  continue;
  System.out.println( "Terminada esta vuelta );
}
System.out.println( "Terminado"  );
```

En este caso, no se mostraría el texto "Terminada esta vuelta" para la pasada con i=8, pero sí se darían la pasada de i=9 y la de i=10.

También esta la posibilidad de usar una "**etiqueta**" para indicar dónde se quiere saltar con break o continue. Sólo se debería utilizar cuando tengamos un bucle que a su vez está dentro de otro bucle, y queramos salir de golpe de ambos. Es un caso poco frecuente, así que no profundizaremos más, pero sí veremos un ejemplo:

```
for ( i=1 ; i<=10 ; i++ ) {
  System.out.println( "Comenzada la vuelta" );
  System.out.println( i );
  if (i==8)  break salida;
```

```
      System.out.println( "Terminada esta vuelta );
    }
    System.out.println( "Terminado"  );
    salida:
    ...
```

En este caso, a mitad de la pasada 8 se saltaría hasta la posición que hemos etiquetado como "salida" (se define como se ve en el ejemplo, terminada con el símbolo de "dos puntos"), de modo que no se escribiría en pantalla el texto "Terminado" (lo hemos "saltado").

Ejemplo.

Hacer un programa que permita calcular la siguiente sumatoria

$$S = \sum_{i=0}^{64} 2^i$$

```
/**
 * Title: Sumatoria
 * Description: Este programa calcula la suma de 1 + 2 + 4 + 8+ 16 +
...2n
 * Copyright: Copyright (c) 2003
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */


class ej005
{
  public static void main (String args[])
  {
    float i=1;
    float s=0;
    do
    {
      s=((2*s) +1);
      System.out.println("s="+s);
      i++;
    }
    while (i<=64);
  }
}
```

Regresar.

Métodos iterativos para resolver *x=g(x)*

Necesitamos una regla, fórmula o función $g(x)$ con la cual calcularemos temimos sucesivos, para un valor de partida P_0. Lo que se produce es una sucesión de valores $\{P_k\}$ obtenida mediante el proceso iterativo $P_{k+1}= g(P_k)$. La sucesión se ajusta al siguiente patrón:

$$P_1= g(P_0)$$
$$P_2= g(P_1)$$
$$P_3= g(P_2)$$

$$P_{k+1}= g(P_k)$$

Lo que podemos observar de ésta sucesión es que si $P_{k+1}->P_k$ la sucesión converge a un valor, pero en el caso de que P_{k+1} no tienda a P_k tenemos una sucesión divergente.

Ejemplo.

$$x = Ax \text{ con } P_0=1$$

$$P_1= A$$
$$P_2= A*A=A^2$$
$$P_3= A* A^2= A^3$$
$$\ldots$$
$$P_k= A^k$$

Métodos de punto fijo.

Un punto fijo de una función $g(x)$ es un número real P tal que $P=g(P)$. Geométricamente hablando, los puntos fijos de una función $g(x)$ son los puntos de intersección de la curva $y=g(x)$ con la recta $y=x$.

Ejemplo.

Consideremos la función $x=e^{-x}$ y damos un valor inicial $P_0 = 0.5$

$P_1= e^{-0.5} = 0.606531$
$P_2= 0.545339$
$P_3= 0.579703$
...
$P_k=0.567143$

La solución en Java para este algoritmo es:

```
/**
 * <p>Title: Iteracion de punto fijo</p>
 * <p>Description: resuelve una ecuacion por el método de punto fijo</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: UMSNH</p>
 * Dr. Felix Calderon Solorio.
```

```
 * @version 1.0
 */

public class ej018 {
  static public void main(String args[])
  {
    int iter = 0;

    double x0, x1;

    x0 = x1 = 0.5;

    do
    {
      System.out.println("X("+iter+")= " + x1);
      x0 = x1;
      x1 = Math.exp(-x0);
      iter ++;
    }while (Math.abs((x0-x1)/x1) > 0.001);
  }
}
```

Solución de Sistemas no lineales.

Consideremos el caso de la funciones :

$$f_1(x,y) = x^2 - 2x - y + 0.5$$
$$f_2(x,y) = x^2 + 4y^2 - 4$$

y queremos encontrar un método para resolver

$$f_1(x,y) = 0 \; y \; f_2(x,y) = 0$$

Podemos resolver el sistema, utilizando el método de iteración de punto fijo. Para ello despejamos de la ecuación 1 a *x*:

$$x = (x^2 - y + 0.5)/2$$

A la segunda ecuación agregamos un término

$$x^2 + 4y^2 - 8y - 4 = -8y$$
$$y = (-x^2 - 4y^2 + 8y + 4)/8$$

De esto tenemos un sistema en sucesión dado por:

$$x_{k+1} = (x_k^2 - y_k + 0.5)/2$$
$$y_{k+1} = (-x_k^2 - 4y_k^2 + 8y_k + 4)/8$$

La solución de esta sucesión para un valor inicial [0,1] es:

k	X	y
0	0.0000	1.0000
1	-0.2500	1.0000
2	-0.2188	0.9922

3	-0.2222	0.9940
4	-0.2223	0.9938
5	-0.2222	0.9938
6	-0.2222	0.9938

Regresar.

Manejo de matrices y vectores.

Los vectores.

Imaginemos que tenemos que hallar el promedio de 10 números que introduzca el usuario (o realizar cualquier otra operación con ellos). Parece evidente que tiene que haber una solución más cómoda que definir 10 variables distintas y escribir 10 veces las instrucciones de avisar al usuario, leer los datos que teclee, y almacenar esos datos. Si necesitamos manejar 100, 1.000 o 10.000 datos, resulta todavía más claro que no es eficiente utilizar una variable para cada uno de esos datos.

Por eso se emplean los arrays (o arreglos). Un array es una variable que puede contener varios dato del mismo tipo. Para acceder a cada uno de esos datos emplearemos corchetes. Por ejemplo, si definimos una variable llamada "m" que contenga 10 números enteros, accederemos al primero de estos números como m[0], el último como m[9] y el quinto como m[4] (se empieza a numerar a desde 0 y se termina en n-1). Veamos un ejemplo que halla la media de cinco números (con decimales, "double"):

```
// Array1.java
// Aplicación de ejemplo con Arrays
// Introducción a Java

class Array1 {
    public static void main( String args[] ) {

       double a[] = { 10, 23.5, 15, 7, 8.9 };
       double total = 0;
       int i;

       for (i=0; i<5; i++)
         total += a[i];

       System.out.println( "La media es:" );
       System.out.println( total / 5 );
    }
}
```

Para definir la variable podemos usar **dos formatos**: "double a[]" (que es la sintaxis habitual en C y C++) o bien "double[] a" (que es la sintaxis recomendada en Java, y posiblemente es una forma más "razonable" de escribir "la variable a es un array de doubles").

***Nota**: Quien venga de C y/o C++ tiene que tener en cuenta que en Java no son válidas definiciones como float a[5]; habrá que asignar los valores como acabamos de hacer, o reservar espacio de la forma que veremos a continuación.*

Lo habitual no será conocer los valores en el momento de teclear el programa, como hemos hecho esta vez. Será mucho más frecuente que los datos los teclee el usuario o bien que los leamos de algún fichero, los calculemos, etc. En este caso, tendremos que reservar el espacio, y los valores los almacenaremos a medida que vayamos conociéndolos.

Para ello, primero **declararíamos** que vamos a utilizar un array, así:

```
double[] datos;
```

y después **reservaríamos** espacio (por ejemplo, para 1.000 datos) con

```
datos = new double [1000];
```

Estos dos pasos se pueden dar en uno solo, así:

```
double[] datos = new double [1000];
```

y daríamos los **valores** de una forma similar a la que hemos visto en el ejemplo anterior:

```
datos[25] = 100 ; datos[0] = i*5 ; datos[j+1] = (j+5)2 = new double
[1000];
```

Vamos a ver un ejemplo algo más completo, con tres arrays de números enteros, llamados a, b y c. A uno de ellos (a) le daremos valores al definirlo, otro lo definiremos en dos pasos (b) y le daremos fijos, y el otro lo definiremos en un paso y le daremos valores calculados a partir de ciertas operaciones:

```
// Array2.java
// Aplicación de ejemplo con Arrays
// Introducción a Java

class Array2 {
    public static void main( String args[] ) {

       int i;  // Para repetir con bucles "for"


       // ------  Primer array de ejemplo
       int[] a = { 10, 12345, -15, 0, 7 };

       System.out.println( "Los valores de a son:" );
       for (i=0; i<5; i++)
         System.out.println( a[i] );

       // ------  Segundo array de ejemplo
       int[] b;
       b = new int [3];
       b[0] = 15; b[1] = 132; b[2] = -1;
```

```
            System.out.println( "Los valores de b son:" );
            for (i=0; i<3; i++)
              System.out.println( b[i] );

            // ------  Tercer array de ejemplo
            int j = 4;
            int[] c = new int[j];

            for (i=0; i<j; i++)
              c[i] = (i+1)*(i+1);

            System.out.println( "Los valores de c son:" );
            for (i=0; i<j; i++)
              System.out.println( c[i] );
        }
    }
```

Matrices.

Las matrices se definen, como un arreglo bidimensional, en donde tenemos un número de reglones N y un número de columnas M. La representación matemática de una matriz es :

a_{11}	a_{12}	...	a_{1M}
a_{21}	a_{22}	...	a_{2M}
...	...	...	...
a_{N1}	a_{N2}	...	a_{NM}

Para hacer la definición en Java de este arreglo hacemos

```
double A[][] = new double [N][M];
```

donde A es el nombre del arreglo N el número de renglones y M el número de columnas.

Para hacer referencia al elemento en el i-esimo renglón y la j-esima columna hacemos A[i][j].

Otra forma de hacer la definición de un arreglo es mediante la siguiente instrucción

```
double a[][] = {{1,2,3}, {4,5,6}, {7,8,9}};
```

esta instrucción crea un arreglo de 3 renglones con 3 columnas y los elementos lo dispone de la siguiente manera

A =

1	2	3
4	5	6
7	8	9

Ejemplo.

Hacer un programa que almacene la matriz anterior en un arreglo, la eleve al cuadrado e imprima el resultado en la pantalla.

```
class arreglos
{
  static public void main(String args[])
  {
    double a[][] = {{1,2,3}, {4,5,6}, {7,8,9}};
    double b[][] = new double [3][3];
    int i, j;

    for(i=0; i< 3; i++)
      for(j=0; j<3; j++)
        b[i][j] = a[i][j] * a[i][j];


    for(i=0; i< 3; i++)
    {
      for(j=0; j<3; j++)
        System.out.print(b[i][j] + " ");
      System.out.println("");
    }

  }
}
```

Las cadenas de texto.

Una cadena de texto (en inglés, "string") es un bloque de letras, que usaremos para poder almacenar palabras y frases. En algunos lenguajes, podríamos utilizar un "array" de "chars" para este fin, pero en Java no es necesario, porque tenemos un tipo "cadena" específico ya incorporado en el lenguaje.

Realmente en Java hay **dos "variantes"** de las cadenas de texto: existe una clase llamada "String" y otra clase llamada "StringBuffer". Un "**String**" será una cadena de caracteres constante, que no se podrá modificar (podremos leer su valor, extraer parte de él, etc.; para cualquier modificación, realmente Java creará una nueva cadena), mientras que un "**StringBuffer**" se podrá modificar "con más facilidad" (podremos insertar letras, dar la vuelta a su contenido, etc) a cambio de ser ligeramente menos eficiente (más lento).

Vamos a ver las principales posibilidades de cada uno de estos dos tipos de cadena de texto y luego lo aplicaremos en un ejemplo.

Podemos "concatenar" cadenas (juntar dos cadenas para dar lugar a una nueva) con el signo +, igual que sumamos números. Por otra parte, los métodos de la clase **String** son:

Método	***Cometido***
length()	Devuelve la longitud (número de caracteres) de la cadena
charAt (int pos)	Devuelve el carácter que hay en una cierta posición
toLowerCase()	Devuelve la cadena convertida a minúsculas
toUpperCase()	Devuelve la cadena convertida a mayúsculas
substring(int desde, int cuantos)	Devuelve una subcadena: varias letras a partir de una posición dada
replace(char antiguo, char nuevo)	Devuelve una cadena conun carácter reemplazado por otro
trim()	Devuelve una cadena sin espacios de blanco iniciales ni finales
startsWith(String subcadena)	Indica si la cadena empieza con una cierta subcadena
endsWith(String subcadena)	Indica si la cadena termina con una cierta subcadena
indexOf(String subcadena, [int desde])	Indica la posición en que se encuentra una cierta subcadena (buscando desde el principio, a partir de una posición opcional)
lastIndexOf(String subcadena, [int desde])	Indica la posición en que se encuentra una cierta subcadena (buscando desde el final, a partir de una posición opcional)
valueOf(objeto)	Devuelve un String que es la representación como texto del objeto que se le indique (número, boolean, etc.)
concat(String cadena)	Devuelve la cadena con otra añadida a su final (concatenada) También se pueden concatenar cadenas con "+"
equals(String cadena)	Mira si las dos cadenas son iguales (lo mismo que "= =")
equals-IgnoreCase(String cadena)	Comprueba si dos cadenas son iguales, pero despreciando las diferencias entre mayúsculas y minúsculas
compareTo(String cadena2)	Compara una cadena con la otra (devuelve 0 si son iguales, negativo si la cadena es "menor" que cadena2 y positivo si es "mayor").

Como se ve, en ningún momento estamos modificando el String de partida. Eso sí, en muchos de los casos creamos un String modificado a partir del original.

El método "compareTo" se basa en el **órden lexicográfico**: una cadena que empiece por "A" se considerará "menor" que otra que empiece por la letra "B"; si la primera letra es igual en ambas cadenas, se pasa a comparar la segunda, y así sucesivamente. Las mayúsculas y minúsculas se consideran diferentes.

Hay alguna otra posibilidad, de uso menos sencillo, que no veremos (al menos por ahora), como la de volcar parte del String en un array de chars o de bytes.

Los métodos de la clase **StringBuffer** son:

Método	***Cometido***
length()	Devuelve la longitud (número de caracteres) de la cadena
setLength()	Modifica la longitud de la cadena (la trunca si hace falta)
charAt (int pos)	Devuelve el carácter que hay en una cierta posición
setCharAt(int pos, char letra)	Cambia el carácter que hay en una cierta posición
toString()	Devuelve el StringBuffer convertido en String
reverse()	Cambia el orden de los caracteres que forman la cadena
append(objeto)	Añade otra cadena, un número, etc. al final de la cadena
insert(int pos, objeto)	Añade otra cadena, un número, etc. en una cierta posición

Al igual que ocurre con los strings, existe alguna otra posibilidad más avanzada, que no he comentado, como la de volcar parte del String en un array de chars, o de comprobar la capacidad (tamaño máximo) que tiene un StringBuffer o fijar dicha capacidad.

Un **comentario extra** sobre los Strings: Java convertirá a String todo aquello que indiquemos entre comillas dobles. Así, son válidas expresiones como "Prueba".length() y también podemos concatenar varias expresiones dentro de una orden System.out.println:

```
System.out.println( "Texto: " + texto1 + 5  + 23.5 );
```

Vamos a ver un **ejemplo** que aplique la mayoría de todo esto:

```
// Strings1.java
// Aplicación de ejemplo con Strings
// Introducción a Java,

class Strings1 {
    public static void main( String args[] ) {

       String texto1 = "Hola";                 // Forma "sencilla"
       String texto2 = new String("Prueba");   // Usando un constructor

       System.out.println( "La primera cadena de texto es :" );
       System.out.println( texto1 );
```

```
        System.out.println( "Concatenamos las dos: " + texto1 + texto2 );
        System.out.println( "Concatenamos varios: " + texto1 + 5 + " " + 23.5 );
        System.out.println( "La longitud de la segunda es: " + texto2.length() );
        System.out.println( "La segunda letra de texto2 es: "
          + texto2.charAt(1) );
        System.out.println( "La cadena texto2 en mayúsculas: "
          + texto2.toUpperCase() );
        System.out.println( "Tres letras desde la posición 1: "
          + texto2.substring(1,3) );
        System.out.println( "Comparamos texto1 y texto2: "
          + texto1.compareTo(texto2) );

        if (texto1.compareTo(texto2) < 0)
          System.out.println( "Texto1 es menor que texto2" );

        StringBuffer texto3 = new StringBuffer("Otra prueba");

        texto3.append(" mas");
        System.out.println( "Texto 3 ahora es: " + texto3 );
        texto3.insert(2, "1");
        System.out.println( "Y ahora es: " + texto3 );
        texto3.reverse();
        System.out.println( "Y ahora: " + texto3 );


    }
}
```

El resultado de este programa sería el siguiente:

```
La primera cadena de texto es :
Hola
Concatenamos las dos: HolaPrueba
Concatenamos varios: Hola5 23.5
La longitud de la segunda es: 6
La segunda letra de texto2 es: r
La cadena texto2 en mayúsculas: PRUEBA
Tres letras desde la posición 1: ru
Comparamos texto1 y texto2: -8
Texto1 es menor que texto2
Texto 3 ahora es: Otra prueba mas
Y ahora es: Ot1ra prueba mas
Y ahora: sam abeurp ar1tO
```

Regresar.

Algoritmos de Búsqueda

Secuencial

La forma más simple de búsqueda es la búsqueda secuencial. Esta búsqueda es aplicable a una tabla "organizada", ya sea como un arreglo o como una lista ligada. Supongamos que x es un arreglo de N llaves, de x(0) a x(N-1) y y el termino que deseamos encontrar en el arreglo, el algoritmo de búsqueda queda como:

```
static public int busqueda_secuencial(double x[], double y, int N)
{
  int i;

  for(i=0; i<N; i++)
    if(x[i] == y) return i;

  return -1;
}
```

Búsqueda Binaria

El método de búsqueda más eficiente en una tabla secuencial sin usar índices o tablas auxiliares es el método de búsqueda binaria. Básicamente, se compara el argumento con la llave del elemento medio de la tabla. Sin son iguales, la búsqueda termina con éxito; en caso contrario, se busca de manera similar en la mitad superior o inferior de la tabla, según sea el caso.

```
static public int busqueda_binaria(double x[], double y, int N)
{
  int inicio = 0, fin = N, mitad;

  while(inicio <= fin)
  {
    mitad = (fin + inicio)/2;

    if(y == x[mitad]) return mitad;

    if(y < x[mitad]) fin = mitad -1;
    else inicio = mitad + 1;
  }
  return -1;
}
```

El siguiente código muestra la implementación de ambos y un programa principal para hacer un llamado a cada uno de los métodos de búsqueda antes mencionados.

```
class busqueda
{
  static public void main(String args[])
  {

    double x[] = {1,2,4,7, 10, 11, 20, 21, 32};
    int N = 9, i;
    double y = 10;

    i = busqueda_secuencial(x, y, N);
    System.out.println("El valor "+ y + " se encuentra en la posicion " +
i);
    System.out.println("El valor "+ x[i] + " se encuentra en la posicion
"
      + busqueda_binaria(x, y, N));
  }

  static public int busqueda_secuencial(double x[], double y, int N)
  {
    int i;

    for(i=0; i<N; i++)
      if(x[i] == y) return i;

    return -1;
  }

  static public int busqueda_binaria(double x[], double y, int N)
  {
    int inicio = 0, fin = N, mitad;

    while(inicio <= fin)
    {
      mitad = (fin + inicio)/2;

      if(y == x[mitad]) return mitad;

      if(y < x[mitad]) fin = mitad -1;
      else inicio = mitad + 1;
    }
    return -1;
  }

}
```

Regresar.

Ordenamiento de Burbuja

El primer ordenamiento que presentamos es quizá el más ampliamente conocido entre los estudiantes que se inician en la programación. Una de las características de este ordenamiento es que es fácil de entender y programar. Aunque, es uno de los algoritmos mas ineficiente.

La idea básica subyacente en el ordenamiento de burbuja es pasar a través del arreglo de datos varias veces en forma secuencial. Cada paso consiste en la comparación de cada elemento en el arreglo con su sucesor (x[i] con x[i+1]) y el intercambio de los dos elementos si no están en el orden correcto. Considérese el siguiente arreglo de datos:

25 57 48 37 12 92 86 33

En el primer paso se realizan las siguientes operaciones:

x[0] con x[1] (25 con 57) no intercambio.
x[1] con x[2] (57 con 48) intercambio.
x[2] con x[3] (57 con 32) intercambio.
x[3] con x[4] (57 con 12) intercambio.
x[4] con x[5] (57 con 92) no intercambio.
x[5] con x[6] (92 con 86) intercambio.
x[6] con x[7] (92 con 33) intercambio.

Así después del primer paso, el arreglo está en el siguiente orden:

25 48 37 12 57 86 33 92

Obsérvese que después del primer paso, el elemento mayor (en este caso 92) está en la posición correcta dentro del arreglo. En general x[n-i] estará en su posición correcta después de la iteración i. El método se lama ordenamiento de burbuja porque cada número "burbujea" con lentitud hacia su posición correcta. El conjunto completo de iteraciones es:

iteración 0 : 25 57 48 37 12 92 86 33
iteración 1: 25 48 37 12 57 86 33 92
iteración 2: 25 37 12 48 57 33 86 92
iteración 3: 25 12 37 48 33 57 86 92
iteración 4: 12 25 37 33 48 57 89 92
iteración 5: 12 25 33 37 48 57 89 92

La implementación de este algoritmo en Java queda como:

```
 void Burbuja(int x[], int n)
{
  int b, j, t;
  do
  {
    b = 0;
    for(j=0; j<n; j++)
    {
       if(x[j] > x[j+1])
       {
         t = x[j];
         x[j] = x[j+1];
         x[j+1] = t;
         b++;
       }
    }
    n--;
  }
  while(b > 0);
}
```

Ver ejemplo de ejecución en : ver

Regresar.

Funciones y la estructura del programa.

Habíamos comentado en la pequeña introducción a la Programación Orientada a Objetos que cada clase podría tener una serie de "**atributos**" (o "**variables miembro**"), que son sus características, y varios "**métodos**" (o "**funciones miembro**"), que son aquellas cosas que es capaz de hacer.

Pero hasta ahora no lo habíamos aplicado: las clases que hemos creado tenían un único método (el llamado "main", que representaba el cuerpo de la aplicación) y no tenían atributos. Ahora vamos a crear un ejemplo de clase que tendrá un par de atributos y varios métodos, unos de los cuales devolverán valores y otros no:

```
// RectanguloTexto.java
// Primer ejemplo de clase con varios
//   métodos y atributos
// Introducción a Java

class RectanguloTexto {


    // --- Atributos:
    // --- la base y la altura del rectángulo

    int altura;
    int base;


    // --- Método "fijaAltura":
    // --- Da valor al atributo "altura"
    // --- No devuelve ningún valor

    public void fijaAltura( int valor ) {
       altura = valor;
    }


    // --- Método "fijaBase":
    // --- Da valor al atributo "base"
    // --- No devuelve ningún valor

    public void fijaBase( int valor ) {
       base = valor;
    }


    // --- Método "leeAltura":
    // --- Devuelve el valor del atributo "altura",
    // --- para no acceder directamente a él
```

```
public int leeAltura( ) {
   return altura;
}


// --- Método "leeBase":
// --- Devuelve el valor del atributo "base",
// --- para no acceder directamente a él

public int leeBase( ) {
   return base;
}


// --- Método "leeArea":
// --- Devuelve el area del cuadrado
// --- para no acceder directamente a él

public int leeArea( ) {
   return altura * base;
}


// --- Método "dibuja":
// --- Dibuja un rectangulo en pantalla
// --- No devuelve ningún valor

public void dibuja(  ) {

   int i, j;             // Para bucles
   String lineaActual;   // Cada linea horizontal
                         // rectangulo que dibujamos

   for (i=0; i<altura; i++) {  // Cada fila horiz
     // Primero la fila esta vacia
     lineaActual = "";
     // Luego añado cuadraditos
     for (j=0; j<base; j++)
       lineaActual = lineaActual + "#";
     // Y finalmente escribo la líea
     System.out.println( lineaActual );
   }
}


// --- Método "main":
// --- Cuerpo de la aplicación,
// --- Dibuja un par de rectangulos de ejemplo

public static void main( String args[] ) {

   RectanguloTexto r = new RectanguloTexto();
```

```
            r.fijaAltura( 10 );
            r.fijaBase( 20 );
            System.out.println( "El área del rectángulo es" );
            System.out.println( r.leeArea() );
            System.out.println( "Vamos a dibujarlo..." );
            r.dibuja();
        }


    }
```

Comentemos cosas:

- Nuestra clase rectángulo tiene dos **atributos**: la base y la altura (ambos son números enteros).
- Tiene varios **métodos**. Los más "comprensibles" son "dibuja", que lo muestra el rectángulo en pantalla (y no devuelve ningún valor, por eso lo definimos como "void"), y "leeArea", que devuelve un número entero ("int"), que es el área del rectángulo.
- Pero también hemos incluido métodos leeAltura, leeBase, fijaAltura y fijaBase. Esto es para lograr otra de las características deseables de la Programación Orientada a Objetos, que ya habíamos comentado: la **ocultación** de los detalles. Así, un cualquier aplicación que emplease nuestra clase RectanguloTexto no necesita saber cómo son internamente los atributos "altura" y "base" para acceder a ellos. Si más adelante, nosotros decidimos optimizar al máximo el espacio y utilizar atributos de tipo "byte" (por ejemplo), bastaría con modificar ligeramente estos 4 métodos para que conviertan de "byte" a "int" y viceversa, pero todas las aplicaciones que se basen en esta clase seguirían funcionando sin cambios. Este tipo de detalles hacen que un programa sea más "robusto".

La salida de este programa sería esta:

```
El área del rectángulo es
200
Vamos a dibujarlo...
####################
####################
####################
####################
####################
####################
####################
####################
####################
####################
```

Por lo demás, este programa no debería presentar mayor dificultad. Vemos que los métodos se definen de la forma que ya habíamos empleado para "main", salvo que en un caso

general, puede ocurrir que "devuelvan" un valor de un cierto tipo, distinto de "void" (en nuestro caso, tenemos varios que devuelven números enteros -int-).

Para indicar qué valor queremos devolver, se emplea la sentencia "**return**", como se ve en el ejemplo.

Dentro de un cierto método podemos tener **variables temporales**. Es algo que ya hemos usado más de una vez en "main", con variables como las típicas "i" y "j" que empleábamos para contar en los bucles. Estas variables auxiliares sólo existen dentro del método en el que las hemos definido. Por ejemplo, podríamos haber creado una variable auxiliar para calcular el área:

```
public int leeArea( ) {
   int valorArea;

   valorArea =  altura * base;
   return valorArea;
}
```

En este caso, hemos definido una variable llamada "valorArea". Le damos el valor que nos interesa y devolvemos ese valor. Pero no tiene sentido mencionar la variable "valorArea" desde ninguna otra parte de nuestro programa, porque desde ningún otro sitio se conoce esta variable, y obtendríamos un mensaje de error.

Esto es lo que se conoce como una "**variable local**" (o una "variable de **ámbito local**").

También podemos indicar **parámetros** (datos adicionales) para nuestros métodos. Es algo que también llevamos haciendo desde el principio, casi sin darnos cuenta. Por ejemplo, siempre hemos indicado "main" con un extraño parámetro llamado "args[]", de tipo String, y que todavía no hemos utilizado. De igual modo, cada vez que utilizamos "println", indicamos entre comillas el texto que queremos que aparezca en pantalla (y que también es un parámetro).

En este ejemplo vamos teniendo parámetros más fáciles de comprender, como en el caso de:

```
public void fijaAltura( int valor ) {
   altura = valor;
}
```

Este método "fijaAltura" tiene un parámetro llamado "valor", que es un número entero (int), y que es el valor que queremos que tome la altura.

Podemos indicar **más de un parámetro** a un método. En ese caso, los separaremos empleando comas. Por ejemplo, nos podría interesar dar valores a la vez a la base y a la altura, creando un método como éste:

```
    public void fijaAlturaYBase( int valorAlt, int valor Anc ) {
       altura = valorAlt;
       base = valorAnc;
    }
```

Todo esto es la base sobre cómo se usan las "funciones" en Java (que en este lenguaje llamamos "funciones miembro" o "métodos", siguiendo la nomenclatura habitual.en Programación Orientada a Objetos). Cada vez las iremos empleando con más frecuencia, y eso será lo que ayude a terminar de comprender su empleo en la práctica... espero...

Programación modular y Java.

Lo habitual en una aplicación medianamente seria es que no tengamos una sola "clase", como hasta ahora, sino que realmente existan varios objetos de distintas clases, que se relacionan entre sí.

En Java podemos definir **varias clases** dentro de un mismo fichero, con la única condición de que sólo una de esas clases sea declarada como "pública". En un caso general, lo más correcto será definir **cada clase en un fichero**. Aun así, vamos a ver primero un ejemplo que contenga dos clases en un solo fichero

```
// DosClases.java
// Primer ejemplo de una clase nuestra
//   que accede a otra también nuestra,
//   ambas definidas en el mismo fichero
// Introducción a Java

class Principal {


    public static void main( String args[] ) {

       Secundaria s = new Secundaria();

       s.saluda();    // Saludo de "Secundaria"
       saluda();      // Saludo de "Principal"
    }


    public static void saluda() {
       System.out.println( "Saludando desde <Principal>" );
    }


}


// ----------------------------------------------------


class Secundaria {
```

```
    public void saluda() {
       System.out.println( "Saludando desde <Secundaria>" );
    }


}
```

Como siempre, hay cosas que comentar:

- En este fuente hay **dos clases**, una llamada "Principal" y otra llamada "Secundaria".
- La clase "Secundaria" sólo tiene **un método**, llamado "saluda", mientras que la clase "Principal" tiene **dos métodos**: "main" (el cuerpo de la aplicación) y otro llamado "saluda", al igual que el de "Secundaria".
- Ambos métodos "s**aluda**" se limitan a mostrar un mensaje en pantalla, que es distinto en cada caso.
- En el método "**main**", definimos y creamos un objeto de la clase "Secundaria", después llamamos al método "saluda" de dicho objeto (con al expresión "s.saluda()") y luego llamamos al método "Saluda" de la propia clase "Principal" (escribiendo solamente "saluda()")..

Para **compilar** este programa teclearíamos, como siempre:

```
javac DosClases.java
```

y entonces se crearían **dos ficheros** llamados

```
Principal.class
Secundaria.class
```

que podríamos **probar** tecleando

```
java Principal
```

El **resultado** se mostraría en pantalla es:

```
Saludando desde <Principal>
Saludando desde <Secundaria>
```

Ahora vamos a ver un ejemplo en el que las dos clases están en **dos ficheros distintos**. Tendremos una clase "sumador" que sea capaz de sumar dos números (no es gran cosa, sabemos hacerlo sin necesidad de crear "clases a propósito, pero nos servirá como ejemplo) y un programa principal que la utilice.

La clase "Sumador", con un único método "calcularSuma", que acepte dos números enteros y devuelva otro número entero, sería:

```
// Sumador.java
// Segundo ejemplo de una clase nuestra
//   que accede a otra también nuestra.
//   Esta es la clase auxiliar, llamada
//   desde "UsaSumador.java"
// Introducción a Java

class Sumador {

    public int calcularSuma( int a, int b ) {
       return a+b;
    }


}
```

Por otra parte, la clase "UsaSumador" emplearía un objeto de la clase "Sumador", llamado "suma", desde su método "main", así:

```
// UsaSumador.java
// Segundo ejemplo de una clase nuestra
//   que accede a otra también nuestra.
//   Esta es la clase principal, que
//   accede a "Sumador.java"
// Introducción a Java

class UsaSumador {


    public static void main( String args[] ) {

       Sumador suma = new Sumador();

       System.out.println( "La suma de 30 y 55 es" );
       System.out.println( suma.calcularSuma (30,55) );
    }


}
```

Para **compilar** estos dos fuentes, si tecleamos directamente

```
javac usaSumador.java
```

recibiríamos como respuesta un mensaje de error que nos diría que no existe la clase Sumador:

```
UsaSumador.java:13: Class Sumador not found.
                Sumador suma = new Sumador();
                ^
UsaSumador.java:13: Class Sumador not found.
                Sumador suma = new Sumador();
                               ^
2 errors
```

La **forma correcta** sería compilar primero "Sumador" y después "UsaSumador", para después ya poder probar el resultado:

```
javac Sumador.java
javac UsaSumador.java
java UsaSumador
```

La respuesta, como es de esperar, sería:

```
La suma de 30 y 55 es
85
```

Ejemplo.

Hacer la implementación de la función seno utilizando serie de Taylor.

La función seno puede ser calculada utilizando la serie

$seno(x) = x - x^3/3! + x^5/5! - x^7/7! + x^9/9! - ....$

Para llevar a cabo la implementación necesitamos de dos funciones mas, la función factorial y la función potencia.

Comenzaremos por escribir la función factorial. El factorial de un número se define como

```
1! = 1
2! = 1*2
3! = 1*2*3
4! = 1*2*3*4

n! = 1*2*3*4*5*…*n-1*n
```

y la implementación en Java queda como

```
  double factorial(int x)
  {
    int i;
    double fact = 1;

    for(i=1; i<=x; i++)
      fact *= i;

    return fact;
  }
```

Tenemos que elevar un número real a una potencia entera, así pues, si x es el número real para elevarlo a cualquier potencia hacemos :

$x^2 = x*x$
$x^3 = x*x*x$

$x^n = x*x*x* \ldots *x$

La implementación en Java de esta función es:

```
  double pow(double x, int n)
  {
    int i;
    double pow =1;

    if(x==0) return 0;

    for(i=1; i<=n; i++)
      pow = pow*x;

    return pow;
  }
```

Finalmente para llevar a cabo la función seno, definimos una clase a la cual llamamos funciones. En esta clase se encuentra definida la función seno, pow y factorial, como miembros estáticos.

```
class funciones
{

  public static double seno(double x)
  {
    int i;
    double s = 0;
    int signo = 1;

    for(i=1; i<10; i+=2)
    {
      s += signo*pow(x, i) / factorial(i);
      signo *= -1;
    }

    return s;

  }

  public static double factorial(int x)
  {
    int i;
    double fact = 1;

    for(i=1; i<=x; i++)
      fact *= i;

    return fact;
  }
```

```
  public static double pow(double x, int n)
  {
    int i;
    double pow =1;

    if(x==0) return 0;

    for(i=1; i<=n; i++)
      pow = pow*x;

    return pow;
  }

}
```

Para hacer uso de esta clase escribimos el siguiente programa.

```
class prueba
{
  static public void main(String args[])
  {

    int i,N;

    double inicio = -3.1416, fin = 3.1416, incremento = 0.1;

    N = (int)((fin - inicio)/incremento) + 1;

    double x[] = new double [N];
    double y[] = new double [N];

    for(i=0;  i<N; i++)
    {
      x[i] = inicio + incremento *i;
      y[i] = funciones.seno(x[i]);
    }

    grafica g = new grafica("Funcion seno");
    g.SerieX(x);
    g.SerieY(y);
    g.show();
  }

}
```

Note, en la clase prueba se están utilizando dos clases, la clase funciones y la clase gráfica. En la primera no se hace declaración de la clase, simplemente se utiliza ya que sus funciones fueron declarados como miembros estáticos. En la segunda no se da esta situación y se hace necesario hacer la declaración para su uso.

Regresar.

Recursividad

Un método recursivo es un método que se llama a si mismo directa o indirectamente o a través de otro método. Un ejemplo interesante de recursion es la función factorial. La definición de factorial esta dada por el producto:

n! = n*(n-1)*(n-2)*…*3*2*1

Note que esta definición la podemos escribir de manera recursiva como:

n! = n*(n-1)!

Este método lo podemos escribir en JAVA como:

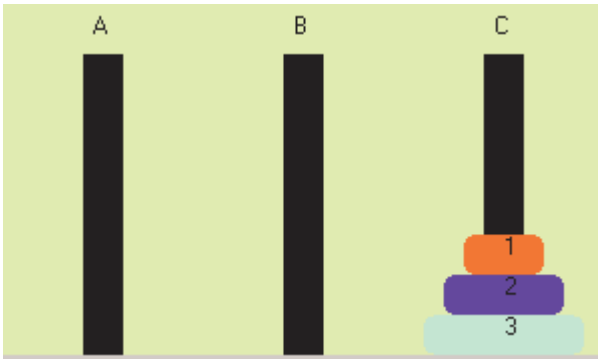
```
class factorial
{
  static public void main(String args[])
  {
    for(int i=0; i<10; i++)
      System.out.println(i+"! = " + factorial (i));
  }

  static public double factorial(int N)
  {
     if(N == 0) return 1;
     else return (N*factorial(N-1));
  }
}
```

Otro ejemplo de definición recursiva es la multiplicación de números naturales. El producto a*b, donde a y b son enteros positivos, puede definirse

```
class multiplica
{
  static public void main(String args[])
  {
    int x=4, y=5;
    System.out.println(multiplica(4,5));
  }

  static public double multiplica(int x, int y)
  {
     if(y == 1) return x;
     else return (x + multiplica(x,y-1));
  }
}
```

El ejemplo más conocido, de recursividad entre los programadores, es el problema de las torres de Hanoi. Para este juego, se tienen tres postes (A, B, C) y un conjunto n de discos colocados de mayor a menor diámetro en alguno de los postes, por ejemplo el poste A. La

meta es mover todos los discos del poste A al poste C con las siguientes restricciones: en cada movimiento solo se puede tomar un disco y colocarlo en cualquier poste y no se debe colocar, un en un poste, un disco de diámetro mayor sobre un disco de diámetro menor (ejemplo).



La implementación en Java queda de la siguiente manera

```
public class ej051 {

  public static void main(String[] args)
  {
     mover("A", "C", "B", 3);
  }

  public static void mover(String a, String b, String c, int n)
  {
    if (n > 0) {
      mover(a, c, b, n - 1);
      System.out.println("Mover un disco del poste " + a + " al poste " +
c);

      mover(b, a, c, n - 1);
    }
  }

}
```

La solución con tres discos es:

```
Mover un disco del poste A al poste B
Mover un disco del poste A al poste C
Mover un disco del poste B al poste C
Mover un disco del poste A al poste B
Mover un disco del poste C al poste A
Mover un disco del poste C al poste B
Mover un disco del poste A al poste B
```

Regresar.

Método de Biseccciones.

Este método es conocido también como de corte binario. de partición en dos intervalos iguales o método de Bolzano. Es un método de búsqueda incremental donde el intervalo de búsqueda se divide en dos. Si la función cambia de signo sobre un intervalo, se calcula el valor en el punto medio. El nuevo intervalo de búsqueda será aquel donde el producto de la función cambie de signo.

Ejemplo.

Considere la función $f(x) = x\text{-}cosx$, a priori sabemos que la función tiene un cruce por cero en el intervalo [0,1], así que nuestra búsqueda se concentrará en este.

iter	inicio	mitad	fin	f(ini)	f(mitad)	f(fin)
0	0.0	0.5	1.0	-1.0	-0.3775	0.4596
1	0.5	0.75	1.0	-0.3775	0.0183	0.4596
2	0.5	0.625	0.75	-0.3775	-0.1859	0.0183
3	0.625	0.6875	0.75	-0.1859	-0.0853	0.0183
4	0.6875	0.71875	0.75	-0.0853	-0.0338	0.0183
5	0.71875	0.734375	0.75	-0.0338	-0.0078	0.0183
6	0.734375	0.7421875	0.75	-0.0078	0.0051	0.0183
7	0.734375	0.73828125	0.7421875	-0.0078	-0.0013	0.0051
8	0.73828125	0.740234375	0.7421875	-0.0013	0.0019	0.0051
9	0.73828125	0.7392578125	0.740234375	-0.0013	0.0002	0.0019

La implementación recursiva de este algoritmo es:

```
public static double Biseccion(double ini, double fin)
{
  double mitad;    mitad = (fin + ini)/2.0;
  if((fin - ini) > 0.001)
  {
    if(funcion(ini)*funcion(mitad) < 0)
      return Biseccion(ini, mitad);
    else return Biseccion(mitad, fin);
  }
  else return (mitad);
}

public static double funcion(double x)
{
  return (x - Math.cos(x));
}
```

Regresar.

Algoritmo de Ordenamiento Quicksort.

Sea x un arreglo y n el número de elementos en arreglo que se debe ordenar. Elegir un elemento a de una posición especifica en el arreglo (por ejemplo, a puede elegirse como el primer elemento del arreglo. Suponer que los elemento de x están separados de manera que a está colocado en la posición j y se cumplen las siguientes condiciones.

1.- Cada uno de los elementos en las posiciones de 0 a j-1 es menor o igual que a.
2.- Cada uno de los elementos en las posiciones j+1 a n-1 es mayor o igual que a.

Observe que si se cumplen esas dos condiciones para una a y j particulares, a es el j-ésimo menor elemento de x, de manera que a se mantiene en su posición j cuando el arreglo está ordenado en su totalidad. Si se repite este procedimiento con los subarreglos que van de x[0] a x[j-1] y de x[j+1] a x[n-1] y con todos los subarreglos creados mediante este proceso, el resultado final será un archivo ordenado.

Ilustremos el quicksort con un ejemplo. Si un arreglo esta dado por:

x = [25 57 48 37 12 92 86 33]

y el primer elemento se coloca en su posición correcta, el arreglo resultante es:

x = [12 25 57 48 37 92 86 33]

En este punto 25 esta en su posición correcta por lo cual podemos dividir el arreglo en

x = [12] 25 [57 48 37 92 86 33]

Ahora repetimos el procedimiento con los dos subarreglos

x = 12 25 [48 37 33] 57 [92 86]

x = 12 25 33 [37 48] 57 [86] [92]

x = 12 25 33 [37 48] 57 86 92

x = 12 25 33 37 48 57 86 92

El procedimiento es entonces.

- Buscar la partición del arreglo j.
- Ordenar el subarreglo x[0] a x[j-1]
- Ordenar el subarreglo x[j+1] a x[n-1]

Su implementación en Java es:

```
public void quiksort(int x[],int lo,int ho)
{
  int t, l=lo, h=ho, mid;

  if(ho>lo)
  {
    mid=x[(lo+ho)/2];
    while(l<=h)
    {
      while((l<ho)&&(x[l]<mid))  ++l;
      while((h>lo)&&(x[h]>mid))  --h;
      if(l<=h)
      {
        t    = x[l];
        x[l] = x[h];
        x[h] = t;
        ++l;
        --h;
      }
    }

    if(lo<h) quiksort(x,lo,h);
    if(l<ho) quiksort(x,l,ho);
  }
}
```

Para ver este programa corriendo haga clic aquí.

Regresar.

Programación Orientada a Objetos.

Clase Complejo.

A continuación se presenta un ejemplo de la implementación de la clase complejo.

Para ver un ejemplo de ejecución de esta, presionar aquí.

```
/**
 * <p>Title: Caladora para numeros complejos</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: UMSNH</p>
 * @author Felix Calderon Solorio y sus alumons
 * @version 1.0
 */

public class complejo
{
  double real, imag;
  boolean espolar;

  complejo(double r, double x, boolean Esp)
  {
    real = r;
    imag = x;
    espolar = Esp;
  }

  complejo()
  {
     real = 0;
     imag = 0;
     espolar = false;
  }


  public void suma(complejo a, complejo b)
  {
    if(a.espolar) a.rectangular();
    if(b.espolar) b.rectangular();
    this.real = a.real + b.real;
    this.imag = a.imag + b.imag;
    this.espolar = false;
  }

  public void resta(complejo a, complejo b)
  {
    if(a.espolar) a.rectangular();
    if(b.espolar) b.rectangular();
    this.real = a.real - b.real;
```

```
    this.imag = a.imag - b.imag;
    this.espolar = false;
  }

  void rectangular()
  {
    double r,x;
    r = this.real;
    x = this.imag;

    this.real = r*Math.cos(x);
    this.imag = r*Math.sin(x);
    this.espolar = false;
  }


  public void multiplica(complejo a, complejo b)
  {
    if(!a.espolar)a.polar();
    if(!b.espolar)b.polar();
    this.real = a.real * b.real;
    this.imag = a.imag + b.imag;
    this.espolar = true;
  }

  public void division(complejo a, complejo b)
  {
    if(!a.espolar)a.polar();
    if(!b.espolar)b.polar();
    this.real = a.real / b.real;
    this.imag = a.imag - b.imag;
    this.espolar = true;
  }

  String Real()
  {
    String datos="";
    datos += this.real;
    return datos;
  }

  String Imag()
  {
    String datos="";
    datos += this.imag;
    return datos;
  }
  void polar()
  {
    double mag, arg;
    mag = Math.sqrt(this.real*this.real + this.imag*this.imag);
    arg = Math.atan2(this.imag,this.real);
    this.real = mag;
    this.imag = arg;
    this.espolar = true;
  }
}
```

Regresar.

La Clase Matriz

Introducción.

Una matriz se define como un arreglo bidimensional de datos, que tiene n renglones y m columnas. Un elemento del arreglo puede ser identificado con a_{ij}

$$A = \begin{vmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ & & & & \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mm} \end{vmatrix}$$

Algunas de las operaciones básicas que pueden realizarse con matrices son suma, resta y multiplicación. La división de matrices como tal no existe y en su lugar se calcula la inversa.

Suma de matrices.

Para que la sumar las matrices A y B, se requiere que las matrices tengan el mismo número de renglones y de columnas. Si queremos encontrar la suma C = A + B, cada elemento de la matriz C lo calculamos de la siguiente forma:

$$c_{ij} = a_{ij} + b_{ij}$$

para todos lo i,j en la matriz C

Resta de matrices.

En este caso, se deben cumplir las mismas propiedades que la resta de matrices y el calculo de los elemento de la matriz C se calculan como:

$$c_{ij} = a_{ij} - b_{ij}$$

para todos lo i,j en la matriz C

Multiplicación de matrices.

Para realizar el producto C = A*B tenemos que considerar que el producto existe si

1.- El número de columnas de A es igual al número de renglones de B.

C(n,l) = A(n,m)*B(m,l)

2.- Las dimensiones de la matriz resultante tendrá el mismo numero de renglones que la matriz A y el mismo número de columnas que la matriz B.

3.- El cálculo de los elementos de la matriz C se lleva a cabo haciendo :

$$c_{ij} = \Sigma_{k=1..m} a_{ik} * b_{k,j}$$

para todos lo i,j en la matriz C

```
public class Matriz
{
  int nren, ncol;
  double datos[][];


/**********************************************************************
   * crea una matriz nula

**********************************************************************/

  public Matriz()
  {
    nren = 0;
    ncol = 0;
  }


/**********************************************************************
   *  crea una matriz con n renglones y m columnas de puros ceros

**********************************************************************/

  public Matriz(int n, int m)
  {
    inicializa(n, m);
  }


/**********************************************************************
   * Inicializa las variables para los constructores.

**********************************************************************/

  private void inicializa(int r, int c)
  {
      // Si la matriz es nula reserva memoria.
      if((nren == 0 && ncol ==0) || this == null)
      {
        nren = r;
        ncol = c;

        datos = new double[nren][ncol];
        for(int i=0; i<r; i++)
          for(int j=0; j<c; j++)
            datos[i][j] = 0;
```

```
      }
  }

/*********************************************************************
   * Inicializa la matriz con un arreglo bidimensional

*********************************************************************/

  public Matriz(int n, int m, double d[][])
  {
    // crea una matriz con n renglones y m columnas de puros ceros

    nren = n;
    ncol = m;

    datos = new double[nren][ncol];
    for(int i=0; i<nren; i++)
      for(int j=0; j<ncol; j++)
        datos[i][j] = d[i][j];
  }

  /*********************************************************************
   *    imprime el contenido en una matriz
   ********************************************************************/

  public void imprime()
  {
    int i, j;
    if( (nren == 0) && (ncol == 0) )
    {
      System.out.println("No tiene informacion la matriz");
      return;
    }

    for(i=0; i<this.nren; i++)
    {
      for(j=0; j<this.ncol; j++)
        System.out.print(datos[i][j] + " ");
      System.out.println(" ");
    }
  }

  /*****************************************************
    Algoritmo para suma de matrices
  *****************************************************/

  public Matriz suma(Matriz a, Matriz b)
  {
    if ( (a.nren == b.nren) && (a.ncol == b.ncol)) {

      this.inicializa(a.nren, a.ncol);

      if (this.nren == a.nren && this.ncol == a.ncol) {

        for (int i = 0; i < a.nren; i++)
          for (int j = 0; j < a.ncol; j++)
```

```
            this.datos[i][j] = a.datos[i][j] + b.datos[i][j];

        return this;
      }
      else
        System.out.println("Matrices con diferente tamaño");
    }
    else
      System.out.println("Matrices con diferente tamaño");
    return null;
  }
  /*******************************************************
    Algoritmo para resta de matrices
  *******************************************************/

public Matriz resta(Matriz a, Matriz b)
{
  if ( (a.nren == b.nren) && (a.ncol == b.ncol)) {

    this.inicializa(a.nren, a.ncol);

    if (this.nren == a.nren && this.ncol == a.ncol) {

      for (int i = 0; i < a.nren; i++)
        for (int j = 0; j < a.ncol; j++)
          this.datos[i][j] = a.datos[i][j] - b.datos[i][j];

      return this;
    }
    else
      System.out.println("Matrices con diferente tamaño");
  }
  else
    System.out.println("Matrices con diferente tamaño");
  return null;
}

  /*******************************************************
    Algoritmo para multiplicacion de matrices
  *******************************************************/

public Matriz multiplica(Matriz a, Matriz b)
{
  int i, j;

  double suma;

  if(a.ncol == b.nren)
  {
    inicializa(a.nren, b.ncol);
    if(this.nren == a.nren && this.ncol == b.ncol)
    {
      for(i=0; i< this.nren; i++)
        for(j=0; j< this.ncol; j++)
        {
          suma = 0;
          for(int k=0; k<a.ncol; k++)
```

```
            suma += a.datos[i][k]*b.datos[k][j];
          this.datos[i][j] = suma;

        }
      return this;
    }
    else System.out.println("Matrices con diferente tamaño");
  }
  else System.out.println("Matrices con diferente tamaño");
  return null;
}

 /*******************************************************
   Algoritmo para encontrar la transpuesta de un matriz
 *******************************************************/

 public Matriz transpuesta(Matriz a)
 {
   inicializa(a.ncol,a.nren);

   for(int i=0;i<nren;i++)
     for(int j=0;j<ncol;j++)
       this.datos[i][j]=a.datos[j][i];
   return this;
  }
}
```

Regresar.

Herencia

Si se supone que somos buenos programando, cuando creemos una clase es posible que sea algo útil. De modo que cuando estemos haciendo un programa distinto y necesitemos esa clase podremos incluirla en el código de ese nuevo programa. Es la manera más sencilla de reutilizar una clase.

También es posible que utilicemos esa clase incluyendo instancias de la misma en nuevas clases. A eso se le llama composición. Representa una relación "tiene un". Es decir, si tenemos una clase Rueda y una clase Coche, es de esperar que la clase Coche tenga cuatro instancias de Rueda:

```
class Coche {
  Rueda rueda1, rueda2, rueda3, rueda 4;
  ...
}
```

Sin embargo, en ocasiones, necesitamos una relación entre clases algo más estrecha. Una relación del tipo "es un". Por ejemplo, sabemos bien que un gato es un mamífero. Sin embargo es también un concepto más específico, lo que significa que una clase Gato puede compartir con Mamífero propiedades y métodos, pero también puede tener algunas propias.

Herencia.java

```
class Mamifero {
  String especie, color;
}

class Gato extends Mamifero {
  int numero_patas;
}

public class Herencia {
  public static void main(String[] args) {
    Gato bisho;
    bisho = new Gato();
    bisho.numero_patas = 4;
    bisho.color = "Negro";
    System.out.println(bisho.color);
  }
}
```

Como vemos en el ejemplo, el objeto bisho no sólo tiene la propiedad numero_patas, también color que es una propiedad de Mamifero. Se dice que Mamifero es la clase padre y Gato la clase hija en una relación de herencia. Esta relación se consigue en Java por medio de la palabra reservada extends.

Pero, además de heredad la funcionalidad de la clase padre, una clase hija puede sobreescribirla. Podemos escribir un método en la clase hija que tenga el mismo nombre y los mismos parámetros que un método de la clase padre:

Herencia.java
```
class Mamifero {
  String especie, color;
  public void mover() {
    System.out.println("El mamífero se mueve");
  }
}

class Gato extends Mamifero {
  int numero_patas;
  public void mover() {
    System.out.println("El gato es el que se mueve");
  }
}

public class Herencia {
  public static void main(String[] args) {
    Gato bisho = new Gato();
    bisho.mover();
  }
}
```

Al ejecutar esta nueva versión veremos que se escribe el mensaje de la clase hija, no el del padre.

Conviene indicar que Java es una lenguaje en el que todas las clases son heredadas, aún cuando no se indique explícitamente. Hay una jerarquía de objetos única, lo que significa que existe una clase de la cual son hijas todas las demás. Este Adán se llama Object y, cuando no indicamos que nuestras clases hereden de nadie, heredan de él. Esto permite que todas las clases tengan algunas cosas en común, lo que permite que funcione, entre otras cosas, el recolector de basura.

Clases y métodos abstractos

Como vimos anteriormente, es posible que con la herencia terminemos creando una familia de clases con un interfaz común. En esos casos es posible, y hasta probable, que la clase raíz de las demás no sea una clase útil, y que hasta deseemos que el usuario nunca haga instancias de ella, porque su utilidad es inexistente. No queremos implementar sus métodos, sólo declararlos para crear una interfaz común. Entonces declaramos sus métodos como abstractos:

public abstract void mi_metodo();
Como vemos, estamos declarando el método pero no implementandolo, ya que sustituimos el código que debería ir entre llaves por un punto y coma. Cuando existe un método abstracto deberemos declarar la clase abstracta o el compilador nos dará un error. Al declarar como abstracta una clase nos aseguramos de que el usuario no pueda crear instancias de ella:

Abstractos.java

```
abstract class Mamifero {
  String especie, color;
  public abstract void mover();
}

class Gato extends Mamifero {
  int numero_patas;
  public void mover() {
    System.out.println("El gato es el que se mueve");
  }
}

public class Abstractos {
  public static void main(String[] args) {
    Gato bisho = new Gato();
    bisho.mover();
  }
}
```

En nuestro ejemplo de herencia, parece absurdo pensar que vayamos a crear instancias de Mamifero, sino de alguna de sus clases derivadas. Por eso decidimos declararlo abstracto.

Ejemplo de la Clase punto.

Consideremos que deseamos hacer una clase punto en dos dimensiones y después hacer la representación de la misma en tres dimensiones. La clase punto tendrá un constructor y calculará la distancia entre dos puntos.

```
public class punto {
  double x, y;

  public punto() {
    x = 0;
    y = 0;
  }

  public punto(double a, double b) {
    x = a;
    y = b;
  }
```

```
  public void imprime(String a)
  {
    System.out.println(a +"(" + x + ", " + y +")");
  }

  public double distancia(punto a, punto b)
  {
    double dx, dy;

    dx = a.x - b.x;
    dy = a.y - b.y;

    return Math.sqrt(dx*dx + dy*dy);
  }

}
```

La extensión de la clase punto a clase punto en tres dimensiones, utilizando herencia queda como:

```
public class punto3d extends punto
{
  double z;

  public punto3d()
  {
    super();
    z = 0;
  }

  public punto3d(double a, double b, double c)
  {
    super(a,b);
    z = c;
  }

  public void imprime(String a)
  {
    System.out.println(a +"(" + x + ", " + y + ", " + z +")");
  }
  public double distancia(punto3d a, punto3d b)
  {
    double dx = super.distancia(a, b);
    double dz = a.z - b.z;

    return Math.sqrt(dx*dx + dz*dz);
  }

}
```

Interfaces

Los interfaces tienen como misión en esta vida llevar el concepto de clase abstracta un poco más lejos, amén de permitirnos algo parecido a la herencia múltiple. Pero vamos pasito a pasito. Un interfaz es como una clase abstracta pero no permite que ninguno de sus métodos esté implementado. Es como una clase abstracta pero en estado más puro y cristalino. Se declaran sustituyendo class por interface:

```
interface Mamifero {
  String especie, color;
  public void mover();
}

class Gato implements Mamifero {
  int numero_patas;
  public void mover() {
    System.out.println("El gato es el que se mueve");
  }
}
```

No tenemos que poner ningún abstract en ningún sitio porque ya se le supone. Hay que fijarse también que ahora Gato no utiliza extends para hacer la herencia, sino implements.

Sin embargo, la mayor utilidad de los interfaces consiste en permitir la existencia de herencia múltiple, que consiste en que una clase sea heredera de más de una clase (que tenga varios papás, vamos). En C++ existía pero daba enormes problemas, al poder estar implementado un mismo método de distinta forma en cada una de las clases padre. En Java no existe ese problema. Sólo podemos heredar de una clase, pero podemos a su vez heredar de uno o varios interfaces (que no tienen implementación). Modifiquemos nuestro adorado ejemplo gatuno:

Interfaces.java

```
interface PuedeMoverse {
  public void mover();
}

interface PuedeNadar {
  public void nadar();
}

class Mamifero {
  String especie, color;
  public void mover() {
    System.out.println("El mamífero se mueve");
  }
}

class Gato extends Mamifero
        implements PuedeMoverse, PuedeNadar {
  int numero_patas;
```

```
  public void mover() {
    System.out.println("El gato es el que se mueve");
  }
  public void nadar() {
    System.out.println("El gato nada");
  }
}

public class Interfaces {
  public static void main(String[] args) {
    Gato bisho = new Gato();
    bisho.mover();
    bisho.nadar();
  }
}
```

Vemos que Gato tiene la obligación de implementar los métodos mover() y nadar(), ya que sino lo hace provocará un error de compilación. Podría no implementar mover(), ya que hereda su implementación de Mamifero. Pero si decidiéramos no hacerlo no habría problemas, ya que tomaría la implementación de su clase padre, ya que los interfaces no tienen implementación. Así nos quitamos los problemas que traía la herencia múltiple de C++.

Regresar.

Clase complejo.

```
public class complejo
{
  double real, imag;

  complejo(double r, double x)
  {
    real = r;
    imag = x;
  }

  complejo()
  {
     real = 0;
     imag = 0;
  }

  public void suma(complejo a, complejo b)
  {
    this.real = a.real + b.real;
    this.imag = a.imag + b.imag;
  }

  public void resta(complejo a, complejo b)
  {
    this.real = a.real - b.real;
    this.imag = a.imag - b.imag;
  }

  public void multiplica(complejo a, complejo b)
  {
    this.real = a.real * b.real - a.imag * b.imag;
    this.imag = a.real * b.imag + a.imag * b.real;
  }

  public void division(complejo a, complejo b)
  {
    double den = b.real * b.real + b.imag * b.imag;

    this.real = ( a.real * b.real + a.imag * b.imag)/den;
    this.imag = (-a.real * b.imag + a.imag * b.real)/den;
  }

  void imprime()
  {
    if(this.imag >= 0)
      System.out.print("(" + this.real + " + j" + this.imag + ") ");
      else
        System.out.print("(" +this.real + " - j" + this.imag*-1+ ") ");
  }

  void igual(complejo a)
```

```
  {
    this.real = a.real;
    this.imag = a.imag;
  }
}
```

Clase Matriz Complejo.

Dada la clase complejo podemos escribir la clase matriz complejo como:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.util.*;

public class MatrizC extends complejo
{
  int nren, ncol;
  complejo datos[][];

  //
*********************************************************************

  MatrizC(double Re[][], double Im[][])
  {
    nren = Re.length;
    ncol = Re[0].length;

    datos = new complejo[nren][ncol];

    for (int i = 0; i < nren; i++)
      for (int j = 0; j < ncol; j++)
        datos[i][j] = new complejo(Re[i][j], Im[i][j]);
  }

  //
*********************************************************************

  MatrizC(int n, int m) {
    inicializa(n, m);
  }

  //
*********************************************************************

  private void inicializa(int r, int c) {
    // Si la matriz es nula reserva memoria.
    if ( (nren == 0 && ncol == 0) || this == null) {
      nren = r;
      ncol = c;

      datos = new complejo[nren][ncol];
      for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
```

```
            datos[i][j] = new complejo(0,0);
        }
    }

    //
**************************************************************************
*

    public void imprime()
    {
      int i, j;
      double aux;
      if ( (nren == 0) && (ncol == 0)) {
        System.out.println("No tiene informacion la matriz");
        return;
      }

      for (i = 0; i < this.nren; i++) {
        for (j = 0; j < this.ncol; j++)
          datos[i][j].imprime();
        System.out.println(" ");
      }
      System.out.println("\n\n ");
    }

    //
**************************************************************************
*

    public void suma(MatrizC a, MatrizC b) {
      if ( (a.nren != b.nren) || (a.ncol != b.ncol))
        return;

      this.inicializa(a.nren, a.ncol);

      if (this.nren == a.nren && this.ncol == a.ncol) {

        for (int i = 0; i < a.nren; i++)
          for (int j = 0; j < a.ncol; j++)
            this.datos[i][j].suma(a.datos[i][j], b.datos[i][j]);

      }
      else
        System.out.println("Matrices con diferente tamaño");
    }

    //
**************************************************************************
*

    public void resta(MatrizC a, MatrizC b) {
      if ( (a.nren != b.nren) || (a.ncol != b.ncol))
        return;

      this.inicializa(a.nren, a.ncol);

      if (this.nren == a.nren && this.ncol == a.ncol) {
```

```
      for (int i = 0; i < a.nren; i++)
        for (int j = 0; j < a.ncol; j++)
          this.datos[i][j].resta(a.datos[i][j], b.datos[i][j]);
    }
    else
      System.out.println("Matrices con diferente tamaño");
  }
  //
*************************************************************************
*
  public void multiplica(MatrizC a, MatrizC b) {
    int i, j;

    complejo sum = new complejo();
    complejo aux  = new complejo();

    if (a.ncol != b.nren)
      return;

    inicializa(a.nren, b.ncol);

    if (this.nren != a.nren || this.ncol != b.ncol)
      return;

    for (i = 0; i < this.nren; i++)
      for (j = 0; j < this.ncol; j++) {
        sum.real = 0;
        sum.imag = 0;
        for (int k = 0; k < a.ncol; k++)
        {
          aux.multiplica(a.datos[i][k], b.datos[k][j]);
          sum.suma(sum, aux);
        }
        this.datos[i][j].igual(sum);
      }
  }
}
```

Regresar.

Manejo de Excepciones.

Las excepciones son la manera que ofrece Java de manejar los errores en tiempo de ejecución. Muchos lenguajes imperativos, cuando tenían un error de este clase lo que hacían era detener la ejecución del programa. Las excepciones nos permiten escribir código que nos permita manejar ese error y continuar (si lo estimamos conveniente) con la ejecución del programa.

El error en ejecución más clásico en Java es el de desbordamiento. Es decir, el intento de acceso a una posición de un vector que no existe. Por ejemplo:

```
Desbordamiento.java

public class Desbordamiento {

  static String mensajes[] = {"Primero", "Segundo", "Tercero" };

  public static void main(String[] args) {

    for (int i=0; i<=3; i++)

      System.out.println(mensajes[i]);

    System.out.println("Ha finalizado la ejecución");

  }

}
```

Este programa tendrá un serio problema cuando intente acceder a mensajes[3], pues no existe dicho valor. Al ejecutarlo nos dirá lo siguiente (o algo parecido):

Primero

Segundo

Tercero

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

at Desbordamiento.main(Desbordamiento.java, Compiled Code)

Nos da un error de ejecución (en esta terminología se diría que se lanzó una excepción) al intentar acceder a dicho valor inexistente. Vemos que, por ahora, el comportamiento de nuestro código es el mismo que en los lenguajes imperativos. Cuando encuentra el error, se para la ejecución. Ahora veremos como evitar esto.

try...catch...finally

Existe una estructura que nos permitirá capturar excepciones, es decir, reaccionar a un error de ejecución. De este modo podremos imprimir mensajes de error "a la medida" y continuar con la ejecución del programa si consideramos que el error no es demasiado grave. Para ver como funcionan vamos a modificar el ejemplo anterior, pero asegurandonos ahora de que capturamos las excepciones:

NuestroPrimerCatch.java

```
public class NuestroPrimerCatch {

  static String mensajes[] = {"Primero", "Segundo", "Tercero" };

  public static void main(String[] args) {

    try {

      for (int i=0; i&lt;=3; i++)

        System.out.println(mensajes[i]);

    }

    catch ( ArrayIndexOutOfBoundsException e ) {

      System.out.println("El asunto se nos ha desbordado");

    }

    finally {

      System.out.println("Ha finalizado la ejecución");

    }

  }
```

```
}
```

Dentro del bloque que sigue a try colocamos el código a ejecutar. Es como si dijeramos que vamos a intentar ejecutar dicho código a ver qué pasa. Después de try deberemos colocar al menos un bloque catch o un bloque finally, pudiendo tener ambos e incluso más de un bloque catch.

En el bloque finally ponemos el código que se ejecutará siempre, tanto si se lanza la excepción como si no. Su utilidad no es mucha, ya que si se permite continuar con la ejecución del programa basta con poner el código después del bloque try...catch. En nuestro ejemplo podríamos haber puesto lo siguiente:

```
try {
  for (int i=0; i<=3; i++)
    System.out.println(mensajes[i]);
}
catch ( ArrayIndexOutOfBoundsException e ) {
  System.out.println("El asunto se nos ha desbordado");
}
System.out.println("Ha finalizado la ejecución");
```

Y habría funcionado exactamente igual. La miga está en los bloques catch.

Clase Exception

Cuando se lanza una excepción, en nuestro mundo orientado objetos lo que se hace es lanzar una instancia de Exception o de una clase derivada de él. Normalmente las clases derivadas no añaden mucha funcionalidad (muchas veces ninguna en absoluto), pero al ser distintas nos permiten distinguir entre los distintos tipos de excepciones.

En el programa anterior, por ejemplo, en el bloque catch capturábamos una excepción del tipo ArrayIndexOutOfBoundsException, ignorando cualquier otro tipo de excepción.

Esta clase tiene como cositas interesantes dos constructores y dos métodos (tiene más métodos pero sólo vamos a ver éstos):

Exception e = new Exception()

Crea una excepción si ningún mensaje específico.

Exception e = new Exception (String)

Crea una excepción con un mensaje que detalla el tipo de excepción.

String e.getMessage()

Devuelve el mensaje detallado si existe o null en caso contrario.

void e.printStackTrace()

Escribe por la salida de error estándar una traza que nos permitirá localizar donde se generó el error. Es muy útil para depurar, pero no es conveniente que lo usuarios finales vean estas cosas.

Capturar excepciones

Ahora sabemos lo suficiente como para entender cómo funcionan los catch. Entre paréntesis, cual parámetro se pone la declaración de una excepción, es decir, el nombre de una clase derivada de Exception (o la misma Exception) seguido por el nombre de la variable. Si se lanza una excepción que es la que deseamos capturar o una derivada de la misma se ejecutará el código que contiene el bloque. Así, por ejemplo:

```
catch (Exception e) {
  ...
}
```

Se ejecutará siempre que se produzca una excepción del tipo que sea, ya que todas las excepciones se derivan de Exception. No es recomendable utilizar algo así ya que estamos capturando cualquier tipo de excepciones sin saber si eso afectará a la ejecución del programa o no.

Se pueden colocar varios bloques catch. En ese caso, se comprobará en orden si la excepción lanzada es la que deseamos capturar y si no pasa a comprobar la siguiente. Eso sí, sólo se ejecuta un bloque catch. En cuanto captura la excepción deja de comprobar los demás bloques. Por eso, lo siguiente:

```
catch (Exception e) {

  ...

}

catch (DerivadaDeException e) {

 ...

}
```

Regresar.

Archivos.

La clase file (archivo) de la biblioteca de entrada salida de Java, proporciona una abstracción independiente de la plataforma para obtener información sobre un archivo, tales como, su ruta, carácter de separación, tamaño y fecha. Para ello debe crear un objeto File con alguno de los siguientes constructores.

```
File arch = new File(String ruta);
File arch = new File(String ruta, String nombre,);
File arch = new File(String dir, String nombre,);
```

Para saber si un archivo existe podemos utilizar el siguiente código

```
File f = new File(“temp.txt”);

if(f.exists())
  System.out.println(“El archivo temp.txt, si existe”);
else
  System.out.println(“El archivo temp.txt, no existe”);
```

Muchos programas necesitan extraer información a partir de un objeto. La clase RandomAccessFile (Archivo de acceso aleatorio) proporciona acceso a un archivo de manera secuencia. Para hacer la declaración procedemos como

```
String archivo = “algo.txt”;
RandomAccessFile DIS = new RandomAccessFile(archivo, "r");
```

La variable archivo representa el nombre con el cual se almacena una archivo de texto y la “r” indica que se trata de solo lectura.

```
String linea;
try
{
  RandomAccessFile DIS = new RandomAccessFile(archivo, "r");

  while((linea=DIS.readLine())!=null)
  {
     System.out.println(linea);
  }
  DIS.close();
}
catch(IOException e)
{
   System.out.println(“no pude abrir el archivo”);
}
```

Para escribir en un archivo hacemos

```
try
{
  RandomAccessFile DIS = new RandomAccessFile("salida.txt", "rw");
```

```
      DIS.writeBytes("hola mundo");
      DIS.close();
    }

    catch(IOException e)
    {
      System.out.println("no pude abrir el archivo");
    }
```

Separación en fichas (tokens)

Al leer una línea, hay veces que es necesario partir la cadena en piezas especificas (conocidas como tokens). Para esto utilizaremos la clase StringTokenizer. Los constructores para esta case son:

```
StringTokenizer(String cadena, String delim, bolean devoverTokens);
StringTokenizer(String cadena, String delim);
StringTokenizer(String cadena);
```

Con estos constructores podemos hacer la separación de los elementos de una cadena de la siguiente manera.

```
StringTokenizer st = new StringTokenizer(“1001 Tips de Java”, “ “);

while(st.hasMoreTokens())
{
  System.out.println(“token:” + st.nextToken());
}
```

Nota: Es importante hacer el uso de las clases de entrada salida y de utilerías de java, las cuales invocamos haciendo :

import java.io.*;
import java.util.*;

al principio del programa.

Ejemplo.

Uniendo las piezas podemos construir un constructor de la clase matriz que nos permita leer de un archivo los valores de una matriz .

```
  public Matriz(String archivo) {
    String linea = "", dato = "";
    int i = 0, j = 0;
    StringTokenizer st;
    double x;
```

```
    Dimensiones(archivo) ;
    datos = new double[nren][ncol];

    try {

      RandomAccessFile DIS = new RandomAccessFile(archivo, "r");

      i = 0;
      while ( ( (linea = DIS.readLine()) != null) && i < nren) {
        st = new StringTokenizer(linea);
        if(st.countTokens() != 0)
        {
          st = new StringTokenizer(linea);
          for (j = 0; j < ncol; j++) {
            dato = st.nextToken();
            x = Double.valueOf(dato).doubleValue();
            this.datos[i][j] = x;
          }
          i++;
        }
      }
      DIS.close();
    }
    catch (IOException e) {
      JOptionPane.showMessageDialog(null,    "Error"    +    e.toString(),
"ERROR",
                                    JOptionPane.ERROR_MESSAGE);
    }
  }
```

Para leer las dimensiones de la matriz hacemos la función:

```
  public void Dimensiones(String archivo) {
    StringTokenizer st;
    String linea;
    int num_ren = 0, num_col = 0;

    try {
      RandomAccessFile DIS = new RandomAccessFile(archivo, "r");

      while ( (linea = DIS.readLine()) != null) {
        st = new StringTokenizer(linea);
        if(st.countTokens() != 0) num_ren ++;

        if(num_col == 0)
        {
          while (st.hasMoreTokens())
          {
            num_col++;
            st.nextToken();
          }
        }
      }
      this.nren = num_ren;
      this.ncol = num_col;
      DIS.close();
```

```
    }
    catch (IOException e) {
      JOptionPane.showMessageDialog(null,    "Error"    +    e.toString(),
"ERROR",
                                    JOptionPane.ERROR_MESSAGE);
    }
  }
```

Regresar.

Solución de Circuitos.

Una posibilidad de solución de un circuito eléctrico, es la solución utilizando análisis de nodos la cual, consiste en solucionar un sistema de ecuaciones dada por

G v = I

donde:

G es la matriz de conductancias
v el vector de voltajes nodales e
I el vector de corriente en cada nodo.

La descripción del circuito la haremos en un archivo de datos y consideraremos que un elemento esta conectado a un par de nodos. Un nodo será de salida y otro de entrada tal como se muestra en la siguiente tabla:

Elem. No. k	Nodo salida i	Nodo entrada j	Conductancia mho g	Fuente corriente F
1	0	1	1	1
2	1	2	2	0
3	2	0	3	0
4	2	3	2	0
5	3	0	4	-5
6	1	3	0	3

nota: la información en amarillo no es parte del archivo de datos.

De tabla anterior podemos ver que el circuito esta formado por cuatros nodos, numerados del 0 al 3 y que este tiene 6 elementos. Por lo tanto, solucionaremos un sistema de tres ecuaciones con tres incógnitas.

Para una conductancia g conectada entre el nodo i y el j con una fuente de Corriente F, tomadas de la tabla, llenaremos la matriz de conductancias G y el vector de corrientes utilizando las siguientes reglas.

1.- Si tanto los nodo *i* y *j* son diferentes de cero, hacemos:

$$G_{i,j} = G_{i,j} - g$$
$$G_{j,i} = G_{j,i} - g$$

2.- Si el j-esimo nodo es diferente de cero, hacemos:

$$G_{i,i} = G_{i,i} + g$$
$$I_i = I_i - F$$

3.- Si el j-esimo nodo diferente de cero, hacemos:

$$G_{j,j} = G_{j,j} + g$$
$$I_j = I_j + F$$

Aplicando estos paso tenemos:

Para el elemento 1 tendremos:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Para el elemento 2:

$$\begin{bmatrix} 3 & -2 & 0 \\ -2 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Para el elemento 3

$$\begin{bmatrix} 3 & -2 & 0 \\ -2 & 5 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Para 4

$$\begin{bmatrix} 3 & -2 & 0 \\ -2 & 7 & -2 \\ 0 & -2 & 2 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 0 \\ 8 \end{bmatrix}$$

Para 5

$$\begin{bmatrix} 3 & -2 & 0 \\ -2 & 7 & -2 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

0	-2	6	V_3		5

Finalmente para el elemento 6

3	-2	0	V_1		1
-2	7	-2	V_2	=	0
0	-2	6	V_3		8

La solución de este sistema es :

V_1		-0.4889
V_2	=	0.2667
V_3		1.4222

La implementación en Java queda como:

```
/**
 * Programa para resolver un circuito utilizando analisis nodal
 * Este programa lee la descripción de un circuito y lo soluciona
 * Copyright: Copyright (c) 2003</p>
 * UMSNH
 * Dr. Felix Calderon Solorio
 * @version 1.0
 */

public class ej027 {
  public static void main(String[] args) {
    Matriz desc = new Matriz("cto.txt");
    int i, j, k, nodos = 0;

    for(k=0; k<desc.nren; k++)
    {
      if (nodos < desc.datos[k][0]) nodos = (int) desc.datos[k][0];
      if (nodos < desc.datos[k][1]) nodos = (int) desc.datos[k][1];
    }

    Matriz G = new Matriz(nodos, nodos);
    Matriz v = new Matriz(nodos, 1);
    Matriz I = new Matriz(nodos, 1);

    for(k=0; k<desc.nren; k++)
    {
      i = (int) desc.datos[k][0];
      j = (int) desc.datos[k][1];

      if(i!=0 && j != 0)
      {
        G.datos[i - 1][j - 1] -= desc.datos[k][2];
        G.datos[j - 1][i - 1] -= desc.datos[k][2];
      }

      if(i!=0)
```

```
      {
        G.datos[i-1][i-1] += desc.datos[k][2];
        I.datos[i-1][0]   -= desc.datos[k][3];
      }

      if(j!=0)
      {
        G.datos[j-1][j-1] += desc.datos[k][2];
        I.datos[j-1][0]   += desc.datos[k][3];
      }
    }
    G.imprime();
    I.imprime();
    v.eliminacion_gaussiana(G, I);
    v.sustitucion_hacia_atras(G,I);
    v.imprime();
  }
}
```

Formato Estándar de Pspice.

Un formato estándar muy utilizado es el del programa de Pspice. Para un circuito simple, formado por resistencias, fuentes de corriente y fuentes de voltaje, la descripción se hace de la siguiente manera:

Tipo de Elemento	Nodo salida	Nodo entrada	Valor
V1	0	1	20
I2	1	2	2
R23	2	0	3.5

Para describir un elemento hacemos:

1.- Dar un nombre que comenzará con V, I o R si se trata de una fuente de voltaje, corriente o resistencia respectivamente.
2.- Dar la información del par de nodos a los cuales esta conectado. El nodo de salida indicará el nodo negativo del elemento y el nodo de entrada el nodo positivo del elemento.
3.- Dar el valor de la fuente en unidades del Sistema internacional, por ejemplo, Volts, Amperes y ohms de acuerdo al tipo de elemento.

```
import java.io.*;
import java.util.*;
import javax.swing.*;

/**
 * Programa para leer Un archivo en formato Pspice
 * Lee los datos de un archivo y soluciona el circuito.
 * Copyright: Copyright (c) 2003
 * Company: UMSNH
```

```
 * @author Dr. Felix Calderon Solorio.
 * @version 1.0
 */

public class ej028 {
  public static void main(String[] args)
  {

    int nodos;
    String archivo = "resnick.txt";
    nodos = NumNodos(archivo);
    Matriz G = new Matriz(nodos, nodos);
    Matriz I = new Matriz(nodos, 1);
    Matriz v = new Matriz(nodos, 1);

    LlenaSistema(G, I, archivo);
    G.imprime();
    I.imprime();
    v.eliminacion_gaussiana(G, I);
    G.imprime();
    v.sustitucion_hacia_atras(G, I);
    v.imprime();
  }

  public static int NumNodos(String archivo) {
    StringTokenizer st;
    String linea, elemento, inicio, fin;
    int nodos = 0, val;

    try {
      RandomAccessFile DIS = new RandomAccessFile(archivo, "r");

      while ( (linea = DIS.readLine()) != null) {
        st = new StringTokenizer(linea);
        if (st.countTokens() != 0) {

          elemento = st.nextToken();
          inicio   = st.nextToken();
          fin      = st.nextToken();

          val = (int) Double.valueOf(inicio).doubleValue();
          if(val > nodos) nodos = val;

          val = (int) Double.valueOf(fin).doubleValue();
          if(val > nodos) nodos = val;
        }

      }
      DIS.close();
    }
    catch (IOException e) {
      JOptionPane.showMessageDialog(null, "Error" + e.toString(),
"ERROR",
                                    JOptionPane.ERROR_MESSAGE);
    }
    return nodos;
  }
```

```
  //
**************************************************************************

  public static void LlenaSistema(Matriz G, Matriz I, String archivo) {
    StringTokenizer st;
    String linea, elemento, inicio, fin, valor;
    char tipo ;
    int i, j;
    double val;

    try {
      RandomAccessFile DIS = new RandomAccessFile(archivo, "r");

      while ( (linea = DIS.readLine()) != null) {
        st = new StringTokenizer(linea);
        if (st.countTokens() != 0) {

          elemento = st.nextToken();
          inicio   = st.nextToken();
          fin      = st.nextToken();
          valor    = st.nextToken();

          tipo = elemento.charAt(0);
          i   = (int) Double.valueOf(inicio).doubleValue();
          j   = (int) Double.valueOf(fin).doubleValue();
          val =       Double.valueOf(valor).doubleValue();

          switch(tipo)
          {
            case 'R': case 'r':
              if(i!=0 && j != 0)
              {
                G.datos[i - 1][j - 1] -= 1.0/val;
                G.datos[j - 1][i - 1] -= 1.0/val;
              }
              if(i!=0) G.datos[i-1][i-1] += 1.0/val;
              if(j!=0) G.datos[j-1][j-1] += 1.0/val;
             break;

             case 'I': case 'i':
               if(i!=0) I.datos[i-1][0] -= val;
               if(j!=0) I.datos[j-1][0] += val;
               break;

             case 'V': case 'v':
               val = val / 1e-06;
               if(i!=0) I.datos[i-1][0] -= val;
               if(j!=0) I.datos[j-1][0] += val;

               val = 1e+06;

               if(i!=0 && j != 0)
               {
                 G.datos[i - 1][j - 1] -= val;
                 G.datos[j - 1][i - 1] -= val;
               }
```

```
                if(i!=0) G.datos[i-1][i-1] += val;
                if(j!=0) G.datos[j-1][j-1] += val;
            }
          }

        }
        DIS.close();
      }
      catch (IOException e) {
        JOptionPane.showMessageDialog(null, "Error" + e.toString(),
"ERROR",
                                      JOptionPane.ERROR_MESSAGE);
      }
    }
}
```

regresar.

¿Qué es un applet?

Un **applet** es una pequeña aplicación escrita en Java, pero que no está diseñada para funcionar "por sí sola", como los ejemplos que habíamos visto en modo texto, sino para formar parte de una página **Web**, de modo que necesitaremos un visualizador de páginas Web (como Internet Explorer de Microsoft, o Netscape Navigator -parte de Netscape Communicator-) para poder hacer funcionar dichos applets. Si no poseyeramos alguna utilidad de este tipo, en principio tampoco será un problema grave, porque el entorno de desarrollo de Java incluye una utilidad llamada "**appletviewer**", que nos permitirá probar nuestros applets.

¿Cómo se crea un applet?

Tendremos que dar **tres pasos**:

- El primer paso, similar a lo que ya hemos hecho, consiste en **teclear** el programa empleando un editor de texto (como Edit, que ya vimos) o un entorno integrado (que veremos más adelante).
- El segundo paso es nuevo: tendremos que crear la **página Web** desde la que se accederá al applet (o modificarla, si se trata de una página que ya existía pero que queremos mejorar).
- El último paso será, al igual que antes, **probar** el programa, pero esta vez no emplearemos la utilidad llamada "java", sino cualquier navegador Web o la utilidad "AppletViewer".

Vamos a verlo con un ejemplo. Crearemos un applet que escriba el texto "Hola Mundo!" en pantalla, al igual que hicimos en nuestra primera aplicación de ejemplo. Recordemos que lo que habíamos escrito en aquel momento fue:

```
//
// Aplicación HolaMundo de ejemplo
//

class HolaMundo {
    public static void main( String args[] ) {
        System.out.println( "Hola Mundo!" );
    }
}
```

Esta vez tendremos que escribir más. Primero vamos a ver cómo debería ser el applet, y después comentaremos los cambios (bastantes) con relación a este programa:

```
import java.awt.Graphics;

public class AppletHola extends java.applet.Applet {
```

```
public void paint(Graphics g) {
      g.drawString( "Hola Mundo!", 100, 50);
    }
}
```

Los cambios son los siguientes:

- Al principio aparece una orden "import", porque vamos a emplear una serie de posibilidades que no son parte del lenguaje Java "estándar", sino del paquete "Graphics" (funciones gráficas) que a su vez forma parte del grupo llamado AWT ("Abstract Windowing Toolkit", herramientas "abstractas" para el manejo de ventanas).
- Después no creamos una clase nueva "desde cero", sino a partir de una que ya existe, la clase "Applet". Por eso aparece la palabra "extends" en la definición de la clase, como ya vimos en el apartado sobre las clases y Java.
- Sólo utilizamos un método de los que nos permite la clase Applet: el método "paint", que es el que dibuja (o escribe) en pantalla la información que nos interese. Tiene un parámetro de tipo "Graphics", que es la pantalla en la que dibujaremos.
- En concreto, lo que hacemos en esta pantalla es dibujar una cadena de texto (drawString), en las coordenadas 100 (horizontal), 50 (vertical).

(Un poco más adelante se verá el resultado de este Applet).

Y vamos a ver los tres pasos que tenemos que dar, con mayor detalle.

Primer paso: teclear el fuente.

Repetimos básicamente lo que ya vimos:

- Tecleamos el fuente con "Edit", el bloc de notas de Windows, o cualquier otro editor de textos.
- Lo guardamos, con el mismo nombre que tiene la clase. En nuestro caso sería "AppletHola.java" (si empleamos otro nombre, el compilador "javac" nos regañará).
- Lo compilamos empleando "javac".
- Si hay algún error, se nos informará de cual es; si no aparece ningún error, obtendremos un fichero llamado "AppletHola.class", listo para utilizar.

Segundo paso: crear la página Web.

No pretendo "formar maestros" en el arte de crear páginas Web. Apenas veremos lo necesario para poder probar nuestros Applets.

Las páginas Web normalmente son ficheros escritos en un cierto lenguaje. El más sencillo, y totalmente válido para nuestros propósitos, es el lenguaje HTML. Los ficheros HTML son ficheros de texto, que podremos crear con cualquier editor de texto (más trabajoso, pero es como más control se obtiene, de modo que eso es lo que haremos por ahora) o con editores específicos (como Netscape Composer, Microsoft Frontpage, Hot Dog, etc, que permiten "crear" más rápido, aunque a veces eso supone perder en posibilidades).

En nuestro caso, recurriremos nuevamente a "Edit" (o al bloc de notas de Windows, o cualquier otro editor de textos) y escribiremos lo siguiente:

```
<html>
  <head>
      <title>Prueba de applet</title>
  </head>
  <body>
      <h1>Prueba de applet</h1>
      <hr>
      <applet code=AppletHola.class width=300 height=120>
        alt="El navegador esta no esta mostrando el APPLET"
      </applet>
  </body>
</html>
```

Grabaremos este texto con el nombre "prueba.html" (serviría cualquier otro nombre, con la única condición de que termine en ".htm" o en ".html") para utilizarlo un poco más adelante.

Vamos a ver qué hemos escrito:

- Es un fichero de texto, en el que aparecen algunas "etiquetas" encerradas entre < y >. El propio fichero en sí comienza por la etiqueta <html> y termina con </html> (esto será habitual: la mayoría de las etiquetas van "por pares", la primera crea un "bloque" y la segunda -la que contiene una barra al principio- indica el final del bloque).
- Después aparece otro bloque, opcional: la cabecera del fichero, que empieza con <head> y termina con </head>.
- Dentro de la cabecera hemos incluido una información que también es opcional: el "título" de la página, delimitado entre <title> y </title>. Este dato no es necesario, pero puede resultar cómodo, porque aparecerá en la barra de título del menú (ver las imágenes de ejemplo, un poco más abajo).
- Después va el cuerpo del fichero html, que se delimita entre <body> y </body>. (El cuerpo sí es obligatorio).
- Dentro del cuerpo, lo primero que aparece es un título (en inglés, "heading"), también opcional, delimitado entre <h1> y </h1>. Tenemos distintos niveles de títulos. Por ejemplo, podríamos poner un título de menor importancia delimitándolo enrtre <h2> y </h2>, o más pequeño aún entre <h3> y </h3>, y así sucesivamente.
- La etiqueta <hr> muestra una línea horizontal (en inglés, "horizontal ruler"; claramente, también es opcional), y no necesita ninguna etiqueta de cierre.

- Finalmente, la parte en la que se indica el applet comienza con <applet> y termina con </applet>. También es opcional en un "caso general", pero imprescindible en nuestro caso. En la etiqueta de comienzo se pueden indicar una serie de opciones, que veremos después con más detalle. Nosotros hemos utilizado sólo las tres imprescindibles:
    - code sirve para indicar cual es el nombre del applet a utilizar (en nuestro
    - width y height indican la anchura y altura de un rectángulo, medida en puntos de pantalla (pixels). Este rectángulo será la "ventana de trabajo" de nuestro applet.

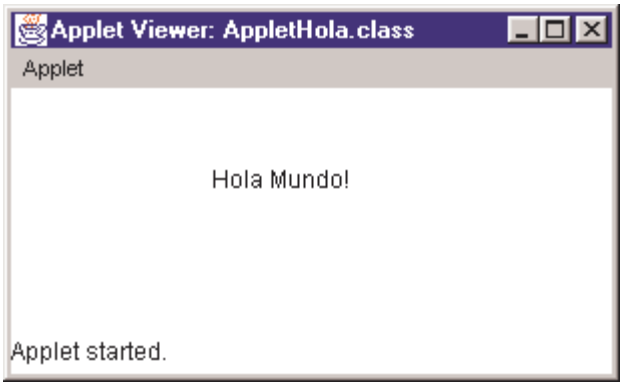
El resultado de todo esto se puede ver en el siguiente apartado...

Tercer paso: probar el resultado.

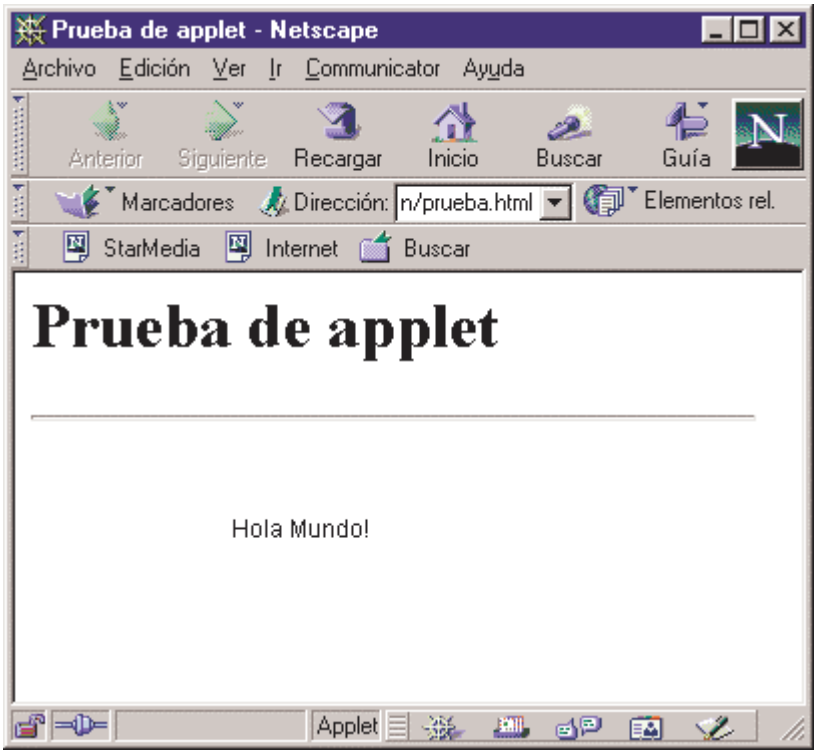
Si queremos usar la utilidad "AppletViewer" que incorpora el JDK, teclearemos:

```
appletviewer prueba.html
```

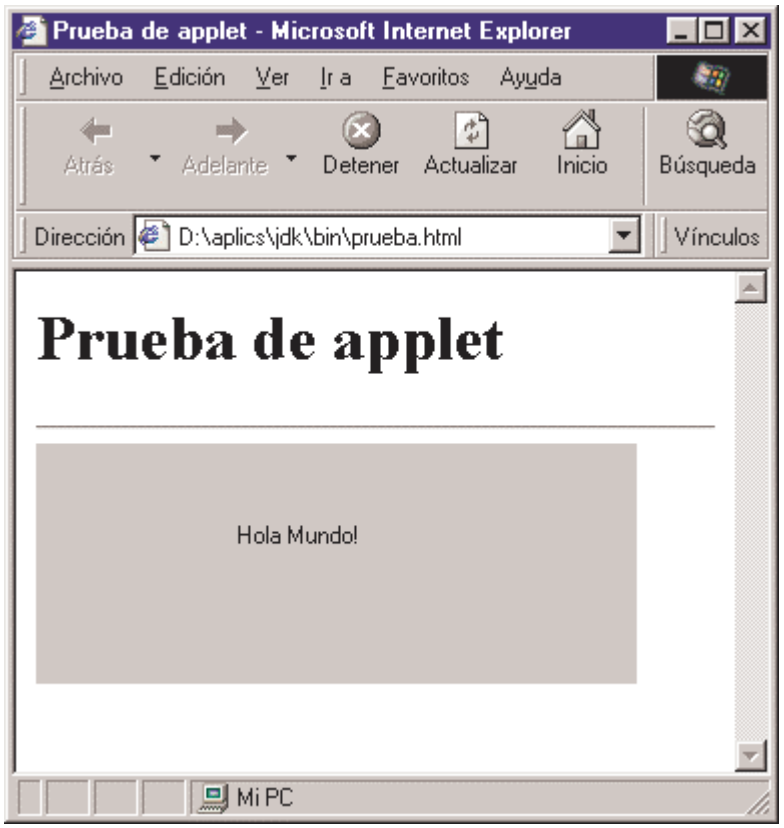
y al cabo de un momento, tendremos en pantalla algo parecido a:



También podemos usar cualquier otro navegador que soporte Java (cualquiera moderno debería permitirlo). Por ejemplo, Netscape Navigator 4.5 mostraría lo siguiente:



y el Internet Explorer 4 de Microsoft



La "vida" de un applet.

En los applets hay un detalle importante que hemos dejado pasar hasta ahora: no hemos utilizado una función "main" que represente el cuerpo de la aplicación. En su lugar, hemos empleado el método "paint" para indicar qué se debe mostrar en pantalla. Esto es porque un applet tiene un cierto "ciclo de vida":

- Cuando el applet se carga en el navegador, se llama a su método "**init**" (que nosotros no hemos "reutilizado"). Hace las funciones de un constructor.
- Cuando el applet se hace visible, se llama a su método "**start**" (que tampoco hemos empleado). Se volverá a llamar cada vez que se vuelva a hacer visible (por ejemplo, si ocultamos nuestro navegador y luego lo volvemos a mostrar).
- Cuando el applet se va a dibujar en pantalla, se llama a su método "**paint**" (como hemos hecho nosotros).
- El método "**repaint**" se utiliza para redibujar el applet en pantalla. Es normalmente el que nosotros utilizaremos como programadores, llamándolo cuando nos interese, en vez de "paint", que será el que se llame automáticamente. A su vez, "repaint" llama internamente a "**update**", que en ciertos casos nos puede interesar redefinir (por ejemplo, si no queremos que se borre toda la pantalla sino sólo una parte). Ambos los veremos con más detalle un poco más adelante.
- Cuando el applet se oculta (por ejemplo, si minizamos la ventana de nuestro navegador), se llama a su método "**stop**". Sería conveniente que desde él parásemos las animaciones o cualquier otra tarea que consumiese recursos "sin necesidad".
- Cuando se termina de utilizar el applet, se llama a su método "**destroy**".

De momento sólo hemos empleado "paint". Más adelante iremos sacando partido a otros de estos métodos (y a otros que nos permiten los applets).

Ejemplo.

Hacer un applet que permita elevar un número al cuadrado y al cubo. La entrada será desde un campo de texto y las operaciones se realizarán al momento de presionar un botón asociado con la operación a realizar.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;


public class ej016 extends Applet
{
  TextField campo1 = new TextField("0", 15);
  int bandera = 0;
  Button boton1 = new Button("Uno");
  Button boton2 = new Button("Dos");
```

```
  public void init()
  {

    add(boton1);
    add(boton2);
    add(new Label("resultado", Label.RIGHT));
    add(campo1);
  }

  public void paint(Graphics g)
  {
    switch(bandera)
    {
      case 1 : g.drawString("has precionado el boton 1", 20, 50); break;
      case 2 : g.drawString("has precionado el boton 2", 20, 50); break;
      default : g.drawString("No has precionado boton", 20, 50);
    }
  }

  public boolean action(Event evt, Object arg)
  {
    if(evt.target instanceof Button)
    {
      if(arg.equals("Uno"))
      {

campo1.setText(cuadrado(Double.valueOf(campo1.getText()).doubleValue()));
      }
      else if(arg.equals("Dos"))
      {

campo1.setText(cubo(Double.valueOf(campo1.getText()).doubleValue()));
      }
      return true;
    }
    return false;
  }

  String cuadrado(double n)
  {
    String a = "";

    a+= n*n;

    return a;
  }

  String cubo(double n)
  {
    String a = "";

    a+= n*n*n;

    return a;
  }
}
```

Ejemplo que describe una interfaz de entrada salida.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/**
 * <p>Title: Interfaz pspice</p>
 * <p>Description: Entrada salida de datos</p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: UMSN</p>
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

public class Applet1 extends Applet {
  private boolean isStandalone = false;
  TextArea textArea1 = new TextArea();
  TextArea textArea2 = new TextArea();
  Label label1 = new Label();
  Label label2 = new Label();
  TextField textField1 = new TextField();
  Label label3 = new Label();
  Button button1 = new Button();

  public void init() {
    textArea1.setText("");
    textArea1.setBounds(new Rectangle(41, 222, 196, 198));
    this.setLayout(null);
    textArea2.setBounds(new Rectangle(254, 223, 196, 198));
    textArea2.setText("");
    label1.setText("datos de entrada");
    label1.setBounds(new Rectangle(96, 203, 92, 15));
    label2.setText("datos de Salida");
    label2.setBounds(new Rectangle(311, 204, 82, 15));
    textField1.setText("");
    textField1.setBounds(new Rectangle(187, 101, 118, 20));
    label3.setText("Archivo de entrada");
    label3.setBounds(new Rectangle(198, 82, 112, 15));
    button1.setLabel("Ok");
    button1.setBounds(new Rectangle(211, 141, 71, 25));
    this.add(textArea1, null);
    this.add(textArea2, null);
    this.add(label1, null);
    this.add(label2, null);
    this.add(textField1, null);
    this.add(label3, null);
    this.add(button1, null);
  }

  public boolean action(Event evt, Object arg)
  {
    if(evt.target instanceof Button)
    {
      if (arg.equals("Ok")) {
```

```
        String aux = textField1.getText();
        textArea1.setText("Entrada " + aux);
        textArea2.setText("salida"+ aux);
       }
    }
    return false;
  }

}
```

Regresar.

Hilos

Un Hilo es un flujo de ejecución, con un comienzo y un fin, de una tarea. Todos los programas manejan un hilo y la ejecución depende de las condiciones en las que fue diseñado.

Cuando se ejecuta un applet, es deseable, ver como va el avance del proceso o como se desempeña un algoritmo en su ejecución. Dado que Java realiza la tarea de ejecución de este hilo, solo presentará algunos resultados. Normalmente en la ejecución, de un applet veremos el inicio y fin.

Una alternativa es crear nuestro propio hilo para realizar la ejecución de nuestro applet. La clase Thread, permite crear nuestros propios hilos y así controlar el flujo de ejecución.

La clase Thread, contiene los siguientes métodos para controlar la ejecución.

public void run()

> Este método es invocado a tiempo de ejecución. Uno puede sobre escribir este método y proveer el código que uno quiere que se ejecute al correr el hilo.

public void start()

> Arranca el hilo, lo cual provoca que el método run() sea invocado.

public void stop()

> Detiene la ejecución de un hilo.

public void resume()

> Reanuda la ejecución de un hilo.

public static void sleep(long t)

> Detiene temporalmente la ejecución de un hilo un tiempo dado por la variable t, en milisegundos.

public void interrupt()

> Interrumpe la ejecución de un hilo.

public static boolean isInterrupted()

> Prueba si el hilo en ejecución ha sido interrumpido.

public boolean isAlive()

Prueba si un hilo esta corriendo.

public void setPriority(int p)

Asigna prioridad p a un hilo.

Ejemplo 1.

El siguiente código muestra a un texto moviéndose a lo largo de la pantalla (ver ejecución).

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Applet2 extends Applet implements Runnable
{
  Thread hilo = null;
  int x, y;

  public void init()
  {
    x = 10;
    y = 100;
  }

  public void start()
  {
    if(hilo == null)
    {
      hilo = new Thread(this);
      hilo.start();
    }
  }

  public void paint(Graphics g)
  {
     //start();
     g.drawString("Hola Mundo", x, y);

  }

  public void run()
  {
    principal();
    hilo.interrupt();
    hilo = null;
  }

  // ************************************************************************

  void pausa(int m) {
    repaint();
```

```
    try {
      Thread.sleep(m);
    }
    catch (InterruptedException e) {}
    ;
  }

  // ************************************************************************

  void principal()
  {
     for(int i=0; i< 100; i++)
     {
       x = x + i;
       repaint();
       pausa(300);
     }
  }
}
```

Ejemplo 2.

El siguiente código de Java muestra la ejecución de un hilo. Este hilo fue diseñado para mostrar como se modifica las curva $y=2x-2$ y $z = m$, donde m es una variable. (ver ejecución)

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Applet1 extends Applet implements Runnable
{
  Thread hilo = null;
  grafica g;

  public void init()
  {
    g = null;
  }

  public void start()
  {
    if(hilo == null)
    {
      hilo = new Thread(this);
      hilo.start();
    }
  }

  public void paint(Graphics g)
  {
     start();
     g.drawString("dibujando", 100,100);
  }
```

```
  public void run()
  {
    principal();
    hilo.interrupt();
    hilo = null;
  }

  // ********************************************************************************

  void pausa(int m) {
    repaint();

    try {
      Thread.sleep(m);
    }
    catch (InterruptedException e) {}
    ;
  }

  // ********************************************************************************

  void principal()
  {
    double x[] = new double[10];
    double y[] = new double[10];
    double z[] = new double[10];
    double m = 0;

    for(int it=0; it<10; it++)
    {
      g = new grafica("salida");
      for(int i=0; i<10; i++)
      {
        x[i] = i;
        y[i] = 2*x[i] - 2;
        z[i] = m;
      }
      m += 1;
      g.SerieX(x);
      g.SerieY(y);
      g.SerieX(z);
      g.SerieY(x);
      g.show();
      pausa(300);
      g.dispose();
    }
  }
}
```

Regresar.